時相論理とモデル検査を用いた
Ｃコンパイラ最適化器

東京工業大学
情報理工学研究科
数理・計算科学専攻

藤原 一貴
(09M37258)

平成22年度修士論文

指導教官 佐々 政孝 教授

２０１１年1月２７日

目 次

第 I 部　概論 5

第 1 章　はじめに 6

1.1　背景 6
1.2　概要 6
1.3　論文の構成 7

第 II 部　予備知識 8

第 2 章　プログラムの最適化 9

2.1　最適化と正しさ 9
2.2　制御フローグラフ 9
2.3　データフロー解析 11
2.3.1　利用可能な式の解析 12
2.3.2　生存変数解析 12
2.4　最適化の種類 13
2.4.1　無用命令除去 13
2.4.2　コピー伝播 (定数伝播を含む) 14

第 3 章　時相論理とモデル検査 15

3.1　状態遷移システム 15
3.2　時相論理 16
3.2.1　線形時間時相論理 ー LTL 16
3.2.2　計算木論理 ー CTL 18
3.2.3　CTL* 21
3.3　モデル検査 23
3.3.1　モデル検査の計算量 23
3.3.2　μ 計算への変換 23

第 III 部　提案手法 24

第 4 章　提案手法で採用する時相論理 25

4.1　プログラムのモデル化 25
4.1.1　制御フローモデル 25
4.1.2　原始命題の種類 25

4.2 双方向経路を扱う CTL 26
4.3 自由変数の導入 28

第 5 章 モデル検査 29
5.1 モデル検査 29
5.1.1 アルゴリズムの概要 29
5.1.2 各演算子ごとのアルゴリズム 29

第 6 章 提案手法 38
6.1 最適化システムの説明 38
6.2 ＜ step 1 ＞ コードのモデル化 39
6.3 ＜ step 2 ＞最適化検査仕様作成 40
6.3.1 最適化検査仕様 40
6.3.2 無用命令除去の最適化検査仕様 42
6.3.3 コピー伝播の最適化検査仕様 44
6.4 自由変数の束縛 45
6.4.1 無用命令除去での自由変数の束縛 45
6.4.2 コピー伝播（定数伝播を含む）での自由変数の束縛 46
6.5 ＜ step 3 ＞モデル検査 & 書き換え作業 47
6.5.1 無用命令除去でのモデル検査 & 書き換え作業 47
6.5.2 コピー伝播（定数伝播を含む）でのモデル検査 & 書き換え作業 . . 49

第 IV 部 実装 51

第 7 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 52
7.1 概要 52
7.2 構成 52
7.3 COINS バックエンドの構造 52
7.4 LIR におけるモデルの定義 53
7.4.1 モデルの状態 57
7.4.2 状態間の遷移関係 57
7.4.3 原始命題の意味 57

第 8 章 実験と考察 59
8.1 実験 59
8.1.1 実験環境 59
8.1.2 実行時間 60
8.1.3 最適化時間 61
8.2 考察 62
8.2.1 非効率なモデル検査器 63
8.2.2 無駄な CTL 式 63

第V部　結論 **64**

第9章　関連研究 **65**

9.1　Laceyらの研究 65

9.2　方の研究 65

9.3　Warburtonの研究 65

第10章 まとめ **66**

第11章 今後の課題 **67**

図 目 次

2.1 プログラムの制御フローグラフの例 11
2.2 無用命令除去 13
2.3 コピー伝播 14

3.1 LTL 式の例 18
3.2 CTL 式の例 20

5.1 CTL 式の構文木の例 29

6.1 提案システムの概要 38
6.2 中間言語のモデル化 39
6.3 中間言語のモデル化 39
6.4 無用命令除去の CTL-FV 式の直感的意味 42
6.5 コピー伝播の CTL-FV 式の直感的意味 44

7.1 COINS の構成図 53
7.2 Module の構成 54
7.3 FlowGraph の構成 55
7.4 instrList の構成 56

8.1 無用命令除去の実行時間 60
8.2 コピー伝播と無用命令除去の実行時間 61

第Ⅰ部

概論

第1章　はじめに

1.1　背景

コンパイラのフェーズにおいて最適化は重要なフェーズの一つであり，近年研究が盛んに行われている。

最適化の役割は，目的コードの時間的・空間的効率を向上させることである．このため，最適化器は中間言語上で様々な情報を集め，中間言語をより効率のよいものに変換したり，集めた情報をコード生成器に渡して，そこで効率の良いコードを生成させる．

伝統的なコンパイラの最適化は，コントロールフロー解析やデータフロー解析といったプログラム解析の結果を用いて，適切なプログラム変換をするものである．こういった，近年考えられている最適化のアルゴリズムは複雑なものが多く，それを実装した最適化器は，ソースコードが複雑でその量も長くなりバグの混入を防ぐのは困難である．

そういった流れから，コンパイラの新たな最適化器の研究と同時に，最適化器そのものの正当性に関する研究というものがなされるようになった．

この最適化器の正当性に関する研究の一つに，時相論理という論理体系を用いて最適化の正しさを証明するというものがある．これは，まずLaceyらの研究 [10][11] によって，典型的な種類の最適化に対して時相論理を用いた手書きの証明がなされ，その後，方 [22] やWarburton[21] らによって，時相論理の証明を元にモデル検査を用いた最適化手法が確立され，JAVA 最適化器として実装されてきた．

1.2　概要

本研究では，方 [22] や Warburton[21] らと同様の手法で，時相論理とモデル検査を用いた C コンパイラ最適化器の実装を行った．

既存研究と本研究の大きな違いは，既存研究が JAVA バイトコードの解析系である soot[20] を用いて JAVA 最適化器を実装しているのに対して，本研究では，コンパイラ・インフラストラクチャである COINS[5] 上で，C コンパイラ最適化器を実装したことである．

この違いにより本研究がもつ特徴は以下である．

- 既存研究のほとんどが JAVA 最適化器を実装しているのに対して，本研究では C 最適化器を実装した．これは，Lacey らの研究 [10][11] によって証明された最適化器の種類は，JAVA よりも C 言語の方がより効果が表れると思われるためである．

- 既存研究では，システムの実装において扱う中間言語が高級言語に近い言語であるのに対して，本研究では，より機械語に近い中間言語である LIR というものを対象としたシステムを実装した．

また，実装した以下の最適化の効果を評価するために，ベンチマークとしてSPEC2000を使用した.

- 無用命令除去
- コピー伝播（定数伝播を含む）

1.3 論文の構成

本論文の構成を以下に示す.

第II部 予備知識

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

7章 ： COINSの説明
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

予備知識

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, 2, 13, 16]．制御フローグラフは，**基本ブロック**をノードとし[1]，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．

[1]一つの文をノードとすることもある．

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，$B_1 \to B_2$ 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 $n_1 \to n_2$ 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 $n_0, n_1, n_2, ..., n_m (m \geq 0)$ について，$\forall i, 0 \leq i < m;\ n_i \to n_{i+1}$ 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．

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，$n_1\ dom\ n_2$ と表す．この定義によると，$n\ dom\ n$，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，$n_1\ dom\ n_2$ 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 $n_2 \to n_1$ は，$n_1\ dom\ n_2$ が成り立つとき**帰辺**であるという．図 *2.1(b)* の例では，B7→B5 が帰辺となる．

定義 2.8 (自然ループ) フローグラフの辺 $b \to h$ が帰辺であるとする．この帰辺 $b \to h$ 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）

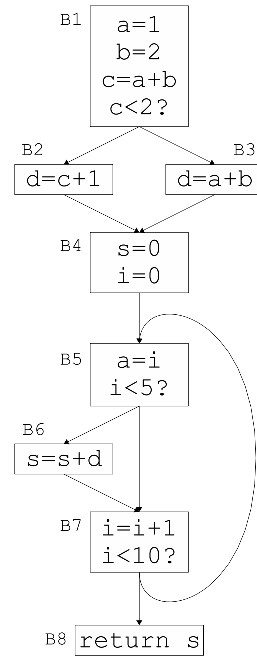
[2] 当然，一つの文をノードとするフローグラフも考えられるわけだが，ここでは図は記載しない．

```
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

を合わせたものである．つまり，$loop(h, b) = \{h\} \cup \{n \mid n$ から h を通らずに b に到達するパスがある $\}$ と表せる．

自然ループ $loop(h, b)$ 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 $n \in loop(h, b)$ が $n' \notin loop(h, b)$ 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，$n_2\ pdom\ n_1$ と表す．この定義によると，$n\ pdom\ n$，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，$n_2\ pdom\ n_1$ 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, 2, 13, 16]．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[3]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[4]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っ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 $\mathrm{a}+\mathrm{b}$ を再び計算しているが，これは冗長な計算であり，$\mathrm{a}+\mathrm{b}$ 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 $\mathrm{a}+\mathrm{b} \in AVAIL(\text{B3 の文})$ 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．

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**生きている**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[5]．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 $\mathrm{a} \notin LIVE(\text{B5 の文})$ 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．

[3]例えば $\mathrm{a}+\mathrm{b}$ などは式である．
[4]式 $\mathrm{a}+\mathrm{b}$ の場合，a も b も定義されていないということ．
[5]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．

2.4　最適化の種類

ここでは本研究で扱う最適化，**無用命令除去**，**コピー伝播 (定数伝播を含む)** の説明を行う．

2.4.1　無用命令除去

実行されるはずのない命令，すなわち，そこへ制御の流れが届かない命令は無用命令 (dead code) である．また，例えば

```
a = b + c;
```

という命令が制御の流れ上にあっても，その結果が得られる a の値がどこにも使われないのなら，この命令は無用命令として削除できる．

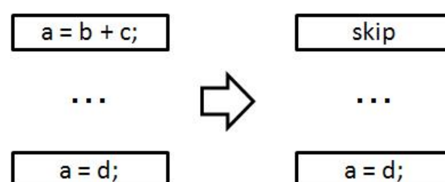


図 2.2: 無用命令除去

2.4.2 コピー伝播 (定数伝播を含む)

例えば

```
（1） a = b;
        …
（2） c = a + d;
```

において以下の条件が成り立つときに，(2) の a を b に変更することができる．

(A) (2) で使用される a の値を定義しているのは (1) だけである

(B) (1) を実行してから (2) を実行するまでの間に b の定義はない

(C) (1) で定義された a を使う全ての命令について上記の (A)，(B) が成り立つ

(1) のようにコピー伝播された命令は後で無用命令として削除できる可能性が高い．また，b，d が定数である場合は定数の畳み込みもできる．

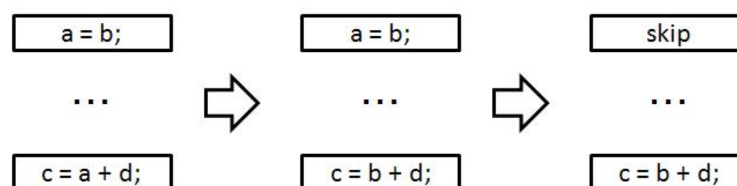


図 2.3: コピー伝播

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[1]．

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，

- 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，$s \to s'$ 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 $s_0, s_1, s_2, ...$ が，任意の $i \geq 0$ について $s_i \to s_{i+1}$ 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 $s_0, s_1, ..., s_m$ が，任意の $i\,(0 \leq i < m)$ について $s_i \to s_{i+1}$ かつ $\forall s \in S, \neg(s_m \to s)$ 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 $p = s_0, s_1, ..., s_i, ...$ 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 $s_i, s_{i+1}, ...$ も経路であり，p_i と表す．

[1]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．

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 $L\colon S \to 2^{AP}$ 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．

3.2.1 線形時間時相論理 ー LTL

線形時間時相論理（Linear-time Temporal Logic，LTL）[15] 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．

$$
\begin{array}{lcl}
LTL\text{-}formula & ::= & proposition \\
 & | & \neg\ LTL\text{-}formula \\
 & | & LTL\text{-}formula\ \wedge\ LTL\text{-}formula \\
 & | & X\ LTL\text{-}formula \\
 & | & G\ LTL\text{-}formula \\
 & | & LTL\text{-}formula\ U\ LTL\text{-}formula
\end{array}
$$

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．

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

LTL の意味論

Kripke 構造 $M = (S, R, L)$ の経路 $p = s_0 \rightarrow s_1 \rightarrow ...$ で LTL 式 ϕ 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 ϕ が成立する．
$p \models G\phi$	:	経路 p 上の全ての状態で ϕ が成立する．
$p \models \phi_1 U \phi_2$	:	経路 p 上で，ϕ_2 が成立するまで ϕ_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．

$p \models proposition$	iff	$proposition \in L(s_0)$
$p \models \neg\phi$	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．

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

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 ϕ が成立する．
$p \models \phi_1 W \phi_2$: 経路 p 上で，ϕ_2 が成立するまで ϕ_1 が成立し続ける．U と違い，ϕ_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．

(a) $p \models X\phi$ を満たす経路　(b) $p \models \phi_1 U \phi_2$ を満たす経路

図 3.1: LTL 式の例

3.2.2 計算木論理 ー CTL

計算木論理（Computational Tree Logic，CTL）[3] 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．

$$
\begin{array}{rcl}
CTL\text{-}formula & ::= & state\text{-}formula \\
state\text{-}formula & ::= & proposition \\
 & | & \neg\ state\text{-}formula \\
 & | & state\text{-}formula\ \wedge\ state\text{-}formula \\
 & | & E\ path\text{-}formula \\
 & | & A\ path\text{-}formula \\
path\text{-}formula & ::= & X\ state\text{-}formula \\
 & | & G\ state\text{-}formula \\
 & | & state\text{-}formula\ U\ state\text{-}formula \\
 & | & state\text{-}formula\ W\ state\text{-}formula
\end{array}
$$

表 3.4: CTL の構文規則

CTL の構文

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

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 ϕ 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 ψ が成り立つことを，$M, p \models \psi$ と表す[3]．

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．

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．

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．

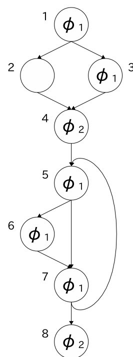
[3]LTL と同様，M を省略することもある．

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 $p = s \rightarrow s_1 \rightarrow ...$
$s \models A\psi$	iff	$p \models \psi$ for any path $p = s \rightarrow s_1 \rightarrow ...$
path formula	$(p = s_0 \rightarrow s_1 \rightarrow ...)$	
$p \models X\phi$	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

$$\begin{array}{lcl} \phi_1 \vee \phi_2 & \equiv & \neg(\neg\phi_1 \wedge \neg\phi_2) \\ \phi_1 \rightarrow \phi_2 & \equiv & \neg\phi_1 \vee \phi_2 \\ F\phi & \equiv & true\ U\ \phi \end{array}$$

表 3.6: CTL の演繹体系



$$\begin{array}{lcl} \{s \mid s \models EX\ \phi_1\} & = & \{1,4,5,6,7\} \\ \{s \mid s \models AX\ \phi_1\} & = & \{5,6\} \\ \{s \mid s \models E\,[\phi_1\ U\ \phi_2]\} & = & \{1,3,4,5,6,7,8\} \\ \{s \mid s \models A\,[\phi_1\ U\ \phi_2]\} & = & \{3,4,8\} \\ \{s \mid s \models A\,[\phi_1\ W\ \phi_2]\} & = & \{3,4,5,6,7,8\} \end{array}$$

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，$state\text{-}formula$は状態式，$path\text{-}formula$は経路式である．

$$
\begin{array}{lcl}
CTL*\text{-}formula & ::= & state\text{-}formula \\
state\text{-}formula & ::= & proposition \\
 & | & \neg\ state\text{-}formula \\
 & | & state\text{-}formula\ \wedge\ state\text{-}formula \\
 & | & E\ path\text{-}formula \\
 & | & A\ path\text{-}formula \\
path\text{-}formula & ::= & CTL*\text{-}formula \\
 & | & \neg\ path\text{-}formula \\
 & | & path\text{-}formula\ \wedge\ path\text{-}formula \\
 & | & X\ path\text{-}formula \\
 & | & G\ path\text{-}formula \\
 & | & path\text{-}formula\ U\ path\text{-}formula
\end{array}
$$

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．

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 $p = s \rightarrow s_1 \rightarrow ...$
$s \models A\psi$	iff	$p \models \psi$ for any path $p = s \rightarrow s_1 \rightarrow ...$
path formula	$(p = s_0 \rightarrow s_1 \rightarrow ...)$	
$p \models \phi$	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

$$
\begin{aligned}
\phi_1 \vee \phi_2 &\equiv \neg(\neg\phi_1 \wedge \neg\phi_2) \\
\phi_1 \rightarrow \phi_2 &\equiv \neg\phi_1 \vee \phi_2 \\
F\phi &\equiv true\ U\ \phi \\
\phi_1 W \phi_2 &\equiv (\phi_1 U \phi_2) \vee (G\phi_1)
\end{aligned}
$$

表 3.9: CTL*の演繹体系

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 ϕ 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 ϕ を満たすかどうか，つまり

$$M, s \models \phi$$

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，ϕ 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 [4]．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．

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

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 μ 計算への変換

μ **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，μ 計算に変換して記述することができる．μ 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 μ 計算により記述できる．

$$\{s \mid s \models A\,[\phi_1\ U\ \phi_2]\} = \mu.Z\ (\phi_2 \vee (\phi_1 \wedge \Box Z))$$

ここで，$\Box$ は，AX に相当する様相記号である．

文献 [19] によると，μ 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

本手法ではLaceyらと同様，CTL-FV[10]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$L\colon N \to 2^{AP}$とする．三つ組$M = (N, E, L)$を**制御フローモデル**という．これは$Kripke$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[10]や[22]と共通のものの多い．

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 $mark(\mathrm{M})$ 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．

$L(n)$	$=$	$\{ use(\mathrm{X})$	$\mid$	X は n で「使用される」変数 $\}$
	$\cup$	$\{ def(\mathrm{X})$	$\mid$	X は n で「定義される」変数 $\}$
	$\cup$	$\{ trans(\mathrm{E})$	$\mid$	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 $\}$
	$\cup$	$\{ stmt(\mathrm{M})$	$\mid$	M は n における命令 $\}$

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 7.4 節で述べる．

4.2 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 $\mathrm{CTL_{bd}}$

と呼ぶことにする．$\mathrm{CTL_{bd}}$ 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．$\mathrm{CTL_{bd}}$ 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．

$$
\begin{array}{lcl}
\mathrm{CTL_{bd}}\text{-}formula & ::= & state\text{-}formula \\
state\text{-}formula & ::= & proposition \\
 & | & \neg\ state\text{-}formula \\
 & | & state\text{-}formula\ \wedge\ state\text{-}formula \\
 & | & E\ path\text{-}formula \\
 & | & A\ path\text{-}formula \\
 & | & \overleftarrow{E}\ path\text{-}formula \\
 & | & \overleftarrow{A}\ path\text{-}formula \\
path\text{-}formula & ::= & X\ state\text{-}formula \\
 & | & G\ state\text{-}formula \\
 & | & state\text{-}formula\ U\ state\text{-}formula \\
 & | & state\text{-}formula\ W\ state\text{-}formula
\end{array}
$$

表 4.2: $\mathrm{CTL_{bd}}$ 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 $p = s \to s_1 \to ...$
$s \models A\psi$	iff	$p \models \psi$ for any path $p = s \to s_1 \to ...$
$s \models \overleftarrow{E}\psi$	iff	$p \models \psi$ for some path $p = s \to^{\circ} s_1 \to ...$
$s \models \overleftarrow{A}\psi$	iff	$p \models \psi$ for any path $p = s \to^{\circ} s_1 \to ...$
path formula	($p = s_0 \to' s_1 \to'$ ... in which $\to'$ is $\to$ or $\to^{\circ}$)	
$p \models X\phi$	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: $\mathrm{CTL_{bd}}$ の意味定義

4.3 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，$\mathrm{CTL_{bd}}$ を用いて

$$s \models EF\ use(\mathrm{x}) \tag{4.1}$$

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．

$$s \models EF\ use(x) \tag{4.2}$$

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**ということにする．

$\mathrm{CTL_{bd}}$ 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 [10]．本論文では，以後，自由変数を含む $\mathrm{CTL_{bd}}$ 式を **CTL-FV 式**，自由変数を全く含まない $\mathrm{CTL_{bd}}$ 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　モデル検査

5.1　モデル検査

CTL のモデル検査のアルゴリズムはいくつか存在する [4]．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．

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 [3] 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．

5.1.1　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．

例えば，$A\,[\neg use(v)\ U\ (def(v) \wedge use(v))]$ の CTL 式における部分式の関係を明示すると，図 5.1 のようになる．これは CTL の構文木を表す．

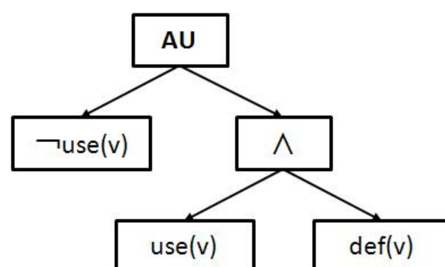


図 5.1: CTL 式の構文木の例

「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 5.1 では，$\wedge$ を頂点とする部分式について計算したければ，まず $use(v)$ と $def(v)$ の成り立つ状態の集合を先に求めておく必要がある．

5.1.2　各演算子ごとのアルゴリズム

以下，各演算子ごとに，CTL 式 ϕ 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 ϕ 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 ϕ 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

ϕ が原始命題のとき，ϕ 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，ϕ 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，ϕ_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，ϕ_1，ϕ_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 ϕ を満たすとは，定義より「$n \to n'$ 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，ϕ のラベル付けは，アルゴリズム 5.1 となる．ただし，$succ(n) = \{n' \mid n \to n'\}$ とする．

アルゴリズム 5.1 $EX\ \phi_1$ のラベル付け

labelEX()

```
1: for all n ∈ N do
2:   val ← false
3:   for all n' ∈ succ(n) do
4:     val ← val ∨ n' ⊨ φ
5:   if val = true then
6:     label n ⊨ φ
7:   else
8:     label n ⊭ φ
```

$\phi = \overleftarrow{E}X\ \phi_1$ についても経路を逆にするだけで，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 $succ(n)$ を $pred(n)$ とすればよい．ただし，$pred(n) = \{n' \mid n \to^{\circ} n'\}$ とする．

AX

$\phi = AX\ \phi_1$ のとき，状態 n が ϕ を満たすとは，定義より「$n \to n'$ 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，ϕ のラベル付けは，アルゴリズム 5.2 となる．

アルゴリズム 5.2 $AX\ \phi$ のラベル付け

labelAX()

```
1:  for all n ∈ N do
2:    if succ(n) is empty then
3:      label n ⊭ φ
4:    else
5:      val ← true
6:      for all n' ∈ succ(n) do
7:        val ← val ∧ n' ⊨ φ
8:      if val = true then
9:        label n ⊨ φ
10:     else
11:       label n ⊭ φ
```

$\phi = \overleftarrow{A}X\ \phi_1$ についても経路を逆にするだけで，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 $succ(n)$ を $pred(n)$ とすればよい．

EU

$\phi = E\,[\phi_1\ U\ \phi_2]$ のとき，状態 n が ϕ 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 ϕ_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 ϕ_1 を満たす．

つまり，次のようにラベル付けすればよい．

- ϕ_2 を満たす状態 n には，ラベル $n \models \phi$ を付ける．
- ϕ_1 を満たす状態 n が「n の次の状態 $n'(n \to n')$ 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. ϕ_2 を満たす状態 n に，ラベル $n \models \phi$ を付ける．
2. ϕ_2 を満たす状態 n に遷移可能で，かつ ϕ_1 を満たす全ての状態 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 に遷移可能で，かつ ϕ_1 を満たす全ての状態 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，ϕ_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．

アルゴリズム 5.3 $E\,[\phi_1\ U\ \phi_2]$ のラベル付け

labelEU()

```
1: for all n ∈ N do
2:    unlabel n
3: for all n ∈ N do
4:    if n is not labeled then
5:       if n ⊭ φ1 ∧ n ⊭ φ2 then
6:          label n ⊭ φ
7:       else if n ⊨ φ2 then
8:          label n ⊨ φ
9:          for all n′ ∈ pred(n) do
10:            subEU(n′)
11: for all n ∈ N do
12:    if n is not labeled then
13:       label n ⊭ φ
```

subEU(n**)**

```
14: if n is not labeled then
15:    if n ⊭ φ1 ∧ n ⊭ φ2 then
16:       label n ⊭ φ
17:    else
18:       label n ⊨ φ
19:       for all n′ ∈ pred(n) do
20:          subEU(n′)
```

AU

$\phi = A\,[\phi_1\ U\ \phi_2]$ のとき，状態 n が ϕ 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 ϕ_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 ϕ_1 を満たす．

つまり，次のようにラベル付けすればよい．

- ϕ_1 も ϕ_2 も満たさない状態 n には，ラベル $n \not\models \phi$ を付ける．
- ϕ_2 を満たす状態 n には，ラベル $n \models \phi$ を付ける．
- ϕ_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．

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[1]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，ϕ_1 が成立し続けるが ϕ_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，$A\,[\phi_1\ U\ \phi_2]$ 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．

$\phi = \overleftarrow{A}\,[\phi_1\ U\ \phi_2]$ 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 5.4 の 12，15 行目の $succ(n)$ を $pred(n)$ とすればよい．

[1]本手法における変形箇所に付けるマークとは当然別の意味である．

アルゴリズム 5.4 $A\,[\phi_1\ U\ \phi_2]$ のラベル付け

labelAU()

```
 1: for all n ∈ N do
 2:    unlabel n
 3:    unmark n
 4: for all n ∈ N do
 5:    subAU(n)
```

subAU(n)

```
 6: if n is not marked then
 7:    mark n
 8:    if n ⊭ φ1 ∧ n ⊭ φ2 then
 9:       label n ⊭ φ; return false
10:    else if n ⊨ φ2 then
11:       label n ⊨ φ; return true
12:    else if succ(n) is empty then
13:       label n ⊭ φ; return false
14:    else
15:       for all n′ ∈ succ(n) do
16:          if ¬subAU(n′) then
17:             label n ⊭ φ; return false
18:       label n ⊨ φ; return true
19: else
20:    if n ⊭ φ ∨ n is not labeled then
21:       return false
22:    else // n ⊨ φ
23:       return true
```

AW

$\phi = A\,[\phi_1\ W\ \phi_2]$ のとき，状態 n が ϕ 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 ϕ_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 ϕ_1 を満たす．
- 経路は無限経路で，経路の全ての状態が ϕ_1 を満たす．

つまり，次のようにラベル付けすればよい．

- ϕ_2 を満たす状態 n には，ラベル $n \models \phi$ を付ける．
- ϕ_2 を満たさない状態 n が，ϕ_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．
- ϕ_2 を満たさない状態 n が「n の次の状態 $n'(n \to n')$ 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 5.5 の 8 行目の $succ(n)$ を $pred(n)$ に，11，22 行目の $pred(n)$ を $succ(n)$ 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 ∈ N do
2:    unlabel n
3: for all n ∈ N do
4:    if n is not marked then
5:       if n ⊨ φ2 then
6:          label n ⊨ φ
7:          mark n
8:       else if n ⊭ φ1 ∨ succ(n) is empty then
9:          label n ⊭ φ
10:         mark n
11:         for all n' ∈ pred(n) do
12:            subAW(n')
13: for all n ∈ N do
14:    if n is not marked then
15:       label n ⊨ φ
```

subAW(n)

```
16: if not marked n then
17:    mark n
18:    if n ⊨ φ2 then
19:       label n ⊨ φ
20:    else
21:       label n ⊭ φ
22:       for all n' ∈ pred(n) do
23:          subAW(n')
```

第6章　提案手法

6.1　最適化システムの説明

本研究で実装した最適化システムの概要図を図 6.1 に示す．本システムは最適化オプションと LIR を入力とし，様々な処理を行って最適化を行った新たな LIR を出力する．

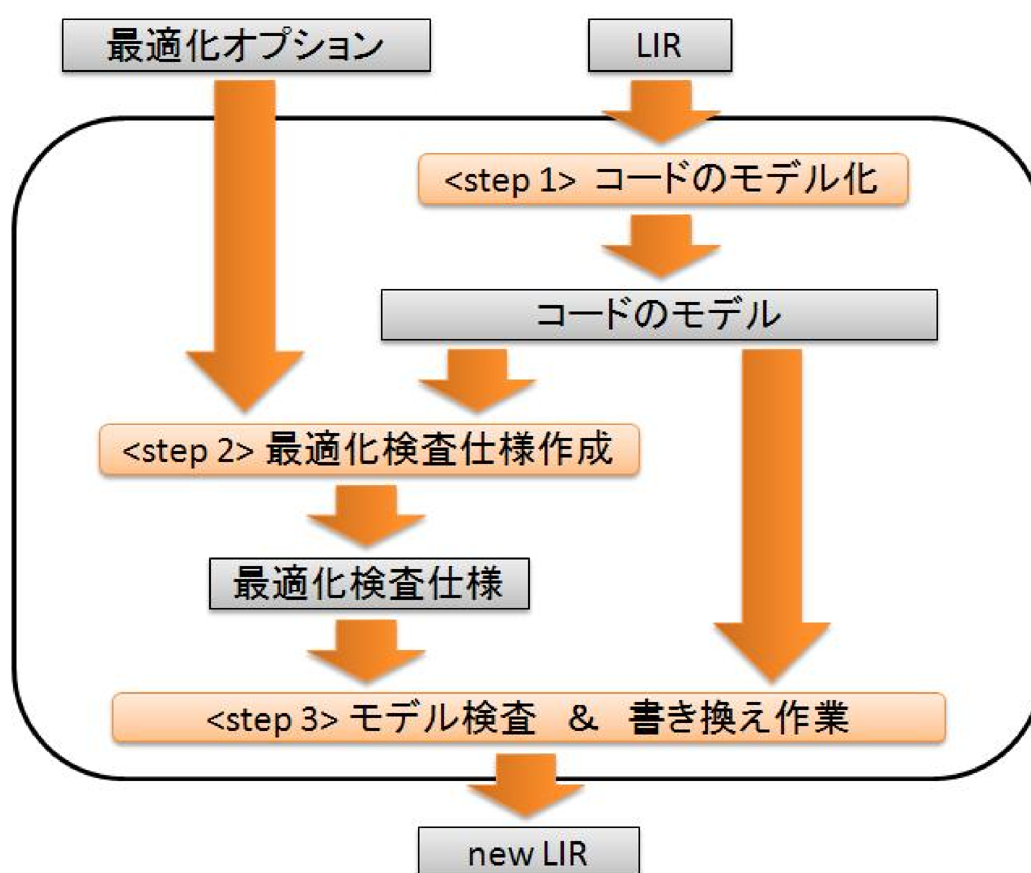


図 6.1: 提案システムの概要

6.2 ＜step 1＞ コードのモデル化

このステップでは，与えられた中間言語を，状態の単位を命令文とした制御フローモデルにする．図 6.2 の C 言語の例[1]を制御フローモデルとしたものを図 6.3 に示す．その際，初期状態として entry ノードを，終了状態として exit ノードを追加する．

```
int x, y; // static変数

int test() {
  int a;
  x = 1;
  a = x;
  if  ( y > 0 ) {
    y = x − 1;
  } else {
    y = x + 1;
  }
  a = y;
  return  a;
}
```

図 6.2: 中間言語のモデル化

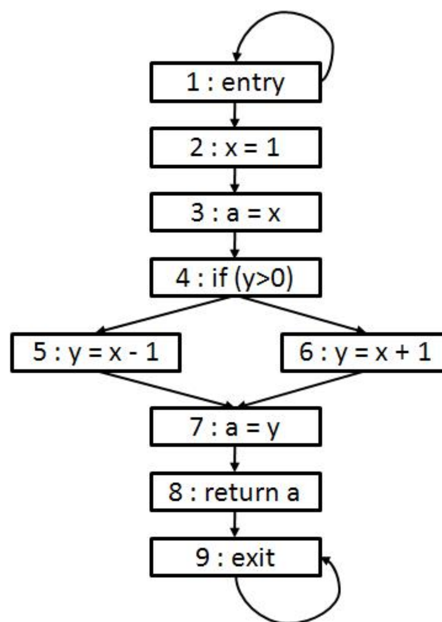


図 6.3: 中間言語のモデル化

次に，各状態に対してどのような原子命題が真となるかを解析する．解析する原子命題の種類と意味は以下である．

- $entry$, $exit$

  それぞれ entry ノード，exit ノードの場合に真となる命題．

- $stmt(M)$

  M は命令．その状態が示す命令が M の時に真となる命題．

- $def(v)$

  v は自由変数．変数 v が定義されているときに真となる命題．

- $use(v)$

  v は自由変数．変数 v が使用されているときに真となる命題．

- $trans(e)$

  e は自由変数を含む式．式 e で使用される変数が，その状態で値を更新していない (つまり使用されていない) ときに真となる命題．

[1]本論文で扱うのは LIR であるが，読者の読みやすさのために C 言語で示す

図 6.3 の各状態に対する原子命題を表 6.1 に示す.

状態	原子命題			
1	$entry$			
2	$stmt(x=1)$,	$def(x)$,	$use(1)$,	$trans(y)$
3	$stmt(a=x)$,	$def(a)$,	$use(x)$,	$trans(y), trans(x=1)$
4	$stmt(if(y>0))$,		$use(y), use(0)$,	$trans(a=x), trans(y), \cdots$
5	$stmt(y=x-1)$,	$def(y)$,	$use(x), use(1)$,	$trans(x=1), trans(a=x)$
6	$stmt(y=x+1)$,	$def(y)$,	$use(x), use(1)$,	$trans(x=1), trans(a=x)$
7	$stmt(a=y)$,	$def(a)$,	$use(y)$,	$trans(x=1), trans(y=x+1)$
8	$stmt(return\ a)$,		$use(a)$	$trans(a=y), trans(x=1), \cdots$
9	$exit$			

表 6.1: 図 6.3 の各状態の原子命題

6.3 ＜step 2＞最適化検査仕様作成

このステップは＜ step 1 ＞で作成したコードのモデルをもとに，最適化オプションに対応する最適化検査仕様というものを作成する．

6.3.1 最適化検査仕様

最適化検査仕様とは最適化の内容を独自の表現で表したものであり，大きく MATCH 部・CONDITION 部・PROCESS 部の 3 パートで構成されている．なお，ここに表れている変数は全て自由変数である．

MATCH

変数 = 式

CONDITION

$point_$文字列 : CTL-FV 式

$edge_$文字列 : $point_$ 文字列 $\rightarrow$ $point_$ 文字列

PROCESS

$point_$文字列 : $Delete$ 命令文

$point_$文字列 : $Replace$ 式 $\rightarrow$ 式

MATCH 部

最適化の対象となる式の形を表すもの．例えば

$$\textbf{MATCH} \quad v := e$$

の左辺 v を変数，右辺 e を二項式とすると，$x := y + z$ は対象となるが，$x := y$ は対象外となる．

CONDITION 部

最適化において成り立つべき条件を CTL-FV で表すもの．

条件式とすぐ後で述べる部分式を複数書いたり，それらに式名をつけることができる．条件式は書換えをするときに成り立つべき条件である．条件式には PROCESS 部での処理と対応させるための式名を付ける．部分式は長い条件式を書きやすいように，分解して書けるようにするためのものである．条件式の右側に部分式の式名が書かれているときは，その名前が表す論理式に置き換えて使用する．

PROCESS 部

CONDITION 部で表された条件が成り立つ式に対して処理する内容を表すもの．

CONDITION 部の条件式を満たした命令文または辺の集合をどのように変換するかを記す．変換を表す処理文に条件式と同じ式名を付けることによって，条件式との対応関係をつける．

処理式の種類と意味は以下である．

- $Delete$: 削除
- $Replace$: 書き換え

6.3.2 無用命令除去の最適化検査仕様

以下で v は変数，e は式である．

MATCH

$v = e$

CONDITION

$unreachable : \neg(\overleftarrow{E}\ [true\ U\ entry])$

$unuse : AG\ \neg use(v)$

$unuseTillDef : A\,[\neg use(v)\ U\ (def(v) \wedge \neg use(v))]$

$delete : (stmt(v := e) \wedge AX\ (unuse \vee unuseTillDef)) \vee unreachable$

PROCESS

$delete : Delete\ v = e$

この式の直感的な意味を図 6.4 に表す．

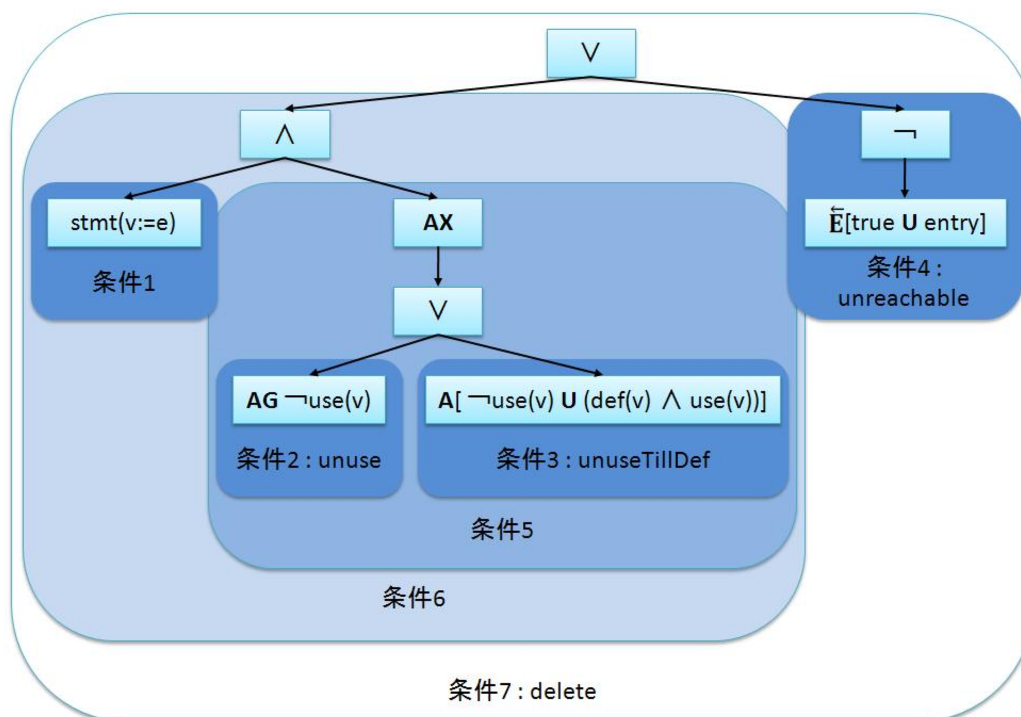


図 6.4: 無用命令除去の CTL-FV 式の直感的意味

条件 1

検査対象となる命令

条件 2 unuse

検査対象の命令以降の全ての経路において変数 v を使用しない

条件 3 unuseTillDef

変数 v が使用されないで再定義されるまで使用されない

条件 4 unreachable

入口から到達しない文

条件 5

全ての経路において次の命令が条件 2 あるいは条件 3 を満たす

条件 6

条件 1 かつ条件 5 を満たす

条件 7 delete

条件 6 または条件 4 を満たす命令は無用命令である

6.3.3 コピー伝播の最適化検査仕様

以下で v, r は変数である．

MATCH

$v := r$

CONDITION

$trans : trans(v) \wedge trans(r)$

$avail : \overleftarrow{A}\,[trans\ U\ stmt(v := r)]$

$copypropagate : use(v) \wedge (\overleftarrow{A}X\ avail)$

PROCESS

$copypropagate : replace\ v \rightarrow r$

この式の直感的な意味を図 6.5 に表す．

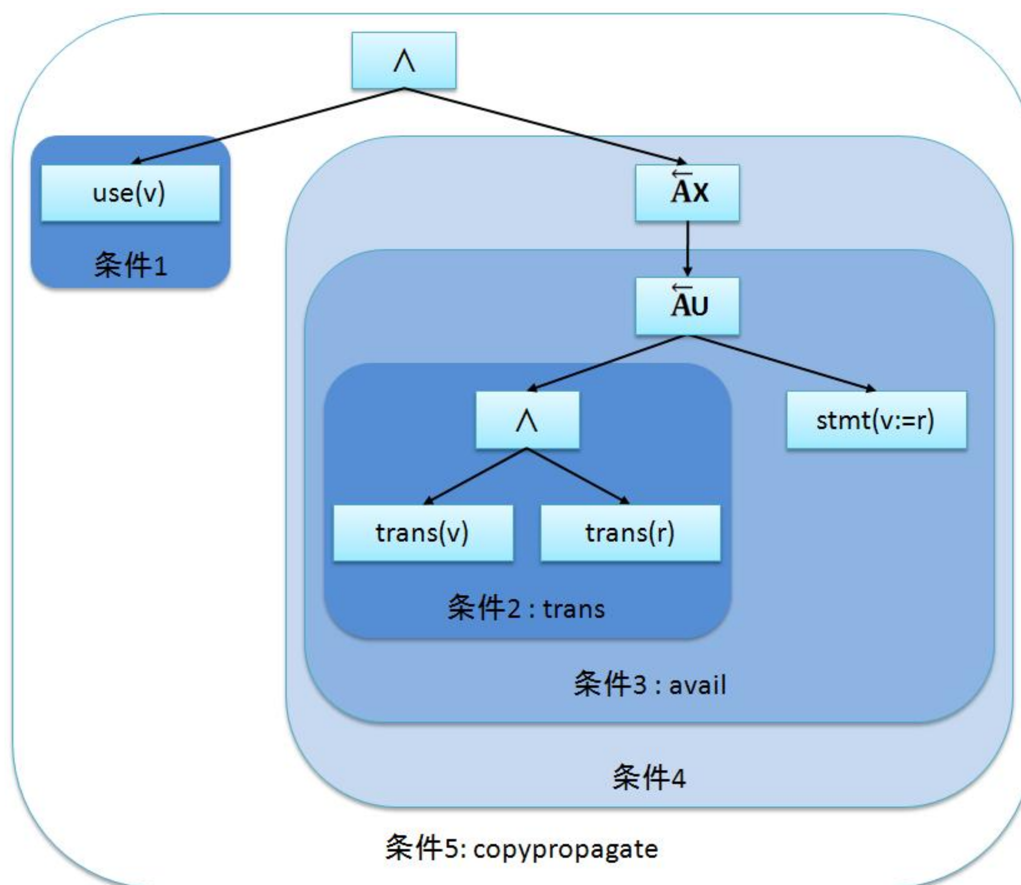


図 6.5: コピー伝播の CTL-FV 式の直感的意味

条件 1

変数 v が使用されている

条件 2 trans

変数 v や変数 r は値の変更がない

条件 3 avail

過去の全ての経路において条件 2 を満たすまま命令 v=r にいたる

条件 4

過去の全ての経路において次の命令は条件 3 を満たす

条件 5 copypropagate

条件 1 かつ条件 4 を満たす命令はコピー伝播の条件式である

6.4 自由変数の束縛

前節で説明した最適化検査仕様にあらわれる変数は自由変数である.

最適化検査仕様作成のステップでは，最適化検査仕様と先ほど作成したコードのモデルを受け取り，最適化検査仕様の自由変数を束縛していく．このステップと後の「＜ step 3 ＞モデル検査＆コード変換」を合わせた処理の流れは，最適化の種類によって異なる.

共通する大まかな流れとしては，最適化検査仕様の MATCH 部で表されている命令の形と適合した命令に対して，対応する変数同士で自由変数を束縛していく.

具体的なアルゴリズムと束縛の例を以下に示す.

6.4.1 無用命令除去での自由変数の束縛

無用命令除去の最適化検査仕様は MATCH 部に適合する命令と，PROCESS 部で処理を行う命令が同じため，自由変数の束縛と「＜ step 3 ＞モデル検査 & 書き換え作業」が各命令ごと一度ですむ.

ここでは無用命令除去での自由変数の束縛だけのアルゴリズムを以下に示す.

入力 N:モデルのノードセット

l':最適化検査仕様（自由変数の束縛前）

出力 l:自由変数を束縛した最適化検査仕様

(1) N からノード n を取り出す

取り出せる⇒ (2)，取り出せない⇒終了

(2) n の式が l' の MATCH 部と適合するかチェック

適合⇒ (3)，不適合⇒ (1)

(3) l' の CONDITION 部，PROCESS 部に対して自由変数を束縛し，l として出力

⇒ (1)

図 6.3 と無用命令除去の最適化検査仕様を例として，自由変数の束縛を行う．

1. 状態 1 の命令「$entry$」に対して MATCH 部「$v = e$」と適合するかチェック

   ⇒ False

2. 状態 2 の命令「$x = 1$」に対して MATCH 部「$v = e$」と適合するかチェック

   ⇒ True

3. 自由変数 v を x に，自由変数 e を 1 に束縛する．CONDITION 部，PROCESS 部は次のようになる．

   $x = 1$ で束縛された DCE の最適化検査仕様

   CONDITION

   $point_unreachable : \neg(\overleftarrow{E}\,[true\ U\ entry])$

   $point_unuse : AG\ \neg use(x)$

   $point_unuseTillDef : A\,[\neg use(x)\ U\ (def(x) \wedge \neg use(x))]$

   $point_delete : (stmt(x = 1) \wedge AX\ (point_unuse \vee point_unuseTillDef)) \vee point_unreachable$

   PROCESS

   $point_delete : Delete\ x = 1$

4. 状態 3 の命令 $a = x$ に対して MATCH 部 $v = e$ と適合するかチェック

   ⇒ True

   …

6.4.2 コピー伝播（定数伝播を含む）での自由変数の束縛

コピー伝播は自由変数の束縛そのものの処理は変わらないが,「＜ step 3 ＞モデル検査 & 書き換え作業」と合わせた処理の流れが無用命令除去と異なる．具体的には，最適化検査仕様の MATCH 部に適合する式と CONDITION 部で処理を行う命令とが異なるため，まず全命令に対して自由変数の束縛を行い，その都度束縛した最適化検査仕様を集合として保存しなくてはならない．

アルゴリズムを以下に示す．

入力 N:モデルのノードセット

l':最適化検査仕様（自由変数の束縛前）

L:自由変数を束縛した最適化検査仕様の集合（入力時は空）

出力 L

(1) N からノード n を取り出す

取り出せる⇒ (2)，取り出せない⇒終了

(2) n の式が l' の MATCH 部と適合するかチェック

適合⇒ (3)，不適合⇒ (1)

(3) l' の CONDITION 部，PROCESS 部に対して自由変数を束縛し，L に入れる

⇒ (1)

6.5　＜step 3＞モデル検査 & 書き換え作業

このステップでは，各ノードに対応した束縛された CONDITION 部の CTL-FV 式とコードのモデルを受け取り，モデル検査を行って適切な処理を行う．

前節で述べた通り，最適化の種類によって「＜step 2＞最適化検査仕様作成」と合わせた処理の流れが異なっている．

アルゴリズムを以下に示す．

6.5.1　無用命令除去でのモデル検査 & 書き換え作業

無用命令除去でのモデル検査 & 書き換え作業だけのアルゴリズムを以下に示す．

入力 N:モデルのノードセット

l:最適化検査仕様（自由変数の束縛前）

(1) N からノード n を取り出す

取り出せる⇒ (2)，取り出せない⇒終了

(2) n の式が l' の MATCH 部と適合するかチェック

適合⇒ (3)，不適合⇒ (1)

(3) l の PROCESS 部の条件式 (CTL-FV 式)ctl と n に対してモデル検査を行う

True ⇒ (4)，False ⇒ (1)

(4) ctl に対応した l の PROCESS 部の処理を行う

⇒ (1)

前述の通り，無用命令除去では最適化検査仕様の MATCH 部と適合する命令と PROCESS 部で処理される命令が同じため，実際はモデルの各ノードにおいて「＜処理 2＞最適化検査仕様」「＜step 3＞モデル検査 & 書き換え作業」を同時に行うことができる．このステップを合わせたアルゴリズムをアルゴリズム 6.1 に示す．

アルゴリズム 6.1 の説明を以下に示す．

N モデルのノード集合

アルゴリズム 6.1 DCE の＜step2＞&＜step3＞

DCE()

```
1: for all n ∈ N do
2:    if match(n) then
3:       ctl ← condition(n)
4:       if modelCheck(ctl, n) then
5:          process(n)
```

n モデルのノード

ctl CTL 式

match(n)

最適化検査仕様の MATCH 部にあたるメソッド．ノード n の命令が MATCH 部の形に適合したら True を，不適合なら False を返す．

condition(n)

最適化検査仕様の COINDITION 部にあたるメソッド．ノード n に対応して COINDITION 部にある CTL-FV 式に自由変数の束縛を行い，その CTL-FV 式を返す．

process(n)

最適化検査仕様の PROCESS 部にあたるメソッド．ノード n に対して PROCESS 部で表されている処理を施す．

modelCheck(ctl,n)

モデル検査をするメソッド．ノード n において CTL-FV 式 *ctl* が成り立てば True，成り立たなければ False を返す．

図 6.3 と DCE の最適化検査仕様を例として，モデル検査 & 書き換え作業の流れを示す．

1. ノードの集合 N から状態 1「$entry$」を取り出す

   ⇒取り出せたので True

2. $entry$ が DCE の最適化検査仕様の MATCH 部「$v = r$」と適合するかチェック

   ⇒ False

3. ノードの集合 N から状態 2「$x = 1$」を取り出す

   ⇒取り出せたので True

4. $x = 1$ が DCE の最適化検査仕様の MATCH 部「$v = r$」と適合するかチェック

   ⇒適合するので True

5. $v \rightarrow r$，$r \rightarrow 1$ として自由変数 v, r を束縛する

6. 状態 2「$x = 1$」で自由変数を束縛された CONDITION 部の CTL 式と状態 2 に対してモデル検査を行う

   ⇒ False

7. ノードの集合 N から状態 3「$a = x$」を取り出す

   ⇒取り出せたので True

8. $a = x$ が DCE 最適化検査仕様の MATCH 部「$v = r$」と適合するかチェック

   ⇒ True

9. $v \rightarrow a$, $r \rightarrow x$ として自由変数 v, r を束縛する

10. 状態 3「$a = x$」で自由変数を束縛された CONDITION 部の CTL 式と状態 2 に対してモデル検査を行う

    ⇒ True

11. 状態 3「$a = x$」に対して PROCESS 部で表されている処理（ここでは削除）を行う

12. ノードの集合 N から状態 4「$if(y > 0)$」を取り出す

    $\cdots$

6.5.2 コピー伝播（定数伝播を含む）でのモデル検査 & 書き換え作業

コピー伝播（定数伝播を含む）でのモデル検査 & 書き換え作業のアルゴリズムを以下に示す.

入力 N:モデルのノードセット

L:自由変数を束縛した最適化検査仕様の集合

(1) N からノード n を取り出す

取り出せる⇒ (2), 取り出せない⇒終了

(2) L から自由変数を束縛した最適化検査仕様 l を取り出す

取り出せる⇒ (3), 取り出せない⇒ (1)

(3) l の PROCESS 部の条件式 (CTL-FV 式)ctl と n に対してモデル検査を行う

True ⇒ (4), False ⇒ (2)

(4) ctl に対応した l の PROCESS 部の処理を行う

⇒ (2)

アルゴリズム 6.2 CPYP の＜ step2 ＞&＜ step3 ＞

CPYP()

1: **for all** $n \in N$ **do**
2:   **if** $match(n)$ **then**
3:     $ctl \leftarrow condition(n)$
4:     $ctlSet\ add\ ctl$
5: **for all** $n \in N$ **do**
6:   **for all** $ctl \in ctlSet$ **do**
7:     **if** $modelCheck(ctl, n)$ **then**
8:       $process(n)$

前述の通り，コピー伝播（定数伝播を含む）では最適化検査仕様の MATCH 部と適合する命令と PROCESS 部で処理される命令が異なるため,「＜ step 2 ＞最適化検査仕様」「＜ step 3 ＞モデル検査 & 書き換え作業」はそれぞれ行わなくてはいけない．

この流れを合わせた，実際のアルゴリズムをアルゴリズム 6.2 に示す.

アルゴリズム 6.2 の説明を以下に示す.

N モデルのノード集合

n モデルのノード

ctlSet CTL 式の集合

ctl CTL 式

match(n)

最適化検査仕様の MATCH 部にあたるメソッド．ノード n の命令が MATCH 部の形に適合したら True を，不適合なら False を返す.

condition(n)

最適化検査仕様の COINDITION 部にあたるメソッド．ノード n に対応して COINDITION 部にある CTL-FV 式に自由変数の束縛を行い，その CTL-FV 式を返す.

process(n)

最適化検査仕様の PROCESS 部にあたるメソッド．ノード n に対して PROCESS 部で表されている処理を施す.

modelCheck(ctl,n)

モデル検査をするメソッド．ノード n において CTL 式 ctl が成り立てば True，成り立たなければ False を返す.

第Ⅳ部

実装

第7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．

7.1　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[5].

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[7, 8, 9, 18]，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.

7.2　構成

図7.1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

7.3　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上での最適化を対象に行った．この節では，LIRの構成について説明する[12].

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7.2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．

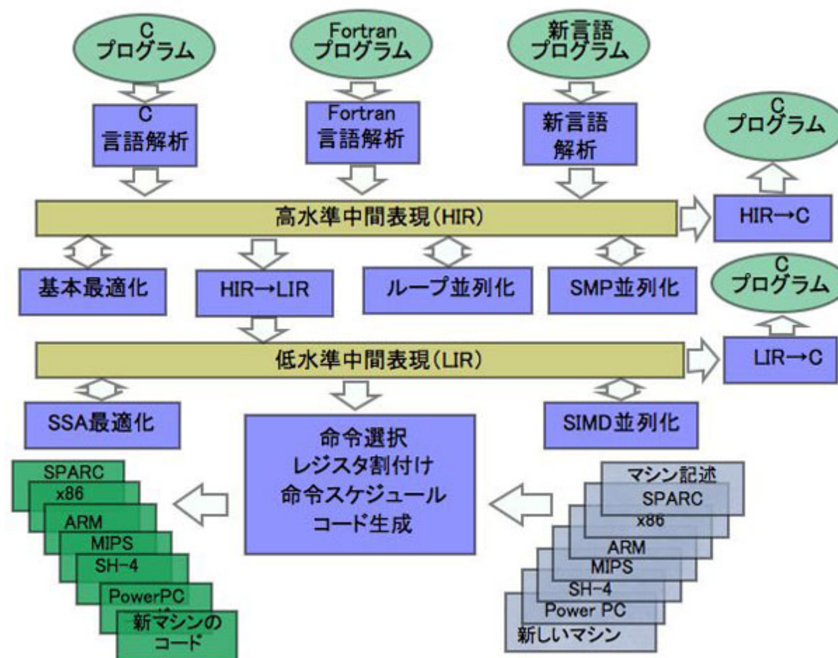


図 7.1: COINS の構成図

Function は，局所変数の記号表である localSymtab と，命令の文の列から構成されている．文の列は LIR 生成直後に制御フローグラフに変換され，この形で保持される．FlowGraph が制御フローグラフを表す．FlowGraph の構成を図 7.3 に示す．FlowGraph は基本ブロックを表す BasicBlk のリストから構成されている．また，BasicBlk は，先行ブロックのリストである predList，後続ブロックのリストである succList，ブロック内の文の列である instrList から構成されている．

instrList の構成を図 7.4 に示す．instrList の要素である LIR の文は，BiLink という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 7.4 の LirNode が L 式を表している．

7.4 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．

この節では，L 関数上で制御フローモデルを定義する[1]．以下，制御フローモデルを $M = (N, E, L)$ とする．

[1] LIR の正確な言語仕様は，文献 [6] 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．

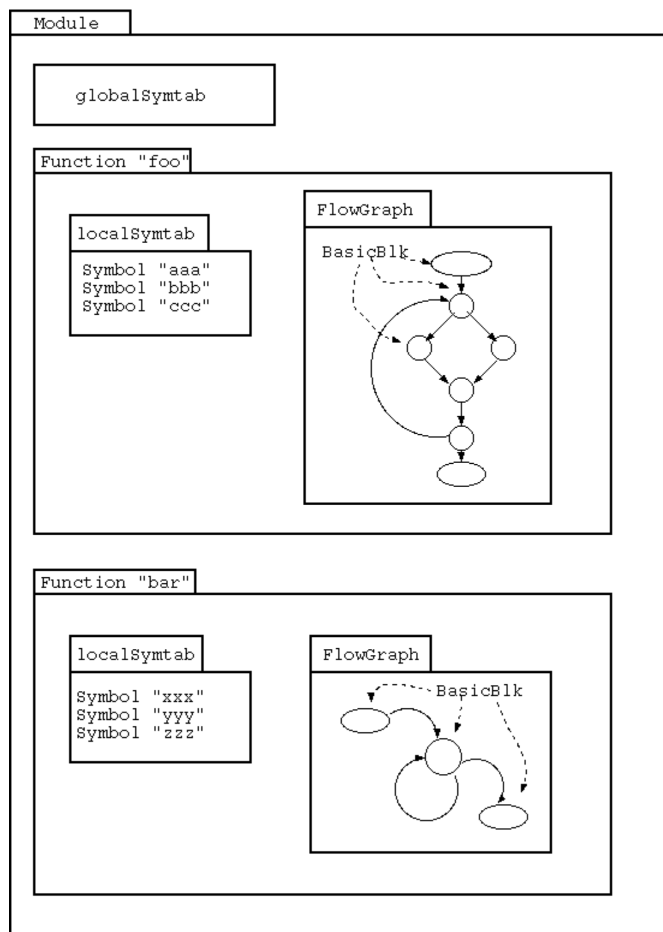


図 7.2: Module の構成

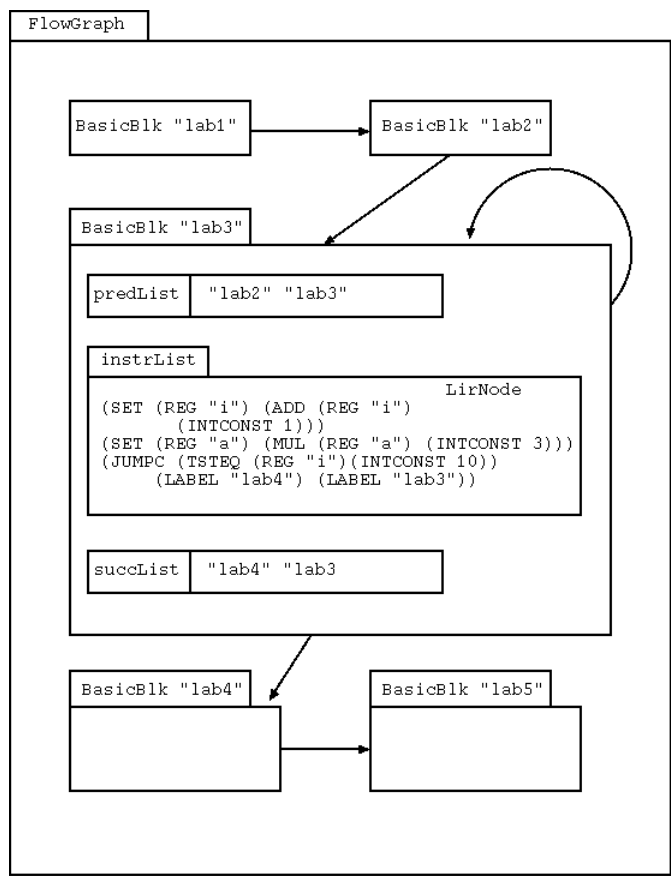


図 7.3: FlowGraph の構成

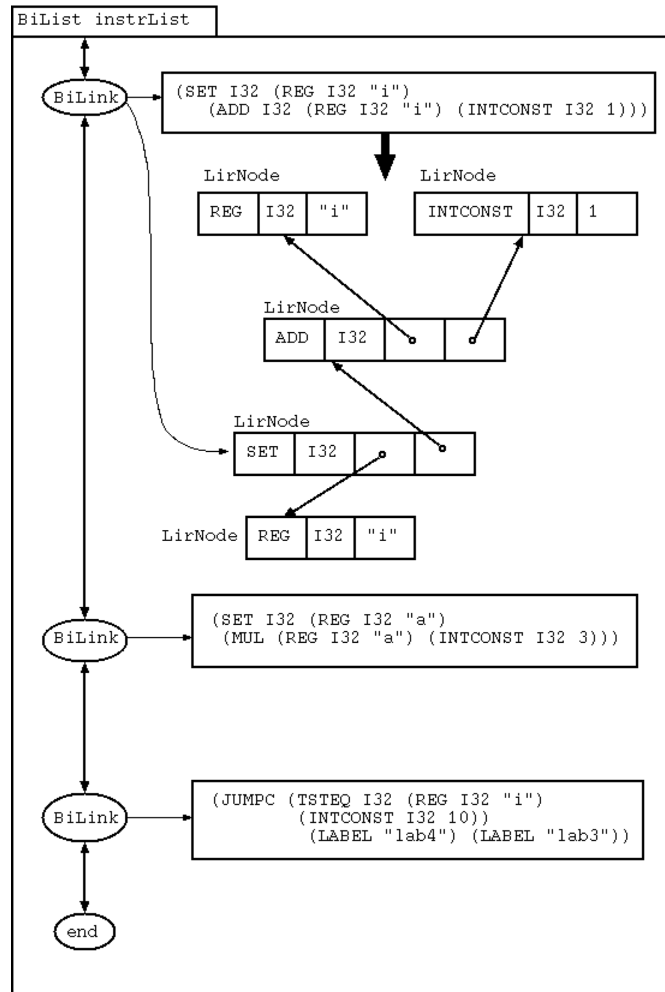


図 7.4: instrList の構成

7.4.1 モデルの状態

モデルの状態の集合 N の要素は，LIR の文とする．すなわち，図 7.4 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に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．

7.4.2 状態間の遷移関係

文献 [6] 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 7.3 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．

7.4.3 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，$stmt$(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[2]，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[3]．

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．

[2]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 [17]．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．

[3]LIR には部分レジスタを表す SUBREG 式というのもあるが，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．

各原始命題

定義 7.1 (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 7.2 (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．

定義 7.3 (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8章　実験と考察

8.1　実験

8.1.1　実験環境

本研究ではCOINS上で実装されている既存のSSA最適化と本研究で実装した最適化に対して，目的コードの実行時間とコンパイルにおいてかかる最適化の時間を比較した．
実験に使用した計算器の主要なスペックを表8.1に示す．
ベンチマークとしてSPEC CPU2000から以下の7種類を使用した．

- 164.gzip
- 175.vpr
- 181.mcf
- 183.equake
- 197.parser
- 256.bzip2
- 300.twolf

詳しくは，無用命令除去に関する実験では上の7種類を，コピー伝播に関する実験では上から197.parser，300.twolfを除く5種類を使用した．
実験において比較した最適化の種類は無用命令除去，コピー伝播で，COINS上に実装されているSSA最適化と本研究で実装した最適化を対象とした．コピー伝播に関しては，最適化を行った後に無用命令が現れることを期待するものなので，無用命令除去と合わせて実行した．
実験でCOINS上の最適化器に対するCOINSオプションは以下である．

CPU	UltraSPARC　750MHz
OS	SunOS 5.8
JavaVM	1.5.0_01
Memory	1Gbyte
COINS	1.4.4.4
Benchmark	SPEC CPU2000 version 1.2

表 8.1: 実験環境

- 無用命令除去

  -coins:ssa-opt=prun/dce/srd3

  prun：SSA 形式への変換を行うオプション

  dce：無用命令除去のオプション

  srd3：SSA 逆変換を行うオプション

- コピー伝播

  -coins:ssa-opt=prun/cpyp/dce/srd3

  cpyp：コピー伝播のオプション

  コピー伝播の最適化を行った後，無用命令除去を行う

8.1.2 実行時間

無用命令除去の最適化に関して，目的コードの実行時間の結果を図 8.1 に示す．これは最適化無しの実行時間を 1 とした時の相対値のグラフである．

・**no-opt** 最適化無し

・**coins-dce** COINS 上で実装されている無用命令除去

・**my-dce** 本研究で実装した無用命令除去

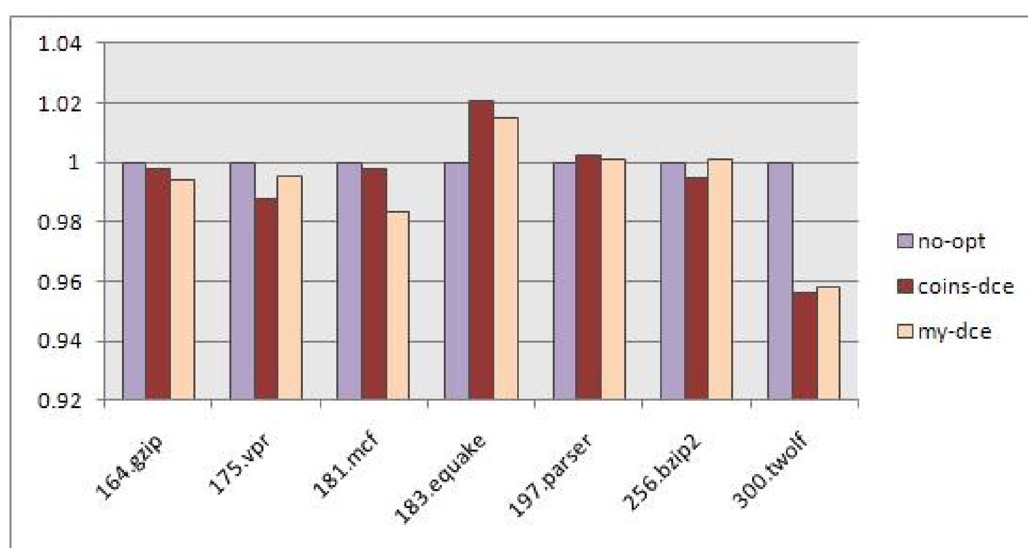


図 8.1: 無用命令除去の実行時間

図 8.1 を見ると，本研究で実装した無用命令除去は COINS 上で実装されているそれとほぼ同じ効果を示している．

300.twolf1 では，最適化無しと比べると COINS 上の最適化と同様 4% 程度実行時間が改善された．

コピー伝播と無用命令所に関して，目的コードの実行時間の結果を図 8.2 に示す．これは最適化無しの実行時間を 1 とした時の相対値のグラフである．

- **no-opt** 最適化無し
- **coins-cpyp-dce** COINS 上で実装されているコピー伝播，無用命令除去
- **my-cpyp-dce** 本研究で実装したコピー伝播，無用命令除去

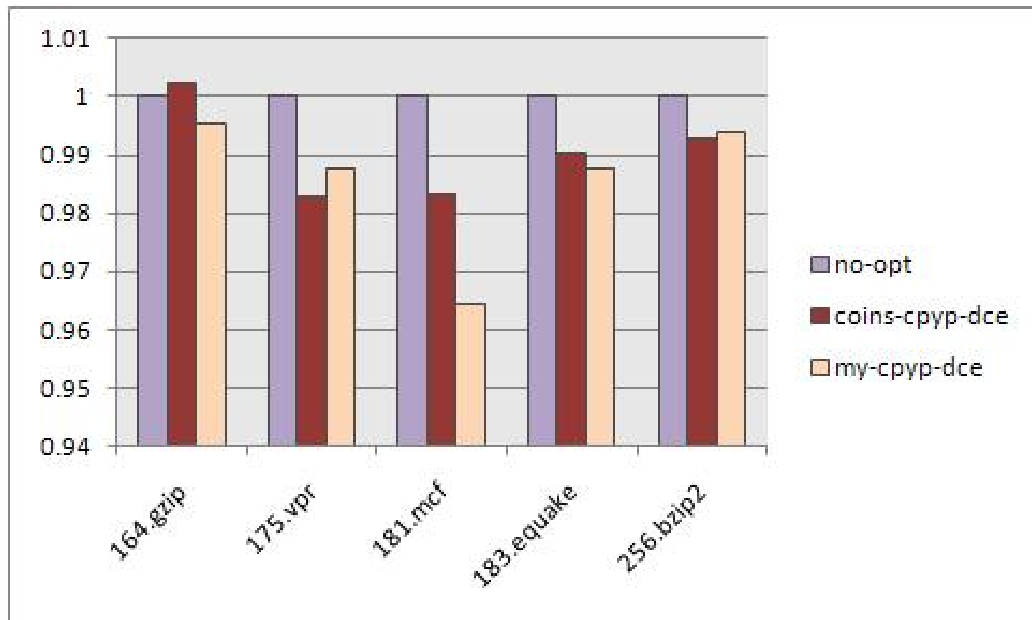


図 8.2: コピー伝播と無用命令除去の実行時間

図8.2を見ると，181.mcf以外では本研究で実装したコピー伝播と無用命令除去は，COINS上で実装されているそれとほぼ同じ効果を示している．

さらに，181.mcf においては最適化無しと比べて 3.5% 程度，COINS 上での最適化と比べて 2% 程度実行時間が改善された．

8.1.3 最適化時間

無用命令除去とコピー伝播に関して，コンパイル時の最適化時間の結果を表 8.2 に示す．これは COINS 上に実装されている最適化の最適化時間を 1 とした時の相対値のグラフである．

表の行項目の詳しい説明を以下に示す．

- $dce = \dfrac{\text{本研究で実装した } DCE \text{ の最適化時間}}{COINS \text{ 上で実装されている } DCE \text{ の最適化時間}}$

- $cpyp \cdot dce = \dfrac{\text{本研究で実装した } CPYP/DCE \text{ の最適化時間}}{COINS \text{ 上で実装されている } CPYP/DCE \text{ の最適化時間}}$

	dce(my/coins)	cpyp・dce(my/coins)
164.gzip	58.14	338.16
175.vpr	73.56	277.72
181.mcf	21.64	64.52
183.equake	315.98	3733.04
197.parser	3.02	-
252.eon	7.18	-
254.gap	4.23	-
255.vortex	3.27	-
256.bzip2	4.85	4115.69
300.twolf	1.52	-

表 8.2: コンパイル時の最適化時間の比

表の列項目は検査したSPEC2000の種類である．

表8.2を見ると，本研究で実装した最適化はCOINS上で実装されている最適化に比べて，無用命令除去は最適化時間が1.5〜315倍程度，コピー伝播/無用命令除去は64〜4000倍とバラつきはあるが，全体的に本研究で実装した最適化はコンパイル時にかなりの時間を有する結果となった．
特にコピー伝播/無用命令除去での183.equake，256.bzip2は約4000倍の差が出てしまい，実用的な数値から大分離れてしまった．

8.2 考察

コンパイル時にかかる最適化の時間に関して，本研究の最適化がCOINS上で実装されている最適化に比べて，無用命令除去の場合で300倍程度，コピー伝播の場合で4000程度最悪の場合で4000倍以上かかってしまうことについて考察する．
アルゴリズム6.1，アルゴリズム6.2より，ノード数Nのモデルに対する，本研究で実装した無用命令除去のアルゴリズムの計算量は，$O(N^2)$，本研究で実装したコピー伝播のアルゴリズムの計算量は，$O(N^3)$となる．COINS上で実装される無用命令除去のアルゴリズムの計算量が$O(N)$であることを考えると，コード数がモデルのノード数にほぼ比例の関係にあるため，それぞれ300倍，4000倍という結果になったと考えられる．
考えられる原因としては以下のことが挙げられる．

- 非効率なモデル検査器
- 無駄なCTL式

8.2.1 非効率なモデル検査器

本研究ではモデル検査の実装にシンプルなアルゴリズムを採用した．そのためモデル検査にかかる計算コストが，検査対象であるモデルの状態数に大きく依存してしまったモデル検査器となってしまった．

モデル検査の分野では二分決定グラフ (Binary Decision. Diagram, BDD) や充足可能性問題への帰着など高速化を図る研究が盛んに進められているため，そういったモデル検査器を採用することで，この問題はある程度改善されると予想される．

8.2.2 無駄な CTL 式

本研究で実装した最適化システムのアルゴリズムでは，まず最適化検査仕様の MATCH 部に適合する式全てに対してモデル検査を行い，その結果が真であったとき初めてコード変換の処理を行う．この MATCH 部に適合するか否かは，単純に式の形のみを見て判断しているので，コード変換の対象となりえない命令に対する無駄な処理が非常に多くなってしまう．

これは最適化対象となるコードの大きさが大きい時などは致命的となり，その結果，既存の最適化アルゴリズムと比べてコンパイルにおける最適化時間が 4000 倍となってしまったと予想される．

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 [10, 11]．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 $I \to I'$ if ϕ 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 ϕ 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，無用命令除去・定数伝播・ループ不変式の移動を実際に証明した．

9.2　方の研究

方の研究 [22] は，Lacey らの研究 [10, 11] を元にして Java バイトコードの解析系である soot[20] 上で CTL-FV とモデル検査を利用した JAVA 最適化器を作成した．

実装した最適化の種類は無用命令除去・コピー伝播（定数伝播を含む）・部分冗長性除去の 3 種類で，無用命令除去・コピー伝播（定数伝播を含む）を表す CTL-FV 式は Lacey らの研究を元にして独自に改良したものを使用している．なお，部分冗長性除去に関する CTL-FV 式は，データーフロー解析を行い最適化する Paleri のアルゴリズム [14] を元に独自に定式化したものを使用している．

9.3　Warburton の研究

Warburton の研究 [21] では方の研究 [22] と同様に Lacey らの研究 [10, 11] を元にして Java バイトコードの解析系である soot[20] 上で CTL-FV とモデル検査を利用した JAVA 最適化器を作成した．

実装した最適化の種類は無用命令除去・定数伝播・Lazy Code Motion の 3 種類で，それぞれの最適化の特徴を表す CTL-FV 式は方の研究に比べて Lacey らの研究により忠実な形で使用している．その分，最適化の種類の幅が方の研究に比べて狭くなっている．

第10章　まとめ

本研究では時相論理 CTL-FV とモデル検査技術を利用した C コンパイラ最適化器を COINS 上で実装した．本研究の特徴は次のとおりである．

- Lacey らの研究 [10, 11] を元にした，CTL-FV とモデル検査技術を利用したコンパイラ最適化の手法を C コンパイラ最適化器に初めて適用した．また，その性能を SPEC2000 という商用のコンパイラでも使われるようなベンチマークを通して，その性能を評価した．
- システムの実装において，COINS 上の LIR という機械語に近い中間言語を対象として行った．

実験の結果は，目的の性能としては既存のものとほぼ同程度の成果を上げることができた．それに加え，本研究での最適化器は既存のものに比べて，行うコード変換が証明可能であるため有益である．

しかし，コンパイル時の最適化にかかる時間は既存のものと比べてもっとも大きくて 4000 倍程度かかってしまい，実用化にはまだ技術的な壁が高いことが判明した．

第11章　今後の課題

今後の課題としては，主に次の 2 つが挙げられる．

冗長な検査式の削除

8.2.1 節でも述べた通り，本研究で実装した最適化システムにはコード変換対象外の命令に対する CTL-FV 検査式が非常に多く混入する可能性があり，それがコンパイル時の最適化時間に大きく影響を与えている．この検査式の冗長性は，MATCH 部で適合と判定する基準が単純に式の形だけを見ていることにある．そこで，今後はこの冗長性を無くすような工夫を行えば，コンパイル時の最適化時間の問題も大幅に改善されると思われる．

モデル検査の高速化

こちらも 8.2.2 節で述べた通り，本研究ではモデル検査器の実装に一番シンプルなアルゴリズムを適用した．そのため, 今後二分決定グラフ (Binary Decision. Diagram, BDD) や充足可能性問題への帰着といった検査の高速化を図る手法を取り入れたモデル検査器を使用すれば，大幅な検査時間の短縮が望めると思われる．

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 OG の方玲さんにも，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] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java 2nd ed.* Cambridge University Press, 2002.

[3] 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.

[4] E. M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[5] COINS Project. COINS home page. `http://www.coins-project.org/`.

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

[7] 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.

[8] GNU Project. GCC homepage. `http://gcc.gnu.org/`.

[9] IBM. Jikes Research Virtual Machine home page. `http://jikesrvm.sourceforge.net/`.

[10] 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, New York, NY, USA, 2002. ACM.

[11] 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.

[12] 森公一郎. COINS 新 LIR 内部構造, 2003. `http://www.coins-project.org/050303/base/BackEnd/NewLir.html`.

[13] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.

[14] Vineeth Kumar Paleri, Y. N. Srikant, and Priti Shankar. A simple algorithm for partial redundancy elimination. *SIGPLAN Not.*, Vol. 33, No. 12, pp. 35–43, 1998.

[15] Amir Pnueli. The temporal logic of programs. In *In Proceeding of the 18th IEEE Symposium on Foundations of Computer Science*, pp. 46–77, 1977.

[16] 佐々政孝. プログラミング言語処理系. 岩波書店, 1989.

[17] 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, 2010. `http://www.is.titech.ac.jp/~sassa/coins-www-ssa/japanese/ssa-external-japanese.pdf`.

[18] Scale Compiler Group. Scale home page. `http://www-ali.cs.umass.edu/Scale/`.

[19] 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.

[20] Soot a Java optimization framework. Soot home page. `http://www.sable.mcgill.ca/soot/`.

[21] Richard Warburton and Sara Kalvala. From specification to optimisation: An architecture for optimisation of java bytecode. In *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, pp. 17–31, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] 方玲, 佐々政孝. 双方向 CTL による Java 最適化器の生成. 情報処理学会論文誌. プログラミング, Vol. 48, No. 10, pp. 76–89, 2007-06-15.