

# 双方向CTLによるJava最適化器の生成

東京工業大学  
数理・計算科学専攻

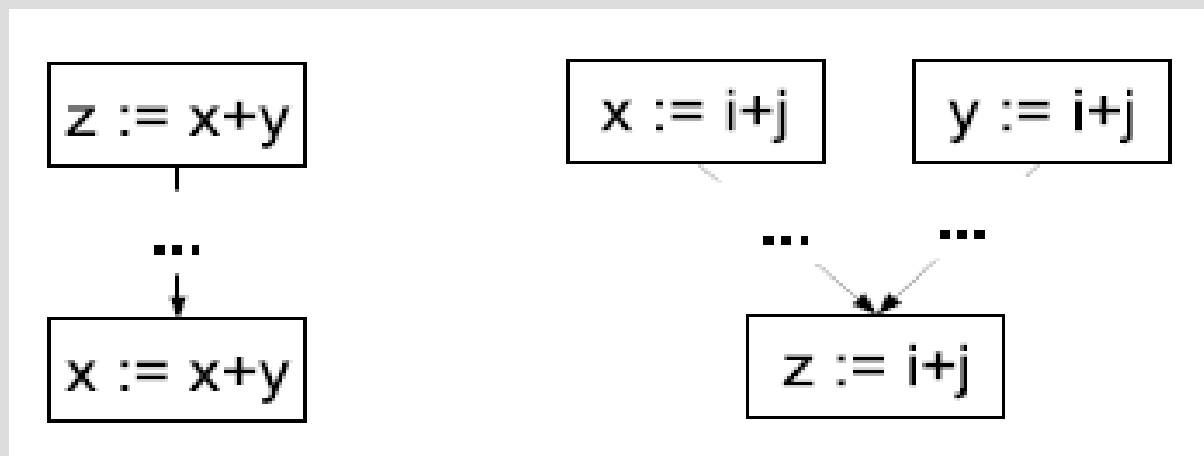
方 玲      佐々 政孝

# 研究の背景

- CTLによる最適化とは  
コンパイラの最適化は、プログラムによるものがほとんどだが、近年CTLという論理による最適化の研究も行われている。  
それは条件付き書換え規則「 $I \rightarrow I' \text{ if } \phi$ 」を用いて表現する。  
★条件付き書換え規則「 $I \rightarrow I' \text{ if } \phi$ 」を以後**最適化記述**と呼ぶ。
- この手法のメリット
  - 論理式で一行か、多くても十数行で最適化を記述できる。
  - 証明できる。

- 本研究の動機

- 2001年, 従来の研究として, Laceyらの研究は, 過去時制を含む CTL-FV を提唱し, 最適化器の正しさを証明した. しかし, 証明した式は実際の最適化の一部しか扱えない例が多い. 記述しきれないものは通常プログラムを呼び出すことになる. 実装についてもあまり触れておらず, 最適化時間などのデータもない.



- 2003年, 山岡らの研究は, 既存のモデル検査器SMVを用い, Laceyらの理論の一部を実装した. 過去時制を用いることができず, 無用命令除去しか扱えない.
- 2004年, 番らの研究は過去時制を除去することによってコピー伝播を行えるが, 除去処理の時間と, 除去によって式が長くなるため, 最適化時間がかかなり長い.

- 未解決の問題

CTL等の論理による手法で通常のアлゴリズムと同じように最適化できるか

- 本研究の貢献

本研究は「CTL等の論理による手法で通常のアлゴリズムと同じように最適化できるか」という問題の解決に向けて研究を行った。

- 通常のお最適化器に大幅に近づいたCTLによるJava最適化器を実装した。従来研究と比べて下記の点がある。

- 記述しやすい
- 最適化時間が格段に短い
- 最適化効果も十分見込める

- 実現に当たり種々試行錯誤を行った結果、問題点が明らかになり、今後に役立つ多くの知見が得られた。

- 従来の研究と比べて、本研究の独創性は次のとおりである。
  - モデル検査器は従来研究と異なり、何も変換せずに直接処理できる形で実装し、効率を大幅に向上した。
  - 最適化記述について、従来研究は条件式が特定の番号の命令文しか対象とできないが、本研究の条件式は命令文の集合を対象とすることができる。
  - 通常の実装に近い性能を持った、CTLによる最適化器の実装は本研究がはじめてである。

# 発表の構成

## 理論

- 時相論理、Kripke構造、双方向CTL

## 本研究の最適化器

- 双方向CTLモデル検査器
- 最適化記述

## 実験と考察

※ 以下は双方向 CTL を CTL<sub>b</sub> と記す.

# 時相論理と分類

- 時相論理(temporal logic)は命題の真理値が時間とともにどのように変わるかを記述する論理体系.
- 「時刻0の世界」, 「時刻1の世界」, 「時刻nの世界」..
- 時間の概念は「時間」と「空間」
- 分類
  - 分岐的か線形的か.
  - 過去時制か未来時制か.
  - ...

LTL, CTL, CTL\*, CTL<sub>b</sub>, PCTL, NCTL

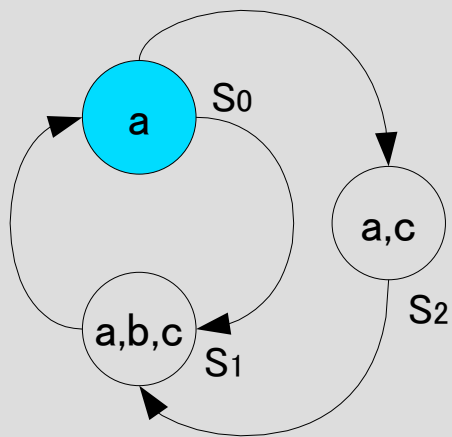
# CTL<sub>b</sub>

- CTL<sub>b</sub> (Computational Tree Logic) 演算子

経路限定子:  $A$ (futureAll),  $E$ (futureExist),  $\overset{\leftarrow}{A}$ (pastAll),  $\overset{\leftarrow}{E}$ (pastExist)

時相演算子:  $U$ (Until),  $X$ (neXt),  $G$ (Global),  $F$ (Future)

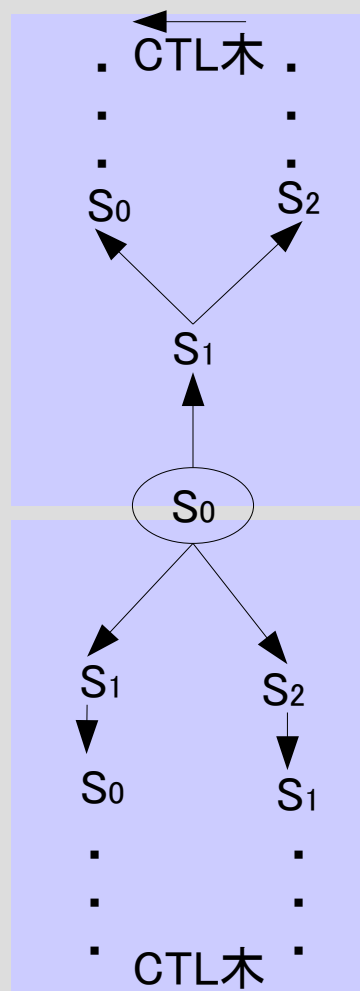
コンパイラ最適化は CTL<sub>b</sub> を用いると自然かつ簡潔に記述できる.



**Kripke構造:**

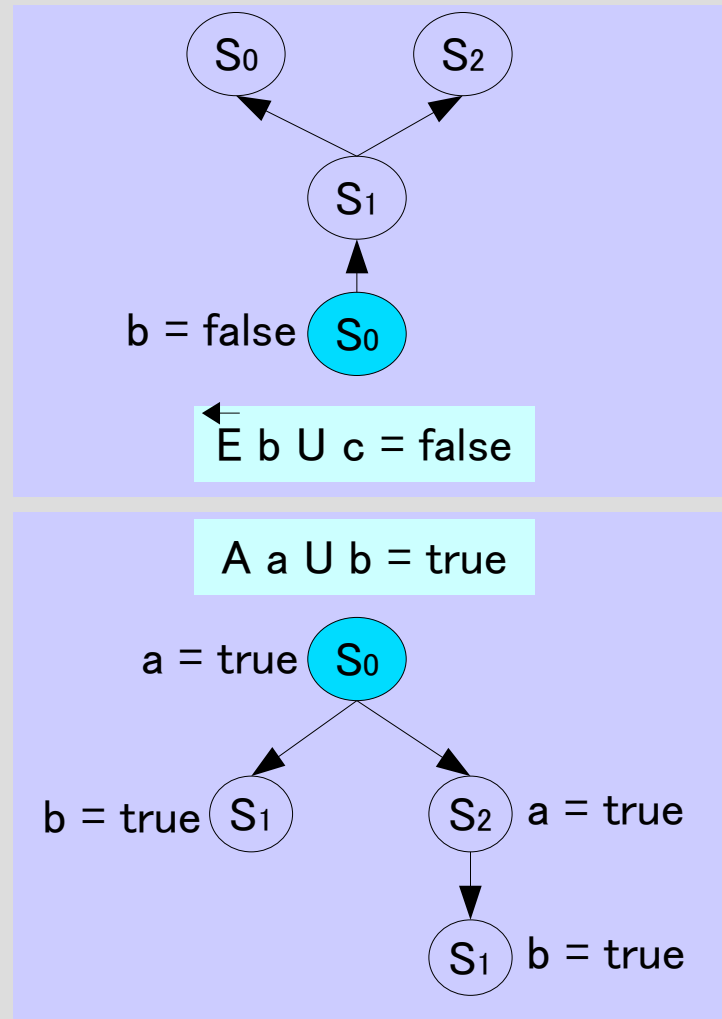
- S: 状態の集合
- $S_{start}$ : 始状態の集合
- $R \subseteq S \times S$ : 遷移関係
- $L: S \rightarrow 2^{Prop}$

Kripke構造



双方向CTL木

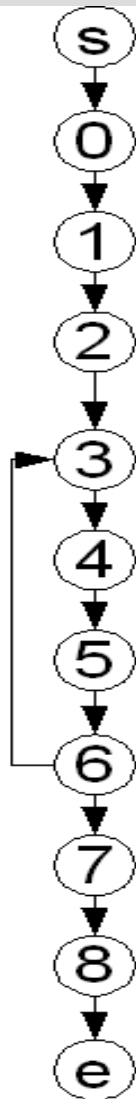
注: CTL構文木と違う



将来時制と過去時制計算の例

# プログラムと制御フローモデルの例

```
0: read N
1: X := 0
2: goto 3
3: skip
4: X := X+N
5: N := N-1
6: if N > 0 goto 3
   else 7
7: skip
8: write X
```



図はコードと制御フローモデルである.

S: 命令文の集合

R: 制御フロー遷移関係

L(n):

L(0)={stmt(readN),def(N),trans(N-1)...}

L(1)={stmt(X:=0),def(X),trans(N)...}

...

L(8) = {stmt(write X), use(X)}

# CTL-FV

- CTL-FVとは

述語  $\phi$  の変数が特定のプログラムの変数や式の定義域を指定し、束縛することによって関係付けられる.

$$AX ( ( AG \neg \text{use}(v) ) \vee A \neg \text{use}(v) U \text{def}(v) )$$

$$v \rightarrow \{ x, y, z \dots \}, e \rightarrow \{ a+b, x+y, -z \dots \}$$

すなわち

$$\{ v \rightarrow x, e \rightarrow a+b \}$$

$$\{ v \rightarrow x, e \rightarrow x+y \}$$

...

のように束縛する.

# CTL<sub>b</sub> モデル検査器

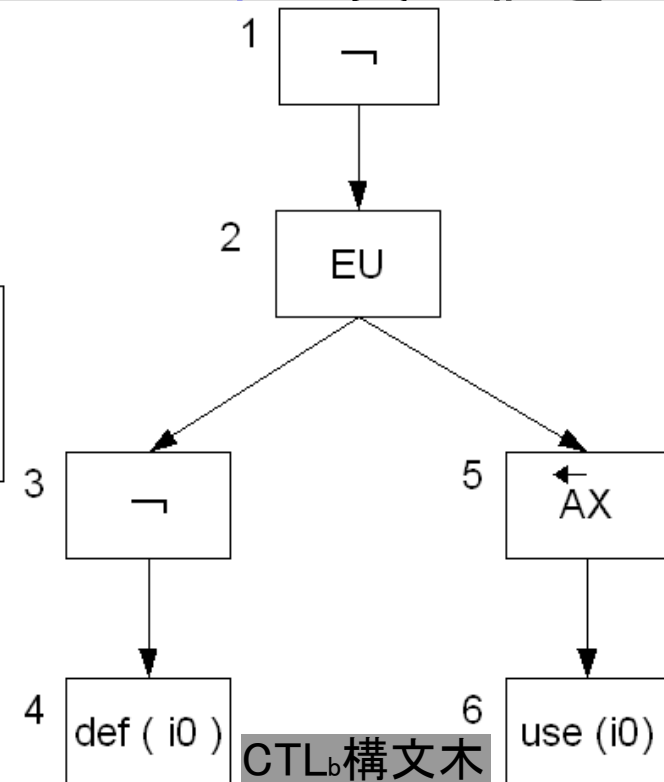
- CTL<sub>b</sub> 式の解析

CTL<sub>b</sub> 式は木構造で表せる. これはCTL<sub>b</sub>構文木という. 葉は原子述語である. 構文木ノードに深さ優先順位で番号を付ける.

- 将来時制の検査はCTL木, 過去時制の検査は $\overleftarrow{\text{CTL}}$ 木の真理値を求める.

- CTL構文木はCTL木とは異なる

$\neg( E \neg \text{def}(i0) U ( \overleftarrow{\text{AX}} \text{use}(i0) ) )$   
CTL<sub>b</sub>式の例

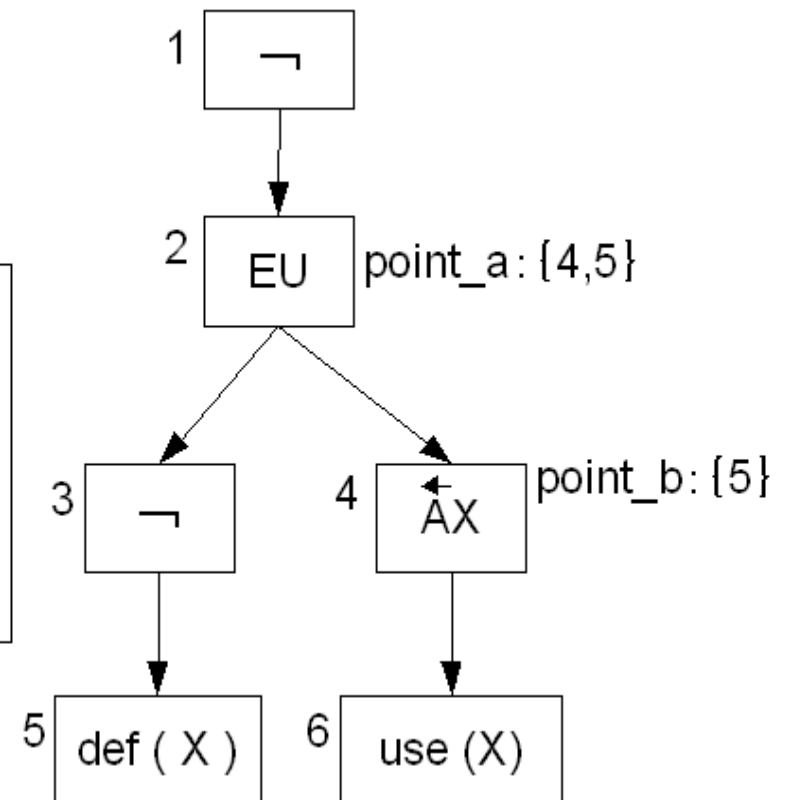
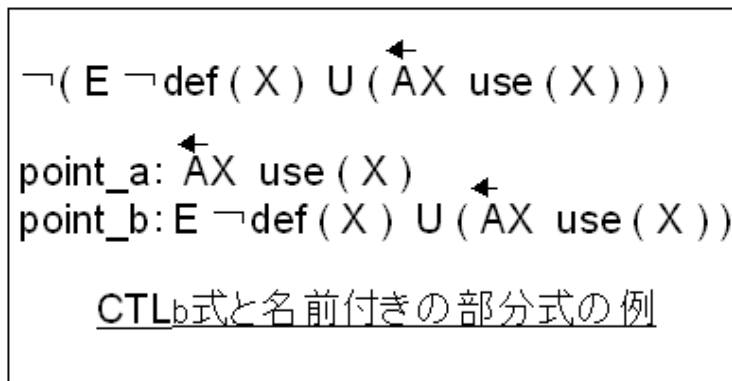




# モデル検査結果の例

- 後の書換え処理のため、モデル検査の結果を覚えておく

```
0: read N
1: X := 0
2: goto 3
3: skip
4: X := X+N
5: N := N-1
6: if N > 0 goto 3
  else 7
7: skip
8: write X
```



# 本研究の最適化記述

- 条件付き書き換え規則:

**MATCH**の形の文に対して,  
**CONDITION**の条件を満たした文であれば,  
**PROCESS**の処理をする

という流れ.

- 文法

## MATCH

変数:=式

## CONDITION

point\_文字列: CTL式(条件式)

point\_文字列: CTL式(部分式)

edge\_文字列: point\_文字列 → point\_文字列

## PROCESS

point\_文字列: Command 命令文

point\_文字列: Replace 式 → 式

edge\_文字列: EdgeSplit 命令文

例:

**MATCH**

$v := b$  (変数 := 二項式)

**CONDITION**

point\_comp:  $\text{use}(b) \wedge \text{trans}(b)$

point\_spavin: . . .

point\_spantout: . . .

point\_insert:  ~~$\text{use}(b) \wedge \text{trans}(b) \wedge \text{point\_spavin} \wedge \text{point\_spantout}$~~

point\_replace: . . .

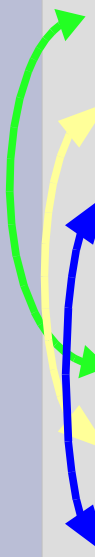
edge\_split:  $\text{point\_spavin} \rightarrow \text{point\_spantout}$

**PROCESS**

point\_insert: InsertBefore temp := b

point\_replace: Replace b  $\rightarrow$  temp

edge\_split: EdgeSplit temp := b



# MATCH

- MATCH部の例:

**MATCH**

**$v := b$**

- MATCH部の役目は  **$z := x+y$**  が対象となったとき, 束縛の可能性は,

**$\{ v \rightarrow x, b \rightarrow x+y \}$**

**$\{ v \rightarrow y, b \rightarrow x+y \}$**

**$\{ v \rightarrow z, b \rightarrow x+y \}$**

...

だが, これを

**$\{ v \rightarrow z, b \rightarrow x+y \}$**

のように一対一に関係付け, 組合せを減らすかなくすることができる.

# CONDITION

- CONDITIONの例

**point\_comp: use(b)  $\wedge$  trans(b)**

**point\_spavin: . . .**

**point\_spantout: . . .**

**point\_insert: point\_comp  $\wedge$   $\neg$ point\_spavin  $\wedge$  point\_spantout**

**point\_replace: . . .**

**edge\_split: point\_1  $\rightarrow$  point2**

- 部分式と条件式:

- 条件式: 書換えの条件.
- 部分式: 長い式を書きやすくするため.
- 辺についても条件式を書ける.

# PROCESS

- PROCESSの例

**point\_insert : InsertBefore temp := e**

**edge\_split : EdgeSplit temp := e**

**point\_replace : replace e → temp**

- PROCESSの役目:

条件式を満たした命令文または辺をどう処理するか.

- 条件式と一対一の関係で名前を付ける.

## 従来研究の記述と比べた本記述の改善点

- 多数の命令文を対象とした記述が容易になる  
モデル検査器が計算するのは特定の番号の命令文ではなく、命令文の集合である.
- 最適化器の効率を向上できた

**delete v := e**

**if point\_delete : AX ((AG¬use(v)) ∨ A¬use(v) U def(v))**

既存研究:

**n: v := e → skip**

**if n ⊢ AX ((AG¬use(v))) ∨ A¬use(v) U def(v))**

# 最適化の CTL<sub>b</sub> による定式化

- 最適化の条件をもとにした CTL<sub>b</sub> 式

無用命令除去:

$$AX ((AG \neg use(v)) \vee A \neg use(v) U def(v))$$

- データフロー方程式をもとにした CTL<sub>b</sub> 式

$$DEAD(B) = \{x \mid x \notin \bigcup_{S \in succ(B)} LIVE(S)\}$$

$$LIVE(B) = USE(B) \cup \left[ \left[ \bigcup_{S \in succ(B)} LIVE(S) \right] - KILL'(B) \right]$$

$$USE(B) = \dots$$

$$KILL(B) = \dots$$

## 部分冗長性除去のデータフロー方程式 (Paleriらの手法)

$$TRANSP_i = trans(e)$$

$$COMP_i = use(e) \cdot TRANSP_i(e)$$

$$ANTLOC_i = use(e) \cdot TRANSP_i(e)$$

$$AVIN_i = \begin{cases} false & \text{if entry} \\ \prod_{j \in preds(i)} AVOUT_j & \text{if otherwise} \end{cases}$$

$$AVOUT_i = COMP_i + AVIN_i \cdot TRANSP_i$$

$$ANTOUT_i = \begin{cases} false & \text{if exit} \\ \prod_{j \in preds(i)} ANTIN_j & \text{if otherwise} \end{cases}$$

$$ANTIN_i = ANTLOC_i + ANTOUT_i \cdot TRANSP_i$$

$$SAFEIN_i = AVIN_i + ANTIN_i$$

$$SAFEOUT_i = AVOUT_i + ANTOUT_i$$

$$SPAVIN_i = \begin{cases} false & \text{if } i = \text{entry or } \neg SAFEIN_i \\ \sum_{j \in preds(i)} SPAVOUT_j & \text{if otherwise} \end{cases}$$

$$SPAVOUT_i = \begin{cases} false & \text{if } i = \text{entry or } \neg SAFEOUT_i \\ COMP_i + SPAVIN_i \cdot TRANSP_i & \text{if otherwise} \end{cases}$$

$$SPANTOUT_i = \begin{cases} false & \text{if } i = \text{entry or } \neg SAFEOUT_i \\ \sum_{j \in preds(i)} SPANTIN_j & \text{if otherwise} \end{cases}$$

$$SPANTIN_i = \begin{cases} false & \text{if } i = \text{entry or } \neg SAFEIN_i \\ ANTLOC_i + SPANTOUT_i \cdot TRANSP_i & \text{if otherwise} \end{cases}$$

$$INSERT_i = COMP_i \cdot \neg SPAVIN_i \cdot SPANTOUT_i$$

$$INSERT_{i,j} = \neg SPAVOUT_i \cdot SPAVIN_j \cdot SPANTIN_j$$

$$REPLACE_i = ANTLOC_i \cdot SPAVIN_i + COMP_i \cdot SPANTOUT_i$$

## 部分冗長性除去のCTL記述 (Paleriらの手法)

### MATCH

$v := e$

### CONDITION

$\text{point\_comp} : \text{use}(e) \wedge \text{trans}(e)$

$\text{point\_avin} : AX (A \text{ trans}(e) \cup \text{use}(e))$

$\text{point\_avout} : \text{point\_comp} \vee (\text{point\_avin} \wedge \text{trans}(e))$

$\text{point\_antout} : AX(A \text{ trans}(e) \cup \text{use}(e))$

...

$\text{point\_edge2} : \neg \text{point\_spavout}$

$\text{edge\_split} : \text{point\_edge1} \rightarrow \text{point\_edge2}$

### PROCESS

$\text{point\_insert} : \text{InsertBefore temp} := e$

$\text{edge\_split} : \text{EdgeSplit temp} := e$

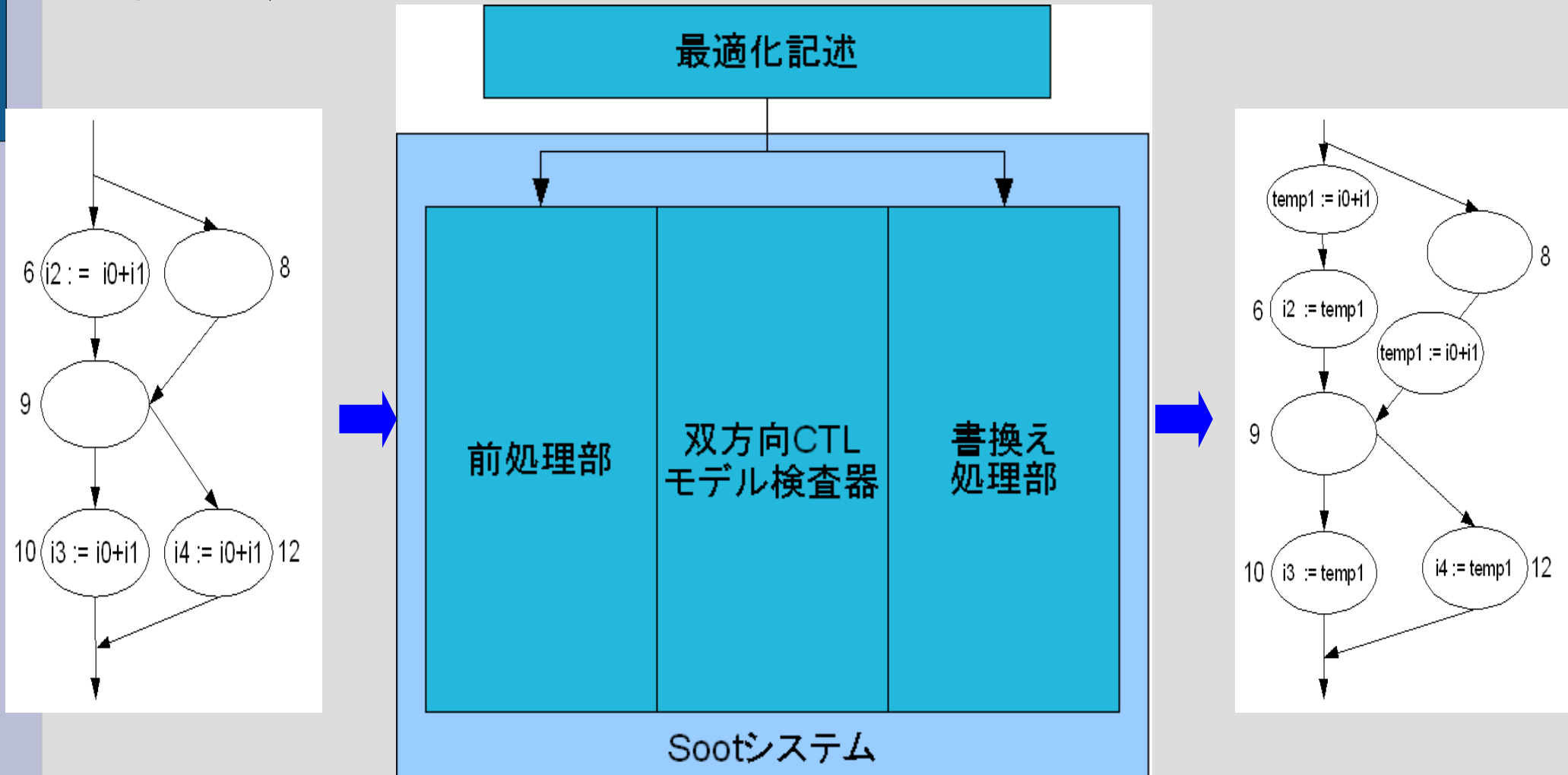
$\text{point\_replace} : \text{replace } e \rightarrow \text{temp}$

# CTL<sub>b</sub> による最適化器

- 3部分から構成:
  - 前処理部: 入力プログラムを必要な形に直す自由変数の束縛.
  - モデル検査器: モデル検査を行う.
  - 書換え部: 最適化書換え規則を適用し, 出力.
- 本研究はSootというJava最適化フレームワークを利用した.  
新しい最適化処理は, Soot があらかじめ持っている最適化処理に加えて実行される.  
最適化処理は中間言語の3番地コードJimpleで行われる.

# 最適化処理の例

- 部分冗長性を除去した例.



# 計算量

$$N \doteq O(n_1^{n_2} \times n_1 \times n_3) = O(n_1^{(n_2+1)} \times n_3)$$

$n_1$ : プログラムの大きさ

$n_2$ : 自由変数の数

$n_3$ : CTL<sub>b</sub>構文木のノード数

# 実験

• SPECjvm98の中の7つと奥村のJavaコード使って, 実験を行った.

## 実験環境

– CPU: Celeron 2GHz

– Memory: 512MB

– Soot: version 2.2.0

– 計測のオプション:

–Xint -Xms128m -Xmx128m(JITとメモリの影響を取除く)

## 適用した最適化

部分冗長性除去(共通部分式除去とループ不変式), コピー伝播(定数伝播), 無用命令除去.

(比較対照のSootが適用した最適化: 共通部分式除去, 部分冗長性除去, コピー伝播, 定数伝播 & 畳込み, 条件分岐の畳込み, 無用命令除去, 無用分岐除去, 無用変数除去)

# 最適化の処理時間

- 左図はSPECjvm98の最適化時間, 右図は奥村のコードの最適化時間(単位:秒)

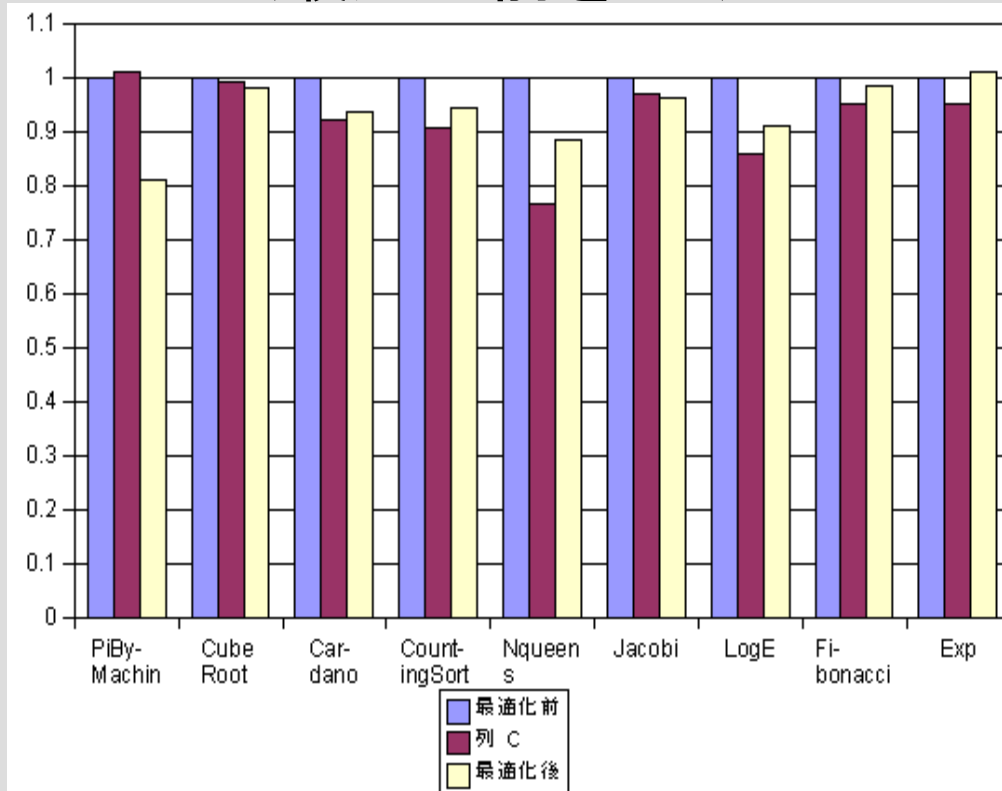
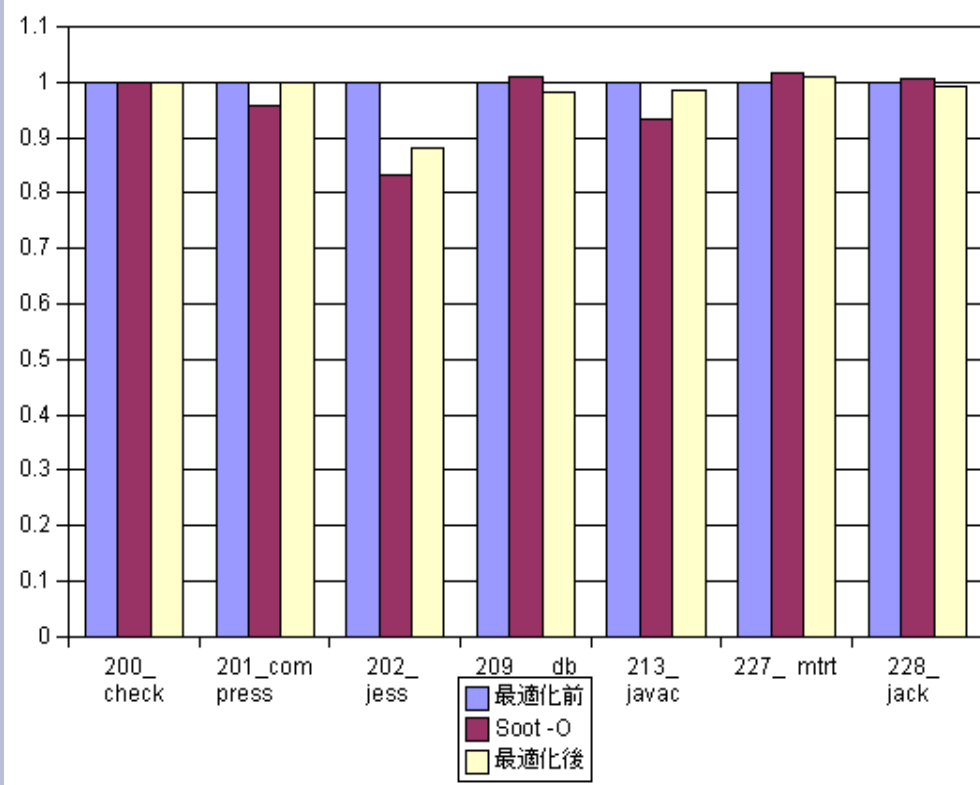
test code	行数	compile time	test code名	行数	compile time
200_check	5145	20	PiByMachin	108	0.8
201_compress	5050	29	CubeRoot	224	1.5
202_jess	26674	123	Cardano	242	7.1
209_db	5316	15	CountingSort	175	2.5
213_javac	57937	219	Nqueens	247	3.7
227_mtrt	9713	160	Jacobi	1061	48.6
228_jack	21895	316	LogE	1496	20.4
			Fibonacci	206	1.4
			Exp	1045	12.1

- データの分析:

時相論理による最適化は全探索などの処理のため, 実用的な時間内でできないと言われたが, 本研究の結果は遅くても5分以内に収まった.

# 最適化前後の実行時間の比較

- 左図はSPECjvm98, 右図は奥村のコード(最適化前を1に).



- データの分析:

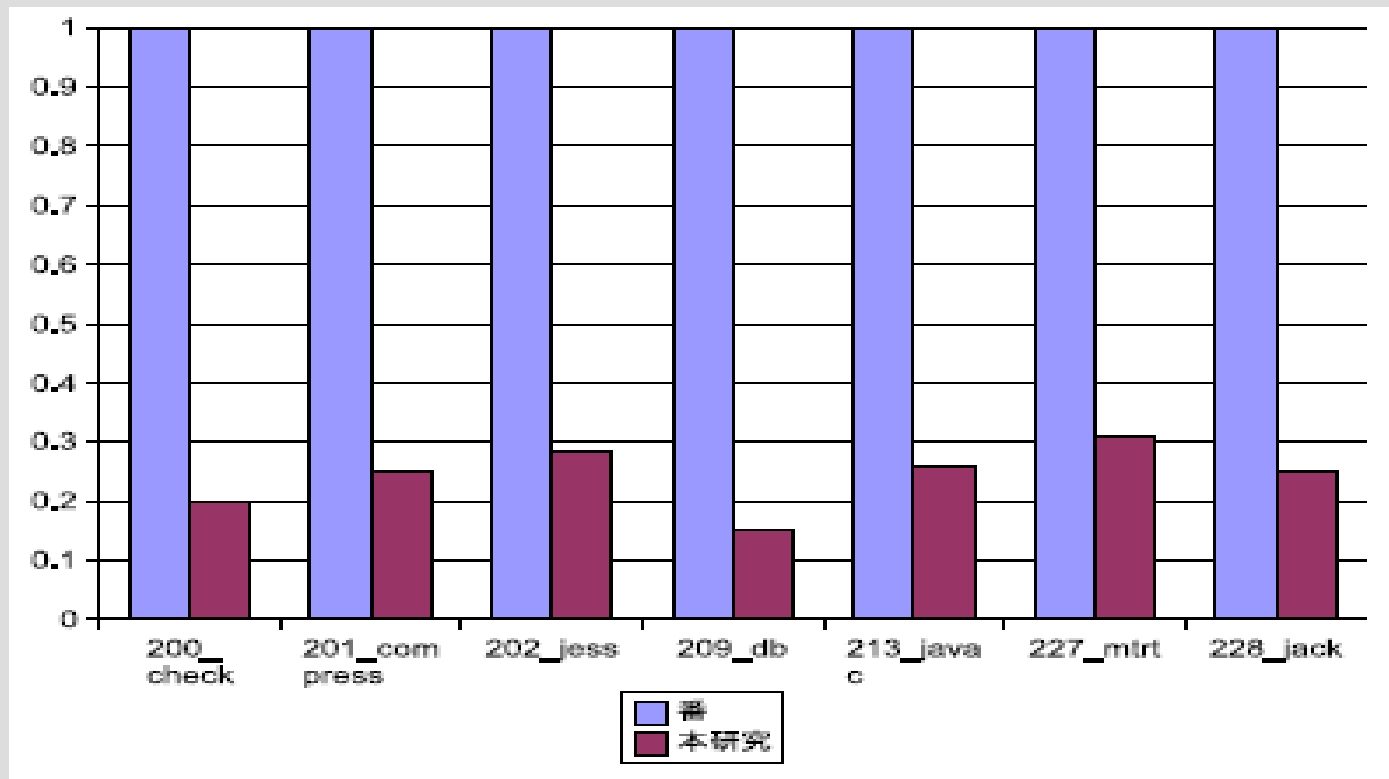
- 本研究の最適化はSootの最適化の一部だが, それなりの効果がある
- 本研究の最適化はSootの最適化の一部だが, Sootに勝ったものもある

# 従来手法との比較(最適化について)

- Laceyらの手法との比較

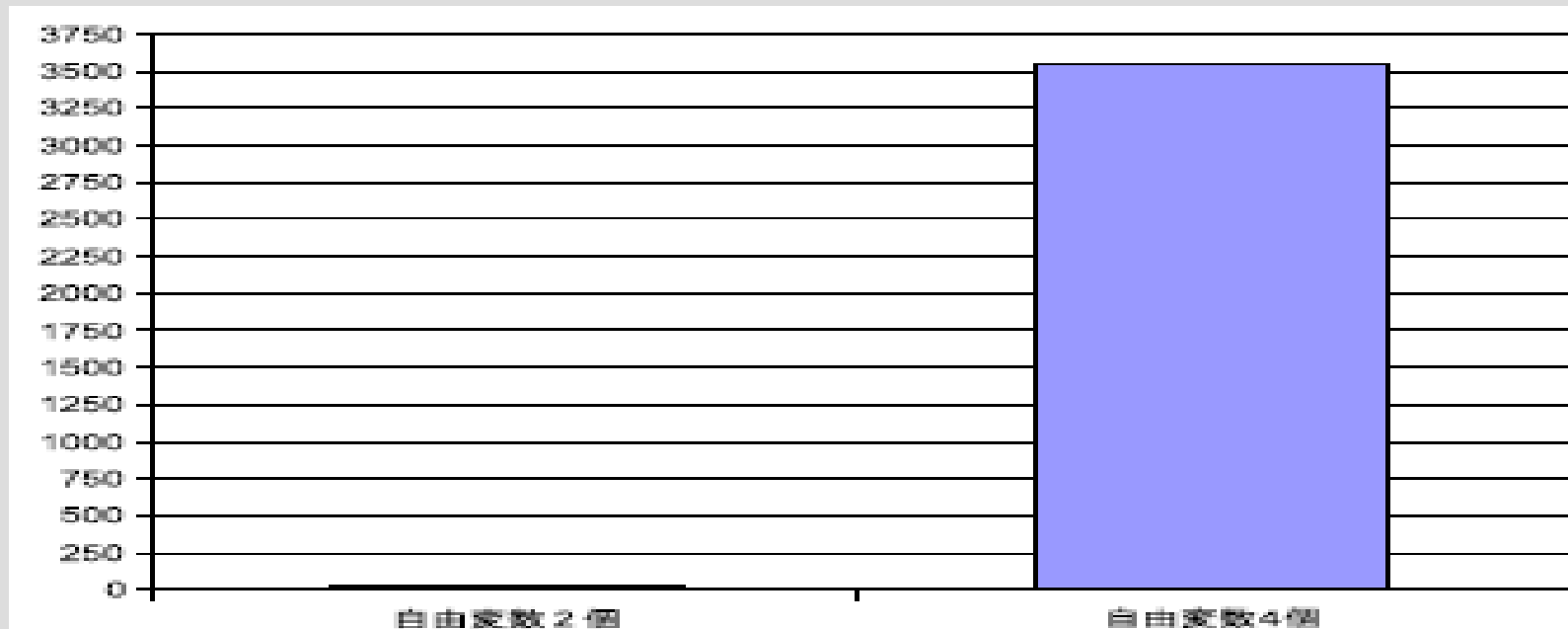
Laceyらは最適化時間や実行時間のデータを与えていない.

- 番らの手法との比較(番らの手法による最適化を1に)  
コピー伝播を適用した最適化時間を比較した. 本研究の最適化時間は15%から30%になっている.



# 自由変数による爆発

- 図の左は自由変数2つ, 最適化時間が**20秒**, 図の右は自由変数4つ, 最適化時間が**59分**である.  
MATCHの段階で2つを一對一に束縛した.
- 本研究は実用的な最適化器は自由変数を使わないように記述すべきことを明らかにした.



# 考察

- CTLの表現力

CTLによる最適化は簡単に数行で最適化記述を書け、簡潔だが、細かい処理をCTL式で書こうとすると、式が長くなるし、式の正しさの証明も難しい。

表現しにくい例：

- プロフィールを用いた部分冗長性除去
- もとの命令が無用命令除去により消されるときはコピー伝播を行わない
- 条件付定数伝播

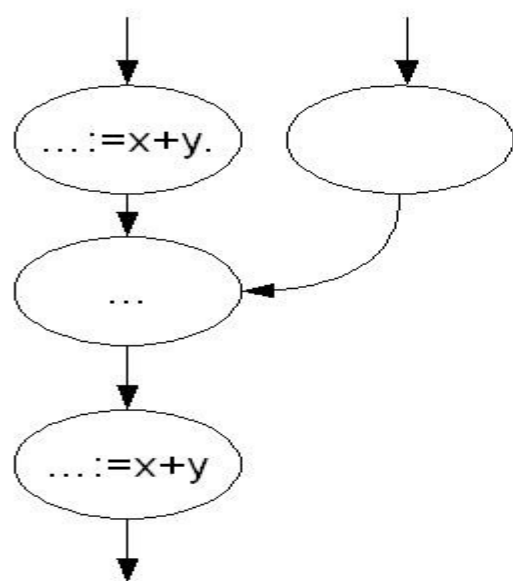
- 最適化器の効率

- 自由変数の全組み合わせのため、遅い
- モデル検査器は全探索のため、遅い

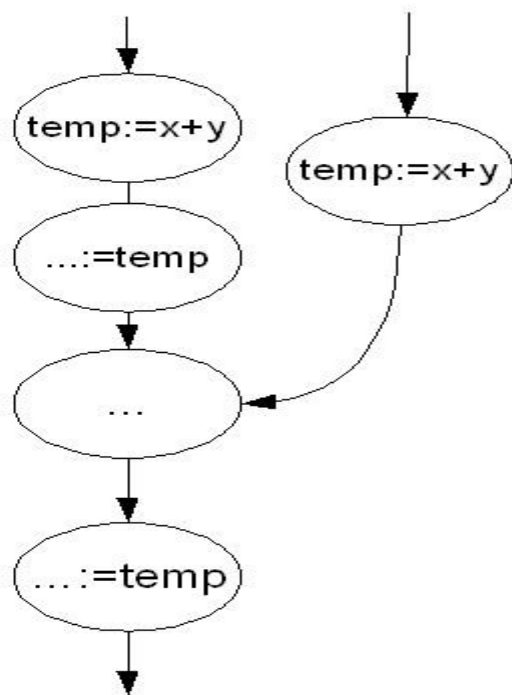
★ 最も影響するのは自由変数の数

## 最適化器の効果

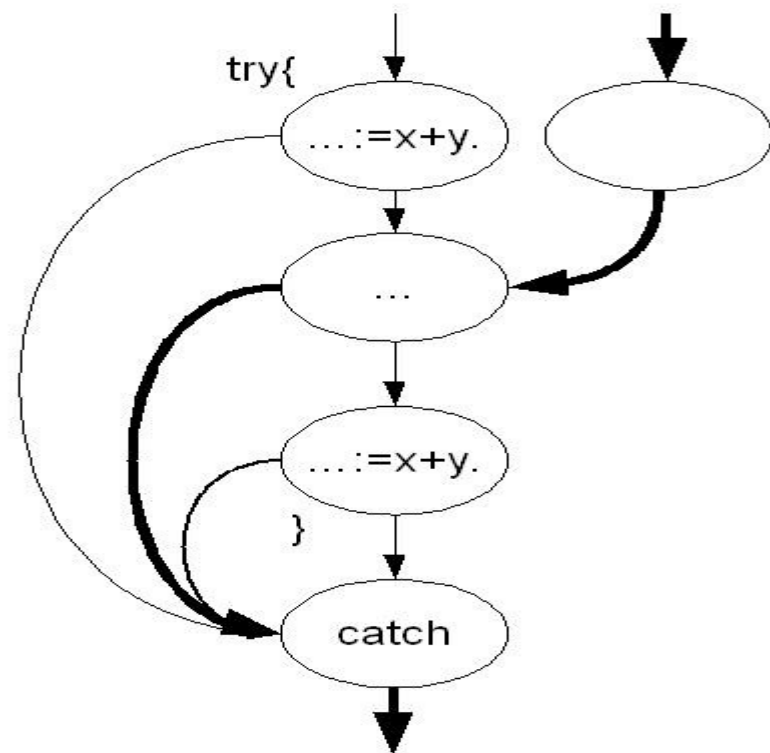
- Javaの例外を起し得る演算と例外を超える移動は対象外
- 記述できない最適化は対象外



BriefGraph  
(部分冗長除去する前)



BriefGraph  
(部分冗長除去した後)



CompleteGraph  
(部分冗長除去した後)

# 今後の課題

- 最適化時間の短縮
  - モデル検査器を早くするために, BDDを導入したり, 部分評価などの手法で, モデル検査器の全探索を避けることが考えられる.
  - ★ 自由変数は計算時間を爆発させるため, なくすべき
- 最適化の効果の向上
  - ★ 例外による最適化の阻害を克服
    - ループ, for文, goto文など細かい解析.
    - 条件付定数伝播など複雑な最適化をどう書くか.

# 本研究の位置づけ

- CTLによる最適化器の性能は、記述能力、最適化時間、最適化効果の三つの基準で評価できる。

	記述能力	最適化効率	最適化効果	特徴
Laceyら	○	×	△	最初提案、証明したが、実際の最適化の一部しか扱えない、記述は通常アルゴリズムに頼る。
山岡ら	×	×	×	過去演算子がなく、無用命令除去しか扱えない。
番ら	×	×	×	過去演算子を変換して未来演算子のみにするが、除去後の式が長い。無用命令文とコピー伝播しか扱えない。
本研究	○	○	△	モデル検査器が速い、記述しやすい。実用的な時間内で処理可能。

# まとめ

- CTL<sub>b</sub> モデル検査器と改善した最適化記述を用いるJava最適化器を開発し、複雑な最適化を実現した。通常アルゴリズムによる最適化器の性能に大幅に近づいた。
- 最適化の時間や最適化の効果について、ベンチマークやテストコードを使ってデータを取った。CTLによる最適化の問題点をいくつか明らかにすることができた。
- 今後は、問題点の解決、性能の改善をし、CTL式に基づく実用的なJava最適化器を伝統的な最適化器の一部として組み込めるよう努めたい。

おわり