

Array SSA とそれを用いた最適化の実装と評価

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

米倉 翔一
(06M37303)

平成 19 年度 修士論文

指導教官 佐々 政孝 教授

平成 20 年 1 月 25 日

概要

静的単一代入 (SSA) 形式とは、変数の定義がプログラムの字面上で唯一となるようにしたプログラムの中間表現の形式である。SSA 形式では、定義と使用の関係が明確になるため、コンパイラにおけるデータフロー解析や最適化を見通しよく行うのに適している。

SSA 形式を配列に適用できるように拡張したものを Array SSA という。Array SSA の過去の手法としては Knobe らの手法がある。この手法は、コンパイル時に十分な解析が出来ない場合、実行時に解析が引き継がれるため、実行時に大きなオーバーヘッドがかかってしまう。そのほかの手法も保守的で効果が制限されてしまうなど実用的なものではなかった。

最近、Array SSA の実用的な手法が Rus らによって提案された。この手法は配列要素の領域ごとに、定義された時の SSA 名を求めるというもので、複雑な参照パターンもコンパイル時に解析できるなど実用的なものであった。しかし、Rus らの手法は Fortran を対象としており、C 言語のプログラムに対してそのまま適用することは出来ない。また Array SSA の実装はまだあまり行われておらず、そのため、その効果も筆者の知る限りほとんど評価されていない。

本研究では Rus らの手法をもとに COINS コンパイラ上で Array SSA の実装を行った。その際、Rus らの手法にいくつかの変更を行い、C 言語のプログラムに対しても Array SSA を適用できるようにした。また、Array SSA の効果を評価するために、4 つの最適化を実装し、その効果を計った。

C 言語に Array SSA を適用する場合、以下のような問題がある。

1 つ目の問題点は別名の問題である。配列が別名参照されていると、別名によって非明示的に配列要素が書き換えられてしまうことがある。本研究では、別名参照されている可能性がある配列に対しては Array SSA を適用しないこととした。

2 つ目の問題として、C 言語では複雑なループや、goto 文のような非構造的な文がある。これらがあると、配列要素の定義位置を正確に求めることが出来ない。本研究ではこのような場合、解析の範囲を制限することにより、安全に解析が行われるようにした。

本手法での最適化の効果を実験した結果、数%~10%程度の実行時間の改善が見られた。コンパイル時間も実用的な時間内に収まり、Array SSA が効果的な手法であることが確認できた。

目次

| | | |
|--------------|---------------------------------|-----------|
| 第 1 章 | はじめに | 8 |
| 1.1 | 背景 | 8 |
| 1.2 | 本研究の概要 | 8 |
| 1.3 | 構成 | 9 |
| 第 2 章 | 準備 | 10 |
| 2.1 | 基本ブロック | 10 |
| 2.2 | 制御フローグラフ | 11 |
| 2.3 | 支配関係 | 12 |
| 2.4 | 支配木 | 13 |
| 2.5 | 支配境界 | 13 |
| 2.6 | ループ | 15 |
| 2.6.1 | ループ制御変数 | 15 |
| 2.7 | 抽象構文木 | 15 |
| 第 3 章 | 静的単一代入 (SSA) 形式 | 18 |
| 3.1 | SSA 形式 | 18 |
| 3.2 | SSA 変換 | 18 |
| 3.3 | SSA 形式に基づく最適化 | 20 |
| 3.3.1 | コピー伝播 | 20 |
| 3.3.2 | SSA 形式を用いたコピー伝播 | 20 |
| 3.4 | SSA 逆変換 | 20 |
| 3.5 | SSA 形式の性質 | 22 |
| 第 4 章 | Array SSA | 23 |
| 4.1 | Array SSA とは | 23 |
| 4.2 | 過去の研究 | 24 |
| 4.3 | Rus らの手法 (Region Array SSA) | 24 |
| 4.3.1 | 概要 | 24 |
| 4.3.2 | USR (Uniform Set of References) | 25 |
| 4.3.3 | Array SSA の構成 | 26 |
| 4.3.4 | 定義の探索 | 31 |

| | | |
|--------------|-------------------------------------|-----------|
| 第 5 章 | Array SSA の C 言語への適用 | 35 |
| 5.1 | Array SSA を C 言語へ適用する場合の問題点 | 35 |
| 5.2 | ポインタ | 35 |
| 5.3 | 複雑な制御構造 | 36 |
| 5.3.1 | ループ | 36 |
| 5.3.2 | switch 文 | 39 |
| 5.3.3 | goto 文 | 41 |
| 第 6 章 | 並列化コンパイラ向け共通インフラストラクチャ COINS | 43 |
| 6.1 | 概要 | 43 |
| 6.2 | 構成 | 43 |
| 6.3 | HIR と LIR の選択 | 44 |
| 6.4 | 高水準中間言語 HIR について | 44 |
| 6.4.1 | 概要 | 44 |
| 6.4.2 | 具体表現 | 45 |
| 第 7 章 | 実装 | 47 |
| 7.1 | Rus らの実装との違い | 47 |
| 7.1.1 | グローバル配列 | 47 |
| 7.1.2 | 手続き間解析 | 47 |
| 7.2 | 本手法の適用について | 47 |
| 7.2.1 | ループの正規化 (loop normalization) | 48 |
| 7.2.2 | スカラー変数の SSA 形式への変換 | 49 |
| 7.2.3 | Array SSA 形式への変換 | 49 |
| 7.2.4 | 各種最適化 | 49 |
| 7.2.5 | SSA, Array SSA から通常形式への逆変換 | 56 |
| 第 8 章 | 実験と考察 | 57 |
| 8.1 | 実験 | 57 |
| 8.1.1 | 実験の環境 | 57 |
| 8.1.2 | 実行時間 | 58 |
| 8.1.3 | コンパイル時間 | 59 |
| 8.2 | 今後の課題 | 60 |
| 8.2.1 | 新たな最適化の追加 | 60 |
| 8.2.2 | ループ構造の変換 | 60 |
| 8.2.3 | 手続き間の解析 | 61 |
| 8.2.4 | ポインタ解析とグローバル配列への Array SSA の適用 | 61 |
| 8.2.5 | LIR 上の最適化との組み合わせ | 62 |
| 第 9 章 | 関連研究 | 63 |
| 9.1 | Array SSA | 63 |
| 9.1.1 | Fink らの研究 | 63 |

| | | |
|---------------------------------|------------------------------|-----------|
| 9.2 | スカラー最適化の配列への拡張 | 63 |
| 9.2.1 | Vanbroekhoven らの研究 | 64 |
| 9.2.2 | Wonnacott らの研究 | 64 |
| 第 10 章 まとめ | | 65 |
| 付録 A Array SSA 形式への 変換アルゴリズム | | 70 |
| A.1 | ϕ 関数について | 70 |
| A.2 | ϕ 関数の挿入アルゴリズム | 71 |
| A.3 | 変数の名前替え | 72 |
| 付録 B HIR に SSA 形式を導入する際の問題点 | | 80 |

目次

| | | |
|-----|---------------------------------------|----|
| 2.1 | 基本ブロック | 10 |
| 2.2 | 制御フローグラフ (CFG) | 11 |
| 2.3 | 支配木 | 13 |
| 2.4 | X の支配境界は Y であり Z の支配境界も Y の例 | 14 |
| 2.5 | Z の支配境界は Y であり X の支配境界は Y ではない例 | 14 |
| 2.6 | 式 2.1 に対する解析木 | 17 |
| 2.7 | 式 2.1 に対する抽象構文木 | 17 |
| 4.1 | USR の定義 | 25 |
| 4.2 | definition | 26 |
| 4.3 | use | 26 |
| 4.4 | 定義探索アルゴリズム | 33 |
| 5.1 | プログラム 5.7 の Array SSA 形式 | 40 |
| 5.2 | goto 文を使ったプログラムの例 | 41 |
| 5.3 | 図 5.2 の Array SSA 形式 | 41 |
| 6.1 | COINS 構成図 | 44 |
| 6.2 | プログラム 6.1 の HIR | 46 |
| 7.1 | Array SSA 形式でのプログラム 7.2 の制御フローグラフ | 51 |
| 8.1 | テストプログラムの実行時間 | 58 |
| 8.2 | コンパイル時間 (no opt のコンパイル時間を 1 とした時の相対値) | 59 |
| 8.3 | コンパイル時間 (scalar のコンパイル時間を 1 とした時の相対値) | 60 |
| A.1 | 配列に対する ϕ 関数の挿入アルゴリズム | 73 |
| A.2 | ループでの配列に対する ϕ 関数の挿入のアルゴリズム | 74 |
| A.3 | 変数の名前替えのアルゴリズム 1 | 75 |
| A.4 | 変数の名前替えのアルゴリズム 2 | 76 |
| A.5 | 変数の名前替えのアルゴリズム 3 | 77 |
| A.6 | 変数の名前替えのアルゴリズム 4 | 78 |
| A.7 | 変数の名前替えのアルゴリズム 5 | 79 |
| B.1 | プログラム B.1 の SSA 形式 | 81 |
| B.2 | 図 B.1 の lab2 の HIR(通常形式の場合) | 81 |
| B.3 | 図 B.1 の lab2 の HIR(SSA 形式の場合) | 81 |

| | | |
|-----|--------------------|----|
| B.4 | 危険辺 | 82 |
| B.5 | 危険辺除去 | 82 |
| B.6 | サンプルプログラム | 83 |
| B.7 | 図 B.6 に SSA 逆変換を適用 | 83 |

表 目 次

| | | |
|-----|--------------------------------|----|
| 2.1 | 簡単な文法の例 | 16 |
| 4.1 | ϕ 関数の名前 | 27 |
| 5.1 | ループの条件 | 39 |
| 8.1 | PRIMEPOWER 250 の主な仕様 | 57 |

第1章 はじめに

1.1 背景

静的単一代入 (SSA) 形式とは、変数の定義がプログラムの字面上で唯一となるようにしたプログラムの中間表現の形式である [2, 3, 11, 18]. SSA 形式では定義と使用の関係が明確になるため、コンパイラにおけるデータフロー解析や最適化を見通しよく行うのに適している.

SSA 形式を配列に適用できるように拡張したものを Array SSA といい、その表現形式を Array SSA 形式という. 配列の定義は、スカラー変数の定義と違い、配列全体を書き換えるとは限らないため、配列に対して SSA を適用することは容易ではない.

SSA 形式の配列への素朴な拡張手法は配列のインデックスを無視し、配列全体を 1 つのスカラー変数のように扱うというものである. この方法では、配列の定義によってすべての配列要素が書き換えられると判断する. そのため、非常に保守的で、効果が非常に制限されていた.

初めて配列の個々の要素を扱った Array SSA の手法は、Knobe らにより提案されたものである [17, 24]. この手法は大まかにいうと、すべてのプログラム点で、配列の要素それぞれの定義位置を計算するというものである. この手法はコンパイル時に要素の定義位置の解析が完了しない場合、解析が実行時に引き継がれる. そのため実行時のオーバーヘッドが大きくなり、あまり実用的ではなかった.

最近、実用的な Array SSA の手法として、Rus らによって Region Array SSA という手法が提案された [23]. Rus らの手法では、個々の配列要素ではなく、配列の要素の領域 (Region) ごとに定義されたときの SSA 名を求める. 配列要素の領域の表現には USR という抽象構文木を用いる. この手法は、複雑なパターンでもコンパイル時に解析を行うことが出来る.

Array SSA の実装はまだあまり行われておらず、そのためその効果もあまり評価されていない. また、Rus らの手法は Fortran を対象としており、ポインタや goto 文などを考慮していないため、そのままでは C 言語のプログラムに適用することが出来ない.

1.2 本研究の概要

本研究では、Rus らの手法による Array SSA の実装を行った. その際、Rus らの手法にいくつかの変更を行い、C 言語のプログラムに対しても Array SSA を適用できるようにした. また、Array SSA を用いた最適化の効果を評価するため、以下の 4 つ

の最適化を実装し、その効果を計った。

- 定数伝播
- コピー伝播
- ループ不変式移動
- 配列参照の一時変数化

1.3 構成

本論文の構成を以下に示す。

- 2章: 準備 (本論文で用いる用語の説明)
- 3章: 静的単一代入 (SSA) 形式の説明
- 4章: Array SSA の説明
- 5章: Array SSA の C 言語への適用
- 6章: 並列化コンパイラ向け共通インフラストラクチャCOINSの説明
- 7章: 実装
- 8章: 実験と考察
- 9章: 関連研究
- 10章: まとめ

第2章 準備

本論文の説明にあたって予備知識が必要となる。本章ではフローグラフ等の予備知識の説明を行う。

2.1 基本ブロック

基本ブロック (Basic Block) あるいはブロックとはプログラムの一部分であり、そのブロックの実行を開始する文が、そのブロックの先頭の文だけであり、末尾が無条件飛び越し、あるいは条件つき飛び越しであるものである。基本ブロック途中への飛び込みや、基本ブロック途中からの飛び出しはない。基本ブロックと、そのブロック間の関係を示した制御フローグラフ (後述) は、プログラムを解析する基本的なグラフとして現在広く用いられている [1, 18, 3, 26]。図 2.1 の (a) のような C プログラムを基本ブロックに分割したものが (b) である。図中の矩形は基本ブロックを表しており、矩形の左上の L1 等は基本ブロックにつけた便宜的なラベルである。

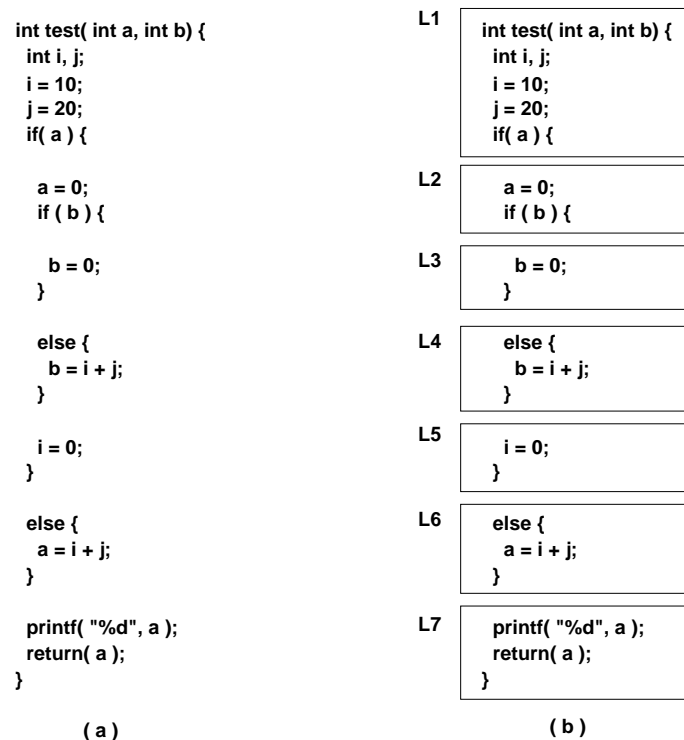


図 2.1: 基本ブロック

2.2 制御フローグラフ

制御フローグラフ (Control Flow Graph : CFG) とは, プログラムの入り口から出口までの基本ブロックを, 制御の流れに沿って図にしたグラフである [1, 18, 3]. CFG の節点 (ノード) は基本ブロックであり, ノード間是有向辺 (エッジ) で結ばれる. コンパイラでの最適化の多くは CFG の情報を用いている.

図 2.2 に, 図 2.1 に対応する CFG を示す. 図中の矩形は基本ブロックであり矢印はエッジを表している. また, 矩形の中にある L1 等は基本ブロックに対して付けた便宜的なラベルである. 各ノードのラベルは, 図 2.1 のそれと対応している. なお, 定式化によっては, これ以外に入り口ノードと出口ノードを別途付加することもあるが, ここでは立ち入らない.

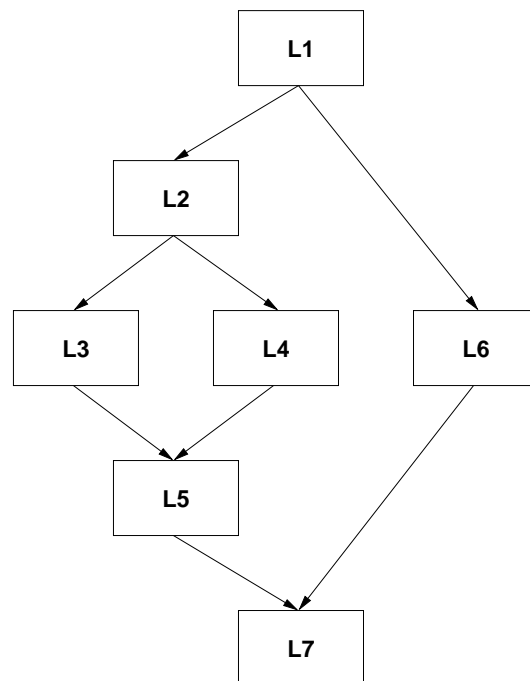


図 2.2: 制御フローグラフ (CFG)

基本ブロック X と Y が存在し, X から Y への有向辺がある場合, X は Y の先行ノード (predecessor) といい, Y は X の後続ノード (successor) と言う. 以下では X の predecessor を $pred(X)$ とし, X の successor を $succ(X)$ とする. たとえば図 2.2 の各ノードの $pred()$ と $succ()$ は以下ようになる.

- L1 : $pred(L1) = \{\}, succ(L1) = \{L2, L6\}$
- L2 : $pred(L2) = \{L1\}, succ(L2) = \{L3, L4\}$
- L3 : $pred(L3) = \{L2\}, succ(L3) = \{L5\}$
- L4 : $pred(L4) = \{L2\}, succ(L4) = \{L5\}$

- $L5 : pred(L5) = \{L3, L4\}, succ(L5) = \{L7\}$
- $L6 : pred(L6) = \{L1\}, succ(L6) = \{L7\}$
- $L7 : pred(L7) = \{L5, L6\}, succ(L7) = \{\}$

2.3 支配関係

CFG 上のふたつのノード X, Y について, CFG の入り口から Y に達するどの路も必ず X を通る場合に, X は Y を支配 (dominate) するという [1, 18, 3]. また, X は Y の支配ノード (dominator) であるという. 以下では X の支配ノードの集合を $dom(X)$ と書く. 支配関係は反射的 (reflexive) である. すなわち, ノード X は自分自身を支配する. また, 支配関係は推移的 (transitive) である. すなわち, ノード X がノード Y を支配し, ノード Y がノード Z を支配するのであれば, ノード X はノード Z を支配する [18]. 例えば図 2.2 の CFG では以下のような支配関係がある.

$$\begin{aligned}
 dom(L1) &= \{L1\} \\
 dom(L2) &= \{L1, L2\} \\
 dom(L3) &= \{L1, L2, L3\} \\
 dom(L4) &= \{L1, L2, L4\} \\
 dom(L5) &= \{L1, L2, L5\} \\
 dom(L6) &= \{L1, L6\} \\
 dom(L7) &= \{L1, L7\}
 \end{aligned}$$

ノード X がノード Y を支配し, $X \neq Y$ である場合, X は Y を厳密に支配 (strictly dominate) するという [1, 18, 3]. 例えば図 2.2 の $L1$ は $L3$ を厳密に支配している. 以下, X を厳密に支配するノードの集合を $sdom(X)$ と書く.

ノード X がノード Y を厳密に支配し, X から Y への路に, X 以外に Y を厳密に支配するノードがないとき, X は Y を直接支配 (immediately dominate) するという [1, 18, 3]. 以下では X が Y を直接支配するときに $X = idom(Y)$ と書く. 例えば図 2.2 の CFG では以下のような直接支配の関係がある.

$$\begin{aligned}
\text{idom}(L1) &= \{\} \\
\text{idom}(L2) &= \{L1\} \\
\text{idom}(L3) &= \{L2\} \\
\text{idom}(L4) &= \{L2\} \\
\text{idom}(L5) &= \{L2\} \\
\text{idom}(L6) &= \{L1\} \\
\text{idom}(L7) &= \{L1\}
\end{aligned}$$

2.4 支配木

支配木 (dominator tree) とは, CFG 上のあるノード X と, X を直接支配するノード Y を有向辺でむすんだグラフである. 得られたグラフは木構造となる. なぜなら, CFG のあるノード Z を直接支配するノードはひとつだからである. 支配木の根は CFG の入口ノードである. 以下, 支配木のノード X の子ノード Y を $Y \in \text{domChild}(X)$ と書く. 図 2.2 の CFG に対応した支配木を図 2.3 に示す.

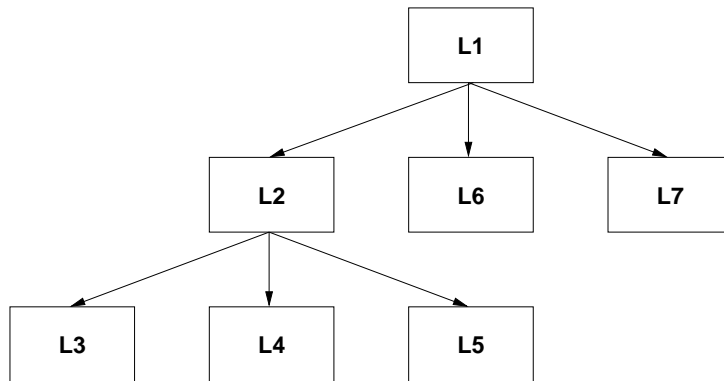


図 2.3: 支配木

2.5 支配境界

支配境界とは, CFG 上で, あるノード X からエッジを辿ってゆき, はじめて X の支配から外れたノード Y の集合である. ノード X の支配境界 $DF(X)$ は以下のように定義される [18].

$$DF(X) = \{Y \mid U \in \text{pred}(Y) \text{ が存在し, } X \text{ は } U \text{ を支配し, } X \text{ は } Y \text{ を厳密に支配はしない}\}$$

この定義の X と U は同じノードの場合もある. X と U が同じノードと仮定すると, $Y \in \text{succ}(X)$ であり, かつ X は Y に対して厳密な支配をしないものである. また

$X \neq U$ である場合には、定義から $X \in dom(U)$ である。ここで X が厳密な支配をするノード Z があると仮定して $DF(X)$ と $DF(Z)$ について考えてみる。 $Y \in DF(X)$ であり、かつ X から Y に至る際に必ず U を通る場合は図 2.4 のような関係になる。この場合は $Y \in DF(Z)$ であることは明らかである。

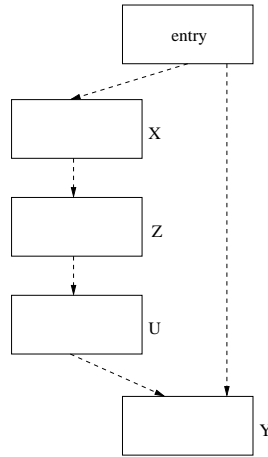


図 2.4: X の支配境界は Y であり Z の支配境界も Y の例

逆に、 $Y \in DF(Z)$ であるが $Y \notin DF(X)$ の場合の各々のノードの関係は図 2.5 のようになる。 $Y \in DF(Z)$ であるから、入口ノードから Z を通らずに Y に到達することはできる。しかし $Y \notin DF(X)$ であるため、入口ノードから Y に到達するには必ず X を通ることになる。別の言葉で言えば、 $X \in dom(Y)$ となり、かつ $X \in sdom(Y)$ となる。

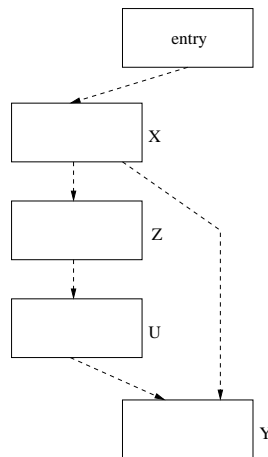


図 2.5: Z の支配境界は Y であり X の支配境界は Y ではない例

以上のことから、 X の支配境界は以下の Y の集合といえる。

- $Y \in succ(X)$ であり、 $X \neq idom(Y)$

- $Z \in \text{dom}(X)$ であるときに $Y \in \text{DF}(Z)$ であり, $X \neq \text{idom}(Y)$

例えば図 2.2 の CFG の各ノードに対する支配境界は以下ようになる.

$$\begin{aligned} \text{DF}(L1) &= \{\} \\ \text{DF}(L2) &= \{L7\} \\ \text{DF}(L3) &= \{L5\} \\ \text{DF}(L4) &= \{L5\} \\ \text{DF}(L5) &= \{L7\} \\ \text{DF}(L6) &= \{L7\} \\ \text{DF}(L7) &= \{\} \end{aligned}$$

2.6 ループ

2.6.1 ループ制御変数

ループ内での定義が 1 箇所、定義が $i = i + c$ または $i = i - c$ の形をとるような変数のことをループの帰納変数 (induction variable) とよぶ [1, 18, 3, 26]. ただし, c はループ不変である. 本論文では, さらに以下の 3 つの条件を加えたものを満たす変数 i をループ制御変数 (loop control variable) と呼ぶ.

- c の値が定数
- ループの繰返し条件の式に変数 i が含まれる
- 変数 i の初期値が一つに定まる

例えば, プログラム B.1 の変数 i は, 初期値が 1 で, ループ内で 1 ずつ値が変化していき, ループの繰返し条件式 $i < 100$ に含まれている. よって変数 i は, このループのループ制御変数である.

プログラム 2.1 (ループの例)

```
1: for ( $i = 1$ ;  $i < 100$ ;  $i = i + 1$ ){
2:    $a[i] = 1$ ;
3: }
```

2.7 抽象構文木

文が与えられたとき, 文法によってその文がどのように導かれるか, つまり生成規則をどのように適用すればその文が得られるかを, 木の形で示すと都合がよい. このような木を解析木 (parse tree) または導出木 (derivation tree) という [26].

$$\begin{aligned}
S &\rightarrow \mathbf{i} := E \\
E &\rightarrow E * E \\
E &\rightarrow E / E \\
E &\rightarrow E + E \\
E &\rightarrow E - E \\
E &\rightarrow (E) \\
E &\rightarrow \mathbf{i} \\
E &\rightarrow \mathbf{num}
\end{aligned}$$

表 2.1: 簡単な文法の例

例として, 式 2.1 に対する, 表 2.1 の文法から導かれる解析木を図 2.6 に示す.

$$i_1 := (i_2 + i_3) * i_4 / \mathbf{num} \quad (2.1)$$

解析木によって, 演算とそのオペランド (被演算子) がはっきり示される. 例えば, 図 2.6 では, i_2 と i_3 を加えたものに i_4 をかけ, それを \mathbf{num} で割ったものを i_1 へ代入することが読み取れる.

解析木は構文を正確に表しているが, 演算の意味を示すには内部の節がやや冗長である. 解析木のかわりに, 解析木のうち演算に本質的なものだけを取り出した木を用いることがある. これは, 節として演算子, 節の子供として節の演算子のオペランドがくるようにした木であり, 抽象構文木 (abstract syntax tree) あるいは単に構文木 (syntax tree) と呼ばれる. 抽象構文木では, 葉をつなげても, 一般に文にはならない. 式 2.1 に対する抽象構文木を図 2.7 に示す. 抽象構文木には非終端記号 (表 2.1 の S と E) は現れない. 演算に本質的でない構造, 例えば子供が一人だけの枝や, $()$ をまとめるためだけの構造は取り除かれる. その結果, 一般に抽象構文木のほうが解析木より簡潔である.

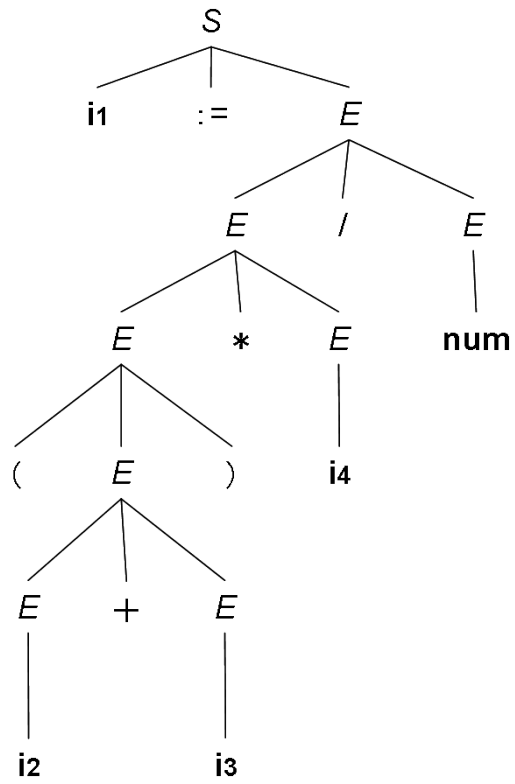


図 2.6: 式 2.1 に対する解析木

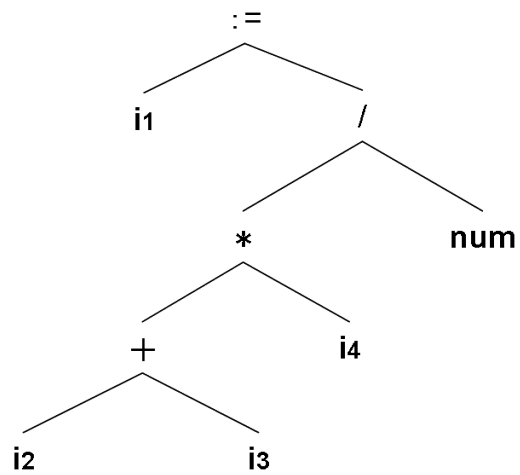


図 2.7: 式 2.1 に対する抽象構文木

第3章 静的単一代入 (SSA) 形式

本章では, 静的単一代入 (static single assignment, SSA) 形式について説明を行う.

3.1 SSA 形式

静的単一代入形式とは, 変数の定義がプログラムの字面上で唯一となるようにしたプログラムの表現形式である [18, 3, 11]. 静的とは, プログラムの字面上で, という意味である. 変数の定義が唯一になるように変数の名前替えを行うが, これは普通変数に添え字をつけてあらわす. この結果, SSA 形式では, プログラムあるいは中間表現の上で, 各変数の定義が 1 箇所だけになる. 以下に, SSA 形式の例をあげる.

プログラム 3.1 (SSA 形式の簡単な例)

```
1:  $a_1 = x_0 + y_0;$   
2:  $a_2 = a_1 + 3;$   
3:  $b_1 = x_0 + y_0;$ 
```

3.2 SSA 変換

SSA 形式に対して, 元のプログラムの形式を通常形式 (normal form) と呼ぶ. 通常形式から SSA 形式に変換することを SSA 変換 (SSA translation) という. 次のような通常形式のプログラムがあったとする.

プログラム 3.2 (通常形式の例)

```
1:  $a = x + y;$   
2:  $a = a + 3;$   
3:  $b = x + y;$ 
```

これを SSA 形式に変換すると次のようになる.

プログラム 3.3 (プログラム 3.2 の SSA 形式)

```
1:  $a_1 = x_0 + y_0;$   
2:  $a_2 = a_1 + 3;$   
3:  $b_1 = x_0 + y_0;$ 
```

たとえば a について見ると、代入があるごとに変数の新しい version を作り、 a_1, a_2 のように添え字を付けて区別する。

以下は制御の合流がある例である。次のような通常形式のプログラムがあったとする。

プログラム 3.4 (制御の合流がある例)

```
1:  $x = \dots$ ;  
2: if ( $x > 0$ ) {  
3:      $a = x$ ;  
4: } else {  
5:      $a = -x$ ;  
6: }  
7:  $print(a)$ ;
```

これを SSA 形式に変換すると次のようになる。

プログラム 3.5 (プログラム 3.4 の SSA 形式)

```
1:  $x_1 = \dots$ ;  
2: if ( $x_1 > 0$ ) {  
3:      $a_1 = x_1$ ;  
4: } else {  
5:      $a_2 = -x_1$ ;  
6: }  
7:  $a_3 = \phi(a_1, a_2)$ ;  
8:  $print(a_3)$ ;
```

文 7 が属するブロックでは、 a_1 と a_2 の値が合流するので、それ以降に a が使われていると、 a_1 か a_2 が決められなくなる。そこで、SSA 形式では ϕ 関数 (ϕ function) と呼ばれる仮想的な関数を導入する。文 7 「 $a_3 = \phi(a_1, a_2)$ 」により、これ以後使われる a は a_3 であることをあらわす。また、この ϕ 関数 $\phi(a_1, a_2)$ は、制御が文 3 「 $a_1 = x_1$ 」が属するブロックから来た時には a_1 の値を返し、文 5 「 $a_2 = -x_1$ 」が属するブロックから来た時には a_2 の値を返す関数をあらわす。

SSA 変換についての効率的なアルゴリズムは、代表的なものに

- Cytron らの方法 [11]
- Brandis らの方法 [5]
- Sreedhar らの方法 [28]
- Aycock らの方法 [4]

が知られている。本研究では Cytron らの方法を用いて SSA 変換を行った。

3.3 SSA 形式に基づく最適化

SSA 形式では、変数の使用に対する定義が1つだけなので、SSA 形式を用いると、コンパイラにおけるデータフロー解析やさまざまな最適化が見通しよく行える。SSA 形式を用いることで、最適化が容易になる例としてコピー伝播について説明する。

3.3.1 コピー伝播

次のようなプログラムを考える。

プログラム 3.6 (通常形式)

```
1:  $x = y$ ;  
2:  $u = 2 + x$ ;  
3:  $v = x + 1$ ;
```

$x = y$ のような複写をする代入文をコピー文 (copy statement) と呼ぶ。この時 x が再定義されるまでは、 x と y は同じ値を持つ。よってこのコピー文の後方に現れて再定義されるまで、あるいは x の別の定義が合流するまでの x は y に置き換えることが出来る。これをコピー伝播 (copy propagation) という。つまり、コピー文の後方に現れる x を y に置き換えて下のように変換できる。

プログラム 3.7 (コピー伝播)

```
1:  $x = y$ ;  
2:  $u = 2 + y$ ;  
3:  $v = y + 1$ ;
```

すると一番上のコピー文は不要なので削除することが出来る。

3.3.2 SSA 形式を用いたコピー伝播

通常、コピー伝播をするときは、コピー文に現れる変数の持つ値がどこで定義されたものかという情報を考える必要がある。

しかし、SSA 形式で表されたプログラムに現れる変数の定義は一箇所だけなので、定義される場所を考える必要がない。例えば、プログラム中に「 $a = b$ 」が現れるとする。このとき、「 $a = b$ 」を削除し、プログラム中に現れる a を b で置き換えることでプログラムの意味を換えずに「 $a = b$ 」を取り除くことが出来る。

3.4 SSA 逆変換

SSA 形式を通常形式に戻すことを SSA 逆変換 (SSA reverse translation) という。 ϕ 関数をそのまま実行できるアーキテクチャはないので、目的コードを生成する前に ϕ 関数の除去を行う必要がある。次に ϕ 関数がある場合の SSA 逆変換の例を示す。以下のような SSA 形式のプログラムがあったとする。

プログラム 3.8 (SSA 形式)

```
1:  $x_1 = \dots;$ 
2: if ( $x_1 > 0$ ) {
3:      $a_1 = x_1;$ 
4: } else {
5:      $a_2 = -x_1;$ 
6: }
7:  $a_3 = \phi(a_1, a_2);$ 
8:  $print(a_3);$ 
```

これを SSA 逆変換すると次のようになる。

プログラム 3.9 (プログラム 3.8 の SSA 逆変換の結果)

```
1:  $x_1 = \dots;$ 
2: if ( $x_1 > 0$ ) {
3:      $a_1 = x_1;$ 
4:      $a_3 = a_1;$ 
5: } else {
6:      $a_2 = -x_1;$ 
7:      $a_3 = a_2;$ 
8: }
9:  $print(a_3);$ 
```

前述したとおり、プログラム 3.8 の文 7 「 $a_3 = \phi(a_1, a_2)$ 」は、どちらの基本ブロックからこの合流点に来たかによって a_3 の値を決めるものであった。そこで、プログラム 3.9 では、通常形式でそれを復元するために、合流点に入ってくるそれぞれの基本ブロックの最後に文 4 「 $a_3 = a_1$ 」と文 7 「 $a_3 = a_2$ 」のコピー文を置き、 ϕ 関数を消去する。

ただし、このままでは無駄なコピー文があるので、合併 (coalescing) という手法を使って、それらを取り除くことが多い [6, 8, 14, 20]。プログラム 3.9 で a_1, a_2, a_3 を合併すると、元の通常形式であるプログラム 3.4 と同じものが得られる。

しかし SSA 逆変換では、上記の素朴な方法がうまくいかない例が知られている [7]。それらの問題点を解決した SSA 逆変換についての効率的なアルゴリズムは、代表的なものに、

- Briggs らの方法 [7]
- Sreedhar らの方法 [29]
- Rastello らの方法 [21]

が知られている。本研究では、Sreedhar らの方法で SSA 逆変換を行った。

3.5 SSA 形式の性質

性質 3.1 (SSA 形式) 元プログラムのすべての変数 v に対して, 次の 3 条件を満たすとき, そのプログラムは SSA 形式になっているという.

- もしブロック Z がパス $X \rightarrow Z$ と $Y \rightarrow Z$ (ブロック X とブロック Y は v の定義を含んでいる) の最初の結合ブロックならば, v に対する ϕ 関数が Z の先頭に挿入されている.
- v に対する新しい名前 v_i はプログラムテキスト上で唯一の定義を持つ.
- 任意の制御フローグラフのパスにおいて, v に対する新しい名前 v_i の使用と, 元プログラムでそれに対応する v の使用を考える. このとき v と v_i は同じ値を持つ.

第4章 Array SSA

本章では、本研究で実装を行うプログラムの表現形式である Array SSA について説明を行う。

4.1 Array SSA とは

Array SSA とは、3 章で説明したスカラー変数に対する SSA 形式を、配列に対して適用したものである。これを行えば配列の定義と使用の関係が明確になり、配列に対する最適化が見通しよく行えるようになる。しかし、スカラー変数に対する SSA 変換の手法をそのまま配列に適用することは出来ない。以下に配列に対する SSA 形式の適用の問題点の例をあげる。例えば次のような通常形式のプログラムがあったとする。

プログラム 4.1 (通常形式)

```
1: A[2] = 5;
2: A[5] = 3;
3: ... = A[2];
```

これをスカラー変数と同様の手法で SSA 形式に変換すると次のようになる。

プログラム 4.2 (プログラム 4.1 の SSA 形式)

```
1: A1[2] = 5;
2: A2[5] = 3;
3: ... = A2[2];
```

プログラム 4.1 では、文 3 で使用されている $A[2]$ は、文 1 で定義されている。プログラム 4.2 では、変数の名前替えにより、使用する要素が $A_2[2]$ になっている。文 1 で定義されているのは $A_1[2]$ なので、文 3 で使用する値が変わってしまう。よってこの変換は間違いである。この問題は、一般に配列に対する定義はスカラー変数と違って、その配列のすべての要素を書き換えるわけではないことから起こる。そのため配列に対して SSA 形式を適用するには、配列全体をスカラー変数のように扱うのではなく、個々の配列要素を扱わなければならない。

4.2 過去の研究

初期の手法

初期の SSA 形式の配列への拡張は、配列のインデックスを無視し、配列全体を 1 つのスカラ変数として扱うものである。この方法では、配列の定義ではすべての配列要素が書き換えられたと判定する。これは非常に保守的な方法であり、機能が非常に限定されている。

Knobe らの手法

配列の個々の要素を扱った Array SSA として Knobe らの手法 [17, 24] がある。Knobe らの手法では、制御フローの合流点だけでなく、配列の定義の後にも ϕ 関数を挿入する。 ϕ 関数では、個々の配列要素がどの SSA 名で定義されたかを計算する。また、 ϕ 関数の意味を表すために、すべての配列がそれぞれ $@array$ というものを持つ。 $@array$ は対応する配列の定義が最も最近行われた時間を表す。

Knobe らの手法での、配列 X に対する ϕ 関数 $X_3 = \phi(X_2, X_1)$ の意味は、式 4.1 のようになる。

$$\begin{aligned} X_3[j] = & \text{if } @X_2[j] \succeq @X_1[j] \text{ then } X_2[j] \\ & \text{else } X_1[j] \\ & \text{end if} \end{aligned} \tag{4.1}$$

$@X_2[j]$ は配列 X_2 の要素 j が定義された時間を表す。 \succeq は、 $@array$ の要素の値の辞書式順序の \geq 関係を表す。よって $X_3[j]$ の値は、 $@X_2[j]$ のほうが $@X_1[j]$ より辞書式に大きい（つまり後に定義されている）場合 $X_2[j]$ になり、そうでなければ $X_1[j]$ になる。 ϕ 関数がループ内にある場合は、繰り返しベクトル (iteration vector) を用いて、さらにループの繰り返しごとに定義を求めなければならない。

この計算をコンパイル時に完成させることは難しく、残りの計算は実行時に引き継がれる。そのため、この手法は実行時のオーバーヘッドが大きくあまり実用的ではない。

4.3 Rus らの手法 (Region Array SSA)

本節では、本研究で利用する Array SSA の手法である Rus らの Array SSA の手法 (Region Array SSA) について説明する [23]。

4.3.1 概要

Rus らの手法は、Knobe らの手法のように配列要素ごとに、どこで定義されたのかを求めるのではなく、配列要素の領域 (Region) ごとに定義の SSA 名を記憶する。配列要素の領域の表現には USR (Uniform Set of References) というメモリアクセスの

表現形式を利用する。Rus らの手法は goto 文のような非構造的文は扱えない。本研究では Array SSA の実装にこの Rus らの手法を利用する。

4.3.2 USR (Uniform Set of References)

Rus らの手法での、配列の参照要素の領域の表現に利用する USR について説明する [22].

USR は、プログラムの任意の階層レベルや制御フローにおいてのメモリ参照の集合を表現できる記号的なメモリ参照の集合の表現である。USR の最も単純な形は LMAD (Linear Memory Access Descriptor) [19] である。LMAD は添え字式が線形関数である配列参照によって、アクセスされるメモリ参照の集合を表現する。例として次のプログラムを考える (ここでの例は Fortran 風のプログラムを用いる)。

プログラム 4.3 (プログラムの例 1)

```

1 Do i = 1, 10
2   A[i] = 0
3 EndDo
4 Do i = 1, 5
5   ... = A[i]
6 EndDo

```

プログラム 4.3 の文 2 で定義される配列 A の領域は LMAD で [1 : 10] と表現でき、文 5 で使用される配列 A の領域は LMAD で [1 : 5] と表現できる。

$$\begin{aligned}
\Sigma &= \{\cap, \cup, -, (,), \#, \otimes^{\cup}, \otimes^{\cap}, \bowtie, \\
&\quad LMAD, Gate, Recurrence, CallSite\} \\
N &= \{USR\}, S = USR \\
P &= \{USR \rightarrow LMAD \mid (USR) \\
&\quad USR \rightarrow USR \cap USR \\
&\quad USR \rightarrow USR \cup USR \\
&\quad USR \rightarrow USR - USR \\
&\quad USR \rightarrow Gate \# USR \\
&\quad USR \rightarrow \otimes_{Recurrence}^{\cup} USR \\
&\quad USR \rightarrow \otimes_{Recurrence}^{\cap} USR \\
&\quad USR \rightarrow USR \bowtie CallSite\}
\end{aligned}$$

図 4.1: USR の定義

USR は図 4.1 で表される言語の抽象構文木として表現する. $\cap, \cup, -$ はそれぞれ積集合, 和集合, 差集合を表す. $Gate\#USR$ は $Gate$ で表される条件付きの参照を表す集合である. $\otimes_{Recurrence}^{\cup} USR, \otimes_{Recurrence}^{\cap}$ はそれぞれ $Recurrence$ で表される再帰式を繰り返した再帰的な和集合と積集合である. $USR \bowtie CallSite$ は $Call$ 文の呼び出し先での USR を表す. 線形な関数で表現されるメモリ参照は構文木の葉である LMAD で表され, LMAD で表せない非線形な参照パターンは内部ノードを加えることによって表現する.

以下に USR の例を示す. プログラム 4.4 の文 3 で定義される配列要素の USR は $\otimes_{i=1,10}^{\cup}\{(C[i] > 0)\#\{i\}\}$, 文 8 で使用される配列要素の USR は $\otimes_{i=1,5}^{\cup}\{(C[i] > 0)\#\{i\}\}$ となる. これらを木構造で表したものを図 4.2, 図 4.3 に示す. $\otimes_{i=1,10}^{\cup}\{(C[i] > 0)\#\{i\}\}$ は $i = 1, 10$ のうち, $(C[i] > 0)$ が成り立つ要素 i が定義されていることを表している.

プログラム 4.4 (プログラムの例 2)

```

1  Do  $i = 1, 10$ 
2    If  $(C[i] > 0)$ 
3       $A[i] = 0$ 
4    EndIf
5  EndDo
6  Do  $i = 1, 5$ 
7    If  $(C[i] > 0)$ 
8      ... =  $A[i]$ 
9    EndIf
10 EndDo

```

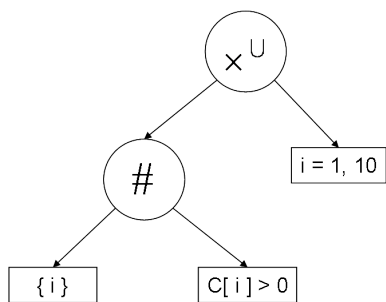


図 4.2: definition

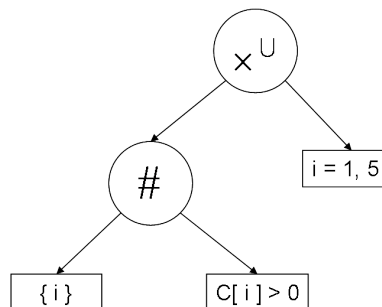


図 4.3: use

4.3.3 Array SSA の構成

本節では, Rus らの手法での Array SSA の構成法について述べる.

SSA 形式では, 制御フローの合流点に ϕ 関数を挿入する. Array SSA では制御フローの合流点だけでなく, 配列の定義の後にも ϕ 関数を挿入する. Array SSA の

| | |
|----------|----------------------------------|
| δ | 配列の定義の後に挿入する場合 |
| π | Then, Else のような制御依存の部分の先頭に挿入する場合 |
| γ | If ブロックの終わりの合流点に挿入する場合 |
| μ | ループに進入する辺とループの戻り辺の合流点に挿入する場合 |
| η | ループの出口の合流点に挿入する場合 |
| θ | 関数呼び出しの後に挿入する場合 |

表 4.1: ϕ 関数の名前

関数では、関数に配列要素の領域の情報を組み込む。以下では、USR を \mathcal{R} で表現する。

$$[A_n, \mathcal{R}_n] = \phi(A_{undef}, [A_1, \mathcal{R}_1^n], [A_2, \mathcal{R}_2^n], \dots, [A_m, \mathcal{R}_m^n]) \quad (4.2)$$

$$\text{where } \mathcal{R}_n = \bigcup_{k=1}^m \mathcal{R}_k^n \text{ and } \mathcal{R}_i^n \cap \mathcal{R}_j^n = \emptyset, \quad (4.3)$$

$$\forall 1 \leq i, j \leq m, i \neq j$$

式 4.2 は Rus らの Array SSA での ϕ 関数の一般形である。 $A_k (k = 1, \dots, m, n)$ は配列 A の SSA 名を表す。 \mathcal{R}_k^n は SSA 名 A_k で定義され、この ϕ 関数まで運ばれてきた配列要素の領域、つまり A_n までの間に上書きされていない要素の領域である。 A_{undef} は A_n が含まれるブロックで未定義の領域を表す変数である。 \mathcal{R}_n は SSA 名 A_n で定義されている全領域 ($\bigcup_{k=1}^m \mathcal{R}_k^n$) を表す。式 4.3 から $\mathcal{R}_i^n (i = 1, \dots, m)$ は互いに素となっていることがわかる。この性質により、例えばプログラム内で $A_n[j] (j \in \mathcal{R}_n)$ が参照されている場合、 $\{j\}$ は $\mathcal{R}_i^n (i = 1, \dots, m)$ のどれか一つの領域にのみ含まれることがわかる。そのため、 $A_n[j]$ の定義されている SSA 名が一意に定まる。Rus らの手法ではこのように、利用する配列要素の含まれている ϕ 関数の引数をたどっていくことにより、その要素が実際に定義された位置が求められる。

Rus らの手法では ϕ 関数をその種類に応じて表 4.1 のように $\delta, \pi, \gamma, \mu, \eta, \theta$ と表す。図 4.6 に $\delta, \pi, \gamma, \mu, \eta$ の生成法を示す。以下ではこれらの ϕ 関数について、プログラム 4.5 とプログラム 4.6 を用いて説明する。

δ 関数

δ 関数は配列の定義の直後に挿入され、直前の文で定義された配列要素の領域と、それ以前に定義された領域とを合成する。例えば、後述のプログラム 4.5 の A_4 は δ 関数で定義されている。 A_4 では、直前の割り当て文で定義された A_3 と、その前の定義である A_2 を合成している。このとき A_4 の定義された配列要素の領域は、 A_2 で定義されている $\{0\}$ と A_3 で定義されている $\{1\}$ を合成した $[0 : 1]$ となる。

一般的な δ 関数の生成法は後述のプログラム 4.6 の (a) のようになる。例えば A_4 の δ 関数について考える。 A_4 は、直前で定義された A_3 と、それ以前に定義された要

素を表している A_2 を合成する. このプログラムでは, A_1 で \mathcal{R}_1 が定義され (文 1), A_3 で \mathcal{R}_2 が定義されている (文 2). このとき A_4 で定義されている要素の領域は, A_2 で定義されている要素の領域 \mathcal{R}_1 と, A_3 で定義された要素の領域 \mathcal{R}_2 を合わせたものになるので, $\mathcal{R}_1 \cup \mathcal{R}_2$ と表せる. この内, A_3 で定義されている要素の領域は, A_3 から A_4 の間に他の定義はないので \mathcal{R}_2 となる. また, A_2 で定義され, A_4 まで運ばれる要素の領域は, A_2 で定義されている要素の領域 \mathcal{R}_1 から, A_3 で書き換えられた要素の領域を除いたものなので $\mathcal{R}_1 - \mathcal{R}_2$ と表せる.

μ 関数

μ 関数はスカラーの SSA における ϕ 関数と意味が異なる. 配列の定義では, すべての配列要素が書き換えられるわけではない. そのためループ内の配列の定義では, ループの繰り返しごとに定義されている要素の状態が異なる. 例えばプログラム 4.5 の文 7 では, $i = 1$ のときは要素 $\{3\}$ が定義され, $i = 2$ のときは要素 $\{4\}$ が定義される.

$$[A_n, \mathcal{R}_n] = \mu(A_{undef}, (i = 1, p), [A_1, \mathcal{R}_1^n], \dots, [A_m, \mathcal{R}_m^n]) \quad (4.4)$$

式 4.4 に μ 関数の定義を示す. $(i = 1, p)$ は, i がループ制御変数であり, i の値が 1 から p まで変化しながらループを反復することを表している. μ 関数の引数は, ループ内部の δ, γ, η 関数で定義されているものとなる. ただしループ内に別のループや if 文などの制御構造が含まれている場合, その内部にある δ, γ, η 関数は引数に含まない. 例えば, プログラム 4.5(b) の A_9 は μ 関数で定義されており, その引数はループ内で δ 関数で定義されている A_{11}, A_{13} の 2 つになっている.

集合 \mathcal{R}_k^n はループ制御変数 i の関数となる. これらはある $j (j < i)$ 回目のループの実行時に A_k で定義され, それ以降で書き換えられずに i 回目の実行に到達した要素の集合をあらわしている. これをあらわすために, \mathcal{R}_k^n は式 4.5 のように表現する.

$$\mathcal{R}_k^n(i) = \bigotimes_{j=1, i-1}^{\cup} \left[\mathcal{R}_k(j) - \left(\text{Kill}_s(j) \cup \bigotimes_{l=j+1, i-1}^{\cup} \text{Kill}_a(l) \right) \right] \quad (4.5)$$

$$\text{where } \text{Kill}_s = \bigcup_{h=k+1}^m \mathcal{R}_h, \text{ and } \text{Kill}_a = \bigcup_{h=1}^m \mathcal{R}_h$$

$\text{Kill}_s(j)$ はループの j 回目の実行時に A_k 以降で定義された要素の集合を表す. $\bigotimes_{l=j+1, i-1}^{\cup} \text{Kill}_a(l)$ は, $j+1$ 回目から $i-1$ 回目までのループの実行時に定義された要素の集合を表している. $\mathcal{R}_k(j)$ とこれらの差集合をとったものが, ループの j 回目の実行時に A_k で定義され, i 回目の実行まで運ばれてきた要素の集合となる. これを $j = 1$ から $i - 1$ まで和をとることにより, $\mathcal{R}_k^n(i)$ となる.

例として, プログラム 4.5 の A_9 について説明する. A_9 の μ 関数の引数は A_{11} と A_{13} の 2 つである. そこで A_{11} の定義領域 $\mathcal{R}_{11}^9(i)$ を求めることを考える. $\mathcal{R}_{11}(i) = \{i + 2\}$, $\mathcal{R}_{13}(i) = \{i + 10\}$ となるので, $\text{Kill}_s(j)$ と $\text{Kill}_a(l)$ は次のようになる.

$$Kill_s(j) = \bigcup_{h=12}^{13} \mathcal{R}_h(j) = \mathcal{R}_{13}(j) = \{j + 10\} \quad (4.6)$$

$$Kill_a(j) = \bigcup_{h=1}^{13} \mathcal{R}_h(l) = \mathcal{R}_{11}(l) \cup \mathcal{R}_{13}(l) = \{l + 2\} \cup \{l + 10\} \quad (4.7)$$

これらを式 4.5 に代入すると次のようになる.

$$\begin{aligned} \mathcal{R}_{11}^9(i) &= \bigotimes_{j=1, i-1}^{\cup} \left[\{j + 2\} - \left(\{j + 10\} \cup \bigotimes_{l=j+1, i-1}^{\cup} (\{l + 2\} \cup \{l + 10\}) \right) \right] \\ &= \bigotimes_{j=1, i-1}^{\cup} \left[\{j + 2\} - \left(\{j + 10\} \cup [j + 3 : i + 1] \cup [j + 11 : i + 9] \right) \right] \\ &= \bigotimes_{j=1, i-1}^{\cup} \left[\{j + 2\} - \left([j + 3 : i + 1] \cup [j + 10 : i + 9] \right) \right] \\ &= [3 : i + 1] \end{aligned} \quad (4.8)$$

このようにして $\mathcal{R}_{11}^9(i) = [3 : i + 1]$ と求まる. 例えば $i = 5$ でのループの実行開始時に, A_9 で定義されている値は, $i = 1$ から 4 での実行時に定義された $[3 : 6]$ となり, 式 4.8 の i に 5 を代入した結果と一致する. 式 4.5 の表現は複雑になっているが, 上記の例のように単純化することが出来る場合が多い.

一般的な μ 関数の構成法はプログラム 4.6(c) のようになる.

π 関数

π 関数は If 文などの条件分岐での分岐先ブロックの先頭に挿入される. π 関数は配列の定義を含まず, このブロックへの分岐条件 *cond* を保持するだけである.

$$[A_n, \emptyset] = \pi(A_0, cond) \quad (4.9)$$

式 4.9 が π 関数の一般形である. π 関数は配列の定義を含まないが, 定義と使用の関係がより正確になる役割をする. 例えば, プログラム 4.5 の文 14 で使用されている $A_{15}[2]$ の定義位置を探索することを考える. 配列 A の要素 2 は文 4 の A_6 で定義されているが, これが定義されるのは条件 $(x > 0)$ が成り立つときのみである. そのため, $(x > 0)$ が成り立つかどうか分からないと配列 A の要素 2 が定義されているかどうか分からない. しかし, プログラム 4.5(b) にあるように A_{15} の π 関数で $(x > 0)$ の条件がついているため, 文 14 では $(x > 0)$ の条件が成り立っていることが明確になる. よって文 14 の $A_{15}[2]$ は文 4 で定義された $A_6[2]$ の値を使用していることがわかる.

プログラム 4.5 (Array SSA の例)

```

1  A[0] = 1
2  A[1] = 0
3  If (x > 0)
4    A[2] = 1
5  EndIf
6  Do i = 1, 8
7    A[i + 2] = 3
8    ... = A[...]
9    A[i + 10] = 4
10 EndDo
11 ... = A[1]
12 ... = A[5]
13 If (x > 0)
14   ... = A[2]
15 EndIf

```

(a) サンプルコード

```

A0 : [A0, ∅] = Undefined
1  A1 : A1[0] = 1
A2 : [A2, {0}] = δ(A0, [A1, {0}])
2  A3 : A3[1] = 0
A4 : [A4, [0 : 1]] = δ(A0, [A2, {0}], [A3, {1}])
3    If (x > 0)
A5 : [A5, ∅] = π([A4, (x > 0)])
A6 : A6[2] = 1
4  A7 : [A7, {2}] = δ(A5, [A6, {2}])
5    EndIf
A8 : [A8, [0 : 1] ∪ (x > 0)#{2}] =
      γ(A0, [A4, [0 : 1]], [A8, (x > 0)#{2}])
6    Do i = 1, 8
A9 : [A9, [3 : i + 2] ∪ [11 : i + 10]] =
      μ(A8, (i = 1, 8), [A11, [3 : i + 1]], [A13, [11 : i + 9]])
7  A10 : A10[i + 2] = 3
A11 : [A11, {i + 2}] = δ(A9, [A10, {i + 2}])
8    ... = A11[...]
9  A12 : A12[i + 10] = 4
A13 : [A13, {i + 2, i + 10}] = δ(A9, [A11, {i + 2}], [A12, {i + 10}])
10   EndDo
A14 : [A14, [0 : 1] ∪ (x > 0)#{2} ∪ [3 : 18]] =
      = η(A0, [A8, [0 : 1] ∪ (x > 0)#{2}], [A9, [3 : 18]])
11   ... = A14[1]
12   ... = A14[5]
13   If (x > 0)
A15 : [A15, ∅] = π(A14, (x > 0))
14   ... = A15[2]
15   EndIf

```

(b) Array SSA 形式

プログラム 4.6 (Array SSA への変換)

| | |
|----------------------------------|---|
| | $[A_0, \emptyset] = Undefined$ |
| 1 : $A(\mathcal{R}_1) = \dots$ | 1 : $A_1(\mathcal{R}_1) = \dots$ |
| 2 : $A(\mathcal{R}_2) = \dots$ | $[A_2, \mathcal{R}_1] = \delta(A_0, [A_1, \mathcal{R}_1])$ |
| n : $A(\mathcal{R}_n) = \dots$ | 2 : $A_3(\mathcal{R}_2) = \dots$ |
| | $[A_4, \mathcal{R}_1 \cup \mathcal{R}_2] = \delta(A_0, [A_2, \mathcal{R}_1 - \mathcal{R}_2], [A_3, \mathcal{R}_2])$ |
| | n : $A_{2n-1}(\mathcal{R}_n) = \dots$ |
| | $[A_{2n}, \cup_{i=1}^n \mathcal{R}_i] =$ |
| | $\delta(A_0, [A_{2n-2}, \cup_{i=1}^{n-1} \mathcal{R}_i - \mathcal{R}_n], [A_{2n-1}, \mathcal{R}_n])$ |

(a) 直線コード

| | |
|--------------------------------|---|
| | $[A_0, \emptyset] = Undefined$ |
| 1 : $A(\mathcal{R}_x) = \dots$ | 1 : $A_1(\mathcal{R}_x) = \dots$ |
| 2 : If (<i>cond</i>) | $[A_2, \mathcal{R}_x] = \delta(A_0, [A_1, \mathcal{R}_x])$ |
| 3 : $A(\mathcal{R}_y) = \dots$ | 2 : If (<i>cond</i>) |
| 4 : EndIf | $[A_3, \emptyset] = \pi(A_2, cond)$ |
| | 3 : $A_4(\mathcal{R}_y) = \dots$ |
| | $[A_5, \mathcal{R}_y] = \delta(A_3, [A_4, \mathcal{R}_y])$ |
| | 4 : EndIf |
| | $[A_6, \mathcal{R}_x, \cup cond \# \mathcal{R}_y] =$ |
| | $\gamma(A_0, [A_2, \mathcal{R}_x - cond \# \mathcal{R}_y], [A_5, cond \# \mathcal{R}_y])$ |

(b) If ブロック

| | |
|-----------------------------------|---|
| | $[A_0, \emptyset] = Undefined$ |
| 1 : Do $i = 1, n$ | 1 : Do $i = 1, n$ |
| 2 : $A(\mathcal{R}_x(i)) = \dots$ | $[A_5, \mathcal{R}_2^5(i) \cup \mathcal{R}_4^5(i)] =$ |
| 3 : $A(\mathcal{R}_y(i)) = \dots$ | $\mu(A_0, (i = 1, n), [A_2, \mathcal{R}_2^5(i)], [A_4, \mathcal{R}_4^5(i)])$ |
| 4 : EndDo | 2 : $A_1(\mathcal{R}_x(i)) = \dots$ |
| | $[A_2, \mathcal{R}_x(i)] = \delta(A_5, [A_1, \mathcal{R}_x])$ |
| | 3 : $A_3(\mathcal{R}_y(i)) = \dots$ |
| | $[A_4, \mathcal{R}_x(i) \cup \mathcal{R}_y(i)] =$ |
| | $\delta(A_5, [A_2, \mathcal{R}_x(i) - \mathcal{R}_y(i)], [A_3, \mathcal{R}_y(i)])$ |
| | 4 : EndDo |
| | $[A_6, \otimes_{i=1, n}^{\cup} (\mathcal{R}_2^5(i) \cup \mathcal{R}_4^5(i))] =$ |
| | $\eta([A_0, \emptyset], [A_5, \otimes_{i=1, n}^{\cup} (\mathcal{R}_2^5(i) \cup \mathcal{R}_4^5(i))])$ |

(c) Do ブロック

4.3.4 定義の探索

Rus らの Array SSA では, SSA 名にしたがって ϕ 関数をたどっていくことにより定義位置を求める. 例えばプログラム 4.5 で, $A_4[1]$ がどこで定義されているのかを

考えてみると, A_4 では $[0 : 1]$ の要素が定義されているので, $A_4[1]$ は定義されていることがわかる. そこで右辺の δ 関数の引数を見ていくと, 定義領域に要素 1 を含んでいるのは A_3 である. A_3 の定義は $A_3[1] = 0$ となっているので, $A_4[1]$ は A_3 で 0 で定義されていることがわかる. 本節では, このような Array SSA での定義の探索方法について述べる.

SSA 名 A_u の配列の要素 $\mathcal{R}_{Use(A_u)}$ を使用するとき, その要素の値を設定した可能性のある定義の集合を探索することを考える. そのような集合のことを到達する定義 (reaching definition) という. 定義探索のアルゴリズムを図 4.4 に示す.

このアルゴリズムは A_u に到達する定義のリスト

$$\{[A_1, \mathcal{R}_1^{RD}], [A_2, \mathcal{R}_2^{RD}], \dots, [A_n, \mathcal{R}_n^{RD}], [A_{undef}, \mathcal{R}_{undef}^{RD}]\}$$

を返す. \mathcal{R}_k^{RD} は $\mathcal{R}_{Use(A_u)}$ のうち, A_k で定義され, その後書き換えられずに A_u に到達する要素の集合をあらわす. \mathcal{R}_{undef}^{RD} は与えられたブロック (*GivenBlock*) 内で未定義な要素の集合である. この *GivenBlock* はループ内のみを探索したいなどの場合のように, 解析範囲を制限するために使われる. 例えばループを解析する場合, ループのある j 回目の実行に定義されたもののみ探索したい場合は (*GivenBlock* = loop body) のようにし, ループ全体を探索したい場合は (*GivenBlock* = whole loop) のようにすればいい.

アルゴリズムは, A_u の定義が ϕ 関数ではなく, 通常の配列の定義文の場合, その定義領域の \mathcal{R}_u と \mathcal{R}_{use} との共通部分をとることにより, 到達する定義が求まる.

A_u の定義が $\delta, \gamma, \eta, \theta$ 関数の場合, 2 つの演算を行う. まず ϕ 関数のそれぞれの引数に対してこの Search メソッドを適用し, 定義を探索する. 次にこのブロックでは未定義の領域を探索するため, A_0 に対して Search メソッドを適用する.

A_u の定義が μ 関数の場合 (図 4.4 の Case μ), まずループ内の定義を探索するために, μ 関数のそれぞれの引数に対して Search メソッドを適用する. これにより求まる定義の領域はループ制御変数 i の関数となっている. そのため i にある値 j を代入すれば, $i = j$ でのループの実行開始時の, 到達する定義が求まる. 次にループ以前で定義された要素を探索するために, ループ内で未定義の領域を表す A_{undef} に対して Search メソッドを適用して探索を行う.

A_u の定義が π 関数の場合, \mathcal{R}_{use} に条件を付けて探索を続ける.

次に図 4.4 のアルゴリズムを用いて定義を探索する例を 2 つ示す.

まずプログラム 4.5 の文 11 での $A_{14}[1]$ の到達する定義を探索する. 探索範囲はプログラム全体とする. A_{14} の定義は η 関数であり, η 関数の引数のうち, 1 と共通部分があるのは A_8 のみである. 次に A_8 を探索する. A_8 の定義は γ 関数であり, 同様に, $\{1\}$ と共通部分があるのは A_4 のみである. 同様にして A_4 から A_3 へ探索していき, A_3 は通常の見出し文で, \mathcal{R}_{use} との共通部分が $\{1\}$ であるので, 到達する定義は $[A_3, \{1\}]$ である. それ以外はすべて $\mathcal{R}_{use} = \emptyset$ となるため定義は A_3 のみとなる.

今度はプログラム 4.5 の文 12 で使用している $A_{14}[5]$ の到達する定義を探索する. 探索範囲は同様にプログラム全体とする. A_{14} の定義で, 要素 5 は A_9 とのみ共通部分がある. A_9 は μ 関数で定義されているので, まず μ 関数の引数について探索を行うが, A_{14} はループの外にあるので, ループ制御変数 i の値は 9 だとわかっている. そこ

```

Algorithm Search ( $A_u, \mathcal{R}_{use}, GivenBlock$ )
/**
 * このアルゴリズムでは, 到達する定義のリストを求める.
 * DefList は到達する定義のリスト
 *  $\{[A_1, \mathcal{R}_1^{RD}], [A_2, \mathcal{R}_2^{RD}], \dots, [A_n, \mathcal{R}_n^{RD}], [A_{undef}, \mathcal{R}_{undef}^{RD}]\}$ 
 * を表す.
 * DefList  $\leftarrow$  Call Search( ...) は
 * 右側の関数呼び出しから返されたリストの要素を
 * DefList に加えることを意味するものとする.
**/
DefList  $\leftarrow \perp$  /* DefList の初期化 */
If  $A_u \notin GivenBlock$  or  $\mathcal{R}_{use} = \emptyset$  then
    return DefList
EndIf
Switch  $A_u$  の定義文
    Case 通常の文 ( $\phi$  関数でない通常の定義文の場合):
         $\mathcal{R}_u^{RD} = \mathcal{R}_u \cap \mathcal{R}_{use}$ 
        DefList に  $[A_u, \mathcal{R}_u^{RD}]$  を加える
        Return DefList
    Case  $\delta, \gamma, \eta, \theta$ :  $[A_u, \mathcal{R}_u] = \phi(A_{undef}, [A_1, \mathcal{R}_1^u], \dots)$ 
        ForEach  $[A_k, \mathcal{R}_k^u]$  /*  $\phi$  関数の引数 */
            DefList  $\leftarrow$  Call Search( $A_k, \mathcal{R}_{use} \cap \mathcal{R}_k^u, GivenBlock$ )
            /* Search() から返されたリストの要素を DefList に加える */
        EndFor
        DefList  $\leftarrow$  Call Search( $A_{undef}, \mathcal{R}_{use} - \mathcal{R}_n, GivenBlock$ )
    Case  $\mu$ :  $[A_u, \mathcal{R}_k^u(i)] = \mu(A_{undef}, (i = 1, p), [A_1, \mathcal{R}_1^u(i)], \dots)$ 
        ForEach  $[A_k, \mathcal{R}_k^u(i)]$  /*  $\mu$  関数の引数 */
            DefList  $\leftarrow$  Call Search( $A_k, \mathcal{R}_{use}(i) \cap \mathcal{R}_k^u(i), Block(A_k)$ )
        EndFor
        DefList  $\leftarrow$  Call Search( $A_{undef},$ 
             $\bigotimes_{i=1,p}^{\cup} (\mathcal{R}_{use}(i) - \mathcal{R}_u(i)), GivenBlock$ )
    Case  $\pi(A_{undef}, cond)$ :
        DefList  $\leftarrow$  Call Search( $A_{undef}, cond \# \mathcal{R}_{use}, GivenBlock$ )
EndSwitch
Return DefList

```

図 4.4: 定義探索アルゴリズム

で i に 9 を代入すると, 要素 5 と共通部分があるのは, A_{11} のみである. よって $A_{14}[5]$ は A_{11} で定義されていることがわかる. このように, ループ内の到達する定義を探索

する場合、ループ制御変数に定まった値を代入する場合もある。図 4.4 では、複雑になるのを避けるため、この部分を省略してある。

第5章 Array SSA のC言語への適用

Rusらの手法によるArray SSAはFortranを対象としているため、C言語のプログラムにそのまま適用することは出来ない。本章では、本研究で行ったArray SSAのC言語への適用方法について述べる。以下ではArray SSAとは4.3節で説明したRusらの手法をさすものとする。

5.1 Array SSA をC言語へ適用する場合の問題点

C言語にArray SSAを適用するにはRusらの手法で考慮されていない次のようないくつかの問題がある。

- ポインタ参照されている場合
- 複雑な構造のループがある場合
- switch文の扱い
- goto文の扱い

以下でこれらの問題を本研究での解決法について説明する。

5.2 ポインタ

C言語ではポインタがあり、配列がポインタを利用して定義や使用されている場合がある。プログラム5.1にポインタを使った配列参照の例を示す。

プログラム 5.1 (ポインタを使った配列参照の例)

```
1: int main(){
2:     int a[10];
3:     int *p,x;
4:
5:     p = a;
6:     *(p + 3) = 1;
7:     x = *(p + 3);
8:     return x;
9: }
```

プログラム 5.1 では、文 5 でポインタ p に配列 a の先頭のアドレスが代入されているため、文 6, 7 の $*(p+3)$ はポインタを用いて $a[3]$ に 1 を代入し、 $a[3]$ の値を x に代入している。

本研究の実装では、実装を行ったコンパイラである COINS で行われているポインタ解析の機能を利用して、プログラム 5.1 のようにポインタを用いて参照されている配列に対しては Array SSA を適用しないことにした。

5.3 複雑な制御構造

C 言語では、5.1 節であげたような複雑なプログラムの制御構造がある。本節では、本研究で行ったこれらの複雑な制御構造に対する解決策について説明する。

これらの解決策の基本方針は、

$$[A_n, All] = Undefined \quad (5.1)$$

という式を、正確に解析できない部分の前後に挿入することによって、解析を安全側に倒すことである。式 5.1 が挿入されると、すべての要素の定義探索がこの式で止まるため、正確に定義領域を求めることは出来ないが、解析が安全に終了するようになる。

5.3.1 ループ

C 言語には Fortran のような Do ループはない。しかしプログラム 5.2 のようなループは、ループ制御変数 i の値が 0 から 9 まで変化しながらループ内の式を実行していくので、プログラム 5.3 の Do ループと意味は変わらない。そのためループ制御変数があり、ループの繰返し条件式が、ループ制御変数の比較式 ($<$, $>$, \leq , \geq) になっているような単純なループの場合は、Rus らの Array SSA を Do ループと同じようにして適用することが出来る。

プログラム 5.2 (単純なループ)

```
1: for ( $i = 0$ ;  $i < 10$ ;  $i = i + 1$ ) {  
2:    $a[i] = 1$ ;  
3: }
```

プログラム 5.3 (Do ループ)

```
1: Do  $i = 0, 9$   
2:    $a[i] = 1$ ;  
3: EndDo
```

しかし、プログラム 5.4 のように goto 文などによって、ループ外からジャンプしてループの途中に進入してくるようなものがある場合は、ループ制御変数の初期値が定まらないため、同じ手法を適用することが出来ない。

プログラム 5.4 (goto 文があるループ)

```
1: ...
2: goto LABEL;
3: ...
4: for( $i = 0$ ;  $i < 10$ ;  $i = i + 1$ ){
5:     ...
6: LABEL:
7:     ...
8: }
```

次にループ内に break 文がある場合 (プログラム 5.5) を考える. この場合, ループ出口では, break 文によってループ出口に到達したのか, ループの繰返し条件が成り立たずにループ出口に到達したのかがわからない. 一方でループ内での解析は, break 文が実行されずに, ループの繰返し条件が成り立たなくなるまで実行した場合を考慮に入れればプログラムの意味が変わることはない. そのため, ループ内には Rus らの Array SSA (μ 関数) を適用できるが, ループ出口には Rus らの Array SSA (η 関数) を適用できない.

プログラム 5.5 (break 文があるループ)

```
1: for( $i = 0$ ;  $i < 10$ ;  $i = i + 1$ ){
2:     ...
3:     if (...){
4:         break;
5:     }
6:     ...
7: }
```

次に continue 文がある場合 (プログラム 5.6) を考える. 文 4 の continue 文に到達するとループボディの残りの部分は実行されない. そのため continue 文を含む基本ブロックを支配していない基本ブロックに配列の定義があると (文 7), その定義が実行されるか分からないため, Rus らの Array SSA を適用できない.

プログラム 5.6 (continue 文があるループ)

```
1: for( $i = 0$ ;  $i < 10$ ;  $i = i + 1$ ){
2:     ...
3:     if (...){
4:         continue;
5:     }
6:     ...
7:      $a[i] = i$ ;
8: }
```

上記以外の, ループ制御変数がないループや, ループの反復条件が比較式でない場合には Rus らの Array SSA を適用することが出来ない.

本研究での手法

本研究では、これらのループの問題に対し、以下のようにループの種類を3種類に分けて Array SSA の適用を行った。

1. NORMALIZE

- Rus らの手法をそのまま適用する (μ 関数, η 関数を挿入)

2. HAS_BREAK

- ループ内には Rus らの手法を適用するが (μ 関数を利用), ループ出口には, ループ内で定義されている変数に対して, 式 5.1 を挿入する

3. NOT_NORMALIZE

- ループ内で定義されている変数に対し, ループボディの先頭とループ出口に式 5.1 を挿入する.

ループの種類の名前を NORMALIZE としているのは, この種類のループに対してループの正規化 (loop normalization) を行うためである (ループの正規化については7章で詳しく述べる).

ループの分類の仕方は以下ようになる.

- 表 5.1 の 5 条件を満たすループ

→ NORMALIZE

- 表 5.1 の条件 1 ~ 4 を満たすが, break 文を含むループ

→ HAS_BREAK

- 表 5.1 の条件 1 ~ 4 を満たすが, continue 文を含むループ

1. continue 文を支配していないブロックに配列の定義は含まれない (continue 文を支配しているブロックには, 配列の定義が含まれても良い)

→ NORMALIZE

2. continue 文を支配していないブロックに配列の定義が含まれている

→ NOT_NORMALIZE

- それ以外のループ

→ NOT_NORMALIZE

- 1: for ループまたは while ループ
- 2: ループのループ制御変数を持つ
- 3: ループの繰返し条件が比較式 ($<$, $>$, \leq , \geq) になっている
- 4: goto 文などのループ外からループの途中に進入してくるものがない
- 5: break 文, continue 文を含まない

表 5.1: ループの条件

NOT_NORMALIZE のループに対する処理について説明する. この場合, ループの反復ごとの配列の定義領域を求めることが出来ない. そこでループボディの先頭とループ出口に式 5.1 を挿入すると, ループ内では, ループの同じ反復内のみ定義を探索できて, それ以前の反復時に定義されたものに対しては探索できない. またループ出口からループ内の定義を探索することが出来なくなるため, 定義が不確定な部分に探索が進まないようになる.

NOT_NORMALIZE のループの例として, 次のようなプログラムを考える.

プログラム 5.7 (Rus らの手法が適用できないループの例)

```

1:  while (true){
2:     A[i] = i;
3:     x = x * i;
4:     if(x > ...){
5:         break;
6:     }
7:     ... = A[i] + 1;
8:     i = i + A[i - 1];
9: }
10: ...

```

このプログラムを Array SSA に変換した時の制御フローグラフは図 5.1 のようになる. この図を見ると, ループボディの先頭 (B2) とループ出口 (B5) に式 5.1 が挿入されていることがわかる. ここでブロック B4 の $A_3[i_1]$ の定義を探索すると, A_3 の定義から $A_3[i_1]$ は A_2 で定義されていることがわかる. この場合, i_1 を伝播して, B4 の一行目を「 $\dots = i_1 + 1$ 」に変換することも可能である. 一方で $A_3[i_1 - 1]$ の定義を探索すると, 定義先は A_1 でこれは *Undefined* なので, 最適化を行うことは出来ない. このように NOT_NORMALIZE の場合では, ループの同じ繰返しの実行時に定義されているものは探索を行えるが, そうでない場合はループボディの先頭で解析が終わるようになり解析が安全に終了する.

5.3.2 switch 文

Rus らの Array SSA では switch 文は考慮されていない. switch 文では, switch 文の最後か break 文に到達するまでは文を順番に実行していく. 例えばプログラム 5.8

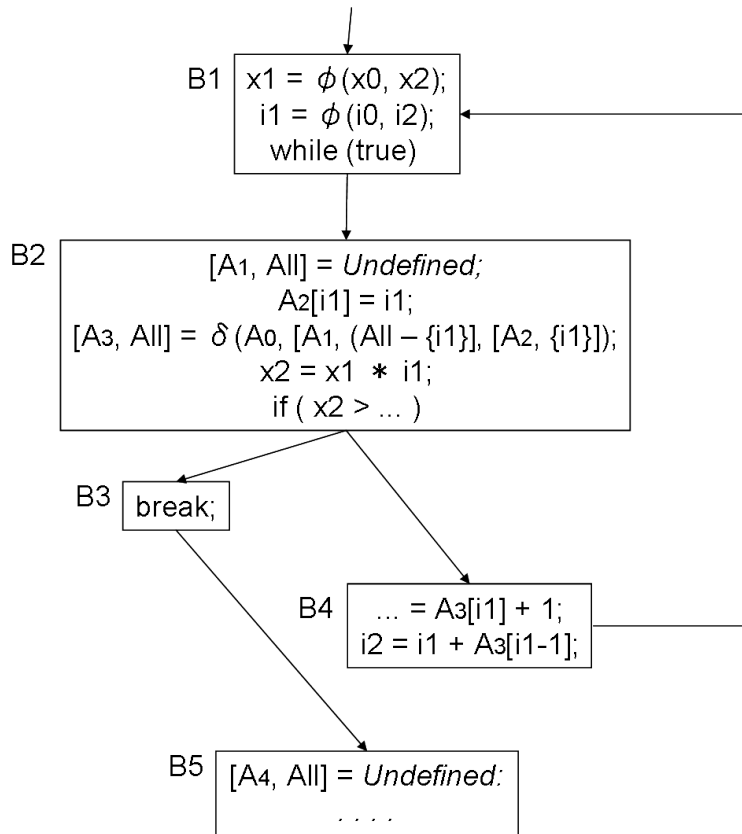


図 5.1: プログラム 5.7 の Array SSA 形式

では, a が 1 の場合は, 文 3 と文 5 が実行され, a が 2 では文 5 のみ実行される. また, a が 3 では文 8 と文 10, それ以外では文 10 のみが実行される.

このように, switch 文では制御フローが複雑になる場合があるため, 本研究では以下のように Array SSA での処理を行った.

- switch 文内で定義されている配列に対して, switch 文の前後に式 5.1 を挿入する.

これにより switch 文内に配列の定義があると, switch 文をまたいだ定義の探索は行われなくなる.

プログラム 5.8 (switch 文の例)

```

1:  switch(a){
2:  case 1 :
3:    A[1] = 1
4:  case 2 :
5:    A[2] = 2
6:    break;
7:  case 3 :
8:    A[3] = 3
9:  default :
10:   A[4] = 4
11: }

```

5.3.3 goto 文

プログラム内に goto 文があると, Array SSA をうまく適用することが出来ない. 例
 えば図 5.2 のプログラムを考える.

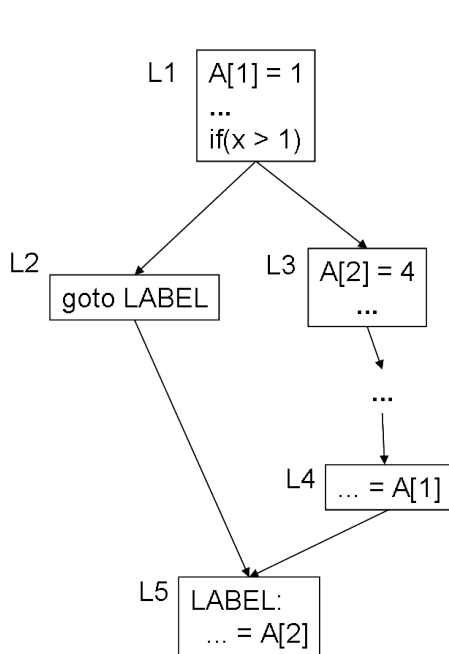


図 5.2: goto 文を使ったプログラムの例

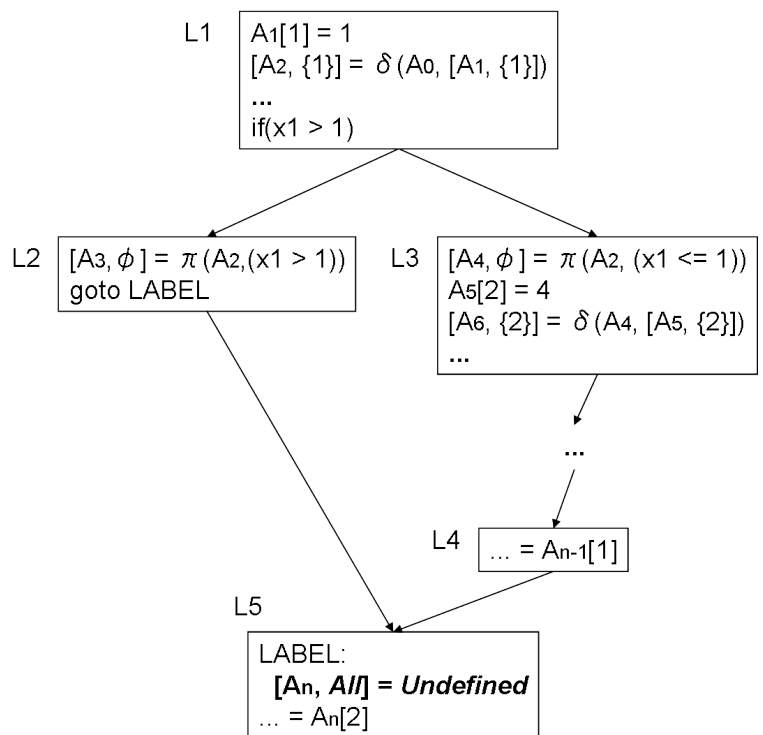


図 5.3: 図 5.2 の Array SSA 形式

このプログラムでは, ブロック L2 に goto 文があり, LABEL が付いているブロック L5 へジャンプしている. この時ブロック L5 ではプログラムの制御の流れがブロック L2 とブロック L4 のどちらから来たのかわからない. L2 を通ってきた場合, A[2]

は未定義なので、ブロック L5 で $A[2]$ が定義されているかどうかを求めることが出来ない。

そこで本研究では goto 文のジャンプ先であるブロック L5 の先頭に式 5.1 を挿入し、定義の探索がそこから先へ行かないようにした (図 5.3)。この時、goto 文と関係ない部分では通常通り探索することが可能である。

以上をまとめると、goto 文が存在した場合の本研究での処理は以下のようになる。

- プログラム内に goto 文があった場合、goto 文のジャンプ先のブロックの先頭に、すべての Array SSA を行う配列に対して、式 5.1 を挿入する。

第6章 並列化コンパイラ向け共通インフラストラクチャCOINS

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 INfraStructure, 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 [9]。現在他機関でも、多くの最適化コンパイラの研究、開発が行われているが [13, 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

一般にコンパイラは、フロントエンド (front end) とバックエンド (back end) から構成される。フロントエンドは、原始プログラム (source program) を中間コード (intermediate code) と呼ばれる内部形式に変換する。バックエンドは、中間コードを計算機の機械コード (machine code) に変換する。フロントエンドはさらに、字句解析器 (lexical analyzer)、構文解析器 (syntax analyzer)、意味解析器 (semantic analyzer) に分けられる。バックエンドは、最適化器 (optimizer) とコード生成器 (code generator) に分けられる。これら各部分は、コンパイラのフェーズと呼ばれる。

COINS では、複数の入力言語、複数の対象機種に対応する 2 つの中間コードがある (図 6.1)。入力言語の論理構造に近いレベルの中間コードを、高水準中間表現 (high-level intermediate representation, HIR) ¹ と呼び、機械語に近いレベルの中間コードを、低水準中間表現 (low-level intermediate representation, LIR) ² と呼ぶ。

¹以後 HIR と呼ぶ

²以後 LIR と呼ぶ

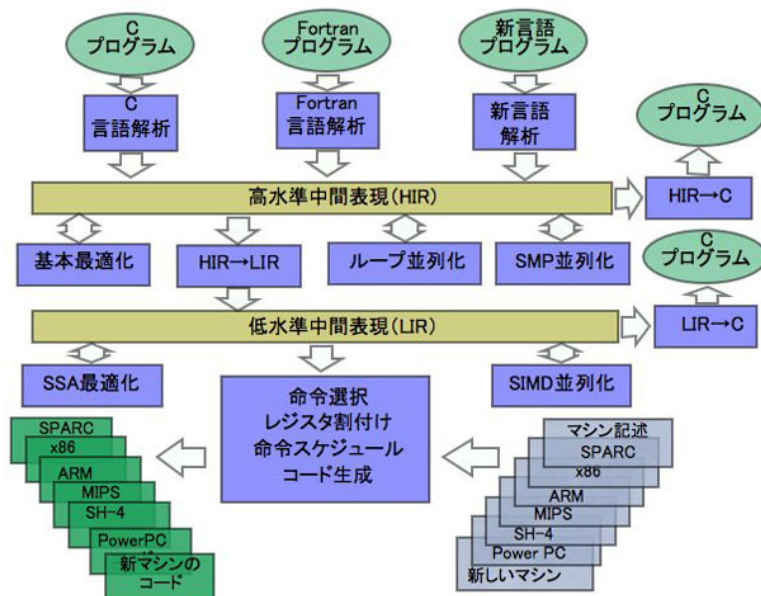


図 6.1: COINS 構成図

6.3 HIR と LIR の選択

本研究を実装する際に HIR と LIR のどちらの上で Array SSA の実装を行うのが問題となる。COINS の LIR 上では既に多くの SSA 最適化が行われている [25]。しかし、LIR では

- $a[i]$ のような配列参照がそのアドレス計算を含むいくつかの命令列に分解されてしまう。
- 機械語に近いため、for 文のような入力プログラムの構造が反映されない

という問題がある。そのため、本研究では HIR 上で Array SSA の実装を行った。

6.4 高水準中間言語 HIR について

本節では、本研究で使用する中間表現である HIR について説明する [10]。

6.4.1 概要

HIR は一般的な手続き型言語において共通的な概念を抽出し、特定の言語に依存しないように表現された抽象構文木である。HIR 自身は Java 言語で実装される。

6.4.2 具体表現

HIR をテキストで表すとき, 節は

(オペレータ 型 第1子 第2子 … 第n子)

葉は

< 種別 記号 型 >

と表す. HIR ではコンパイル単位全体を1つの木として表現する. その中には一般に, 複数の副プログラム定義が含まれる. 例えば次のプログラムがあったとする.

プログラム 6.1 (入力プログラム)

```
int fact(int p) {  
    if (p <= 1)  
        return 1;  
    else  
        return p * fact(p - 1);  
}
```

このプログラムから生成される HIR は図 6.2 のようになる.

```

(prog
  <null 0 void>
  <nullNode>
  (subpDef void
    <subp <SUBP < int > false false int> fact>
  <null void>
    (labeldSt void
      (list <labelDef _lab1>)
      <block void
        (if void
          (cmpLe bool <var int p> <const int 1>)
          (labeldSt int
            (list <labelDef _lab3>)
            (return int <const int 1>))
          (labeledSt int
            (list <labelDef _lab4>)
            (return int
              (mult int
                <var int p>
                (call int
                  (addr <PTR <SUBP < int > false false int>>
                    <subp <SUBP < int > false false int> fact>)
                  (list
                    (sub int <var int p> <const int 1>))))))
            (labeledSt void
              (list <labelDef _lab5>)
            <null void>))))))
  )

```

図 6.2: プログラム 6.1 の HIR

第7章 実装

本章では、本研究での Array SSA の実装と本手法の適用手順について述べる。

7.1 Rus らの実装との違い

本研究での実装では、5.1 章で述べた部分以外にも、Rus らの実装と異なる部分がある。本節ではこれらについて述べる。

7.1.1 グローバル配列

COINS のポインタ解析では、グローバル定義されている配列はアドレスを取られている可能性があるとしてしまう。そのため、本手法ではローカル定義されている配列のみ扱う。

7.1.2 手続き間解析

Rus らの手法では θ 関数を用いて手続き間の解析の情報を利用している。しかし、COINS では手続きごとに最適化を行う為、手続き間の解析を行うことが難しい。そのため、本研究では手続き間の解析は行わない。

また、配列が参照渡しで別の手続きの引数として渡される場合は、配列の全要素が書き換えられている可能性があるとして、その関数呼び出しの直後に

$$[A_n, All] = Undefined \quad (7.1)$$

を挿入して、到達する定義の探索が安全に終わるようにしている。

7.2 本手法の適用について

本手法は以下の順番でプログラムの変換を行っていく。

1. ループの正規化 (loop normalization)
2. スカラー変数の SSA 形式への変換
3. Array SSA 形式への変換

4. 各種最適化

- (a) 定数伝播
- (b) コピー伝播
- (c) ループ不変式移動
- (d) ループ内での配列参照の一時変数化

5. SSA, Array SSA の通常形式への逆変換

以下, これらについて順に述べていく

7.2.1 ループの正規化 (loop normalization)

μ 関数では, ループの繰り返しごとの配列の定義を表すためにループの制御変数を利用する. しかし, そのままでは初期値や繰り返し時の増分値 (減分値) などがバラバラのため扱いにくい. そこで本手法では, SSA 形式へ変換する前に, 初期値:0, 終値:ループの全繰り返し数-1, 増分値:1 となる新しいループ制御変数を導入する. このような変換をループの正規化 (loop normalization) という [18].

例えば,

$$\text{for}(i = e1; i < e2; i = i + e3) \{ \dots \}$$

のようなループを考える. 新しいループ制御変数を i' とすると, 正規化されたループは

$$\text{for}(i' = 0; i' < (e2-e1)/e3; i' = i' + 1) \{ \dots \}$$

となる. この時 i の値は $i = e3 * i' + e1$ で求まる. そのため, ループ内での変数 i の使用は $e3 * i' + e1$ で置き換えればよい.

本手法では, for ループと while ループに対して, ループの正規化が可能ならば, それを行うこととした. ループの正規化は, 以下の 4 条件を満たすときに行うことが出来る.

- ループ制御変数を持つ
- ループ制御変数の初期値が 1 つに定まる.
- ループの繰り返し条件式がループ制御変数の比較式 ($<$, $>$, \leq , \geq) となっている

これを用いると, 5.3.1 節で説明したループへの Array SSA の適用条件は以下のよう書くことが出来る.

1. NORMALIZE

- 正規化されていて, break 文や continue 文を含まないループ

- 正規化されていて, continue 文を含むが, ループ内の continue 文を含む基本ブロックを支配していないブロックに, 配列に対する定義がないループ

2. HAS_BREAK

- 正規化されているが, break 文を含むループ

3. NOT_NORMALIZE

- それ以外のループ

本手法では, もともと正規化されているループに対しても, 新しい変数を導入し, ループ内では, 新しい変数を使用するようにしている. これは, ループ出口以降でのループ制御変数の値を一定にするためである (新しい変数を導入しない場合, ループ制御変数の値が変更されてしまうことがある).

7.2.2 スカラー変数の SSA 形式への変換

ループの正規化の終了後, スカラー変数の SSA 形式への変換を行う. SSA 変換は Cytron らの方法 [11] を用いて実装を行った. この時, ループの正規化で作成されたループ制御変数は, Array SSA で利用するため, SSA 形式への変換を行わない.

そのほかに, 以下のような変数に対しては SSA 変換を行わない.

- グローバル変数
- 別名参照されている可能性のある変数

7.2.3 Array SSA 形式への変換

スカラー変数の SSA 変換が終了したら, 4 章, 5 章で説明した手法を用いて, Array SSA 形式への変換を行う. Array SSA 形式への変換アルゴリズムは付録 A に示す.

7.2.4 各種最適化

本節では, 本研究で作成した, Array SSA 上で行った最適化について説明を行う.

定数伝播

定数伝播 (constant propagation) とは $a = 1$ のように定数が代入されている変数の参照を, 定数に置き換える手法である [18, 26].

例えば, プログラム 7.1(a) に対して定数伝播を適用すると, (b) のようになる. このとき, a_1, x_1 は定数が代入されているので, 文 3,4 内の a_1, x_1 がそれぞれ 1, 3 に置き換えられる. すると a_1, x_1 は使用されないので, 文 1,2 は削除することが出来る.

プログラム 7.1 (定数伝播の例)

```
1:  $a_1 = 1;$   
2:  $x_1 = 3;$   
3:  $b_1 = a_1 + b_0;$   
4:  $x_2 = x_1 + b_1;$ 
```

(a) 最適化前

```
3:  $b_1 = 1 + b_0$   
4:  $x_2 = 3 + b_1$ 
```

(b) 最適化後

本研究では、定数が代入されている配列要素の参照に対しても、定数伝播を適用した。つまり、プログラム内で配列要素を使用しているとき、その要素の到達する定義がすべて同じ定数の場合、配列参照を定数で置き換えるというものである。

配列に対する定数伝播の例を示す。プログラム 7.2 について考える。(a) のプログラムを Array SSA に変換したものの制御フローグラフは図 7.1 になる。

プログラム 7.2 (配列に対する定数伝播の例)

```
1:  $x = 0;$   
2: for( $i = 0; i < 10; i = i + 1$ ) {  
3:    $A[i] = 1;$   
4: }  
5: for( $j = 0; j < 10; j = j + 1$ ) {  
6:    $x = x + A[j];$   
7: }  
8: return  $x;$ 
```

(a) 最適化前

```
1:  $x = 0;$   
2: for( $i = 0; i < 10; i = i + 1$ ) {  
3:    $A[i] = 1;$   
4: }  
5: for( $j = 0; j < 10; j = j + 1$ ) {  
6:    $x = x + 1;$   
7: }  
8: return  $x;$ 
```

(b) 最適化後

プログラム 7.2(a) の文 6 で使用している $A[j]$ について考える。これは図 7.1 では B6 の 2 行目の $A_5[j]$ である。この $A_5[j]$ の到達する定義を探索する。 $A_5[j]$ で参照している要素は、 j の値が 0 から 9 まで変化するので、 $[0 : 9]$ となる。ループ内に A の定

義がないので、未定義を表す A_4 を探索する。 A_4 では $[0:9]$ は A_1 で定義されている。 A_4 はループ出口なので、ループ制御変数 i に 10 を代入して A_1 を探索すると、 $[0:9]$ は A_3 で定義されている。 A_3 では要素 i が A_2 で定義されていることがわかる。 i は 0 から 9 まで変化するので、 $[0:9]$ は A_2 で定義されていることがわかる。 アルゴリズム 4.4 を用いると、 $[A_5, \{j\}] \rightarrow [A_4, [0:9]] \rightarrow [A_1, [0:9]] \rightarrow [A_3, [0:9]]$ と探索することによって、同様に求まる。 A_2 では定数 1 が代入されているため、定数伝播を行うことが出来る。 プログラム 7.2(a) に定数伝播を適用したものはプログラム 7.2(b) となる。 ここでは、定数伝播により文 6 での $A[j]$ の使用が 1 に置き換わっている。

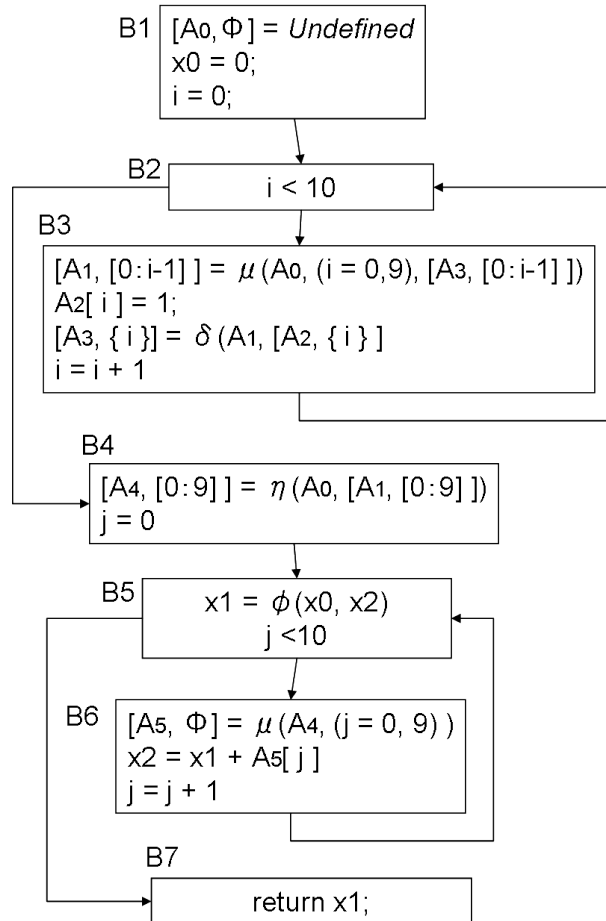


図 7.1: Array SSA 形式でのプログラム 7.2 の制御フローグラフ

コピー伝播

コピー伝播 (copy propagation) とは、ソースコード中の $a = b$ のようなコピー文を除去する手法である。 SSA 形式では、各変数の値は静的に単一である保証があるため、プログラム内で使用されている代入先変数 ($a = b$ の場合には a) を、代入元変数 ($a = b$ の場合には b) に置き換えればよい。

以下に例を示す (プログラム 7.3)。 (a) のプログラムにコピー伝播を行うと (b) の

ようになる。ここでは、文 2 の $x_1 = a_1$ のコピー文を伝播している。伝播後は $x_1 = a_1$ は不要となるので除去されている。

プログラム 7.3 (コピー伝播の例)

```
1:  $a_1 = 1;$   
2:  $x_1 = a_1;$   
3:  $a_2 = x_1 + b_0$   
4:  $b_1 = a_1 + x_1$   
(a) 最適化前
```

```
1:  $a_1 = 1$   
2:  $a_2 = a_1 + b_0$   
3:  $b_1 = a_1 + a_1$   
(b) 最適化後
```

本研究では、この手法を配列に拡張し、 $A[i] = b$ のように配列要素の変数をコピーしている場合に $A[i]$ の使用を b に置き換えることを行った。このようにすることによって、配列要素のアドレス計算の必要がなくなるので、実行時間を速くすることが出来ると思われる。

次に配列に対するコピー伝播の例を示す (プログラム 7.4)。

プログラム 7.4 (配列に対するコピー伝播の例)

```
1:  $A[i] = x;$   
2:  $y = a + b;$   
3:  $z = y + A[i]$   
(a) 通常形式
```

```
1:  $[A_0, \emptyset] = Undefined$   
2:  $A_1[i_0] = x_0;$   
3:  $[A_2, \{i_0\}] = \delta(A_0, [A_1, \{i_0\}])$   
4:  $y_1 = a_0 + b_0$   
5:  $z_1 = y_1 + A_2[i_0];$   
(b) Array SSA 形式
```

```
1:  $A[i] = x;$   
2:  $y = a + b;$   
3:  $z = y + x;$   
(c) 最適化後
```

(b) の 5 行目で使用されている $A_2[i_0]$ の定義を探索すると、文 2 の A_1 で定義されていることがわかる。 A_1 は x_0 が代入されているので、コピー伝播を行うと (c) のようになる。

ループ不変式移動

ループの繰り返しによらず一定の結果を与える式をループ不変式という。このループ不変式をループの外、つまりループの始まる前へ移動することにより、ループ内の命令を減らして効率をあげることが出来る。このような最適化をループ不変式移動という [1, 18, 3, 26].

ループ不変式とは、ループ内に存在する式 $e : t = a_1 \text{ op } a_2$ の各オペランドが以下のいずれかの性質を持つときの計算である。SSA 形式では、その性質は次のようになる。

- a_i が定数である
- a_i の定義がループ外にある
- a_i を定義する文にループ不変の印が付いている

本研究では、式 e のオペランドに配列要素がある場合 (式 e を $e : t = A[e1] \text{ op } a_2$ とする) でも、配列参照 $A[e1]$ が次の 2 つの条件を満たす場合、ループ不変式移動を行えるようにした。

- 添え字式 $e1$ がループ不変式である
 - $A[e1]$ で参照されている配列 A の要素の到達する定義が、すべてループ外にある
- ループ不変式移動の例を示す (プログラム 7.5).

プログラム 7.5 (ループ不変式移動の例)

```
1:  x = 1;
2:  for(i = 0; i < 100; i = i + 1){
3:      A[i] = x;
4:  }
5:  for(i = 0; i < 50; i = i + 1){
6:      y = A[x] + 5;
7:      z = A[i] + 3;
8:  }
```

(a) 最適化前

```
1:  x = 1;
2:  for(i = 0; i < 100; i = i + 1){
3:      A[i] = x;
4:  }
5:  y = A[x] + 5;
6:  for(i = 0; i < 50; i = i + 1){
7:      z = A[i] + 3;
8:  }
```

(b) 最適化後

(a) の文 6 の右辺を見ると, x はループ内で定義されていないのでループ不変であり, $A[x]$ は文 3 で定義されているため, 上記の条件を満たす. よって, 文 6 ではループ不変式となるのでループ外へ移動することが出来る. 一方で文 7 は, $A[i]$ の i がループの制御変数でループ不変ではない. そのためループ外へ移動することが出来ない. 最適化後のプログラムは (b) のようになる.

ループ内の式 e がループ不変式であった場合, その式を安全にループ外に追い出すには, その式を含む基本ブロック B が, すべてのループ出口ブロックを支配していなければならない [1]. そのため, 上記の例ではプログラム 7.5(a) の文 6 を含むブロックは, このループの出口ブロックを支配していないので, このままではループ不変式移動を行うことが出来ない. 本研究の実装では, COINS の HIR のループにある Conditional init part というものを利用してループ不変式移動を行っている. Conditional init part はループの繰り返し条件が成立した場合に最初の一度だけ実行される部分である (プログラム 7.5 ではこの部分を省略してある). そこで本研究では, 不変式 e と e を含む基本ブロック B に対し,

- 式 e は if 文の then ブロック, else ブロック内ではない
- ブロック B がループ内のジャンプ (continue 文, break 文, goto 文) や, return 文を含む基本ブロックを支配している

という条件が成り立つ場合にループ不変式移動を行っている.

ループ内での配列参照の一時変数化

ループ内での配列参照の一時変数化とは, ループ内でのある配列 A の定義, または使用している要素がすべて一致する場合, それらを一時変数に置き換えることを行うものである.

まず例を示す (プログラム 7.6).

プログラム 7.6 (配列参照の一時変数化の例 1)

```
1: for( $i = 0; i < 100; i = i + 1$ ){
2:   for( $j = 0; j < 10; j = j + 1$ ){
3:      $A[i] = A[i] + j;$ 
4:   }
5: }
```

(a) 最適化前

```
1: for( $i = 0; i < 100; i = i + 1$ ){
2:    $t = A[i];$ 
3:   for( $j = 0; j < 10; j = j + 1$ ){
4:      $t = t + j;$ 
5:   }
6:    $A[i] = t;$ 
7: }
```

(b) 最適化後

(a) の文 2 の for ループを考える. このループ内では文 3 の $A[i]$ を利用している. このループ内では配列 A の参照要素は i のみで, i はループ不変である. これに配列参照の一時変数化を適用すると (b) のようになる. (b) では, 文 2 で $A[i]$ の値を一時変数 t に代入し, ループ内では t を利用している (文 4). ループの出口では t を配列要素 $A[i]$ に戻している. このように配列参照の一時変数化を行うと $A[i]$ の値を再利用することが出来る.

次に別の例 (プログラム 7.7) について考える.

プログラム 7.7 (配列参照の一時変数化の例 2)

```
1: for( $i = 0; i < 100; i = i + 1$ ){
2:    $A[i] = A[i] + i;$ 
3:    $x = x + A[i];$ 
4:    $y = y - A[i];$ 
5: }
```

(a) 最適化前

```
1: for( $i = 0; i < 100; i = i + 1$ ){
2:    $t = A[i];$ 
3:    $t = t + i;$ 
4:    $x = x + t;$ 
5:    $y = y - t;$ 
6:    $A[i] = t;$ 
7: }
```

(b) 最適化後

(a) の文 1 の for ループを考えると、このループ内で使用されている配列 A の要素は $A[i]$ のみである。しかし、 i はこのループの制御変数なのでループの繰り返しごとに使用する要素は異なる。そのためプログラム 7.6 のようにループの入り口で変数にロードし、ループ出口で配列に戻すということは出来ない。しかし、参照している要素の式の中の変数が、ループ制御変数以外すべてループ不変の場合、ループの繰り返し内では一定である。そのため、ループ内で 3 箇所以上参照されている場合、(b) のようにループボディの入り口と出口で一時変数に置き換える。

以上をまとめて、配列参照の一時変数化では次のような変換を行う。ただしループ内に配列の定義がない場合は、ストア命令は挿入しない。

- ループ内のすべての配列参照の参照要素が一致し、添え字式 ($A[e]$ の e) がループ不変の場合
 - ループ前後で使用する配列要素の値を一時変数にロード、ストアし、ループ内での配列参照をすべて一時変数に置き換える。
- ループ内のすべての配列参照の参照要素が一致し、添え字式 ($A[e]$ の e) がループ制御変数以外ループ不変かつ、配列参照が 3 箇所以上ある場合
 - ループボディの入り口と出口で一時変数にロード、ストアし、ループ内での配列参照をすべて一時変数に置き換える。

7.2.5 SSA, Array SSA から通常形式への逆変換

Array SSA 形式上での最適化が終わったら、通常形式への変換を行う。スカラー変数の SSA 逆変換は Sreedhar らの方法 [29] 中の Method III を用いた。SSA 形式上でのコアレシシングも行っている。

配列の通常形式への変換 (Array SSA 逆変換) は以下の 2 つを行うことで通常形式に戻すことが出来る。

- プログラム内に挿入した ϕ 関数を削除する
- SSA 名になっている配列の名前を元の名前に戻す

Array SSA への変換では、配列の名前を変えている以外、配列の定義の状態に変化はない (配列に対する ϕ 関数は、データフロー情報を挿入しているだけといえる)。また、本研究で行った Array SSA 形式上の最適化は、配列の定義の状態を変えるような変換 (不要命令除去や配列の定義式のコード移動など) は行っていない。そのため、最適化後でも配列の定義の状態に変化はない。また、ループ不変式移動では、配列を参照している式を移動しているが、移動の条件に“参照している要素がすべてループ外で定義されている”というものがあるため、参照している配列要素の定義を飛び越してコード移動することはない。よって、上記の 2 つの処理を行うことで通常形式のプログラムに戻すことが出来る。

第8章 実験と考察

本章では, 本研究で実装した Array SSA を用いた最適化の効果を実験により検証, 評価する.

8.1 実験

8.1.1 実験の環境

実験には, COINS version 1.4.3 を用い, 富士通の PRIMEPOWER 250 (表 8.1) で行った.

実験では, 次のような 3 種類で比較を行う.

- no opt : 最適化なし
- scalar : スカラー変数に対してのみ最適化を適用
SSA 変換 → 定数伝播 → コピー伝播 → ループ不変式移動 → SSA 逆変換
- array : スカラー変数と配列に対して最適化を適用
ループの正規化 → SSA 変換 → Array SSA 変換 → 定数伝播
→ コピー伝播 → ループ不変式移動 → 配列参照の一時変数化 →
SSA, Array SSA 逆変換

効果を確認するためのベンチマークプログラムとして, 4 個の小さなプログラムと, SPEC CPU 2000 v1.1 [27] の 4 個のベンチマークを使用して, 各最適化を行った. 使用した小さなベンチマークプログラムは以下の 4 個である.

| | |
|------------|-------------------|
| 機種 | PRIMEPOWER 250 |
| プロセッサ種別 | 1.98GHz SPARC64 V |
| プロセッサ数 | 2 |
| 1 次キャッシュ | 256KB |
| 2 次キャッシュ | 3MB |
| メモリ容量 | 10GByte |
| オペレーティング環境 | SunOS 5.10 |

表 8.1: PRIMEPOWER 250 の主な仕様

- 素数算出 (prime)
- 相関係数の計算 (soukan)
- 行列やベクトルの積 (matmult)
- 選択ソート (Selection Sort)

SPEC からは以下の 4 個を使用した.

- 183.equake
- 179.art
- 255.vortex
- 256.bzip2

本研究では, 目的コードの実行時間を測定する. また SPEC にベンチマークに対しては, コンパイル時間についても測定を行った.

8.1.2 実行時間

目的コードの実行時間の結果をを図 8.1 に示す. これは no opt の実行時間を 1 としたときの相対値のグラフである. 実行時間は 3 回の測定の中央値を用いた.

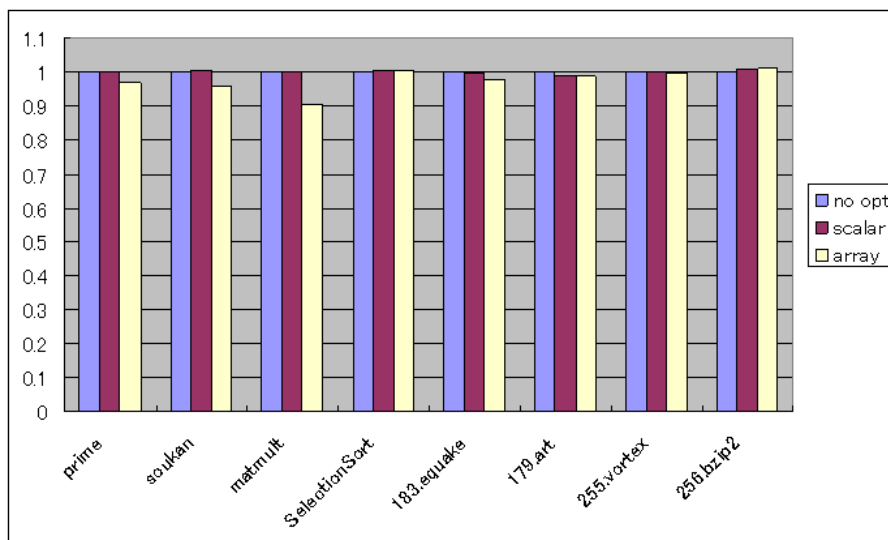


図 8.1: テストプログラムの実行時間

図 8.1 を見ると, 小さいプログラムでは選択ソートを除いて, Array SSA によって数%から 10%程度実行時間が改善している. SPEC では 183.equake, 179.art で若干の実行時間の改善が見られた.

そのほかのプログラムでは最適化の効果がほとんど出ていない。これは本手法での配列に対する最適化がほとんど適用できなかったためである。その原因としては、本研究の実装上の問題がいくつかあり、Array SSA による最適化の適用範囲が制限されてしまったことがあげられる（これらの問題については8.2節で触れる）。そのような状況下でも上記のような効果が出たことから、Array SSA は効果的な手法であるといえる。

また、全体としてスカラー最適化の効果が出ていないことがわかる。これは、COINS のHIR 上に SSA 形式を導入するにあたり、いくつかの問題（付録 B 参照）があり、SSA 逆変換によって挿入されるコピー文が、本来の Sreedhar らの手法の Method III より多くなってしまった。そのため、最適化による実行時間の改善が相殺されてしまったことが原因として挙げられる。また、本研究で実装を行った高水準中間表現 HIR は、ソースプログラムに近く、LIR のようにコードが細かく分解されていない。そのため、HIR でスカラー変数の最適化を行う場合、細かいコードの再利用などを行えないため、スカラー変数に対する最適化の対象が少なくなっていることも考えられる。

実験結果より、HIR 上ではスカラー変数の最適化の効果が低いため、HIR 上ではスカラー変数の最適化は行わず（配列に対してのみ最適化を行う）、LIR 上でスカラー変数に対する最適化を行ったほうが効果が大きいのではないと思われる。

8.1.3 コンパイル時間

図 8.2,8.3 にコンパイル時間の実験結果を示す。図 8.2 は no opt のコンパイル時間を 1 とした時の相対値、図 8.3 は scalar のコンパイル時間を 1 とした時の、array のコンパイル時間の相対値を表す。コンパイル時間は 3 回の測定の中央値を用いた。

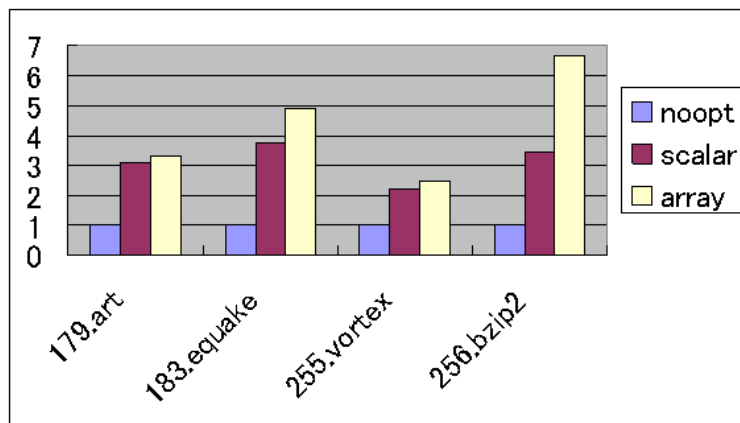


図 8.2: コンパイル時間 (no opt のコンパイル時間を 1 とした時の相対値)

図 8.2 を見ると、array のコンパイル時間は no opt のコンパイル時間の 2 倍から 7 倍の間となっている。256.bzip2 のコンパイル時間は no opt と比べると 7 倍近くなくなってしまっているが、図 8.3 を見ると、scalar に対しては 2 倍以内に収まっていること

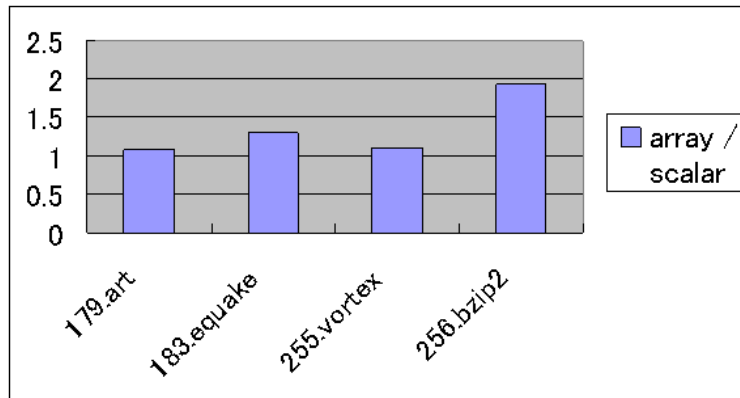


図 8.3: コンパイル時間 (scalar のコンパイル時間を 1 とした時の相対値)

がわかる。そのほかのベンチマークでは, scalar のコンパイル時間に対して, 1.3 倍以内の増加で収まっていることがわかる。

以上のように, Array SSA のコンパイル時間は, スカラー変数のみに SSA 最適化を適用したものに比べて, 全体的にわずかな上昇で収まり, コンパイル時間が大きく増える場合でも 2 倍以内に収まることから, 実用的な時間で解析が終了するといえる。

8.2 今後の課題

本研究の実装にはいくつかの問題があり, そのために Array SSA による最適化の適用範囲が制限されてしまった。本節では, 今後の課題としてこれらの問題について述べる。

8.2.1 新たな最適化の追加

本研究では Array SSA 上で 4 つの最適化を行ったが, 新しい最適化を追加することで, さらに実行時間が改善すると思われる。今後新たに最適化を実装し, その効果を確かめたい。

8.2.2 ループ構造の変換

本手法での, ループ内での配列参照に対する定数伝播やコピー伝播では, その配列参照で参照されるすべての要素の定義が 1 箇所に定まることが必要となる。例えば, プログラム 8.1 について考える。

プログラム 8.1 (ループ内での配列参照の例)

```
1:  for( $i = 0; i < 50; i = i + 1$ ) {
2:       $A[i] = 1;$ 
3:  }
4:  for( $i = 50; i < 100; i = i + 1$ ) {
5:       $A[i] = -1;$ 
6:  }
7:  for( $i = 0; i < 10; i = i + 1$ ) {
8:       $\dots = \dots + A[i];$ 
9:  }
10: for( $i = 0; i < 100; i = i + 1$ ) {
11:      $\dots = \dots + A[i];$ 
12: }
```

このプログラムでは、配列 A は要素 0 から 49 は 1、要素 50 から 99 は -1 で定義されている。この A に対して定数伝播をすることを考えると、文 8 の $A[i]$ で参照する要素はすべて文 2 で定義されているため、定数伝播を行うことが出来る。一方、文 11 の $A[i]$ で参照する要素は、文 2 で定義されているものと文 5 で定義されているものがあるため、定数伝播を行うことが出来ない。しかし、この例の場合、文 10 のループを i の値が 0 から 49 までと、50 から 99 までの 2 つに分割することによって定数伝播を行うことが可能である。このように、配列の一部が定数で定義されている場合、ループ分割やループ展開を行うことにより、最適化を行うことが出来る。

一般に、ループ内での配列参照で使用する要素がすべて一箇所で定義されていることは少ないが、部分的に定数で定義されている場合はしばしばあるため、上記のようなループ変換を組み合わせれば、さらに最適化の効果を挙げることが可能である。しかし、SSA 形式上で制御フローを変更すると、 ϕ 関数の扱いが難しい。そのため、本研究の実装では、このようなループ構造の変換を伴う最適化は行わなかった。今後このようなループ構造の変換を伴う最適化も行っていきたい。

8.2.3 手続き間の解析

COINS では手続きごとに最適化が行われるため、手続き間の解析を行うのは難しい。しかし、配列の初期化と配列への演算を行う部分が別の手続きになっている場合も多い。この場合は、定義の探索を行ってもすべて *Undefined* となってしまうため、最適化を行うことが出来ない。そこで手続き間の解析を行うことにより、さらに実行時間が改善されると思われる。

8.2.4 ポインタ解析とグローバル配列への Array SSA の適用

現在の COINS の HIR でのポインタ解析では、グローバル配列はポインタを利用して参照されている可能性があるとしてしまうため、グローバル配列には Array

SSA を適用していない。またポインタ参照されている場合も Array SSA を適用していない。この場合でも、ポインタを利用して定義されている部分などを考慮に入れば Array SSA の利用が可能になるとと思われる。

8.2.5 LIR 上の最適化との組み合わせ

本研究ではスカラー変数にも最適化を行ったがほとんど効果が出なかった。COINS コンパイラでは低水準中間表現 LIR 上でも、スカラー変数に対する SSA 最適化を行っているため、スカラー変数の最適化はそれを利用したほうが効果が出ると思われる。そこで、HIR で配列に対しての最適化を行い、LIR でスカラー変数に対する最適化を行った場合の、最適化の効果を計り確かめたい。

第9章 関連研究

本章では、本研究に関連の深い研究について述べる。

9.1 Array SSA

9.1.1 Fink らの研究

Fink らの研究 [12] は、Java や Modula-3 のような型付けの強い言語に対して、オブジェクトのフィールドと配列の要素の、解析と最適化を行うものである。彼らの手法では、オブジェクトのフィールドや配列の要素ごとに、heap arrays というものを用意する。この heap arrays の構築の際には、Knobe らの Array SSA [17, 24] を拡張して利用している。

Fink らの手法では、3 種類の ϕ 関数を導入する

1. control ϕ : スカラー変数に対する SSA 形式と同様に、制御フローの合流点に挿入する。
2. definition ϕ : 配列要素への定義のように、一部分を書き換える定義の後に挿入する。Array SSA での定義の後に挿入する ϕ 関数に対応する。
3. use ϕ : heap array の要素の読み込み (オブジェクトのフィールドの参照や、配列要素の参照) の後に挿入される。

Fink らはこれらの ϕ 関数を挿入し Array SSA 形式に変換した上で、冗長なロード命令の除去と不要なストア命令の除去の最適化を行っている。彼らの手法は強い型付けによる情報を利用しているため、型付けの弱い C 言語のようなプログラムでは効果が落ちてしまう。

9.2 スカラー最適化の配列への拡張

スカラー変数への最適化を配列へ拡張したものとして Vanbroekhoven らの研究 [31] と Wonnacott の研究 [33] がある。

9.2.1 Vanbroekhoven らの研究

Vanbroekhoven らの研究は, DSA 形式というプログラムの表現形式を利用して, 配列に対してコピー伝播を行っている. DSA 形式とは, プログラムの実行時にすべての変数とすべての配列要素が一度しか定義されないというプログラムの表現形式である. DSA 形式は強力な表現形式であるが, 適用できるプログラムに制約があり, C 言語にはそのままでは適用できない. DSA 形式への効率の良い変換手法は Vanbroekhoven らによって提案されている [32].

9.2.2 Wonnacott らの研究

Wonnacott の研究は, 配列へのデータフロー解析を用いた, 配列に対して定数伝播や不要命令除去などのいくつかの最適化のアルゴリズムを示している. 配列のデータフローの情報の表現と利用には, オメガライブラリ [16] を用いている. 彼の研究では最適化されたコードの実行時間は測定されていない. また, 依存のサイクルがなく, 配列の添え字式がアフィンであるという制約があるため, 一般の C 言語のプログラムに適用することは出来ない.

第10章 まとめ

本研究では, Rus らの提案した Fortran 言語用の Array SSA の手法に C 言語にも適用できるようにいくつかの拡張を施し, COINS 上で Array SSA の実装を行った.

また, Array SSA 上でいくつかの最適化を実装し, その効果を計った. その結果, いくつかのプログラムで数%から 10%の実行時間の改善が見られ, コンパイルも実用的な時間で終わることが分かり, Array SSA が効果的な手法であることが確認できた.

本研究で行った Array SSA 形式への変換アルゴリズムを付録 A に示した. これを用いれば, C 言語へ対応した Array SSA への変換が可能である.

本研究の実装では, 手続き間の解析やループ構造の変換は行っていない. 今後これらの解析を行っていけば, 更なる実行時間の改善が出来ると思われる.

謝辞

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

また, 佐々研究室の皆様にはさまざまな面で助力を頂きました. あらためまして, 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 2nd ed.* Addison Wesley, 2006.
- [2] Alpern, B., Wegman, M. N., and Zadeck, F. K. 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, New York, NY, USA, 1988. ACM Press.
- [3] Andrew W. Appel. *Modern Compiler Implementation in Java second edition.* Cambridge University Press, 2002.
- [4] Aycock, J. and Horspool, R. N. Simple Generation of Static Single Assignment Form. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction* , pp. 110-124, London, UK, 2000. Springer-Verlag.
- [5] Brandis, M. M. and Mössenböck, H. Single-pass generation of static single assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, Vol.16, No.6, pp. 1684-1698, 1994.
- [6] Briggs, P., Cooper, K. D., and Torczon, L. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, Vol. 16, No. 3, pp. 428-455, 1994.
- [7] Briggs, P., Cooper, K. D., Harvey, T. J., and Simpson, L. T. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, Vol. 28, No. 8, pp. 859-881, 1998.
- [8] Chaitin, G. J. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pp. 98-101, New York, NY, USA, 1982. ACM Press.
- [9] COINS Project. COINS home page. <http://www.coins-project.org/> .
- [10] COINS Project. 高水準中間表現の概要, 2004. <http://www.coins-project.org/050303/base/HirOutline.pdf>.
- [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. 41-486, October 1991.
- [12] S. Fink, K. Knobe, and V. Sarkar. Unified Analysis of Array and Object References in Strongly Typed Languages. In *Proc. Static Analysis Symp.*, LNCS 1824, pp. 155-174. London, UK, 2000.
- [13] GNU-Project. GCC homepage. <http://gcc.gnu.org/>.
- [14] George, L. and Appel, A. W. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, Vol. 18, No. 3, pp. 300-324, 1996.
- [15] INTEL. Intel Compilers homepage. <http://www.intel.com/cd/software/products/asmo-na/eng/compilers/index.htm>.
- [16] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, March 1995.
- [17] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *ACM POPL*, pp. 107-120, San Diego, CA, Jan. 1998.
- [18] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.
- [19] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM TOPLAS*, 24(1):65-109, 2002.
- [20] Park, J. and Moon, S. M. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, Vol 26, No. 4, pp. 735-765, 2004.
- [21] F. Rastello, F. de Ferrière, and C. Guillon. Optimizing Translation Out of SSA Using Renaming Constraints. In *IEEE CGO '04*, pp. 265-276, March 2004.
- [22] S. Rus, J. Hoeflinger, and L. Rauchwerger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. of Parallel Programming*, 31(3):251-283, 2003.
- [23] S. Rus, G. He, C. Alias, and L. Rauchwerger. Region Array SSA. In *ACM PACT'06*, pp. 43-52, Seattle, Washington, USA, Sep. 2006.
- [24] V. Sarkar and K. Knobe. Enabling sparse constant propagation of array elements via array ssa form. In *Proc. Static Analysis Symp.*, LNCS 1503, pp. 33-56, Pisa, Italy, Sept., 1998.
- [25] 佐々研究室. 静的単一代入形式最適化システム外部仕様書, 2007. <http://www.is.titech.ac.jp/~sassa/coins-www-ssa/japanese/ssa-external-japanese.pdf>.

- [26] 佐々政孝. プログラミング言語処理系. 岩波書店, 1989.
- [27] SPEC. Standard performance evaluation corporation home page.
<http://www.spec.org/> .
- [28] Sreedhar, V. C. and Gao, G. R. A Linear Time Algorithm for Placing ϕ -nodes. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 62-73, San Francisco, California, 1995.
- [29] Sreedhar, V. C., Ju, R. D.-C., Gillies, D. M., and Santhanam, V. 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, London, UK, 1999. Springer-Verlag.
- [30] The Machine SUIF Group. Machine SUIF homepage.
<http://www.eecs.harvard.edu/hube/research/machsuiif.html>.
- [31] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. Advanced copy propaagation for arrays. In *ACM LCTES '03*, pp. 24-33, SanDiego, CA, June, 2003.
- [32] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. A Practical Dynamic Single Assignment Transformation. In *ACM TODAES '07*, Vol. 12, No. 4, Article 40, Pub. date: Sept. 2007.
- [33] D. Wonnacott. Extending scalar optimizations for arrays. In *LCPC '00*, LNCS 2017, pp. 97-111, Yorktown Heights, NY, 2000.

付録 A Array SSA 形式への変換アルゴリズム

本研究で実装した Array SSA 形式への変換アルゴリズムについて述べる。このアルゴリズムでは、すでにループの正規化とスカラー変数の SSA 形式への変換を行っていることを前提としている。

A.1 ϕ 関数について

本研究の実装で用いる、配列に対する ϕ 関数の具体的な定義について示す。以下では A は配列を表し、 \mathcal{R} は定義領域 (USR で表される) を表す。

- δ 関数

$$[A_n, \mathcal{R}_n] = \delta(A_{undef}, [A_{n-2}, \mathcal{R}_{n-2}], [A_{n-1}, \mathcal{R}_{n-1}])$$

- * $[A_{n-1}, \mathcal{R}_{n-1}]$: n-1 回目の A の定義での SSA 名と定義された領域
- * $[A_{n-2}, \mathcal{R}_{n-2}]$: n-2 回目の A の定義での SSA 名と定義された領域
- * A_{undef} : この文を含むブロックでの未定義領域を表す変数

- π 関数

$$[A_n, \emptyset] = \pi(A_{undef}, (cond))$$

- * $cond$: この文を含むブロックへの分岐条件

- γ 関数

$$[A_n, \mathcal{R}_n] = \gamma(A_{undef}, [A_{prev}, \mathcal{R}_{prev}], [A_{then}, cond_{then} \# \mathcal{R}_{then}], [A_{else}, cond_{else} \# \mathcal{R}_{else}])$$

- * $[A_{prev}, \mathcal{R}_{prev}]$: if 文以前での A の SSA 名とその定義された領域
- * $[A_{then}, cond_{then} \# \mathcal{R}_{then}]$: then 部での SSA 名とその定義された領域
- * $cond_{then}$: then 部への分岐条件
- * $[A_{else}, cond_{else} \# \mathcal{R}_{else}]$: else 部での SSA 名とその定義された領域
- * $cond_{else}$: else 部への分岐条件

- μ 関数

$$[A_n, \mathcal{R}_n(i)] = \mu(A_{undef}, (i = 0, e), [A_k, \mathcal{R}_k^n(i)], \dots)$$

- * i : 対応するループの制御変数
 - * e : ループ制御変数の終値
 - * $[A_k, \mathcal{R}_k^n(i)]$: ループ内の定義の SSA 名とその定義領域 (i の関数となっている)
- (本研究では, μ 関数を挿入するループは正規化してあるため, ループ制御変数の初期値は常に 0 となる)

- η 関数

$$[A_n, \mathcal{R}_n] = \eta(A_{undef}, [A_{prev}, \mathcal{R}_{prev}], [A_{loop}, \mathcal{R}_{loop}])$$

- * $[A_{prev}, \mathcal{R}_{prev}]$: ループ以前の SSA 名とその定義された領域
- * $[A_{loop}, \mathcal{R}_{loop}]$: ループ内の対応する μ 関数で定義された SSA 名と, ループ内で定義された領域

- $undef$ 関数

$$[A_n, \mathcal{R}_n] = undef(A_{undef}, Undefined, bool)$$

- * \mathcal{R}_n : プログラムの先頭では \emptyset , それ以外では All となる
- * A_{undef} : この文を含むブロックで未定義な領域を表す変数. プログラムの先頭では $null$ となる.
- * $bool$: HAS_BREAK のループの出口に挿入された場合のみ $true$, そうでなければ $false$ となるブール変数

A.2 ϕ 関数の挿入アルゴリズム

図 A.1, A.2 に配列に対する ϕ 関数の挿入アルゴリズムを示す. アルゴリズム内ではループの種類を NORMALIZE, HAS_BREAK, NOT_NORMALIZE の 3 種類に分類している. それぞれのループの意味は以下のようになる.

- NORMALIZE
 - 正規化されていて, break 文や continue 文を含まないループ
 - 正規化されていて, continue 文を含むが, ループ内の continue 文を含む基本ブロックを支配していないブロックに, 配列に対する定義がない
- HAS_BREAK
 - 正規化されているが, break 文を含むループ
- NOT_NORMALIZE
 - それ以外のループ

A.3 変数の名前替え

図 A.3, A.4, A.5, A.6, A.7 に配列変数の名前替えアルゴリズムを示す。アルゴリズム内で、各 ϕ 関数内の変数名の初期値が A_n や A_{prev} のように区別されているが、これはアルゴリズムの表記上の区別で、実装上は、すべてオリジナルの変数名である A となっている。

```

for each 文 S do
  if S が配列 A の定義文 then
    S の直後に A に対する  $\delta$  関数を挿入
  else if S が if 文 then
    for each if 文 S の then 部で定義 or 使用されている配列 A do
      then 部の入り口ブロックの先頭に, A に対する  $\pi$  関数を挿入
      /*  $\pi$  関数内の分岐条件 cond はこの段階で設定しておく */
    end for
    for each if 文 S の else 部で定義 or 使用されている配列 A do
      else 部の入り口ブロックの先頭に, A に対する  $\pi$  関数を挿入
      /*  $\pi$  関数内の分岐条件 cond はこの段階で設定しておく */
    end for
    for each then 部または else 部で定義されている配列 A do
      if 文 S の出口ブロックの先頭に  $\gamma$  関数を挿入
      /**  $\gamma$  関数内の then 部, else 部への分岐条件式は,
          * この段階で設定しておく
          * また, then 部または else 部の片方だけに A の定義がある場合,
          * 定義のない部分に対応する  $\gamma$  関数の引数は null となる **/
    end for
  else if S がループ L の先頭 then
    call insertPhiForLoop(L)
  else if S が switch 文 then
    for each switch 文内で定義されている配列 A do
      文 S の直前と, switch 文 S の出口ブロックの先頭に
      undef 関数を挿入
    end for
  else if S が goto 文 then
    for each Array SSA を適用する配列 A do
      goto 文のジャンプ先ブロックの先頭に undef 関数を挿入
    end for
  else if S が Call 文 then
    for each 参照渡しで渡されている実引数内に含まれている配列 A
do
  S の直後に undef 関数を挿入
end for
end if
end for
for each Array SSA が適用されている配列 A do
  関数の先頭に undef 関数 (左辺の領域は  $\emptyset$ ) を挿入
end for

```

図 A.1: 配列に対する ϕ 関数の挿入アルゴリズム

```

insertPhiForLoop(L)
  if ループ L の種類が NORMALIZE の場合 then
    for each ループ内で定義または使用されている配列 A do
      ループボディの入り口ブロックの先頭に  $\mu$  関数を挿入
      /* ループ制御変数や, その終値はここで設定する */
    end for
    for each ループ内で定義されている配列 A do
      ループの出口ブロックの先頭に  $\eta$  関数を挿入
      (ループ内の goto 文などによるループの出口は除く)
    end for
  else if ループ L の種類が HAS_BREAK の場合 then
    for each ループ内で定義または使用されている配列 A do
      ループボディの入り口ブロックの先頭に  $\mu$  関数を挿入
      /* ループ制御変数や, その終値はここで設定する */
    end for
    for each ループ内で定義されている配列 A do
      ループ出口ブロックの先頭に undef 関数 (bool が true) を挿入
    end for
  else /* ループ L が NOT_NORMALIZE の場合 */
    for each ループ内で定義されている配列 A do
      ループボディの入り口ブロックの先頭とループ出口
      ブロックの先頭に undef 関数を挿入
    end for
  end if
end if

```

図 A.2: ループでの配列に対する ϕ 関数の挿入のアルゴリズム

```

for each 配列 A do
  /**
   * Stack(A) は  $[A_i, \mathcal{R}_i]$  のように配列変数名と領域のペアを積む
   * UndefStack(A) は Undefined を表す配列変数名を積む
  **/
  Stack(A)  $\leftarrow \perp$ 
  UndefStack(A)  $\leftarrow \perp$ 
end for
call renameVariables(Entry)

renameVariables(X)
  for each 文 S in ブロック X do
    if 文 S の右辺が  $\phi$  関数でない then /*条件文も含む*/
      for each 配列変数 A in 文 S の右辺 do
        右辺の変数 a を Stack(A) のトップで置き換える
      end for
    end if
    if S が配列変数 A の割り当て文 then
      call renameAssign(A, S) /* 割り当て文の名前替え */
    else if S の右辺が配列変数 a に対する  $\delta$  関数の場合 then
      call renameDelta(A, S) /*  $\delta$  関数の名前替え */
    else if S の右辺が配列変数 a に対する  $\pi$  関数の場合 then
      call renamePi(A, S) /*  $\pi$  関数の名前替え */
    else if S の右辺が配列変数 a に対する  $\gamma$  関数の場合 then
      call renameGamma(A, S) /*  $\gamma$  関数の名前替え */
    else if S の右辺が配列変数 a に対する  $\mu$  関数の場合 then
      call renameMu(A, S) /*  $\mu$  関数の名前替え */
    else if S の右辺が配列変数 a に対する  $\eta$  関数の場合 then
      call renameEta(A, S) /*  $\eta$  関数の名前替え */
    else if S の右辺が配列変数 a に対する Undefined 関数の場合 then
      call renameUndef(A, S) /* Undefined 関数の名前替え */
    endif
    for each ブロック  $Y \in \text{succ}(X)$  do
      Y 内の  $\gamma$  関数の引数  $[A, \text{cond}\#\mathcal{R}]$  の  $[A, \mathcal{R}]$  を
      Stack(a) のトップ  $[A_{top}, \mathcal{R}_{top}]$  で置き換える
    end for
    for each ブロック  $Z \in \text{domChild}(X)$  do
      call renameVariables(Z) /* 支配木の上から下の順で名前替え */
    end for
  ブロック X の処理中にスタックに積んだものを全て降ろす

```

図 A.3: 変数の名前替えのアルゴリズム 1

```

renameAssign(A, S)
  for each 配列変数 B in 左辺の配列の添え字式 do
    添え字式内の変数 B を Stack(B) のトップで置き換える
  end for
  左辺の添え字式から配列変数 A の定義領域  $\mathcal{R}_{def}$  を作成
  左辺の A を新しい変数  $A_{new}$  で置き換える
  Stack(A) に  $[A_{new}, \mathcal{R}_{def}]$  を積む

renameUndef(A, S)
  /*  $[A_n, \mathcal{R}_n] = undef(A_{undef}, Undefined, bool)$  */
  if  $\mathcal{R}_n$  が  $\emptyset$  の場合 then
    左辺の配列変数  $A_n$  を新しい変数  $A_{new}$  に置き換える
    UndefStack(A) に  $A_{new}$  を積む
    Stack(A) に  $[A_{new}, \emptyset]$  を積む
  else /*  $\mathcal{R}_n$  が All の場合 */
     $A_{undef}$  を UndefStack(A) のトップで置き換える
    左辺の配列変数  $A_n$  を新しい変数  $A_{new}$  に置き換える
    Stack(A) に  $[A_{new}, All]$  を積む
    if UndefStack(A) のトップの変数が  $\mu$  関数で
      定義されている場合 then
        /* 文 S がループ内にある場合 */
        対応する  $\mu$  関数の引数に  $[A_{new}, All]$  を追加
      end if
    end if
  end if
  if bool が true then
    /**
     * Undef 関数が HAS_BREAK ループの出口に挿入されて
     * いる場合, 対応する  $\mu$  関数の領域の作成を行う
     * 対応する  $\mu$  関数を
     *  $[A_\mu, \mathcal{R}_\mu(i)] = \mu(A_0, (i = 0, e), [A_k, \mathcal{R}_k^\mu(i)], \dots)$ 
     *  $1 \leq k \leq m$  とする.
     */
    for each  $[A_k, \mathcal{R}_k^\mu(i)]$  ( $1 \leq k \leq m$ ) do
      式 4.5 を用いて,  $\mathcal{R}_k^\mu(i)$  を求める
    end for
     $\mathcal{R}_\mu(i)$  を  $(\bigcup_{k=1, m} (\mathcal{R}_k^\mu(i)))$  で置き換える
  end if

```

図 A.4: 変数の名前替えのアルゴリズム 2

```

renameDelta(A, S)
/**
 *  $[A_n, \mathcal{R}_n] = \delta(A_{undef}, [A_{n-2}, \mathcal{R}_{n-2}], [A_{n-1}, \mathcal{R}_{n-1}])$ 
 * Stack(A) のトップを  $[A_{top}, \mathcal{R}_{top}]$  とする
 * Stack(A) の 2 番目の要素を  $[A_{sec}, \mathcal{R}_{sec}]$  とする
 * UndefStack(A) のトップを  $A_{u.top}$  とする
 */
 $\delta$  関数の  $A_{undef}$  を  $A_{u.top}$  に置き換える
 $\delta$  関数の引数  $[A_{n-1}, \mathcal{R}_{n-1}]$  を  $[A_{top}, \mathcal{R}_{top}]$  で置き換える
if  $A_{sec}$  と  $A_{u.top}$  が一致するとき then
   $[A_{n-2}, \mathcal{R}_{n-2}]$  を null とする
   $\mathcal{R}_n$  を  $\mathcal{R}_{top}$  で置き換える
  新しい変数  $A_{new}$  を作成し,  $A_n$  を  $A_{new}$  で置き換える
   $\mathcal{R}_n$  を  $\mathcal{R}_{top}$  で置き換える
  /* 置き換えたものを以下では  $\mathcal{R}_{new}$  と呼ぶ */
else /*  $A_{sec}$  と  $A_{u.top}$  が一致しない場合 */
   $A_{n-2}$  を  $A_{sec}$  で置き換える
   $\mathcal{R}_{n-2}$  を  $(\mathcal{R}_{sec} - \mathcal{R}_{top})$  で置き換える
  新しい変数  $A_{new}$  を作成し,  $A_n$  を  $A_{new}$  で置き換える
   $\mathcal{R}_n$  を  $(\mathcal{R}_{top} \cup \mathcal{R}_{sec})$  で置き換える
  /* 置き換えたものを以下では  $\mathcal{R}_{new}$  と呼ぶ */
end if
Stack(A) に  $[A_{new}, \mathcal{R}_{new}]$  を積む
if  $A_{u.top}$  が  $\mu$  関数で定義されている場合 then
  /* 文 S がループ内にある場合 */
  対応する  $\mu$  関数の引数に  $[A_{new}, \mathcal{R}_{new}]$  を追加
end if

renamePi(A, S)
/**
 *  $[A_n, \emptyset] = \pi(A_{undef}, cond)$ 
 *  $cond$  は  $\phi$  関数の挿入フェーズで設定済みとする
 */
 $A_{undef}$  を Stack(A) のトップの変数で置き換える
新しい変数  $A_{new}$  を作成し,  $A_n$  を  $A_{new}$  で置き換える
UndefStack(A) に  $A_{new}$  を積む
Stack(A) に  $[A_{new}, \emptyset]$  を積む

```

図 A.5: 変数の名前替えのアルゴリズム 3

```

renameGamma(A, S)
/**
 *  $[A_n, \mathcal{R}_n] = \gamma(A_{undef}, [A_{prev}, \mathcal{R}_{prev}], [A_{then}, cond_{then}\#\mathcal{R}_{then}],$ 
 *  $[A_{else}, cond_{else}\#\mathcal{R}_{else}])$ 
 *  $cond_{then}, cond_{else}$  は  $\phi$  関数の挿入フェーズで設定済みとする
 *  $\mathcal{R}_{then}, \mathcal{R}_{else}$  は renameVariables() 内で設定されている
 * if 文の then 部に A の定義がない場合は,  $[A_{then}, \dots]$  は null に
 * else 部に A の定義がない場合は,  $[A_{else}, \dots]$  は null になっている
 **/
 $A_{undef}$  を UndefStack(A) のトップで置き換える
/* Stack(A) のトップを  $[A_{top}, \mathcal{R}_{top}]$  とする */
 $A_{prev}$  を  $A_{top}$  で置き換える
 $\mathcal{R}_{prev}$  を  $(\mathcal{R}_{top} - (cond_{then}\#\mathcal{R}_{then} \cup cond_{else}\#\mathcal{R}_{else}))$  で置き換える
新しい変数  $A_{new}$  を作成し,  $A_n$  を  $A_{new}$  で置き換える
 $\mathcal{R}_n$  を  $(\mathcal{R}_{top} \cup cond_{then}\#\mathcal{R}_{then} \cup cond_{else}\#\mathcal{R}_{else})$  で置き換える
/* 置き換えたものを以下では  $\mathcal{R}_{new}$  と呼ぶ */
Stack(A) に  $[A_{new}, \mathcal{R}_{new}]$  を積む
if UndefStack(A) の先頭の変数の定義文が  $\mu$  関数の場合 then
  対応する  $\mu$  関数の引数に  $[A_{new}, \mathcal{R}_{new}]$  を追加する
end if

renameMu(A, S)
/**
 *  $[A_n, \mathcal{R}_n(i)] = \mu(A_{undef}, (i = 0, e), [A_k, \mathcal{R}_k^n(i)], \dots)$ 
 * この関数が呼ばれた時点では,
 *  $\mu$  関数の引数が決定していないので,
 * ここでは名前替えのみ行う
 **/
 $A_{undef}$  を UndefStack(A) のトップで置き換える
新しい変数  $A_{new}$  を作成し,  $A_n$  を  $A_{new}$  で置き換える
UndefStack(A) に  $A_{new}$  を積む
Stack(A) に  $[A_{new}, \emptyset]$  を積む

```

図 A.6: 変数の名前替えのアルゴリズム 4

```

renameEta(A, S)
/**
 *  $[A_n, \mathcal{R}_n] = \eta(A_{undef}, [A_{prev}, \mathcal{R}_{prev}], [A_{loop}, \mathcal{R}_{loop}])$ 
 * この手続きでは, 対応する  $\mu$  関数の領域の作成と,
 *  $\eta$  関数の名前替えと領域の作成を行う
 * 対応する  $\mu$  関数を
 *  $[A_\mu, \mathcal{R}_\mu(i)] = \mu(A_0, (i = 0, e), [A_k, \mathcal{R}_k^\mu(i)], \dots)$ 
 *  $1 \leq k \leq m$  とする.
 **/
/* Step 1:  $\mu$  関数の領域の作成 */
for each  $[A_k, \mathcal{R}_k^\mu(i)] (1 \leq k \leq m)$  do
  式 4.5 を用いて,  $\mathcal{R}_k^\mu(i)$  を求める
end for
 $\mathcal{R}_\mu(i)$  を  $(\bigcup_{k=1,m}(\mathcal{R}_k^\mu(i)))$  で置き換える
/* 以下, 置き換えたものを  $\mathcal{R}_\mu(i)$  と呼ぶ */
/* Step 2:  $\eta$  関数の名前替えと領域の作成 */
 $A_{undef}$  を  $\text{UndefStack}(A)$  のトップで置き換える
 $A_{loop}$  を  $A_\mu$  で置き換える
 $\mathcal{R}_{loop}$  を  $(\bigotimes_{i=0,n}^\cup(\mathcal{R}_\mu(i)))$  で置き換える
/*  $\text{Stack}(A)$  の先頭を  $[A_{top}, \mathcal{R}_{top}]$  とする */
 $A_{prev}$  を  $A_{top}$  で置き換える
 $\mathcal{R}_{prev}$  を  $(\mathcal{R}_{prev} - (\bigotimes_{i=0,n}^\cup(\mathcal{R}_\mu(i))))$  で置き換える
新しい変数  $A_{new}$  を作成し,  $A_n$  を  $A_{new}$  で置き換える
 $\mathcal{R}_n$  を  $(\mathcal{R}_{prev} \cup (\bigotimes_{i=0,n}^\cup(\mathcal{R}_\mu(i))))$  で置き換える
/* 以下, 置き換えたものを  $\mathcal{R}_{new}$  と呼ぶ */
 $\text{Stack}(A)$  に  $[A_{new}, \mathcal{R}_{new}]$  を積む
if  $\text{UndefStack}(A)$  のトップの変数の定義先が  $\mu$  関数の場合 then
  その  $\mu$  関数の引数に  $[A_{new}, \mathcal{R}_{new}]$  を追加する
end if

```

図 A.7: 変数の名前替えのアルゴリズム 5

付録B HIRにSSA形式を導入する際の問題点

COINSの高水準中間表現を導入するに当たって、いくつかの問題点があった。付録として、この問題点と本研究での解決策について説明する。

問題点1

SSA形式では、ループの繰返し条件の直前に ϕ 関数が挿入される場合がある。例えば次のプログラムを考える。

プログラム B.1 (ループの例)

```
1:  $i = 0$ ;  
2:  $while(i < 10)\{$   
3:    $i = i + 1$ ;  
4:  $\}$ 
```

これをSSA形式に変換したもののフローグラフは図B.1のようになる。これを見ると、ブロックlab2のwhile文の条件式の直前に ϕ 関数が挿入されていることがわかる。

このようにSSA形式では、ループの繰返し条件式の直前に ϕ 関数が挿入することがあるが、COINSのHIR上のループの仕様では、この式を挿入する場所がない。

プログラムB.1のブロックlab2の通常形式のHIRは、図B.2のようになる(見やすくするため、一部簡略化してある)。これはlabel付きの文を表すLabeledStmtの中に、条件式を表すExpStmtが入っていることを表している。このときHIRのループ条件式の仕様では、LabeledStmtの中にExpStmt以外のものを入れることは禁止されている。

本研究では、LabeledStmtの中に文の列を表すBlockStmtを入れることを許すことによってこれを実現した。本研究での図B.1のブロックlab2のSSA形式のHIRは図B.3のようになる。

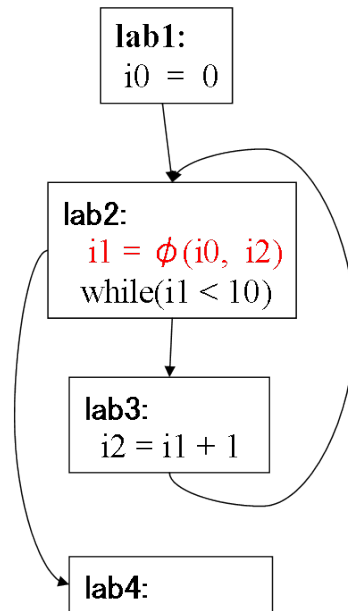


図 B.1: プログラム B.1 の SSA 形式

```

(labeldSt (list <labelDef _lab2>
  (expStmt ( i < 10))
)
)
  
```

図 B.2: 図 B.1 の lab2 の HIR(通常形式の場合)

```

(labeldSt (list <labelDef _lab2>
  (block
    (assign i_1 (phi (i_0 : lab1)(i_2 : lab3)))
    (expStmt ( i_1 < 10))
  )
)
)
  
```

図 B.3: 図 B.1 の lab2 の HIR(SSA 形式の場合)

問題点 2

SSA 形式から通常形式への変換や、ある種の最適化を行う際には、危険辺 (Critical Edge) に文を挿入する必要がある。危険辺とは、 $|succ(B1)| > 1$ と成るような基本ブロック B1 から $|pred(B2)| > 1$ となるような基本ブロック B2 へのエッジ $B1 \rightarrow B2$ のことである。危険辺の例を図 B.4 に示す。図の例では、 $P1 \rightarrow S2$ が危険辺となって

いる.

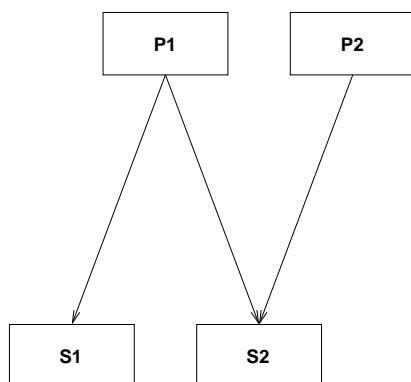


図 B.4: 危険辺

危険辺は, その辺にあらたな基本ブロックを挿入することにより除去することができる (edge split)[18]. 例えば図 B.4 の例では, 危険辺 $P1 \rightarrow S2$ に対して新しい基本ブロック $N1$ を作成し, $P1 \rightarrow N1 \rightarrow S2$ となるように挿入することにより危険辺を除去できる. 図 B.4 の危険辺を除去したものを図 B.5 に示す.

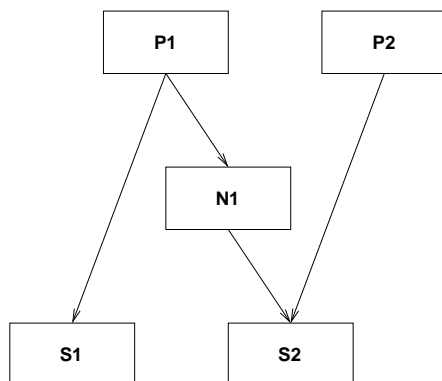


図 B.5: 危険辺除去

SSA 形式から通常形式への逆変換では, ϕ 関数を除去するために, 危険辺にコピー文を挿入しなければならない場合がある. 例えば図 B.6 のプログラムを考える. ここでエッジ $B2 \rightarrow B6$ が危険辺となっている.

図 B.6 を SSA 逆変換したものは図 B.7 のようになる.

このとき図 B.6 のブロック $B6$ にある ϕ 関数を除去するために, 危険辺に新しい基本ブロック $B7$ を作成し, そこに $a_3 = a_1$ というコピー文を挿入している.

HIR では上記の例のような, ループの繰返し条件式を含む基本ブロックから, ループ出口へのエッジが危険辺となる場合, 図 B.7 のように基本ブロック $B7$ を作成してコピー文を挿入するということが出来ない.

本研究では, 上記の例のようにループの危険辺に文を挿入する場合は, ループを表すクラス内にコピー文を記憶するフィールドを用意しておき, HIR から低水準中間

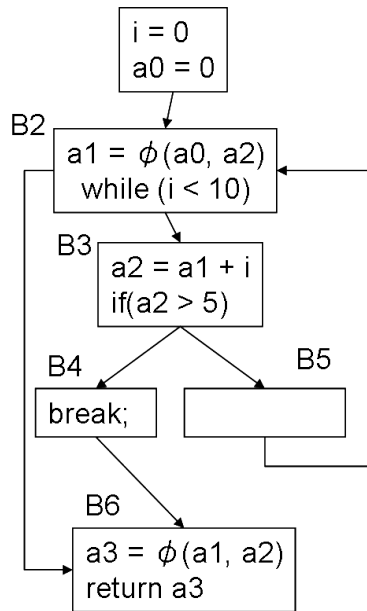


図 B.6: サンプルプログラム

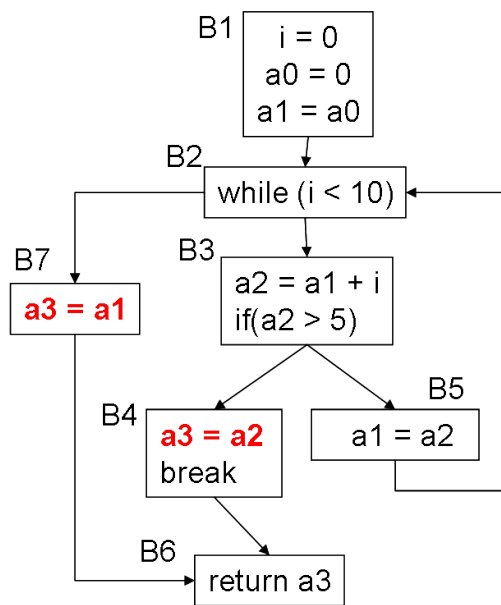


図 B.7: 図 B.6 に SSA 逆変換を適用

表現 LIR に変換する際に, 実際にプログラム内に追加するようにした.