静的単一代入形式最適化システム
および関連ユーティリティ

外部仕様書

２０１３年２月

東京工業大学
大学院情報理工学研究科
佐々政孝研究室

目 次

図 目 次

表 目 次

第1章　概要

静的単一代入形式最適化システム (以下本システム) は, 文部科学省科学技術振興調整費「並列化コンパイラ向け共通インフラストラクチャの研究」の一環として, 静的単一代入形式 (SSA 形式 : Static Single Assignment form) に基づく最適化を行うものである. 本システムは並列化コンパイラ向け共通インフラストラクチャ(以下 COINS) の低水準中間表現 (LIR : Low-level Intermediate Representation) を対象としたものである.

本システムは, COINS に含まれるさまざまな最適化や解析と同様, コンパイラドライバから起動される. コンパイラドライバから起動される最適化や解析等の処理は, COINS では 'パス' と呼んでいる. 本システムもパスのひとつであり, 以下の処理を行う.

- COINS コンパイラドライバから LIR を受け取る
- SSA 形式上での最適化を行う (*)
- COINS コンパイラドライバに最適化された LIR を返却する

(*) の部分は, さらに以下のように細分化される.

1. LIR 上での通常形式から SSA 形式への変換
2. SSA 形式上での最適化
3. LIR 上での SSA 形式から通常形式への変換

通常形式から SSA 形式への変換は Cytron らが提案した dominance frontier を用いたものを実装した [12]. SSA 形式にはいくつか種類が存在する. 本システムでは COINS コンパイラドライバへのオプションを用いて, mimimal SSA, semi-pruned SSA, そして pruned SSA への変換が選択可能である [6]. また通常形式から SSA 形式への変換時に Copy Folding と無用な ϕ 関数を除去することが可能である. これは COINS コンパイラドライバへのオプションを用いて選択的に実行することができる. コンパイラ上である種のループ最適化を行う際には, その最適化を行う前にループの構造を変換したほうが最適化の効果が上がる場合がある. しかし SSA 形式上でのループ構造の変換は ϕ 関数の扱いが難しい. そこで, 本システムでは SSA 形式への変換直前の通常形式上でこの変換を行う. これは COINS コンパイラドライバへのオプションを用いて選択的に実行することができる. 通常形式から SSA 形式への変換の詳細は第 4 章で述べる.

SSA 形式から通常形式への変換は Sreedhar らが提案したものを実装した [16]. Sreedhar らは 3 種類のアルゴリズムを提案しており, 本システムではそれら全て (Method I, Method II, Method III) を実装した. これらのアルゴリズムは COINS コンパイラドライバへのオプションを用いて選択することが可能である. また, SSA 形式から通常形式への変換の途中および通常形式に変換後にコアレシングを行って無駄なコピー文を除去することができる. これらは COINS コンパイラドライバへのオプションを用いて選択的に実行可能である. SSA 形式から通常形式への変換の詳細は第 5 章で述べる.

SSA 形式上での最適化および変換は以下のものを実装した.

- コピー伝播
- 共通部分式除去
- 質問伝播に基づく大域値番号付けと部分冗長除去
- ループ不変式のループ外追い出し
- 帰納変数の演算の強さの軽減と判定の置き換え
- 条件分岐を考慮した定数伝播
- 無用コード除去
- SSA グラフの作成と変数の等価性を見つけるアルゴリズム
- 3 番地コードへの変換
- 空の基本ブロック除去
- 基本ブロックの結合
- 危険辺除去
- Global Reassociation
- 無用 ϕ 命令除去

上記の各最適化は, COINS コンパイラドライバへのオプションの指定により, 任意のものを任意の回数, 任意の順序で実行可能である. SSA 形式上での最適化の詳細は第 6 章で述べる.

本システムは SSA 形式での最適化の対象を仮想レジスタとし, ひとつのレジスタに対する代入を静的に単一とすることにより SSA を実現している. しかし一般的な C 言語プログラムでは, 配列やポインタで指される値など, ひとつの値が複数の別名を持つ場合がある. SSA 形式における最適化では別名が存在するものの扱いが難しい. そこで本システムでは, 別名解析の第一近似として, プログラム中で参照されるメモリをひとつの大きなオブジェクトとして扱う方式を実装した. この簡単な別名解析の詳細は第 7 章で述べる.

本システムでは LIR を入力とし, 出力もまた LIR である. しかし, 最適化を行ったコードが正しく実行できるかどうか, 最適化による LIR の変換が意味的に正しいかどうかを LIR 上で確認することは困難である. そこで本システムでは LIR を, それに対応する C プログラムに変換する変換器を実装した. これにより LIR プログラムを, それに相当する C プログラムに変換することができる. LIR から C プログラムへの変換の詳細は第 8 章で述べる.

本システムで利用できる COINS コンパイラドライバへのオプションについては付録 A を参照してもらいたい. なお, COINS 全般については [10] を参照されたい.

第2章　諸定義と前提知識

2.1　諸定義

本仕様書では以下の記号を使う. また, 一般的な数学の記号も使用する.

= この記号で結ばれる変数や集合が等しいことを示す. 例えば a=b と書いた場合は a と b が等しい. ただし, C プログラムを記述している場合などは代入を示す場合がある.

← この記号で結ばれる変数や集合に関して, 記号の左側の変数や集合に, 右側の変数や集合を代入することを示す. 例えば a←b と書いた場合は a に b を代入する.

⟶ この記号で結ばれるものの間にエッジが存在することを示す. 例えば a⟶b は, a を始点とし b を終点とするエッジを表す.

2.2　前提知識

SSA 形式上での最適化について述べる前に, コンパイラや最適化一般で広く用いられている用語や概念についての本仕様書での定義を述べる. なお, ここで述べる定義を実現するアルゴリズム等は [1, 19, 3] を参照してもらいたい.

2.2.1　基本ブロック

基本ブロック (Basic Block) とはプログラムの一部分 (ブロック) 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, プログラムを視覚的に表現する方法として現在広く用いられている [1, 19, 3]. 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.

2.2.2　制御フローグラフ

制御フローグラフ (Control Flow Graph： CFG) とは, プログラムの入口から出口までの基本ブロックを, 制御の流れにそって図にしたグラフである [1, 19, 3]. CFG の節点 (ノード) は基本ブロックであり, ノード間は有向辺 (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.

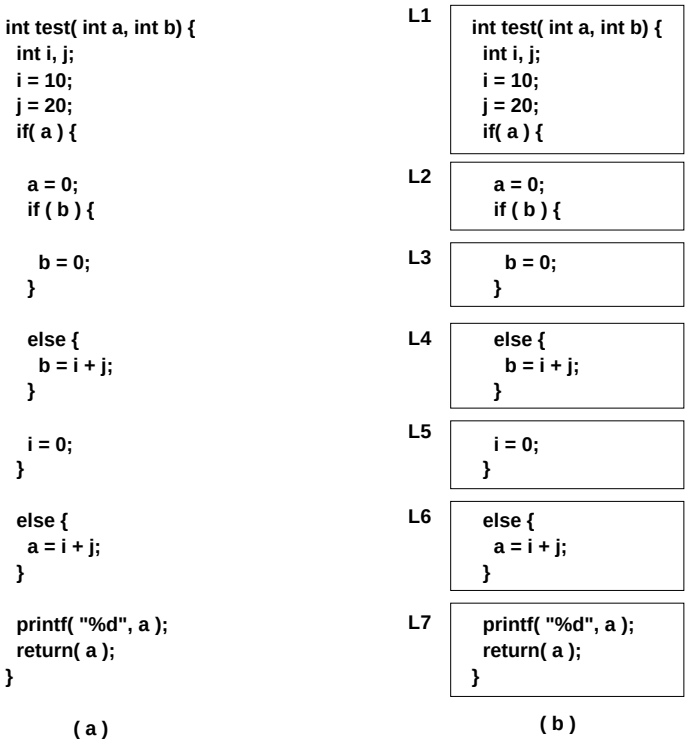


図 2.1: 基本ブロック

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. 例えば図 2.2 の各ノードの $pred()$ と $succ()$ は以下のようになる.

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

2.2.3 支配関係・後支配関係

支配関係

CFG 上のふたつのノード X, Y について, CFG の入口から 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, 19, 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,

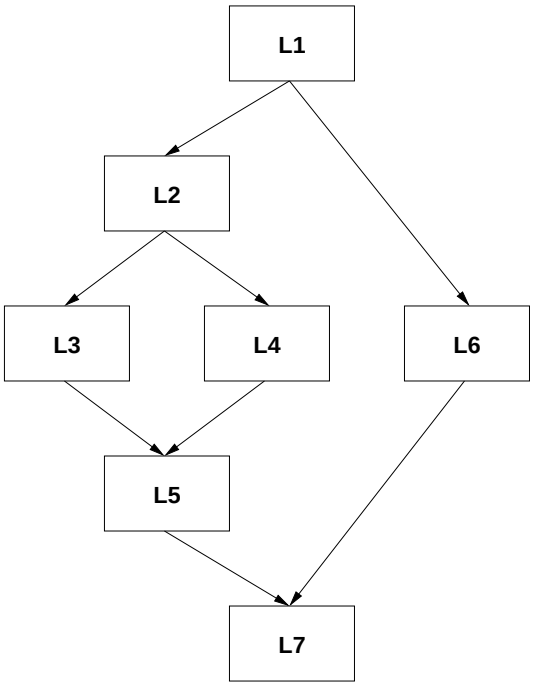


図 2.2: 制御フローグラフ (CFG)

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 [19]. 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≠Y である場合, X は Y を厳密に支配 (strictly dominate) するという [1, 19, 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, 19, 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.

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

後支配関係

CFG 上のふたつのノード X, Y について, CFG の出口から Y に達するどの路も必ず X を通る場合に, X は Y を後支配 (postdominat) するという [1, 19, 3]. また, X は Y の後支配ノード (postdominator) であるという. 以下では X の後支配ノードの集合を $pdom(X)$ とする. 後支配関係は反射的 (reflexive) である. すなわち, ノード X は自分自身を後支配する. また, 後支配関係は推移的 (transitive) である. すなわち, ノード X が ノード Y を後支配し, ノード Y が ノード Z を後支配するのであれば, ノード X は ノード Z を後支配する. 例えば図 2.2 の CFG では以下のような後支配関係がある.

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

ノード X が ノード Y を後支配し, X≠Y である場合, X は Y を厳密に後支配 (strictly postdominate) するという [1, 19, 3]. 例えば図 2.2 の L5 は L2 を厳密に後支配している. 以下, X を厳密に後支配するノードの集合を $spdom(X)$ と書く.

ノード X が ノード Y を厳密に後支配し, X から Y への路に, X 以外に Y を厳密に後支配するノードがないとき, X は Y を直接後支配 (immediately postdominate) するという [1, 19, 3]. 以下では X が Y を直接後支配するときに $X = ipdom(Y)$ と書く. 例えば図 2.2 の CFG では以下のような直接後支配の関係がある.

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

2.2.4 支配木・後支配木

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 $Y \in domChild(X)$ と書く. 図 2.2 の CFG に対応した支配木を図 2.3 に示す.

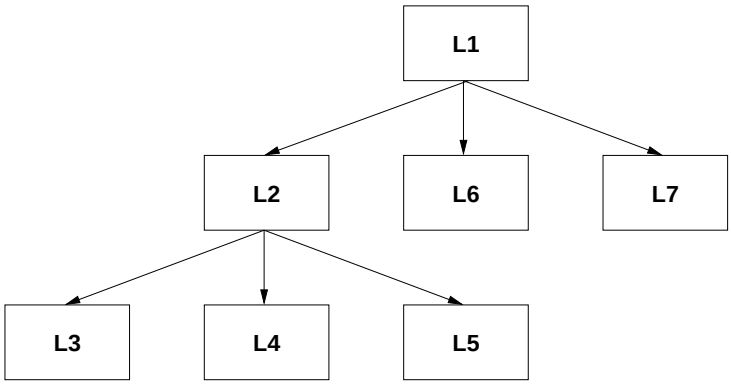


図 2.3: 支配木

後支配木

後支配木 (postdominator tree) とは, CFG 上のあるノード X と, X を直接後支配するノード 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 を直接後支配するノードはひとつだからである. 後支配木の根は CFG の出口ノードである. プログラムによっては, その CFG に出口ノードが存在しない場合がある. その場合は, プログラムの入口から出口への仮想的なエッジを用いて後支配木を作成する. 以下, 後支配木のノード X の子ノード Y を $Y \in pdomChild(X)$ と書く. 図 2.2 の CFG に対応した後支配木を図 2.4 に示す.

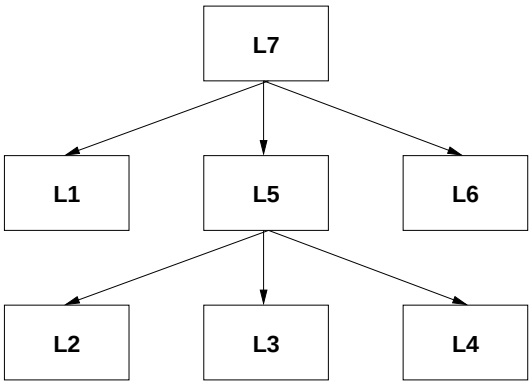


図 2.4: 後支配木

2.2.5 支配辺境・後支配辺境

支配辺境

支配辺境とは, 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 の支配辺境 $DF(X)$ は以下のように定義される [19].

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

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, $Y \in succ(X)$ 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≠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 に至る際にかならず U を通る場合は図 2.5 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.

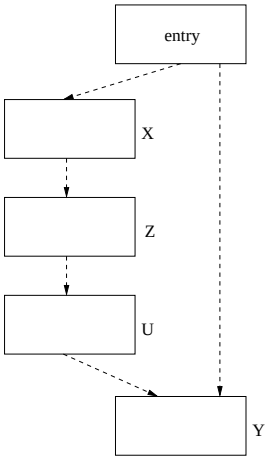


図 2.5: X の支配辺境は Y であり Z の支配辺境も Y の例

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 2.6 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 に到達する際には必ず 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.

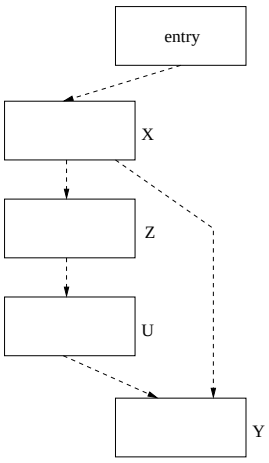


図 2.6: Z の支配辺境は Y であり X の支配辺境は Y ではない例

以上のことから, X の支配辺境は以下の Y の集合といえる.

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

例えば図 2.2 の CFG の各ノードに対する支配辺境は以下のようになる.

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

後支配辺境

後支配辺境とは, 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 の後支配から外れたノード Y の集合である. ノード X の後支配辺境 $PDF(X)$ は以下のように定義される [19].

$$PDF(X) = \{Y | U \in succ(Y) \text{ が存在し}, X \text{ は } U \text{ を後支配し}, X \text{ は } Y \text{ を厳密に後支配はしない } \}$$

支配辺境と同様, X の後支配辺境は以下の Y の集合といえる.

- $Y \in pred(X)$ であり, $X \neq ipdom(Y)$
- $Z \in pdom(X)$ であるときに $Y \in PDF(Z)$ であり, $X \neq ipdom(Y)$

例えば図 2.2 の CFG の各ノードに対する後支配辺境は以下のようになる.

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

2.2.6 自然なループ

最適化の対象として最も重要なもののひとつがループである. CFG 上のループ構造は, CFG ノード間の支配関係を利用して定義できるもの [1, 3, 19] と, そうでないものに分類される. 自然なループとは支配関係を用いて求められるループである.

戻り辺 有向辺 b⟶h において, b を始点, h を終点としたとき, CFG 中の有向辺の中で終点が始点を支配するものを戻り辺 (back edge) という.

戻り辺 b⟶h が与えられたとき, h を通らないで b に到達できるノードの集合と h を加えたものを, その辺の自然なループと定義する. h はループのヘッダ (入口ノード) である. 図 2.7 に自然なループの例を示す.

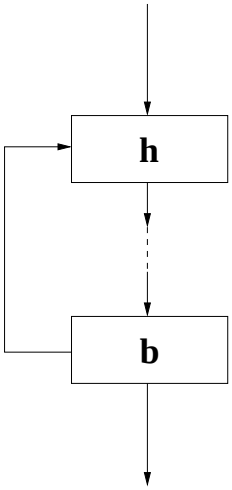


図 2.7: 自然なループ

自然なループは, 本質的に次の性質をもつ [1].

- ループは, ヘッダと呼ばれるただひとつの入口ノードを持たなければならない. このヘッダノードはループ中の全てのノードを支配する. もしそうでないとすると, ヘッダはループのただひとつの入口にならなくなってしまう.
- ループの繰り返し方は必ずひととおりは存在する. すなわちループにはヘッダへ戻るエッジが必ずひとつは存在しなければならない.

以下, ヘッダに h を持ち, 戻り辺を持つブロック b を持つループを $LOOP(h, b)$ と書く.

2.2.7 プリヘッダ

多くのループ最適化では, ある命令列 (例えばループ不変式) をループ実行前の部分に挿入する. しかしループヘッダ h に対して, ループ外の複数ノードからエッジがある場合には, それらすべてのノードに命令列を挿入しなければならない. そこでそのような命令列の挿入を一箇所にまとめるために, プリヘッダと呼ばれる空のノード p を p⟶h となるようにループの外に挿入する. 図 2.8 にプリヘッダの例をあげる. 図中の矩形は基本ブロックであり, 矩形中の h はヘッダノードを, b は戻り辺のあるノードを, p はプリヘッダノードをそれぞれ表している. 図 2.8 の (a) で示されているループは, ヘッダに対してループ外の複数のノードからエッジがある. そこでプリヘッダ p を (b) のように挿入することによって, ループ外からヘッダへのエッジをひとつにすることができる.

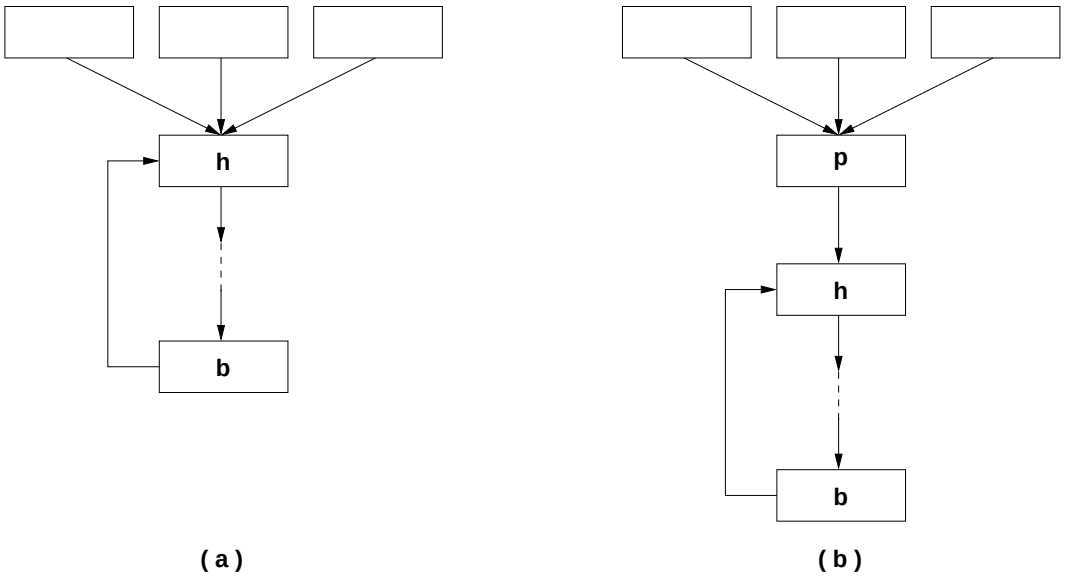


図 2.8: プリヘッダ

2.2.8 制御依存

CFG 上のふたつのノード X, Y について, 次のふたつの条件が成り立つとき Y は X に制御依存 (Control Dependent) するという [19].

1. X から Y への空でない路があり, Y はその路の X より後ろの全てのノードを後支配する.
2. Y は X を厳密に後支配しない. すなわち $Y \notin spdom(X)$.

制御依存は, 別の言葉でいえば, X からあるエッジをたどると, その後必ず Y を通るが, 別のエッジをとれば Y は通らないことを表す. すなわち Y が X に制御依存するということは, $X \in PDF(Y)$ と等価である. 図 2.9 に制御依存の例を示す. この例では Y が X に制御依存している.

2.2.9 データ依存

CFG 上のあるノード X において使用される変数 a が, X に含まれる文で a を使用する前で定義されず, かつ a が CFG の入口から X へ至るパス上のノード Y で定義されている場合, X は Y にデータ依存するという. この場合, X にある a を使用する式は, Y を越えて移動することはできない. 図 2.10 にデータ依存の例を示す. この例では X が Y にデータ依存している.

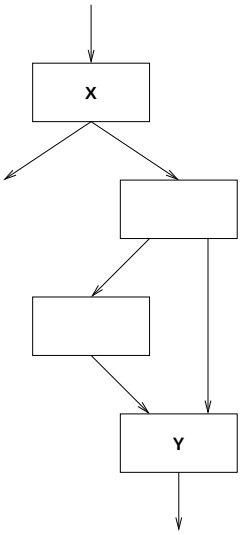


図 2.9: 制御依存

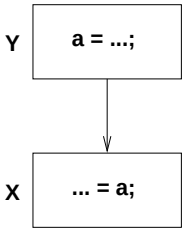


図 2.10: データ依存

第3章　SSA 変換前の処理

SSA 形式上での最適化の効果を向上させるために, 最適化の前に CFG に対してある種の変換をすることが望まれる場合がある. そのような変換は, CFG に対して新たにノードを追加するなど, CFG の構造を変更してしまう場合がある. SSA 形式では, CFG の構造を考慮して ϕ 関数が挿入されているので, CFG の構造を変更することは困難である. そこで本システムではいくつかの変換を, プログラムを SSA 形式に変換する前に行うようにした.

3.1　ループ構造の変換

コンパイラ上である種のループ最適化を行う際には, その最適化を行う前にループの構造を変換したほうが最適化の効果が上がる場合がある. 例えば, ループ不変式のループ外追い出しの最適化を安全に行うには, 不変式があるブロック (CFG のノード) は, 対象となるループの全ての出口ブロックを支配している必要がある [1](6.4 参照). この条件の下では, ループのヘッダブロックと出口ブロックが同じであるループ, つまり while 型のループ, に対してループ不変式の追い出しを行おうとしても, ヘッダブロック以外のループ構成ブロックから不変式を追い出すことができない. しかし while 型のループを, do-while 型のループに等価変換することによりヘッダブロック以外のブロックからもループ不変式をループ外に追い出すことができる.

本システムでは, ループの構造変換として以下のものを実装した.

- 共通のヘッダを持つ入れ子でないループのマージ [19, 1]
- while 型ループから do-while 型ループへの変換 [19, 1, 3]

ループの構造変換はプログラムの制御構造を変更する. SSA 形式上では ϕ 関数という, 制御の流れとデータの流れの両方を持つ疑似コードが挿入されており, プログラムの制御構造を変更するとそれの扱いが難しい. よって, 以下で述べるループの構造変換は SSA 形式への変換直前の通常形式上で行うことにする.

3.1.1　共通のヘッダを持つ入れ子でないループのマージ

2.2.6 節で述べたループの定義では, ヘッダが異なるふたつの自然なループ LOOP(h,b) と LOOP(h',b') は, 全く共通部分がないか, あるいはどちらか一方が他方を含む, すなわち入れ子になっている, かのどちらかである. このことは自然なループの定義から明らかである.

しかしこの定義には, 共通部分を持つが入れ子になっていないループは別々のループとして認識されてしまうという問題点がある. 例として図 3.1 をあげる.

図 3.1 では

$$dom(B3) = \{B1, B2, B3\}, succ(B3) = \{B1\}$$

であり,

$$dom(B4) = \{B1, B2, B4\}, succ(B4) = \{B1\}$$

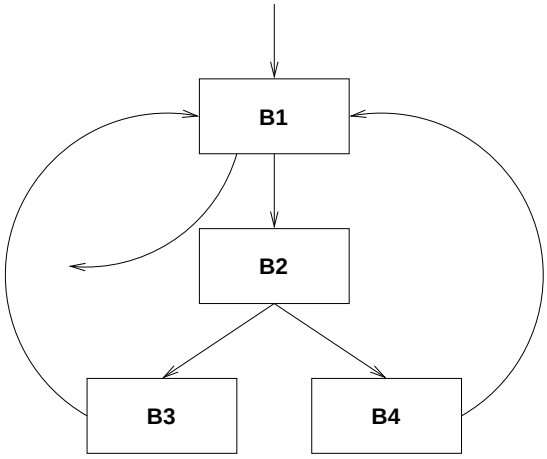


図 3.1: ヘッダが共通なループ

となるので, 自然なループの定義から

$$LOOP(B1, B3), LOOP(B1, B4)$$

というふたつのループが求められる. しかしこれらふたつのループは入れ子になっておらず, かつ共通なヘッダを持っている. このようなふたつのループを別々に最適化の対象とするのは困難である.

そこで, このようなふたつのループはひとつのループにまとめることにする [19, 1]. 図 3.1 の例を用いると, 新しい空の基本ブロック Bt を作成し, 戻り辺であった $B3 \longrightarrow B1$ と $B4 \longrightarrow B1$ をそれぞれ $B3 \longrightarrow Bt$ と $B4 \longrightarrow Bt$ とする. そして新たに $Bt \longrightarrow B1$ という戻り辺を作成する. この作業を行ったものを図 3.2 に示す.

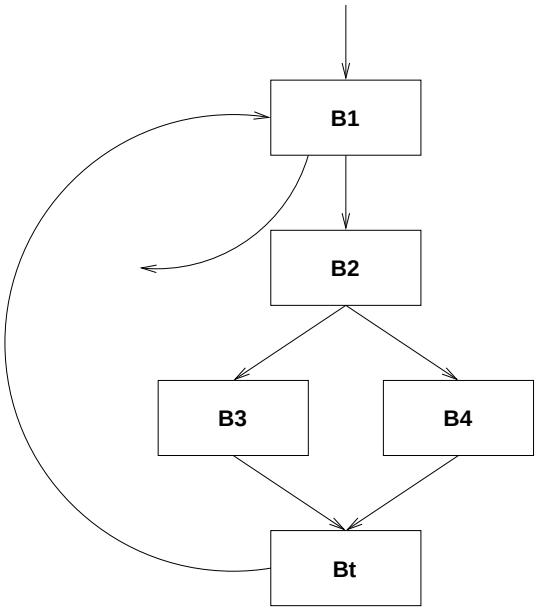


図 3.2: ヘッダが共通なループをマージしたループ

これにより, 本来あった LOOP(B1,B3) と LOOP(B1,B4) のふたつのループが LOOP(B1,Bt) というひとつのループにマージされたことがわかる.

3.1.2 while 型ループから do-while 型ループへの変換

while 型のループから do-while 型のループへの変換は以下の手順で行う.

1. ヘッダブロック H のコピーであるブロック C を作成
2. ブロック C をループの戻り辺に挿入

while 型のループ例として, 図 3.3 のプログラムをあげる. 図 3.3 のプログラムの制御フローグラフは図 3.4 となる. 制御フローグラフの図中にある矩形はブロックを表している. 矩形内部にある文字列は疑似的な命令列を表しており, 矩形の横にある Li (i は数字) はブロックの番号を表している.

```
int main(void){
  int i=0,j=0,sum=0;
  int a=10,b=20;
  int x=0,y=0;
LAB001:
  i=i+1;
  j=i+j+1;
  if(i<10 && j<30){
    sum=i+j;
    x=a*b;
    y=a+b;
    goto LAB001;
  }
  i=i+x;
  j=j+y;
  printf("%d,%d\n",i,j);
}
```

図 3.3: while 型ループの例

図 3.4 の例では, ループはブロック L1, L2, L3 から構成されており, そのヘッダは L1, 出口は L1 と L2 である. このループは, L1 がヘッダであり出口であるので, ループ変換を行うと最適化の効果が向上すると期待される. そこで, L1 に含まれる命令列と, その successor をコピーしたブロック L1' を作成し, それをループの戻り辺であるエッジ $L3 \longrightarrow L1$ が $L3 \longrightarrow L1'$ となるように挿入する. (図 3.5). この変換を繰り返すことにより while 型のループを do-while 型のループに変換することができる (図 3.6).

実装上の制限

LIR の JUMPN 命令がループヘッダの末尾にある場合は, do-while 型ループへの変換を行わない.

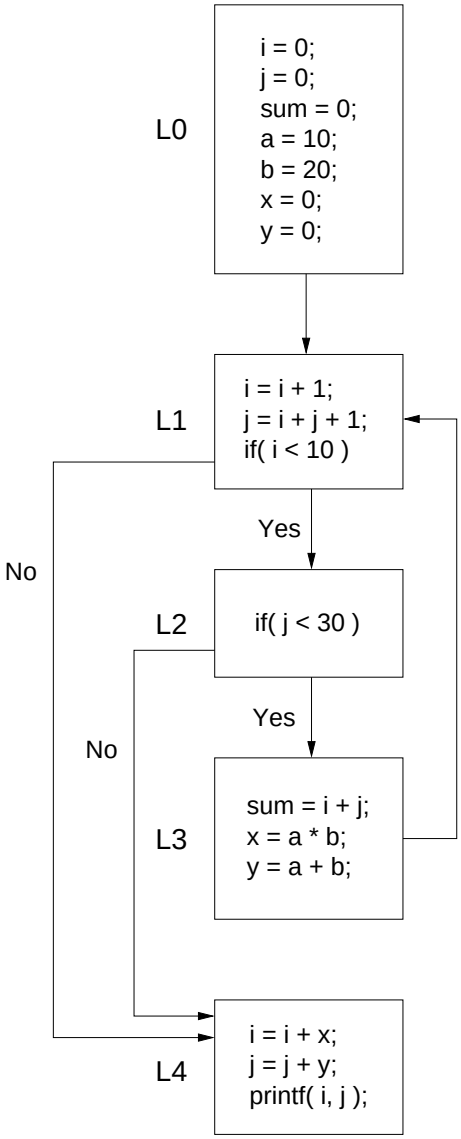


図 3.4: while 型ループの例：制御フローグラフ

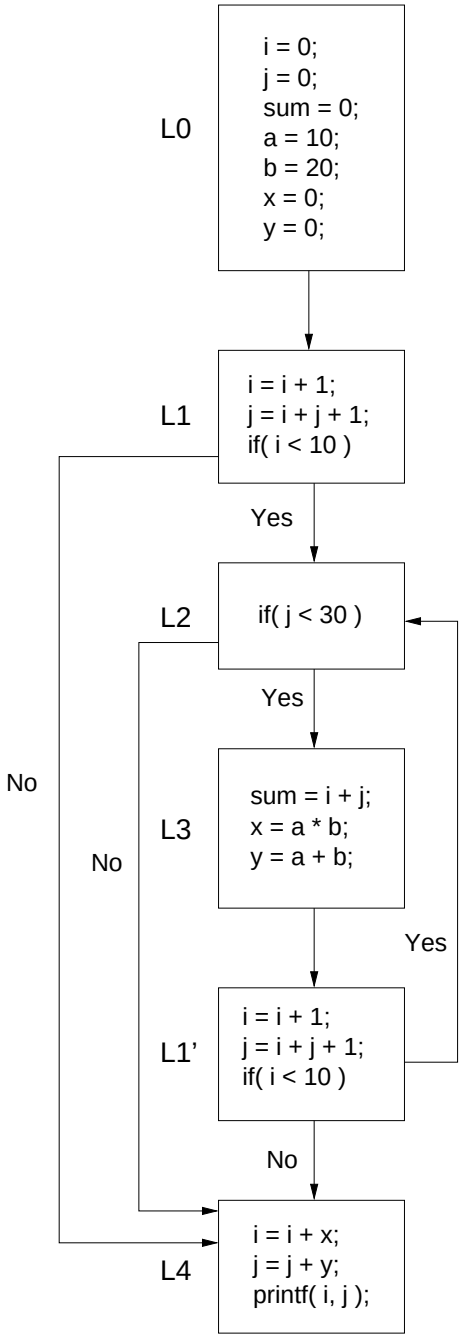


図 3.4 の CFG からブロック L1' が追加された

図 3.5: while 型ループの例：制御構造変更 (1)

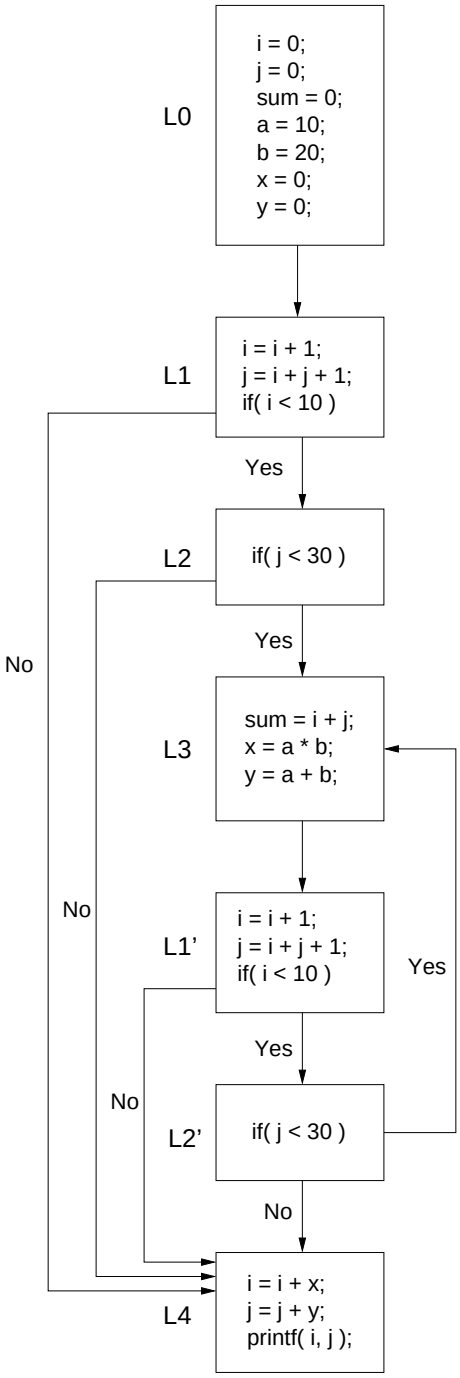


図 3.5 の CFG からブロック L2' が追加された

図 3.6: while 型ループの例：制御構造変更 (2)

3.2 実装ノート

LIR の IF ノードは，SSA 変換前に行うループ構造の変換の前に，JUMPC ノードを用いた等価なものに変換している．したがって，SSA 形式 LIR には IF ノードは現れない．

第4章　通常形式から SSA 形式への変換

本システムで実装した SSA 形式への変換アルゴリズムは, Cytron らが提案した支配辺境を用いるものである [12]. CFG に対する支配木が与えられているとき, 本システムが実装した SSA 形式への変換アルゴリズムの手順は以下の通りである.

1. CFG の各ノードの支配辺境を求める
2. 支配辺境をもとに, 通常形式の各変数について ϕ 関数を挿入するノードを求め, ϕ 関数を挿入する
3. 各変数の名前替えを行う

本システムでは 2 のフェーズにおいて, 異なる精度の変数の生存区間解析を用いることにより 3 種類の SSA 形式を出力することができる. これらは COINS コンパイラドライバへのオプションを用いて選択することができる.

4.1 ϕ 関数の挿入

SSA 形式には minimal SSA, pruned SSA, そして semi-pruned SSA の 3 種類が広く知られている [6, 19]. これら 3 種類の SSA 形式の違いを生じさせるのが, 本節で述べる ϕ 関数挿入の際である. 基本となる ϕ 関数挿入アルゴリズムは minimal SSA のものであり, pruned SSA や semi-pruned SSA は, minimal SSA の ϕ 関数挿入アルゴリズムに, 異なる精度の変数の生存区間解析を加えたものである. 図 4.1 に ϕ 関数挿入のアルゴリズムを示す. また, 3 種類の SSA 形式の違いを図 4.2 に示す.

```
for each ブロック X do
   Inserted(X) と Work(X) の初期化
end for
W ← 空集合
for each 変数 V do
   for each V の定義があるブロック X do
      if semi-pruned SSA に変換 then
         if V が “non-local” である then
            W に X を加える
            Work(X) ← V /* 変数 V のために W に X を加えたしるし */
         end if
      else /* semi-pruned SSA 以外への変換 */
         W に X を加える
         Work(X) ← V /* 変数 V のために W に X を加えたしるし */
      end if
   end for
   while W が空集合でない do
      W からあるブロック X を取り除き, その X について
      for each Y ∈ DF(X) do
         if Inserted(Y) ≠ V then
            /* Y にまだ V の φ 関数を挿入していない */
            if Pruned SSA に変換 then
               if 変数 V が Y に Live In している then
                  Y に “V ← φ(V, ...., V)” を挿入
                  Inserted(Y) ← V /* Y に V の φ 関数を挿入した*/
               end if
            else /* Pruned SSA 以外への変換 */
               Y に “V ← φ(V, ...., V)” を挿入
               Inserted(Y) ← V /* Y に V の φ 関数を挿入した*/
            end if
            if Work(Y) ≠ V then
               /* まだ V のための W に Y を加えていない */
               W に Y を加える
               Work(Y) ← V /* V のために W に Y を加えた */
            end if
         end if
      end for
   end while
end for
```

図 4.1: ϕ 関数挿入のアルゴリズム

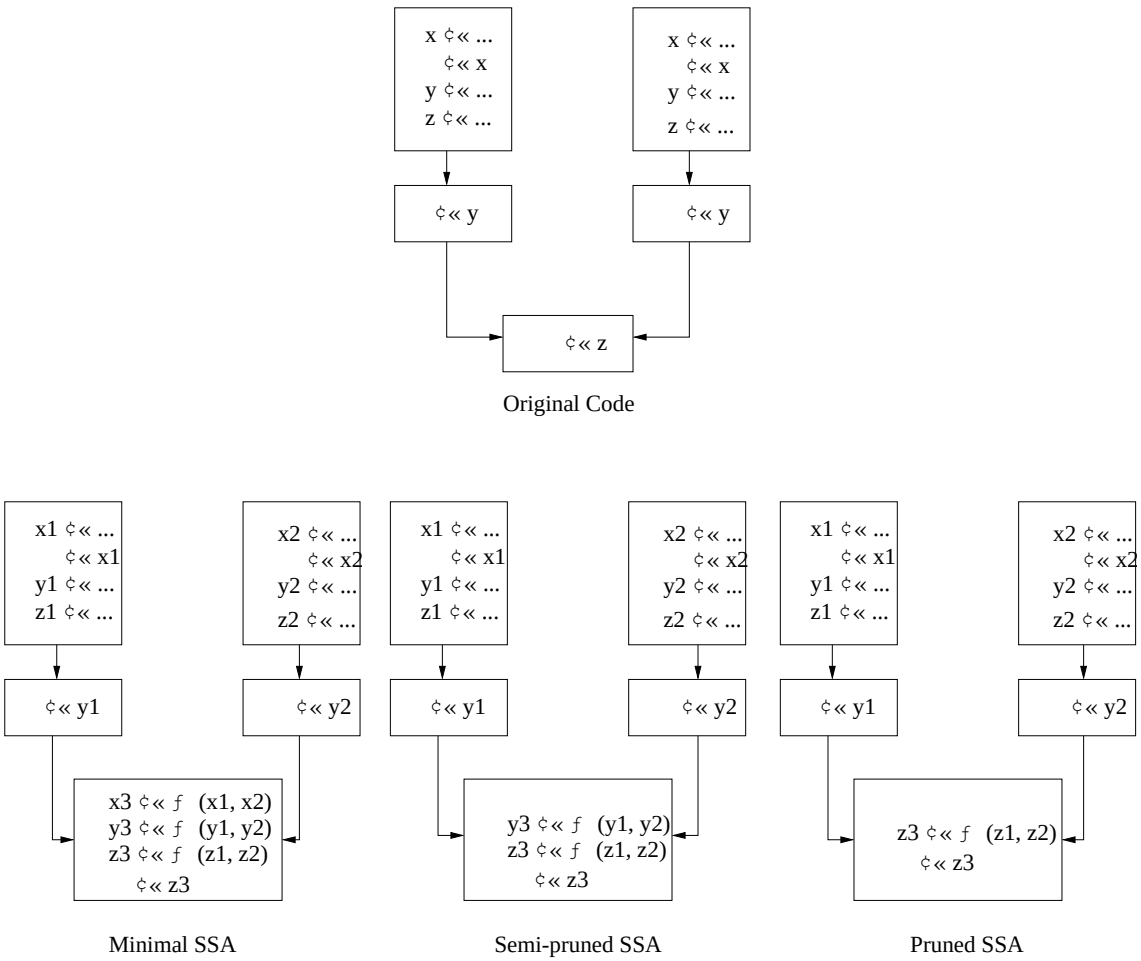


図 4.2: 3 種類の SSA 形式

4.1.1 minimal SSA

ノード X に変数 v の定義が存在するとき, $Y \in DF(X)$ には v に関する ϕ 関数が必要である. Y に v に関する ϕ 関数を定義すると, Y でもあらたに v の定義をしたことになるので, DF(Y) にも v に関する ϕ 関数を挿入する必要がある. これは v に関する ϕ 関数を挿入した際にあらたな支配辺境が表れなくなるまで繰り返す.

minimal SSA の定義自体はシンプルであるが, 生存区間がひとつの基本ブロック内であるような一時変数に対する ϕ 関数など, 無駄な ϕ 関数が挿入されてしまうことがある. また minimal とは定義が minimal なのであって, 挿入される ϕ 関数の数は 3 つの変換方法のなかでいちばん多い.

4.1.2 pruned SSA

minimal SSA では, 上述のような一時変数に対する ϕ 関数を挿入するなど, 無駄な ϕ 関数が挿入されてしまう. そこで ϕ 関数を挿入する際に, その ϕ 関数が本当に必要かどうかを確認し, 必要であるときにはじめて ϕ 関数を挿入するという方法がある. これを適用したものが pruned SSA である.

pruned SSA ではあらかじめ変数の生存区間を計算しておく必要がある. そして, 変数 v が定義されている基本ブロックの支配辺境である基本ブロックに, v が Live In している場合に限り, 変数 v に対する ϕ 関数を挿入する[1]. 変数 v がある基本ブロック X に Live In するということは, X の入口で v が生きているということである[2].

pruned SSA は ϕ 関数の数や, SSA 形式への変換過程で作成される変数の数が最も少なく効率的であるが, 事前にすべての変数の生存区間解析を行うコストがかかる.

4.1.3 semi-pruned SSA

semi-pruned SSA は minimal SSA と pruned SSA の中間的な形式である. ϕ 関数を挿入する方針としては, 単一の基本ブロックでのみ利用されている変数の定義に対しては ϕ 関数を挿入しない[3]. 逆に ϕ 関数を挿入することにより定義される変数がそれ以降使用されなくても, 元の変数が複数の基本ブロックにまたがって使用されている場合にはその ϕ 関数は挿入される[4].

semi-pruned SSA で ϕ 関数が挿入される変数は “non-local” と呼ばれる [7, 6]. semi-pruned SSA では, あらかじめ “non-local” な変数をみつけておかなければならないが, これは全ての変数の生存区間を解析するのと比較してコストが低い. “non-local” な変数をみつけるためには, 各基本ブロックを 1 回ずつ巡ればよい. そして各基本ブロック内で使用されている変数 v がその基本ブロック内での使用の前に定義されていなければ, 変数 v は “non-local” である.

[1] 図 4.2 の z がこれに相当する
[2] 逆に変数 v がある基本ブロック X から Live Out するということは, X の出口で v が生きているということである
[3] 図 4.2 の x がこれに相当する
[4] 図 4.2 の y がこれに相当する

4.2 変数の名前替え

4.2.1 アルゴリズム

SSA 形式において, 同じ名前の変数は必ず同じ値を持つ. すなわち各変数の定義ごとに異なる変数名をつける必要がある. 変数の名前替えは, 支配木を根から深さ優先で辿りながら再帰的に行う. 変数名前替えのアルゴリズムを図 4.3 に示す [19, 3].

SSA 形式では, CFG の入口ブロックで全ての変数に対する初期化が行われていると仮定している. 本方式ではその初期値を $\perp$ とする.

本システムでは, 各 $\perp$ についてプログラムの先頭で初期値を代入するコードを挿入する. 初期値は, 整数型 (LIR の Ixx で表現される型) に対しては 0 を, 浮動小数点型 (LIR の Fxx で表現される型) に対しては 0.0 を代入する.

```
for each 変数 v do
    /**
      * 変数をいれるスタックの初期化をする.
      * 『コンパイラの構成と最適化』の初版にでているアルゴリズムでは
      * 初期値の代入が抜けている.
      **/
    Stack(v) ← ⊥
end for
call renameVariables(Entry)

renameVariables(X) /* 手続き renameVariables() */
    for each 文 A in ブロック X do /* ブロックの先頭から順に */
      if 文 A の右辺が φ 関数でない then /* 条件文も含む */
        for each 変数 v in 文 A の右辺 do /* 条件式に現れる変数も含む */
          右辺の v を Stack(v) のトップで置き換える
        end for
      end if
      for each 変数 v in 文 A の左辺 do /* 右辺が φ 関数の場合も含む */
        if 文 A がコピー文であり, Copy Folding をするならば then
          Stack(v) に右辺の変数 vr を積む
          文 A を除去する
        else
          Stack(v) に新しい変数 v_new を積む
          v を v_new で置き換える
        end if
      end for
    end for
    for each ブロック Y ∈ succ(X) do
      Y にある φ 関数の引数 v を v_new に置き換える
    end for
    for each ブロック Z ∈ 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 4.3: 変数名前替えのアルゴリズム

4.2.2 Copy Folding

本システムでは, プログラムを SSA 形式へ変換する際に, 同時に Copy Folding を行うことができる. Copy Folding 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り, 6.1 節で

示すコピー伝播と機能は同じである．ここでは SSA 形式への変換中に行うコピー伝播という意味で Copy Folding という用語を使用する [6].

Copy Folding は, SSA 形式への変換時の変数の名前替えの際にコピー文に着目する．名前替えの対象となる文がコピー文 a←b であった場合には, コピー先変数のスタック Stack(a) に b を積み, 文 A を除去する．これにより, 以後 a を使用する式は b に名前替えされる．

アルゴリズムは名前替えのアルゴリズム (図 4.3) を参照されたい.

4.3 SSA 形式変数の初期値について

最新の実装では，SSA 形式変数の初期値 ⊥ として__dummy_function() を代入しているが，以前はたんに 0 を代入していた（4.2.1 節）．定数伝播の最適化などをより正確に扱うためには，⊥ のまま処理することが望ましい．

4.4 実装上の制限

LIR の仕様で規定されている PARALLEL と SUBREG は, SSA 形式として取り扱う上であやうい場合がある．PARALLEL を正確に扱うのであれば, 同時代入性を考慮して PARALLEL を分解し, その後 SSA 形式に変換すればよい．しかしそれを行うと, SIMD 最適化など, PARALLEL を用いる別の最適化効果を無効にしてしまう．また SUBREG はひとつのレジスタ変数の一部に対して代入をするという意味であるが, これを SSA 形式として正確に表現するのは困難である．そこで本システムではこれらの L 式が存在する L 関数は最適化しないこととする．

LIR の IF ノードの扱いについては, 3.2 節を参照．

第5章　SSA形式から通常形式への変換

本章では, SSA 形式から通常形式への変換の基本方針と変換する際の問題点について説明し, その後, 本システムに実装した Briggs らの方法 [6] と Sreedhar らの方法 [16] について述べる.

SSA 形式に見られる ϕ 関数は SSA 形式の定義を満たすための仮想的な関数であり, 一般的な計算機では実行不可能である. よって, 実行可能なプログラムを得るためには SSA 形式になっているプログラムから ϕ 関数を除去する必要がある. この変換にはいくつかの危うい状況が知られていて, それらについても配慮しなければならない.

なお, 本システムでは Sreedhar らの方法を基本として採用しており, まずそれを実装した. その後, 小濱による Briggs らの方法の実装 [22] を組み込んだ. Sreedhar らの Method III がお勧めである.

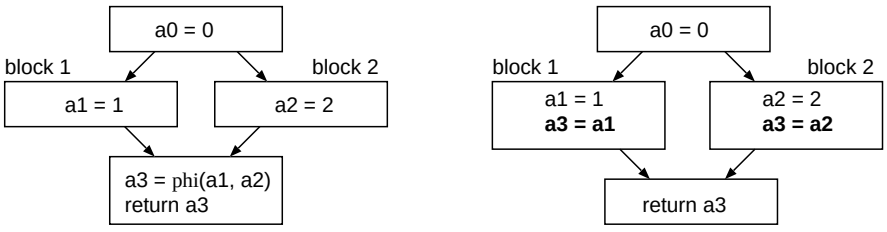
5.1　SSA 形式から通常形式への変換の基本方針

最初に, Cytron らが示した基本的な方法 [12] について述べる. 図 5.1 の左側の制御フローグラフを考えると, a3 の値は block1 を通って来た時は a1 の値が入り, block2 を通って来た時は a2 の値が入る. よって図 5.1 の右側のグラフのように block1 の最後尾に「 a3 ← a1 」, block2 の最後尾に「 a3 ← a2 」というコピー文を挿入することで ϕ 関数を除去することができる. 一般的には ϕ 関数のあるブロックの先行ブロックへ対応するコピー文を挿入する. これによってコピー文が ϕ 関数の代わりになり, ϕ 関数を削除することができる. この方法をここでは従来の方法と呼ぶ.

5.2　SSA 形式から通常形式へ変換する際の危うい状況

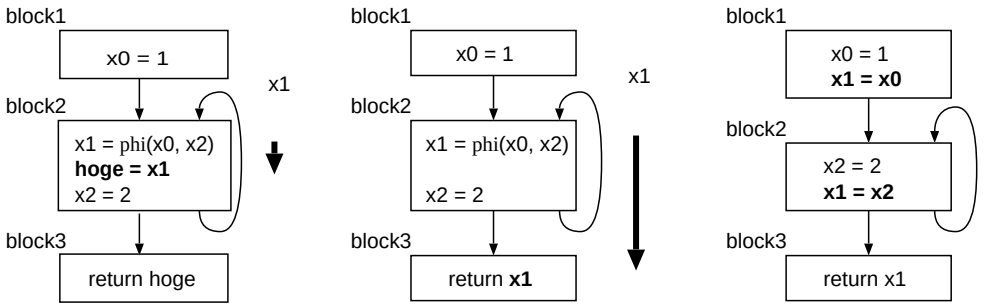
SSA 形式から通常形式への変換をする際に従来の方法では危うい状況がいくつか知られている. その具体例と原因について説明する. また, これより以下では ϕ 関数, 及びコピー文の左辺に現われる変数を**ターゲット**(target) と呼び, 右辺に現われる変数を**ソース**(source) と呼ぶことにする.

今後の議論では, ϕ 関数内にある変数の生存区間について注目するため, その扱いについて簡単に説明する.



SSA form　　After back translation from SSA form

図 5.1: 従来の方法



Graph1:
SSA form

Graph2:
After copy propagation

Graph3:
After back translation
using the naive method

図 5.2: 生存区間の拡張

- ϕ 関数のターゲットの生存区間は, その ϕ 関数のあるブロックの入口を必ず含むが, 先行ブロックの出口を含む必要はない
- ϕ 関数のソースの生存区間は, 対応する先行ブロックの出口を必ず含むが, その ϕ 関数のあるブロックの入口を含む必要はない

5.2.1 生存区間の拡張が問題となるケース (lost copy 問題)

プログラム中の変数が, それ以後に使われることがあるとき, その変数はその時点で**生きている (live)** という. 逆に使われることのないとき, **死んでいる (dead)** という. また, プログラムを実行順序の逆向きに見て「初めて現れる使用」から「定義」までの期間は変数が生きている期間で, **生存区間**と呼ぶ.

まず図 5.2 のグラフ 1 で表わされるプログラムについて考える. グラフの横の矢印は ϕ 関数のターゲットである x1 の生存区間を表している. ここでグラフ 1 に対してコピー伝播を行なうと, コピー文「hoge ← x1」が除去され, block3 の「return hoge」が「return x1」に置き換えられる (図 5.2 グラフ 2). グラフ 2 を見てわかるように, x1 の使用が後方へ延びるので x1 の生存区間も block2 を越えてしまう.

ここで図 5.2 のグラフ 2 を通常形式へ変換することを考える. 従来の方法 (第 5.1 節) では, ϕ 関数のあるブロックの先行ブロックの最後尾へ適当なコピー文を挿入する. この場合は, block1 へ「x1 ← x0」を, block2 へ「x1 ← x2」を挿入する (図 5.2 のグラフ 3). しかし, グラフ 2 では x1 の生存区間が block2 の出口を含んでいる. ここで「x1 ← x2」を block2 の最後尾に挿入してしまうと, x1 の生存区間内で不正に値を書き換えられることになってしまい, block3 において使用する x1 の値が破棄される可能性がある. 実際, グラフ 1, 2 では block1-block2-block3 のように制御が流れたとき, block3 で return される値が 1 であるのに対し, グラフ 3 においては block3 で return される x1 の値は 2 となってしまい, プログラムの意味を変えてしまっている. これを **lost copy 問題**と呼ぶ.

このように, SSA 形式から通常形式への変換を行なう際に, 挿入されるコピー文のターゲットが挿入される場所で生存している場合は, 注意が必要となる.

5.2.2 ϕ 関数の同時代入性が問題となるケース (simple ordering 問題)

同じブロック内に複数の ϕ 関数がある場合, それらの ϕ 関数は同時に処理されるものとして考えるべきである. これを **ϕ 関数の同時代入性**と呼ぶことにする.

今, 図 5.3 のグラフ 1 で表わされるプログラムについて考える. このグラフ 1 に対してコピー伝播が施されると block2 内の「y2 ← x1」が除去され, 「y1 ← ϕ(y0, y2)」が「y1 ← ϕ(y0, x1)」に置き換えられる (図 5.3 グラフ 2). また, この場合のように同一ブロックに現われた 2 つの ϕ 関数, 「x1 ← ϕ(x0 , x2)」と「y1 ← ϕ(y0 , x1)」は同時に処理されると考える. ここで block1-block2-block2-block3 のように制御が流れた場合, 2 回目の block2 において y1 に代入される x1 の値は, 2 回目の block2 で代入される値ではなく, 1 回目の block2 で代入された値である.

ここで図 5.3 グラフ 2 を通常形式へ変換することを考える. 従来の方法では

```
x1 ← x0
y1 ← y0
```

もしくは,

```
y1 ← y0
x1 ← x0
```

を block1 の最後尾へ挿入する. また同様に,

```
x1 ← x2
y1 ← x1
```

もしくは,

```
y1 ← x1
x1 ← x2
```

を block2 の最後尾へ挿入する.

ϕ 関数は同時代入性をもっているものとして処理されるべきものなので, 本来なら挿入されるコピー文の順序は関係ないはずである. しかし, block2 へ挿入されるコピー文の組を見ると前者と後者で y1 に入る値は異なる可能性がある (図 5.3 グラフ 3). グラフ 3 では, block3 における x1 と y1 の値が常に 5 となり, グラフ 1 と異なってしまう. これを **simple ordering 問題** と呼ぶ.

このように同一のブロックに複数の ϕ 関数が存在する場合, 通常形式に変換する際に ϕ 関数の同時代入性に注意しなければならない.

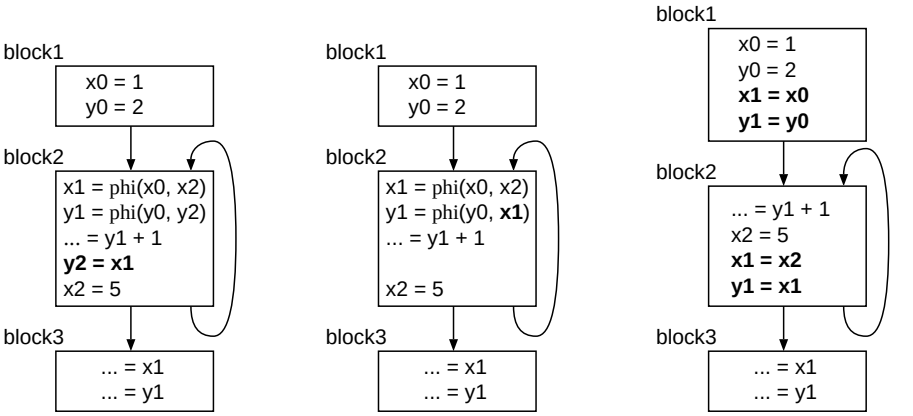


図 5.3: ϕ 関数の同時代入性

5.3 Briggs らの方法

Briggs らが提案するアルゴリズム [6] は 5.2 節で示した問題を解決したものである．本節では, Briggs らのアルゴリズムについて説明する．

5.3.1 基本方針

Briggs らのアルゴリズムの基本方針は, 従来のものと同様に ϕ 関数のあるブロックの先行ブロックへ適当なコピー文を挿入する．つまり, ϕ 関数をコピー文で代用する方法である．また, 危うい状況に遭遇したときは値の保持やコピー文のスケジューリングによって回避する．

この方針を取ると, 多くのコピー文が挿入される．しかし, Briggs らはレジスタ割り当ての際に生存区間の合併 (コアレシング) を行うことでそれらのコピー文の多くは除去されると主張している．

5.3.2 lost copy 問題に対する解決法

5.2.1 節では, ϕ 関数のターゲットの生存区間中で SSA 正規化によるコピー文が挿入されることにより, 期待されるターゲットの値が破棄される危険について述べた (lost copy 問題)．そのような危険を防ぐため, Briggs らの方法ではターゲットの値を別の変数に格納する策を取る．そして, ターゲットの値がその変数に格納されているという情報を保持し, その後のブロックに現れるターゲットの使用を格納した変数の使用で置き換える．このことにより, もしコピー文によってブロックの最後尾でターゲットの値が壊されても, その後のブロックでの影響をなくすことができる．

アルゴリズム的には, 「`temp` ← ターゲット」というコピー文を ϕ 関数のあるブロックの先頭に挿入することで, 壊される前のターゲットの値を `temp` に格納する．また, 変数名をインデックスに持つような `Stack` を用意し, ターゲットの値を `temp` に格納したという情報を保持する．具体的には `Stack[`ターゲット`]` に `temp` を積んでおき, その後のブロックに現われるターゲットについては `Stack[`ターゲット`]` から `temp` を取り出してこれと置き換える．

例えば, 図 5.4 [1] の上段のグラフを考える．従来の方式では, 先行ブロックとしての `block2` の最後尾にコピー文「`x1` ← `x2`」を挿入するわけであるが, 挿入する場所が `x1` の生存期間であるので, `block2` の先頭で「`temp` ← `x1`」を挿入し, 新しい変数 `temp` に `x1` の値を格納しておく．そして `Stack[x1]` に `temp` を積み, コピー文によって書き換えられる前の `x1` の値が `temp` に格納されているという情報を憶えておく．

そして `block3` で使用されている `x1` は `Stack[x1]` に積まれている `temp` で置き換える (図 5.4 の下段)．

これで「`x1` ← `x2`」を挿入しても, プログラムの意味を変えずに ϕ 関数を削除することができる (図 5.5)．

[1] 生存区間は注目すべきブロック (`block2`) のみ矢印で示し, ブロックの出入口を含むときは ● で明記する．また, プログラムのうち太字で表わした部分はこれから挿入することを意味し, 生存区間には反映されない．以降, この節ではこの記法を用いる．また, 図は考え方を示すものであり, 必ずしもアルゴリズムと一対一に対応するものではない．

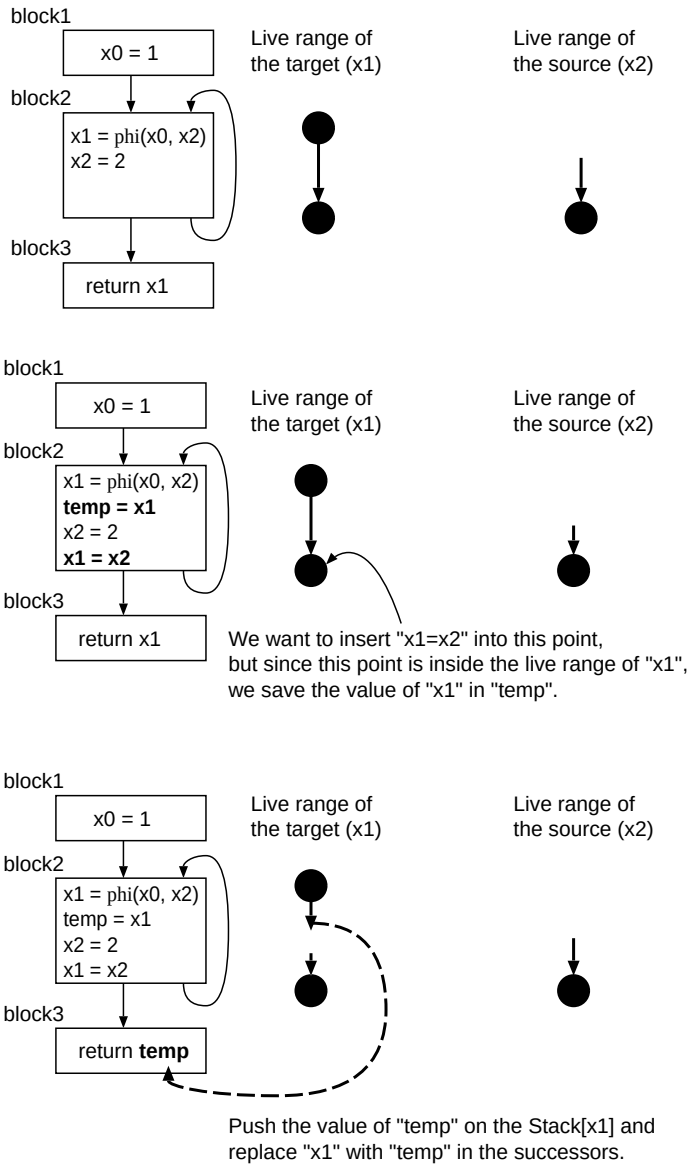


図 5.4: lost copy 問題に対する変換工程

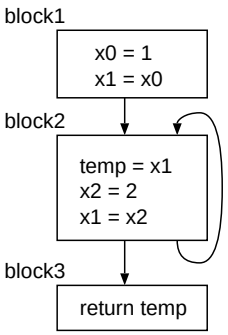


図 5.5: lost copy 問題に対する変換結果

	ターゲット	ソース
x1 ← x2	x1	x2
y1 ← x1	y1	x1

表 5.1: コピー文と ϕ 関数との関係

5.3.3 simple ordering 問題に対する解決法

5.2.2 節では 1 つのブロックに複数の ϕ 関数が存在するとき, ϕ 関数の同時代入性を考慮しないために期待される値が破棄される危険がある (simple ordering 問題) ことについて述べた. これを防ぐため, Briggs らの方法ではコピー文を挿入する際, プログラムの意味を変えないようにコピー文をスケジューリングする.

例えば, 図 5.6 の上段のグラフを考える. 従来の方式では, 先行ブロックとしての block2 の最後尾にコピー文「x1 ← x2」と「y1 ← x1」を挿入するわけだが, これらのコピー文は挿入される順序によって y1 の値が異なってしまう.

ここで ϕ 関数の同時代入性の理解を簡単にするため, 前述の 2 つのコピー文と, それぞれに対応した ϕ 関数との関係をターゲットとソースで分けて表す (表 5.1).

この表において, ターゲットの項目にある変数はこのコピー文が実行されることによって値が書き換えられる変数で, ソースの項目にある変数はこのコピー文が実行されるときに値が書き換えられていて欲しくない変数である. ここで x1 について考えると, x1 は「x1 ← x2」が実行されるときに値が書き換えられる変数であり, 「y1 ← x1」が実行されるときに値が書き換えられていて欲しくない変数でもある. つまり, 「y1 ← x1」を先に実行し, その後で「x1 ← x2」を実行するようにスケジューリングすれば, 「y1 ← x1」が実行されるときには x1 の値は書き換えられていないので, 期待される値を得ることができる.

ここで図 5.6 に話を戻す. Briggs らの方法では, まずブロック内の ϕ 関数の中でターゲットが他の ϕ 関数のソースになっているものがないか調べる. すると「x1 ← ϕ(x0, x2)」のターゲットである x1 は「y1 ← ϕ(y0, x1)」のソースであることがわかる. そこでまず, 「y1 ← x1」が先に実行されるように挿入する. 挿入する場所は y1 の生存区間なので, まず「temp ← y1」を block2 に挿入して y1 の値を temp に保持してから「y1 ← x1」を挿入する (図 5.6 の上段).

次に, 「x1 ← x2」を挿入することを考える. 挿入する場所は x1 の生存区間なので, 上記と同様に「temp' ← x1」を block2 の先頭に挿入して x1 の値を temp' に保持してから「x1 ← x2」を挿入する (図 5.6 の中段).

最後に block3 で使用されている x1 と y1 をそれぞれ Stack[x1], Stack[y1] に積まれている temp' と temp で置き換える (図 5.6 下段).

これでプログラムの意味を変えずに ϕ 関数を削除することができる (図 5.7).

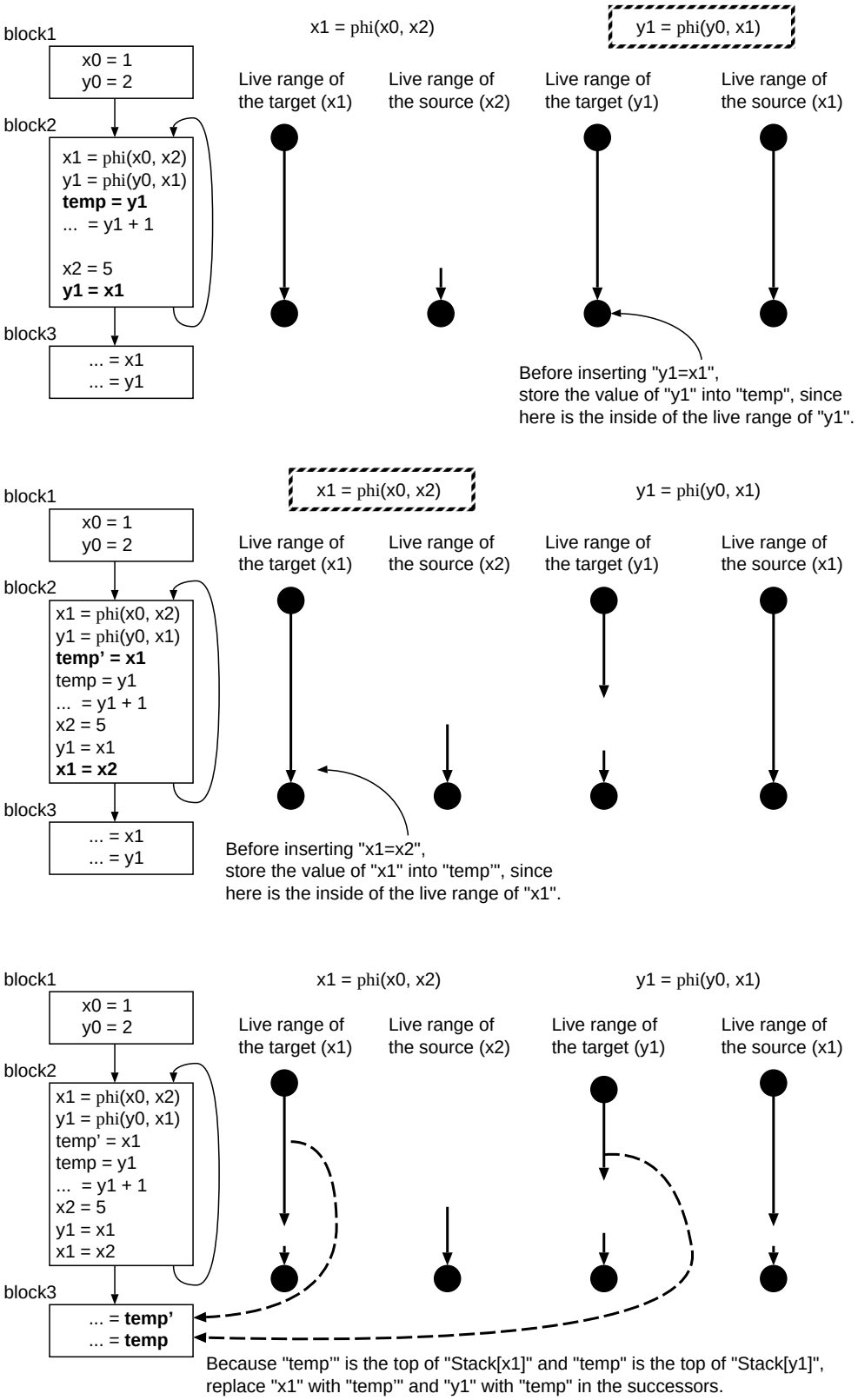


図 5.6: simple ordering 問題に対する変換工程

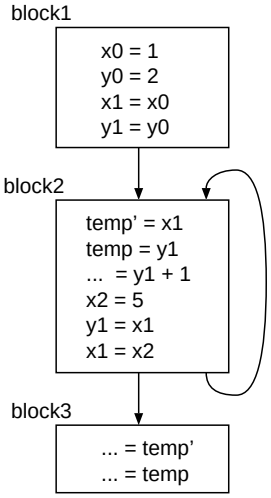


図 5.7: simple ordering 問題に対する変換結果

swap 問題

ここまでで, ある ϕ 関数のターゲットが別の ϕ 関数のソースになっている場合は挿入するコピー文の順番に気を付ければよいことがわかった. しかし, 実際には 2 つの ϕ 関数のターゲットがそれぞれお互いのソースになっていることも考えられる. つまり,

```
x1 ← φ(x0, y1)
y1 ← φ(y0, x1)
```

のような場合が考えられるのである. この問題を **swap 問題**(図 5.8) と呼ぶ.

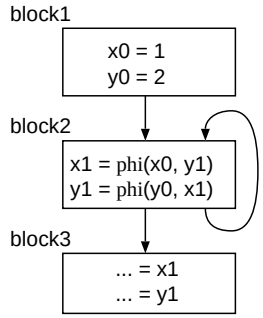


図 5.8: swap 問題

従来の方法では `block2` に「`x1 ← y1`」と「`y1 ← x1`」が挿入される. このとき, どちらのコピー文を先に実行しても期待する結果は得ることができない. そこで, どちらかの ϕ 関数に対応するコピー文を選び, そのターゲットを別の変数に一時的に保持する. 例えば「`x1 ← y1`」を選んだとすると,

```
temp ← x1
x1 ← y1
y1 ← temp
```

のように `temp` に「`x1 ← y1`」の実行前の `x1` の値を一時的に保持しておけば `y1` に適切な値を代入することができる.

ここで具体的な変換工程について説明する. 図 5.9 の上段のグラフで「x1 ← ϕ(x0, y1)」に対応するコピー文「x1 ← y1」を選んだとすると, まず「temp ← x1」を block2 の最後尾に挿入しておく. ここで x1 の値が temp に格納されているという情報を map[x1] ← temp という形で一時的に保持しておく (図 5.9 の上段). Stack とは違い, map は同じブロック内でコピー文を挿入するときに x1 の値が必要な場合に使う.

次に, コピー文「x1 ← y1」を挿入する. しかし, 挿入する場所は x1 の生存区間であるので「temp' ← x1」を挿入し, Stack[x1] に temp' を積む必要がある. 次に「y1 ← ϕ(y0, x1)」に対応したコピー文「y1 ← x1」を挿入するのだが, ここで map の情報が使われる. x1 の値は map[x1] ← temp より temp に一時的に格納されていることがわかるので「y1 ← temp」を挿入すればよいことになる. ここでも, 挿入する場所は y1 の生存区間なので上と同様な作業を行う (図 5.9 の中段).

そして最後に block3 で使用されている x1, y1 はそれぞれ Stack[x1], Stack[y1] に積まれている temp', temp'' で置き換える (図 5.9 の下段).

これでプログラムの意味を変えずに ϕ 関数を削除できる (図 5.10).

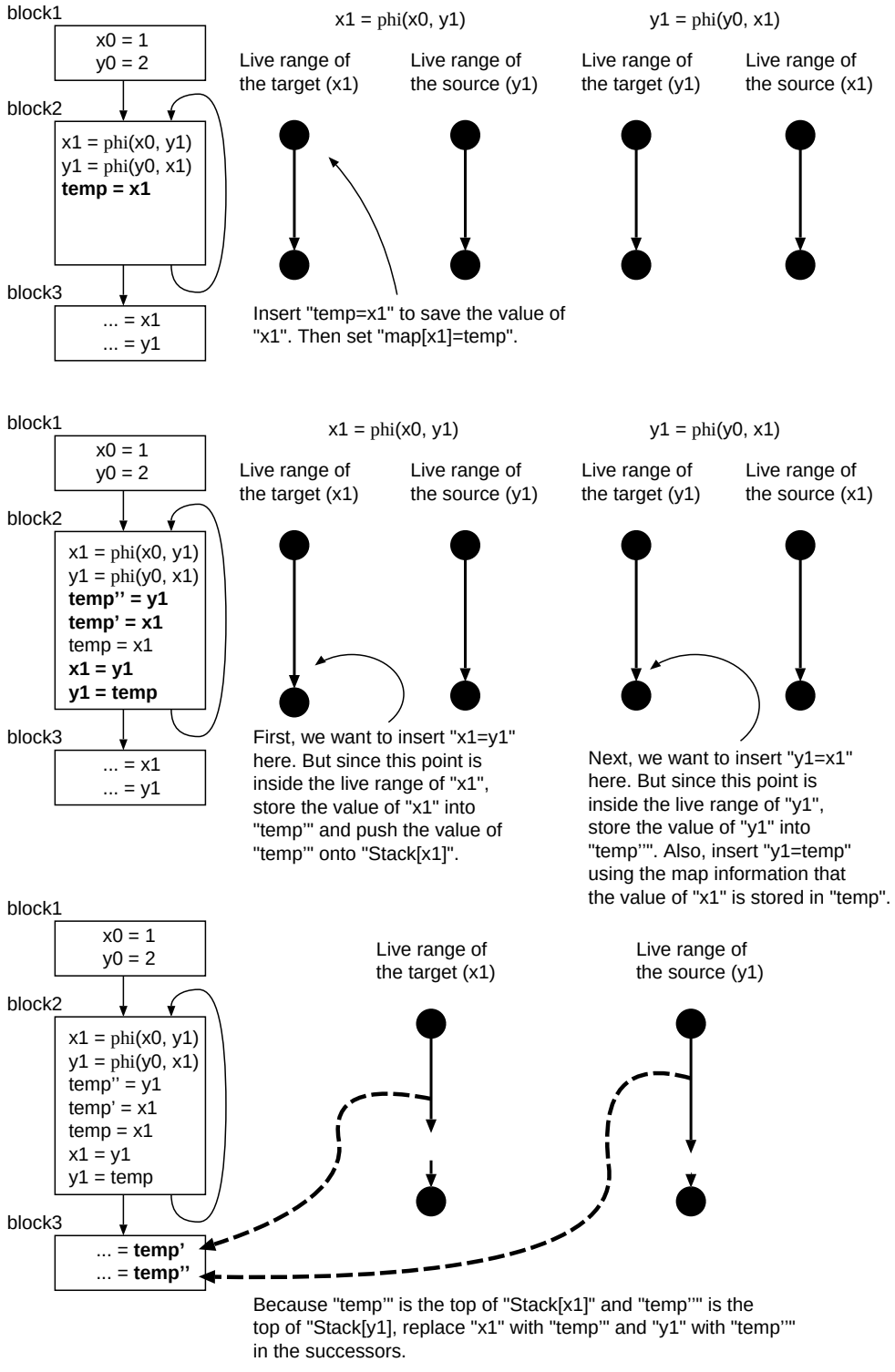


図 5.9: swap 問題の解決する変換工程

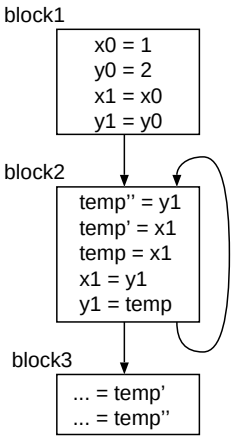


図 5.10: swap 問題を解決した変換結果

5.3.4 アルゴリズム

以下に Briggs らのアルゴリズムと, 簡単な説明を与える.

アルゴリズム (Briggs らのアルゴリズム)

/* SSA 形式から通常形式への変換を呼び出す部分 */
生存区間の解析
for each プログラムに現れる変数 v
Stack[v] ← **empty**
insert_copy(start)

/* 手続き insert_copy(block) */
pushed ← **null** //Stack に push された変数のリスト
for each i of block 内の命令
使用されている u があれば, u を Stack[u] のトップで置き換える
schedule_copy(block)
for each child of block in dominator tree
insert_copy(child)
for each pushed に入れられた変数 n
pop(Stack[n])

/* 手続き schedule_copy(block) */
/* Pass One: 各種データ構造の初期化 */
copy_set ← ∅ //copy_set の初期化
for each successors s of block
j ← s 中の ϕ 関数から見て, block が何番目の引数か
for each ϕ 関数に現れる変数 v
used_by_another ← false //used_by_another の初期化
for each ϕ 関数 in s
//src: ソース, target: ターゲット

```
    src←φ 関数の j 番目のオペランド
    copy_set ← copy_set ∪ {<src,target>}
    map[src] ← src
    map[target] ← target
    used_by_another[src] ← true

/* Pass Two: 同時代入性を気にせずコピー文を挿入できる組を worklist に入れる*/
for each copy <src,target> in copy_set
  if used_by_another[target] ≠ true //target が他の φ 関数のソースでなければ
    worklist ← worklist ∪ {<src,target>}
    copy_set ← copy_set - {<src,target>}

/* Pass Three: 生存区間に気を付けてコピー文を挿入する */
while worklist ≠ ∅ or copy_set≠ ∅
  while worklist が ∅ になるまで
    Pick a <src,target> from worklist
    worklist ← worklist - {<src,target>}
    if target の生存区間が block の出口を含む
      新しく変数 temp を作り,
      「 temp ← target 」を target を定義している φ 関数のある場所へ挿入する
      Stack[target] に temp を積む
      pushed ← pushed ∪ target
    「 target ← map[src] 」を block の最後尾に挿入する
    map[src] ← target
    if src が target として copy_set にある
      その組は target の値を保持しているので, copy_set から除去し,
      worklist へ入れる
  if copy_set ≠ ∅
    Pick a <src,target> from copy_set
    copy_set ← copy_set - {<src,target>}
    新しく変数 temp を作り,
    「 temp ← target 」を block の最後尾に挿入する
    map[target] ← temp
    worklist ← worklist ∪ {<src,target>}
```

5.3.5 実装上の注意

Briggs らの方法ではコピー文をブロックの最後尾に挿入することがある. しかし実装する中間表現が条件分岐命令, 多方向分岐命令などを持つ場合はその直前に挿入することになる. このとき挿入されるコピー文のターゲットの生存区間が, 挿入されるブロックの出口を含まなくても, ブロックの最後にある条件式で使用されていると, その条件式でターゲットが参照される前にその値を更新してしまうことになるので, 注意が必要であった. このことに関しては, Briggs らの実装の仕様書 [7] にも明記されていない.

そこで今回, 以下のように場合分けを行い, このような状況を回避した. 挿入するコピー文のター

ゲットを v とし, 挿入するブロック B の出口で生存している変数の集合を $LiveOut(B)$, B の最後にある条件式で使用される変数の集合を V とすると,

- $v \in V$ かつ $v \notin LiveOut(B)$ のとき
  「$temp \leftarrow v$」を対応する ϕ 関数が定義されている場所に挿入し, 条件式で使用されている v を $temp$ で置き換える
- $v \in V$ かつ $v \in LiveOut(B)$ のとき
  すでに「$temp \leftarrow v$」が挿入されていて Stack[v] には $temp$ が積まれているはずなので, 条件式で使用されている v を $temp$ で置き換える
- その他
  なにもせず

具体的にはアルゴリズム (5.3.4 節) の Pass Three を以下のように修正した.

アルゴリズム (Briggs らのアルゴリズムの修正 (Pass Three))

```
/* Pass Three: 生存区間に気を付けてコピー文を挿入する */
while worklist ≠ ∅ or copy_set≠ ∅
  while worklist が ∅ になるまで
    Pick a <src,target> from worklist
    worklist ← worklist - {<src,target>}
    if target の生存区間が block の出口を含む
      新しく変数 temp を作り,
      「 temp ← target 」を target を定義している φ 関数のある場所へ挿入する
      Stack[target] に temp を積む
      pushed ← pushed ∪ target

  === ここから
      if target が block の条件式で使用されている
        条件式の target を temp で置き換える
    else /* target の生存区間が block の出口を含まない */
      if target が block の条件式で使用されている
        新しく変数 temp を作り,「 temp ← target 」を
        target を定義している φ 関数のある場所へ挿入する
        条件式の target を temp で置き換える
  === ここまで

    「 target ← map[src] 」を block の最後尾に挿入する
    map[src] ← target
    if src が target として copy_set にある
      その組は target の値を保持しているので, copy_set から除去し,
      worklist へ入れる
  if copy_set ≠ ∅
    Pick a <src,target> from copy_set
    copy_set ← copy_set - {<src,target>}
    新しく変数 temp を作り,
    「 temp ← target 」を block の最後尾に挿入する
```

```
map[target] ← temp
worklist ← worklist ∪ {<src,target>}
```

5.4 Sreedhar らの方法

Sreedhar らが提案するアルゴリズム [16] は 5.2 節で示した問題を解決したものである．本節では, Sreedhar らのアルゴリズムについて説明する．

5.4.1 基本方針

Sreedhar らのアルゴリズムの基本方針は, 従来のものや Briggs らによるものとは考え方がかなり異なっている．イメージとしては Briggs らの方法が ϕ 関数をコピー文で代用するのに対し, Sreedhar らの方法では ϕ 関数をコアレシングすることで ϕ 関数を取り除く．

まず, Sreedhar らの方法では ϕ 関数内に現れる変数の干渉について考える．図 5.11 において ϕ 関数に現れる変数は a1，a2，a3 である (ターゲットを含む)．このとき **phiCongruenceClass** という集合を考える．phiCongruenceClass とは ϕ 関数を介して, 干渉のない変数の集合である．初期状態を,

phiCongruenceClass[a1] ← a1
phiCongruenceClass[a2] ← a2
phiCongruenceClass[a3] ← a3

として持つ．生存区間の干渉については後述するが, 図 5.11 において a1，a2，a3 は互いに干渉がないので, 同一の phiCongruenceClass としてまとめることができる．

phiCongruenceClass[a1] ← {a1，a2，a3}
phiCongruenceClass[a2] ← {a1，a2，a3}
phiCongruenceClass[a3] ← {a1，a2，a3}

ここで phiCongruenceClass 内の変数は互いに干渉がないことから, 同じ変数名に置き換えることができる．つまり, ϕ 関数内の変数をコアレシングしているものと見ることができる．例えばこの 3 つの PhiCongruenceClass は同じなので, その変数 {a1，a2，a3} を全て A に置き換えると, 図の右側のように変換することができる．

また, プログラム中に以下のような ϕ 関数が含まれていたとする．

a1 ← ϕ(a2，a3) --- (1)
b1 ← ϕ(a2，b3) --- (2)

このとき, a2 は ϕ 関数 (1)，(2) の両方に属している．このように, 複数の ϕ 関数に含まれるような変数があった場合にも上と同様にコアレシングすることを考える．つまり, 変数 a1，a2，a3，b1，b3 に互いに干渉がなければ, これら全てをコアレシングすることになる．

このように, Sreedhar らの方法では基本的に新しいコピー文は挿入しない．しかし, ϕ 関数内の変数に干渉があった場合には, これらをコアレシングすることはできない．そこで Sreedhar らの方法では ϕ 関数内に現れる変数の干渉を取り除くため (phiCongruenceClass を構成するため) コピー文が挿入される．

どのように干渉を取り除くかを以下で詳しく説明する．

5.4.2 Sreedhar らの方法におけるコピー文の意味

Sreedhar らの方法におけるコピー文の意味は, 以下の ϕ 関数の性質に注目すると理解しやすい．

- ϕ 関数のターゲットの生存区間は, その ϕ 関数のあるブロックの入口を必ず含むが, 先行ブロックの出口を含む必要はない

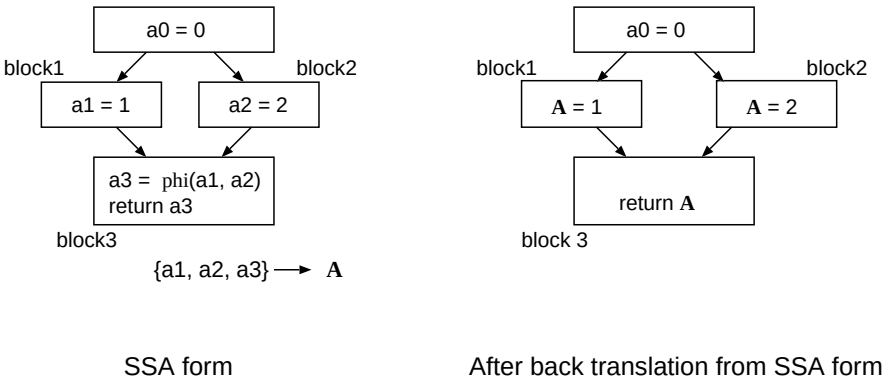


図 5.11: Sreedhar 法による基本的な方針

- ϕ 関数のソースの生存区間は, 対応する先行ブロックの出口を必ず含むが, その ϕ 関数のあるブロックの入口を含む必要はない

Sreedhar らの方法で挿入されるコピー文の役割は, ϕ 関数内のターゲットとソース, もしくは, ソース同士に干渉があるとき, それら ϕ 関数のターゲットやソースの生存区間が必要最小限になるように ϕ 関数を書き換えることである. 具体的には以下のように生存区間を短縮する.

ターゲットに対するコピー文

図 5.12 [2] の上段で ϕ 関数のターゲットの生存区間を短くしたいとする. そのときは ϕ 関数「`x2` $\leftarrow$ ϕ `(x1, x3)`」を「`x2'` $\leftarrow$ ϕ`(x1, x3)`」と書き換え, 「`x2` $\leftarrow$ `x2'`」を ϕ 関数の直下に挿入することで, プログラムの意味を変えずに新しい ϕ 関数のターゲットである `x2'` の生存区間を必要最小にすることができる (図 5.12 の中段).

ソースに対するコピー文

図 5.12 の上段で ϕ 関数の二番目のソースの生存区間を短くしたいとする. そのときは ϕ 関数「`x2` $\leftarrow$ ϕ`(x1, x3)`」を「`x2` $\leftarrow$ ϕ`(x1, x3')`」と書き換え, 「`x3'` $\leftarrow$ `x3`」を先行ブロックの最後尾に挿入することで, プログラムの意味を変えずに新しい ϕ 関数の二番目のソースである `x3'` の生存区間を必要最小にすることができる (図 5.12 の下段).

以上のようなコピー文を使い分けることで, ϕ 関数内の変数の干渉を取り除くことができる. 次節で具体例をいくつか紹介する.

[2] 生存区間を矢印で表記し, ブロックの出入口を含むときは ● で示し, 必要があれば, ブロックの出入口を含まないことを ○ で明記する.

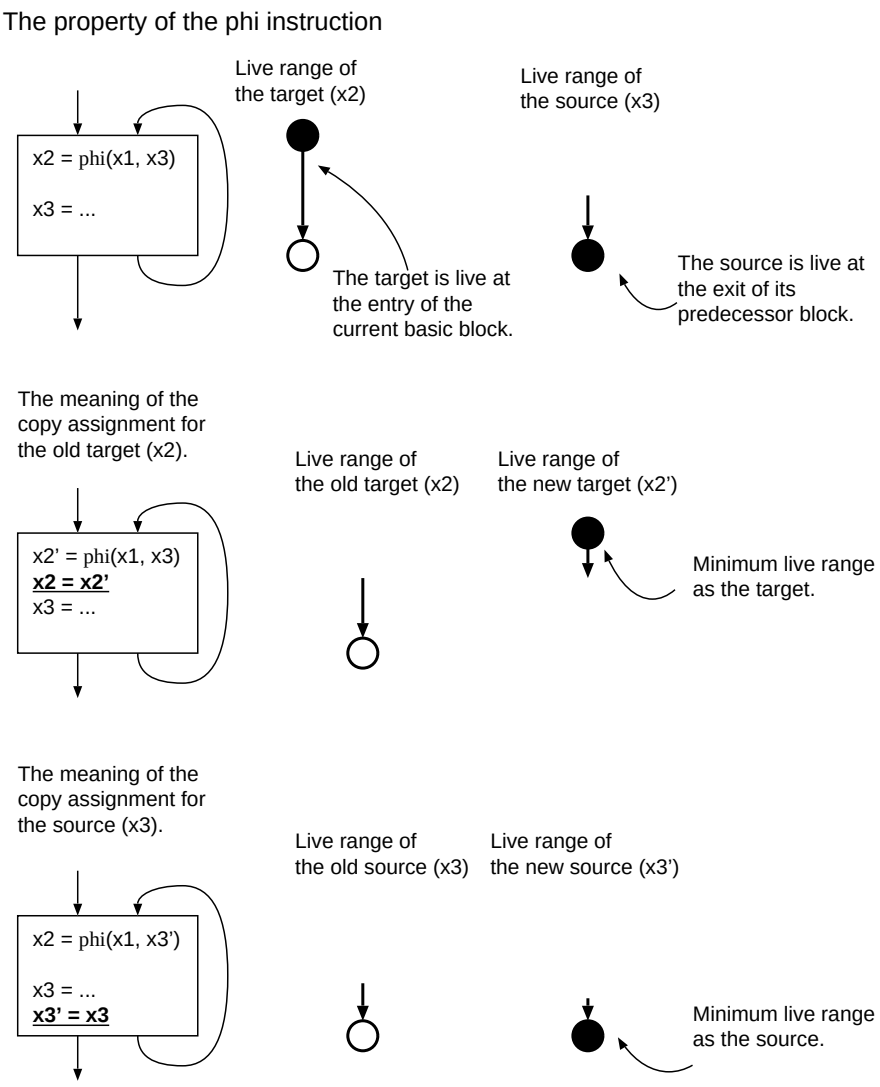


図 5.12: Sreedhar 法のコピー文の意味

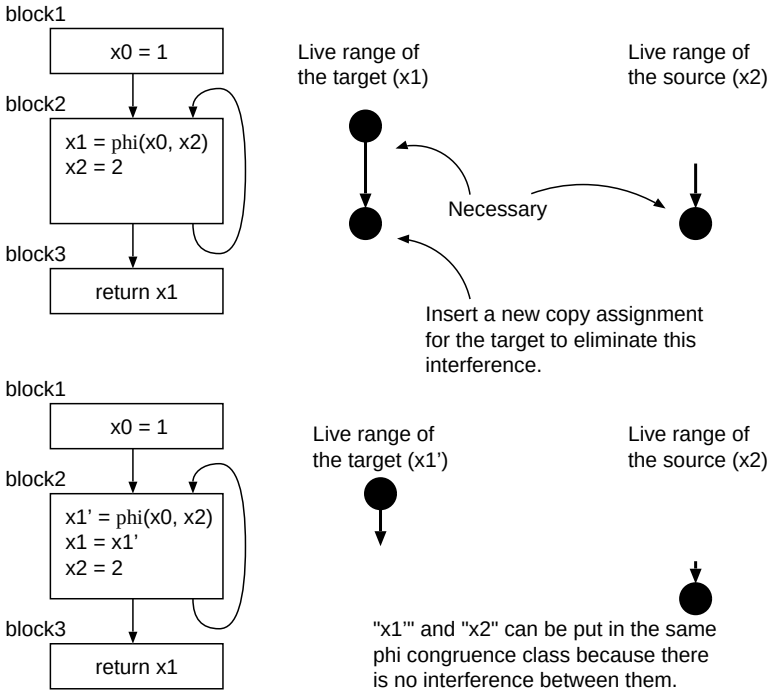


図 5.13: lost copy 問題に対する変換工程

5.4.3 lost copy 問題に対する解決法

図 5.13 の上段のグラフ[3] を考えると, ϕ 関数内に現れる変数は {x0, x1, x2} である. x1 と x2 の生存区間を見るとわかるように x1 と x2 は干渉している.

この干渉をなくすには, ϕ 関数「x1← ϕ(x0, x2)」のターゲットである x1 の生存区間が block2 の出口を含まないようにしてやればよいので, ターゲットに対するコピー文の挿入をする (図 5.13 の下段).

すると, 新しい ϕ 関数内に現れる変数 {x0, x1', x2} は互いに干渉するものがなくなるので, 同一の phiCongruenceClass に入れることができる. この phiCongruenceClass 内の変数を X で書き換えることで問題を回避できる (図 5.14).

[3]生存区間は注目すべきブロック (block2) のみ矢印で示し, ブロックの出入口を含むときは ● で明記する. 以降, この節ではこの記法を用いる. また, 図は考え方を示すものであり, 必ずしもアルゴリズムと一対一に対応するものではない.

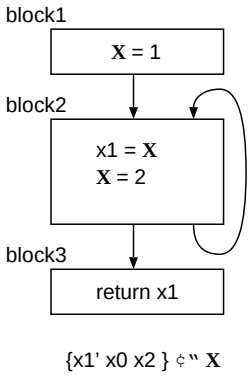


図 5.14: lost copy 問題に対する変換結果

5.4.4 simple ordering 問題に対する解決法

図 5.15 の上段を考えると, ϕ 関数内に現れる変数は {x1，x0，x2} の組と {y1，y0，x1} の組である. まず, {x1，x0，x2} の組を先に考えるとする. {x1，x0，x2} の組の中ではターゲットとしての x1 と, ソースとしての x2 が block2 の出口で干渉している. x2 はソースなので, 生存区間が出口を含むのは仕方がない. よってターゲットの生存区間を短くするために, ターゲットに対するコピー文「x1 ← x1'」を block2 の先頭に挿入し, 「x1' ← ϕ(x0，x2)」と書き換える (図 5.15 の中段). すると新しい ϕ 関数「x1' ←ϕ(x0，x2)」の変数の組 {x1'，x0，x2} に干渉はなくなる.

次に {y1，y0，x1} の組について考える. {y1，y0，x1} の組の中では, ターゲットとしての y1 と, ソースとしての x1 が干渉している (図 5.15 の中段). x1 はソースなので, 生存区間が出口を含むのは仕方がない. よってターゲットの生存区間を短くするために, ターゲットに対するコピー文「y1 ← y1'」を block2 の先頭に挿入し, 「y1' ← ϕ(y0，x1)」と書き換える. すると新しい ϕ 関数「y1' ← ϕ(y0，x1)」の変数の組 {y1'，y0，x1} に干渉はなくなる (図 5.15 の下段).

すると, {x1'，x0，x2} と {y1'，y0，x1} はそれぞれ phiCongruenceClass としてまとめることができる. そこで , {x1'，x0，x2} を X に, {y1'，y0，x1} を Y に書き換えるとプログラムの意味を変えずに ϕ 関数を取り除くことができる (図 5.16).

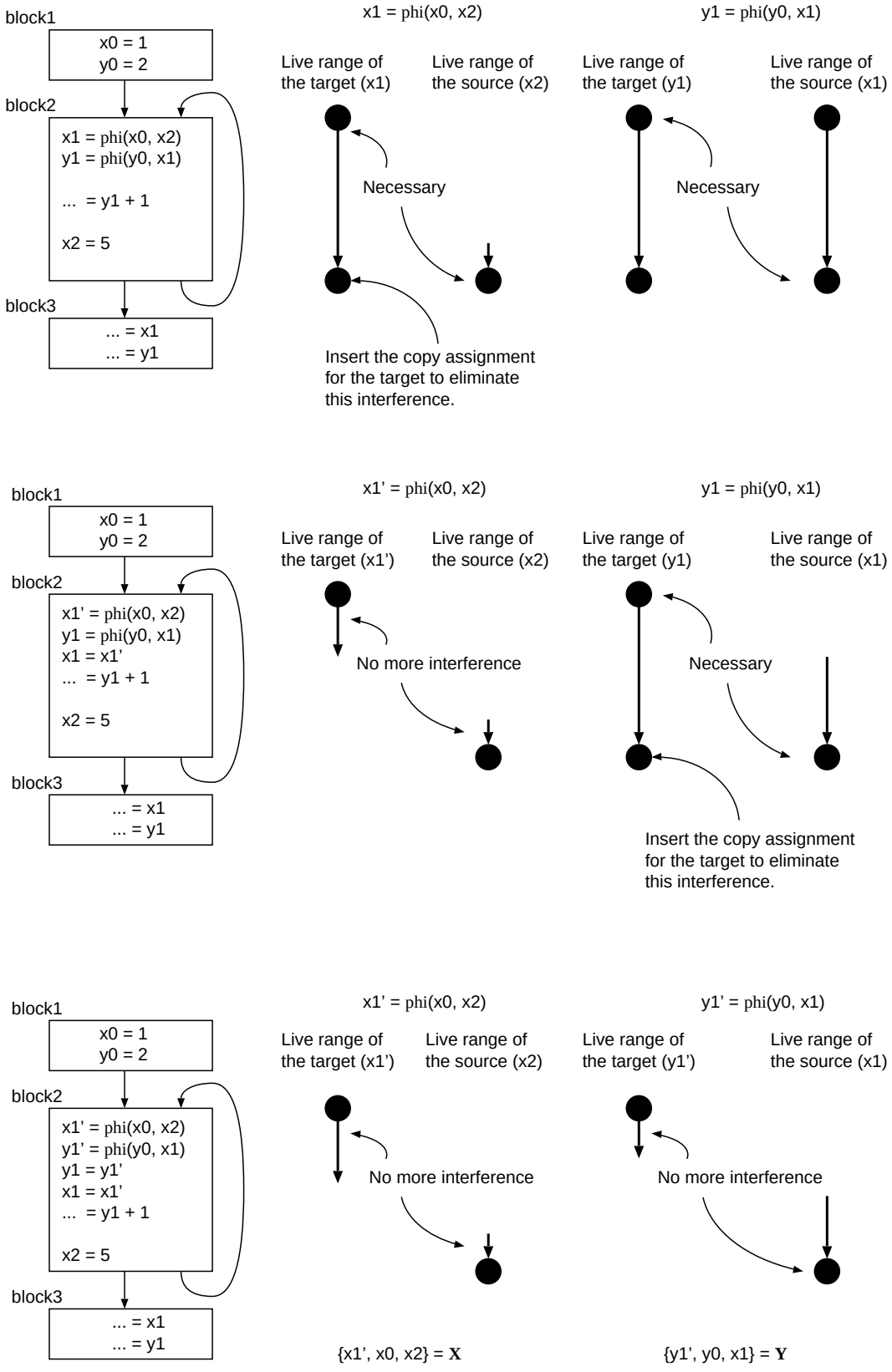


図 5.15: simple ordering 問題に対する変換工程

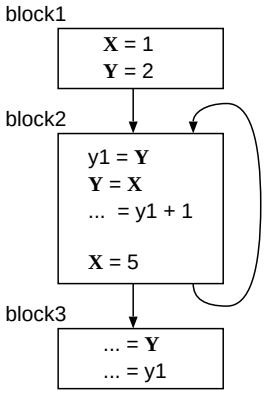


図 5.16: simple ordering 問題に対する変換結果

5.4.5 swap 問題

図 5.17 の上段を考えると, ϕ 関数内に現れる変数の組は {x1，x0，y1} の組と {y1，y0，x1} の組である.

まず, {x1，x0，y1} の組を先に考えるとする. {x1，x0，y1} の組の中ではターゲットとしての x1 とソースとしての y1 が block2 の入口と出口の両方で干渉している. この干渉を取り除くには x1 に対するコピー文と y1 に対するコピー文を両方挿入してやればよい. x1 はターゲットなので,「x1 ← x1'」を block2 の先頭に挿入し, ϕ 関数を「x1' ←ϕ(x0，y1)」と書き換える. また, y1 はソースなので,「y1' ← y1」を挿入し, ϕ 関数を「x1' ←ϕ(x0，y1')」と書き換える. これにより, 新しい ϕ 関数に現れる変数の組 {x1'，x0，y1'} に干渉はなくなる (図 5.17 の中段).

次に {y1，y0，x1} の組について考える. このとき, ターゲットとしての y1 と, ソースとしての x1 は block2 の中で干渉している (図 5.17 の中段). この場合は, y1 に対するコピー文を挿入すればよい. そこで y1 に対して, ターゲットに対するコピー文を挿入することにする.「y1 ← y1''」を block2 の先頭に挿入し, ϕ 関数を「y1'' ←ϕ(y0，x1)」と書き換える. すると, 新しい ϕ 関数に現れる変数の組 {y1''，y0，x1} の中に干渉はなくなる (図 5.17 の下段).

よって {x1'，x0，y1'} と {y1''，y0，x1} はそれぞれ phiCongruenceClass としてまとめられる. そこで {x1'，x0，y1'} を X に, {y1''，y0，x1} を Y に書き換えると swap 問題を回避して ϕ 関数を取り除くことができる (図 5.18).

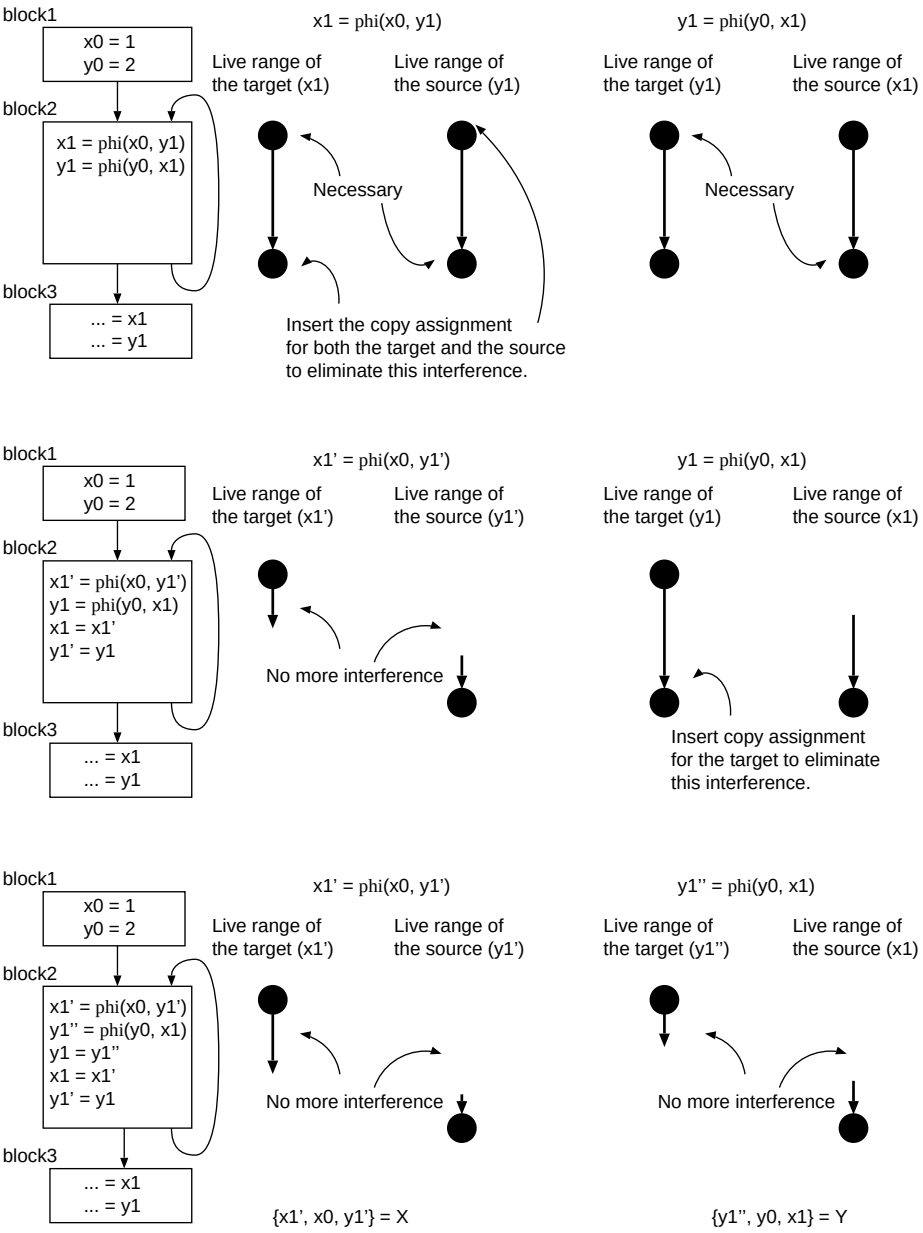


図 5.17: swap 問題に対する変換工程

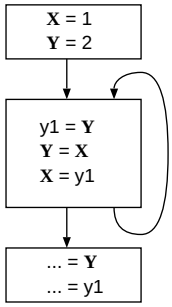


図 5.18: swap 問題に対する変換結果

5.4.6 TSSA 形式から CSSA 形式への変換

CSSA 形式 (Conventional SSA form) とは, Cytron らのアルゴリズムで作成した直後の SSA 形式のことをいう. 以下リソースとは変数の意味で使用する. CSSA の特徴として, 同じ Phi Congruence Class に属する全てのリソースを同じ代表変数に置き換え, 全ての ϕ 関数を除去してしまってもオリジナルのプログラムの意味を変えないことが保証されている. しかし, SSA 形式での最適化変換を行うと, このような CSSA 形式の性質は一般的には保たれない. つまり ϕ 関数のリソース間に干渉が発生する. ここでいう干渉とは, 2 つの変数 a, b について, a の生存区間と b の生存区間に重なりがあるときのことを指している [1, 3, 8]. このような SSA 形式を TSSA 形式 (Transformed SSA form) という.

図 5.19 は CSSA 形式の例である. 同じ Phi Congruence Class に属する変数 x1, x2, x3 は, その代表変数である x に置き換えることができる. 図 5.19 の x1, x2, x3 を代表変数 x に置き換えて ϕ 関数を除去したものが図 5.20 の通常形式である.

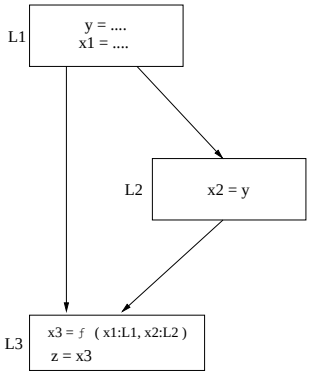


図 5.19: SSA 形式の例

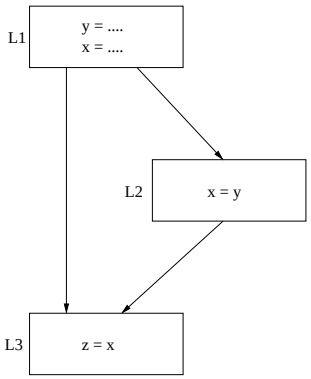


図 5.20: CSSA から通常形式への変換例

一方, 図 5.19 にある x2←y という代入文をコアレシングすると図 5.21 のようになる.

図 5.21 では, 変数 x1, x3, y が同じ Phi Congruence Class に属している. しかしこれらを代表変数 x に置き換えることはオリジナルのプログラムの意味を変えてしまう. これは x2←y というコピー文を除去してコアレシングしたことにより, x1 と y の生存区間が干渉してしまったからである. このような TSSA 形式では, コピー文をいれることにより CSSA 形式に変換することができる.

TSSA 形式から通常形式に変換する手順は以下の通りである.

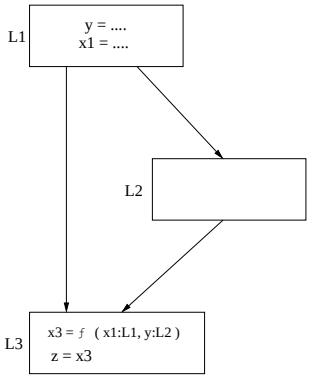


図 5.21: TSSA の例

1. TSSA 形式を CSSA 形式に変換
2. 無駄なコピー文の除去
3. ϕ 関数を除去して通常形式に変換

以下では, 1 に関して, 本システムで実装した方式について述べる．2 に関しては 5.5 節で述べる．3 については次のことに注意する．ϕ 関数のリソースを代表変数に置き換えるとき, その Phi Congruence Class に複数の ϕ 関数のリソースが入っているときには, それらを一気にひとつの代表変数に置き換えるようにする．このこと以外は自明であるので 3 の説明は省略する．

まず 1 のための前処理について述べる．

ϕ 関数に関する前処理

Sreedhar らが [16] で述べている方法は ϕ 関数のソースリソースが変数の場合を考えている．しかし本システムを用いて最適化を行うと, 最適化の途中で ϕ 関数のソースリソースが定数になる場合がある．上述のように Sreedhar らの方法では変数の生存区間を元に Phi Congruence Class を設定して通常形式への変換を行うが, ϕ 関数のリソースが定数の場合は, 定数の生存区間や Phi Congruence Class をどう考えるか等の問題が発生する．そこで本システムでは, Sreedhar らの方法を利用して通常形式に変換する直前に ϕ 関数のソースリソースをチェックする．もし ϕ 関数のソースリソースに定数がある場合には, 新しい変数 temp を作成し, $temp \leftarrow$ 定数 というコピー文を predecessor に挿入する．その後, ϕ 関数のソースリソースの定数を変数 temp に置き換える．

テンポラリ変数に関する前処理

本システムを用いて最適化を行うと, 例えば 6.9 節で述べる方法で式を 3 番地コードに分解するとテンポラリ変数が大量に発生してしまう．テンポラリ変数を用いることでプログラムの意味は変わらないが, コード生成部で効率の良いコードを生成することを阻害する場合がある．そこで本システムでは, SSA 形式から通常形式へ変換する直前で以下の条件に合うテンポラリ変数を除去する．

- 変数の生存区間が基本ブロックの内だけ (基本ブロックローカル) であり, かつ一度しか参照されない

このようなテンポラリ変数の除去は, 対象となるテンポラリ変数が使用されている部分を, テンポラリ変数を定義している式の右辺と置換することで実現する.

(例) $temp$ は基本ブロックローカルとする.

$$
\begin{aligned}
temp &\leftarrow a+b \\
&\quad ... \qquad (\text{ここで } temp \text{ の使用は無い}) \\
c &\leftarrow temp + ... \\
&\quad ... \qquad (\text{ここで } temp \text{ の使用は無い})
\end{aligned}
$$

上記のような場合, $c \leftarrow temp + ...$ は $c \leftarrow (a+b) + ...$ に置換される.

Method I

Method I では, コピー文を ϕ 関数のすべてのリソースについて挿入する. ϕ 関数のリソースは 2 種類あり, ϕ 関数によって定義されるリソースをデスティネーションリソース, ϕ 関数の引数になっているリソースをソースリソースとする. デスティネーションリソースに関するコピー文は ϕ 関数と同じ基本ブロックに, ソースリソースに関するコピー文は, そのリソースに対応した predecessor に挿入する. コピー文の挿入位置は, 同じ基本ブロックに挿入する場合には ϕ 関数の直後 (複数の ϕ 関数がある場合には, すべての ϕ 関数のあと), predecessor に挿入する場合には, その predecessor の最後の命令の後 (最後の命令が条件分岐文の場合はその直前) とする[4]. 図 5.22 に Method I の例を示す.

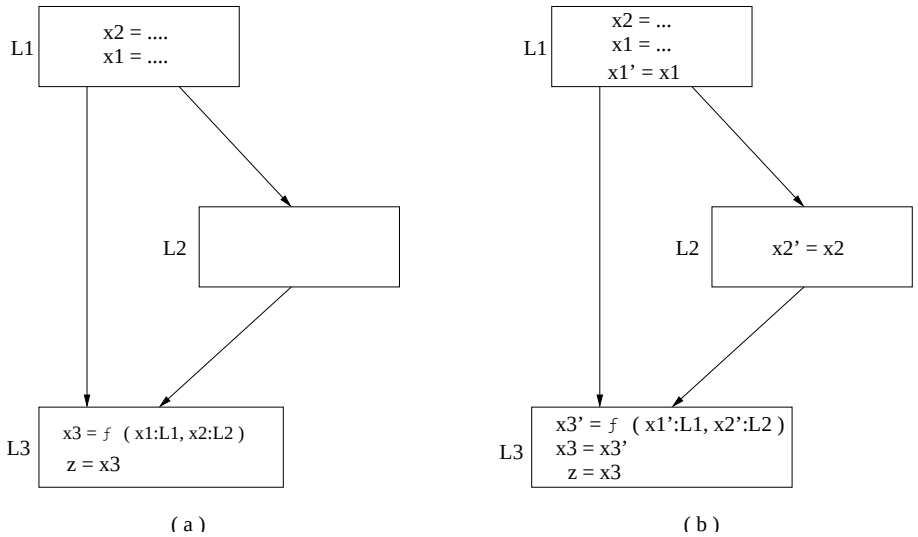


図 5.22: TSSA 形式 (a) から CSSA 形式 (b) への Method I を用いた変換例

図 5.22(a) が TSSA 形式の例であり, (b) が Method I を利用して (a) を CSSA 形式に変換したものである.

Method II

Methid II では, ϕ 関数のリソースのなかで互いに干渉しているものだけを対象としてコピー文を挿入する. 図 5.22 の (a) では, リソース x1 と x2 が互いに干渉している. そこでその干渉を打ち消

[4]実は Sreedhar らは, コピー文の挿入場所について, 基本ブロック単位での指定はしているものの, 基本ブロック内部でどの位置にいれるべきかを言及していない. そのため危険辺がある場合に問題が生じる場合がある. 詳しくは 5.4.6 節を参照.

すために x1 と x2 に関するコピー文を挿入する必要がある. x3 は x1 とも x2 とも干渉しないためコピー文は挿入されない.

コピー文を挿入した後には, 干渉グラフの更新と Phi Congruence Class の更新が必要となる.

Method III

Method III ではプログラム中の各変数の生存情報を利用する. 図 5.22 (a) を用いて Method III を説明する.

図 5.22 (a) では, ϕ 関数のリソース x1 と x2 が干渉している. そして変数の生存情報をみると, LiveOut[L1]={x1,x2} であり, LiveOut[L2]={x2} である. LiveOut[L1] に x2 が入っているので, ブロック L1 の最後にコピー文 x1'←x1 を挿入しただけでは ϕ 関数のリソース間の干渉は解消されない (この場合には L2 にもコピー文を挿入する必要がある). 一方, x1 は LiveOut[L2] に入っていないので, コピー文 x2'←x2 を L2 に挿入するだけで ϕ 関数のリソース間の干渉を解消することができる. Method III では ϕ 関数のリソース間の干渉を打ち消すためのコピー文挿入をなるべく最小にするようにコピー文挿入場所を選択する. よって, 上記の例では x2 に関するコピー文のみが挿入される.

ϕ 関数のソースリソース間の干渉を解消する際には LiveOut を利用したが, デスティネーションリソースとソースリソース間の干渉を解消する際には LiveIn と LiveOut の情報が必要である. また, Method III で挿入するコピー文は, 干渉グラフに基づく標準的なコアレシングアルゴリズムでは除去することができないものであることが知られている. ただし, 挿入されるコピー文の数が最小であるという保証はない.

以下で Method III のアルゴリズムを述べる. また, 変数 y が x の Phi Congruence Class に属しているということを $y \in pcc(x)$ と書く.

1. for each x (x はカレント CFG 上の ϕ 関数のリソース) do
   pcc(x)←x
   end for
   /*
   このアルゴリズムは ϕ 関数をひとつずつ処理する.
   各 ϕ 関数について以下のステップ 2 〜 4 を実行する.
   */

2. for each カレント CFG 上の ϕ 関数 do
   /*
   ϕ 関数は次のようなフォーマットであると仮定する.
   x0←f(x1:L1, x2:L2, ..., xn:Ln)
   つまり, 以下では左辺のターゲットも含めて単に ϕ 関数という.
   L0 は ϕ 関数を含む基本ブロックである.
   */

3. 対象となる ϕ 関数の candidateResourceSet{} をセットする
   for each xi (xi は ϕ 関数のリソース, $0 \leq i \leq n$) do
   unresolvedNeighborMap(xi)←{}
   end for
   /*
   candidateResourceSet{} はコピー文を挿入する変数を保持する.

unresolvedNeighborMap は, どのリソースに関してコピー文をいれる
かが一意に決まらない場合に, それを保留するために変数を保持する.
*/

4. for each xi:Li と xj:Lj
(xi, xj は今扱っている ϕ 関数のリソース, $0 \leq i, j \leq n \wedge xi \neq xj$) do
干渉情報を調べる
if (yi ∈ pcc(xi) ∧ yj ∈ pcc(xj) ∧ yi と yj が互いに干渉している) then
xi と xj の干渉をなくすために以下の場合分けにそって
コピー文をいれる位置を決定する
/*
pcc(xi) と pcc(xj) が同一の場合はその組合せでの干渉は調べない.
(これは原論文には書かれていない)
Phi Congruence Class の性質として, 同じ Phi Congruence Class に
属している変数は, 同じ変数名に置き換えることができるというもの
がある. すなわちこの場合は xi と xj が後で同じ変数名になることが
決定している. よってもしこの状態で干渉を調べると必ず干渉して
いることになるので余計なコピー文が入ってしまう.
干渉はソースリソースの場合は LiveOut で, デスティネーション
リソースの場合は LiveIn で調べる.
ふたつのリソースが干渉している場合には, 以下の 4 通りのケース
によって挿入するコピー文を決定する.
(Live と記述がある箇所は, 場合によって LiveIn や LiveOut になる)
Case 1:
pcc(xi) と Live(Lj) に共通部分があり,
pcc(xj) と Live(Li) に共通部分がない場合.
この場合は xi'←xi というコピー文を Li に挿入するため
candidateResourceSet{} に xi を加える
Case 2:
pcc(xi) と Live(Lj) に共通部分がなく,
pcc(xj) と Live(Li) に共通部分がある場合.
この場合は xj'←xj というコピー文を Lj に挿入するため
candidateResourceSet{} に xj を加える
Case 3:
pcc(xi) と Live(Lj) に共通部分があり,
pcc(xj) と Live(Li) にも共通部分がある場合.
この場合は xi'←xi というコピー文を Li に挿入し,
かつ xj'←xj というコピー文を Lj に挿入するため
candidateResourceSet{} に xi と xj を加える
Case 4:
pcc(xi) と Live(Lj) に共通部分がなく,
pcc(xj) と Live(Li) にも共通部分がない場合.
この場合はどちらのブロックにコピー文を挿入しても干渉を
解消することができる. よって判断を保留し,
unresolvedNeighborMap(xi) に xj を,

```
            unresolvedNeighborMap(xj) に xi を追加する.
        */
      end if
    end for

5.  /*
      Case 4 で発生した未解決なリソースの処理
    */
    for each v (v は unresolvedNeighborMap をもつ変数) do
      if v ∉ candidateResourceSet{} then
        for each xi ∈ unresolvedNeighborMap(v) do
          if xi ∉ candidateResourceSet{} then
            v を candidateResourceSet{} に加える
          end if
        end for
      end if
    end for

6.  /*
      コピー文の挿入
    */
    for each xi ∈ candidateResourceSet{} do
      if xi が φ 関数のソースリソース then
        for each Lk (Lk は xi に対応する predecessor) do
          xnew_i←xi を Lk の最後に挿入
          /*
            原論文では単に at the end of Lk と書いてあるが,
            条件分岐命令がある場合には, その命令の前に挿入する
          */
          現在の φ 関数のソースで使われている xi を xnew_i で置き換える
          pcc(xnew_i) を作成し, xnew_i を入れる
          xnew_i を LiveOut(Lk) に加える
          /*
            Lj ∈ succ(Lk) ∧ xi ∉ LiveIn(Lj) ∧
            Lj 内の Lk と関連付けられた φ 関数で xi が使われていない
            ならば xi を LiveOut(Lk) から除去する.
          */
          remove←true
          for each Lj ∈ succ(Lk) do
            if xi ∈ LiveIn(Lj) then
              remove←false
            end if
            if Lj にある φ 関数のソースリソースに xi があり
              かつ, xi に対応する predecessor が Lk である then
              remove←false
```

```
            end if
          end do
          if remove が true then
            xi を LiveOut(Lk) から除去
          end if
        end for
      else /* xi が φ 関数のデスティネーションリソース */
        xi←xnew_i を L0 の先頭に挿入
        /*
          原論文では単に at the beginning of L0 と書いてあるが,
          すべての φ 関数の後に挿入する
        */
        現在の φ 関数のターゲットで使われている xi を xnew_i で置き換える
        pcc(xnew_i) を作成し, xnew_i を入れる
        xi を LiveIn(L0) から除去
        xnew_i を LiveOut(L0) に加える
      end if
      干渉グラフの更新
    end for

7.  /*
      φ 関数のすべてのリソースの Phi Congruence Class をマージする
    */
    currentPhiCongruenceClass←{}
    for each xi (xi は φ 関数のリソース, 0 ≤ i ≤ n) do
      currentPhiCongruenceClass に pcc(xi) を加える
      /*
        yj ∈ pcc(xi) としたときの
        pcc(yj) を currentPhiCongruenceClass に
        加えていない場合は, それも加える (注 1)
        原論文のアルゴリズムにはこの記述はない
      */
      for each yj ∈ pcc(xi) do
        if pcc(yj) を currentPhiCongruenceClass に加えていない then
          currentPhiCongruenceClass に pcc(yj) を加える
        end if
      end for
    end for
    for each xi ∈ currentPhiCongruenceClass do
      pcc(xi)←currentPhiCongruenceClass
      /*
        これはコピーではなく, 左辺が右辺をポインタ参照するようにする (注 2)
      */
    end for
  end for /* ステップ 2 の for each φ 関数 の終り */
```

8. for each x (x は ϕ 関数のリソース) do
   if pcc(x) がひとつのリソースしか含んでいない then
     pcc(x)←{}
   end if
  end for

(注 1) いくつかの ϕ 関数を含むプログラムを考える. まず

$$x1 \leftarrow \phi(x2, x3)$$

を処理したとする. これについてステップ 7 の処理後は

$$\begin{aligned} pcc(x1) &= pcc(x2) \\ &= pcc(x3) \\ &= \{x1, x2, x3\} \end{aligned} \tag{5.1}$$

となる. 次に

$$x4 = \phi(x5, x1)$$

を処理したとする. これについて, アルゴリズムの (注 1) の部分の処理がないと, ステップ 7 の処理後に

$$\begin{aligned} pcc(x4) &= pcc(x5) \\ &= pcc(x1) \\ &= \{x4, x5, x1\} \end{aligned} \tag{5.2}$$

となるが, これは誤りである. 正しくは

$$\begin{aligned} pcc(x1) &= pcc(x2) \\ &= pcc(x3) \\ &= pcc(x4) \\ &= pcc(x5) \\ &= \{x1, x2, x3, x4, x5\} \end{aligned} \tag{5.3}$$

となるべきである. これを行うのが (注 1) の部分の処理である. 一般に, 同じリソースが複数の Phi Congruence Class に属することはないという性質が成り立つ. なぜなら Phi Congruence Class は和集合の推移的閉包をとったものに相当するからである. 例えばリソース x1 が (5.1) と (5.2) の Phi Congruence Class に同時に属するということはない.

(注 2) このアルゴリズムでは ϕ 関数をひとつずつ処理している. (注 1) と同じ例を考える. 処理後に Phi Congruence Class が (5.3) となるように, 本実装では, 以前に処理した ϕ 関数のリソース x2, x3 も含めて pcc(x1), pcc(x2), pcc(x3), pcc(x4), pcc(x5) のすべてが, いずれも同一の {x1,x2,x3,x4,x5} というオブジェクト (リスト) をポインタで指すようにする. こうすることにより, 同じ Phi Congruence Class の実体はひとつとなり, ひとつのリソースが複数の Phi Congruence Class の実体に属することはおこらない.

LIR 上での SSA 逆変換時の問題点

Sreedhar らの方法に限らず, SSA 形式から通常形式に変換する際には, ϕ 関数のある基本ブロックを B とすると, 一般的に P $\in$ pred(B) に対して ϕ 関数のソースリソースのコピー文を挿入し ϕ 関数を除去する. そしてこのコピー文は, P にある命令列のいちばん最後に挿入されることが多い. しかし LIR では, 基本ブロックの命令列のいちばん最後は successor への JUMP/JUMPC/JUMPN 命令である. 当然挿入されるコピー文は JUMP/JUMPC/JUMPN 命令の前に挿入されるわけであるが, JUMPC/JUMPN 命令ではそこに変数の参照が存在する. つまりコピー文挿入によって JUMPC/JUMPN で参照される変数の値が変わってしまう可能性, すなわちプログラムの意味を壊してしまう可能性がある.

JUMPC/JUMPN のある基本ブロックを Bj とし, Bsj $\in$ succ(Bj) に ϕ 関数が存在するということは, 一般的に Bj $\longrightarrow$ Bsj が危険辺になっている. そこで保守的に考えれば, SSA 形式から通常形式への変換の直前で危険辺除去を行うのがよい. しかし全ての危険辺で必ずしも上記の現象が起こるとは限らない. 例えば Sreedhar らの方法では, 多くの場合危険辺を安全に取り扱うことができる. しかし特定の条件下ではそれを安全に取り扱うことができないことが確認されている.

なお危険辺に関しては 6.12 節を参照してもらいたい.

5.5 コアレシング

通常形式 LIR を SSA 形式 LIR に変換すると, 一般にひとつの変数が複数の SSA 変数[5] になり, それをふたたび逆変換すると, 多数のコピー文が作り出される. そこでこれらのコピー文で結ばれる変数をできるだけひとつの変数に割り当て, コピー文を取り除く手法としてコアレシングが有効である. コアレシングとは, そのコピー文を取り除き, コピー元変数とコピー先変数とを同一視することである. 今日までに数多くのコアレシング手法が提案されているが, 本システムでは Chaitin の手法と Sreedhar らの SSA-based コアレシングの 2 種類を実装した. Chaitin の手法はいつでも選択的に実行できるが, Sreedhar の逆変換を行ったあとでは効果がないことがある. 一方, SSA-based コアレシングは Sreedhar らの SSA 逆変換とともにしか実行できない.

本システムでは Chaitin の手法は SSA 逆変換後の通常形式 LIR に対して適用され, Sreedhar の SSA-based コアレシングは Sreedhar らの方法による SSA 逆変換時に適用される.

5.5.1 SSA-based コアレシング

例として図 5.23 に示す SSA 形式プログラムを考える. この SSA 形式プログラムは CSSA 形式であるので, そのまま ϕ 関数を除去することができる. ϕ 関数を除去した通常形式プログラムは図 5.24 になる.

図 5.24 のプログラムに対し, Chaitin のコアレシング手法を適用すると, コピー文 X←Y は X と Y の生存区間が重なっているためコアレシングできない. Sreedhar らは図 5.23 中のコピー文 X1←Y1 を除去できるような, CSSA 形式プログラム上で行うコアレシング手法を提案した [16]. この方法は Sreedhar らが提案した SSA 逆変換手法の中で用いるもので, これにより 2 つの変数 X, Y の生存区間が干渉している場合でも, ある条件が成り立てばコピー文 X←Y を除去できるようになる.

Sreedhar らの手法を以下に説明する. コピー文 X←Y について考え, X と Y は同じ Phi Congruence Class には属していないとする. コピー文 X←Y をコアレシングするときに以下の 3 つのケースを考える.

[5] SSA 変数とは静的単一代入が成立している変数である.

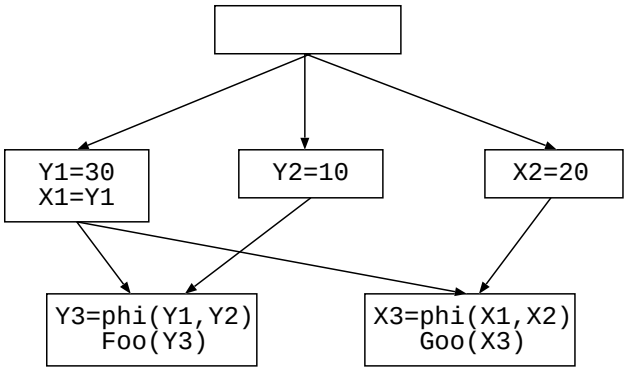


図 5.23: SSA 形式 (CSSA 形式) プログラムの例

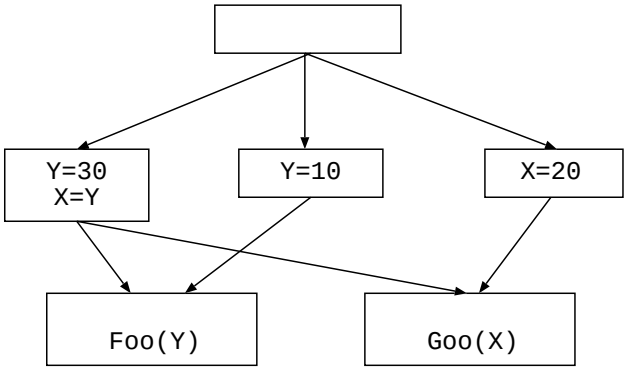


図 5.24: 図 5.23 の SSA 形式 (CSSA 形式) プログラムを通常形式に変換

Case 1: pcc(X)={ } ∧ pcc(Y)={ } の場合
この場合 X と Y はどの ϕ 関数からも参照されていないことを意味している. このコピー文は X と Y の生存区間が干渉している場合でも除去できる.

Case 2: pcc(X)={ } ∧ pcc[Y] ≠ { } の場合
もし X が集合 {pcc(Y) − Y} 中のあるリソースと生存区間が干渉する場合は X←Y を除去できない. そうでなければ除去できる.
pcc(X) ≠ { } ∧ pcc(Y)={ } の場合はこのケースと反対になる.

Case 3: pcc(X) ≠ { } ∧ pcc(Y) ≠ { } の場合
もし pcc(X) 中のあるリソースが集合 {pcc(Y) − Y} 中のあるリソースと生存区間が干渉する場合, あるいは pcc(Y) 中のあるリソースが集合 {pcc(X) − X} 中のあるリソースと生存区間が干渉する場合は X←Y を除去できない. そうでなければ除去できる.

図 5.23 の例を考える. 図から pcc(X1)={ X1,X2,X3 } および pcc(Y1)={ Y1,Y2,Y3 } であることがわかる. コピー文 X1←Y1 において, X1 と Y1 の生存区間は干渉しているが Case 3 が適用され, X1←Y1 は除去できる. コピー文 X1←Y1 を除去した後は pcc(X1) と pcc(Y1) はマージされる.

5.5.2 Chaitin のコアレシング手法

2 つの変数 a, b について, 互いの生存区間に重なりがあるとき, より正確には, 一方が生存しているときに他方への代入があるとき, a と b は干渉するという [1, 3, 8]. Chaitin が提案したコアレシ

ング手法 [8] は, コピー文 a←b において, a と b の間に干渉が無いときに必ずコピー文 a←b を除去する. そのため Chaitin の手法は「積極的」な手法と呼ばれる.

Chaitin が [8] で提案した手法は, もともとレジスタ割り当てのためのアルゴリズムである. そのため本システムではレジスタ数を ∞ とし, コアレシングできる間繰り返す. 図 5.25 に Chaitin のコアレシングアルゴリズムを示す.

```
/* ステップ 1 */
for each ブロック X do
  for each LIR node do
    if node がコピー文であり, コピー元とコピー先が干渉していない then
      テーブルにコピー文を追加する
      追加したコピー文を LIR から削除する
    end if
  end for
end for

/* ステップ 2 */
for each ブロック X do
  for each LIR node do
    if node でコピー文の左辺値を使っている then
      node で使われているコピー文の左辺値を
      コピー文の右辺値に置き換える
      (連鎖的[a] にコピーされている場合には根までたどる)
    end if
  end for
end for
```

[a] 連鎖的とは x←y; z←x; となっている場合に, z は x でなく y に置き換えられることを示す.

図 5.25: Chaitin のコアレシングアルゴリズム

Chaitin のコアレシングはどの SSA 逆変換法を適用したあとでも実行できる. しかし, 5.4.6 節で述べたように, Sreedhar の逆変換法の Method III で挿入するコピー文は, 干渉グラフに基づくコアレシングでは除去できないものであることが知られている. したがって, Sreedhar の Method III による逆変換を行ったあとに, Chaitin のコアレシングを実行しても効果はないと思われる.

第6章　SSA形式上での最適化

本章では SSA 形式上での最適化および変換について述べる. 以下, 簡単のため, 変換も含めて最適化と呼ぶ. SSA 形式上での最適化は以下のものを実装した.

- コピー伝播
- 共通部分式除去
- 質問伝播に基づく大域値番号付けと部分冗長除去
- ループ不変式のループ外追い出し
- 帰納変数の演算の強さの軽減と判定の置き換え
- 条件分岐を考慮した定数伝播
- 無用コード除去
- SSA グラフの作成と変数の等価性を見つけるアルゴリズム
- 3 番地コードへの変換
- 空の基本ブロック除去
- 基本ブロックの結合
- 危険辺除去
- Global Reassociation
- 無用 ϕ 命令除去
- Lazy Code Motion
- 要求駆動型部分無用コード除去
- コード降下および無用コード除去の繰り返し適用
- 大域ロード命令集約
- 効率的な要求駆動型部分冗長除去（EQP）
- 部分冗長除去の手法にもとづくスカラー置換 PRESR

本システムでは, 各最適化は独立したパスとなっており, COINS コンパイラドライバへのオプションによって, 任意の最適化を任意の順序で任意の回数実行することができる.

以下, 各々の最適化について述べる.

6.1 コピー伝播

コピー伝播は 4.2.2 節で示した Copy Folding と効果は同じである. つまりソースコード中の a←b のようなコピー文を除去する手法である. 一般的に, コピー伝播では各変数の生存区間を調べ, そのコピー文以降に使用される代入先変数 (a←b の場合には a) を, その変数が生きている間 (つまり a に対して新たな代入文がない間) 代入元変数 (a←b の場合には b) に置き換える. しかし SSA 形式の場合は, 各変数の値は静的に単一である保証があるため変数の生存区間を調べる必要はなく, 単純に変数の置き換えをすればよい. 4.2.2 節で示したものが SSA 変換中のものであったのに対し, ここでいうコピー伝播は, 最適化として独立しているものである.

コピー伝播は 2 つのステップにわかれている. まずステップ 1 でコピー文を発見し, ステップ 2 で変数の名前替えを行う. コピー伝播のアルゴリズムを図 6.1 に示す.

```
/* ステップ 1 */
for each ブロック X do
  for each LIR node do
    if node がコピー文である then
      テーブルにコピー文を追加する
      追加したコピー文を LIR から削除する
    end if
  end for
end for

/* ステップ 2 */
for each ブロック X do
  for each LIR node do
    if node でコピー文の左辺値を使っている then
      node で使われているコピー文の左辺値を
      コピー文の右辺値に置き換える
      (連鎖的[a] にコピーされている場合には根までたどる)
    end if
  end for
end for
```

[a] 連鎖的とは x←y; z←x; となっている場合に, z は x でなく y に置き換えられることを示す.

図 6.1: コピー伝播のアルゴリズム

6.2 共通部分式除去

ある式 X op Y (X, Y に対する演算の式) をプログラム上のある点で計算する場合, その点へ至るまでの間に必ずその式が計算されていることがわかれば, その点での計算を冗長な式であるとして省略することができる. データの流れの等式を使って冗長な式を求めるためには, まず, ある点で計算されている式, あるいは利用されている式を求める必要がある. 以下に SSA 形式による共通部分式除去のアルゴリズムを述べる [19].

例として, 以下のプログラムを考える.

```
1) a = b + c
2) d = b
3) e = b + c
4) a = c + d
```

このプログラムを SSA 形式に変換する.

```
1)  a0 = b0 + c0
2)  d0 = b0
3)  e0 = b0 + c0
4)  a1 = c0 + d0
```

式と変数名の対応表およびコピーの関係にある変数の対応表を用意する. 1) および 2) を処理したところで, これらの表は表 6.1 のようになる.

表 6.1: 共通部分式除去：1) と 2) の処理後

式	変数名
b0+c0	a0

変数名	変数名
d0	b0

表 6.1 を用いて, 3) の処理では, 右辺の式が既に計算されて a0 の値となっているのがわかり, 4) の処理では d0 が b0 のコピーであることから同様のことがわかる. したがって, 3) および 4) の処理後, 式と変数名の対応表およびコピーの関係にある変数の対応表は表 6.2 のようになる.

表 6.2: 共通部分式除去：3) と 4) の処理後

式	変数名
b0+c0	a0

変数名	変数名
d0	b0
e0	a0
a1	a0

共通部分式除去後のプログラムは次のようになる.

```
1)  a0 = b0 + c0
2)  d0 = b0
3)  e0 = a0
4)  a1 = a0
```

SSA 形式での共通部分式除去は, 合流点にある ϕ 関数を利用してプログラム全体に適用することができる. あるブロックに, ある変数の値が伝えられるのは, そのブロックを直接支配するブロックからであるか, あるいはそのブロックの先頭にある ϕ 関数を通してかのどちらかである. したがってこの処理は支配木の根から葉に向かって行うのがよい.

共通部分式除去を実行するアルゴリズムを図 6.2 に示す. 式と変数名の対応は「変数名 ←NameExp(式)」の形で表し, 変数名と変数名の対応は, a1 が a0 のコピーであるとき a1←Copy(a0) と表す. 図 6.2 で「 V←Copy(V1) とする」というのは, V は V1 のコピーであるということを表に書き込むことを意味する. 同様に「 V←NameExp(e) とする」というのは, 式 e とそれに対応する変数名 V を表に書き込むことを意味する. ϕ 関数もひとつの式である.

現状の実装では, 式 V←X op Y において, 左辺 V が REG 式であり, 右辺 X op Y には MEM 式を含まないものを表に書き込む. また, 右辺 X op Y が定数や関数呼び出しのの場合には表に書き込まない. ただし第 7 章で述べる簡単な別名解析を行う場合には右辺 X op Y に MEM 式を含む場合でも表に書き込む.

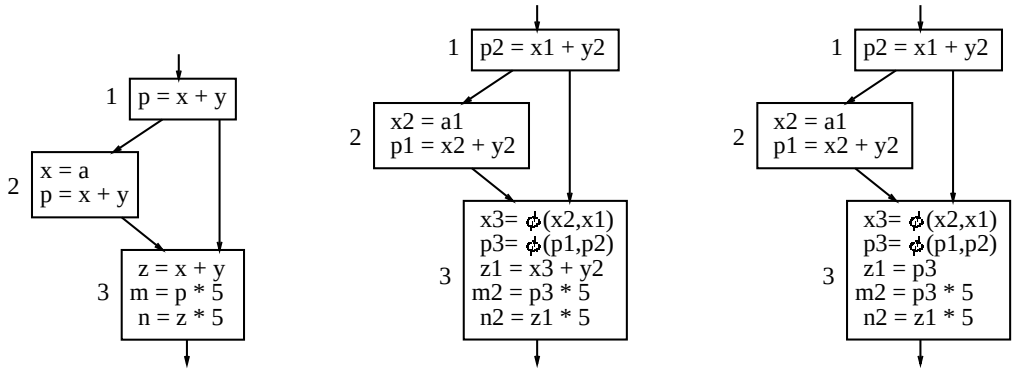
```
cse(ブロック B) /* 手続き cse */
  for each φ 関数 in B : V← φ(···) do
    if 右辺と同じ φ 関数 e が表にある then
      V←Copy(NameExp(e)) として, この文を除去する
    else if φ 関数の引数はすべて同じ変数 V1 である then
      V←Copy(V1) として, この φ 関数を除去する
    else if φ 関数の引数はすべて同じ定数 C である then
      その φ 関数を除去する
      そのブロックの先頭にあるすべての φ 関数の直後に V←C を挿入する
    end if
  end for
  for each 文 S in B do
    if S が CALL 式を含まない定義文: V←X op Y then
      if S に X←Copy(X0) なる変数 X が使用されている then
        X を X0 で置き換える
      end if
      if S に部分式 X op Y で Z←NameExp(X op Y) なるものがある then
        部分式 X op Y を Z に置き換える
      end if
      if S がコピー文 V←V1 である then
        V←Copy(V1) として S を除去する
      end if
      if V が REG 式である then
        if 式の右辺 e に MEM 式を含んでいない then
          V←NameExp(e) とする
        else if 別名解析を行っている then
          V←NameExp(e) とする
        end if
      end if
    else /* S が CALL 文を含む定義文, または定義文以外の文である場合 */
      if S に X←Copy(X0) なる変数 X が使用されている then
        X を X0 で置き換える
      end if
      if S に部分式 X op Y で Z←NameExp(X op Y) なるものがある then
        部分式 X op Y を Z に置き換える
      end if
    end if
  end for
  for each ブロック C∈succ(B) do
    for each φ 関数 in C do
      φ 関数の引数 V で V←Copy(V1) なるものがあったら,
      それを V1 で置き換える
    end for
  end for
  for each ブロック C ∈ domChild(B)    do
    cse(C)
  end for
  このブロック B で対応表に書き込んだものは除去する
```

図 6.2: 共通部分式除去のアルゴリズム

6.3 質問伝播に基づく大域値番号付けと部分冗長除去

SSA 形式では, プログラム中で同じ名前をもつ変数は同じ値をもつことが保証されているので, ある式の値があるプログラム点において利用可能かどうかを判断するのは通常形式と比べて容易である. しかしながら, 通常形式上で利用可能であると判断される式でも SSA 形式上では, 利用可能であるとみなされない場合がある.

例えば, 図 6.3(a) において, `x+y` の値は, 節 3 の入口で利用可能であり, 節 3 の代入文 `z=x+y` の右辺は, 変数 `p` によって置き換えることができる. 一方, 図 6.3(a) に対する SSA 形式である図 6.3(b)



(a) Normal form (b) SSA form (c) Eliminating redundant expressions

図 6.3: SSA 形式上での式の利用可能性

では, 節 3 の入口において, `x+y` に対応する `x3+y2` は, 利用可能ではない. この違いは, 辺 $(2,3)$ から到達する変数 `x` と辺 $(1,3)$ から到達する変数 `x` が, それぞれ異なる定義をもつことから, SSA 形式において異なる変数名 `x2`, `x1` にそれぞれ変更されることから生じる. 最終的に, これらの変数の値は, ϕ 関数によって `x3` に代入され, 節 3 の入口に到達するので, 辺 $(2,3)$, 辺 $(1,3)$ 上で, `x2+y2`, `x1+y2` がそれぞれ利用可能であるにも関わらず, 節 3 の `x3+y2` は利用可能であるとみなされず, 共通部分式として除去することができない.

この SSA 形式における利用可能性の問題を解決するために, Rosen らは, 質問伝播 [15] を提案している. 質問伝播は, "式 e の値が利用可能かどうか" という質問を後向きに尋ねていき, 最終的に, e が特定のプログラム点において利用可能であるかどうかを $true$ あるいは $false$ として得る解析である. もし, 質問が e のオペランドを定義する ϕ 関数に到達した場合は, e のオペランドをその ϕ 関数の引数で置き換えて, さらに各先行節に伝播させる. 質問が到達した節 n が, 式 e が存在する節か, e と同じ式の質問が既に到達している節であった場合, その質問の答えは $true$ となり, 伝播は終了する. また, n が, 開始節か, 既に e と異なる式の質問が到達している節か, e のオペランドの定義（ϕ 関数を除く）が存在する節であった場合には, その答えは $false$ になり, 答えが $true$ であったときと同様に, 伝播は終了する. 伝播してきた各節における質問の答えは, 先行節が 1 つの場合, 先行節と一致した結果になる. 先行節が複数存在する節に関しては, すべての先行節において答えが $true$ となった場合にだけ $true$ となり, それ以外の場合は $false$ となる.

例えば, 図 6.3(b) において, 節 3 にある `x3+y2` に関する質問伝播を考える. 節 3 には `x3` を定義している ϕ 関数があるので, オペランドを変更し, 式 `x2+y2`, `x1+y2` の質問をそれぞれ節 2, 1 に伝播する. 最終的に, 節 2, 1 において, `x2+y2`, `x1+y2` のそれぞれの値が利用可能であることが分かるので, 節 3 のすべての先行節における質問の答えが $true$ となり, `x3+y2` は, 利用可能であることが分かる, 結果として, `x3+y2` は, 共通部分式であり, `p3` によって置き換えることができる（図 6.3(c)）.

6.3.1 質問伝播を用いた共通部分式の除去

質問伝播は, 利用可能性の判定を行う解析であるが, 到達する変数を管理するためのスタック操作を加えることによって, SSA 形式を保持したまま, 共通部分式の除去を行うことができる.

CFG 節 w の式 e に対する共通部分式の発見で行うスタック操作は, 次のとおりである.

1. 質問の式が出現する節に達した場合, e の代入先変数をスタックに積む.
2. 訪問済みの節 v に再び到達した場合, 新しい一時変数 t を生成し, t をスタックに積む. 同時に, 以降 t を使用できるように右辺が未定義の代入文 $t \leftarrow \perp$ を生成し, 配列 $insertCpy$ の要素 $insertCpy[v]$ に記録しておく.
3. 節 v から先行節を検査した結果が, すべて $true$ であった場合, 先行節の数だけスタックから変数を取り出し, その変数を引数として ϕ 関数を作成する. ϕ 関数の代入先は, 新しい一時変数 t とする. もし, v への戻り辺が存在するなら, 対応する引数を t にする.

   作成した ϕ 関数を v の入口に挿入し, t をスタックに積む. $insertCpy[v] \neq \emptyset$ なら, 各 $t' \leftarrow \perp \in insertCpy[v]$ の右辺を t で置き換えたコピー代入 $t' \leftarrow t$ を生成し, v の出口に挿入する.

最終的な答えが $true$ であった場合, スタックの一番上にある変数によって, v に含まれる e の右辺を置き換える. もし, 答えが $false$ であった場合, 挿入した文をすべて削除する. $insertCpy$ に含まれるコピー代入は, 実際に挿入せずに, 伝播してしまうこともできる.

以上のように, 質問伝播は, 特定のプログラム点について特定の式の利用可能性を必要に応じて計算し, 利用可能な場合には除去することができるが, 部分冗長除去のような式の挿入をともなう利用可能性は計算できない. また, すべての式の出現に対して, 質問伝播を適用するとコストが大きくなる. 滝本らは, より効率的で式の挿入を扱える**効率的な質問伝播**（Efficient Question Propagation, 以降 EQP と呼ぶ）を提案した [17]. 以降で, 滝本らのアルゴリズムを説明する.

6.3.2 ランク

滝本らは, 質問伝播のコストを下げるために, 節 v から v の支配節 d に直接質問伝播させる方法を用いている. その際, d に直接伝播してよいかどうかを**ランク**に基づいて動的に検査する.

ランクは, CFG の深さに相当する値であり, CFG 節 v とその支配節 d を通る実行パス上に, ある性質を満たす節が存在するかどうかを調べたい場合, 性質を満たす節が存在するランクを記録しておき, d のランクより大きく v のランクより小さい範囲に, 記録したランクが存在するかどうかを検査するために利用する. このランクに基づく方法は, 各節を訪問して検査する場合に比べて, 検査の回数を少なくすることができるだけでなく, ある性質を満たすランクをそのランクに対応するビット位置に 1 をセットしたビットベクタとして記録することによって, 1 ワード分の範囲を一度に検査する高速な判定を可能にする.

ランクは, CFG の節 v に対して, 以下の関数 $rank(v)$ を用いて再帰的に定義される.

定義 1 (ランク)

1. もし v が開始節なら, $rank(v) = 0$,
2. さもなければ,

$$rank(v) = max(\{rank(p) \mid p \in pred(v) \wedge (p, v) \neq \textit{up-edge}\}) + 1$$

ここで, $max(S)$ は, 集合 S の最大要素を表す. また, $up\text{-}edge$ は, CFG の深さ優先完全木において, 祖先に向かう辺である. この辺を**上昇辺**と呼ぶ.

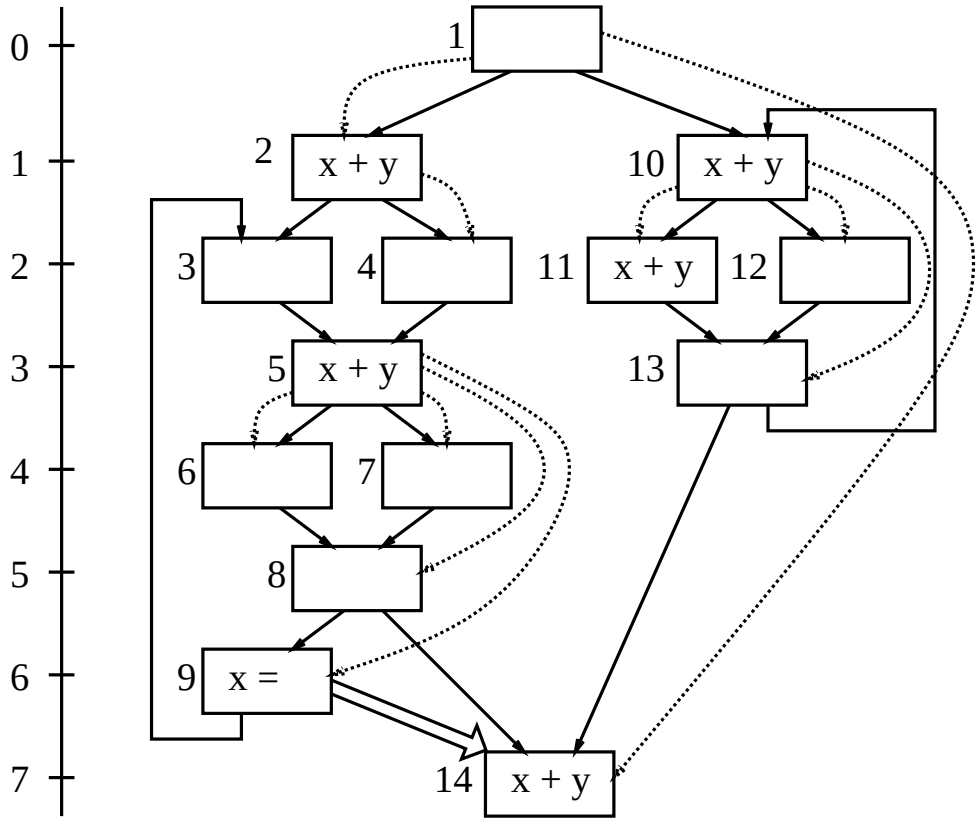


図 6.4: 補助辺と直接の伝播先を示す辺

ここで, CFG 節 v とその支配節 d について次のことが言える. 節集合 $V = \{x | rank(d) < rank(x) < rank(v)\}$ の中に, V に含まれない節を出元とする上昇辺が指す節も, 質問の式が出現する節も存在せず, v 自身も上昇辺が指す節でない場合, 節 v から先行節へ伝播される質問は, 節 d へ直接伝播させることができる [17].

ランクを CFG 上で計算するためには, 制御が戻ってくる辺である上昇辺を取り除かなければならない. ランクの定義が表している $(p, v) \neq \textit{up-edge}$ は, このことを表している. しかしながら, 上昇辺の除去は, ランクの計算に不都合を生じる場合がある. 例えば, 図 6.4 の CFG に対して, 上昇辺 $(9, 3)$ と $(13, 10)$ を取り除いてランクを計算すると, 節 14 より必ず先に実行される節 9 が節 14 と同じランクになってしまう. これでは, 節 2 と節 14 によって挟まれる節の検査をランクによって行えなくなってしまう.

そこで, ランクを計算する前に, CFG に対して, 上昇辺の元から その上昇辺を含むループ出口の後続節に補助辺を付加する. 図 6.4 に示す CFG に必要な補助辺は, 2 重線 $(9, 14)$ によって示す. 図 6.4 の左には, 補助辺を付加した CFG 上でランクを計算した結果を示してある.

6.3.3 短絡節と短絡辺

節 v の質問が v の支配節 d に直接伝播可能なとき, d を v の**短絡節**と呼び, v から d への伝播可能性を明示した辺 (d, v) を**短絡辺**と呼ぶ. 質問伝播は, 質問の答えがどこかの CFG 節で false になるか, すべての伝播先の CFG 節で true になった場合に即座に伝播を終了できるので, 伝播する範囲を局所化できる特徴がある. しかし, そのいずれでもなかった場合, 質問は開始節まで伝播してしまい, 伝播する範囲を局所化できない. この局所化できない場合を効率的に扱うために, 短絡節として, 開

始節に近い CFG 節を採用する．ループの内部から外部への直接伝播は, 前述のランクに基づく検査の条件に反していることから, 節 v の短絡節を, 節 v を最も内部で囲むループの入口に最も近いループ内の支配節とする．v を囲むループが存在しない場合は, 開始節を短絡節とする．例えば, 図 6.4 の短絡辺は, 点線矢印であり, 短絡辺の元が短絡節である．

6.3.4 効率的な質問伝播

EQP は, 部分冗長除去を含む大域値番号付けを実現するために, 次の点で質問伝播を拡張している．

1. 先行節へ伝播した質問の中で, 一部の先行節（すべての先行節ではない）の答えが $false$ であったとき, その先行節に, 質問の式を挿入する．
2. 式を挿入する際には, どの実行パス上においても式の数が増えないことを保証するために, 挿入する節から, 前向きの質問伝播を行う．ここで, 前向きの質問伝播が返す結果は**予期可能性**と呼ばれる．実際に挿入を行うのは, 前向きの質問伝播が $true$ を返した場合だけである．

以降で, EQP を用いた大域値番号付けのアルゴリズムを, 図 6.6, 6.7, 6.8 に示した仮想コードに基づいて示す．

図 6.6 に示すように, EQP は, 式 e が 節 v に伝播可能かどうかを検査する 関数 eqp と 実際に伝播を行う関数 propagate の相互再帰で実現される．eqp(e,v) は, 関数 isOccurWithStackPush によって節 v に e の出現を発見するか（1–2 行目）, v が以前に訪問した節であったか（6–9 行目）, 利用可能変数が availVar[v] に求まっているか（11 行目）のいずれかの場合に, その後の伝播を許し, propagate を呼ぶ．ここで, isOccurWithStackPush は, e の出現を検査するだけでなく, 発見した式の代入先をスタックに プッシュ する．一方, 12 行目に示すように, v が開始節であるか, オペランドへの変更を検査する関数 isMod が true であるか, 以前訪問した際の対象の式が異なるか（visited[v]$\neq \perp$）のいずれかの場合には, この時点で false を返す．

節 v の式 e に対する EQP は, propagate(e,v) を呼び出すことによって実行する．以降, propagate の処理を中心に説明する．

伝播方法

関数 propagate における質問の伝播は, 17–29 行目の前半部分と 30–41 の後半部分とに分けられる．前半部分は, 22 行目の関数 rankCheck の呼出しによって, 短絡節 shortCutTarget[v] と v との間の実行パス上に, e が出現するかどうかを, 短絡節の ランク lower と v の ランク upper との間の e の ランク の存在を調べることによって検査する．もし, lower と upper との間 に e の ランク が存在しなければ（rankCheck(e,lower,upper)=false）, 質問は, 短絡節に直接伝播される（23 行目）．19 行目に示すように, 実際の lower は, rankCheckTarget[v] の ランク であるが, ほとんどの v において, rankCheckTarget[v] と shortCutTarget[v] は同じである．これらが, 異なる節を指す場合については, 21 行目で扱う lower と upper との大小の逆転とともに, 可約ループに関する効率化で後述する．以上の短絡節への伝播は, ϕ 関数によるオペランドの付替えが生じない場合に可能である．

前半部分で伝播されなかった場合, 後半部分で, 各先行節へ伝播させる（30 行目）．その際, 31 行目に示すように, 関数 transPhi によって, オペランドの付替えが必要かどうか検査し, 必要な場合には, 式を変形する．先行節への伝播は, 32 行目の eqp の呼出しによって行う．eqp が先行節 p での呼出しについて false を返した場合は, p に対象の式の挿入を試みる．このとき, 挿入が可能なのは, 節 v から終了節へのすべての実行パス上に, 対象の式 e が存在するという予期可能性を満たす場合だけである．そこで, 33 行目に示すように, 逆向きの質問伝播を行う関数 postEqp を用いて, 予期可能性を検査する（33 行目）．

(a) Availability through loops　　(b) Hoisting through loops

図 6.5: ループを通過する利用可能性と巻上げ

postEqp の仮想コードを, 図 6.7 に示す. postEqp は, 式が予期可能かどうかを検査し, true か false を返す関数なので, eqp と伝播方向が逆の propagate を合わせて, 単純化した形をしている. postEqp における 式の出現を検査する関数 isOccur と オペランドへの変更を検査する isMod の位置が, eqp と異なるのも, 伝播方向の相違によるものである.

propagate の説明に戻ると, postEqp が true を返した場合, 新しい一時変数を t として, t←e を節 p に割り当て, t は利用可能な変数としてスタックに プッシュ する. t←e を 配列 tmpInsert[p] に付加する部分が, 節 p への割当てを意味している. tmpInsert[p] は, createPhiAndIsItAvail で v における利用可能性が決定されたのち挿入候補として insertOut[p] に付加される. 一方, postEqp が false を返した場合は, スタックトップにある変数は利用可能でないので, スタックを ポップ する.

新しいφ関数の作成と利用可能性

従来の質問伝播では, 32 行目の eqp の呼出しが, 実際の利用可能性を意味していたのに対して, EQP では, 単に式の挿入を行うかどうかを意味する. 実際の利用可能性は, スタック上の各要素に付加した利用可能性の情報によって決まる. これは, 部分冗長除去の式の挿入から生じる可能性がある**ループの通過問題**を避けるためである.

例えば, 図 6.5 (a) の節 3 で 式 `x+y` は利用可能である. したがって, 3 から開始する質問伝播は true でなければならないので, 節 2 からさらに伝播する質問は, すべての先行節について true を返す必要がある. 質問伝播では, すでに訪問した節に再度訪問した場合, 以前の質問が現在の質問と同じなら (4 行目：visited[v]=e) true を返す. この方法は, 節 2 の戻辺に沿った質問伝播の結果を true にするので, ループを通過する利用可能性を扱うことができる. 一方, 図 6.5 (b) を考えると, 質問が節 2 に到達したのち, 先行節への伝播において, 戻辺から true が帰ってくることを考えると, EQP の挿入の条件では, 節 1 に式を挿入することになる. しかし, この挿入は, 冗長性を除去する効果をもたないので, 不要な挿入を生じることになる.

EQP では, スタックに利用可能変数をプッシュする際に, 属性として利用可能性 AV と**弱利用可能性** WAV を同時にプッシュする. AV は, 1 行目で式の出現を見付けたときに関数 isOccurWithStackPush によって true としてプッシュされるか, 11 行目で計算済みの利用可能変数を発見したときに true としてプッシュされ, それ以外は false としてプッシュされる. 一方, WAV は, AV=true か, 8 行目で以前訪問したが答えが求まっていない節を再度訪問した場合に, true としてプッシュされ, そうでなければ false である.

スタック上に存在する v の先行節の情報 AV, WAV は, 関数 createPhiAndIsItAvail によってポッ

プされ, 次の方程式によって, v の利用可能性を決定するために利用される.

$$WAV_v = AV_v \vee \prod_{v' \in pred(v)} WAV_{v'}$$
$$AV_v = \sum_{v' \in pred(v)} AV_{v'}$$

計算された v の AV と WAV は, 新たにスタックにプッシュされれ, WAV の値は, createPhiAndIsItAvail の返り値となる. また, createPhiAndIsItAvail は ポップ した利用可能変数から ϕ 関数を生成し, 挿入する文の候補として insertIn[v] に付加する. 生成された ϕ 関数の左辺は, 計算した利用可能性とともにスタックに積む. 先行節が 1 つの場合には, ϕ 関数を生成せず, スタックの内容も createPhiAndIsItAvail を呼ぶ前の状態と同じであることに注意が必要である.

解析範囲

EQP は, ランクを用いた効率化以外に, 計算済みの結果を利用して不要な伝播を行わない効率化を行っている. 計算結果は, 利用可能変数として, CFG 節をインデックスとする配列 availVar に記録している. 節 v に対する availVar[v] は, 関数 propagate によって次のように設定される.

return true の直前 ： top() によって, スタックトップの変数が設定される. ただし, スタックトップの利用可能情報が false の場合には, top() は $\perp$ を返す.

return false の直前 ： $\top$ が設定される.

availVar は, 関数 eqp によって次のように利用される.

availVar[v]=$\perp$**の場合** ： t$\leftarrow \perp$ を生成し, 挿入する文の候補として insertCpy[v] に加え, t をスタックにプッシュ したのち true を返す. t$\leftarrow \perp$ の右辺は, v における利用可能変数が決定されたのちに関数 replaceBottom（24, 43 行目）によってスタックトップと置き換えられる.

availVar[v]=$\top$**の場合** ：単に false を返す.

その他の場合（availVar[v] **が変数である場合）** ：availVar[v] をスタックに プッシュ したのち true を返す.

図 6.7 の postEqp における isAnt も, availVar と同様, 計算済みの予期可能性を保持するが, isAnt が伝播を終了させるのは, 値が true のときだけである. これは, 後述する EQP の適用順序から, 処理が進むにつれて各式の予期可能な節が増えていく可能性があるからである.

可約ループに関する効率化

入口が 1 つのループは, **可約ループ**と呼ばれる. 可約ループの入口節 h から伝播された質問は, 戻辺に沿って伝播されたものだけが h に戻ってくるので, ループ内に対象の式が存在しなければ, h の質問は, そのままループの外に伝播してよいことになる. したがって, 訪問中の節が可約ループの入口 h であった場合には, h のランクと すべての戻辺の出元のランク中で最大のランク $r_{maxBackEdge}$ との間で rankCheck による検査を行う. もし, 結果が false の場合には, 戻辺に沿った伝播を行わずに直接 h の先行節に伝播させることができる. この効率化は, rankCheckTarget[h] を $r_{maxBackEdge}$ に設定しておくだけで容易に実現することができる. この場合, lower と upper の値が逆転してしまうので, 図 6.6 の 21 行目のように, 両者をスワップする必要がある.

6.3.5 EQP を用いた大域値番号付け

EQP を用いた大域値番号付けを, 仮想コードとして図 6.8 に示す. 大域値番号付けにおいて重要な点は, 冗長な式を除去したことによって新たに明らかになる冗長な式を, 除去の対象にしていくということである. 14–16 行目に示すように, 本大域値番号付けでは, EQP による冗長な式の除去とコピー伝播を交互に行うことによって, 新たに冗長な式を発見できるようにする. また, 2 行目に示すように, 新たに冗長であると判明する式を除去の対象にしていくために, 各式への EQP の適用は, ランク の順に CFG 節を訪問して行う. ここで, 各 EQP の適用の直後には, insertIn と insertOut に含まれる文を実際に挿入する処理が必要である（10–13 行目）. 一方, insertCpy 中のコピー代入は, insertIn 中の ϕ 関数に伝播しておくことができる（5–9 行目）.

```
eqp(e,v)
1   if isOccurWithStackPush(e,v)
2     availVar[v] = top(); return(true)
3   end if
4   if visited[v]=e then
5     avVar←availVar[v]
6     if (avVar=⊥)
7       t←⊥ を insertCpy に加える（t は新しい一時変数）
8       push(t,false,true) // 属性 AV = false, WAV = true も同時に積む
9       return(true)
10    else if (avVar=⊤) return(false)
11    else push(avVar,true,true); return(true) // 属性 AV = true, WAV = true も同時に積む

12  if v が 開始節である ∨ isMod(e, v) ∨ visited[v]≠⊥) then
13    return(false)
14  end if
15  return propagate(e,v);

propagate(e,v)
16  visited[v]←e
17  if ¬modPhi(e, v) then
18    target←shortCutTarget[v]
19    lower←rank[rankCheckTarget[v]]
20    upper = rank[v]
21    if lower>upper then swap(lower,upper) end if
22    if (¬rankCheck(e,lower,upper))
23      if (eqp(e,target)
24        replaceBottom(v,top())
25        availVar←top(); return(true)
26      else return false
27      end if
28    end if
29  end if
30  for each ブロック p ∈ pred(v) do
31    e'←transPhi(e,p,v)
32    if (¬eqp(e',p))
33      if (postEqp(e,v))
34        t←e を tmpInsert[p] に加える（t は新しい一時変数）
35        push(t,false,false)
36      else
37        pred(v) の数だけ pop() する
38        availVar[v]← ⊤; return false
39      end if
40    end if
41  end for
42  if (createPhiAndIsItAvail(v,tmpInsert)) // スタックの属性 AV, WAV の計算も同時に行う
43    replaceBottom(v,top())
44    availVar[v]←top(); return(true)
45  else availVar[v]← ⊤; return(false)
46  end if
```

図 6.6: EQP のアルゴリズム

```
postEqp(e,v)
1  if isAnt[e,v]∨visited'[v]=e return(true) end if
2  if v が CFG の終了節である ∨ isMod(e, v) then
3    return(false)
4  end if
5  if isOccur(e,v) then return(true)
6  else if visited'[v]≠ ⊥ then return(false)
7  end if
8  visited'[v]←e
9  for each ブロック s ∈ succ(v) do
10   e'← transPhi'(e,v,s)
11   if(¬postEqp(e',s)) return(false) end if
12 end for
13 isAnt[e,v]←true; return(true)
```

図 6.7: EQP のアルゴリズム (続き)

```
gvn()
1  for r = 0 to ランクの最大値 do
2    for ランク r をもつ各 CFG 節 b do
3      for v←e の形をした各文 s in b do
4        if ¬isMod(e)∧ propagate(e,b)
5          for each d←s∈insertCpy do
6            for each CFG 節 v do
7              insertIn[v] 中の φ 関数の d の出現を s で置き換える
8            end for
9          end for
10         for each CFG 節 v do
11           insertIn[v] 中の φ 関数を v の入口に挿入
12           insertOut[v] 中の文を v の出口に挿入
13         end for
14         top() で得られる変数を newSrc として
15         v←e の e を newSrc で置き換える
16         v←newSrc をコピー伝播する
17       end if
18     end for
19   end for
20 end for
```

図 6.8: EQP を用いた大域値番号付けのアルゴリズム

6.4 ループ不変式のループ外追い出し

ループ最適化として, ループ内で不変な式をループ外に追い出すものがある. ループ不変式とは, ループ内に存在する式 e : $t \leftarrow a_1$ 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.

1. a_i が定数である
2. a_i の定義がループ外にある
3. a_i を定義する文にループ不変の印がついている

ループ内の各式が不変式かどうかを調べるには, 上記の性質があてはまるかどうかをループ内の全ての式に対して調べればよい.

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]. Appel らはこの条件を緩和し, 不変式を含むブロック B が, 不変式によって定義される変数 t を Live Out に含む全てのループ出口ブロックを支配しなければならないとしている [3]. 本実装では, Appel の条件で実装を行った. 図 6.9 にループ不変式のループ外追い出しのアルゴリズムを示す.

```
for each ループ構造 L in CFG do
    for each ブロック B in L do
      ループ不変式 e をみつける
      if e が割り算等でないなら then
        e で定義される変数を def(e) とする
        def(e) が Live Out する出口ブロックを exit(e) とする
        if 全ての E ∈ exit(e) に対して B ∈ dom(E) である then
          e を L のプリヘッダ P に追い出す
        end if
      else
        L の出口ブロックを exit(L) とする
        if 全ての E ∈ exit(L) に対して B ∈ dom(E) である then
          /* 全ての出口を支配する，が追い出し条件 */
          e を L のプリヘッダ P に追い出す
        end if
      end if
      /* Dragon Book によると本来こうすべきである */
    end for
end for
```

図 6.9: ループ不変式のループ外追い出しアルゴリズム

ループ不変式をループ外に追い出す場合, その不変式 e で定義される変数 def(e) を Live Out に含むループ出口ブロック exit(e) を, e を含む基本ブロック B が全て支配する必要がある. よって, while 型のループ, つまりループの入口ブロックと出口ブロックが同じブロックである形のループでは, 入口ブロック以外のブロックから不変式を追い出すことができない. そこで, この最適化を行う前に, while 型のループを do-while 型のループ, つまりループの入口ブロックと出口ブロックが異なるブロックである形のループにループ構造を変更することが望ましい. 本システムでは, 最適化のひ

とつとして, このループ構造変換を実装している. ループ構造変換については 3.1 節を参照してもらいたい.

6.5 帰納変数の演算の強さの軽減と判定の置き換え

演算の強さの軽減とは, コンパイラが出力するコードにおいて, コストの高い演算を同じ意味のコストの低いものに置き換える最適化である [19, 11]. 本システムでは, Cooper らの論文 [11] に基づき, 基本帰納変数及び一般の帰納変数の演算の強さの軽減を実装した.

また, 基本帰納変数及び一般の帰納変数の最適化を行うことによりループの終了判定式が置き換えられる場合がある [19, 11]. 本システムではこれも実装した.

Cooper らの手法 (以下 OSR : Operator Strength Reduction) では, 直接 CFG に対して最適化を行うのではなく SSA グラフに対して最適化を行う. SSA グラフを用いることにより, データの使用から定義へのリンクが明確になり, アルゴリズムが理解しやすいものとなる. SSA グラフに関しては 6.8 節を参照してもらいたい.

6.5.1 帰納変数

帰納変数 (induction variable) とは, ループ内で定数刻みで値が変わる変数である. ここで定数とはループ内で不変な値を持つものである [19]. SSA 形式では, 図 6.10 の形をしている変数 i は基本帰納変数である. 図 6.10 では, i0 はループに最初に入るときの値, i2 はループをまわってきたときの値, RC はループ不変式である. i が基本帰納変数であるとき, その i の 1 次式

$$j \leftarrow RC_1 * i + RC_2$$

は一般の帰納変数である. ただし, RC_1 および RC_2 はループ不変式である. OSR はこのような基本帰納変数及び一般の帰納変数に対して最適化を行う.

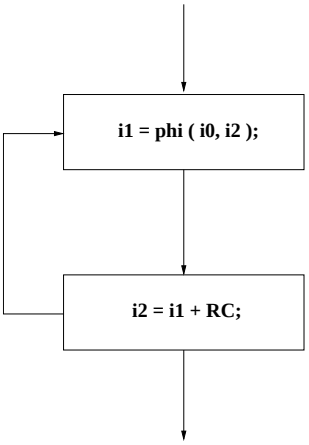


図 6.10: 帰納変数の例

一般的に, すべての基本帰納変数は SSA グラフ内で強連結成分 (Strongly Connected Component : SCC) の形をしている. ただし, すべての SCC が基本帰納変数であるとは限らない. 基本帰納変数集合は SCC 集合の部分集合であり, あるプログラムの SSA グラフ中のすべての SCC の中には, そのプログラムの基本帰納変数が必ずすべて含まれていることがいえる. そこで OSR では, SSA グラフ中から SCC を抽出・検査することにより基本帰納変数を発見する. もし, ある SCC 内のすべての演算が $IV + RC$, $IV - RC$, $COPY$ 演算, ϕ 関数[1] であれば, その SCC は基本帰納変数に対応するものである. ここで IV は基本帰納変数を, RC はループ不変値を示す. 詳しくは Cooper らの論文 [11] を参照してもらいたい.

[1] ϕ 関数は, 全ての引数が SCC 内部に含まれているかループ不変値である必要がある.

6.5.2 演算の強さの軽減

OSR は SSA グラフに対して深さ優先探索を行い, SSA グラフから SCC を発見する. 図 6.11 に OSR のドライバ部分と深さ優先探索の部分のアルゴリズムを示す.

```
OSR(SSAgraph)
  while まだ辿られていないノード n が SSAgraph 中にある do
    call DFS(n)
  done

DFS(node)
  node に DFSnum(DFS 番号) をつける
  node に辿った印をつける
  node.low←node.DFSnum
  PUSH(node)
  for each 「node のオペランド o」 do
    if まだ o を辿っていない then
      call DFS(o)
      node.low←MIN(node.low,o.low)
    end if
    if (o.DFSnum < node.DFSnum ∧ o がスタックに含まれている) then
      node.low←MIN(o.DFSnum,node.low)
    end if
  end for
  if node.low と node.DFSnum が等しい then
    SCC を空のリストとする
    do
      x←POP()
      SCC に x を加える
    while (x ≠ node)
    call ProcessSCC(SCC)
  end if
```

図 6.11: OSR : ドライバ部分と深さ優先探索

深さ優先探索を用いて SCC を発見したところで, OSR ではその SCC が基本帰納変数かどうかを検査する. 図 6.12 にそのアルゴリズムを示す. 基本帰納変数を発見した場合は, その変数を IV として登録し, また SCC が基本帰納変数でない場合には, その演算の強さを軽減可能かどうかをチェックして, 可能であれば演算の強さの軽減を行う.

演算の強さの軽減は, 基本的に以下の 3 つの関数からなる.

- *Replace*
  現在の演算を, 強さを軽減された演算に対する $COPY$ に置き換える
- *Reduce*
  基本帰納変数及び一般の帰納変数に対する演算の強さの軽減を行うコードを挿入し, その結果の SSA 変数を返す
- *Apply*
  あたえられたふたつのオペランドに, あたえられた演算子を適用する命令を挿入し, その結果の SSA 変数を返す

図 6.13 に演算の置き換えに関するアルゴリズムを, 図 6.14 に演算の適用に関するアルゴリズムを示す.

```
ProcessSCC(SCC)
  if SCC が単一のメンバ n しか持たない then
    if n が置き換え可能な演算である (注) then
      call Replace(n,IV,RC)
    else
      n.header←NULL
    end if
  else
    call ClassifyIV(SCC)
  end if

ClassifyIV(SCC)
  for each n ∈ SCC do
    if header.RPOnum > n.block.RPOnum then
      header←n.block
    end if
  end for
  for each n in SCC do
    if n の演算が φ, +, −, COPY でない then
      SCC は基本帰納変数ではない
    else
      for each o ∈ n のオペランド do
        if o ∉ SCC ∧ ¬RegionConst(o,header) then
          SCC は基本帰納変数ではない
        end if
      end for
    end if
  end for
  if SCC が基本帰納変数である then
    for each n ∈ SCC do
      n.header←header
    end for
  else
    for each n ∈ SCC do
      if n が置き換え可能な演算である (注) then
        call Replace(n,IV,RC)
      else
        n.header←NULL
      end if
    end for
  end if
```

(注) 置き換え可能な演算とは, $x \leftarrow IV \times RC$, $x \leftarrow RC \times IV$, $x \leftarrow IV \pm RC$, $x \leftarrow RC + IV$, $x \leftarrow IV \ll RC$ のことである.

図 6.12: OSR : 基本帰納変数の検査

```
RegionConst(name,header)
  if name の値が定数 ∨ name.block ∈ sdom(header) then
    return true
  else
    return false
  end if

Replace(node,IV,RC)
  result←call Reduce(node.op,IV,RC)
  node を result からの COPY に置き換える
  node.header←IV.header
```

図 6.13: OSR : 演算の置き換え

```
SSAname Reduce(opcode,IV,RC)
  result←call search(opcode,IV,RC)
  if result が表に登録されていない then
    result に新しい SSA の名前をつける
    call add(opcode,IV,RC,result)
    newDef←call copyDef(IV,result)
    newDef.header←IV.header
    for each o ∈ newDef のオペランド do
      if o.header=IV.header then
        call Reduce(opcode,o,RC)
        o を Reduce の結果で置き換える
      else if opcode=x ∨ newDef.op=φ then
        call Apply(opcode,o,RC)
        o を Apply の結果で置き換える
      end if
    end for
  end if
  return result

SSAname Apply(opcode,op1,op2)
  result←call search(opcode,op1,op2)
  if result が表に登録されていない then
    if op1.header ≠ NULL ∧ RegionConst(op2,op1.header) then
      result←call Reduce(opcode,op1,op2)
    else if op2.header ≠ NULL ∧ RegionConst(op1,op2.header) then
      result←call Reduce(opcode,op2,op1)
    else
      result に新しい SSA の名前をつける
      call add(opcode,op1,op2,result)
      演算を適用した新しい演算の挿入位置を決定
      定数畳み込みができるのであればそれを行う
      新しい演算 newOper を決定された位置に挿入
      newOper.header←NULL
    end if
  end if
  return result
```

図 6.14: OSR : 演算の適用

6.5.3 判定の置き換え

演算の強さの軽減を行った後の変換されたプログラムには, ループの終了判定のみに使用されている基本帰納変数が含まれる場合がある. このような基本帰納変数は, 判定の置き換え (LFTR : Linear Function Test Replacement) を行うことにより除去できる可能性がある [11].

判定の置き換えを行うためには, まず基本帰納変数 iv1 とループ不変値 rc1 との比較命令を捜し出す必要がある. そして iv1 が, どのようにして演算の強さを軽減されたかという履歴を用いて, 同じ演算を rc1 に対して行う. 例えば帰納変数 iv1 が以下のように演算の強さを軽減されていたとする.

$$iv1 \stackrel{-1}{\Rightarrow} iv2 \stackrel{\times 4}{\Rightarrow} iv3 \stackrel{+a}{\Rightarrow} iv4$$

この情報をもとに, 新しい比較先のループ不変値 rc を作成する

$$rc \leftarrow ((rc1 - 1) \times 4) + a$$

これにより, 本来 iv1 と rc1 で構成されていた比較演算を, 各々 iv4 と rc に置き換えることができる. 置き換えられた帰納変数 iv1 は別途無用コード除去 (6.7 節) 等で除去することができる.

6.6 条件分岐を考慮した定数伝播

定数伝播を行うアルゴリズムには, 条件文分岐を考慮しない単純なアルゴリズムと, 条件分岐を考慮したアルゴリズムが存在する. ここでは条件分岐を考慮して, 実行される可能性のあるブロックについてだけ解析を行うアルゴリズムを考える [18, 19].

まず CFG の中で, 実行中に到達する可能性のある事が分かった辺とブロックにしるしを付けていく. 辺 e がブロック B に向かう辺であるとき, B=target(e) と書く. 2 つの作業用集合 Flow Work と SSA Work を使用する. アルゴリズムを以下に示す. なお, 「未定」とはその変数定義が実行されないときの変数の評価値であり, 「不定」とはフローによりその変数定義が一意に定まらないときの変数の評価値である. 「定数 c」とはその変数定義が定数値 c に一意に定まる時の変数の評価値である.

1. 初期設定:
   Flow Work = { プログラムの入口ノードから出ている辺 }
   SSA Work = 空集合
   全ての辺の到達可能性は偽 (false)
   全てのブロックの到達可能性は偽
   全ての式の値は「未定」

2. Flow Work と SSA Work がともに空なら 6 へ行く.

3. Flow Work が空でなかったならば, そこから辺 e をひとつとり, それを Flow Work から除く. e の到達可能性が真 (true) であったら何もしない. そうでなければ:

   - e の到達可能性を真とする.
   - B=target(e) にある ϕ 関数に対して「ϕ 関数の評価」(後述) をする.
   - B の到達可能性が偽なら, それを真とし, B の中の全ての式に対して「式の評価」(後述) をする.
   - B の後続ブロックがひとつだけ (B の最後は条件分岐でない) ならば B から出る辺を Flow Work に加える.

4. SSA Work が空でなければ, そこから定義 v をひとつとり, それを SSA Work から除く. v を使用している全ての式 exp について, exp が到達可能なブロックに入っていたならば, 以下のことを行う.

   - exp が ϕ 関数ならば,「ϕ 関数の評価」をする.
   - exp が ϕ 関数でなければ, その「式の評価」をする.

5. 2 へ行く.

6. ϕ 関数の引数に到達不能ブロックからのものがあったら，その引数を削除する．そしてそのような ϕ 関数の書換えが起こっていたら，1 に戻る（注．理由は不明）.

「ϕ 関数の評価」:

- 引数のうち到達可能な辺に対応するものの値を求め, それらに表 6.3 の演算を施して ϕ 関数の評価値を求める.
- その値が以前の値と違ったならば, この ϕ 関数の定義を SSA Work に加える.

「式の評価」:

- オペランドの値に「不定」なものがあれば式の評価値を「不定」とする.
- 全てのオペランドが定数ならば, 式の値 c を計算して, その式の評価値を c とする.
- それ以外の場合は式の評価値を「未定」とする.
- ただし, 演算子が論理和 (or) で一方のオペランドが真 (true) であるときは式の評価値を真, 演算子が論理積 (and) で一方のオペランドが偽 (false) であるときは式の評価値を偽とする.
- 以上の評価結果が以前の値と違う場合は, それが「w← 式」の形であるときはそれを SSA Work に加える. それが (条件文の) 条件式であるときはそれによって選ばれる辺を Flow Work に加える. すなわち, 条件式の評価値が定数 (真か偽) ならば, 真または偽に対応する辺, 不定なら全ての辺を加える.

表 6.3: ϕ 関数について定数の判定をする演算

	未定	定数 c'	不定
未定	未定	定数 c'	不定
定数 c	定数 c	定数 c(c=c' の場合) 不定 (c≠c' の場合)	不定
不定	不定	不定	不定

左端の列と上端の行が ϕ 関数の第 1, 第 2 引数である

表 6.3 は, ϕ 関数の引数同士の組合せによって, 式の評価値がどのように決定されるかを表している. 例えば, ϕ 関数 ϕ(a_1, a_2) において, a_1 の評価値が「未定」, a_2 の評価値が「定数 c」ならば, ϕ 関数の評価値は「定数 c」となる.

全てのブロックの処理を終えたあとに, 最終的に「未定」であったコードは無用コードとして除去することができる. 本システムではこれを行う. また, 最適化の結果, CFG の入口から到達しない辺が現れた場合には, その辺の始点の条件分岐文を適切に書き換えて, CFG からその辺を除去する.

定数畳み込みにおける 0 による割り算の処理

4.2.1 節では，SSA 形式の変数の初期値 $\perp$ として，実装上は 0 または 0.0 を代入していることを述べた.

しかしこのようにすると，実行時には起こらない 0 による割り算が定数畳み込み時に起こることがある．たとえば,

```
int i;
if (...)  i = ...;
...  = 8 / i;   /* (*) */
```

のようなソースプログラムがあるとする．実行時には i の値はいつも設定されており，(*) で 0 による割り算による例外は発生しないとする.

しかし，最適化時に分かることは，(*) に到達する i の値は，if 文の中で代入された値，もしくは未初期化のままの i の値となることだけである．現在の実装では変数の初期値を 0 としているので，定数伝播の最適化を素朴に適用すると，未初期化の場合の i の値は 0 となる．すると，定数伝播の処理の過程で，一時的に (*) の直前の ϕ 関数の値が 0 となり，その値を使って (*) の割り算を行ってしまい，0 による割り算が起こる.

この問題の解決法として，現在は，定数畳み込み時に 0 による割り算が起こった場合に，その値を「不定」として扱っている．これにより，4.2.1 節のアルゴリズムが適切に動作する．（根本的な解決のためには，未初期化の変数の値を 0 ではなく「不定」として扱うのが望ましい．）

実装での注意–浮動小数点数の定数畳み込み

現状の SSA 最適化での定数畳み込みでは, 浮動小数点数も畳み込んでいる．その計算の仕方は, LIR で提供されている定数畳み込み API を利用したものである．また, LIR では浮動小数点数の定数を Java の double 型の形で保持している．そのため, C 言語の float 型の定数を畳み込むと, double 型で評価されてしまい, 精度の点で, float 型で評価した結果と異なってしまう可能性がある．(同様にして整数型定数を Java の long 型で保持しているが, それも潜在的に問題かもしれない.)

6.7 無用コード除去

実行されないコード, すなわちそこへ制御の流れが届かないコードは無用コードとなる. また a←a といった式も無用コードである. さらに a←b+c という式も, その結果得られる a の値が使用されない場合は無用コードである.

無用コードは以下の手順で発見される.

1. 明らかに「生きている」コードを $LIVE$ とし, それ以外のコードを「死んでいる」($DEAD$) とする.
2. 生きているコード X で使用している変数を定義しているコード Y を $LIVE$ とする.
3. $LIVE$ となっている全てのコードについて 2 を行えば終了. そうでなければ 2 を繰り返す.

本システムでは以下の LIR を明らかに「生きている」コードとした.

- PROLOGUE
- EPILOGUE
- CALL
- レジスタ以外への SET

SSA 形式であるから, 各変数の使用に対してそれを定義している文はたかだかひとつである. よって, 手順の 2 を行うために別途変数の生存区間解析を必要としない.

また無用コード除去では無用な条件分岐文も除去する. 条件分岐文が無用となる条件は以下の通りである.

1. 条件分岐に制御依存する「生きている」コードがない[2]

しかし上記の条件 1 だけでは, 本来無限ループであるプログラムが無用コード除去により無限ループのないプログラムになってしまうなど, プログラムが持つ意味が変わってしまう場合がある [3]. それを避けるため, 現実装ではループから出るための条件分岐は明らかに生きているコードとした.

無用な条件分岐が除去された場合, 複数あった successor へのエッジをひとつにしなければならない. 本実装では生きているコードを含むブロックに到達するエッジを任意にひとつ選択し, それ以外のエッジは除去する. エッジを除去した結果エッジの指し先のブロックに制御が到達しない場合には, そのブロックは CFG から除去される.

無用コード除去アルゴリズムを図 6.15 に示す [19, 3].

SSA 形式上での無用コード除去を行うアルゴリズムは, [19] や [3] に記述してあるものが一般的であると思われる. これらのアルゴリズムは ϕ 関数を, 他のコードと同じに扱っている. また, これらのアルゴリズムでは制御フローを変更するようなコード列の変更も特定の条件のもとで許しており, その条件は最適化の対象となるプログラムが SSA 形式か否かで変化しない. しかし SSA 形式特有のコードである ϕ 関数は他のコードと異なり, コード自身に制御フロー情報とデータフロー情報を含有している. それゆえ, SSA 形式上での無用コード除去では ϕ 関数をすこし特別に扱う必要がある. 改良点は以下の 2 点である.

[2]ただし LIR の JUMP は例外とする.

```
Work ← 空集合
for each 文 S do
  if 文 S が明らかに生きている then
    Live(S) ← true
    Work に S を加える
  else
    Live(S) ← false
  end if
end for
while(Work が空でない) do
  Work から文 S を取り除き, S について以下を行う
  for each D(S で使用されている変数の定義文 D) do
    if Live(D) = false then
      Live(D) ← true
      Work に D を加える
    end if
  end for
  for each ブロック B(S のブロックは B に制御依存) do
    if Live(B の最後の条件文) = false then
      Live(B の最後の条件文) ← true
      Work に B の最後の条件文を加える
    end if
  end for
end while
for each 文 S do
  if Live(S) = false then
    文 S を除去する
  end if
end for
```

図 6.15: 無用コード除去アルゴリズム
注) このアルゴリズムでは, ϕ 関数を他のコードと同じと扱っている.

1. **ϕ 関数のソースリソースと関連付けられているブロックが制御依存する条件分岐文を除去しない**
   例えば図 6.16 (a) では, 前述の条件 1 によると, ブロック P が空のため, P が制御依存する B の最後の条件文が無用コード除去によって除去されてしまう. S の ϕ 関数が無用なもの (6.14 節参照) であればよいが, 一般的にこのことはプログラムの意味を変えてしまう危険がある.

2. **同じ predecessor から異なる SSA 変数名をもつ変数が ϕ 関数のソースリソースに入力されている場合に, その predecessor にある条件文は除去しない**
   例えば図 6.16 (b) では, 前述の条件 1 によると, ブロック S がブロック P に制御依存しないため, P の最後の条件文が無用コード除去によって除去されてしまう. これは明らかにプログラムの意味を変えてしまう. プログラムの意味を変えないために, これを上記の方法 1 で対応しても, $P \in pred(S)$ は, P 自身には制御依存しないため, 結果として P の最後の条件分岐は除去されてしまう.

改良版無用コード除去アルゴリズムを図 6.17 に示す. アルゴリズム中の太字の部分が改良点である.

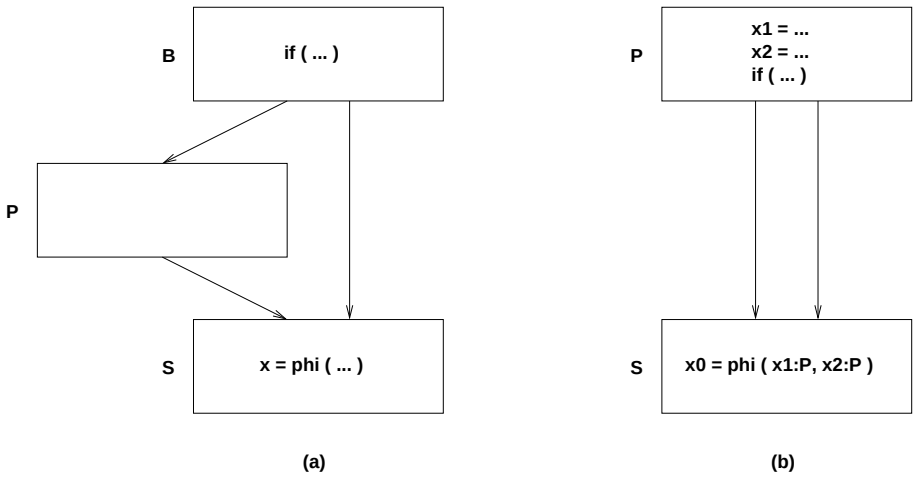


図 6.16: 無用コード除去のあやうい例

```
Work ← 空集合
for each 文 S do
  if 文 S が明らかに生きている then
    Live(S) ← true
    Work に S を加える
  else
    Live(S) ← false
  end if
end for
while(Work が空でない) do
  Work から文 S を取り除き, S について以下を行う
  for each D(S で使用されている変数の定義文 D) do
    if Live(D) = false then
      Live(D) ← true
      Work に D を加える
    end if
  end for
  if S が φ 関数である then
    /* φ 関数のソースリソースに関連づけられたブロックが空の場合
       でもそのブロックが制御依存している条件文を除去しないための処理 */
    for each ブロック P ∈ pred(S のブロック) do
      for each ブロック B(P は B に制御依存) do
        if Live(B の最後の条件文) = false then
          Live(B の最後の条件文) ← true
          Work に B の最後の条件文を加える
        end if
      end for
    end for
    /* 同じ predecessor から異なる SSA 変数名をもつ変数が
       φ 関数のソースリソースに入力されている場合に,
       その predecessor にある条件文を除去しないための処理 */
    if(φ 関数のソースリソース名が異なっており,
        それが同じブロック P に関連づけられている ∧
        Live(P の最後の条件文) = false) then
      Live(P の最後の条件文) ← true
      Work に P の最後の条件文を加える
    end if
  end if
  for each ブロック B(S のブロックは B に制御依存) do
    if Live(B の最後の条件文) = false then
      Live(B の最後の条件文) ← true
      Work に B の最後の条件文を加える
    end if
  end for
end while
for each 文 S do
  if Live(S) = false then
    文 S を除去する
  end if
end for
```

図 6.17: 改良版無用コード除去アルゴリズム
注) 太字の部分が改良点である.

6.8 SSA グラフの作成と変数の等価性を見つけるアルゴリズム

本節では, SSA 形式上での変数の値の流れに注目したデータ構造である SSA グラフ (値グラフとも呼ばれる) について記述する. SSA グラフのノードとして値または演算があり, 各ノードにはその値または演算によって定義される変数名のラベルがつけられている. 演算を表すノードからは, その演算で使用する変数を定義するノードへのエッジが張られる [2, 11]. 例として, 図 6.18 のプログラムから得られる SSA グラフを図 6.19 に示す.

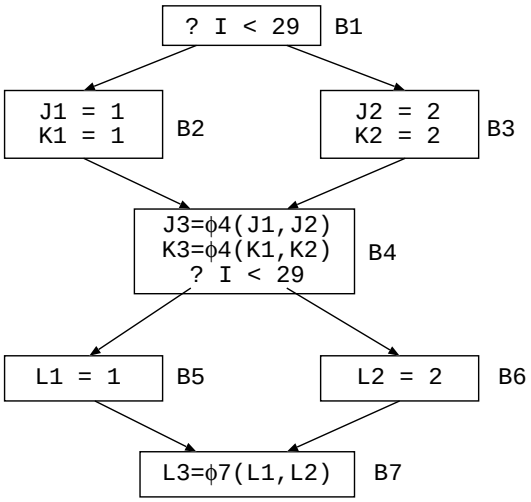


図 6.18: SSA 形式のプログラム例

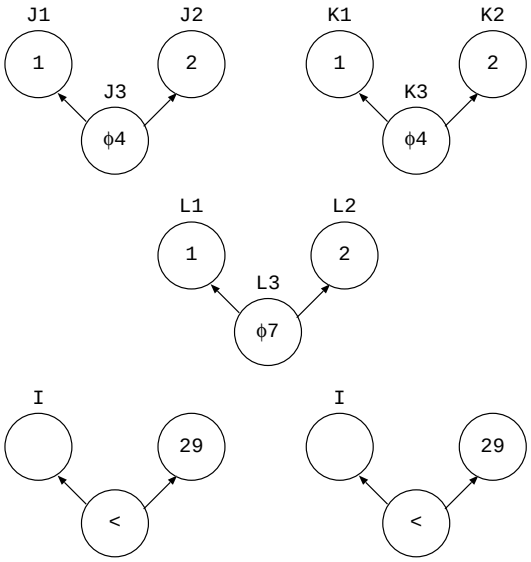


図 6.19: 図 6.18 のプログラムから得られる SSA グラフ

以下 LIR から SSA グラフを作成するアルゴリズムを述べ, 次に SSA グラフを用いた値の等価性を調べるアルゴリズムについて述べる.

6.8.1 LIR から SSA グラフ作成のためのアルゴリズム

LIR から SSA グラフを作成するためには, 各 LIR を読み込み, 値または演算の LIR を抽出し, それを SSA グラフのノードに変換すればよい. 演算が代入文の場合は, その右辺の演算ノードのラベルとして左辺の変数名が使用される.

LIR から SSA グラフを作成する際には MEM に注意する必要がある. LIR の MEM はロードとストアの双方の意味を持つ. SET の第 1 引数, または CALL の第 3 引数として MEM を持つ場合, その MEM はストアを意味する. この MEM を SSA グラフに変換する場合, MEM ノードには代入する値のノードと, 書き込む先のアドレスノードに向けたエッジを張る. 上述以外での MEM はロードを意味する. その場合, MEM ノードから読み込む値のノードへエッジを張る.

SSA グラフではひとつの演算ノードでひとつの演算を表す. 例えば

$$x1 \leftarrow y1 * 4 + z1$$

という式では, '$*$' を表すノードと '$+$' を表すノードのふたつが必要となる. SSA グラフを用いた最適化では, 演算ノードに変数名ラベルがついていない場合は明示的に一時変数名のラベルをつけたほうが便利である. そこで本システムでは上記の式を以下のように分解する.

$$\begin{aligned} t0 &\leftarrow y1 * 4 \\ x1 &\leftarrow t0 + z1 \end{aligned}$$

t0 は一時変数である.

例として図 6.20 のプログラムをあげる. 図 6.20 のプログラムは 6.21 の LIR 列に変換される. 図

```
void func(int a[]){
  int i,ans[100];

  for(i=0;i<100;i++){
    ans[i]=a[i]+i;
  }
}
```

図 6.20: SSA グラフ作成 : 例題プログラム (C 言語形式)

6.21 を SSA グラフに変換すると図 6.22 のようになる. 図中の楕円の中で"def" とあるのは上述の"ラベル" のことである.

```
#1 Basic Block (.L1):
  (PROLOGUE (0 0) (REG I32 "a.1%_1"))
  (JUMP (LABEL I32 ".L2" $1))

#2 Basic Block (.L2):
  (JUMP (LABEL I32 ".L3" $2))

#3 Basic Block (.L3):
  (PHI I32 (REG I32 "i.2%_2")
           ((INTCONST I32 0) (LABEL I32 ".L2" $7) (LABEL I32 ".L3" $2))
           ((REG I32 "i.2%_3") (LABEL I32 ".L5" $8) (LABEL I32 ".L3" $6)))
  (JUMPC (TSTLTS I32 (REG I32 "i.2%_2") (INTCONST I32 100))
         (LABEL I32 ".L4" $3)
         (LABEL I32 ".L6" $4))

#4 Basic Block (.L4):
  (SET I32 (MEM I32 (ADD I32 (FRAME I32 "ans.3")
                             (MUL I32 (REG I32 "i.2%_2")
                                      (INTCONST I32 4))))
           (ADD I32 (MEM I32 (ADD I32 (REG I32 "a.1%_1")
                                      (MUL I32 (INTCONST I32 4)
                                               (REG I32 "i.2%_2"))))
                    (REG I32 "i.2%_2")))
  (JUMP (LABEL I32 ".L5" $5))

#5 Basic Block (.L5):
  (SET I32 (REG I32 "i.2%_3")
           (ADD I32 (REG I32 "i.2%_2")
                    (INTCONST I32 1)))
  (JUMP (LABEL I32 ".L3" $6))

#6 Basic Block (.L6):
  (EPILOGUE (0 0))
```

図 6.21: SSA グラフ作成 : 例題プログラム (LIR 形式)

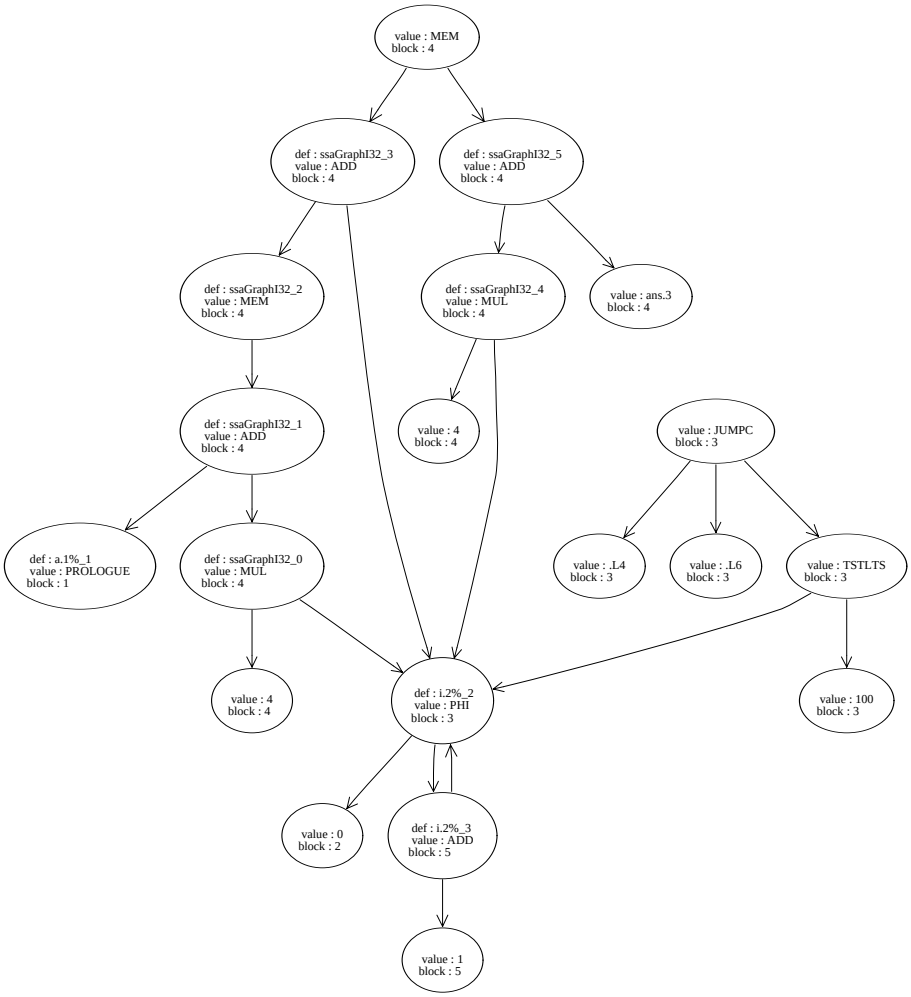


図 6.22: 図 6.21 の LIR から変換した SSA グラフ

最後に図 6.22 で示した SSA グラフから変換された LIR を図 6.23 に示す. 上述の通り, ひとつの式に複数の演算がある場合は, それらが分解されていることがわかる.

```
#1 Basic Block (.L1):
  (PROLOGUE (0 0) (REG I32 "a.1%_1"))
  (JUMP (LABEL I32 ".L2" $1))

#2 Basic Block (.L2):
  (JUMP (LABEL I32 ".L3" $2))

#3 Basic Block (.L3):
  (PHI I32 (REG I32 "i.2%_2")
           ((INTCONST I32 0) (LABEL I32 ".L2" $7) (LABEL I32 ".L3" $2))
           ((REG I32 "i.2%_3") (LABEL I32 ".L5" $8) (LABEL I32 ".L3" $6)))
  (JUMPC (TSTLTS I32 (REG I32 "i.2%_2") (INTCONST I32 100))
         (LABEL I32 ".L4" $3)
         (LABEL I32 ".L6" $4))

#4 Basic Block (.L4):
  (SET I32 (REG I32 "ssaGraphI32_0")
           (MUL I32 (INTCONST I32 4)
                    (REG I32 "i.2%_2")))
  (SET I32 (REG I32 "ssaGraphI32_1")
           (ADD I32 (REG I32 "a.1%_1")
                    (REG I32 "ssaGraphI32_0")))
  (SET I32 (REG I32 "ssaGraphI32_2")
           (MEM I32 (REG I32 "ssaGraphI32_1")))
  (SET I32 (REG I32 "ssaGraphI32_3")
           (ADD I32 (REG I32 "ssaGraphI32_2")
                    (REG I32 "i.2%_2")))
  (SET I32 (REG I32 "ssaGraphI32_4")
           (MUL I32 (REG I32 "i.2%_2")
                    (INTCONST I32 4)))
  (SET I32 (REG I32 "ssaGraphI32_5")
           (ADD I32 (FRAME I32 "ans.3")
                    (REG I32 "ssaGraphI32_4")))
  (SET I32 (MEM I32 (REG I32 "ssaGraphI32_5"))
           (REG I32 "ssaGraphI32_3"))
  (JUMP (LABEL I32 ".L5" $5))

#5 Basic Block (.L5):
  (SET I32 (REG I32 "i.2%_3")
           (ADD I32 (REG I32 "i.2%_2")
                    (INTCONST I32 1)))
  (JUMP (LABEL I32 ".L3" $6))

#6 Basic Block (.L6):
  (EPILOGUE (0 0))
```

図 6.23: 図 6.22 の SSA グラフから変換された LIR

6.8.2 変数の等価性を見つけるアルゴリズム

図 6.2 に示した SSA 形式上の共通部分式除去のアルゴリズムは, 表 6.1 や表 6.2 で示したような, 式と変数名の対応表およびコピーの関係にある変数名の対応表を用いて同じ値になる式を見つけるものである. このアルゴリズムは簡単であるが, ループがある場合などの解析は十分ではない. SSA 形式上での広範囲にわたる式の同値性を見つける手法は多く提案されているが, 本システムでは Alpern らによる SSA グラフ (値グラフ) の上で変数の等価性を見つけるアルゴリズム [2, 19] を実装した.

変数の等価性を調べる方法は以下のようにする. 最初に, 同じ演算子を持つ式は同じ値を持つ可能性があるので同じ集合とする. そして, 同じ集合に含まれる式のうち, オペランドまたは引数の値が異なるものを別の集合に分割する. 分割を繰り返し, それ以上分割できなくなれば, 同じ集合に入っている式は同じ値を持つ. また, 同じ値の定数はひとつの集合に入れておき, 代入文については, その左辺と右辺を同一視して, 左辺を集合の要素とする. なお, ϕ 関数はブロックごとに違う関数とする. その理由は, たとえば図 6.18 の SSA 形式プログラムで B4 と B7 の ϕ 関数を同じ関数とすれば, L3 は J3 および K3 と同じ値であると判定されてしまうからである.

例として, 図 6.24 のプログラムを考える. このプログラムを SSA 形式に変換したものを図 6.25 に示す. 簡単のために条件式の部分を省略した.

```
a=1;
b=1;
do{
  if(...){
    a=a+1;
    b=b+1;
  }
  else{
    a=a+2;
    b=b+2;
  }
}while(...);
```

図 6.24: 変数の等価性：例題プログラム

最初に作られる集合は次のようになる.

$B[1] = \{1, a0, b0\}$

$B[2] = \{2\}$

$B[3] = \{a2, b2\}$ 関数$\phi 2$

$B[4] = \{a3, b3, a4, b4\}$ 加算

$B[5] = \{a1, b1\}$ 関数$\phi 5$

求めるものは, 例えば a+b と c+d において a と c, b と d が同じ集合に入っているときに限り a+b と c+d が同じ集合に入るような分割で, 最も粗い分割である. 逆にいえば, a と c がすでに別の集合に入っていたら a+b と c+d も別の集合に入れなければならない. もしある集合 B[i] について, その要素を第 m オペランドとする式の集合 H を考えたとき, H が既存のある集合 B[j] と共通部分を持ち, かつ B[j] が H に含まれていないとすると, B[j] の中には B[i] 以外の要素を第 m オペランドとする式が存在することになる. したがって B[j] をさらに分割する必要がある. そのような分割のアルゴリズムを図 6.27 に示す. このアルゴリズムで, INVERSE を求めるときに SSA グラフを使用する. 図 6.25 の SSA グラフを図 6.26 に示す.

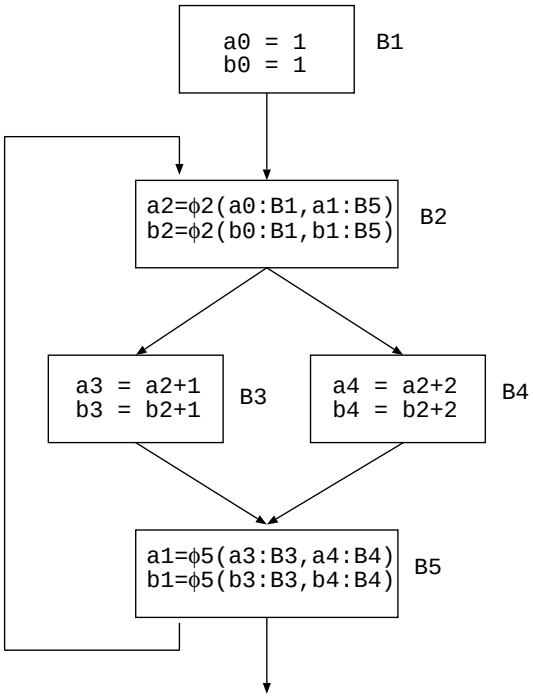


図 6.25: 変数の等価性：SSA 形式に変換した例題プログラム (CFG)

例題プログラムに図 6.27 のアルゴリズムを適用すると次のようになる. まず B[1] の場合, その要素 (値は 1) を第 1 オペランドとする式の集合は,

$$INVERSE = \{a2, b2\} = B[3]$$

で, これは分割の条件に合わない. しかしそれを第 2 オペランドとする式の集合は

$$INVERSE = \{a3, b3\}$$

で, これは B[4] と共通部分を持つが, B[4] は INVERSE に含まれていない. そこで,

$$B[6] = \{a3, b3\}$$

が新たに作られ,

$$B[4] = B[4] - B[6] = \{a4, b4\}, WAITING = \{2, 3, 4, 5, 6\}$$

となる. この後, WAITING の要素から分割の条件に合うものは出てこない. したがって最終的に得られるものは以下のものとなる.

$$\begin{aligned} B[1] &= \{1, a0, b0\} \\ B[2] &= \{2\} \\ B[3] &= \{a2, b2\} \\ B[4] &= \{a4, b4\} \\ B[5] &= \{a1, b1\} \\ B[6] &= \{a3, b3\} \end{aligned}$$

最終的に得られる集合 B[i] に入っているものは, (それらの代入文が実行された時点で) 同じ値を持つ. 同じ集合に入っている 2 つの変数は合同 (congruent) であると言われる.

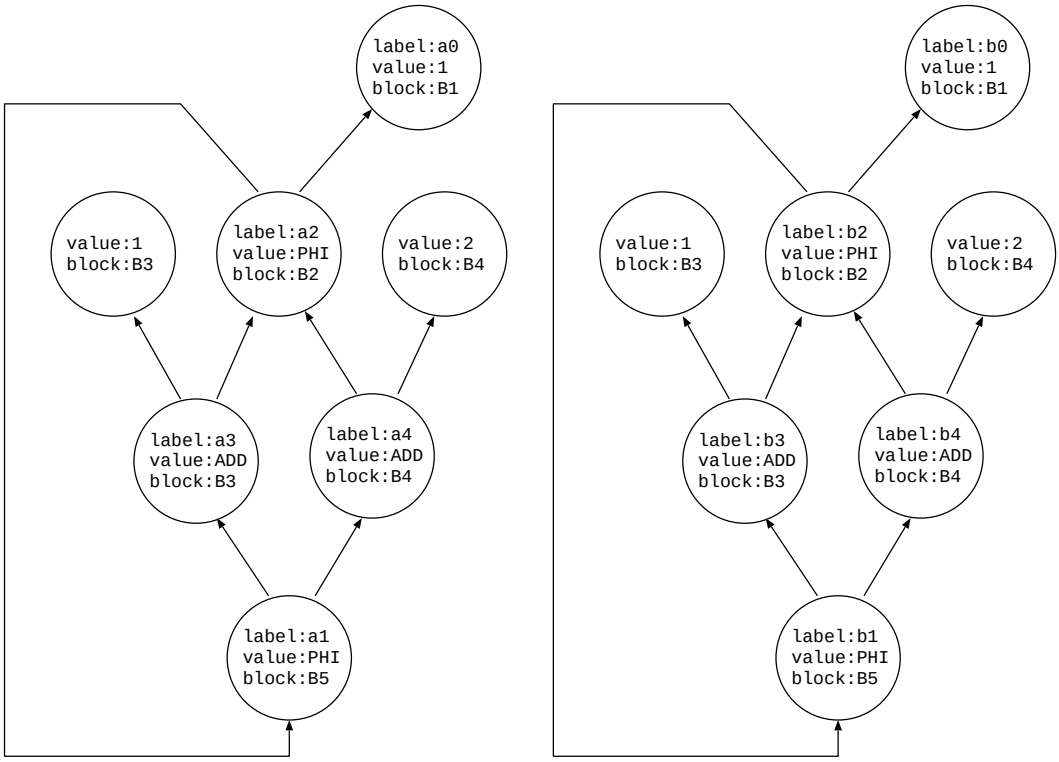


図 6.26: 変数の等価性：図 6.25 の SSA グラフ

```
WAITING ← {1,2, ···, p} /* p は最初の集合の個数 */
q ← p
while WAITING ≠ { } do
   WAITING からひとつの整数 i を選び, それを WAITING から除去する
   for each m (1≤m≤k) do /* k は式のオペランドの個数 */
      INVERSE ← { }
      for each x in B[i] do
         INVERSE ← INVERSE ∪ F^-1[m,x]
         /* F^-1[m,x] は B[i] の要素 x を第 m オペランドとする式の集合 */
      end for
      for each j such that B[j] ∩ INVERSE ≠ { } ∧ B[j] ⊈ INVERSE do
         q ← q+1
         新しい集合 B[q] を作る
         B[q] ← B[j] ∩ INVERSE
         B[j] ← B[j]-B[q]
         if j ∈ WAITING then
            q を WAITING に加える
         else if |B[j]|≤|B[q]| then
            j を WAITING に加える
         else
            q を WAITING に加える
         end if
      end for
   end for
end while
```

図 6.27: 変数の等価性：分割アルゴリズム

6.9　3 番地コードへの変換

コンパイラの入力となる高級言語では, ひとつの変数に対する代入文の右辺は必ずしもひとつの演算からなるとは限らない. つまり $x \leftarrow a + b * c - d$ のような式が許されている. 上記の式を木構造で表すと図 6.28 のようになる. しかし最適化の種類によっては, 式は 3 番地コード (three-address

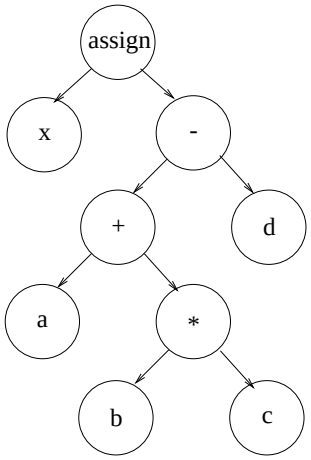


図 6.28: 木構造で表した式

code) のように, ひとつの代入に対してひとつの演算となっていることが都合がいい場合がある. そこで本システムではそのような形への変換を実装した.

この変換の方針は単純である. つまり演算のオペランドが木構造の葉でなければその部分で式を分割する. 図 6.28 の式は図 6.29 の破線の位置で分割する.

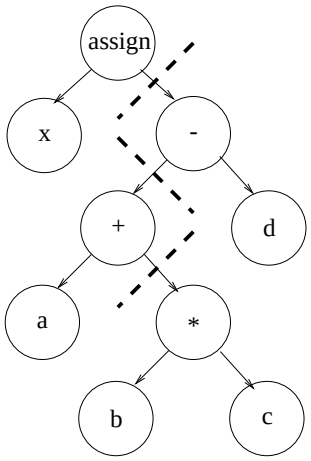


図 6.29: 式の分割位置

式を分割する際には, テンポラリ変数を用意し, その変数に対して演算途中の値を代入すればよい. 図 6.29 の分割では, テンポラリ変数として $tmp1, tmp2, tmp3$ を用意し, 図 6.30 のように再構成すればよい. 図中の破線矢印はテンポラリ変数の代入から使用へのエッジを示している.

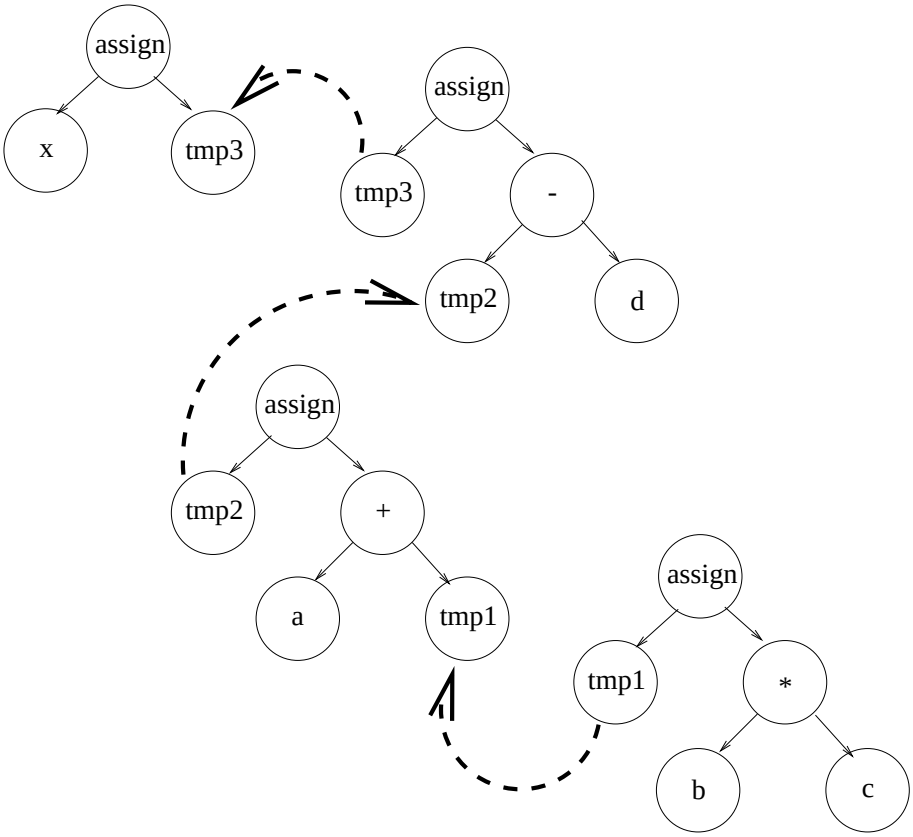


図 6.30: 式の分割

上記の方法により, 式 $x \leftarrow a + b * c - d$ は以下の式に分割される.

$$
\begin{aligned}
tmp1 &\leftarrow b \times c \\
tmp2 &\leftarrow a + tmp1 \\
tmp3 &\leftarrow tmp2 - d \\
x &\leftarrow tmp3
\end{aligned}
$$

なお, 現実装では $x \leftarrow tmp3$ のようなコピー文が多く発生するのを防ぐために $x \leftarrow tmp2 - d$ としている.

6.10 空の基本ブロックの除去

CFG 上でのコード最適化を行う場合, 最適化の結果, 基本ブロック内の全てのコードが最適化によって除去されてしまう場合がある. そのような基本ブロックは, predecessor から制御が到達しても, 単に制御を successor に渡すだけで, プログラムに対して何も作用をもたらさない. また, このような CFG 構造を保持したままアセンブラコードに変換すると, 無条件飛び越しが増えてしまい, 結果的にプログラムの実行コストを増大させてしまう可能性がある.

そこで本システムでは, 空の基本ブロックを除去する最適化を実装した. 空の基本ブロックを以下に定義する.

空の基本ブロック ひとつの基本ブロック内に, 飛び越し命令以外のコードがひとつも含まれないもの.

上記の定義を LIR に適用すると, ひとつの基本ブロック内に JUMP/JUMPC/JUMPN 以外の命令を含まないものとなる. 命令には ϕ 関数も含まれる. しかし, 全ての空の基本ブロックが除去できるわけではない. 本システムでは, まず以下の条件を満たす空の基本ブロックについて除去する.

条件 1 空の基本ブロック E において, ($|\mathrm{succ(E)}| = 1 \wedge |\mathrm{pred(E)}| = 1$) であり, かつ ($|\mathrm{succ(pred(E))}| = 1 \wedge |\mathrm{pred(succ(E))}| = 1$) を満たす場合

条件 1 を図 6.31 (a) に示す.

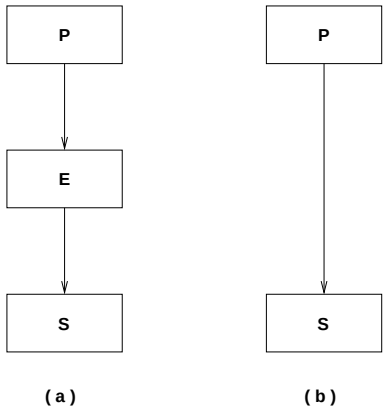


図 6.31: 空の基本ブロック除去 : 条件 1

条件 1 にあてはまる空の基本ブロック E を除去するためには, P $\in$ pred(E) から S $\in$ succ(E) へのエッジ $P \longrightarrow S$ を追加し, E を CFG から除去すればよい. E の除去後は図 6.31 の (b) のようになる.

6.11 基本ブロックの結合

COINS の実装では, 図 6.32 の左側のようなプログラムから制御フローグラフを生成すると, 図 6.32 の真ん中のような制御フローグラフができる.

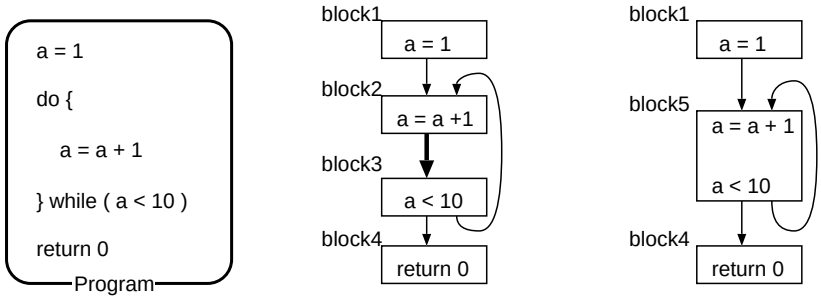


図 6.32: 除去する辺

図 6.32 の真ん中の制御フローグラフにおいて, `block2` と `block3` を結ぶ辺は除去することの可能な辺で, 無駄なジャンプ命令が存在することになる. `block2` のように後続 node をひとつしか持たず, その後続 node である `block3` の先行 node がひとつ (`block2`) だけであるとき, このような辺は除去できる.

そこで本システムでは 図 6.32 の右側の制御フローグラフのように, `block2` と `block3` を合成した新しい node (`block5`) を作り, 無駄なジャンプ命令を除去する最適化を実装した.

なお現実装では `block5` に相当する node は全く新しい node を作成するのではなく, `block2` または `block3` で代用する形になっている.

6.12 危険辺除去

SSA 形式から通常形式への変換の際や, ある種の最適化を行う際には, きわどい辺 (危険辺 : Critical Edge) [6, 19] を除去するかどうかの選択がある. 危険辺とは, $|\mathrm{succ}(B1)| > 1$ となるような基本ブロック B1 から $|\mathrm{pred}(B2)| > 1$ となるような基本ブロック B2 へのエッジ $B1 \longrightarrow B2$ のことである. 危険辺の例を図 6.33 に示す. 図の例では, $P1 \longrightarrow S2$ が危険辺となっている.

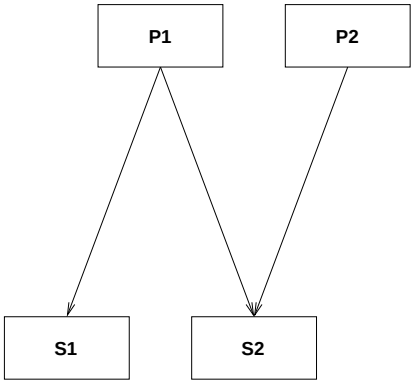


図 6.33: 危険辺

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)[19]. 例えば図 6.33 の例では, 危険辺 $P1 \longrightarrow S2$ に対して新しい基本ブロック N1 を作成し, $P1 \longrightarrow N1 \longrightarrow S2$ 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 6.33 の危険辺を除去したものを図 6.34 に示す.

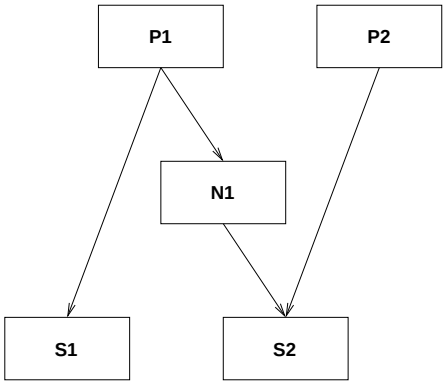


図 6.34: 危険辺除去

なお, この危険辺除去は SSA 形式上で行われる. SSA 形式上で基本ブロックの追加等, 制御フローを変更すると ϕ 関数の扱いが難しい. しかし本実装では ϕ 関数の書き換え等が正しく行われ, SSA 形式が保たれる.

6.13 Global Reassociation

Global Reassociation [5] とは, 最適化の対象となる式の形の問題をうまく扱うためのテクニックである. 例えば以下の式は, 数学的には $x1 = y1$ となる.

$$x1 \leftarrow a1 + b1 + c1 \tag{6.1}$$

$$y1 \leftarrow b1 + c1 + a1 \tag{6.2}$$

しかし LIR を含む多くのコンパイラ中間表現では, 式を木構造で表すことが多いため, (6.1) の右辺と (6.2) の右辺を同じ値とみなすことが困難である (図 6.35). そこで Global Reassociation では算

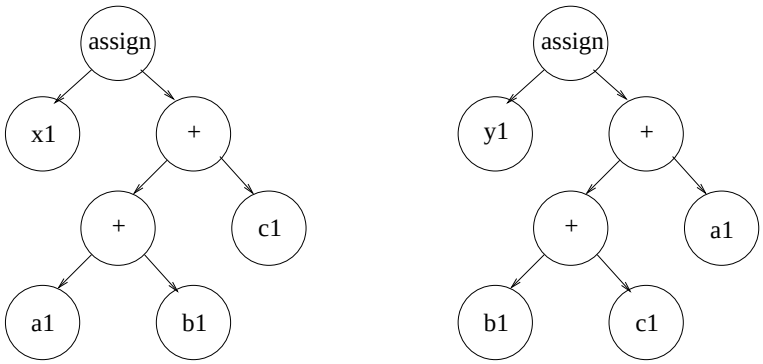


図 6.35: 木構造で表した式

術演算の代数特性 (交換則, 結合則, 分配則) を利用して式を再構成する. これにより共通部分式やループ不変式をより多く発見することが期待される.

本システムに実装した Global Reassociation は次の 2 つの ステップ からなる.

1. 全ての式の rank を計算する
2. 式の再構成およびソーティングをオペランドのランクに基づいて行う

Briggs らの論文 [5] では, 3 つの ステップ があった. 彼らは 2 番目の ステップ として, 「式を使用側に伝播させる」というのを提唱している. しかしこれを行ってしまうと SSA 形式を壊してしまうので, 本実装ではこれを除外した.

6.13.1 rank の計算

式の再構成をするための指針として, 個々の式と部分式に対して rank をつける. 直観的には, rank とは, 定数やループ不変式に対しては小さく, ループ内で値が変わる式に対しては大きいものである.

CFG を reverse postorder で辿る. 辿られる node の rank は辿られた順序の値となる. つまり最初に辿られた node (つまり入口 node) の rank は 1, 2 番目に辿られた node の rank は 2 というようにする. node 内の式に対しては, 以下のルールで rank 付けをする.

1. 定数の rank は 0 とする.
2. ϕ 関数の rank はその ϕ 関数が属する node の rank とする. 関数呼び出しの戻り値が入る変数やロード命令によって値が入れられる変数も, それらの命令が属する node の rank とする.
3. 演算の rank は, その演算に使われるオペランドの最も高い rank とする.

6.13.2 式のソート

rank の低い順に部分式を並べかえる. これを式のソートという. 式のソートには交換則を用いる. より多くの式に対して式のソートを行うためには, 交換則を適用できない部分式を, それと等価な交換則を適用できる部分式に置換することが望ましい. そこで, 式のソートを行う前準備として減算を加算に置き換える. つまり $x-y+z$ のような部分式を $x+(-y)+z$ と書き換える. 除算も乗算で表すことができるが, 精度の問題が発生するのを防ぐために行わない.

rank に基づくソート

rank に基づくソートでは, オペランドの rank が低いものから順に左から並ぶようにソートする. $x+y+z$ という部分式があったとして, 各々の rank を $3,1,1$ とすると, このソートにより $y+z+x$ もしくは $z+y+x$ という部分式に再構成される. 再構成された部分式に対して 6.9 節で述べた 3 番地コードへの変換を行うと

$$\begin{aligned} temp &\leftarrow y+z \text{ (もしくは } z+y) \\ &\leftarrow temp+x \end{aligned}$$

となり, $temp \leftarrow y+z$ (もしくは $z+y$) をループ不変式としてループ外に追い出せる可能性がある. また定数は rank 0 であるから, 定数を複数含む部分式, 例えば $1+y+2$ では定数同士の演算が左側にソートされ, $1+2+y$ のような部分式に再構成される. 再構成された部分式に定数伝播を行えば $3+y$ を容易に導き出すことができる[3].

本実装で rank に基づくソートを実行する LIR の演算は以下の通りである. (SUB は対象としない.)

- ADD
- MUL
- BAND
- BOR
- BXOR

辞書順ソート

rank に基づくソートでは, rank が低いものから順に左から並ぶようにソートされる. しかし rank が同じもの同士はソートされない. 例えば $a1+b2$ という部分式と $b2+a1$ という部分式がある場合, a1 と b2 の rank が異なれば式の形は一意に決定されるが, rank が同じであれば式の形は両方とも存在する可能性がある. 式の形をみて等価性を判断するような共通部分式除去等の効果を考えると, rank が同じであっても式の形をある一定の規則にしたがってソートすることが好ましい.

そこで本実装では, rank が同じオペランド間は辞書順にソートするようにした. ソートされる LIR の演算は以下の通りである.

- ADD
- MUL
- BAND

[3]COINS の LIR に実装されている式の評価メソッドは, 再構成前の式からでも $3+y$ を導き出すことができる.

- BOR
- BXOR

6.13.3 分配則の適用

式のソートが終了後, 加算に対する乗算の分配則適用を行う. つまり $w \times (x + y + z)$ を $w \times x + w \times y + w \times z$ に書き換える. このような分配則の適用は必ずしも全ての場合に対して有効ではない. そこで今一度 rank を変換の指針に用いる. 本実装では, 低い rank を持つ乗算を, 高い rank を持つ加算に対して分配するようにした.

例えば $a+b\times((c+d)+e)$ という式があり, a, b, c, d の rank が 1 で e の rank が 2 の場合は, 分配則を適用し $a+b\times(c+d)+b\times e$ とする. これにより $b\times e$ がループ不変でない場合でも $a+b\times(c+d)$ をループ不変として追い出すことが可能である. なお, 分配則を適用すると式の形が変わってしまうので, 式の再ソートが必要となる.

分配則を適用すると, 式 $a\times(b+c)$ は $(a\times b)+(a\times c)$ に変換される. この変換は数学的には正しいが, あふれを考えると誤った結果を出力する場合がある. 例えば $b = INT\_MAX$, $c = -(INT\_MAX-1)$ とすると, $a\times(b+c)$ と $(a\times b)+(a\times c)$ の値は異なる. 本システムで実装された Global Reassociation にもこの危険性があるので注意が必要である.

6.14 無用 ϕ 命令除去

コピー伝播や Copy Folding(ここでいうコピー伝播とは 6.1 節で記述されている, 最適化パスとしてのものであり, Copy Folding とは 4.2.2 節で記述されている, SSA 形式への変換途中で行うものとする) などの最適化を行うと, 無用な ϕ 命令があらわれる場合がある. この最適化では, ある最適化の結果としてあらわれた無用な ϕ 命令の除去を行う. 無用な ϕ 命令というものは以下のように定義される [7, 9].

1. ϕ 命令の全てのリソースが同じ場合
2. ϕ 命令のソースリソースが全て同じ場合 [4]
3. ϕ 命令のソースリソースにデスティネーションリソースを含み, かつソースリソースのなかのデスティネーションリソース以外のリソースが 1 種類のみの場合 [5]

厳密にいえば, 条件 2, 3 の場合には, ソースリソースが $\perp$ であるかどうかの判断が必要になる. しかし現状では SSA 変数の値が $\perp$ であるかどうかのチェックはしていない. なぜなら SSA 変換途中で作成される $\perp$ となる変数, つまり変数の名前替えで利用されるスタックにつまれる初期値は, CFG のエントリで必ず 0 初期化をしており, 値が不定ではない.

1 から 3 の無用な ϕ 命令を除去する方法は以下の通りである [9, 7].

- 1 の場合はその ϕ 命令を除去するのみである.
- 2 の場合, ϕ 命令を除去し, それ以降使用される ϕ 命令のデスティネーションリソースをソースリソースに置き換える. 例えば y← ϕ(x,x,x) のような場合には, この ϕ 命令を y←x とみなしてコピー伝播を行うことと同じ意味を持つ.
- 3 の場合, ϕ 命令を除去し, それ以降使用される ϕ 命令のデスティネーションリソースを, ソースリソースでデスティネーションリソースでないものに置き換える. 例えば y← ϕ(x,x,y) のような場合には, この ϕ 命令を y←x とみなしてコピー伝播を行うことと同じ意味を持つ.

[4] y← ϕ(x,x,x) のような場合
[5] y← ϕ(x,x,y) のような場合

6.15 Lazy Code Motion

6.15.1 背景

部分冗長除去 (Partial Redundancy Elimination, 以後 PRE と略称することもある) は，部分的に冗長な式を除去する変換であり，共通部分式除去とループ不変式移動の効果を含んだ効果的な最適化である．PRE は，Morel らによって最初のアルゴリズムが提案され，その後も多くの改良されたアルゴリズムが提案されている

これらの研究の中で，PRE を**静的単一代入形式 (Static Single Assignment Form, 以後 SSA 形式と呼ぶ)** 上で行おうという試みがある．SSA 形式は最適化に適したプログラムの表現形式であり，近年盛んに研究が行われている．しかし，PRE を SSA 形式上で行おうとすると，

- 通常形式上では同一の変数であったものが名前替えにより異なる変数名になることがある．
- SSA 形式での変数には満たすべき条件があるので，異なるブロックにコードを移動する際に変数をそのまま移動できない．

といった困難がある．

これらの問題を解決したものに，PRE アルゴリズムを SSA 形式上で実現する汎用的手法がある [20]．本最適化では，この手法にもとづき Lazy Code Motion(以後 LCM と称する) の実装をおこなった．

6.15.2 手法の概要

本手法の概要を説明する (詳細については，[20] を参照のこと)．

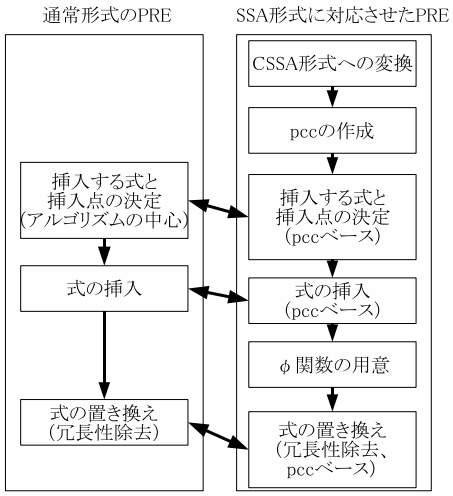


図 6.36: 通常形式上の PRE と SSA 形式上の PRE

一般的な PRE アルゴリズムの手順を図示すると図 6.36（左）のようになる．このうち，アルゴリズムの中心となるのは「挿入する式と挿入点の決定」の部分である．

一方，通常形式での PRE アルゴリズムを本手法によって SSA 形式に対応させると，その手順は図 6.36（右）のようになる．このうち，元の PRE アルゴリズムに対応している部分は「挿入する式と挿入点の決定」と「式の挿入」,「式の置き換え」の三つであるが，後者二つは「$t = a + b$」の挿入や，「$\ldots = a + b$」の「$\ldots = t$」への置き換えであって，どの PRE アルゴリズムでも同じである．よっ

て，元の PRE アルゴリズムに依存するのは実質「挿入する式と挿入点の決定」のみとなる．この処理は，変数の字面の情報の代わりに pcc（後述）の情報を用いることで，アルゴリズムの本質を変えることなく SSA 形式に対応させることが出来る．よって，図 6.36（左）の手順に沿うような PRE アルゴリズムであれば本手法を用いて SSA 形式に対応させることが出来る．

6.15.3 TSSA 形式と CSSA 形式

CSSA 形式とは，すぐ後で述べる phi congruence class (以後，pcc と略称する) に属する変数をすべて同じ代表変数に置き換えてφ関数を除去すると，プログラムの意味の等しい通常形式が得られるような SSA 形式，と定義できる [16]．これは，元々Cytron らのアルゴリズム [12] で通常形式を SSA 形式に変換した直後の SSA 形式が有している性質である．この条件は，同じ pcc 内の変数同士に干渉 (生存区間が重複すること) がない事，と言い換える事もできる．

CSSA 形式における pcc は直感的には，同一の代表変数に置き換えても構わないような変数の集合を意味する．

しかし，SSA 形式での最適化変換を行うと，前述のような CSSA 形式の性質は一般には保たれない．つまり，pcc の変数間に干渉が発生する．このような SSA 形式を TSSA 形式という．本手法では PRE を行う前に，文献 [16] の手法を用いて TSSA 形式から CSSA 形式への変換を行う．

同一の式かどうかを判断する場合に通常形式上においては同じ変数かどうかの判断を行っていたが，CSSA 形式上では代わりに同じ phi congruence class に属する変数かどうかを判断すればよい．

以下に，フローグラフ fg に対する pcc を作成するアルゴリズムの概略を示す．

アルゴリズム 6.15.1 (pcc の作成)

```
makePhiCongruenceClass(FlowGraph fg){
  for(各φ関数 phi ∈ fg){
    phi で結ばれた変数を同じ pcc にする;
  }
  共通部分を持つ pcc をマージする;
}
```

また，pcc を使用して，与えられた二つの式 $expr1$ と $expr2$ の同一性を識別する方法を次に示す．

アルゴリズム 6.15.2 (pcc による式の識別)

```
same(expr1, expr2){
  if(式 expr1, expr2 の 演算子,
    型がそれぞれ等しい){
    if(expr1, expr2 の オペランドが属する
      pcc がそれぞれ等しい){
      return true;
    }
  }
  return false;
}
```

6.15.4 挿入する式と挿入点の決定

挿入する式と挿入点の決定は，実装する PRE アルゴリズムが行う．ただしその際 CSSA 形式では，前節で述べたように，同じ変数かどうかを判断する代わりに同じ phi congruence class に属するかどうかを判断するよう，PRE アルゴリズムを変更する．

LCM における挿入式と挿入点の決定のアルゴリズムは，複数個の流れの等式を解くものである．本最適化では，[19] のアルゴリズムを実装している．

対象となる式を t としておく．t の引数への代入文を t の変更ということにする．基本ブロックを入口部分と出口部分に分けて考える．入口部分というのは，基本ブロックの入口から t の最後の変更までの部分であり，出口部分というのは，基本ブロックの残りの部分である．入口部分に含まれる最初の t の計算を入口計算といい，出口部分に含まれる t の計算を出口計算という．

挿入点の決定は，以下のように定義される．

部分冗長性を除去するには，その計算をもとにあったところより前に移動し，もとの計算をその値で置き換えればよい．そのための情報は，各ブロックで得られるつぎの計算値である．

$$
\begin{aligned}
NComp(B) &= \text{″}B \text{ に入口計算がある″} \\
XComp(B) &= \text{″}B \text{ に出口計算がある″} \\
Transp(B) &= \text{″}B \text{ に変更文がない″}
\end{aligned}
$$

t の計算を置いても安全な場所を「下安全 (down-safe)」と「上安全 (up-safe)」として，つぎのように定義する:

$$
\begin{aligned}
NdSafe(B) &= \text{″}B \text{ の入口挿入点で下安全である″} \\
XdSafe(B) &= \text{″}B \text{ の出口挿入点で下安全である″} \\
NuSafe(B) &= \text{″}B \text{ の入口挿入点で上安全である″} \\
XuSafe(B) &= \text{″}B \text{ の出口挿入点で上安全である″}
\end{aligned}
$$

NdSafe，XdSafe に関しては，つぎの等式が成り立つ:

$$NdSafe(B) = NComp(B) \cup \{Transp(B) \cap XdSafe(B)\}$$

$$XdSafe(B) = XComp(B) \cup \begin{cases} false & if B = endnode \\ \cap_{S \in succ(B)} NdSafe(S) & otherwise \end{cases}$$

NuSafe，XuSafe に関しては，つぎの等式が成り立つ:

$$NuSafe(B) = \begin{cases} false & if B = startnode \\ \cap_{P \in pred(B)} \{XComp(P) \cup XuSafe(P)\} & otherwise \end{cases}$$

$$XuSafe(B) = Transp(B) \cap \{NComp(B) \cup NuSafe(B)\}$$

下安全でもっとも開始点に近い場所は，つぎのように定義される:

$$
\begin{aligned}
NEarliest(B) &= \text{″}B \text{ の入口挿入点に計算を置いても下安全であるが，それを } B \text{ のどれかの} \\
&\quad \text{先行ブロックに移すことはできない (安全でなくなる)″} \\
XEarliest(B) &= \text{″}B \text{ の出口挿入点に計算を置いても下安全であるが，それを } B \text{ の入口} \\
&\quad \text{挿入点に移すことはできない (安全でなくなる)″}
\end{aligned}
$$

$$NEarliest(B) = NdSafe(B) \cap \cap_{P \in pred(B)} \overline{XuSafe(P) \cup XdSafe(P)}$$

$$XEarliest(B) = XdSafe(B) \cap \overline{Transp(B)}$$

計算を Earliest の場所から遅らせることのできる点をもとめる:

$NDelayed(B)$ = ″プログラムの入口から B の入口挿入点に達するどの路にも, $NEarliest(P)$ または $XEarliest(P)$ である挿入点があり，その計算を B の入口挿入点まで遅らせてもよい″

$XDelayed(B)$ = ″プログラムの入口から B の出口挿入点に達するどの路にも, $NEarliest(P)$ または $XEarliest(P)$ である挿入点があり，その計算を B の出口挿入点まで遅らせてもよい″

Delayed をもちいて，計算をもっとも遅らせることのできる場所をもとめる:

$$NLatest(B) = NDelayed(B) \cap NComp(B)$$

$$XLatest(B) = XDelayed(B) \cap \{XComp(B) \cup \cup_{S \in succ(B)} \overline{NDelayed(S)}\}$$

$NLive(B)$ = ″B の入口部分の出口で w(計算した値が代入される変数) がいきている″

$XLive(B)$ = ″B の出口部分の出口で w がいきている″

$$NLive(B) = \{XComp(B) \cup XLive(B)\} \cap \overline{XLatest(B)}$$

$$XLive(B) = \begin{cases} false & if B = endnode \\ \cup_{S \in succ(B)} [\{NComp(S) \cup NLive(S)\} \cap \overline{NLatest(S)}] & otherwise \end{cases}$$

計算の挿入点は，つぎのように定義される:

$NInsert(B)$ = ″基本ブロック B の入口挿入点に計算 $w = t$ を挿入すべきである″

$XInsert(B)$ = ″基本ブロック B の出口挿入点に計算 $w = t$ を挿入すべきである″

$NReplace(B)$ = ″基本ブロック B の入口計算を w で置き換えるべきである″

$XReplace(B)$ = ″基本ブロック B の出口計算を w で置き換えるべきである″

$NInsert(B)$，$XInsert(B)$ は，つぎの式で求められる:

$$NInsert(B) = NLatest(B) \cap NLive(B)$$

$$XInsert(B) = XLatest(B) \cap XLive(B)$$

$$NReplace(B) = NComp(B) \cap \{NInsert(B) \cup \overline{NLatest(B)}\}$$

$$XReplace(B) = XComp(B) \cap \{XInsert(B) \cup \overline{XLatest(B)}\}$$

6.15.5 式の挿入

部分冗長な式を発見すると，PRE アルゴリズムは前節で求められたプログラムの挿入点に文「一時変数 = 式」を挿入し，部分冗長な式を全冗長にする．また後の処理の為に，計算結果を一時変数に格納しておく必要があるため，もう片方の先行ブロックの元の計算の直前にも，やはり同様の文を挿入する．

以下に，式 $expr$ をプログラムポイント p に挿入するアルゴリズムを示す．$expr$ は $a\ op\ b$ とする．op は演算子，a, b はオペランドである．

アルゴリズム 6.15.3 (式の挿入)

$insertExpr(expr, p)$\{
  p を支配する点を上向きに辿り,
  a と同じ pcc に属する変数 a'の定義を探す;
  p を支配する点を上向きに辿り,
  b と同じ pcc に属する変数 b' の定義を探す;
  まだ定義されてない変数 t_iを作成する;
  代入文 $t_i = a'\ op\ b'$を p に挿入する;
\}

6.15.6 φ関数の用意

一時変数の挿入の後は，その挿入した一時変数のためのφ関数を用意する．

φ関数の作成

まず，最小 SSA を作成するアルゴリズム [12] に従って，前節で $t_i = a_j + b_k$ の形の式を挿入したブロックの支配辺境に，一時変数のためのφ関数を作成する．

なお，φ関数の左辺には，まだ使われていない一時変数の名前を付ければよい．また，右辺の変数にはこの時点では仮の名前を付けておく．なぜなら，この手続きで挿入される，他のφ関数の左辺の一時変数を指すべき場合も存在しうるためである．その場合はφ関数の作成順序によっては,「本来指すべき変数」がこの時点ではまだ存在しないことになり，一通りφ関数を挿入した後でないと右辺を正しく決める事ができなくなる．

以下に，φ関数を作成するアルゴリズムを示す．

アルゴリズム 6.15.4 (φ関数の作成)

$insertPhi()$\{
  for(式の挿入を行なった各ブロック blk)\{
    if(blk の支配辺境 df にまだφ関数が
      挿入されていない)\{
      まだ定義されていない変数 t_iを作成する;
      ダミーの変数として任意の変数 t_j, t_kを
      作成する;
      φ関数 $t_i = \phi\ (t_j, t_k)$ を df に挿入する;
    \}
  \}
\}

φ関数内の変数の名前替え

一時変数のためのφ関数を一通り挿入した後で，φ関数の右辺の一時変数を適切な名前に書き換える処理を行う．

以下に名前替えのアルゴリズムを示す．但し，ここではφ関数を正確に表すため「$t_i = \phi(t_j : L_a, t_k : L_b)$」と記している．これは，ブロック L_a からきたときは t_j を使用し，ブロック L_b からきたときは t_k を使用する，ということを示している．

アルゴリズム 6.15.5 (φ関数の引数の名前替え)

```
rename(){
  for(挿入した各φ関数 t_i =
    φ (t_j : L_a, t_k : L_b)){
    ブロック L_aからそのブロックを支配する
    ブロックを上向きに辿り,
    挿入した式か挿入したφ関数を見つけたら,
    その左辺で t_jを置き換える;
    ブロック L_bからそのブロックを支配する
    ブロックを上向きに辿り,
    挿入した式か挿入したφ関数を見つけたら,
    その左辺で t_kを置き換える;
    名前替えを行なったφ関数の引数の変数を
    同じ pcc にする;
  }
}
```

6.15.7 冗長となった式の除去

冗長な式の除去は，実際には式の，今導入した一時変数への置き換えである．

以下にプログラムポイント p の右辺の式を一時変数で置き換えるアルゴリズムを示す．点 p はすでに求まっており，一時変数は t と同じ pcc に属しているとする．

アルゴリズム 6.15.6 (冗長となった式の除去)

```
replace(p, t){
  p を支配する点を上向きに辿り,
  t と同じ pcc に属する変数 t'の定義を探す;
  p に存在する文の右辺の式を t'で置き換える;
}
```

以上の処理によって SSA 形式上での PRE は達成される．

6.16 要求駆動型部分無用コード除去 (PDEQP)

無用コード除去の効果を高めるために，コード移動と組み合わせた手法として，部分無用コード除去（partial dead code elimination，以降 PDE と呼ぶ）がある．本最適化では，個々の代入文に対して，必要に応じて PDE を適用するアルゴリズムを導入し，より効果的な最適化を，効率良く行う手法を実装した (詳細に関しては，[21] を参照のこと).

本最適化は，従来プログラム全体を網羅型に解析していた PDE に対して，1 つの代入文に必要な範囲だけを解析する要求駆動型の解析を導入する手法であり，プログラム点 n の代入文 s が無用かどうかを表すクエリを伝播させることによって，プログラムの限られた範囲から，n における s の無用性を検査する．本最適化で実装した手法は，従来のクエリを用いた要求駆動型の手法と異なり，クエリの伝播と逆向きに解を伝播させる．プログラム点 n' に伝播してくる解が，“無用である” という解と “無用でない” という解を含んでいた場合，“無用でない” という解を伝播してきた n' の後続のプログラム点に文 s を挿入することによって，PDE を実現する．

本最適化はコード移動に基づいているので，危険辺 (critical edge）があると，効果が制限されてしまうため，事前に危険辺除去 (split critical edge) が適用されているものとする．

6.16.1 アルゴリズム

本最適化では最初に，プログラムの意味を変えないように，左辺の変数が右辺にも出現する $x = x+1$ のような代入文を，右辺の式に固有の一時変数 t を用いて，$t = x + 1;\ x = t$ のように分離する．

本最適化の以降の手順は次のとおりである．

1. ある CFG 節 n の 1 つの代入文 s に対して，クエリを伝播させ，無用コードかどうか検査する．
2. n に $true$ の解が伝播してきたなら，s を除去する．
3. (2) の過程で，上向き安全に関わった他の代入文についても，同様にクエリの伝播を行った後，除去する．

クエリの伝播と解の伝播は，局所解を検査する関数の再帰呼出しと，その呼出し系列の過程で返される返戻値によって実現できる．プログラム 1 と 2 は，局所解を検査する関数 $local$ と，クエリを前向きに伝播させる関数 $propagate$ を示す.

$local$ の 2 行目から 5 行目は，局所解の条件を表現している．ここで，lhs と lhr は，それぞれ，文の左辺と右辺を取り出す関数であり，$isUsed$ と $isMod$ は，それぞれ，引数の変数が使用されているかを検査する関数と，引数に現れる変数が変更されているかを検査する関数である．いずれの条件でも解が得られない場合，7 行目で，CFG 節 v の訪問済みを意味する $visit[v] = true$ を実行し，9 行目の $propagate$ の呼出しで，さらに後続節へクエリを伝播する．このとき，対象の $stmt$ の右辺が節 v で，変更されている場合，文を移動するとプログラムの意味が変わってしまうので，以降の文の挿入が許されないことを $canInsert := false$ として，$propagate$ に指示する（8 行目）．この値は，引数 $canInsert$ を通して，以降で呼び出されるすべての $local$ および $propagate$ に伝えられる．最終的に，節 v の局所解は，配列 ans の要素 $ans[v]$ として求まる（10 行目）.

プログラム 1 (局所解)

```
1: Function local(stmt, v, canInsert)
2:   if (v =e) ans[v] := true
3:   else if (isUsed(lhs(stmt))) ans[v] := false
4:   else if (isMod(lhs(stmt))) ans[v] := true
```

```
5:   else if (visit[v] = true) return ans[v]
6:   else
7:     visit[v] = true
8:     if (isMod(rhs(stmt))) canInsert := false
9:     ans[v] := propagate(stmt, v, canInsert)
10: return ans[v]
```

プログラム 2 の関数 *propagate* は，CFG 節の上向き安全性の検査とクエリの伝播を行う．上向き安全性の検査は，関数 *isOccur* を呼び出すことによって行う．*isOccur* は，ワークリストを用いた要求駆動型データフロー解析によって文 *stmt* の出現を検査する．得られた結果は，*canInsert* と論理積を取ることによって，$isSafe$ に記録し，節 6 で，さらに関数 *local* の仮引数 *canInsert* の値として渡す．すなわち，一旦 $isSafe = false$ になると，以降の *propagate* の呼出しは，すべて $isSafe = false$ になり，上向き安全性を検査せずに，安全でない文の挿入を避けることができる．

クエリの伝播は，後続節 s_i に対して *local* を呼び出すことによって，5，6 行目で行う．その際，7 行目で *local* の解が $false$ なら，文を挿入する候補の節として，s_i を *insSet* に記録する（7 行目）．

8 行目で，s_i の解が 1 つでも *true* であり，10 行目で挿入候補の節も存在しないなら，v の解は *true* である．挿入候補の節が存在する場合，11 行目で $isSafe = true$ なら，12-13 行目 で，文を挿入候補の節に挿入し，v の解を *true* にする（14 行目）．このいずれでもない場合，v の解は $false$ になる（15-16 行目）．

プログラム 2 (クエリの伝播)

```
1: Function propagate(stmt, v, canInsert)
2:   let occurSet = ∅ and insSet = ∅
3:   let isSafe := canInsert ∧ isOccur(e, v, occurSet)
4:   let ans := false
5:   foreach s_i ∈ succ(v) do
6:     let succAns := local(stmt, s_i, isSafe)
7:     if succAns = false then insSet := insSet ∪ succ
8:     ans := ans ∨ succAns
9:   if ans = true then
10:    if insSet = ∅ then return true
11:    else if isSafe = true then
12:      insert stmt to the entry of all node s ∈ insSet
13:      workSet := workSet ∪ occurSet
14:      return true
15:    else return false
16: else return false
```

CFG 節 v の代入文 *stmt* へ実際に DDPDE を適用するためには，プログラム 3 の関数 *ddpde* を用いる．*ddpde* は，4，5 行目の *propagate* の呼出しと，解が *true* だった場合の文の除去だけでなく，上向き安全性検査の際に到達した他の文の出現に対しても，10，11 行目で，クエリの伝播と文の除去を行う．

プログラム 3 (DDPDE)

```
1: let workSet := ∅
2: Function ddpde(stmt, v)
3:   let ans[] := {true, ..., true}
```

```
4:  if (propagate(stmt, v, true))
5:     remove the statement of v
6:     while (workSet ≠ ∅)
7:        let v' ∈ workSet
8:        workSet := workSet \ v'
9:        ans[] := {true, ..., true}
10:       if (propagate(stmt,v',true))
11:          remove the statement of v'
```

6.17 要求駆動型部分無用コード除去 (DDPDE)

部分無用コード除去（partial dead code elimination，以降 PDE と呼ぶ）は，コード降下と無用コード除去という 2 つのプログラム変形の組合せによって実現される．

．本手法では，対象の代入文 s が存在するプログラム点 n から到達可能なプログラム点を前向きに辿りながら，各プログラム点 v での降下可能性を検査する．そして，v が降下可能な場合は，さらに降下させ，降下可能でない場合は，v に s を挿入する．降下可能性についても，要求駆動型の解析法を用いて検査できるので，全体の解析が必要最小限の範囲に限定できる．以降，本手法を要求駆動型 PDE （demand-driven PDE，DDPDE）と呼ぶ．

6.17.1 上向き安全の検査

```
Function isUpSafe(target, v)
1: worklist := {v}; query[] := false
2: while worklist \neg=0  do
3: let n \in worklist; remove n from worklist
4: query[n] := true
5: for each p \in  pred(n) do
6: if locdelayed[target, p] then
7: // if p \neg\in donedone then
8: // add p to done
9: // add p to cand
10: continue
11: else if p = entry \vee locblocked[target, p] then
12: return false
13: else if \not query[p] then
14: add p to worklist
15: return true
```

図 6.37: 上向き安全の検査のプログラム

6.17.2 無用性の検査

6.17.3 DDPDE の挿入点の計算

6.17.4 プログラムの変形

```
Function isDead(target, v)
1: worklist := {v}; query[] := false
2: while worklist \neg = 0  do
3: let n \neg = worklist; remove n from worklist
4: query[n] := true
5: for each s \in succ(n) do
6: if used[target, s] then
7: return false
8: if mod[target, s] then
9: continue
10: else if \not query[s] then
11: add s to worklist
12: return true
```

図 6.38: 無用性の検査のプログラム

```
: Function insert(target, v)
1: worklist := {v}; ndelayed :=
2: while worklist \neg= 0 do
3: let n \in worklist; remove n from worklist
4: for each s \in succ(n) do
5: if s = exit \vee \not isUpSafe(target, s) then
6: if \not isDead(target, n) then
7: add n to xinsert
8: break
9: if s \neg \in ndelayed then
10: add s to ndelayed
11: if locblocked[target, s] then
12: if \not(\not used[target, s] \vee (mod[target, s] \vee isDead(target, s))) then
13: add s to ninsert
14: else add s to worklist
```

図 6.39: DDPDE の挿入点の計算のプログラム

```
1: done :=
2: cand :=
3: ninsert :=
4: xinsert :=
Function ddPDE(target, v)
5: cand:={v}
6: done:={v}
7: while cand \neg=0  do
8: let n \in cand; remove n from cand
9: insert(target,n)
10: for each org \in done do
11: eliminate target at org
12: for each vin \in ninsert do
13: insert target into the entry of vin
14: for each vout \in xinsert do
15: insert target into the exit of vout
```

図 6.40: プログラムの変形のプログラム

6.18 大域ロード命令集約

6.18.1 概要

現在のプロセッサは，プロセッサの演算速度に比べて低速なメモリと，メモリより高速なキャッシュメモリを備えていることが多い．このような構成のプロセッサ上で，プログラムを効率的に動作させるためには，キャッシュメモリの有効な利用が重要である．本最適化では，任意のプログラムに対して，同じ配列や構造体を参照するロード命令を互いに近いプログラム点に移動させることによって，キャッシュメモリのヒット率を向上させる手法を実装した．本手法は，単に，ロード命令を同じプログラム点へ集約させるのではなく，集約後のメモリの参照順序を保存したまま，元のプログラム点に近い場所にロード命令を巻き戻す．このロード命令の巻戻しは，ロード先の一時変数の生存期間を短縮し，レジスタ割付けの際にスピルされる可能性を低減させる．

6.18.2 コード移動に基づく大域ロード命令集約

コード移動に基づく大域ロード命令集約（Global Load Instruction Aggregation ，以降 GLIA と呼ぶ）は，LCM で定義したデータフロー方程式をメモリアクセスの順序の考慮と，冗長でないロード命令のコード移動をを行えるように拡張し，ロード命令に限定して適用する．

以降の説明では，同じ配列や構造体へのストア命令を含む節 n を述語 Store(n) を使って区別し，特に断らない限り式 e をロード命令とする．式 e のロード元である配列や構造体の先頭アドレスは Addr(e) で表す．

本実装では，つぎのような仮定を設けている：

- 対象とするプログラムは，1 つの式に含まれるロード命令が，高々1 つになるように変形されているものとする．
- クリティカル辺（Critical edges）は，取り除かれているものとする．

GLIA は，LCM で定義したデータフロー方程式をメモリアクセスの順序の考慮と，冗長でないロード命令のコード移動をを行えるように拡張し，ロード命令に限定して適用する．

局所的な性質

局所的な性質は，節単位の式に対するデータフロー情報を示す．式の性質として，あいまいな参照を表す述語 Amb，同じ配列あるいは構造体に属することを表す述語 $sameAddr$，式が一致することを表す述語 $isSame$ を導入する．また，透過性を，式の透過性を表す述語 $Transp_e$ とアドレスの透過性を表す述語 $Transp_Addr$ によって区別する．

定義 6.18.1 (SameAddr)

$$SameAddr(n) \Leftrightarrow Addr(rhs(n)) = Addr(e)$$

定義 6.18.2 ($Transp_e$)

$$Transp_e(n) \Leftrightarrow Def(n) \notin \wedge \neg Store(n) \wedge \neg Amb(n)$$

定義 6.18.3 ($Transp_Addr$)

$$Transp_Addr(n) \Leftrightarrow Transp_e(n) \wedge Addr(rhs(n)) = Addr(e)$$

定義 6.18.4 (isSame)

$$isSame(n) \Leftrightarrow rhs(n) = e$$

大域的な性質

UpSafe，DownSafe，Earliest，Latest，Insert に関するデータフロー方程式を図 6.41 に示す.

$$UpSafe(n) \Leftrightarrow \forall p \in P[s,n], \exists i \leq \lambda\_p \wedge \forall i \leq j < \lambda_p.Transp_e(p_j)$$

$$DownSafe(n) \Leftrightarrow \forall p \in P[n,e], \exists i \leq \lambda\_p.isSame(p\_i) \wedge \forall 1 \leq j < i.Transp_e(p_j)$$

$$Safe(n) \Leftrightarrow UpSafe(n) \vee DownSafe(n)$$

$$Earliest(n) \Leftrightarrow Safe(n) \wedge \begin{cases} true & if n = s \\ \sum_{m \in pred(n)} \neg Transp_e(m) \vee \neg Safe(m) & otherwise \end{cases}$$

$$Latest(n) \Leftrightarrow Delayed(n) \vee (isSame(n) \vee \sum_{m \in succ(n)} \neg Delayed(m))$$

$$Isolated(n) \Leftrightarrow \prod_{m \in succ(n)} (Latest(m) \vee \neg isSame(m) \wedge Isolated(m))$$

$$Insert(n) \Leftrightarrow Latest(n) \wedge \neg Isolated(n))$$

図 6.41: 大域的な性質

6.19 効率的な要求駆動型部分冗長除去（EQP）

6.19.1 概要

コンパイラのコード最適化の 1 つである部分冗長除去は，部分冗長な式の除去とループ不変式のループ外への移動を同時に行う．一般的な部分冗長除去は，データフロー方程式を使用してすべての式の冗長性を同時に除去する．このとき，除去した式に依存する後続の式の冗長性が新たに出現する副次的効果を生じることがあるので，複数回の適用を行う必要があった．近年，1 度の適用で副次的効果を反映できる手法として，要求駆動型部分冗長除去が提案された．要求駆動型部分冗長除去では，式の出現ごとに，質問伝播という手法を用いて冗長性を除去し，要求駆動型コピー伝播を併用することによって副次的効果を効果的に反映する．しかしながら，質問伝播と要求駆動型コピー伝播は不要な解析を行う場合が存在し，データフロー方程式に基づいた手法よりも解析効率が悪くなるときがあった．本項で述べる手法は，質問伝播を行う前に大域値番号付けを適用しておき，各プログラム点に到達可能な値番号を記録しておくことによって不要な質問伝播を防ぎ，値番号に基づいた質問伝播を用いることによって要求駆動型コピー伝播を行わないようにしたものである．

本項の記述は、[24] によるものであるので、詳細に関しては、[24] を参照のこと．

6.19.2 大域値番号付け

本手法は，質問伝播を行う前に，GVN を適用し，すべての式に対して値番号を生成する．

本手法の GVN は，大域値番号付けアルゴリズム (図 6.42) に示すように，CFG をトポロジカルソート順序で訪問し，式の出現ごとに値番号を求める．本手法で生成する値番号は，整数で表す．新しく値番号を生成するときは，生成済みの値番号の中で最大のものに 1 を加える．

値番号付けの際，オペランドの値番号を記録するハッシュテーブル operandTable と，式の値番号を記録するハッシュテーブル operationT able を使用する．式の値番号を求めるとき，各オペランドの値番号と演算子の組みをキーとして operationT able を引く．もし operationT able に同一のキーが既に登録されているならば，operationT able から値番号を得る．さもなければ，新しい値番号を生成し，operationT able に新しくエントリを追加する．式の値番号を求めるとき，ハッシュテーブルに記録されている値番号を得たならば，その式は冗長である．もし節内で冗長ならば，以前の計算結果を参照するように変形する．また，不要なクエリの伝播を防ぐために，開始節から各節に到達するすべての経路上に存在する値番号を記録する．

6.19.3 質問伝播

本手法では，e の値番号 val に対して「val は利用可能か」というクエリを生成する．したがって，本手法における利用可能性と予期可能性の定義は，e と同じ式の出現があるかどうかを基にするのではなく，val と同じ値番号の出現があるかどうかで定義される．

PREQP におけるクエリの伝播は，次の規則を上から優先して適用する．

1. 節 n にクエリが伝播されたとき，n が開始節 s ならば，n におけるクエリの解は false である．
2. 節 n にクエリが伝播されたとき，n にクエリと同じクエリが伝播済みならば，n におけるクエリの解は true である．
3. 節 n にクエリが伝播されたとき，n にクエリの値番号が到達できず，値番号が 関数に依存しないならば，n におけるクエリの解は false である．

```
1:  Function globalV alueNumbering()
2:     foreach n ∈ toporogicalSort(N)
3:        if containPhiNotHaveV alArg(n)
4:           preparePreNumbering(n)
5:        numbering(n; false)

6:  Function numbering(n; lazy)
7:     foreach node ∈ n
8:        let val := value(n)
9:        if ¬lazy ∧ localRedundant(val; n)
10:          remove(node; val; n)
11:       setValue(val; n)

12: Function preparePreNumbering(n)
13:    worklist := n
14:    while worklist ≠⊥
15:       let v ∈ worklist// remove v from worklist
16:       numbering(v; true)
17:       visited :=
18:       preNumbering(v; n; visited)

19: Function preNumbering(n; entry; visited)
20:    foreach s ∈ succ(n)
21:       if ¬visited[s] ∧
             s ∉ dominaceFrontier(entry)
22:       visited[s] := true
23:       if containPhiNotHaveV alArg(s) ∧
             s ∉ worklist
24:          add s to worklist
25:       numbering(s; true)
26:       preNumbering(s; entry; visited)
```

図 6.42: 大域値番号付けアルゴリズム

4. 節 n にクエリが伝播されたとき，n がクエリの生成元の節ならば，自己生成解が true となり，クエリの解も true である．

5. 節 n にクエリが伝播されたとき，n にクエリの値番号が含まれているならば，n におけるクエリの解は true である．

6. 上記の規則のいずれにも該当しない場合，すべての先行節にクエリを伝播させる．このとき，節 n に，クエリのオペランドの値番号を含む 関数が含まれているならば，クエリを各引数に基づいて変更し，その引数に対応する先行節にそれぞれ伝播させる．

本手法の冗長性の検査のアルゴリズムを，図 6.43 に示す．

```
1:  Function propagate(val; ve; n)
2:    let n ' := n
3:      and isDownSafe := antqp(val; n)
4:    foreach p ∈ pred(n)
5:      val_p := val
6:      ve_p := transPhi(ve; n; p)
7:      if (ve_p = ⊥)
8:        return (false; false; false;⊥)
9:      if (ve = ve_p) val_p := getValue(ve_p; p)
10:     let (isAvail_p; isReal_p; isSelf_p; n_p) :=
          local(val_p; ve_p; p)
11:     if (isAvail_p)
12:       add np to link[n]
13:     else
14:       insertCand[p] := translate(ve_p)
15:       add p to link[n]
16:     let isReal := Σ_{p∈pred(n)} isReal_p
17:       and isSelf := Σ_{p∈pred(n)} isSelf_p
18:     if (Π_{p∈pred(n)} isAvail_p ∨
          isReal ∧ (isDownSafe ∨ isSelf))
19:       if (∃n_p ∈ link[n] : n_p domn ∧ ve_p = ve)
20:         link[n] := n_p
21:         n ' := n_p
22:       return (true; isReal; isSelf; n)
23:     else
24:       return (false; false; false;⊥)

25: Function local(val; ve; n)
26:   if (n = s)
27:     return (false; false; false;⊥)
28:   if (visited[n])
29:     if (answer[n] ≠⊥)
30:       return answer[n]
31:     else return (true; false; false; n)
32:   visited[n] := true
33:   let rlt :=⊥ and
34:     (isAvail; isSelf) := isSameV al(val; n)
35:   if (isAvail)
36:     var[n] := getV ar(val; n)
37:     rlt := (true; true; isSelf; n)
38:   else if (¬reacheValue(val; n)∧
          ¬dependPhi(e))
39:     rlt := (false; false; false;⊥)
40:   else
41:     rlt := propagate(val; ve; n)
42:   answer[n] := rlt
43:   return rlt
```

図 6.43: クエリ伝播のアルゴリズム

6.19.4 プログラム変形

プログラム変形のアルゴリズムを，図 6.44 に示す．本手法は，冗長性検査を行うときに，クエリの値番号を含む節において，その変数を記録する．プログラム変形において，変数の書き換えを行う際に記録されている変数を用いる．

```
1:  Function insert(n)
2:     if (var[n] ≠⊥)
3:        return var[n]
4:     else if (insertCand[n] ≠⊥)
5:        var[n] := createNewV ar()
6:        add [var[n]" = "insertCand[n]]
             to the exit of n
7:     else if ((answer[n]) = false)
8:        return ⊥
9:     else if (j link[n] j= 1)
10:       var[n] := insert(link[n])
11:    else
12:       var[n] := createNewV ar()
13:       let args :=
14:       foreach n'_i ∈ link[n]
15:          let var_i := insert(n'_i)
16:          if (var_i = ⊥) return ⊥
17:          add var_i to args
18:          add [var[n]" = ""φ""("args")"]
                to the entry of n
19:       return var[n]
```

図 6.44: プログラム変形のアルゴリズム

6.20 部分冗長除去の手法にもとづくスカラー置換 (PRESR)

6.20.1 概要

効率的な要求駆動型部分冗長除去の手法（[24]）にもとづいてスカラー置換をおこなう。効率的な要求駆動型部分冗長除去において、冗長な式の探索のために使われるクエリを、ループ帰納変数を添字にもつ配列について、そのループ帰納変数の更新箇所を越えて伝播できるように拡張し、異なる添字をもつ配列間で冗長を除去できるようにする。従来のスカラー置換では、本体が 1 つの基本ブロックからなるループを仮定していたが、効率的な要求駆動型部分冗長除去の手法をもちいることにより、この仮定を必要としない場合も扱うことができる。

6.20.2 質問伝播

本拡張では、帰納変数を添え字としてもつ配列参照に対して、その帰納変数の更新箇所でクエリをブロックしないように変更する。具体的には、更新箇所で、配列の添え字を、更新文の右辺で置き換える。例えば、次のようなプログラム（図 6.45）があったとする。

```
1:  i = 1;
2:  do {
3:      a[i+1] = a[i-1] + b[i-1];
4:      a[i] = a[i] + b[i] + b[i+1];
5:      i = i + 1;
6:  } while (p)
```

図 6.45: プログラム例

プログラム中の a[i-1] に対して、帰納変数について拡張した効率的な要求駆動型部分冗長除去を適用すると、戻り辺を通ってループの最後に伝播した a[i-1] のクエリは、5 行目のループ帰納変数の更新でブロックされずに単に i を i+1 で置換した a[(i+1)-1]（a[i] に畳込まれる）として引き続き伝播する。最終的に 4 行目の左辺 a[i] に到達すると、次のようなプログラム（図 6.46）に変形できる。

```
1:  i = 1;
2:  t0 = a[i-1];
3:  do {
4:      a[i+1] = t0 + b[i-1];
5:      t0 = a[i] + b[i] + b[i+1];
6:      a[i] = t0;
7:      i = i + 1;
8:  } while (p)
```

図 6.46: a[i-1] への適用

次に、5 行目の a[i] に適用すると、次のプログラム（図 6.47）が得られる。

```
1:  i = 1;
2:  t0 = a[i-1];
3:  t2 = a[i];
4:  do {
5:      t1 = t0 + b[i-1];
6:      a[i+1] = t1;
7:      t0 = t2 + b[i] + b[i+1];
8:      a[i] = t0;
9:      i = i + 1;
10:     t2 = t1;
11: } while (p)
```

図 6.47: a[i] への適用

b[i-1]、b[i] についても同様に適用すると、最終的に次のプログラム（図 6.48）が得られる。

```
1:  i = 1;
2:  t0 = a[i-1];
3:  t2 = a[i];
4:  r0 = b[i-1];
5:  r1 = b[i];
6:  do {
7:      t1 = t0 + r0;
8:      a[i+1] = t1;
9:      r0 = r1;
10:     r1 = b[i+1];
11:     t0 = t2 + r0 + r1;
12:     a[i] = t0;
13:     i = i + 1;
14:     t2 = t1;
15: } while (p)
```

図 6.48: 最終的なプログラムの形

最終的に得られたコードを見ると、ループ中に配列要素のロードが１つしかないのが分かる。元のコードで、ループ中に配列要素のロードが５つ存在したのと比べ、配列要素へのアクセスが大幅に減り、実行効率が上がる。これはロード命令に関して、スカラー置換に等しい結果であり、従来のスカラー置換と比べて、ループ本体が基本ブロック１つから構成されるなどの制限を必要としない点で優れている。

6.20.3 変更内容

本稿で述べるスカラー置換は、効率的な要求駆動型部分冗長除去の手法にもとづくものであり、そのためにつぎのような変更をおこなった：

- プログラムの表現形式の変更
- クエリが解を得るタイミングの変更
- クエリの式の書き換え規則
- クエリの訪問条件

これらの変更について、下記に述べる。

1. プログラムの表現形式の変更

   効率的な要求駆動型部分冗長除去においては、３番地コードに変換されていることを仮定していた。たとえば、$x_0 = a[i_0]$ は、つぎのように変換される（図 6.49）。

```
(SET (REG divex_0) (MUL (REG "i_0") (INTCONST 4)))
(SET (REG divex_1) (ADD (FRAME "a") (REG divex_0))
(SET (REG divex_2) (MEM (REG divex_1)))
(SET (REG x_0) (REG divex_2))
```

図 6.49: ３番地コードに変換された $x_0 = a[i_0]$ の表現

   スカラー置換では、配列要素をこのような３番地コードに変換しない方が扱いやすいので、MEM 式は３番地コードに変換しないようにした。

2. クエリが解を得るタイミングの変更

   配列のインデックスが ϕ 関数を通して得られている場合、クエリを戻り辺 (back edge) に沿って伝播すると、reachValueMap に配列要其の値番号が記録されていないため、伝播先では false が返される。この不都合を解消するために、クエリが解を得るタイミングの変更をおこなう必要がある。

3. クエリの式の書き換え規則

   クエリが配列のインデックスの更新式を通過したとき、クエリ内のインデックスの更新をおこない、クエリの式の書き換えをおこなわなければならない。

4. クエリの訪問条件

   クエリ伝播をおこなうとき、１回目でクエリにマッチしなかったものが、インデックスの更新によって、マッチするようになる場合がある。このような場合に対応できるように、クエリの訪問を２回まで許すように変更した。

6.21 実装上の制限

- volatile 変数

  volatile 変数については, 枠組だけはいれてあるが, 未実装である.

第7章　簡単な別名解析

本システムでは, SSA 形式での最適化の対象を仮想レジスタとし, ひとつのレジスタに対する代入を静的に単一とすることにより SSA を実現している. 仮想レジスタとは, サブプログラムで使用される変数のうち, 以下のような別名をもつ可能性のある変数を除いた変数に限り割り当てられる, 数を無限と仮定したレジスタのことである.

- グローバル変数
- (ローカル変数だが)static 宣言されている変数
- & 演算子の対象となっているもの
- 配列, 配列要素
- 構造体, 構造体要素
- 共用体, 共用体要素
- extern 変数 (ローカル変数ではない)

仮想レジスタに割り当てられた変数は明らかに別名がないとわかっているので SSA 形式での最適化の対象とするのは容易である. しかし一般的な C プログラムでは, 配列やポインタで指された変数など, ひとつの変数が複数の別名を持つ場合があり, さらなる最適化を行おうとした場合, それらの変数も最適化の対象とすることが望ましい. しかし SSA 形式における最適化では別名が存在するものの扱いが難しい. そこで, より精度の高い別名解析を行い, 別名がない変数を新たに仮想レジスタに割り当てて, 最適化の適用範囲を広げることが考えられる.

本システムでは, 別名解析の第一近似として, プログラム中のメモリ参照をひとつの大きなオブジェクトとして扱う方式を実装した. これを簡単な別名解析と呼ぶ. これにより最適化の範囲を LIR の MEM 式にまで広げることが可能となる. つまり, 図 7.1(a) のようなプログラムに対して, 簡単な別名解析と共通部分式除去を行うことで (b) のように変換されることが期待される.

```
a[i] = 1;                    a[i] = 1;
x = a[i];                    x = a[i];
b = a[i] + 1;                b = x + 1;
c = b * a[i] + 1;            c = b * x + 1;
    (a)                          (b)
```

図 7.1: 簡単な別名解析と共通部分式除去を組み合わせて実行した例

この簡単な別名解析の方法では, メモリに対する書き込みを, 書き込まれる変数名に関係なく, メモリ空間のすべての値に対しての変更とみなす. 例えば次の (1) から (3) までの式を考えてみる.

```
(SET I32 (MEM I32 (FRAME I32 "x") (CONST I32 0)) - (1)
(SET I32 (MEM I32 (FRAME I32 "y") (CONST I32 0)) - (2)
(SET I32 (REG I32 r1) (MEM I32 (FRAME I32 "x"))) - (3)
```

これらの式では (1) および (2) の式で x, y には値 0 が代入される. (2) の式で y に対して値 0 を代入した際に x の値は壊されることはないが, この簡単な別名解析の方式では, メモリに対して書き込みがあった場合に, メモリ中のすべての値に対して変更があったとみなすため, (3) の式で x に 0 が入っているとはみなされない.

本システムでは, このことを MEM 式に対して SSA 変換と同じアルゴリズムを適用し, MEM 式の第 2 オペランドを仮想的に設定する. 第 2 オペランドは INTCONST であり, 値は 0 からの連続番号とする. 先程の例は以下のように変換される. つまり, MEM 式の第 2 オペランドが同じ値であれば, メモリの内容が変化していないことが保証されていることになる.

```
(SET I32 (MEM I32 (FRAME I32 "x") (INTCONST 0)) (CONST I32 0)) - (1)'
(SET I32 (MEM I32 (FRAME I32 "y") (INTCONST 1)) (CONST I32 0)) - (2)'
(SET I32 (REG I32 r1) (MEM I32 (FRAME I32 "x") (INTCONST 1)))  - (3)'
```

メモリへの代入がある基本ブロックが支配辺境を持つ場合は, 仮想レジスタの SSA 形式変換と同様, メモリに対する複数の定義を単一化する必要がある. 仮想レジスタの SSA 形式変換では仮想的な命令である ϕ 関数を利用したが, メモリではそれを利用せずに, 支配辺境で第 2 オペランドの値 (連続番号) をひとつインクリメントすることで定義の単一化を行う.

本実装ではメモリ参照の簡単な別名解析の結果を, メモリ参照を行う LIR(MEM 式) に対して情報を付加した. つまり新たな LIR の追加や拡張を行うことなしに実装したので, 簡単な別名解析を行った場合と行わない場合での最適化等の処理を分ける必要がない[1].

[1] これは簡単な別名解析を実行した際に, 既存の最適化等が動かなくなることがないという意味である. メモリについての簡単な別名解析を行った結果を反映して最適化を行う場合には, 当然最適化に処理を追加する必要がある.

第8章　LIR から C プログラムへの変換器

本システムの出力は LIR である. SSA 形式上での最適化のコード変換が正しいものかどうか判断するためには以下の方法が考えられる.

1. 変換された LIR を目で確認する
2. 変換された LIR を基盤部コード生成器にわたし, 実行コードを得て, それを実行する

2 の方法は gcc などの, 比較的信頼のおけるコンパイラが出力した実行コードの実行結果と比較ができて判断は容易である. しかしこの方法では, 本システムが行う最適化のコード変換が, その最適化の意味に対応した正しい変換であるかどうかを判断することはできない. それに対し, 1 の方法では, 本システムでの最適化結果が, 最適化の意味に対応した変換であるかどうかまで判断することができる.

コンパイラの最適化を作成している立場からは, 最終的なコードの正しさの判断は 1 の方法で行うのが妥当であると思われる. しかし最適化されたコードを LIR 上で目で確認するのは困難である. そこで, デバグ用途として, LIR から C プログラムに変換する変換器を実装した. この変換器 (以下 LirToC) は SSA 形式最適化システムに含まれるものではなく, COINS 基盤部の一部として統合されている.

LIR の命令はアセンブラに近く, 演算子等はそのまま C の演算子と対応するものが多い. しかし, LIR 上では既に失われてしまった情報もあり, 入力されたソースプログラムそのものに変換することは困難である. そこで LirToC では, 以下のものについて, LIR から C への特別な変換規則を定めている.

- 条件分岐文
- 配列
- 変数の宣言

以下, これらの詳細について記述する.

8.1　条件分岐文

LIR での条件分岐文には “JUMPC” と “JUMPN” がある. “JUMPC” は二方向分岐であり, “JUMPN” は多方向分岐である. これらは C 言語において, 各々 “if 文” と “switch 文” に相当すると考えてよい[1]. そこで LirToC では, “JUMPC” をすべて if-else の形に変換している. 図 8.1 に “JUMPC” の, 図 8.2 に “JUMPN” の変換例を示す.

[1] if 文以外の場合でも JUMPC に変換されることが考えられるが, if 文に変換したところでプログラムは意味的に変わらない.

(LIR での表現)

```
(JUMPC (TSTLTS I32 (REG I32 "i.3%_1") (REG I32 "n.1%_1"))
       (LABEL I32 ".L4")
       (LABEL I32 ".L15"))
```

(変換後の C プログラムでの表現)

```
if (i_3__1 < n_1__1) {
    goto _L4;
}
else {
    goto _L15;
}
```

図 8.1: 条件付 JUMP の変換例 (JUMPC)

(LIR での表現)

```
(JUMPN (REG I32 "a.1%_3")
       (((INTCONST I32 1) (LABEL I32 ".L6"))
        ((INTCONST I32 2) (LABEL I32 ".L7"))
        ((INTCONST I32 3) (LABEL I32 ".L8"))
        ((INTCONST I32 4) (LABEL I32 ".L9")))
       (LABEL I32 ".L10"))
```

(変換後の C プログラムでの表現)

```
switch(a_1__3) {
    case  1: goto _L6;break;
    case  2: goto _L7;break;
    case  3: goto _L8;break;
    case  4: goto _L9;break;
    default: goto _L10;
}
```

図 8.2: 条件付 JUMP の変換例 (JUMPN)

8.2 配列

LIR では配列という概念はない. LIR では, 入力言語でメモリ参照を表す式は, 全て MEM 式として表現される. つまり C 言語での

$$key = a[j];$$

と

$$key = *(a + j);$$

は, a が int で宣言されていると仮定すると同じ LIR となる. そこで, LIR の MEM 式で表されたものは全て C 言語のポインタを利用した形に変換することにした. また, LIR でのアドレス計算は全て 8 ビット (1 バイト) を単位としているので, C 言語に変換の際にはすべて (unsigned char *) に変換してから計算をするように実装した. 図 8.2 に変換例を示す.

(C 言語での表現)

```
key = a[j]; または key = *(a+j);
```

(LIR での表現)

```
(SET I32 (REG I32 "key.4%_1")
         (MEM I32 (ADD I32 (FRAME I32 "A.1")
                           (LSHS I32 (REG I32 "j.2%_1")(INTCONST I32 2)))))
```

(変換後の C プログラムでの表現)

```
key_4__1 = *(((unsigned char *)&(A_1) + (j_2__1 <<  2)));
```

注) 配列 A_1 は一次元配列で宣言されている

図 8.3: 配列の変換例

8.3 変数の宣言

変数の宣言は, 各サブプログラム内にあるローカルシンボルテーブル及びグローバルシンボルテーブルの情報を元に, 宣言部分を作成する.

LIR の型である Ltype から C 言語の型を得る方法は, 本来ならばコンパイルする環境に依存する. しかし現在では表 8.1 のような対応に固定している.

表 8.1: Ltype と C 言語の型の関係

Ltype	C 言語の型
I8	char
I16	short
I32	int
I64	long
F32	float
F64	double
F128	long double

なお, 配列, 構造体, 共用体の Ltype は A320 のように "A+サイズ" の形をしている. サイズはビット単位である. この情報から配列等の宣言を得るために, サイズを, 同じくシンボルテーブルに書かれている要素の型 (アライン) の情報をもとに配列等の長さを計算し宣言を作成する (式 8.1).

$$(\text{配列等の長さ}) = \frac{(\text{サイズ})}{8 \times (\text{アライン})} \tag{8.1}$$

8.4 既知の問題点

LirToC では以下の機能が制限されている.

- 関数ポインタ
  関数ポインタの宣言のデコードに問題があり, 正しく C プログラムに変換できない問題点がある.

- ポインタ
  ポインタ変数は signed int 型の変数に変換される. この理由は, ポインタ変数は LIR のシンボルテーブル上では Ltype が I32 の変数と見なされ, ポインタ変数であるか否かの情報が書かれていないからである.

- 符号付き / 符号なし
  LIR のシンボルテーブル上には変数が符号つき (signed) であるか符号なし (unsigned) であるかの区別がない. LirToC では, デフォルトで signed の変数に変換する. また, いくつかの LIR の演算は符号付き / 符号なしの 2 種類がある[2]. LirToC では, デフォルトで signed の演算に変換する.

- グローバル変数の初期化
  現状の LirToC の実装においては, グローバル変数の初期化及び宣言は文字列だけをその対象とする.

なお, ϕ 関数は, 実行はできないが C 言語の関数呼び出しの形式で生成される.

[2]例えば DIVS と DIVU 等

第9章　Counting Instructions

9.1　概要

本ユーティリティは、COINS コンパイル時の SSA オプション内に cntbb が記述されている場合に、その cntbb の処理の直前に作成された LIR コードの各基本ブロックの LIR 命令数をあらかじめ COINS コンパイル時に数え (計数) ておき、その基本ブロックが実行されたときにその計数を加算してファイル (計数結果ファイルという) に実行した LIR 命令数を加えこむことにより、各基本ブロックの LIR 実行命令数の総数を測定するものである。

9.2　使い方

ここでは、ソースファイル fib.c に共通部分式除去 (SSA オプション cse) の LIR 実行命令数のカウントをおこなうことを例として説明する。

fib.c は、つぎのようなファイルである：

```
#include<stdio.h>
/* fib.c */
int main(){
  int n;
  int fib;
  int i;
  n = 7;
  fib = 1;
  i = 1;
  do{
    if(i < n){
      i = i + 1;
      fib = fib * i;
    }else{
      break;
    }
  }while(1);
  /* uncomment by FUKUOKA Takeaki */
  printf("fib[%d] = %d\n", n, fib);
  return 0;
}
```

図 9.1: COINS コンパイルの対象ファイル fib.c の内容

1. COINS コンパイル前の注意

   本ユーティリティは、COINS コンパイル時に、作業用ディレクトリに作業用ファイル `file_list.dat`、および計数結果ファイル (9.2.1 節を参照) を書き出すため、COINS コンパイル前に当該ディレクトリにそれらのファイルが存在しないことを確認し、もし存在した場合は、それらを削除 (「9.4.2 作業用ファイル、計数結果ファイルの削除」節を参照) もしくは移動しておく必要がある。

   作業用ディレクトリは、デフォルトでは `/tmp` であるが、COINS オプション `tmpdir` を指定することにより、使用している環境に応じた設定が可能である。

   たとえば、Windows の環境では、COINS オプションとして `tmpdir='d:\cygwin\tmp'` のように作業用ディレクトリを指定することができる。

2. COINS コンパイル

   LIR 実行命令数のカウントをおこなうには、同じソースファイルにたいし、COINS コンパイルを 2 回おこなう必要がある。

   まず、つぎのコマンドにより、COINS コンパイルをおこなう。

```
java.coins.driver.Driver -coins:lir-opt=prun/divex/cse/cntbb/srd3 \
            test/c/TestSsa/fib.c
```

図 9.2: SSA オプション cntbb を指定したコンパイルコマンド例

   ここでは、LIR 実行命令数カウントの対象となる、共通部分式除去 (SSA オプション cse) のつぎに SSA オプション cntbb が挿入されている。したがって、共通部分式除去をおこなって得られた LIR にたいし、実行命令数がカウントされる。

   第 1 回目の COINS コンパイルにより、作業用ディレクトリにファイル `file_list.dat` が作成される。(このファイルには、関数名が記述されるが、これはシステムが内部的に使用するものであり、ユーザはその内容を関知する必要はない。)

   つぎに、第 2 回目の COINS コンパイルでも、第 1 回目と同じコマンドを入力して、COINS コンパイルをおこなう。(この第 2 回目の COINS コンパイルにより、LIR 実行命令の計数命令とプリント関数の呼び出しが LIR として埋め込まれる。)

   このとき、上記の例では、(出力ファイル名が指定されていないので) 実行形式のファイル a.out が作成される。

3. 実行

   実行形式ファイル a.out を実行すると、作業用ディレクトリに計数結果ファイルが書き出される。

   この例では、作業用ディレクトリ `/tmp` に作業用ファイル `fib.c.main.cnt` が書き出される。

   また、計数結果ファイルの表示には、表示用ユーティリティ (9.4.1 節を参照) を使うこともできる。

9.2.1 計数結果ファイルについて

計数結果ファイルは、"< ファイル名 >.< 関数名 >.cnt" という名前をもち、計数結果はそのファイルに、各基本ブロック毎に "< 番号 >:<LIR 実行命令数 >< 改行 >" の形式で書き出される。ただし、ここでの < 番号 > は、" 基本ブロック番号-1" になっている。

上記の例では、計数結果ファイルは、`fib.c.main.cnt` であり、計数結果は、つぎのようになる：

0: 10

1: 7

2: 18

3: 4

4: 0

5: 1

6: 21

図 9.3: 計数結果ファイルの内容

9.3 LIR 実行命令数のカウントと出力の処理概要

本ユーティリティは、「使い方」で述べたように 2 回の COINS コンパイルと 1 回の実行によって LIR 命令実行回数の情報を収集するが、処理の概要はつぎの通りである。

1. コンパイル 1 回目：コンパイルの際に出現した関数名と基本ブロック数とを対にして、作業用ファイル (file_list.dat) に記録する。

   「使い方」の例でいえば、fib.c.main:7 が file_list.dat に書かれる。

2. コンパイル 2 回目：各関数にたいし基本ブロックごとの LIR 命令実行回数を記録する配列 (上記の例では fib.c.main という名前) を用意し、つぎの処理をおこなう。

   (a) 各基本ブロック (基本ブロック番号を “i” とする) に次の LIR 命令を付加する。

   `fib.c.main[i]=fib.c.main[i]+`「基本ブロック i の LIR 命令数」

   (b) すべての関数に対する配列の内容を計数結果ファイルに出力する関数 (LIR で記述されている) を main 関数の終了節の先行節の終わりと exit 関数の呼出しの手前で呼び出すようにする。

9.4 LIR 実行命令数表示、作業用ファイル等削除のユーティリティ

LIR 実行命令数の表示、作業用ディレクトリにある作業用ファイル、計数結果ファイルの削除をおこなうユーティリティクラス ProApp を用意している。

9.4.1 LIR 実行命令数表示

つぎのコマンドを実行することにより、LIR 実行命令数表示をおこなうことができる：

```
java -cp ./classes coins.ssa.ProApp -t 'd:\cygwin\tmp'
```

図 9.4: ProApp による計数結果表示のコマンド入力例

この例では、-t オプションによって、作業用ディレクトリとして 'd:\cygwin\tmp' を指定している。-t オプションが記述されていなければ、/tmp が作業用ディレクトリに指定されているものとする。

実行結果は、いままで用いてきた fib.c の例では、つぎのように表示される：

```
d:\cygwin\tmp\fib.c.main.cnt:
total: 61
1:10
2:7
3:18
4:4
6:1
7:21
```

図 9.5: ProApp をもちいた計数結果の表示

9.4.2 作業用ファイル、計数結果ファイルの削除

つぎのコマンドを実行することにより、LIR 実行命令数カウントに使用された作業用ファイル、計数結果ファイルの削除をおこなうことができる：

```
java -cp ./classes coins.ssa.ProApp -n -t 'd:\cygwin\tmp'
```

図 9.6: ProApp による作業用ファイル、計数結果ファイル削除のコマンド入力例

-n オプションによって、作業用ファイル、計数結果ファイル削除の指示をおこなっている。

この例では、-t オプションによって、作業用ディレクトリとして 'd:\cygwin\tmp' を指定している。-t オプションが記述されていなければ、/tmp が作業用ディレクトリに指定されているものとする。

ただし、本コマンドを実行後は、計数結果は保存されていないので、再表示はできなくなるので注意が必要である。

9.5 使用上の注意

1. lir-opt(もしくは ssa-opt) では、cntbb を複数回指定できないものとする
2. 本ユーティリティの結果の出力に、fprintf などの関数を使用しているため、x86_64-mac では、本ユーティリティを使用できない
3. つぎの LIR 命令は、LIR 実行命令数のカウントから除外する：

- PHI 命令、PROLOGUE 命令、EPILOGUE 命令
- LIR 実行命令数カウントのために埋め込まれた LIR 命令

第10章　Optimistic Register Coalescing

現在、COINS で用いられているレジスタ割り付けアルゴリズムは、グラフ彩色アルゴリズムの改良型である George-Appel の「iterated register coalescing」[13] をベースにしている。この方法は coalescing(合併）を「保守的」にしか行わない。

これにたいし、ここで述べるアルゴリズムは、Park-Moon の「optimistic register coalescing」[14] をベースにしたものである。この方法は coalescing を「楽観的」に行い、iterated register coalescing よりもよいレジスタ割り付けができると主張されている。

今回実装した部分はこのバックエンドのレジスタ割り付け部分にあたる。詳細に関しては、[23] を参照のこと。

10.1　従来 COINS 上で実装されてきたアルゴリズム

従来の COINS 上で実装されてきた彩色アルゴリズムについて説明をする。図 10.1 がアルゴリズムの流れである。

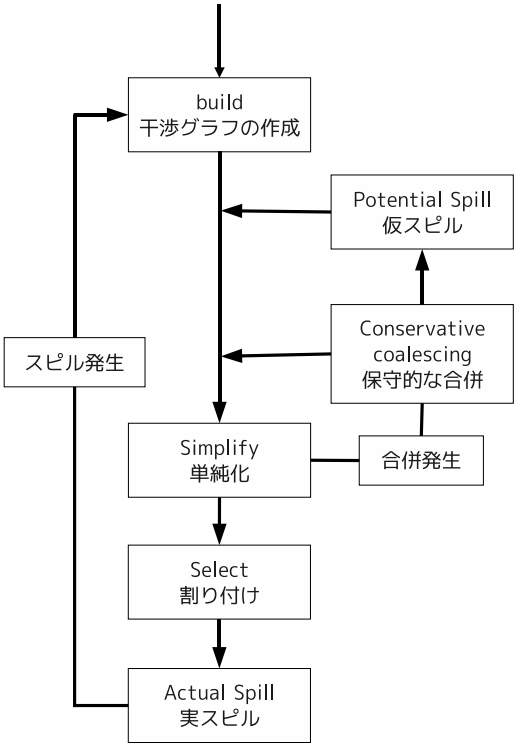


図 10.1: 従来の COINS 上で実装されてきた彩色アルゴリズム

この手法は、George-Appel の「iterated register coalescing」[13] をベースとしており、保守的な合併とスピルを二段階に分ける方法 [4] を組み合わせた彩色アルゴリズムである。この手法は、割り付け可能性を損なうかもしれない合併候補はスピルを恐れ合併を行わず、安全な合併と単純化、仮スピルを繰り返すことでグラフの状態を彩色可能性を損なわず簡略化していく。また保守的な合併により、合併後のノードがスピルの対象となる事を回避している。

従来の Iterated Register Coalescing には、保守的な合併により、合併ノードによるスピルを回避しているという利点もあるが、以下の問題点がある。

保守的な合併の際、合併を恐れた合併候補は合併しても必ずスピルされる訳ではない。よって幾らかの合併は無駄に諦めてしまっている。また合併によりグラフの状態が良くなる可能性もあるが保守的な合併により見逃している。

この問題点にたいし、合併をさらに多く行う、合併ノードのスピルの数は増やさない、といった改善点があげられる。

これらを満たすアルゴリズムが次に紹介するアルゴリズムである。

10.2 今回 COINS 上で実装を行ったアルゴリズム

今回 COINS 上で実装を行った彩色アルゴリズムについて説明をする。図 10.2 がアルゴリズムの流れである。この手法は、Park-Moon の optimistic coalescing [14] をベースとしており、積極的な合併と、合併取り止めを用いたアルゴリズムである。

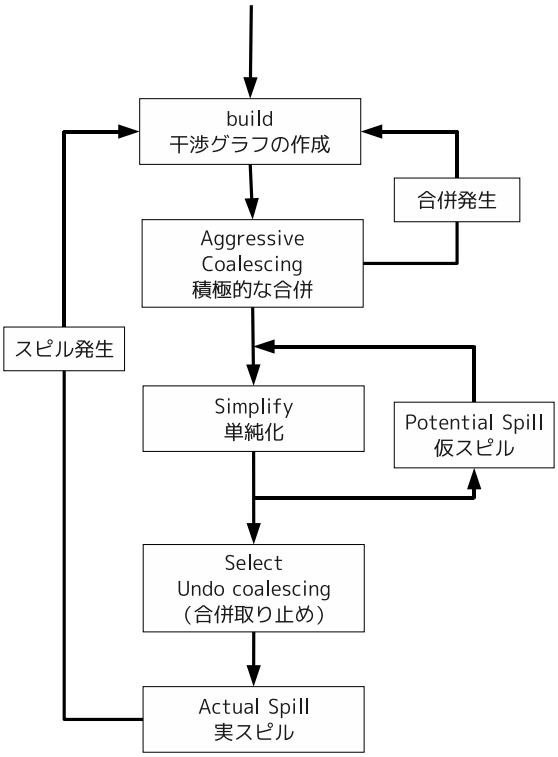


図 10.2: 今回 COINS 上で実装を行った彩色アルゴリズム

合併取り止め (Undo Coalescing) とは、積極的に合併をし、かつ合併ノードのスピルを防ぐ為の手

法である。合併ノードに対してスピルが必要になった場合にその合併を取り止め、前の状態に戻すことで彩色を考え直すというアイデアである。具体的に図で説明をする。図 10.3 のグラフの彩色を考える。（使用可能レジスタは 2 つ）

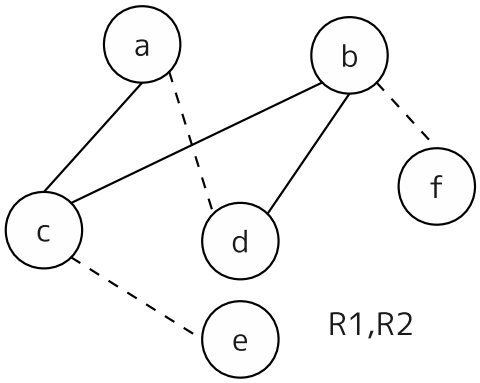


図 10.3: 干渉グラフの例

積極的に合併候補を全て合併すると図 10.4 のようになる。

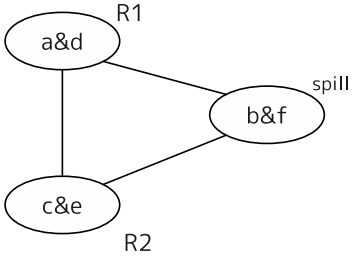


図 10.4: Undo Coalescing の例（合併候補をすべて合併）

この状態ではレジスタ 2 では割り当てが出来ず、合併ノードの中でどれかスピルの候補とならないといけない。（図では b & f）

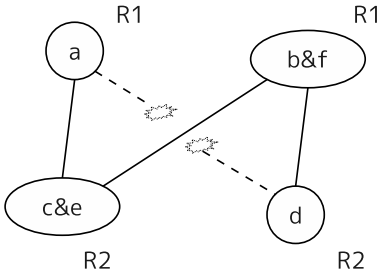


図 10.5: Undo Coalescing の例（合併取り止め）

しかし、ここで合併取り止めで図 10.5 のようにする事で 2 つのレジスタで割り付ける事が可能となる。

10.3 実装について

LIR(低水準中間表現) のバックエンドにおけるレジスタ割り付け (RegisterAllocation) 部分を変更した。今回用いた COINS のバックエンド上では手続きごとにレジスタ割り付けを行っており、本実装もそれと同様とする。

10.4 使い方

Optimistic Coalescing Allocation を使用するためには、coins オプションでその指定をする必要がある（デフォルトは、従来のアルゴリズム）：

```
-coins:regallo=oca
```

10.5 現状の問題点

OCA(Optimistic Coalescing Allocation) 適用により、レジスタ割り付け限度を越えるため、exception が発生する場合がある。

これは、actualSpill によって作成されたコピー文に含まれるレジスタが再び actualSpill の対象になるためであり、改良が望まれる。

第11章　文番号付加

COINS には、もともとソースプログラムでの行番号 (ソース行番号) を取り込む仕組みが存在し、coins オプション debuginfo が設定されていれば、HIR から LIR への変換の際に、行番号が (LINE < 行番号 >) という LIR 式 (LINE 文ともいう) として、各基本ブロックの先頭に挿入される。しかしこの行番号は、 HIR を通って LIR になるまでに、行の改変、挿入がおこなわれ、すべての LIR 式に行番号が対応しているわけではないので、COINS の SSA 最適化をデバッグするとき、かならずしも適切な指標ではない。

本章で扱う文番号は、いま述べた行番号とは異なり、SSA 最適化において各関数毎に１から始まる番号を LIR 文にふった番号である。(もともとあった行番号と区別するために文番号としている。)

また、ソース行番号を表す LINE 文にも文番号がふられるので、ソース行番号と LIR 文番号を対として情報を利用することも可能である。

11.1　使い方

文番号に関する操作は、ssa オプションで指定することにより、行うことができる。
関連する ssa オプションは、以下の通りである。(具体的な使用例は、11.3 節を参照のこと。)

1. stlin

   番号付けをおこなう（各 LIR 文の opt に新に作成された文番号を書き込む）

2. inslin

   LINE 文を挿入する（各 LIR 文の直前にその LIR 文の文番号を表す LINE 文が挿入される）

   ただし、LIR 文が PHI 関数である場合には、文番号を表す LINE 文は挿入されない。

3. rmlin

   LINE 文を削除する（LINE 文を表す LIR 文が Function から削除される）

4. shlin

   LINE 文を表示する（”LINE xx:<LIR 文 >” の形で、各 Function に含まれる LIR 文を標準出力に表示する。ここで xx は、文番号の先頭に”-” をつけて負の整数として表示している。）

文番号付加の処理は、つぎのようにおこなわれる。

- 各 Function のフローグラフの basicBlk リストに属す basicBlk にたいし、それに属す instrList の要素にリストでの出現順に文番号（各 Function で１から始まる番号）付けをおこなう。
- 文番号は、”SSA_LINExxx” の形式の String で、LirNode の opt に保持される。(opt への書き込みは、stlin によっておこなわれる。）ここで、xxx は、正の整数をあらわす数字である。
- inslin では、opt に保持されている番号を負の整数に変換して、LINE 文を作成し、instrList に挿入する。
- shlin では、Function に含まれる LIR 文を標準出力に出力する。

11.2 利用方法

SSA 最適化をおこなった際の LIR の移動を調べる場合には、つぎのように文番号付加機能をつかうことができる。この利用のためには、11.3 節の使用例が参考になる。

1. 調べたい部分の SSA 最適化の直前に、stlin を指定して LIR 文全体に文番号を付加する。
2. shlin により、いまつけた文番号を表示する。
3. つぎに調べたい部分の SSA 最適化をおこない、それが終ったところでふたたび shlin で文番号を表示して、変換の効果を確認する。

また、11.3 節の使用例中に表示されているように (たとえば、LINE -2:(LINE 12 SSA_LINE2))、ソース行番号 12 に文番号 2 が割り当てられているので、この情報をもとにした利用も可能である。

11.3 使用例

対象プログラム coins/test/c/TestSsa/cse_test.c に debuginfo オプションと
ssa-opt=stlin/shlin/prun/shlin/cse/shlin/srd3/shlin
という SSA オプションのもとで coins コンパイルを実行したときの表示例を以下に示す。ここでは、付加した文番号の表示が SSA の各処理によってどのようになるかを示すために、最初に stlin を入れている：

注 ここで stlin が実行される。

注 つぎの表示は、shlin による。

注 LIR 文に含まれる ”SSA_LINExx” は、LIR ノードの opt 内に記述された文番号である。

注 COINS オプションで debuginfo を指定することにより挿入されたソース行番号にも文番号を付加している。たとえば、`LINE -2:(LINE 12 SSA_LINE2)` における “-2” は、LIR 文 (LINE 12) の文番号である。

```
Function "main":
Basic block.L1:
  LINE -1:(PROLOGUE (0 0) SSA_LINE1)
  LINE -2:(LINE 12 SSA_LINE2)
  LINE -3:(SET I32 (REG I32 "a.1%") (INTCONST I32 1) SSA_LINE3)
  LINE -4:(LINE 13 SSA_LINE4)
  LINE -5:(SET I32 (REG I32 "b.2%") (INTCONST I32 2) SSA_LINE5)
  LINE -6:(LINE 14 SSA_LINE6)
  LINE -7:(SET I32 (REG I32 "x.5%") (INTCONST I32 3) SSA_LINE7)
  LINE -8:(LINE 16 SSA_LINE8)
  LINE -9:(SET I32 (REG I32 "y.6%") (ADD I32 (REG I32 "a.1%") (REG I32 "b.2%")) SSA_LINE9)
  LINE -10:(LINE 17 SSA_LINE10)
  LINE -11:(SET I32 (REG I32 "c.3%") (ADD I32 (REG I32 "x.5%") (REG I32 "y.6%")) SSA_LINE11)
  LINE -12:(LINE 19 SSA_LINE12)
  LINE -13:(JUMPC (TSTLTS I32 (REG I32 "a.1%") (INTCONST I32 10)) (LABEL I32 ".L3") (LABEL I32 ".L4") SSA_LINE13)
Basic block.L3:
  LINE -14:(LINE 20 SSA_LINE14)
  LINE -15:(SET I32 (REG I32 "z.7%") (ADD I32 (REG I32 "a.1%") (REG I32 "b.2%")) SSA_LINE15)
  LINE -16:(JUMP (LABEL I32 ".L5") SSA_LINE16)
Basic block.L4:
  LINE -17:(LINE 22 SSA_LINE17)
  LINE -18:(SET I32 (REG I32 "z.7%") (ADD I32 (REG I32 "a.1%") (REG I32 "b.2%")) SSA_LINE18)
  LINE -19:(JUMP (LABEL I32 ".L5") SSA_LINE19)
Basic block.L5:
  LINE -20:(LINE 24 SSA_LINE20)
  LINE -21:(SET I32 (REG I32 "d.4%") (ADD I32 (REG I32 "x.5%") (REG I32 "z.7%")) SSA_LINE21)
  LINE -22:(LINE 26 SSA_LINE22)
  LINE -23:(CALL (STATIC I32 "printf") ((STATIC I32 "string.10") (REG I32 "d.4%"))
                ((REG I32 "functionvalue.9%")) &id ("printf" 21) SSA_LINE23)
  LINE -24:(LINE 27 SSA_LINE24)
  LINE -25:(SET I32 (REG I32 "returnvalue.8%") (REG I32 "d.4%") SSA_LINE25)
  LINE -26:(JUMP (LABEL I32 ".L6") SSA_LINE26)
Basic block.L6:
  LINE -27:(EPILOGUE (0 0) (REG I32 "returnvalue.8%") SSA_LINE27)
```

注 ここで prun が実行される。

注 つぎの表示は、shlin による。

注 新たに付加された LIR 文は、LIR の opt 内に文番号をもたず ”LINE 0” で表示される。

```
Function "main":
Basic block.L1:
  LINE -1:(PROLOGUE (0 0) SSA_LINE1)
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "returnvalue.8%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "functionvalue.9%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "d.4%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "z.7%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "c.3%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "y.6%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "x.5%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "b.2%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "a.1%_0")))
  LINE -2:(LINE 12 SSA_LINE2)
  LINE -3:(SET I32 (REG I32 "a.1%_1") (INTCONST I32 1) SSA_LINE3)
  LINE -4:(LINE 13 SSA_LINE4)
  LINE -5:(SET I32 (REG I32 "b.2%_1") (INTCONST I32 2) SSA_LINE5)
  LINE -6:(LINE 14 SSA_LINE6)
  LINE -7:(SET I32 (REG I32 "x.5%_1") (INTCONST I32 3) SSA_LINE7)
  LINE -8:(LINE 16 SSA_LINE8)
  LINE -9:(SET I32 (REG I32 "y.6%_1") (ADD I32 (REG I32 "a.1%_1") (REG I32 "b.2%_1")) SSA_LINE9)
  LINE -10:(LINE 17 SSA_LINE10)
  LINE -11:(SET I32 (REG I32 "c.3%_1") (ADD I32 (REG I32 "x.5%_1") (REG I32 "y.6%_1")) SSA_LINE11)
  LINE -12:(LINE 19 SSA_LINE12)
  LINE -13:(JUMPC (TSTLTS I32 (REG I32 "a.1%_1") (INTCONST I32 10)) (LABEL I32 ".L3" $1) (LABEL I32 ".L4" $2) SSA_LINE13)
Basic block.L3:
  LINE -14:(LINE 20 SSA_LINE14)
  LINE -15:(SET I32 (REG I32 "z.7%_2") (ADD I32 (REG I32 "a.1%_1") (REG I32 "b.2%_1")) SSA_LINE15)
  LINE -16:(JUMP (LABEL I32 ".L5" $3) SSA_LINE16)
Basic block.L4:
  LINE -17:(LINE 22 SSA_LINE17)
  LINE -18:(SET I32 (REG I32 "z.7%_1") (ADD I32 (REG I32 "a.1%_1") (REG I32 "b.2%_1")) SSA_LINE18)
  LINE -19:(JUMP (LABEL I32 ".L5" $4) SSA_LINE19)
Basic block.L5:
  LINE 0:(PHI I32 (REG I32 "z.7%_3") ((REG I32 "z.7%_2") (LABEL I32 ".L3" $6) (LABEL I32 ".L5" $3))
                                     ((REG I32 "z.7%_1") (LABEL I32 ".L4" $7) (LABEL I32 ".L5" $4)))
  LINE -20:(LINE 24 SSA_LINE20)
  LINE -21:(SET I32 (REG I32 "d.4%_1") (ADD I32 (REG I32 "x.5%_1") (REG I32 "z.7%_3")) SSA_LINE21)
  LINE -22:(LINE 26 SSA_LINE22)
  LINE -23:(CALL (STATIC I32 "printf") ((STATIC I32 "string.10") (REG I32 "d.4%_1"))
                                       ((REG I32 "functionvalue.9%_1")) &id ("printf" 21) SSA_LINE23)
  LINE -24:(LINE 27 SSA_LINE24)
  LINE -26:(JUMP (LABEL I32 ".L6" $5) SSA_LINE26)
Basic block.L6:
  LINE -27:(EPILOGUE (0 0) (REG I32 "d.4%_1") SSA_LINE27)
```

注 ここで cse が実行される。

注 つぎの表示は、shlin による。

```
Function "main":
Basic block.L1:
  LINE -1:(PROLOGUE (0 0) SSA_LINE1)
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "returnvalue.8%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "functionvalue.9%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "d.4%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "z.7%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "c.3%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "y.6%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "x.5%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "b.2%_0")))
  LINE 0:(CALL (STATIC I32 "__dummy_function") () ((REG I32 "a.1%_0")))
  LINE -2:(LINE 12 SSA_LINE2)
  LINE -3:(SET I32 (REG I32 "a.1%_1") (INTCONST I32 1) SSA_LINE3)
  LINE -4:(LINE 13 SSA_LINE4)
  LINE -5:(SET I32 (REG I32 "b.2%_1") (INTCONST I32 2) SSA_LINE5)
  LINE -6:(LINE 14 SSA_LINE6)
  LINE -7:(SET I32 (REG I32 "x.5%_1") (INTCONST I32 3) SSA_LINE7)
  LINE -8:(LINE 16 SSA_LINE8)
  LINE -9:(SET I32 (REG I32 "y.6%_1") (ADD I32 (REG I32 "a.1%_1") (REG I32 "b.2%_1")) SSA_LINE9)
  LINE -10:(LINE 17 SSA_LINE10)
  LINE -11:(SET I32 (REG I32 "c.3%_1") (ADD I32 (REG I32 "x.5%_1") (REG I32 "y.6%_1")) SSA_LINE11)
  LINE -12:(LINE 19 SSA_LINE12)
  LINE -13:(JUMPC (TSTLTS I32 (REG I32 "a.1%_1") (INTCONST I32 10)) (LABEL I32 ".L3" $1) (LABEL I32 ".L4" $2) SSA_LINE13)
Basic block.L3:
  LINE -14:(LINE 20 SSA_LINE14)
  LINE -16:(JUMP (LABEL I32 ".L5" $3) SSA_LINE16)
Basic block.L4:
  LINE -17:(LINE 22 SSA_LINE17)
  LINE -19:(JUMP (LABEL I32 ".L5" $4) SSA_LINE19)
Basic block.L5:
  LINE -20:(LINE 24 SSA_LINE20)
```

```
  LINE -22:(LINE 26 SSA_LINE22)
  LINE -23:(CALL (STATIC I32 "printf") ((STATIC I32 "string.10") (REG I32 "c.3%_1"))
                                       ((REG I32 "functionvalue.9%_1")) &id ("printf" 21) SSA_LINE23)
  LINE -24:(LINE 27 SSA_LINE24)
  LINE -26:(JUMP (LABEL I32 ".L6" $5) SSA_LINE26)
Basic block.L6:
  LINE -27:(EPILOGUE (0 0) (REG I32 "c.3%_1") SSA_LINE27)
```

注 ここで srd3 が実行される。

注 つぎの表示は、shlin による。

```
Function "main":
Basic block.L1:
  LINE -1:(PROLOGUE (0 0) SSA_LINE1)
  LINE -2:(LINE 12 SSA_LINE2)
  LINE -3:(SET I32 (REG I32 "a.1%_1") (INTCONST I32 1) SSA_LINE3)
  LINE -4:(LINE 13 SSA_LINE4)
  LINE -6:(LINE 14 SSA_LINE6)
  LINE -8:(LINE 16 SSA_LINE8)
  LINE -10:(LINE 17 SSA_LINE10)
  LINE -11:(SET I32 (REG I32 "c.3%_1") (ADD I32 (INTCONST I32 3) (ADD I32 (REG I32 "a.1%_1") (INTCONST I32 2))) SSA_LINE11)
  LINE -12:(LINE 19 SSA_LINE12)
  LINE -13:(JUMPC (TSTLTS I32 (REG I32 "a.1%_1") (INTCONST I32 10)) (LABEL I32 ".L3") (LABEL I32 ".L4") SSA_LINE13)
Basic block.L3:
  LINE -14:(LINE 20 SSA_LINE14)
  LINE -16:(JUMP (LABEL I32 ".L5") SSA_LINE16)
Basic block.L4:
  LINE -17:(LINE 22 SSA_LINE17)
  LINE -19:(JUMP (LABEL I32 ".L5") SSA_LINE19)
Basic block.L5:
  LINE -20:(LINE 24 SSA_LINE20)
  LINE -22:(LINE 26 SSA_LINE22)
  LINE -23:(CALL (STATIC I32 "printf") ((STATIC I32 "string.10") (REG I32 "c.3%_1"))
                                       ((REG I32 "functionvalue.9%_1")) &id ("printf" 21) SSA_LINE23)
  LINE -24:(LINE 27 SSA_LINE24)
  LINE -26:(JUMP (LABEL I32 ".L6") SSA_LINE26)
Basic block.L6:
  LINE -27:(EPILOGUE (0 0) (REG I32 "c.3%_1") SSA_LINE27)
```

第12章　スカラー置換（この章は削除）

12.1　概要

スカラー置換は、配列要素をスカラーに置き換えることにより、配列要素へのロード、ストアの回数を減らし、実行効率を高めるためのものである。

たとえば、つぎのコード（図 12.1）

```
DO I=1, 8
  A(I+1)=A(I-1)+B(I-1)
  A(I)=A(I)+B(I)+B(I+1)
ENDDO
```

図 12.1: スカラー置換適用前のコード

にたいしスカラー置換を施すと、つぎのようになる（図 12.1）

```
t0A=A(0), t1A0=A(1), tB=B(0)
DO I=1,N
  t1A1=t0A+tB2
  tB3=B(I+1)
  t0A=t1A0+tB3+tB2
  A(I)=t0A
  t1A0=t1A1; tB1=tB2; tB2=tB3
ENDDO
A(N+1)=t1A1
```

図 12.2: スカラー置換適用前のコード

変換前のコードでは、５個のロードと２個のストアであったものが、変換後のコードでは、ロード、ストア各々１個ずつになっているので、この変換により実行効率の向上が望める。

12.2 スカラー置換のアルゴリズム

```
procedure ScalarReplace(L,G,n)
  // L は、ループ情報である。
  // G は、L に含まれる文間の依存グラフである。
  // n は、配列に使用可能なレジスタの個数である。

  依存グラフで、出力依存、アンチ依存、不整合依存を悪辺（bad edge）として型付融合
  アルゴリズムを適用することにより、名前分割（name partitions）の集合 P をもとめる。

  P の部分集合 {p1,p2,...,pm} で n 個より多くのレジスタをつかわずに、セーブされる
  メモリ参照数を最大にするものをビンパッキングをもちいてもとめる。

  for i:=1 to m do begin
    if pi がサイクリックな名前分割でない
      then ScalarReplacePartition(pi,ki) を適用する;
    // サイクリックな場合、
    else begin
      ScalarReplaceCyclicPartition(pi);
      ki:=1;
    end
    for each \delta（ループ中の各不整合依存）
      InsertMemoryRefs(\delta) を呼び出す;
  end

  // ki は、スカラー置換によって導入される一時変数の個数である
  K を {k1,k2,...,km} の最小公倍数とする。

  // unroll the loop to K loop bodies to eliminate scalar copies
  // スカラーコピーを消去するために、ループを K 個のループ本体に展開する
  UnrollLoop(K);
end ScalarReplace
```

図 12.3: スカラー置換のアルゴリズム

12.3 データ依存グラフ

データ依存関係には、つぎの４種類がある：

- フロー依存 (または真の依存)

  つぎのようなプログラムで、

$$a = \ldots \tag{12.1}$$

$$\ldots$$

$$= a \tag{12.2}$$

(12.1) で定義された a の値が (12.2) で使用される場合、(12.1) の a から (12.2) の a にフロー依存関係があるという。

- アンチ依存 (または逆依存)

  つぎのようなプログラムで、

$$= \quad a \tag{12.3}$$

$$\ldots$$

$$a \quad = \quad \ldots \tag{12.4}$$

  a の値が (12.3) で使用された後に (12.4) で定義される場合、(12.3) の a から (12.4) の a にアンチ依存関係があるという。

- 入力依存

  つぎのようなプログラムで、

$$= \quad a \tag{12.5}$$

$$\ldots$$

$$= \quad a \tag{12.6}$$

  a の値が (12.5) で使用された後に (12.6) でも使用される場合、(12.5) の a から (12.6) の a に入力依存関係があるという。

- 出力依存

  つぎのようなプログラムで、

$$a \quad = \quad \ldots \tag{12.7}$$

$$\ldots$$

$$a \quad = \quad \ldots \tag{12.8}$$

  a の値が (12.7) で使用された後に (12.8) でも使用される場合、(12.7) の a から (12.8) の a に出力依存関係があるという。

12.4 データ依存グラフの枝刈り

12.4.1 データ依存グラフの枝刈りの例

図 12.4 のプログラムにおいて、S1 の A(I+1) から S1 の A(I-1) へのフロー依存、S1 の A(I+1) から S2 の A(I) への出力依存、S2 の A(I+1) から S1 の A(I-1) への入力依存、S2 の B(I) から S1 の B(I-1) への入力依存が枝刈りされる。

```
DO I=1, N
  S1    A(I+1)=A(I-1) + B(I-1)

  S2    A(I)=A(I) + B(I) + B(I+1)
ENDDO
```

図 12.4: データ依存グラフの枝刈りの例

12.4.2 データ依存グラフの枝刈りのアルゴリズム

依存グラフの枝刈りを以下のようにおこなう：

フェーズ 1： kill された依存関係を除去する。

フェーズ 2： generator の特定

このフェーズですべての generator が特定される。枝刈りされた依存グラフでは、generator は、そこから発しループの別の文へのフロー依存が少なくとも 1 つ存在するような代入の reference であるか、そこから発する入力依存を少なくとも 1 つもちそこに至る入力依存もしくはフロー依存が存在するような use reference である。

フェーズ 3： 名前分割の探索と入力依存の除去

各 generator から始めて、その変数の名前分割 (name partition) の一部分として、フロー依存もしくは入力依存でその generator から到達可能な reference に印をつける。名前分割は、reference によって 1 個のスカラー変数に置き換え可能な reference の集合である。同一の名前分割に属す 2 つの要素間の入力依存は、その依存関係のソースが generator 自身でないならば、除去してもよい。

12.4.3 型付融合アルゴリズム

名前分割には、型付融合アルゴリズム (typed fusion algorithm) をもちいるため、本節ではそのアルゴリズムを述べ、次節で名前分割の探索における本アルゴリズムの適用の仕方を述べる。

G=(V,E) をグラフがあたえられたとき、各ノードに種別 (これを型ということにする) を対応させる (型の集まりを T とし、関数 $m : V \to T$ を考える。) t_0 を特定の型とする。また、辺のうち悪辺 (bad edge) というものを指定し、その集まりを B とかく。これらの組 $P = (G, T, m, B, t_0)$ を型付融合問題 (typed fusion problem) という。

これにたいし、つぎの条件を満すグラフ G'=(V', E') を型付融合問題 P の解であるという。すなわち、V' は、つぎの制約にしたがって型 t_0 をもつ V のノードを融合したものである。

Bad edge 制約: Bad edge でつながった 2 つのノードは、融合されない

順序制約: t_0 とは異なる型 t をもつノードを含むパスが存在するような 2 つのノードは融合できない

また、型付融合問題の最適解とは、辺の個数が極小となる解のことである。

```
procedure TypedFusion(G,T,type,B,t_0)
  // G=(V,E) the original typed graph
  // T is a set of types
  // type(n) is a function that returns the type of a node
  // B is the set of bad edges
  // t_0 is a specific type for which we will find a minimal fusion
  // Initialization
  lastnum:=0; lastfused:=0; count[*]:=0; fused:=0; node[*]:=0;
  for each edge e=(m,n) \in E do count[n]:=count[n]+1;
  for each node n \in V do begin
    maxBadPrev[n]:=0; num[n]:=0; next[n]:=0;
    if count[n]=0 then W:=W \cup [n];
  end
  // Iterate over working set,visiting nodes,fusing nodes of type t_0
  while W\=\empty do begin
    let n be an arbitrary element in W; W:=W-{n}; t:=type(n);
L1: if t=t_0 then begin // A node of the type being worked on
      // Compute node to fuse with.
S1:   if maxBadPrev[n]=0 then p:=fused;
      else p:=next[maxBadPrev[n]];
      if p\=0 then begin // Fuse with node at p
        x:=node[p]; num[n]:=num[x];
update_successors(n,t); // visit successors before fusing
fuse x and n and call the result n,
          making all edges out of x be out of n;
      end
      else begin // Make this the first node in a new group
        create_new_fused_node(n);
        update_successors(n,t);
      end
    end
    else begin // t\=t_0
      create_new_node(n);
      update_successors(n,t);
    end
  end
end TypedFusion
```

図 12.5: 型付融合アルゴリズム (その 1)

```
procedure create_new_node(n)
  lastnum:=lastnum+1;
  num[n]:=lastnum;
  node[num[n]]:=n;
end create_new_node
```

図 12.6: 型付融合アルゴリズム (その 2)

```
procedure create_new_fused_node(n)
  create_new_node(n);

  // append node n to the end of fused
  if lastfused=0 then begin
    fused:=lastnum;
    lastfused:=fused;
  end
  else begin
    next[lastfused]:=lastnum;
    lastfused:=lastnum;
  end
end create_new_fused_node
```

図 12.7: 型付融合アルゴリズム (その 3)

```
procedure update_successors(n, t)
l2: for each node m such that (n,m) \in E do begin
      count[m]:=count[m]-1;
      if count[m]=0 then W:=W \cup {m};
      if t\=t_0 then
        maxBadPrev[m]:=MAX(maxBadPrev[m], maxBadPrev[n]);
      else // t=t_0
        if type(m)\=t_0 or (n, m) \in B then //bad edge
          maxBadPrev[m]:=MAX(maxBadPrev[m], num[n]);
        else // equal types and not fusion preventing
          maxBad{rev[m]:=MAX(maxBadPrev[m]. maxBadPrev[n]);
    end
end update_successors
```

図 12.8: 型付融合アルゴリズム (その 4)

12.4.4 型付融合アルゴリズムの適用

各 reference をノードとし、reference 間のデータ依存関係を辺とする。reference の配列名を型とする。出力依存とアンチ依存を bad edges とする。融合されたノードが名前分割となり、融合されたノードに最初に入れられたノードが generator となる。

12.5 ScalarReplacePartition

```
procedure ScalarReplacePartition(g,k)
  // g は、名前分割に含まれる依存の集合である。

  k を g によって張られる繰り返しの総数とする；
  一時変数 {t1,t2,...,tk} を作成する；

  I を置換が適用されるループインデックスとする；
  R(I+l) は、インデックス I の位置に I+l をもつものの
      リファレンスを表すものとする；
  j を g に含まれるリファレンスで I に足される値の最大値とする；
    // これは、j-k+1 が足される値のもっとも小さなものであることを意味する。

  for each R(I+l) （g に含まれる添字参照） begin
    添字参照を t_(j-k+l) で置き換える；
    if 添字参照が代入の左辺である then
      if ループに含まれる別の参照への出力依存が存在しない then
        その参照を含む文の後に、代入文 "R(I+l)=t_(j-k+l)" を挿入する
      else begin // 出力依存が存在する
        R(I+q) を出力依存のシンク側の参照とする；
        if q<l then
          for i:=q+1 to l do
            ループの後に代入文 "R(N+i)=t_(j-k+i)" を挿入する；
               // N=ループ上限値
          end
  end
  l をループの外部からのフロー依存が存在する添字付参照に含まれる I に
      足される値の最大値とする；
  for i:=1 to j-k+l do
    ループの開始の前に "ti=R(i)" を挿入する；
  for i:=1 to k-1 do
    ループの終了の前に "ti=t_(i+1)" を挿入する；
end ScalarReplaceParttion
```

図 12.9: ScalarReplacePartition

上記の ScalarReplacePartition のアルゴリズムに関する注意をのべる。

1. 上記の記述には、入力依存 (input dependence) しかもたないものの処理がない。したがって、たとえば、p.393 の のスカラー置換の処理がおこなえない。この処理をおこなうために、つぎ

```
DO I=1,N
  A(I)=B(I-1)+B(I+1)
ENDDO
```

   の規則を入れることが、考えられる：入力依存しかない partition の場合には、generator G にたいし、t=G を挿入する

12.6 ScalarReplaceCyclicPartition

```
procedure ScalarReplaceCyclicPartition(g)
  // g は、名前分割に含まれる依存関係の集合である。
  if g に含まれる参照がループ不変でない then
    return （置換をおこなわない）;
  R を名前分割のループ不変な参照とする ;
  t を一時変数とする ;
  for each g に含まれる添字付参照 R do R を t で置き換える ;
  if 少なくとも 1 個の参照が代入文の左辺に存在する then
    ループの直後に "R=t" を挿入する ;
  if ループに含まれる R で上方に開かれた使用が存在する then
    ループの直前に "t=R" を挿入する ;
end ScalarReplaceCyclicPartition
```

図 12.10: ScalarReplaceCyclicPartition

12.7 InsertMemoryRefs

```
procedure InsertMemoryRefs(\delta)

  // \delta は、不整合依存である
  if \delta がループ可変な名前分割とループ不変な名前分割をリンクしている then
  begin // この問題を解消するためにインデックス集合分割をつかう
    ループを３つの部分に分割する：
      a) \delta 中のループ不変参照を含む繰り返し（これ自体は入れない）までに
         ある繰り返し
      b) この参照を含む繰り返し
      c) この参照を含む繰り返し以降の繰り返し
    この３つの部分にスカラー置換を適用する；
  end
  else if \delta がフロー依存である then begin
    ソース側の一時変数の対応する参照への格納を挿入する；
    R をループの中でソースと関連するシンクの名前分割内で最初にあらわれる
    参照とする；
    R から R 用の一時変数へのロードをまだそのようなものがなければ、
    その参照の前に挿入する；
  end
  else if データ依存が入力依存である then begin
    R2 をソースにある参照の後に出現するシンクの最初の参照とする；
    R1 を R2 の前に出現するソースの最後の参照とする；
    R1 の一時変数のストアを挿入する（冗長でなければ）；
    R2 から R2 の一時変数へのロードを R2 の前に挿入する（冗長でなければ）；
  end
end InsertmemoryRefs
```

図 12.11: InsertMemoryRefs

12.8 ループ展開

ループをまたがるデータ依存関係が存在するとき、ループのステップを増し、ループ本体をコピーすることでループの内側に依存関係をもってくることにより、より多くの配列要素の参照をスカラー変数に置き換えることができるようになる。この手法をループ展開 (loop unrolling) という。

12.8.1 ループ展開の例

ループ展開を図 12.12 のコードに適用すると、図 12.13 のコードのようになる。

```
t1=B(0)
t2=B(1)
DO I=1, N
  t3=B(I+1)
  A(I)=t1+t3
  t1=t2
  t2=t3
ENDDO
```

図 12.12: ループ展開適用前のコード

```
t1=B(0)
t2=B(1)
mN3=MOD(N,3)
// Preloop:
DO I=1, mN3
  t3=B(I+1)
  A(I)=t1+t3
  t1=t2
  t2=t3
ENDDO
// Main loop:
DO I=mN3+1, N, 3
  t3=B(I+1)
  A(I)=t1+t3
  t1=B(I+2)
  A(I+1)=t2+t1
  t2=B(I+3)
  A(I+2)=t3+t2
ENDDO
```

図 12.13: ループ展開適用後のコード

12.8.2 ループ展開のアルゴリズム

ループ展開は、以下のアルゴリズムにしたがっておこなわれる。

```
procedure UnrollLoop(K)
  // K は、ループ展開因子
  ループを preloop と mainloop に分ける：
    preloop:
      DO I=1, MOD(N,K)
     main loop:
      DO I=MOD(N,K)+1, N;
  main loop からレジスタ間コピーを除去する；
  main loop をループ本体 K 個のコピーに展開し、ループを K ステッ
プにする：
      DO I=MOD(N,K)+1, N, K;
  // ここで、p 番目のコピーにおけるインデックス I を I+p で置き換
える
  ループ本体の q 番目のコピーの中の一意的に生成された変数 t_i へ
の
  各レファレンスを t_MOD(i+q-1,k)+1 で置き換える。
  ただし、k は、t_* のインデックスの最大値である。
end UnrollLoop
```

図 12.14: ループ展開アルゴリズム

ここで、一意的に生成された変数というのは、スカラー置換のために、レファレンスをレジスタで変数で置き換えるときに使用される変数のことである。

付 録A　SSA Options

"-coins:lir-opt=..." can also be used instead of "-coins:ssa-opt=...".
(When "lir-opt"and"ssa-opt" are both written, "lir-opt" is only available.)

A.1　-coins:ssa-opt=xxx/yyy/.../zzz

Use SSA pass. This is necessary for using the SSA module. There are several optimizations in this module. To invoke the optimization, you should specify the optimizers with this option. Specified optimizers are invoked from left to right. First, as 'xxx' you MUST specify to which kind of SSA form you like to translate, such as minimal, semi-pruned or pruned. And then, as 'yyy' you can specify the optimizers which the SSA module invokes. You can specify the same optimizer two times, three times, and so on. Only optimizations that you specify are performed in that order. Finally, as 'zzz' you MUST specify how to back translate from SSA form.

The options are defined as follows:

Translation from normal form LIR to SSA form LIR :
(You MUST specify one of them at the beginning of this SSA option)

mini : Translation to Minimal SSA form

semi : Translation to Semi-Pruned SSA form

prun : Translation to Pruned SSA form (recommended for optimization)

Back Translation from SSA form LIR to normal form LIR :
(You MUST specify one of them at the end of this SSA option)

brig : Back translation using Briggs's Method

srd1 : Back translation using Sreedhar's Method I

srd2 : Back translation using Sreedhar's Method II

srd3 : Back translation using Sreedhar's Method III (recommentded for optimization)

(Options for coalescing are explained later)

Optimization :

cpyp : Copy Propagation

cstp : Constant Folding and Propagation with Conditional Branches

dce : Dead Code Elimination

cse : Common Subexpression Elimination

preqp : Global Value Numbering and Partial Redundancy Elimination with Efficient Question Propagation

hli : Hoisting Loop-invariant Code

osr : Operator Strength Reduction related to Induction Variables and Linear Function Test Replacement

ssag : Making SSA graph

divex : Divide Expression to Three-Address Code (the right-hand side of assignment will have only one operator)

gra : Global Reassociation

ebe : Empty Block Elimination

rpe : Redundant ϕ-function Elimination

cbb : Concatenate Basic Blocks

esplt : Split Critical Edge

lir2c : Make C program from LIR

lcm : Lazy Code Motion

pdeqp : Demand-Driven Partial Dead Code Elimination

ddpde : Demand-Driven Partial Dead Code Elimination

expde : Exhaustive PDE

glia : Global Load Instruction Aggregation

eqp : Effective Demand Driven Partial Redundancy Elimination

presr : Scalar Replacement By Effective Demand Driven Partial Redundancy Elimination

Line numbering :

stlin : Put a line number to each LIR

inslin : Insert LIRs representing line numbers

rmlin : Remove LIRs representing line numbers

shlin : Show LIRs to the standard output

Counting instructions :

cntbb : Count executed LIR instructions for each basic block

notes:

(a) It is not allowed to specify multiple uses of cntbb.

(b) The use of cntbb requires fopen and fprintf functions, so this option is not allowed in the x86_64-mac architecture.

(c) Cntbb doesn't count the following LIR instructions: PHI instruction, PROLOGUE instruction, EPILOGUE instruction, and LIR instructions used for counting instructions.

(d) Results of "cntbb" option are written down on files `<Filename>.<Functionname>.cnt` in the working directory where `<Filename>` and `<Functionname>` are the file name and function name of the source program respectively.

(e) Cntbb uses a working directory which is indicated with the COINS option "tmpdir". If the option is not specified "/tmp" is used as the working directory.

(f) The counting instructions option counts the number of executed LIR instructions with compilation twice and execution once. At the first compilation, the informations which are referenced at the second compilation are written into the file file_list.dat in the working directory. So before the first compilation the file should be removed, if it exists.

(g) See 'Optimization in Static Single Assignment Form - External Specification' in Japanese for details.

Example :

If you specify the option

```
-coins:ssa-opt=prun/cstp/cse/srd3
```

the SSA module performs the following in that order:

1. make puruned SSA form
2. invoke constant folding and propagation with conditional branches
3. invoke common subexpression elimination
4. back translate using Sreedhar's Method III

A.2 -coins:ssa-no-change-loop

Before the optimizations, the SSA module changes the structure of the loops as follows, by default. This is for making effective loop optimization.

1. merge the several loops that have the same header block
2. insert the preheader

3. change the loop structure from 'while' type to 'do-while' type (precisely 'if-do-while'). The 'while' type is a loop such that the header and exit block of the loop are the same block.

The above is performed by default. If you DO NOT want to do that, specify this option.

A.3 -coins:ssa-no-copy-folding

During the translation to the SSA form, the SSA module removes and propagates the copy assignment statements such as 'x←y', by default.

If you DO NOT want to do that, specify this option.

A.4 -coins:ssa-no-redundant-phi-elimination

The SSA module eliminates redundant ϕ-functions after the translation to the SSA form, by default. A ϕ-function is redundant if:

1. the names of the target and the arguments of the ϕ-function are all the same as follows:

   $$x1 \leftarrow \phi(x1, x1, x1)$$

2. the names of the arguments of the ϕ-function are all the same as follows:

   $$x1 \leftarrow \phi(y1, y1, y1)$$

3. there are also other cases as follows:

   $$x1 \leftarrow \phi(y1, y1, x1) \quad \text{or} \quad x1 \leftarrow \phi(y1, y1, \perp)$$

In the first case, the SSA module just eliminates the ϕ-function. In the second case, the SSA module eliminates the ϕ-function and replaces uses of 'x1' by 'y1' in the statements which are evaluated after the ϕ-function. For further details, see [reference 1].

If you DO NOT want to do that, specify this option.

A.5 -coins:ssa-no-sreedhar-coalescing

During the back translation from SSA form by Sreedhar's method, the SSA module coalesces copy-related variables in SSA form, by default. This coalescing is proposed by Sreedhar and is called the SSA-based coalescing. This coalescing module is usually unified with Sreedhar's algorithm for back translation from SSA form. But for researchers' convenience, the SSA module can avoid it.

If you DO NOT want to do SSA based coalescing, specify this option.

A.6 -coins:ssa-with-chaitin-coalescing

Perform coalescing proposed by Chaitin after the back translation to normal form. This coalesces copy-related variables whose live ranges do not interfere each other. In general, after the back translation from SSA form, there may be some copy assignment statements in the program. Some copy assignment statements only change the names of variables, that is, they are useless. Coalescing these variables eliminates the useless copy assignment statements. This optimization is done in normal form LIR after the back translation from SSA form.

If you WANT to do that, specify this option. (This coalescing can be specified after any back translation method. But it may have no effect after the back translation by Sreedhar's Method III since that method does not insert copy assignment statements which can be coalesced by Chaitin's coalescing.)

A.7 -coins:ssa-no-memory-analysis

When Common Subexpression Elimination and/or Global Value Numbering and Partial Redundancy Elimination with Efficient Question Propagation are specified, the SSA module makes a simple alias analysis of memory accesses, by treating the whole memory as a single entity. (cf. section 8 of [reference 1])

If you DO NOT want to do that, specify this option.

A.8 -coins:ssa-no-replace-by-exp

Just before the back translation from SSA form, the SSA module finds the local variables, which are not "live out" from the current basic block and are used only once in the current basic block, and replaces the variables by the expressions which define the variables. (cf. section 5.4.6 "preprocessing for temporary variables" of [reference 1])

If you DO NOT want to do that, specify this option.

A.9 -coins:trace=SSA.xxxx

To output the trace information of this SSA module for debugging, specify the trace level as follows:

1 : Output only the message that the SSA module is invoked

100 : Output the agenda of the SSA module

1500 : Output two kinds of information:

(a) The inserted ϕ-functions when the SSA module translates normal LIR into SSA form.

(b) The inserted copy assignment statements when the SSA module back translates SSA form into normal LIR.

2000 : Output general debug information of all optimizers in the SSA module

10000 : Output much information about Sreedhar's Method III

The trace information includes the levels less than or equal to what you specified. If you specify

```
trace=SSA.1500
```

then the SSA module outputs information related to the level '1', '100' and '1500'.

A.10 -coins:ssa-opt=.../dump/...

For compiler developers, the SSA module provides the option 'dump' for debugging. This option should be specified within 'ssa-opt'. When this option is specified, the SSA module outputs the current LIR into the standard output.

For example, if the option is specified as follows:

```
-coins:ssa-opt=prun/dump/srd3/dump
```

the SSA module outputs the LIR

(1) after translation into the pruned SSA form, and

(2) after back translation from SSA form.

A.11 -coins:regalloc=oca

The register allocation algorithm, "Optimistic Coalescing Allocation" can be selected by this option. When this option is not specified, the default register allocation algorithm is used.

A.12 -coins:tmpdir=xxxx

This option specifies xxxx as the working directory. When this option is not specified, '/tmp' is used for the working directory. The working directory is used for the ssa option "cntbb".

A.13 References

[1] 'Optimization in Static Single Assignment Form - External Specification'

which is available on the web page 'http://www.coins-project.org/'.

参考文献

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 1–11, January 1988.

[3] Andrew W. Appel. *Modern Compiler Implementation in Java second edition*. Cambridge University Press, 2002.

[4] P. Briggs. *Register allocation via graph coloring*. PhD thesis, Rice University, 2004.

[5] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 159–170, June 1994.

[6] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software – Practice and Experience*, Vol. 28, No. 8, pp. 859–881, July 1998.

[7] Preston Briggs, Tim Harvey, and Taylor Simpson. Static single assignment construction, version 1.0, January 1996. ftp://ftp.cs.rice.edu/public/compilers/ai/SSA.ps.

[8] G. J. Chaitin. Register allocation and spilling via graph coloring. In *SIGPLAN Notices 17(6), Proc. of the ACM SIGPLAN '82 Symp. on Compiler Construction*, pp. 98–105, 1982.

[9] Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. Incremental computation of static single assignment form. In *Proceedings of the 1996 International Conference on Compiler Construction*, pp. 223–237, April 1996.

[10] COINS project. http://www.coins-project.org/.

[11] Keith D. Cooper, L. Taylor Simpson, and Christpher A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, Vol. 23, No. 5, pp. 603–625, September 2001.

[12] 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 Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 461–486, October 1991.

[13] L. George and A.W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 1996.

[14] J. Park and S.M Moon. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems*, 2004.

[15] Berry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 12–27, January 1988.

[16] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. *SAS'99 LNCS 1694*, pp. 194–210, 1999.

[17] Munehiro Takimoto and Kenichi Harada. Efficient question propagation.

[18] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 181–210, April 1991.

[19] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.

[20] 今橋孝典, 伊藤陽, 佐々政孝. 静的単一代入形式上で通常形式部分冗長除去を実現する汎用的手法. 情報処理学会論文誌：プログラミング, Vol. 49, No. 1, pp. 84–95, January 2008.

[21] 滝本宗宏. 要求駆動型部分無用コード除去, 2008.

[22] 小濱真樹. SSA 正規化アルゴリズムの比較と評価. 東京工業大学 数理・計算科学専攻 修士論文, 2004.

[23] 副島祐介, 佐々政孝. レジスタ割り付けにおける optimistic register coalescing の実装と評価. 日本ソフトウェア科学会第２５回大会論文集 4A-1, September 2008.

[24] 澄川靖信, 滝本宗宏. 効率的な要求駆動型部分冗長除去, 2012.