中間コードを表すプログラム依存グラフの操作的意味

An Operational Semantics of Program Dependence Graphs for Intermediate Code

伊藤 宗平 Souhei ITO　萩原 茂樹 Shigeki HAGIHARA　米崎 直樹 Naoki YONEZAKI
東京工業大学大学院情報理工学研究科計算工学専攻
Dept. of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
{ito, hagihara, yonezaki}@fmx.cs.titech.ac.jp

プログラム依存グラフ (PDG) はプログラム中の文の間に存在する依存関係を表す有向グラフである. 本論文では中間コードのように自由な goto 文が出現するようなプログラムに対する PDG の操作的意味を定義し, PDG の意味がそれが表すプログラムの意味と一致することを示す. また, PDG 上での最適化コンパイラによる最適化がプログラムの意味を保存することを示す.

1 はじめに

プログラム依存グラフ (program dependence graph: 以下 PDG) [4] はプログラムの中間表現の一つで, 命令をノードとし, プログラム中に存在するデータ依存関係や制御依存関係をエッジで表した有向グラフである. 逐次プログラムの表現と違って, 人工的な文の順序を排除し, 依存関係だけを表しているため命令間の並列性を明らかにする.

データの依存関係や制御の依存関係はコンパイラにおけるコード最適化の際に解析され, どの箇所にどの最適化が適用可能かを検出するのに用いられる. コード最適化は本質的にそういった依存関係の情報のみによるため, PDG はコード最適化に適したプログラムの中間表現である. また, 並列化コンパイラにおいても有用である. 実際, PDG 上での最適化アルゴリズムがいくつか提案されている [3, 4].

PDG を用いることにより, プログラムの最適化を単純化することができる. これは, 多くの最適化が PDG を単純に走査するだけで実行できるからである. さらにプログラムが変化することによって依存関係が変化しても, それを計算しなおす必要はない. なぜなら最適化自体が依存関係の変化として行われているからである. 従ってコントロールフローグラフ (control flow graph: 以下 CFG) を用いた最適化よりも効率よく最適化を行うことができる.

PDG はプログラム中のデータ依存関係や制御依存関係を抽出したものであるので, それ自体がプログラムの意味を表していると見なすことができる. しかし, PDG 自体もプログラムの中間表現であるため, 形式的意味を持つはずである. では, プログラムの意味とその PDG の意味は本当に一致しているのだろうか. この点について, PDG の形式的意味を定義した二つの研究がある [7, 2]. これらは構造化プログラム言語に対する PDG の書き換えによる意味 [7] と表示的意味 [2] を与え, プログラムとその PDG の意味が一致することを示した ([2] のほうは厳密に言えば, プログラムの意味を PDG の意味が “支配する”, すなわち停止しないプログラムに対しても PDG の意味では停止して結果を返すということを示している). これらの論文ではプログラムの最適化の正当性を示すためにこれらの PDG の形式的意味が利用できる, ということを述べているが, プログラム言語が構造的な構文要素 (**if** $e\ S_t\ S_f$, **while** $e\ S$) しか持たないため, その利用は限られている.

そこで我々は構造化でないプログラムに対する PDG においても当てはまるような操作的意味を定義する. これによってプログラムの最適化の正当性を PDG 上で証明することができる. これは CFG 上で行う正当性の証明 [5, 6] よりも単純であり, また並列化などの最適化の正当性の証明にも利用可能となることが期待される. さらに, PDG を実際のコンパイラのコード最適化に利用するためには PDG から別の中間表現または目的コードへ変換する必要があるが, そのためにはその変換がプログラムの意味を保存していることを証明しなければならない. 本論文が提案する操作的意味によりそれが可能となる.

本論文の構成は以下の通りである. 2 節では本論文で考慮するプログラムの表現である CFG とその操

作的意味を定義する. 3 節では本論文で扱う PDG を定義する. 4 節では PDG の操作的意味を定義する. そして 5 節では CFG とそれに対応する PDG は同じ意味を持つことを証明する. 6 節では, コード最適化手法の正当性の証明に PDG の操作的意味が有用であることを示すために, 一例として, よく知られた最適化手法である複写伝搬の正当性を PDG の操作的意味を用いて証明する. 最後に 7 節ではまとめと今後の方向性を述べる.

2 コントロールフローグラフ

この節ではコントロールフローグラフ (CFG) とその操作的意味を定義する.

定義 2.1 (Control flow graph) CFG はノードの集合 N とエッジの集合 E のペア (N, E) である. CFG のノードは文であり, 実行の流れをノード間のエッジで表す. 文のタイプとしては, 代入文 $(x := e)$ と条件文 ($\texttt{if}\ e$) と return 文 ($\texttt{ret}\ x$) だけを考える.

CFG は入口ノードと出口ノードをそれぞれ唯一つ持つ. 入口ノードは predecessor を持たないノードであり, 出口ノードは successor を持たないノードである. CFG の全てのノードは入口ノードから到達可能でなければならない. また, 出口ノードは必ず return 文でなければならず, 他に return 文は現れないとする.

各文の successor の数は, 代入文は一つ, 条件文は二つである. 条件文からのエッジは **T** エッジと **F** エッジがある.

図 1 に CFG の例を挙げる.

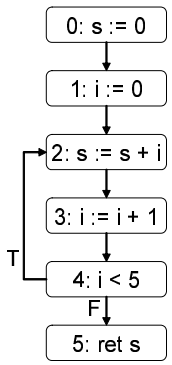

図 1: CFG の例

これから CFG の操作的意味を定義する.

定義 2.2 (CFG の操作的意味) 変数の集合を var, 値の集合を val, 式の集合を exp とする. val は真理値の集合 $\{\mathbf{T}, \mathbf{F}\}$ を含む以外は特定しないこととする. また, 式に対する意味は次の関数 $\mathcal{E}$ によって与えられるとする.

$$\mathcal{E} : exp \longrightarrow store \longrightarrow val$$

ここで, store は

$$store = var \longrightarrow val$$

と定義される.

CFG $G = (N, E)$ の **run** は store の列 $\sigma_0 \xrightarrow{n_0} \sigma_1 \xrightarrow{n_1} \ldots$ である. n_0 は入口ノードであり, σ_0 は与えられる初期ストアである. $n_0 n_1 \ldots$ は G における入口ノードからのパスとなる. $n_i \neq (\texttt{if}\ e)$ ならば n_{i+1} は n_i の G における唯一つの successor である. $n_i = (\texttt{if}\ e)$ ならば, n_{i+1} は $\mathcal{E}(e)\sigma_i$ によって決定される. G の構成より, n_i は 2 つの successor を G 中に持つ. $(n_i, p)_{\mathbf{T}} \in E, (n_i, q)_{\mathbf{F}} \in E$ とすると,

$$n_{i+1} = \begin{cases} p & \text{if } \mathcal{E}(e)\sigma_i = \mathbf{T} \\ q & \text{if } \mathcal{E}(e)\sigma_i = \mathbf{F} \end{cases}$$

となる. また,σ_{i+1} は次のように決定される.

$$\sigma_{i+1} = \begin{cases} \sigma_i[x \mapsto \mathcal{E}(e)\sigma_i] & \text{if } n_i = (x := e) \\ \sigma_i & \text{otherwise} \end{cases}$$

CFG の run は有限か無限である. 有限なら, 最終状態を s とすると, $\cdots \xrightarrow{n} s$ のとき, n は successor を持たない. すなわち, n は $\texttt{ret}$ 文である.

定理 2.1 CFG の run は決定的である. すなわち, 任意の CFG G に対し, 任意の σ_0 に対し, 唯一つの G の run が存在する.

3 プログラム依存グラフ

本論文で定義する PDG は CFG と同様に文をノードとして持ち, 4 種類の依存関係をエッジ情報として含む. 依存関係には制御依存関係, ループ独立データ依存関係, ループ繰越データ依存関係, 定義順序関係の 4 つがある. 以下これらの依存関係を定義する.

3.1 制御依存関係

制御依存関係を定義する前に, CFG に対しあるノードがあるノードを**後支配**するという概念を定義する.

制御依存関係を定義する際に, CFG には出口ノードの successor として特殊なノード *entry*, *exit* を追加し, *entry* から入口ノードへの **T**-エッジと *exit* ノードへの **F**-エッジがあるとする (図 2).

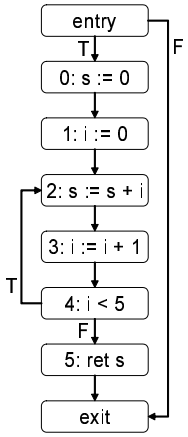

図 2: *exit* を追加した CFG

定義 3.1 (後支配) ある CFG G 中の 2 つのノード s と t について, s が t を**後支配**するとは, G において, t から G の *exit* に達するどのパスも必ず s を通る場合をいう. また, s が t を後支配し, $s \neq t$ であるとき, s は t を**厳密に後支配**するという.

定義 3.2 (制御依存関係) 2 つのノード s, t について, 次の 2 つが成り立つとき, t は s に**制御依存**するという.

1. s から t への空でないパスがあり, t はそのパス上の s より後の全てのノードを後支配する.
2. t は s を厳密に後支配はしない.

上の定義は一般的な後支配と制御依存関係の定義である [4]. 本論文ではそれらの定義を少し変更する.

まず, 後支配の概念を強めた**強い後支配**という概念を定義する.

定義 3.3 (強い後支配) ある CFG G 中の 2 つのノード s と t について, s が t を**強く後支配**するとは, G において, t からの全てのパス上に s がある場合を言う. また, s が t を強く後支配し, $s \neq t$ であるとき, s は t を**厳密に強く後支配**するという.

後支配と強い後支配の概念は CFG にループがない場合は一致する. 図 1 においては, 4 は 5 に後支配されるが, 強く後支配されない (4 から 5 を通らずに回り続けるパスが存在する).

強い後支配の定義を用いて次の**弱い制御依存関係**を定義する.

定義 3.4 (弱い制御依存関係) 2 つのノード s, t について, 次の 2 つが成り立つとき, t は s に**弱く制御依存**するという.

1. s から t への空でないパスがあり, t はそのパス上の s より後の全てのノードを強く後支配する.
2. t は s を厳密に強く後支配しない.

強い後支配と弱い制御依存関係の定義は [8] において初めて用いられている.

通常の後支配と強い後支配, また通常の制御依存関係と弱い制御依存関係は CFG にループがなければ一致する. 図 1 においては, 4 と 5 の間には通常の制御依存関係は成り立たない. なぜなら, 4 は 5 に厳密に後支配されるからである. しかし, 弱い制御依存関係は成り立つ. なぜなら, 4 は 5 に厳密に強く後支配されないからである.

この定義の意図は, A と B が制御依存の関係にある場合, B が実行されるかどうかは A の判定結果に依存する, というものである. 通常の制御依存関係も本来はこれを想定していると思われるが, 厳密には一致しない. 図 1 においてそれを説明する. 図 1 においては, 通常の制御依存関係においては 5 は 4 に制御依存しない. しかし, 5 が実行されるためには 4 の判定結果が偽でなければならない. 従って, 5 が実行されるかどうかは 4 の判定結果に依存している. 弱い制御依存関係では確かに 5 は 4 に制御依存する.

本論文では, これ以降定義 3.4 の弱い制御依存を単に制御依存と呼ぶ. また, s から t へ制御依存関係があることを $CD(s,t)$ と書くことにする.

制御依存関係を true 制御依存と false 制御依存に分類する. s から t へ制御依存関係があるとする. このとき, $(s,u)_{\mathbf{T}} \in E, (s,v)_{\mathbf{F}} \in E$ とする. u が t に後支配される場合, s から t へ **true 制御依存**があるといい, v が t に後支配される場合は s から t へ s から t へ **false 制御依存**があるという.

図 3 に図 1 に対する制御依存関係を表した制御依存グラフ (control dependence graph: 以下 CDG) を示す. 図 3 では, (4,5) 以外は全て true 制御依存関係である. *exit* ノードは便宜上追加したものであるので, CDG には含めない.

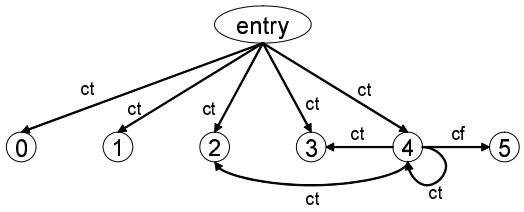

図 3: CDG

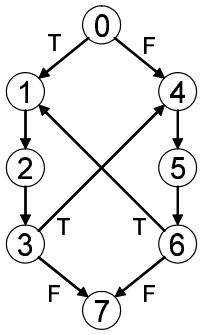

図 4: ループの例

3.2 データ依存関係

データ依存関係はループ独立データ依存関係 (loop independent data dependency) とループ繰越データ依存関係 (loop carried data dependency) がある. これを定義するために CFG 中のループとは何かを定義する必要がある. 可約 (reducible) フローグラフ (ループの入口は一つしかないグラフ)[1] においては, バックエッジの定義を用いた**自然なループ**が定義できた. しかし, 本論文で扱う CFG は可約であるとは限らない. したがって, 可約でないグラフに対するループを定義しなければならない. 以下でそれを定義するが, それらは [10] による.

定義 3.5 (Loop body) $G = (N, E)$ をグラフとする. G 中の loop body は部分グラフ $G' = (N', E')$ である. ここで, N' は, 一つ以上の要素を持つ強連結成分である. ただし, ノードが一つだけの強連結成分は, 自分自身へのエッジを持たなければならない. $E' = E \cap N' \times N'$ である.

定義 3.6 (Loop entry nodes) G をグラフとし, G' を G 中のある loop body とする. G' の loop entry nodes とは,G' 中のノードであり, 少なくとも一つの G' 外からのエッジを持つノードである.

定義 3.7 (Loop closing edges) G' を loop body とする. G' の loop closing edges は, G' 中のエッジであり, そのターゲットが loop entry nodes のどれかであるようなエッジである.

図 4 の例においては, loop body は $\{1, 2, 3, 4, 5, 6\}$ からなる強連結成分であり, loop entry nodes は 1 と 4 で, loop closing edges は $(3, 4)$ と $(6, 1)$ である.

このようにループを厳密に定義することにより, ループ独立データ依存関係とループ繰越しデータ依存関係を定義することができる.

定義 3.8 (データ依存関係) CFG を G とする. s から t にデータ依存関係があるとは, ある変数 w が存在し, s における w の定義が, w を使用している t に到達する, すなわち間に w の定義がない s から t へのパスが存在する場合を言う.

s と t の間にループ繰越データ依存関係があるとは, ある変数 w が存在して, s から t に w においてデータ依存関係があり, s と t が同一の loop body の中にあり, s から t への w の定義が到達するあるパスが存在し, そのパスが loop closing edge を通る場合を言う.

s と t の間にループ独立データ依存関係があるとは, ある変数 w が存在して, s から t に w においてデータ依存関係があり, s と t が同一の loop body の中にない, あるいは同一の loop body の中にあっても, s から t へ w が到達するあるパスが存在し, それが loop closing edge を通らない場合を言う.

s から t へのループ独立依存関係があるということを $LIDD(s, t)$, ループ繰越依存関係があるということを $LCDD(s, t)$ と表す.

3.3 定義順序関係

最後に定義順序関係について定義する.

定義 3.9 (定義順序関係) フローグラフを G とする. s から t に定義順序関係があるとは, ある変数 w が存在して s と t が共に w を定義し, あるノード u が存在して u は w に関して s とも t ともデータ依存関係があり, s から t へ loop closing edge を通らないで到達可能である場合を言う.

s から t に定義順序関係があることを $DefOrd(s, t)$ と表す.

3.4 プログラム依存グラフ

以上の定義から, プログラム依存グラフを定義することができる.

定義 3.10 (Program dependence graph) フローグラフ $G = (N, E)$ に対するプログラム依存グラフは有効グラフ $(N \cup \{entry\}, C, F, L, D)$ である. ここで,N はノードの集合, C, F, L, D はそれぞれエッジの集合である. 以下 $s, t \in N$ とする.

$$
\begin{aligned}
C &= \{(s,t) \mid CD(s,t)\} \\
F &= \{(s,t) \mid LIDD(s,t)\} \\
L &= \{(s,t) \mid LCDD(s,t) \wedge \neg LIDD(s,t)\} \\
D &= \{(s,t) \mid DefOrd(s,t)\}
\end{aligned}
$$

L の定義により, もし $LIDD(s,t)$ かつ $LCDD(s,t)$ であった場合は $LIDD(s,t)$ のみをとる.

これは, $LCDD(s,t)$ である場合には t の方が s より先に実行されるべきであるという依存関係を表すためである. $LIDD(s,t)$ かつ $LCDD(s,t)$ であるのは図 5 のような場合である. この場合には s から t へ loop closing edge を通らずに到達できるので, s のは t よりも先に実行されるべきである. したがって $LIDD(s,t)$ のみをエッジ情報として表せばよい.

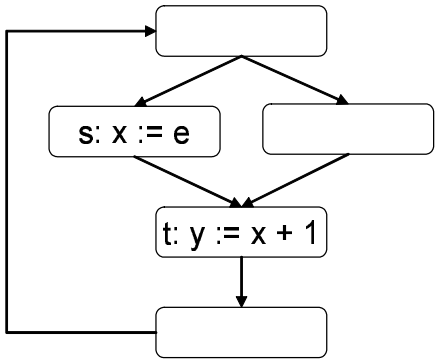

図 5: $LIDD(s,t)$ かつ $LCDD(s,t)$

以降では,C の要素を $c(s,t)$, F の要素を $f(s,t)$, L の要素を $l(s,t)$, D の要素を $d(s,t)$ と表すことがある. また, C の要素のうち true 制御依存関係と false 制御依存関係を区別して $ct(s,t)$, $cf(s,t)$ と表すこともある. また, F, L の要素を関連する変数を明示して $f_x(s,t)$, $l_x(s,t)$ と表すこともある.

図 6 に図 1 の CFG に対する PDG を示す. 煩雑さを避けるため *entry* は省略してある.

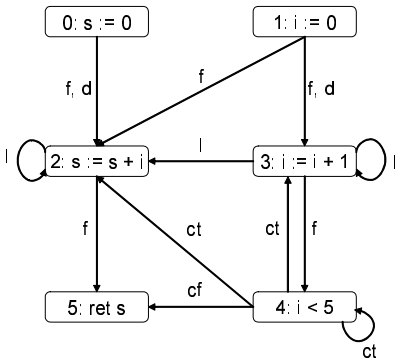

図 6: PDG の例

4 PDG の操作的意味

この節では PDG の操作的意味を定義する. 基本となるアイデアは, PDG の run における状態を, 次のようにすることである.

$$state = avail \times econf$$

ここで, PDG を $G = (N, C, F, L, D)$ とすると, *avail* と *econf* は次のように定義される.

$$
\begin{aligned}
avail &= N \times var \longrightarrow val \\
econf &= C \oplus F \oplus L \oplus D \longrightarrow \{unchk, act, inact\}
\end{aligned}
$$

ここで, $\oplus$ は直和である.

avail は, ノードと変数を取って値を返す関数である. store と違うところは, 各ノードごとにそのノードで使用する変数の値を持っている点である. CFG では変数と値の対応を与えるために store を用いたが, PDG では命令の実行の順番に自由度があるため, store を用いるよりも, 各ノードごとに変数と値の対応を持っていてそれはデータの依存関係の predecessor から与えられる, とするほうが操作的意味を定義しやすく, PDG の意図にも合っている.

econf は, 実行における PDG のエッジの状態を表す関数であるが, これはどのノードが実行可能かを示すために用いられる. 例えば, あるノード n が `if` 文であり, その判定結果が **T** ならば, n から出る ct-エッジを活性化し (act にする), cf-エッジを不活性化する ($inact$ にする). n が実行可能であるためには, 少なくとも一つの n に入ってくる C-エッジが act でなければならない. これらの情報を表すために, 関数 *econf* を用いる.

以下では, これらの概念を用いて PDG の操作的意味がどのように定義されるかを説明していく. そのた

めに, まずは PDG における **looping edge** の概念を定義する.

定義 4.1 PDG $G = (N, C, F, L, D)$ の制御依存サブグラフを $G_C = (N, C)$ とする. G_C 内のある loop body を R とすると, R の loop closing edge を $ct(p, q)$(もしくは $cf(p, q)$) とする. このとき, p からの ct-エッジ全て (もしくは cf- エッジ全て) を, G における looping edge という.

Loop closing edge は一般のグラフ (CFG でなくてもよい) に対して当てはまる概念であることに注意してほしい. 図 3 においては, loop body は 4 だけからなり, その loop closing edge は $ct(4, 4)$ である. 従って looping edge は, $ct(4, 2), ct(4, 3), ct(4, 4)$ となる.

定義 4.2 $G = (N, C, F, L, D)$ を PDG とする. このとき, C から looping edge を除いたものを $\widehat{C}$ とする.

定義 4.3 PDG を $G = (N, C, F, L, D)$ とする. また, $p \in N$ とする. このとき, $G(p) = (N', C', F', L', D')$ は次のように定義される.

$$
\begin{aligned}
N' &= \{n \mid c(p, q) \in C \wedge n \text{ is } \widehat{C}\text{-reachable from } q\} \\
C' &= C \cap N' \times N' \\
F' &= F \cap N' \times N' \\
L' &= L \cap N' \times N' \\
D' &= D \cap N' \times N'
\end{aligned}
$$

$\widehat{C}$-reachable とは, $\widehat{C}$ エッジだけで到達可能であることを指す.

$G(p)$ は, p の C-succsessor から looping edge を通らずに到達できるノードから構成される部分グラフである. ただし, $c(p, q)$ は looping edge でも良いことに注意.

また, $G_{\mathbf{T}}(p) = (N', C', F', L', D')$ は次のように定義される.

$$
\begin{aligned}
N' &= \{n \mid ct(p, q) \in C \wedge n \text{ is } \widehat{C}\text{-reachable from } q\} \\
C' &= C \cap N' \times N' \\
F' &= F \cap N' \times N' \\
L' &= L \cap N' \times N' \\
D' &= D \cap N' \times N'
\end{aligned}
$$

同様に $G_{\mathbf{F}}(p)$ も定義できる.

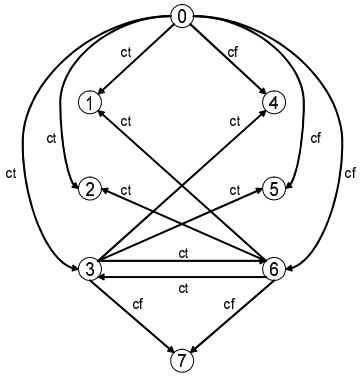

図 7: 図 4 の CFG に対する CDG

図 4 に対する CDG(図 7) で例を示す.

図 7 においては, looping edge は, $ct(3, 4)$, $ct(3, 5)$, $ct(3, 6)$, $ct(6, 1)$, $ct(6, 2)$, $ct(6, 3)$ である. 下に, それぞれのノードに対する部分グラフのノード集合を示す.

$$
\begin{aligned}
G(0) &\quad \{1, 2, 3, 4, 5, 6, 7\} \\
G_{\mathbf{T}}(0) &\quad \{1, 2, 3, 7\} \\
G_{\mathbf{F}}(0) &\quad \{4, 5, 6, 7\} \\
G(3) &\quad \{4, 5, 6, 7\} \\
G_{\mathbf{T}}(3) &\quad \{4, 5, 6\} \\
G_{\mathbf{F}}(3) &\quad \{7\} \\
G(6) &\quad \{1, 2, 3, 7\} \\
G_{\mathbf{T}}(6) &\quad \{1, 2, 3\} \\
G_{\mathbf{F}}(6) &\quad \{7\}
\end{aligned}
$$

これらの言葉を用いて PDG の操作的意味を定義する.

定義 4.4 (PDG の操作的意味) var, exp, val は定義 2.2 と同じものとする. $avail$, $econf$ は以下のように定義される.

$$
\begin{aligned}
avail &= N \times var \longrightarrow val \\
econf &= C \oplus F \oplus L \oplus D \longrightarrow \{unchk, act, inact\}
\end{aligned}
$$

ここで $\oplus$ は直和である.

また, 式に対する意味は次の関数 $\mathcal{E}$ によって与えられるとする.

$$
\mathcal{E} : N \times exp \longrightarrow av \longrightarrow val
$$

式に対する意味関数は定義 2.2 と同じ記号を用いているが, 型の違いから容易に区別できる.

PDG の run における state を次のように定義する.

$$state = avail \times econf$$

PDG $G = (N, C, F, L, D)$ の **run** は state の列 $s_0 \stackrel{n_0}{\to} s_1 \stackrel{n_1}{\to} \ldots$ である. $s_0 = (av_0, ec_0)$ とすると, av_0 は与えられる初期 $avail$ であり, ec_0 は次の条件を満たす初期 $econf$ である.

$$\forall (entry, q) \in C.ec_0((entry, q)) = act$$
$$\forall (p, q) \in C \oplus F \oplus L \oplus D.ec_0((p, q)) = unchk$$

n_i, s_i がどのように決まるかを定義するために以下の述語を定義する. 下の定義において, $s = (av, ec)$ とする.

$$\begin{aligned}
condC(s,n) &\stackrel{\text{def}}{=} \exists c(p,n) \in C.ec(c(p,n)) = act \\
&\quad \wedge \forall q \in G(n).q \neq n \Rightarrow \forall c(r,q) \in C. \\
&\qquad ec(c(r,q)) \neq act \\
condF(s,n) &\stackrel{\text{def}}{=} \forall f(p,n) \in F.ec(f(p,n)) \neq unchk \\
condL(s,n) &\stackrel{\text{def}}{=} \forall l(n,p) \in L. \\
&\quad n \neq p \Rightarrow ec(l(n,p)) \neq unchk \\
condD(s,n) &\stackrel{\text{def}}{=} \forall d(p,n) \in D.ec(d(p,n)) \neq unchk
\end{aligned}$$

$condCFLD(s,n)$ で, $condC(s,n) \wedge condF(s,n) \wedge condL(s,n) \wedge condD(s,n)$ を表すとする.

関数 $Next : state \longrightarrow 2^N$ を次のように定義する.

$$Next(s) = \{n \mid condCFLD(s,n)\}$$

$Next(s)$ は, 状態 s において次に実行可能なノードの集合を表す.

関数 $udav : N \times avail \longrightarrow avail$ を次のように定義する.

$$udav(n, av) = \begin{cases} av[(p,x) \mapsto \mathcal{E}(n,e)av : (n,p) \in F \oplus L] \\ \qquad \text{if } n = (x := e) \\ av \qquad \text{otherwise} \end{cases}$$

ただし, $av[x \mapsto v : x \in X]$ は, 全ての $x \in X$ に対し, x の写像先を v に変更した以外は av と同じ関数を表す.

関数 $udec : N \times state \longrightarrow econf$ を次のように定義する.

$$udec(n, (av, ec)) = ec'$$

ここで, ec' は文 n の型に応じて次のように決定される.

- n は任意の型

  全ての $c(p,n) \in C$ に対し, $ec'(c(p,n)) = inact$
  全ての $l(p,n) \in L$ に対し, $ec'(l(p,n)) = act$

- $n = (x := e)$

  全ての $(n,p) \in C \oplus D$ に対し, $ec'((n,p)) = act$

- $n = (\texttt{if}\ e)$ かつ $\mathcal{E}(n,e)av = \mathbf{T}$ のとき.

  全ての $ct(n,p) \in C$ に対し, $ec'(ct(n,p)) = act$. さらに, 全ての $q \in G(p)$ に対し, 全ての $(q,r) \in C \oplus F \oplus D$ に対し, $ec'((q,r)) = unchk$. 全ての $l(r,q) \in L$ に対し, $ec'(l(q,r)) = unchk$.

  全ての $cf(n,p) \in C$ に対し, $ec'(cf(n,p)) = inact$. さらに, 全ての $q \in G_{\mathbf{F}}(n) - G_{\mathbf{T}}(n)$ に対し, $\forall c(r,q) \in \widehat{C}.r \notin G_{\mathbf{F}}(n) \Rightarrow ec(c(r,q)) = inact$ が成り立つならば, 全ての $(q,r) \in C \oplus F \oplus D$ に対し, $ec'((q,r)) = inact$. 全ての $l(r,q) \in L$ に対し, $ec'(l(q,r)) = inact$.

- $n = (\texttt{if}\ e)$ かつ $\mathcal{E}(n,e)av = \mathbf{F}$ のとき.

  $\mathcal{E}(n,e)av = \mathbf{T}$ のときと同様. ct と cf, $G_{\mathbf{T}}(n)$ と $G_{\mathbf{F}}(n)$ を入れ換える.

上で変更されたエッジ以外に対しては ec と ec' は一致する.

以上の定義から, run における状態遷移関係を次のように定義する.

$$s_i \stackrel{n_i}{\to} s_{i+1} \stackrel{\text{def}}{\Longleftrightarrow}$$
$$n_i \in Next(s_i) \wedge av' = udav(av, n_i)$$
$$\wedge ec' = udec(s_i, n_i)$$

PDG の run は有限か無限である. 有限なら, 終了状態を s とすると, $Next(s) = \emptyset$ である.

これから, PDG の操作的意味の直観を述べる.

まず $condC$ についてであるが, 前半の条件は少なくとも一つの n に入る C-エッジが act でなければならないということを示している. これは, $ct(p,n) \in C$ のとき, p の判定結果が $\mathbf{T}$ の時には n が実行されるということ ($cf(p,n) \in C$ のときも同様) を表している. 後半の条件は, ループがあったとき, まだループ内に実行可能な文が残っているのにループ命令を実行してしまうことがないためのものである.

$condF$ についてであるが, この条件は n に入ってくる全ての F-エッジが act か $inact$ でなければなら

ないということを示す. これは, 全ての F-predecessor が実行済みか, 実行されないことがわかっていなければならないということを意味している. 後者の “実行されないことがわかっている場合” とはどういう場合かを図 8 で説明する.

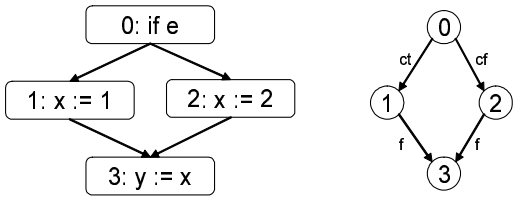

図 8: CFG と PDG

このとき, 0 での判定結果が **T** だった場合, 2 は実行されない. そのとき, 0 を実行した結果 $f(2,3)$ は *inact* になる. この後 1 を実行することで $f(1,3)$ は *act* になり, 3 が実行可能になる.

このように, 実行されないことがわかったノードから出るエッジを *inact* にするのは, `if` 文実行時に *udec* 関数で行われる.

condD についても *condF* と同様である.

condL についてであるが, 他の条件と違って n から出る L-エッジが全て *act* または *inact* であるという条件になっている. これは, $l(n,p)$ のとき, p の方が n より先に実行されなければならないということを反映している (これは, $LIDD(n,p)$ かつ $LCDD(n,p)$ の時には *LIDD* のみを PDG に F-エッジとして表すことにしているため成り立つ). これを図 6 で説明する. 図 6 において, 0 と 1 を実行した後で, もし 3 が実行可能になってしまうと 2 の実行前に 3 が実行されることがありえてしまう. しかし, $l(3,2)$ なので, 3 が実行されるためには $l(3,2)$ が *act* でなければならない. そのためには 2 が実行されなければならない. 2 が実行されたとき, *udec* から $l(3,2)$ は *act* になる.

PDG の run は一般には一意ではない. しかし, 次に定義する deterministic PDG の範囲においては全ての run が同じ最終状態を持つ.

Deterministic PDG を定義する前に, minimal common ancestor (mca) の概念を定義する.

定義 4.5 $G = (N, C, F, L, D)$ を PDG とする. p と q の minimal common ancestor は, G の制御依存サブグラフにおいて, p と q の共通の祖先のうち, 極小のものである. すなわち, どの p と q の他の共通の祖先にもそこから到達することはできないものである. p と q の minimal common ancestor の集合を $mca(p,q)$ と書く.

定義 4.6 (Deterministic PDG) $G = (N, C, F, L, D)$ が deterministic とは, 次の 1 から 3 の条件を満たす場合を言う.

1. $(p,n) \in \widehat{C} \wedge (q,n) \in \widehat{C}$ なら, $\forall r \in mca(p,q).\forall Q \in \{\mathbf{T},\mathbf{F}\}.\{p,q\} \not\subseteq G_Q(r)$.
2. $f_x(p,u) \in F \wedge f_x(q,u) \in F \wedge \exists r \in mca(p,q).\exists Q \in \{\mathbf{T},\mathbf{F}\}.\{p,q\} \subseteq G_Q(r)$ のとき, $d(p,q) \in D \vee d(q,p) \in D$.
3. $f(p,q) \in F$ で, R を G の制御依存サブグラフの強連結成分とする. $\exists r \in R.p \in G(r)$ ならば, $\exists r \in R.q \in G(r)$.

Deterministic PDG(以下 dPDG) の条件 1 は, n が p と q の両方に制御依存しているなら, p と q の全ての mca からは同じ ct-エッジもしくは cf-エッジで到達することはできない, というものである. これは p と q が共に実行可能になることがないために必要である (図 9).

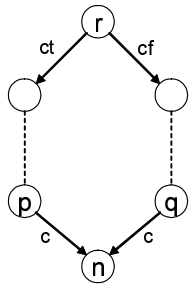

図 9: 条件 1

条件 2 は p と q が同じ変数を定義し, 同じノードに F-エッジを持ち, p と q が共に実行されることがありうるときは, p と q の間には定義順序関係がなければならない, ということを示している (図 10).

条件 3 は p から q にデータ依存関係があり, p があるループ内の文に制御依存している場合は, q もそのループ内の文に制御依存している, ということを示している (図 11).

定理 4.1 定義 2.1 の CFG から構成される PDG は deterministic である.

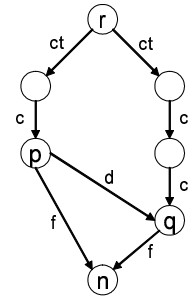

図 10: 条件 2

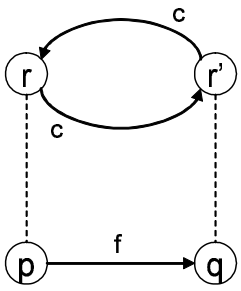

図 11: 条件 3

以下に dPDG の run において成り立つ幾つかの性質を示す.

補題 4.1 $s_0 \xrightarrow{n_0} s_1 \xrightarrow{n_1} \ldots$ を dPDG G の run とする. $q \in Next(s_i)$ のとき, $\forall u \in G(q).u \neq q \Rightarrow (\forall w \notin G(q).(w,u) \in \widehat{C} \Rightarrow ec_i(w,u) = inact)$.

補題 4.2 G を dPDG とする. また, s を G の run 上の state とする. 全ての $p \in Next(s)$ に対し, $\forall q \in Next(s) - \{p\}.s \xrightarrow{q} s'$ なら, $p \in Next(s')$.

補題 4.3 G を dPDG とする. また, s を G の run 上の state とする. また, $p, q \in Next(s)$ とする. このとき, $s \xrightarrow{p} s_1 \xrightarrow{q} s_2 \Leftrightarrow s \xrightarrow{q} s'_1 \xrightarrow{p} s_2$.

以上の補題から次の定理が導かれる.

定理 4.2 G を dPDG とする. また, s を G の run 上の state とする. s からある有限で長さ $m+1$ の run $s \xrightarrow{n} s_1 \xrightarrow{n_1} \cdots \xrightarrow{n_m} s_m$ が存在するとき, 全ての s からの run は有限で長さは $m+1$ であり, 最終 state は s_m である.

証明 m に関する帰納法で示す.
(Base) $m = 0$ のときは自明に成り立つ.
(Induction) s から長さ $m+1$ の有限の run $s \xrightarrow{p} s_1 \to \cdots \to s_m$ が存在するとする. また, $Next(s) = \{p, p_0, \ldots, p_j\}$ とする. 補題 4.2 より, $\{p_0, \ldots, p_j\} \subseteq Next(s_1)$ である. ここで, s_1 から長さ m の有限の run $s_1 \to \cdots \to s_m$ が存在するので, 帰納法の仮定より, s_1 からの全ての run は有限で長さは m で, 最終状態は s_m である. よって, $i \in [0, j]$ に対し $s_1 \xrightarrow{p_i} s_2^i$ とすると, s_2^i からの全ての run は長さ $m-1$ で最終状態は s_m. ここで, 補題 4.3 より, 全ての $i \in [0, j]$ に対して, $s \xrightarrow{p} s_1 \xrightarrow{p_i} s_2^i$ であることから, $s \xrightarrow{p_i} s_1 \xrightarrow{p} s_2^i$ が成り立つ. よって, s からの全ての run は長さ $m+1$ で最終状態は s_m である. □

この定理は dPDG の run は終了するならば全て同じ終了状態となることを意味している.

5 CFG の操作的意味と PDG の操作的意味の一致

この節では CFG とそれに対応する PDG は, 同じ実行結果を返すことを証明する.

定義 5.1 σ, av をそれぞれ *store*, *avail* とする. N をノードの集合, *var* を変数の集合とする. $n \in N$ とするとき, $use(n)$ で, n で使用されている変数の集合を現す.

$\sigma \approx_n av$ を次のように定義する.

$$\forall x \in use(n).\sigma(x) = av(n, x)$$

定理 5.1 (CFG と PDG の操作的意味の一致) $P = (N, E)$ を CFG, $G = (N, C, F, L, D)$ を P の PDG とする. $\forall n \in N.\sigma_0 \approx_n av_0$ とする. このとき, P の有限の run $\sigma_0 \xrightarrow{n_0} \sigma_1 \xrightarrow{n_1} \cdots \xrightarrow{n_m} \sigma_m$ が存在することと G の有限の run $(av_0, ec_0) \xrightarrow{n'_0} (av_1, ec_1) \xrightarrow{n'_1} \ldots \xrightarrow{n'_m} (av_m, ec_m)$ が存在することとは同値であり, $n_m = (\texttt{ret}\ x)$ とすると, $\sigma_m(x) = av_m(n_m, x)$ である.

この定理を証明するために, 以下の補題を用いる.

補題 5.1 $P = (N, E)$ を CFG とし $G = (N, C, F, L, D)$ を P に対する PDG とする. $(n_1, n_2) \in E$ のとき, $\forall p \in N.ct(p, n_1) \in C \Rightarrow ct(p, n_2) \in C$ が成り立つ (ct を cf に置き換えた場合も成り立つ).

補題 5.2 $P=(N,E)$ を CFG, $G=(N,C,F,L,D)$ を P の PDG とする. $\forall n \in N.\sigma_0 \approx_n av_0$ とする. P の run $\sigma_0 \stackrel{n_0}{\to} \sigma_1 \stackrel{n_1}{\to} \cdots \stackrel{n_k}{\to} \sigma_k$ に対し, G の run $(av_0, ec_0) \stackrel{n_0}{\to} (av_1, ec_1) \stackrel{n_1}{\to} \cdots \stackrel{n_k}{\to} (av_k, ec_k)$ が存在するとき, $\stackrel{n_k}{\to} \sigma_k \stackrel{n_{k+1}}{\to}$ とすると, n_{k+1} から p に P において loop closing edges を通らずに到達できないならば,

$$\forall (p,q) \in \widehat{C}.ec_{i+1}((p,q)) = inact$$
$$\forall (p,q) \in F \oplus D.ec_{i+1}((p,q)) \neq unchk$$
$$\forall (p,q) \in L.pec_{i+1}((q,p)) \neq unchk$$

が成り立つ.

上の補題から, 次の補題が成り立つ. この補題は, CFG P から構成される PDG G は P と同じ実行順番の run を持つということを言っている.

補題 5.3 $P=(N,E)$ を CFG, $G=(N,C,F,L,D)$ を P の PDG とする. $\forall n \in N.\sigma_0 \approx_n av_0$ とする. このとき, P の run $\sigma_0 \stackrel{n_0}{\to} \sigma_1 \stackrel{n_1}{\to} \ldots$ に対し, G の run $(av_0, ec_0) \stackrel{n_0}{\to} (av_1, ec_1) \stackrel{n_1}{\to} \ldots$ が存在し, $\forall x \in use(n_{i+1}).\sigma_i(x) = av_i(n_{i+1}, x)$ が成り立つ.

証明 $(av_i, ec_i) = s_i$ とする. $\forall i.(\sigma_i \stackrel{n_{i+1}}{\to} \sigma_{i+1} \Rightarrow n_{i+1} \in Next(s_i)) \wedge \forall x \in use(n_i).\sigma_i(x) = av_i(n_i, x)$ を示せばよい. これを i の帰納法で示す.

(Base) $i = 0$ のとき, n_0 は P における入口ノードである. 入口ノードは $entry$ からの C-エッジを持ち, 操作的意味の定義より ec_0 は $\forall (entry, q) \in C.ec_0((entry, q)) = act$ を満たすので, $ec_0(entry, n_0) = act$ である. また, 全てのエッジ (p,q) に対し, $ec_0((p,q)) = unchk$ なので $\forall q \in G(n_0).q \neq n_0 \Rightarrow \forall c(r,q) \in C.ec(c(r,q)) \neq act$ も成り立つ. よって $condC(s_0, n_0)$ は成り立つ. また, 入口ノードは入ってくる F, D-エッジは持たず, 自分以外に出て行く L- エッジも持たない. したがって $condF, condL, condD$ も成り立つ. また, $\forall n \in N.\sigma_0 \approx_n av_0$ より, $\forall x \in use(n_0).\sigma_0(x) = av_0(n_0, x)$ は明らかに成り立つ.

(Induction) n_i のタイプによって場合分けする.

$n_i = (x := e)$ の時, ある一意な n が存在し, $(n_i, n) \in E$ である. $condCFLD$ の各条件を満たすことをこれから示す.

- $condC$

  まずは $condC$ の始めの conjunct, すなわち,$\exists c(p,n) \in C.ec(c(p,n)) = act$ が成り立つことを示す.

  補題 5.1 より, $ct(p, n_i) \in C \Rightarrow ct(p,n) \in C \vee cf(p, n_i) \in C \Rightarrow cf(p,n) \in C$ である. $n_i \in Next(s_{i-1})$ より, $\exists (p, n_i) \in C.ec_{i-1}((p, n_i)) = act$. よって, $\exists j < i-1.n_j = p \wedge \forall k \in [j, i-1].ec_k((p,n)) = act$. $(n_i, n) \in E$ より, $\forall k \in [j, i-1].n_k \neq n$. 従って $ec_{i-1}((p,n)) = act = ec_i((p,n))$. よって $condC$ の始めの conjunct は成り立つ.

  $condC$ の 2 番目の conjunct, すなわち $\forall q \in G(n).q \neq n \Rightarrow \forall c(r,q) \in C.ec(c(r,q)) \neq act$ が成り立つのは補題 4.1 より明らか.

- $condF$

  ある p が存在して $ec_i((p,n)) = unchk$ と仮定すると, 補題 5.2 より n_{i+1} から p は P 中で loop closing edge を通らずに到達可能である. $(n_i, n) \in E$ より, n から p へも loop closing edge を通らずに到達可能である. また, $f(p,n)$ より p から n へも loop closing edge を通らずに到達可能である. これは矛盾.

- $condL, condD$

  $condF$ の時と同様.

以上から, $n \in Next(s_i)$ は示せた. 次に $\forall y \in use(n_{i+1}).\sigma_i(y) = av_i(n_{i+1}, y)$ が成り立つことを示す. $\sigma_i = \sigma_{i-1}[x \mapsto \mathcal{E}(e)\sigma_{i-1}]$ であり, $av_i = av_{i-1}[(p,x) \mapsto \mathcal{E}(n,e)av_{i-1} : (n,p) \in F \oplus L]$.

$y = x$ のとき, $\sigma_i(x) = \mathcal{E}(e)\sigma_{i-1}$. $x \in use(n)$ とすると, $f(n_i, n) \in F$ なので, $av_i(n,x) = \mathcal{E}(n_i,)av_{i-1}$. 帰納法の仮定より, $\forall y \in use(n_i).\sigma_{i-1}(y) = av_{i-1}(n_i, y)$. 従って, $\mathcal{E}(e)\sigma_{i-1} = \mathcal{E}(n_i, e)av_{i-1}$. よって $\sigma_i(x) = av_i(n,x)$.

$y \neq x$ のとき, $\exists j < i.\exists f(p,n) \in F.n_j = p$ ならば, そのような j のうち最大のものを j^* とすると, 全ての $k \in [j^*, i]$ に対し,n_k は y を定義しない. 従って, $\sigma_i(y) = \sigma_{j^*}(y) = av_{j^*}(n,y) = av_i(n,y)$. $\forall j < i.\forall f(p,n) \in F.n_j \neq p$ ならば, $\sigma_i(y) = \sigma_0(y) = av_0(y) = av_i(y)$.

よって成り立つ.

$n_i = (\texttt{if}\ e)$ の時, $(n_i, n)_{\mathbf{T}} \in E \wedge (n_i, n')_{\mathbf{F}} \in E$ とする. このとき, $\mathcal{E}(e)\sigma_i = \mathbf{T}$ と仮定する ($\mathbf{F}$ のときも同様).

- *condC*

  まずは *condC* の始めの conjunct, すなわち,$\exists c(p,n) \in C.ec(c(p,n)) = act$ が成り立つことを示す. $(n_i,n)_{\mathbf{T}} \in E$ より, $ct(n_i,n) \in C$. また, $\mathcal{E}(e)\sigma_i = \mathbf{T}$ より, 帰納法の仮定から, $\mathcal{E}(n_i,e)av_i = \mathbf{T}$. よって $ec_i(ct(n_i,n)) = act$. よって成り立つ.

  2 番目の conjunct は $n_i = (x := e)$ の時と同様.

- *condF*, *condL*, *condD*

  $n_i = (x := e)$ の時と同様.

  また, $\sigma_i = \sigma_{i-1}, av_i = av_{i-1}$ なので,$\forall x \in use(n_{i+1}).\sigma_i(x) = av_i(n_{i+1},x)$ は成り立つ.

$n_i = (\texttt{ret}\ x)$ の時, CFG の定義から, n_i からは全てのノードに到達不可能. 従って補題 5.2 より, $\forall (p,q) \in \widehat{C}.ec_i((p,q)) = inact$. よって, $Next(s_i) = \emptyset$.

また, $\sigma_i = \sigma_{i-1}, av_i = av_{i-1}$ なので,$\forall x \in use(n_{i+1}).\sigma_i(x) = av_i(n_{i+1},x)$ は成り立つ. □

定理 5.1 は定理 4.1, 4.2 と, 補題 5.3 から示される.

6 最適化の正当性

前節によりプログラムとそれに対応する PDG の意味は一致することを証明した. これを利用してコンパイラが行う最適化の正当性の証明を PDG 上で行うことができる. この節ではコンパイラが行うコード最適化の正当性を, 複写伝搬を例に示す. PDG 上での複写伝搬は次のようなグラフ書き換えとして表される (図 12).

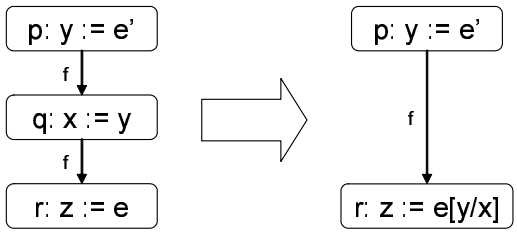

図 12: 複写伝搬

図 12 の左側のグラフを G, 右側のグラフを G' とする. 簡単のため p,q,r は *entry* からの C-predecessor を持つとする. このとき, 初期 *avail* を av_0 とし, G の初期 *econf* を ec_0, G' の初期 *econf* を ec'_0 とするとき, それぞれ一意な G と G' の run $(av_0,ec_0) \xrightarrow{p} (av_1,ec_1) \xrightarrow{q} (av_2,ec_2)$ と $(av_0,ec'_0) \xrightarrow{p} (av'_2,ec'_2)$ が存在する. 複写伝搬の正当性は, $\mathcal{E}(r,e)av_2 = \mathcal{E}(r,e[y/x])av'_2$ として定義される. 以下ではこれを証明する.

まず, 操作的意味より,

$$\begin{aligned} av_1 &= av_0[(q,y) \mapsto \mathcal{E}(p,e')av_0] \\ av_2 &= av_1[(r,x) \mapsto \mathcal{E}(q,y)av_1] \\ &= av_1[(r,x) \mapsto av_1(q,y)] \\ &= av_1[(r,x) \mapsto \mathcal{E}(p,e')av_0] \end{aligned}$$

となる. また,

$$av'_2 = av_0[(r,y) \mapsto \mathcal{E}(p,e')av_0]$$

である.

$\forall v \in use(r).v \notin \{x,y\} \Rightarrow av_2(r,v) = av_0(r,v) = av'_2(r,v)$ であるので, $\mathcal{E}(r,e)av_2 = \mathcal{E}(r,e[y/x])av'_2$ を示すためには, $av_2(r,x) = av'_2(r,y)$ を示せばよい. これは, $av_2(r,x) = \mathcal{E}(p,e')av_0 = av_2(r,y)$ より成り立つ.

7 まとめと今後の方向性

本論文では, 中間コードのように自由に goto 文が出現するようなプログラムに対する操作的意味を定義し, プログラムとそれに対応する PDG が同じ意味を持つことを示した. また, コード最適化手法の正当性の証明に PDG の操作的意味が有用であることを示すために, 一例として, よく知られた最適化手法である複写伝搬の正当性を PDG の操作的意味を用いて証明した.

本論文の成果から以下のような発展が考えられる.

- 他の多くの最適化手法の正当性を証明する. そのための一般的な方法論を考案する. 例えば制御の依存関係やデータの依存関係を述語とする, 最適化の正当性を証明するための推論体系を構築する. また, その意味を時間論理式で与え, 完全性, 健全性などを議論するという方法を考えている.

- 逐次プログラムの意味との一致だけでなく, 並行プログラムとの意味の一致が成り立つかどうかを考察する.

- PDG から他の様々な中間表現への変換アルゴリズムを考案し, その正当性を示す. PDG から逐

次プログラムの再構成アルゴリズムは既に幾つかの論文で与えられているが [9, 10, 11], PDG と対応する逐次プログラムが存在しない時にノードを複製したり, 条件文を追加したりしなければならない. そのように逐次プログラムが得られるように PDG を変換した結果が元の PDG と同じ意味を持つことを確認する必要がある. 本論文で提案した操作的意味を用いてそれを証明する. また, PDG から逐次プログラムへの変換だけでなく, 並行プログラムへの変換アルゴリズムなども考案する.

- PDG を中間表現に用いた最適化器の実現. CFG を利用した最適化は広く成功を収めているが, PDG は CFG よりもコード最適化に適した中間表現であるので実際に PDG 上での最適化器を実装してその有用性を確認する.

参考文献

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers:Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] E. Cartwright and M. Felleisen. The semantics of program dependence. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pp. 13–27, New York, NY, USA, 1989. ACM Press.

[3] Jeanne Ferrante and Karl J. Ottenstein. A program form based on data dependency in predicate regions. In *Principles of Programming Languages*, pp. 217–236, 1983.

[4] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319–349, 1987.

[5] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Symposium on Principles of Programming Languages*, pp. 283–294, 2002.

[6] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Compiler optimization correctness by temporal logic. *Higher Order and Symbolic Computation*, Vol. 17, No. 3, pp. 173–206, 2004.

[7] R. Parsons-Selke. A rewriting semantics for program dependence graphs. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 12–24, New York, NY, USA, 1989. ACM Press.

[8] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, Vol. 16, No. 9, pp. 965–979, 1990.

[9] B. Simons, D. Alpern, and J. Ferrante. A foundation for sequentializing parallel code. In *SPAA '90: Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, pp. 350–359, New York, NY, USA, 1990. ACM Press.

[10] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Redmond, WA, 1993.

[11] Jia Zeng, Cristian Soviani, and Stephen A. Edwards. Generating fast code from concurrent program dependence graphs. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pp. 175–181, New York, NY, USA, 2004. ACM Press.