意味的制約の書き換えによるコンパイラのコード最適化

Compiler Code Optimization by Rewriting Semantic Constraint

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

これまでコンパイラの最適化についてモデル生成によるコードの移動, 削除を自動化する手法を提案してきたが, 複写伝播などのようなコードの書き換えを伴う変換を扱うことができなかった. 本論文ではこの種の最適化を扱うためにコードが満たさなければならない意味的制約を項集合で表し, その項集合の書き換えを行うことで最適なコードが満たすべき意味的制約を得, それを用いて最適コードをモデル生成により獲得する手法について報告する.

1 はじめに

本論文では [5] で提案した, コンパイラにおけるコード最適化を時間論理のモデル生成で行なうという手法の拡張を述べる. [5] では最適なコードが満たすべき制約として, オリジナルコードと同じ意味を持たなければならないという意味的制約と, 無用コードやループ不変式がないなどの最適であるための制約の 2 種類の制約を時間論理で記述し, そのモデルを生成するというものであった. 意味的制約は, オリジナルコードのデータの依存関係を保存するという制約であるため, この手法で扱うことができる最適化は変数の定義参照関係を変えないもの, すなわち命令の移動, 削除を行なうものだけであり, 複写伝搬や共通部分式の削除などのような命令の書き換えや挿入を行なってデータの依存関係を変える最適化は扱うことができなかった.

本論文では [5] において時間論理で記述していた意味的制約を項集合で表し, その項集合の書き換えを行なうことでデータの依存関係を変えるような最適化を扱う手法を提案する. 例えば複写伝搬は, 複写文の左辺の変数の使用を右辺の変数の使用に置き換えるような変換操作であるが, この操作を変数の定義参照関係を表す項の書き換えとして捉えた.

書き換え規則によってコンパイラのコード最適化を表現するという手法は多くなされてきているが, それらは実際のコードに対して行われていた [2, 3]. 本論文では実際のコードではなく, コードが満たすべき意味的制約を項集合で表現し, その書き換えを行うという手法を提案する. この手法ではコード中の実際の命令の順番に関係なく最適化を適用することができるため, コード移動を行う最適化の結果によって項集合書き換えによる最適化の適用可能性が影響を受けることはない.

2 中間言語

この節では単純な中間言語を導入する. これは [4] のものを幾らか制限したものである.

定義 2.1 コード π は以下の形を持つ.

$$\pi = \texttt{read X}; I_1; I_2; \ldots; I_{m-1}; \texttt{write Y}$$

ここで,`X` と `Y` は変数で,$I_1, \ldots, I_{m-1}$ は命令であり, $Nodes_\pi = \{0, 1, 2, \ldots, m\}$ 内の自然数のラベルを持つ. 更に,I_0 は初めの命令 `read` X であり,I_m は最後の命令 `write` Y とする.`read`, `write` はそれぞれコードの先頭と末尾にしか現れない. その他の命令は以下の文法により与えられる.

Inst	$\ni$	I	::=	`skip` | X `:=` E | `goto` n | `if` X `goto` n `else` n'
Expr	$\ni$	E	::=	L | O L L
Op	$\ni$	O	::=	`+` | `-` | `*` | `/` | `**` | ...
Ltrl	$\ni$	L	::=	constant c | X
Var	$\ni$	X	::=	`X` | `Y` | `Z` | ...
Label	$\ni$	n, n'	::=	0 | 1 | ... | m

ここで `**` はべき乗演算子である.

本論文ではコード中のループの形などに制限を加える. ここでいうループはコードのフローグラフにおける, 以降で定義されるような**自然なループ**をさす.

コントロールフローグラフ (または単純にフローグラフ) は以下のようにコードから定められる. グラフのノードは一つの命令に対応し, 辺はコントロールフローに対応する.

これから自然なループを定義するために, あるノードがあるノードを**支配する**という概念や**後向きの辺**という言葉を定義する. これらの定義は [1] に基づく.

フローグラフにおいて, 開始ノードから n へ至る全てのパス上に d があるとき, d が n を**支配する**という.

辺 $a \to b$ において,b が a を支配する時,$a \to b$ は**後向きの辺**という.

後向きの辺 $n \to d$ が与えられた時, この辺に対する**自然なループ**とは, d を通らずに n に到達可能なノードの集合に d を加えたノードの集合である.

ループの**入口**はループ内の全ノードを支配するノードであり, ループの**出口**はループ外の命令への遷移を持つノードである.

二つのループが入口を共有しないならば, それらは互いに素であるか一方が他方に完全に含まれる（入れ子になっている）かのどちらかである. 本論文では入口を共有するループは考えない.

可約フローグラフは, 後向きの辺を全て取り除いたときに開始ノードから全てのノードが到達可能であり, 閉路のないグラフとなるようなフローグラフのことである.

可約グラフには, ループを構成するノードの集合は後向きの辺を一つ持つという性質がある. 従ってフローグラフが可約となるようなコードにおいて全てのループを見つけるには, 全ての後向きの辺を調べれば良い.

本論文ではフローグラフが可約となるようなコードだけを考える. 更に, コード中のループに以下のような制限を加える. まず, 全てのループは入口と出口を一つしか持たず, ループを構成する後向きの辺を $n \to d$ とすると,d がループの入口であり,n がループの出口でなければならない. また, ループの出口は `if` 文でなければならない. すなわち, ループを繰り返すかそれとも抜けるかの判断は出口で行なわれる. これは, ループの形は `do...while` の形をしていなければならないということを意味する. これらの制限は中間言語の表現力を弱めるものではない.

更に, ループ以外にもコードにいくつかの制限を加える. この制限はコードを**基本ブロック**で分割した際の入口と出口に対するものである. 基本ブロックの定義は以下の通りである.

定義 2.2 **リーダー**はコードの最初の命令 (`read`), ジャンプ文 (`goto`, `if`) のジャンプ先の命令, ジャンプ文の (字面上の) 次の命令である. **基本ブロック**はリーダーから次のリーダーの一つ手前までの命令の列である.

定義から, リーダーはコード中の命令の順番に依存して決まる. 本研究では基本ブロック構造を変えるような最適化は考えていないので, 代入文の順番によらず基本ブロックの接続関係が一定となるコードだけを対象とする. すなわち, 基本ブロックの入口は全て何らかの `goto`, `if` のジャンプ先でなければならない, という制限を加える. また,`goto`, `if` のジャンプ先は `skip` 文でなければならないという制限も加える. これは, ジャンプ先の命令文が代入文であった場合, その代入文を移動したり削除したりすると基本ブロックの接続関係が変わってしまうからである.

また, コード中に現れる分岐は全て互いに素か入れ子になっているかのどちらかであるとする. すなわち, 分岐の途中に別の分岐の辺が入ってくることはなく, また分岐の途中でまた分岐があった場合, 内側の分岐が先に合流する.

以上をまとめると, コードが満たさなければならない制限は以下のようになる.

R1: 基本ブロックのリーダーはコードの最初の命令を除いて `skip` であり, 何らかの `goto`, `if` 文のジャンプ先である.

R2: ループは入口と出口をそれぞれ一つずつ持つ.

R3: 後向きの辺の始点は `if` 文でありループの出口でなければならない. 後向きの辺の終点はループの入口でなければならない.

R4: 分岐は互いに素であるか入れ子になっていなければならない.

3 項集合による意味的制約の表現

データの依存関係を変えるような最適化は,[5] において時間論理式によって表現されていた意味的制約を書き換えることで対応することができる. すなわち,

```
5: X := V + W
6: Y := X
```

```
7: Z := Y + 1
```

のようなコードがあったとき, 意味的制約として “5 で定義された X は 6 で使われる” というものと “6 で定義した Y は 7 で使われる” というものがある. このコードに複写伝搬を適用して

```
5: X := V + W
7: Z := X + 1
```

とした後の意味的制約は, “5 で定義された X は 7 で使われる” となる. したがって複写伝搬という操作は, 複写伝搬を適用する前の満たすべき意味的制約から適用後の意味的制約を得るような書き換え操作と捉えることができる.

[5] では “d で定義された x は n で使われる” などの制約を時間論理式で表現していたが, 時間論理式を書き換えの対象として扱うのは煩雑であるので, 時間論理による意味的制約の別表現となるような項集合を定義し, その項集合を書き換えることでデータの依存関係を変えるような最適化を行う.

この節ではコードの意味的制約を項集合で表現する方法を述べる.

手続き 3.1

入力 コード $\pi = I_0; I_1; \ldots; I_m$ とそのフローグラフ G および π の基本ブロックの集合

出力 π の意味的制約を表す項集合

1. (基本ブロックの入口と出口)
   全てのブロック B に対し,p と q をそれぞれ B の最初と最後の命令とする. この時 $entry(B,p), exit(B,q)$ という項を生成する.
2. (ループの入口と出口)
   全ての後向きの辺 $p \rightarrow q$ に対し,$loopentry(p), loopexit(p,q)$ という項を生成する. ここで,**R3** から, ループの入口と出口は後向きの辺の終点と始点であることに注意せよ.
3. (代入文以外の命令)
   全ての $n \in Nodes_\pi$ に対し,I_n が代入文でなければ $stmt(n, I_n)$ という項を生成する.
4. (代入文)
   全ての $n \in Nodes_\pi$ に対し,I_n が代入文ならば次の項を生成する:
   - If $I_n = (x := y\ op\ z)$, then $assign(n, x, y\ op\ z)$
   - If $I_n = (x := c(\text{constant}))$, then $cassign(n, x, c)$
   - If $I_n = (x := y(\text{variable}))$, then $copy(n, x, y)$
5. (データの流れ)
   代入文 n で使用している変数 x を定義しているノード d を見つけるには, フローグラフを n から逆向きに辿り, 初めて x を定義する代入文を全て見つければ良い.$D_n \subseteq Nodes_\pi \times Var(\pi)$ とする.D_n は n で使用する変数と, その変数を定義するノードのうち,n に定義が到達するノードのペアの集合である. また,$c_n \in Nodes_\pi$ とする.c_n は何らかの `if` 文の successor(**R1** より `skip`) で, n が実行されるための分岐の条件を表している. 例えば,n が実行されるかどうかはある分岐文 `if X goto p else q` の結果によるとする. n が実行されるのは $\mathtt{X} = true$ の時であれば $c_n = \mathtt{p}$ となり,$\mathtt{X} = false$ の時であれば $c_n = \mathtt{q}$ となる.

   D_n と c_n を計算する手続きは以下のようになる.

Step1 全てのノード $n \in Nodes_\pi$ に対し,$D_n = \emptyset$ とする.

Step2 G' を G から全ての後向きの辺を除いて得られるグラフとする.

Step3 全ての $n \in Nodes_\pi$ に対し以下を実行する.

Step4 n で使用されている全ての変数 x に対し以下を実行する.

Step5 G' における n からの全ての逆向きのパスに対し,x を初めて定義するような全てのノード d を見つけ,$D_n = D_n \cup \{(d, x)\}$ とする.d がもしある分岐のブロックに含まれるなら, c_d を d を支配する d に最も近い `if` 文の successor のうち, d を支配する方のノードとする.

Step6 更に,n が G 中のあるループに含まれているならば Step7 を行なう.

Step7 n を含むループの後向きの辺を $e_1, \ldots, e_l$ とする. ここで, $e_i, e_j (i < j)$ に対し, e_i で構成されるループは e_j で構成されるループに含まれる. 各 e_i に対し,G を n から n 自身に後向きの辺として e_i だけを通るように逆向きに辿り,x を初めて定義するような全てのノード d を見つけ,$D_n = D_n \cup \{(d, x)\}$

とする.c_d を n から d への逆向きのパス上で最も近いループの入口とする.

全ての $e \in D_n$ に対し,$df(e.1, n, e.2)$ という項を生成する.$e.1$ と $e.2$ はそれぞれ e の 1 番目の成分と 2 番目の成分を表す. また,c_n が定義されていれば,$cf(n, c_n)$ という項を生成する.

以上の規則から生成された全ての項の集合を, コードの意味的制約を表す項集合とする.

この手続きで得られた項集合による意味的制約は,[5] での時間論理式での記述に変換することができる. その変換は自然であり, 紙面の都合から本論文ではその変換手続きを述べることはしない.

4 項集合書き換えによる最適化

この節では 3 節の手続きで得られる項集合に対する書き換え規則を定義する. 項集合書き換えによってデータの依存関係を変えるような最適化を行なう.

4.1 書き換え規則

本論文では以下の最適化技法を書き換えによって扱う.

1. 定数計算
2. 強さの軽減
3. 代数的恒等性の利用
4. 恒等代入の削除
5. 定数伝搬
6. 複写伝搬
7. 共通部分式の削除

これらの技法に対する書き換え規則を以下に定義する.

定義 4.1 制約集合の書き換え規則は次のような形式を持つ.

$$Condition\text{ならば } RewritingProcedure$$

ここで,*Condition* はその制約集合に書き換え規則が適用できるための条件であり,*RewritingProcedure* は書き換え手続きである. 書き換え手続きの原始的な操作は $t \Rightarrow t', \text{add } t, \text{del } t$ である. それぞれの意味をこれから述べる.

- $t \Rightarrow t'$
  $t \in S$ ならば t を t' に置き換える.
- add t
  t を S に追加する.
- del t
  $t \in S$ ならば t を S から削除する.

ただし,S は書き換えの対象となる項集合である. 書き換え規則はノード, 変数, 定数, 演算子などを表すメタ変数を用いて与えられる. 書き換えを実行するにはメタ変数を実際のパラメータに具体化する必要がある. 実際のパラメータのメタ変数への代入は θ で表される. 各書き換え規則中でメタ変数は下線が付されている.

1. 定数計算
   $\underline{c_1}, \underline{c_2}$ が定数ならば,

   $$assign(p, x, \underline{c_1}\ op\ \underline{c_2}) \Rightarrow cassign(p, x, eval(\underline{c_1}\ op\ \underline{c_2}))$$

2. 強さの軽減

   $$assign(p, y, x\ \texttt{**}\ 2) \Rightarrow assign(p, y, x\ \texttt{*}\ x)$$
   $$assign(p, y, 2\ \texttt{*}\ x) \Rightarrow assign(p, y, x\ \texttt{+}\ x)$$
   $$assign(p, y, x\ \texttt{/}\ 2) \Rightarrow assign(p, y, x\ \texttt{*}\ 0.5)$$

3. 代数的恒等性の利用

   $$assign(p, y, x\ \texttt{+}\ 0) \Rightarrow copy(p, y, x)$$
   $$assign(p, y, x\ \texttt{-}\ 0) \Rightarrow copy(p, y, x)$$
   $$assign(p, y, x\ \texttt{*}\ 1) \Rightarrow copy(p, y, x)$$
   $$assign(p, y, x\ \texttt{/}\ 1) \Rightarrow copy(p, y, x)$$

4. 恒等代入の削除
   $copy(\underline{p}, \underline{x}, \underline{x}) \in S$ ならば,

   del $copy(\underline{p}, \underline{x}, \underline{x}), cf(\underline{p}, n)$
   全ての $df(\underline{p}, q, \underline{x}) \in S$ であるような q に対し,
   del $df(\underline{p}, q, \underline{x})$
   全ての d に対し,
   $df(d, \underline{p}, \underline{x}) \Rightarrow df(d, q, \underline{x})$

5. 定数伝搬
   $\forall p(df(p, \underline{q}, \underline{x}) \in S \rightarrow cassign(p, \underline{x}, \underline{c}) \in S)$ ならば,

   $$assign(\underline{q}, z, \underline{x}\ op\ y) \Rightarrow assign(\underline{q}, z, \underline{c}\ op\ y)$$
   $$copy(\underline{q}, z, \underline{x}) \Rightarrow cassign(\underline{q}, z, \underline{c})$$
   全ての p に対し,
   del $df(p, \underline{q}, \underline{x})$

6. 複写伝搬

$$copy(\underline{p},\underline{x},\underline{w}) \in S \wedge \underline{x} \neq \underline{w}$$
$$\wedge \forall q(df(\underline{p},q,\underline{x}) \in S$$
$$\rightarrow \forall p'(p' \neq \underline{p} \rightarrow df(p',q,\underline{x}) \notin S)$$

ならば

全ての $df(\underline{p},q,\underline{x}) \in S$ であるような q に対し

$assign(q,z,\underline{x}\ op\ y) \Rightarrow assign(q,z,\underline{w}\ op\ y)$

$copy(q,z,\underline{x}) \Rightarrow copy(q,z,\underline{w})$

$stmt(q,\texttt{if}\ \underline{x}\ \texttt{goto}\ n\ \texttt{else}\ n')$
$\Rightarrow stmt(q,\texttt{if}\ \underline{w}\ \texttt{goto}\ n\ \texttt{else}\ n')$

del $df(\underline{p},q,\underline{x})$

del $copy(\underline{p},\underline{x},\underline{w}), cf(\underline{p},n)$

全ての d に対し

$df(d,\underline{p},\underline{w}) \Rightarrow df(d,q,\underline{w})$

7. 共通部分式の削除

$$assign(\underline{p},\underline{w},\underline{y}\ \underline{op}\ \underline{z}) \wedge assign(\underline{q},\underline{x},\underline{y}\ \underline{op}\ \underline{z})$$
$$\wedge \forall d(df(d,\underline{p},\underline{y}) \in S \leftrightarrow df(d,\underline{q},\underline{y}) \in S)$$
$$\wedge \forall d(df(d,\underline{p},\underline{z}) \in S \leftrightarrow df(d,\underline{q},\underline{z}) \in S)$$

ならば,

$u = \text{newvar}(), r = \text{newnode}()$ として,

add $assign(r,u,\underline{y}\ \underline{op}\ \underline{z}), df(r,\underline{p},u), df(r,\underline{q},u)$

$assign(\underline{p},\underline{w},\underline{y}\ \underline{op}\ \underline{z}) \Rightarrow copy(\underline{p},\underline{w},u)$

$assign(\underline{q},\underline{x},\underline{y}\ \underline{op}\ \underline{z}) \Rightarrow copy(\underline{q},\underline{x},u)$

全ての d に対し,

$df(d,\underline{p},\underline{y}) \Rightarrow df(d,r,\underline{y})$

$df(d,\underline{p},\underline{z}) \Rightarrow df(d,r,\underline{z})$

del $df(d,\underline{q},\underline{y}), df(d,\underline{q},\underline{z})$

定義 4.2 i 番目の書き換え規則を用いて S を書き換えた結果 S' が得られたとき, 書き換え関係は次のように定義される.

$$S \overset{(i,\theta)}{\Longrightarrow} S'$$

ここで,θ は書き換え規則中に現れるメタ変数への代入である.

定義 4.1 の書き換え規則はコードの意味を変えないということを示すことができる.

定理 4.1 任意の書き換え $S \overset{(i,\theta)}{\Longrightarrow} S'$ に対し,S で表される意味的制約を満たすプログラム π と S' で表される意味的制約を満たすプログラム π' は同じ意味を持つ.

証明概略 π と π' の意味が等しいことを示すのに [4] の手法を拡張して用いることができる. これは π と π' の計算列の長さに関する帰納法で示すことができる. ■

コードに直接書き換えを適用する手法と異なり, 本論文の手法は意味的制約を書き換えるというものである. 従って, 書き換えの適用可能性に実際の命令の順番は影響しないという利点がある. この利点を次の例で示す.

```
0: read X
1: Y := X + 1
2: Z := Y
3: Y := 2 * X
4: A := Z + 1
5: B := A + Y
6: write B
```

2 は複写文であり,4 において 2 で定義している Z を使用しているので, 複写伝搬が適用できそうである. しかし,2 と 4 の間の代入文 3 が Y を定義しているので, このままでは 4 の Z を Y に置き換えることはできない.

このコードに対して, 本研究の手法を適用してみる. このコードの意味的制約を表す項集合のうち, 関係する部分だけを取り出すと,

$df(0,1,\texttt{X}), \underline{df(1,2,\texttt{Y})}, \underline{df(2,4,\texttt{Z})}, df(3,5,\texttt{Y}),$
$df(4,5,\texttt{A}), df(5,6,\texttt{B}), \underline{copy(2,\texttt{Z},\texttt{Y})}, \underline{assign(4,\texttt{A},\texttt{Z + 1})}$

となる. 下線部の項に定義 4.1 複写伝搬の規則が適用できるので適用すると, その結果は

$df(0,1,\texttt{X}), df(1,2,\texttt{Y}), \underline{df(1,4,\texttt{Y})}, df(3,5,\texttt{Y}),$
$df(4,5,\texttt{A}), df(5,6,\texttt{B}), assign(4,\texttt{A},\texttt{Y + 1})$

となる. 下線部の項が書き換えられている. 書き換え前は “2 から 4 の間 Z は定義されてはいけない” という制約を表していたが, 書き換え後は “1 から 4 の間 Y は定義されてはいけない” という制約を表す. 書き換え後の制約を満たすコードとして例えば次のようなコードが得られるであろう.

```
0: read X
```

```
1: Y := X + 1
4: A := Y + 1
3: Y := 2 * X
5: B := A + Y
6: write B
```

ある項集合で表された意味的制約を満たすコードは一般には複数存在する. というのは, 同じデータの依存関係を満たしていても, 命令の順番にはある程度の自由度があるためである. データの依存関係を変えないように命令の入れ換えを行なったコードであれば同じ意味的制約を表す項集合が得られる. 従ってこの手法ではコード移動を行なう最適化とは独立に定義 4.1 で扱われている最適化を実行することが可能である.

実際のコードに複写伝搬などを適用するというアプローチの場合には, コード移動を行う最適化を行った結果複写伝搬などが適用可能になることがあるため, コード移動を行う最適化と複写伝搬を交互に試す必要があった. それに対し本論文が提案する意味的制約の書き換えによって複写伝搬などを扱うならば, 元のデータの依存関係を複写伝播後のコードが満たすべきデータの依存関係に書き換えるため, コード移動と複写伝搬は交互に何回も行う必要がなく, 複写伝搬の書き換え規則を適用して得られた意味的制約と命令の順番が最適であるという最適条件 [5] を満たすモデルを生成することにより, 両方の最適化が適用されたコードを得ることができる.

書き換えを適用することによって項集合が “充足不能”, すなわちその項集合が表す意味的制約を満たすコードが存在しなくなる可能性もある. そのような例として次のコードを考える.

```
0: read X
1: Y := X + 1
2: Z := Y
3: A := X + 4
4: Y := A + 2
5: B := Z + Y
6: write B
```

このプログラムの制約を表す項集合のうち,df 項だけを示す.

$$df(0,1,\mathtt{X}),\ df(0,3,\mathtt{X}),\ \underline{df(1,2,\mathtt{Y})},\ \underline{df(2,5,\mathtt{Z})},$$
$$df(3,4,\mathtt{A}),\ df(4,5,\mathtt{Y}),\ df(5,6,\mathtt{B})$$

このとき,$copy(2,\mathtt{Z},\mathtt{Y})$ であることと下線部より, この項集合は複写伝播の規則を用いて次のように書き換えられる.

$$df(0,1,\mathtt{X}),\ df(0,3,\mathtt{X}),\ df(1,2,\mathtt{Y}),\ df(1,5,\mathtt{Y}),$$
$$df(3,4,\mathtt{A}),\ df(4,5,\mathtt{Y}),\ df(5,6,\mathtt{B})$$

ここで注目してほしいのは,$df(1,5,\mathtt{Y})$, $df(4,5,\mathtt{Y})$ である. 同じノードの同じ変数 (5 の Y) に対して複数の定義 (1 と 4 の Y の定義) が 5 に到達しなければならないが, このコードに分岐はないためそれは不可能である. したがって, この制約を満たすコードは存在しない.

項集合 S で表される意味的制約が充足可能かどうか, すなわちそれを満たすコードが存在するかどうかを調べるには,S 中に正しくないデータの依存関係が含まれているかどうかを調べればよい. なぜなら,S はオリジナルコードから得られた項集合 S_0 のうち,df, cf, $assign$, $cassign$, $copy$, $stmt$ だけを書き換えることにより得られた項集合であり,df 以外の項は充足不可能性を導くことはないからである.

定義 4.3 以下の条件を満たすとき, 項集合 S が充足不能と判断される. 記述を簡単にするために,$t \in S$ を t,$t \notin S$ を $\neg t$ と書くこととする.

$$\exists d \exists u \exists x (df(d,u,x) \wedge \forall p \neg cf(d,p)$$
$$\wedge \exists n (\exists n'\, df(n,n',x) \wedge order(d,n) \wedge order(n,u))$$

ここで $order$ は次のように定義される関係である.

$$order(p,q) \stackrel{\text{def}}{\Longleftrightarrow}$$
$$\exists x (df(p,q,x) \vee \exists r (df(p,r,x) \wedge order(r,q)))$$

直感的には,$order(p,q)$ は S を満たすモデルには p から q へのパスが存在しなければならない, ということを表す.

この条件は d から u へ到達するパスの間 x は定義されてはいけないということと,d と u の間に x を定義するノードが存在しなければならないという矛盾する要件が含まれているということを述べている.

4.2 書き換え規則の停止性

この節では, 定義 4.1 の書き換え規則が停止することを示す. そのために, 項集合間に順序関係を定義する.

定義 4.4 項集合 S と S' が与えられたときに, これらの間に順序関係 $\succ$ を次のように定義する.

$$S \succ S' \Leftrightarrow cost(S) > cost(S')$$

ここで, 関数 $cost$ は項集合を引数に取り, 自然数を返す関数である.$cost(S)$ は次のように計算される.

S 内の全ての $assign$, $cassign$, $copy$ に対し, 下に示すようにコストを付け, それらの総和を $cost(S)$ とする.

定数参照	...	1
変数参照	...	5
演算+, -	...	10
演算*	...	20
演算/	...	30
演算**	...	40

例えば,$assign(n, \texttt{X}, \texttt{Y*2})$ に対しては, 演算 $*\ldots 20, \texttt{Y}$ の参照 $\ldots 5, 2$ の参照 $\ldots 1$ となるので, この項に対するコストは $20+5+1=26$ となる.

この定義を用いて, 書き換え規則が停止することを簡単に示すことができる.

定理 4.2 任意の項集合 S に対し, 無限書き換え列 $S \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \ldots$ は存在しない.

証明概略 定義 4.1 の書き換え規則によって $S \stackrel{(i,\theta)}{\Longrightarrow} S'$ のように関係付けられる任意の S, S' に対して,$S \succ S'$ が成り立つことは容易に確かめられる. 定義 4.4 の順序関係 $\succ$ は well-founded であることから, 無限書き換え列 $S \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \ldots$ は存在しない. ■

4.3 書き換えによる最適化

この節では, 定義 4.1 の書き換え規則を用いて最適化を行う方法を述べる.

まずオリジナルコード π から手続き 3.1 により, 制約を表す項集合を求める. それを S_0 とする. その項集合に定義 4.1 の書き換え規則を適用する. 書き換えの適用は書き換え規則の番号 i と規則中の自由変数への代入 θ の組 (i, θ) によって具体化された書き換え規則によって行われる.

最適化は**書き換え木**を作成することで行われる. 書き換え木は S_0 を根とし,S_0 から適用できる全ての書き換え (i, θ) に対して子を作成していく. もし子が定義 4.3 により充足不能と判断されたならその子を削除する. 全ての子に帰納的にこの手続きを適用する. どの (i, θ) に対しても書き換え規則が適用できなくなるか, 全ての子が充足不能と判定されたら書き換えを停止する.

こうして得られた書き換え木の葉の中から定義 4.4 の順序関係において最も小さい項集合 S_{opt} を選ぶ. 項集合 S_{opt} によって表される意味的制約を時間論理式による記述に変換し,[5] の手法を用いてこの意味的制約と最適条件を満たすモデルを生成することによって最適コードを得る.

5 まとめと今後の課題

本論文では [5] において提案したコンパイラのコード最適化を時間論理のモデル生成で行うという手法の拡張を述べた. [5] ではオリジナルコードのデータの依存関係を意味的制約として時間論理式で記述し, それを満たすモデルを生成するという手法であるため, データの依存関係を変えない最適化しか扱うことができなかった. 本論文では時間論理式で表していた意味的制約を項集合で表し, その項集合の書き換えることでデータの依存関係を変えるような最適化を扱うという手法を提案した.

本論文の提案する書き換えによる最適化は, コードを直接書き換えるのではなく, 意味的制約を書き換えるというアプローチであるため, 実際の命令の順番に依存することなく書き換えを適用することができるという利点がある.

本論文では最も効率の良いコードを得るために, オリジナルコードから得られる意味的制約に対し, あらゆる書き換え規則を試し最も良いものを選ぶという, 全探索による手法を提案した. これは書き換え規則の適用順番によって得られるコードの効率が変わるためである. しかし, 実際に全探索を行うのは現実的には困難であるので, 適用するルールの選択に関するヒューリスティクスを考案するのは今後の重要な課題である.

また, 本論文で提案した項集合書き換えによって扱うことのできる最適化は多くの最適化技法の一部であるので, もっと多くの最適化技法を項集合書き換え規則によって表現することも今後の課題である.

参考文献

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

[2] U. Assmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *CC '96: Proceedings of the 6th Interna-*

tional Conference on Compiler Construction, pp. 121–135. Springer-Verlag, 1996.

[3] David Lacey and Oege de Moor. Imperative program transformation by rewriting. *Lecture Notes in Computer Science*, Vol. 2027, pp. 52+, 2001.

[4] 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.

[5] 伊藤宗平, 萩原茂樹, 米崎直樹. 最適条件の時間論理記述を用いたモデル生成器によるコード最適化. 日本ソフトウェア科学会第 20 回大会論文集, 2003.