リモート参照が可能な循環属性文法による怠けたコード移動の実現

Implementation of Lazy Code Motion with Circular Remote Attribute Grammars

立川 英† 佐々木 晃† 佐々 政孝†
TACHIKAWA Suguru SASAKI Akira SASSA Masataka
† 東京工業大学 大学院情報理工学研究科 数理・計算科学専攻
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology
{suguru.tachikawa, sasaki, sassa}@is.titech.ac.jp

goto 文のような，非構造的ジャンプを持つ言語に対する最適化を属性文法で定式化することは，従来困難であった．それは，既存の属性文法では離れたノードの属性間の関係の記述が難しく，かつジャンプ命令によって属性の依存関係に循環が生じる場合に評価ができなかったからである．そこで我々は，遠隔ノードの属性参照が可能で，かつある条件下での属性依存関係のサイクルを許すという循環リモート属性文法を提案した．本研究では、この拡張した属性文法を用いて非構造的ジャンプを持つ言語に対する，部分冗長性除去の最適化器の実装を行った．データフロー方程式に基づき，**怠けたコード移動** の可読性の優れた記述に成功した．

1 はじめに

属性文法 (*Attribute Grammars*) [1] による記述から自動生成されたコンパイラは，コンパイル速度やコンパイル時に必要となるメモリ量の点で手書きのコンパイラに若干劣るものの，属性文法による記述は可読性が高く，デバッグや保守の作業を能率化することができるという利点がある．我々は，属性文法を用いることでコンパイラの全フェーズを形式的に記述することを目標に，フロントエンドの生成系 *Rie* [2] とバックエンドの生成系 *Jun* [3] を開発し，これらを用いて C 言語 (サブセット) のコンパイラを作成してきた．

従来，属性文法では，離れたノードの属性依存関係の記述が困難であり，また属性の依存関係に循環が生じる場合に属性評価が不可能であった．このため，goto 文のような非構造的ジャンプを許すような言語構造に対応できず，また属性依存関係に循環が生じる最適化の実現は困難であった．そこで我々は，**リモート参照が可能な循環属性文法** (*Circular Remote Attribute Grammars*) [4] を提案した．

今回そのシステムを利用して，goto 文を許す言語構造に対して，属性依存関係に循環が生じる最適化の 1 つである **部分冗長性除去**(*Partial Redundancy Elimination*) [5] のアルゴリズムとして代表的な**怠けたコード移動** (*Lazy Code Motion*)[6] を実装することができたので，その記述性の高さと性能について紹介する．

2 リモート参照可能な循環属性文法

属性文法は文脈自由文法と静的意味とを統合した定式化である．文脈自由文法の各文法記号に意味を表す属性を付加し，その属性に対する値の決め方 (意味規則) を生成規則に付随させることで，構文と意味とを統一的に扱う．与えられた入力に対して構文解析を行ない，意味規則に従って，構文木上のノードの属性値を計算することを，属性評価という．また，属性評価を行うプログラムを属性評価器という．

従来の属性文法では，次の理由により，goto 文のような非構造的ジャンプを許すような言語に対する最適化器を記述することは難しかった．

1. あるノードから離れたノードの属性，すなわち親子あるいは兄弟ノード以外の属性を参照できない (リモート属性が参照できない)
2. データフロー方程式のように循環を含む計算を表現できない

従来 1 の問題に対しては，リモート属性文法などの拡張，2 の問題に対しては，Farrow の有限帰納属性文法 [7] などの属性文法の拡張があった．しかし，両者を同時に許す属性文法の拡張はなかった．

文献 [4] では，これら 2 点の問題を解決する属性文法の拡張である**リモート参照可能な循環属性文法** を提案するとともに，効率の良い評価器の実装法を示した．

従来の属性文法では，属性依存関係にサイクルがある場合，循環定義となって意味は定まらないと見なされていた．しかし，Farrow は，属性依存関係に循環があっても適当な条件のもとでは，依存関係に従って循環定義される属性値を繰り返し評価することで，値が一意に定まることを示した．リモート参照可能な循環属性文法は，この Farrow が示した性質をリモート属性参照を許す属性文法に適用したものである．

サイクルのある属性文法が収束する 1 つの十分条件は次のようになる．

Farrow の収束条件　A を属性依存関係のサイクルに含まれる属性の集合，F を属性に対する意味規則によって定まる意味関数の集合とするとき，

1. A 中の全ての属性のそれぞれの値域は完全半順序集合になっている．
2. F 中の全ての関数が単調で収束する．

2 については，$s_0 < s_1 < s_2 < \ldots$ に対して

$$f(s_0) < f(s_1) < f(s_2) < \ldots < f(s_k) = f(s_{k+1})$$

であることを意味する．ここで，$s \in A$，$f \in F$ であり，s_i の i は，評価の回数である．このとき $f(s_k) = f(s_{k+1})$ を最小不動点という．すなわち，その属性評価ごとの各回の属性値の変化のしかたが数学的な意味で単調かつ有界であれば，有限回の属性評価で属性値が収束する．

この条件は，リモート参照可能な循環属性文法においては，後述 (図4) するようなリモート参照によって起きる循環関係に対しても適用される．次の3節で示すように，データフロー方程式を上記の循環属性文法に適用する場合，フロー関数は大きく (あるいは，小さく) なる向きに単調であるため，最大解 (あるいは，最小解) が求められる．

3　怠けたコード移動の実装

我々は，怠けたコード移動を *Squeak Smalltalk* [8] を用いて実装した．本節では，その記述に基づいて，怠けたコード移動とその実現方法を解説する．

3.1　怠けたコード移動

怠けたコード移動とは，部分的に冗長な計算を除去するアルゴリズムの 1 つであり，レジスタ圧力をなるべく抑えるように工夫したものである [?]．怠けたコード移動は，まず前向きや後ろ向きのデータフロー解析を制御フローグラフに沿って行い，コードを移動するポイントを求める．例えば，以下の *Dsafe* は，制御フローグラフ上で後続ノードの方にある計算をそのポイントまで移動してもプログラムの意味が変わらないポイントを求めるためのデータフロー方程式である．

$$Dsafe(n) = \begin{cases} \textbf{false} & \textbf{if} \quad n = \text{end} \\ Used(n) \vee Transp(n) \wedge \prod_{m \in succ(n)} Dsafe(m) & \textbf{otherwise} \end{cases}$$

各ノードでは，注目している式が使われるかどうか (*Used*) とその式のオペランドへの代入文がないか (*Transp*) を調べ，この情報を用いて *Dsafe* などの 9 つのデータフロー方程式を解く．怠けたコード移動のデータフロー情報は，ビットベクトルによって計算できる．最後に，指定されたノードの計算を移動し，部分的に冗長な計算を除去する．

3.2　データフロー情報の記述

通常，手書きのプログラムでは制御フローグラフを作成してから各データフロー方程式を解く手順で実装を行うが，属性文法で実装する場合には制御フローグラフの解析を行う必要はない．各構文におけるデータフロー情報の意味規則を記述すれば，属性評価器が自動的に属性の評価順序を決めてくれる．

図1に簡単な例として，if 文に対する意味規則の記述例を示す．図 1 の各ノードの *in*，*out* はそれぞれその構文木のノードの入口と出口におけるデータフロー情報 (継承属性，合成属性) を表し，矢印は属性依存関係を表す．図 1 は，前向きの解析に従った属性文法記述である．if 文では，その条件 (*exp*) により真 (*stms1*) と偽 (*stms2*) の二つに分岐する．*if* ノードはその親ノードから *if.in* に属性を受け取り，まず条件文である *exp* ノードの継承属性 *exp.in* に渡す (1)．*exp* ノードでは，そのノードの意味規則に従って属性の受け渡しが行われ，*exp.out* に返ってくる．これを *stms1* ノード (2) と *stms2* ノードに渡す (3)．各 *stms* ノードでは，*stms* ノードの意味規則に従って属性の受け渡しが行われ，*stms1.out*，*stms2.out* が返される．それを *if.out* にまとめ (4)，親ノードに返す．

$$
\begin{aligned}
if \quad &\Rightarrow \quad exp \quad stms1 \quad stms2 \\
exp.in \quad &:= \quad if.in; \qquad (1)\\
stms1.in \quad &:= \quad exp.out; \qquad (2)\\
stms2.in \quad &:= \quad exp.out; \qquad (3)\\
if.out \quad &:= \quad stms1.out \cap stms2.out; \qquad (4)
\end{aligned}
$$

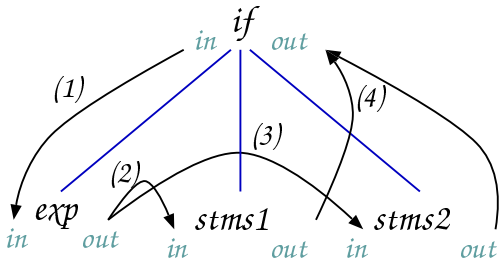

図 1: if 文の意味規則の例とその構文木

3.3 循環評価

do-while 文 (while 文，for 文) は制御フローグラフ上でループを含む．制御フローラフ上でループを含むプログラムに対して，怠けたコード移動を属性文法で実現する場合には，そのデータフロー方程式の解析で属性依存関係に循環が生じる．

図 2 は do-while 文のデータフロー情報の意味規則の記述例である．図 2 は，前向きの解析に従った属

$$
\begin{aligned}
do \quad &\Rightarrow \quad stms \quad exp \\
stms.in \quad &:= \quad do.in \cap exp.out; \qquad (5)\\
exp.in \quad &:= \quad stms.out; \qquad (6)\\
do.out \quad &:= \quad exp.out; \qquad (7)
\end{aligned}
$$

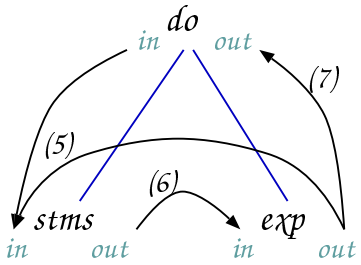

図 2: do-while 文の意味規則の例とその構文木

性文法記述である．do-while 文では，条件 (exp) が満たされる限りループの中身 ($stms$) が繰り返し実行される．このため制御は，exp のあとループを抜ける路と $stms$ に戻る路に分かれる．$stms$ に戻る路に移る．$stms$ に戻る路では，怠けたコード移動のデータフロー方程式に従って，初めてループに入る路からくるデータフロー情報との集合積か集合和を取る．このとき，集合積ならば最小解が求まり，集合和ならば最大解が求まる．

リモート参照可能な循環属性文法の評価器生成系 Jun では，図2 のような意味規則を評価する時に，属性依存関係のサイクルを自動的に検出する．そして，ユーザによって設定された初期値と収束条件に従って，収束するまで自動的にサイクル内を繰り返し評価する．

3.4 リモート参照

goto 文のような非構造的なジャンプを伝統的な属性文法で扱うためには，離れたノード間にまたがる中間木のノードすべてを経由して属性の受け渡しを行なわなければならない．このような記述は煩雑であり，また属性評価に多くの時間を要する．離れたノードの属性をリモート参照することは，このような問題点を回避でき，大変に有効である．

しかし，属性をリモート参照する時，その参照によって依存関係にサイクルが生じる場合がある．まずは，図3にサイクルが生じない例を示す．図 3 の

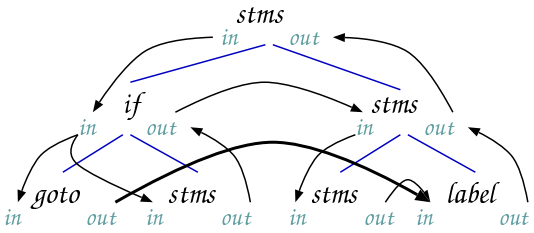

図 3: サイクルの生じないリモート属性の参照

矢印は属性の受け渡しを示す．サイクルが発生しない場合，Jun は中間木中の全属性を 1 回だけ評価すればすむ．一方，図4 は属性をリモート参照することでサイクルが発生する例である ($label \rightarrow stms \rightarrow stms \rightarrow if \rightarrow goto \rightarrow label$)．この場合は，循環評価が必要となる．

図 5 は前向きの解析で属性のリモート参照を行なう属性文法記述の例である．リモート参照は，$goto$ ノードと $label$ ノードに意味規則の記述が必要である．リモート属性を参照するには，$ref.goto.out$ のよ

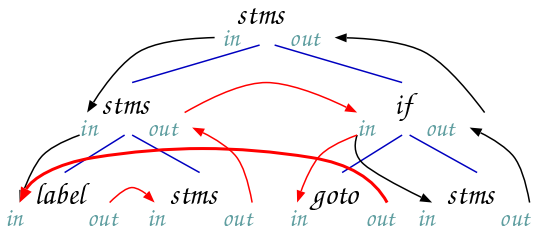

図 4: サイクルの生じるリモート属性の参照

$$label \Rightarrow stms$$
$$stms.in := ref.goto.out \cap label.in; \quad (8)$$
$$label.out := stms.out; \quad (9)$$

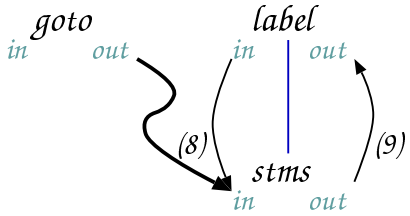

図 5: label 文の意味規則の例とその構文木

うに記述する．(8) の意味規則は，label 文の直後の文の入口の情報 *stms.in* は，そのラベルへ飛んで来る goto 文の情報と label の入口の情報の集合積に等しいことを表す．

ここで，goto と label との関係について述べると，同じラベルのついた goto はプログラム中に複数存在する可能性があるが，label は唯一である．そこで，前向きの解析では，ある *label* ノードに対応した *goto* ノードは一つとは限らないため，複数の対応する *goto* ノードの属性を集合和もしくは集合積でまとめなくてはならない．一方，後ろ向きの解析では，ある *goto* ノードに対応した *label* ノードは一つであるため，そのまま属性を受け渡せばよい．

4 議論

我々は，データフロー解析器を属性文法を用いて簡単に実装することを目指した．本節では，主にその記述性について議論する．また，一般に自動的に生成されたプログラムは手書きのコンパイラよりも効率が悪いとされている．しかし，我々の評価器は効率があまり悪くならない．それについても議論する．

4.1 記述性

以下では，属性文法を用いた我々の実装と，属性文法を用いず一般のプログラミング言語でデータフロー解析器を実装する場合とを比較して議論する．

属性文法を用いず実装する場合には，まず最初に，プログラムの大域的な流れを把握するために制御フローグラフを作成する必要がある．しかし，属性文法を用いて実装する場合には，制御フローグラフの作成は必要でない．これは，属性文法で解析するのが木構造になっているため，各構文のノードの親と子の関係を見て解析することができるためである．

また，データフロー解析を行なう際にも，属性文法を用いて実装する場合には，各構文に従ってデータフロー情報の関係を記述するだけでよい．しかし，属性文法を用いず実装する場合には，各ノードのデータフロー情報の関係を記述するだけでなく，制御フローグラフに沿って全体のノードのデータフロー情報をどういう順番で計算するのかを，手で書き下す必要がある．

そして，属性依存関係にサイクルが生じた場合の解析を属性文法を用いずに実装する場合には，データフロー方程式ごとに，その初期値や収束条件を定め，ループで回しながら解析するように，手で書き下さなくてはならない．しかし，属性文法を用いた我々の実装では，評価器がサイクルを自動的に検出し，自動的に評価を行なう．実際に記述するのは，サイクルの中の属性の初期値と収束条件だけですむ．

以上をまとめると，次の表のようになる．

	属性文法を用いない場合	属性文法を用いた場合
制御フローグラフ	必要	不要
データフロー解析	手で書き下す必要がある	構文に従ってデータフロー情報の関係を記述すればよい
サイクルの評価	手で書き下す必要がある	評価器が自動的に検出し評価する

このように，属性文法を用いた我々の実装は，可読性がよく高潔な記述であると言える．また，複雑な実装は，プログラムのデバッグ作業をも困難にする．我々の実装では，属性文法デバッガ [9] を用いることで，デバッグ作業も容易に行なえる．これにより，プログラマの負担が相当に軽減されたと言える．

4.2 計算量

我々の実装では，怠けたコード移動は，次に挙げる各フェーズに分かれている．

1. 空のノードを挿入する (危うい辺の除去)

2. 計算式の集合をビットベクトル化する

3. 9 つのデータフロー方程式を計算する

4. 冗長なコードを移動する

5. 挿入した空のノードを除去する

1，2，4，5 のフェーズでは，中間木のノード数を N とすると，計算量はすべて中間木のノード数に比例し，$O(N)$ である．また，3 のデータフロー方程式の計算のフェーズでは，サイクルが発生する場合，循環評価が必要である．この場合，深さ[1]を d とすると，$O(N * d)$ である．各属性値が収束するまで属性評価を繰り返すため，計算量は単純に中間木のノード数に比例しない．詳しくは，文献 [4] 参照.

4.3 問題点

本来，危うい辺の除去は，制御フローグラフ上で，2 つ以上の後続ノードを持つノードから 2 つ以上の先行ノードを持つノードへの辺に空のノードを挿入するだけでよい．しかし，今回用いた文献 [?] の怠けたコード移動では，あるノードが 2 つ以上の先行辺を持つ場合は，そのすべての先行辺に空のノードを挿入しなければならない．これは，ノード数の増加を引き起こし，計算量の悪化につながってしまう.

5 おわりに

本研究では，怠けたコード移動をリモート参照可能な循環属性文法を用いて実装した．まず，この実装の経験から得られた知見を述べる．各データフロー情報に従った各構文の依存関係を考えるとき，木構造であるため視覚的にイメージしやすく，またそのイメージをそのまま意味規則として記述すればよいので，非常に記述しやすく感じた．また，その実装に多少の変更を加えたいときにも，その実装が構文ごとの記述であるために部分的に修正することができた．さらに手書きのデータフロー解析器では困難であると思われるサイクル部分の実装を行なわなくてよく，サイクルを自動的に検出して評価してくれることの恩恵は大きいと感じた.

最後に，今後の課題を述べたい．怠けたコード移動は，この後改良され，基本ブロックを用いたアルゴリズム [10] が提案されている．この改良によって，危うい辺の除去のために空のノードを挿入する際の問題点は解決されている．しかし，属性文法で基本ブロックを用いるには，いくつかの課題がある．これは，基本ブロックを部分木としてまとめる処理が必要だからである．我々は，属性文法向けに部分冗長性除去の最適化を効率よく行なうためのアルゴリズムは，通常のアルゴリズムと別なものであると感じている．今後，これを考えていきたいと思う.

[1] フローグラフにおいて制御の流れに逆行する上向きの有向辺の連鎖の最大の長さを深さと呼ぶ

参考文献

[1] P. Deransart, M. Jourdan, B. Lorho. Attribute Grammars. *Lecture Notes in Computer Science.*, Vol.323, Springer, 1988.

[2] M. Sassa, H. Ishizuka, and I. Nakata. Rie, a Compiler Generator Based on a One-pass-type Attribute Grammar. *Software Practice and Experience.*, pp. 229–250, No.3, Vol.25, 1995.

[3] M. Sassa. Rie and Jun: Towards the generation of all compiler phases. *3rd int. Workshop on Compiler Compilers.*, volume 477 of Lecture Notes in Computer Science, pp. 56–70, Springer–Verlag, 1991.

[4] A. Sasaki, M. Sassa. Circular Attribute Grammars with Remote Attribute References. *Third Workshop on Attribute Grammars and their Applications.*, pp. 125–139, July 2000.

[5] Morel, E., and Renvoise, C. Global optimization by suppression of partial redundancies. In *Commun. of the ACM 22*, pp. 96–103, 1979.

[6] J. Knoop, O. Ruthing, and B. Steffen. Lazy Code Motion. *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pp. 224–234, 1992.

[7] R. Farrow. Automatic Generation of Fixed-Point-Finding Evaluator for Circular, but Well-Defined, Attribute Grammars. *ACM SIGPLAN 86 Symp. on Compiler Construction* , pp. 85–98, 1986.

[8] D. Ingalls, T. Kaehler, J. Maloney, S. Wallance, and A. Kay. Back to the future - The story of Squeak, a practical Smalltalk written in itself. *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* , pp. 318–326, 1997.

[9] Y. Ikezoe, A. Sasaki, Y. Ohshima, K. Wakita, and M. Sassa. Systematic Debugging of Attribute Grammars. *Ducasse, M. (ed.) Proc. AADEBUG 2000 -4th International Workshop on Automated Debugging. Munich, Germany* , pp. 235–240, Aug 2000.

[10] J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion: Theory and Practice. *ACM Transactions on Programming Languages and Systems* , 16(4): pp. 1177-1155, July 1994.