疎な要求駆動型データフロー解析

滝本宗宏[†1] 福岡岳穂[†2]
佐々政孝[†3] 原田賢一[†4]

プログラム解析やコード最適化においては，データフロー解析がよく使用される．この解析は，目的とする情報をプログラム全体から収集することを仮定しているので，プログラムの一部分だけを解析したい場合や，一部の解析結果を用いてプログラムを変形し，さらに変形後のプログラム上で他の部分の解析を行うことを繰り返す場合には，余分なコストが必要になる．このような場合には，情報を必要とするプログラム点について，そこに影響を及ぼす部分だけを解析する要求駆動型のデータフロー解析が有効である．しかしながら，一般のデータフロー解析には，1 ワード分の情報を並列計算するビットベクタ法や，解析に不要な部分を訪問せずに解析を行う疎な解析法が考案されているのに対して，要求駆動型のデータフロー解析については，これまで高速化手法の提案が行われていない．

本稿では，解析に不要なプログラム点を訪問しない疎な要求駆動型のデータフロー解析を提案する．あるプログラム点での解析結果に基づいてプログラムを変形した後に，他のプログラム点についての解析を行おうとすると，その変形に伴って，解析が不要な部分も変わる可能性がある．本手法では，プログラムの深さに相当するランク情報を用いて不要なプログラム点を動的に計算するので，プログラム変形が行われた場合でも，変形後のプログラムを反映した疎な解析が実現できる．

Sparse Demand-Driven Dataflow Analysis

Munehiro Takimoto,[†1] Takeaki Fukuoka,[†2] Masataka Sassa[†3]
and Ken'ichi Harada[†4]

Demand-driven dataflow analysis (DFA), which propagates queries as to a specific dataflow fact at a program point, is more efficient than exhaustive DFA if it is used for to interactive program analyzing tools or incremental code optimizations. However, though the exhaustive DFA can be implemented by efficient solving methods like sparse propagation method, there is no such efficient method proposed for demand-driven analysis. This is because, in case where demand-driven DFA and program transformation based on its result are repeatedly applied, sparse information cannot be reflected by results of previous transformation.

So we propose sparse demand-driven DFA which computes sparse information on demand. Our technique enables avoiding unnecessary propagations to some program points by propagating queries along dominator tree edges. To propagate queries on dominator tree edges, validity checking of the propagation is needed. For implementation of the efficient checking, we introduce ranks which mean depth of node from start node in CFG. If the ranks of target program points to be propagated are within range between current program point's rank and its dominator's rank, the shortcut propagation are suppressed.

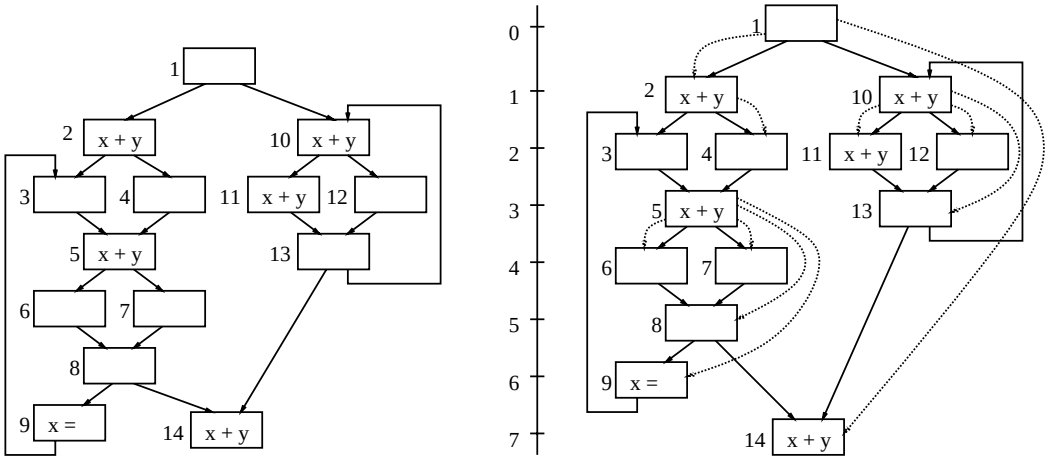
We show the efficiency of our technique by comparing experimental results applying a traditional demand-driven DFA to available expressions.

†1 東京理科大学理工学部情報科学科
Department of Information Sciences, Faculty of Science and Technology, Tokyo University of Science
†2 株式会社 管理工学研究所
Kanrikogaku Kenkyusho, Ltd.
†3 東京工業大学情報理工学研究科数理・計算科学専攻
Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†4 慶應義塾大学理工学部情報工学科
Department of Information and Information Science, Faculty of Science and Technology, Keio University

1. はじめに

現在，コンパイラのコード最適化や並列化においては，**データフロー解析**が広く用いられている．また，デバッガやソフトウェア開発支援ツールにも，性能の向上のためにデータフロー解析が用いられている．

データフロー解析の有効性が認められるにつれて，1 種類あるいは何種類かのデータフロー解析が複数回適用される場合が増えてきた．その主な理由は，次のとおりである．



(a) 制御フローグラフ
Control flow graph

(b) 短絡辺を加えた制御フローグラフ
Control flow graph with short-cutting edges

図 1 要求駆動型データフロー解析の例
Fig. 1 Sample of demand-driven data flow analaysis

多重コード最適化 ：最適化コンパイラは，性質の異なる種々のプログラム変形を行うことが多く，それぞれの目的に応じたデータフロー解析を行う必要がある．

情報の無効化 ：プログラム変形によって，プログラム中のデータフローが変化することがある．その場合には，以前計算したデータフロー情報が使用できなくなることがあるので，データフロー情報の更新あるいは再計算が必要である．

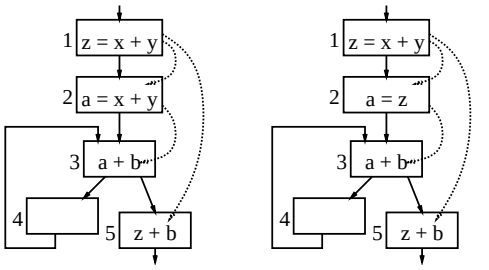
ユーザによる編集 ：ユーザによるプログラムの編集は，編集前のデータフロー情報を非無効化することがあるので，編集のたびにデータフロー情報を再計算する必要がある．

データフロー解析を複数回適用する場合には，その解析の効率が重要な問題になる．現在用いられているデータフロー解析は，すべてのプログラム点におけるデータフロー情報を計算する**網羅型**（exhaustive）のものが多く，高速な計算法を用いた場合でも，多くの計算時間と資源が必要とされる．

網羅型データフロー解析に対して，要求に応じて必要なプログラム点でのデータフロー情報を収集する**要求駆動型**（demand-driven）データフロー解析[6]がある．要求駆動型データフロー解析は，データフロー情報に関するある**事実**（fact）y が，プログラム点 v における網羅型データフロー解析の結果に含まれるかどうかを決定すること，と定義できる．要求駆動型データフロー解析の結果は，プログラム点 v におけるデータフロー情報 y の真偽を意味する**クエリ** $q =< y, v >$ に対して，$true$ あるいは $false$ として得られる．v において q の真偽が直接決定できないときには，網羅型データフロー解析の方向と逆方向にクエリを伝播させ，それらの結果から v における解を得る．

例： 要求駆動型データフロー解析の例として，図 1 (a) に示す**制御フローグラフ**（Control Flow Graph，以降 CFG と呼ぶ）の節 14 において，`x+y` が**利用可能**（available）かどうかを考える．式 e が利用可能であるというデータフロー情報を $avail(e)$ で表すと，クエリ $< avail(\mathtt{x+y}), 14 >$ については，節 14 自身で真偽を決定することができないので，先行節 8，13 にそれぞれ $< avail(\mathtt{x+y}), 8 >$，$< avail(\mathtt{x+y}), 13 >$ として伝播させる．$< avail(\mathtt{x+y}), 8 >$ は，さらに節 6，7 を経て節 5 まで伝播される．同様に，$< avail(\mathtt{x+y}), 13 >$ は，節 11 と 12 に伝播され，節 12 のクエリはさらに節 10 まで伝播される．最終的に節 5，10，11 におけるクエリはすべて $true$ となるので，クエリ伝播は終了し，$< avail(\mathtt{x+y}), 14 >= true$ が得られる． ∎

要求駆動型データフロー解析は，あるプログラム要素だけについてコード最適化を行ったり，ユーザの要求によって指定されたプログラム要素のデータフロー情報を表示したりする場合に有効である．しかしながら，要求駆動型データフロー解析の適用範囲がプログラム全体に及ぶ場合には，その効果は小さくなる．また，複数のプログラム点における情報を同時に得よう



(a) 共通部分式の除去 Eliminating common subexpressions (b) 新たな共通部分式の発見 Exposing new common subexpressions

図 2 疎な解析の問題
Fig. 2 Problem of sparse analysis

とすると，同じプログラム点における情報を何度も集めなければならない場合が生じ，要求駆動型データフロー解析の方が効率が悪くなってしまうこともある．

従来，網羅型データフロー解析に関しては，多くの高速化手法が提案されてきた．それらの手法は，次の 2 つに大別できる．

(1) **ビットベクタ**（bit-vector）法：データフロー情報の集合をビットベクタで表現することによって，1 ワード分のデータフロー計算を 1 度にまとめて行う[2),5),9)]．

(2) **疎な解析**（sparse analysis）法：解析に影響を及ぼさないプログラム点に対しては，データフロー情報を伝播させないようにする[3),8),16)]．

要求駆動型データフロー解析は，特定のプログラム要素についてのデータフロー情報を求めることが主目的であるので，その解析にビットベクタ法を採り入れても，高速計算の特徴を活かすことができない．また，要求駆動型データフロー解析に基づいてプログラム変形を繰り返す場合，疎な解析では，必要な伝播先の情報をそのたびに再計算する必要があり，余分なコストがかかる可能性がある．

例： 伝播先の情報を再計算しなければならない例として，図 2 (a) に示すプログラム上で，共通部分式の除去を行うことを考える．この図で，同じ式どうし，およびオペランドへの代入とそれを参照している式を含むプログラム点を点線矢印で結んでいる．CFG 節 5 の式 `z+b` について要求駆動型データフロー解析を適用したとき，クエリ $< avail(\mathtt{z+b}), 5 >$ は，点線矢印の出元節 1 の出口に直接伝播させることができる．その場合には $< avail(\mathtt{z+b}), 5 >= false$ という結果が得られる．一方，節 1 と 2 の `x+y` が共通部分式であることに注目すると，節 2 の右辺を節 1 の左辺 `z` で置き換えて，図 2 (b) のように変形し，さらに，節 2 の コピー代入 `a=z` を 節 3 に伝播させることによって， 節 3 では，新たに 式 `z+b` を得ることができる．この場合には，最終的に $< avail(\mathtt{z+b}), 5 >= true$ という結果が得られる．しかし，そのためには，直接伝播させることができるプログラム点を再計算しないと，節 3 に共通部分式 `z+b` があることを，見つけることができない． ∎

本稿では，伝播が不要になるプログラム点を解析時に計算し，不必要なプログラム点へのクエリ伝播を避ける疎な要求駆動型データフロー解析手法を提案する．

このようなクエリ伝播を行うために，CFG の各節 v と， v を直接囲むループのヘッダか，ヘッダが存在しない場合には，ループ内に存在して，ループの入口に最も近い v の支配節 d との間に辺を付加し，直接伝播させることが可能であることを明示する．プログラム全体も開始節をヘッダとする 1 つのループとみなして扱う．ここで，d を**短絡節**と呼び，付加した辺 (d, v) を**短絡辺**と呼ぶ．短絡辺は，どのようなデータフロー情報を扱うときにも，共通に使用する．

本手法では，クエリを短絡節に伝播させるか，それとも**先行節**（predecessor）に伝播させるかを動的に決定する．もし，節 v の短絡節 d と v の間のすべての実行パス上で，データの生成（gen）や無効化（kill）によってデータフロー情報が変化されないことが分れば，短絡辺に沿ってクエリを d まで伝播させることができる．そうでなければ，v の先行節へクエリを伝播させる必要がある．

節 v とその短絡節 d との間のすべての実行パス上でデータフロー情報が変化しないかどうかの検査には，**ランク**と呼ぶ CFG の開始節からの深さを表す値を利用する．これは，データフロー情報に変化を与える CFG 節のランクが，d のランクよりも大きくて，v のランクよりも小さい範囲に含まれるかどうかを調べることで近似的に求められる．

例： 図 1 (a) のグラフに，短絡辺（点線矢印で示す）を付加したグラフを図 1 (b) に示す．図 1 (b) において，節 14 のランクは 7，節 1 のランクは 0 であり，それらの間に，`x+y` あるいは `x=` が出現する節のランク 1，2，3，6 が含まれるので，節 1 と節 14 の間でデータフロー情報が変化することが分る．従ってこの場合には，節 14 から 1 への直接伝播は避けなければならない．同様に節 13 のランク 3 と節 10 のラ

ンク 1 の間には，x+y が出現する節のランク 2 が含まれるので，節 13 から 10 への直接伝播も避けなければならない．一方，節 5 のランク 3 と節 8 のランク 5 の間には，データフロー情報が変化する節のランクが含まれないことから，クエリは節 8 から節 5 に直接伝播させることができる．結果として，節 6，7 への伝播を避けることができる． ∎

ランクに基づく検査は，ランクを対応するビット位置に対応させたビットベクタを用いることによって，効率の良い実現ができる．

本手法は，次に示す手順に従って前処理を行ってから解析処理に移るものとする．

(1) 解析対象となる関数内に存在するループの入れ子構造を解析する．
(2) その入れ子構造を基に，ランク付けを行う．
(3) 同じくその入れ子構造を基に，CFG に短絡辺を付加する．
(4) ランクと短絡辺を付加した CFG に対して，要求駆動型データフロー解析を適用する．この際，ランク情報に基づいて，なるべく短絡辺に沿ったクエリ伝播を行うよう試みる．

本稿の以降の構成は次のとおりである．まず，第 2 章で，本手法が基にしている解析対象と基本用語を説明する．次に，第 3 章で，本手法が対象にしている要求駆動型データフロー解析について説明する．第 4 章では，クエリの伝播先を計算するために必要なランクの概念と計算法を示す．次に，第 5 章で，クエリの伝播先の候補を明示する短絡辺の加え方を示し，第 6 章で，実際に，どのようにデータフロー解析を行うか，データフロー方程式を示して説明する．第 7 章で，本手法の効果を実験によって示したのち，第 8 章で，関連研究を述べ，最後にまとめる．

2. 用語と記法

本手法の適用に際して，原始プログラム中で定義される各関数ついて，対応する CFG がすでに作成されているものとする．CFG は，途中に分岐をもたない連続した文からなる**基本ブロック**（basic block）を節とする節集合 N，基本ブロック間の制御の流れを辺とする辺集合 $E \subset N \times N$，特別な節である開始節 s と終了節 e からなる四つ組 $(N, E, \mathrm{s}, \mathrm{e})$ である．CFG 節 n について，その先行節集合 $\{n' | (n', n) \in E\}$ を $pred(n)$ で表し，後続節集合 $\{n' | (n, n') \in E\}$ を $succ(n)$ で表す．また，必ず CFG 節は，開始節から終了節までの実行パス上に存在するものとする．

CFG の開始節からある節 w へ至るすべての実行パスが節 v を含むとき，v は w を**支配する**（dominate）といい，v を w の**支配節**（dominator）[1),2),13)] という．またこのとき，v と w は**支配関係**（dominance relation）にあるという．支配関係は，推移律（transitive）が成り立つので，木構造によって表現することができる．この木を**支配木**（dominator tree）と呼ぶ．支配木において，節 x が節 y の直接の親であるとき，x は y を**即支配**（immediately dominate）するといい，x は y の**即支配節**（immediate dominator）であるという．本稿では，解析対象の CFG について支配木がすでに作成されているものとする．

CFG の循環構造は，深さ優先探索によって得られる**全域木**（spanning tree）上の祖先へ向かう辺（以降，**上昇辺**と呼ぶ）によって見つけることができる．さらに，CFG の循環構造は，上昇辺の先 dst が出元 src を支配するかしないかによって，それぞれ**可約ループ**（reducible loop）と**既約ループ**（irreducible loop）とに区別することができる．ここで，dst が src を支配する上昇辺は**戻辺**（back edge）と呼ばれる．

3. 要求駆動型データフロー解析

網羅型データフロー解析は，ある性質についての情報をプログラム全体から集めるという特徴をもつ．この情報を収集する過程は，**減少鎖性**（decreasing chain property）をもつ**完備束**（complete lattice）とその上での**単調関数**によって定義される．この関数は，**フロー関数**（flow function）と呼ばれる．

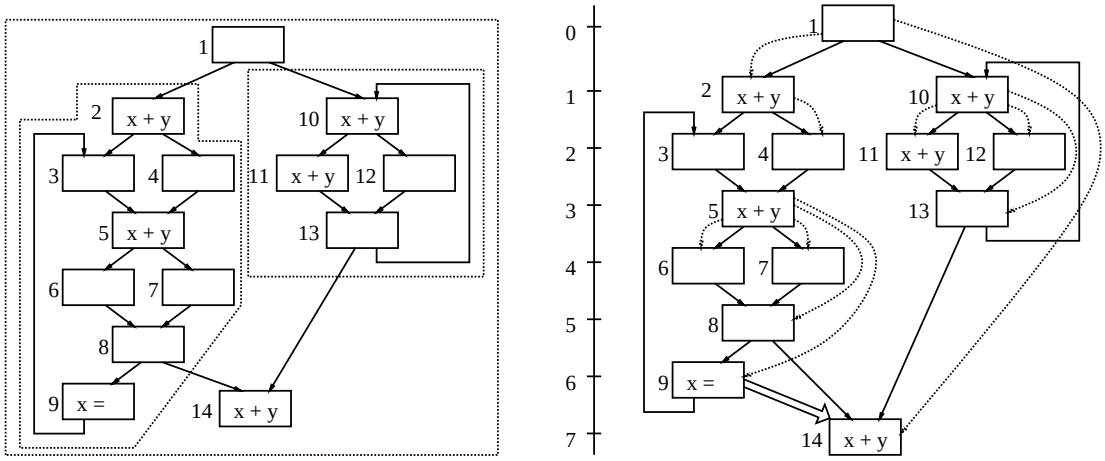
要求駆動型データフロー解析は，プログラム点 n におけるデータフロー情報 y が，$true$ か $false$ を決定するために，クエリ $q =< y, n >$ を網羅型データフロー解析とは逆向きに伝播させることで行われる．要求駆動型データフロー解析は，フロー関数の代わりに，フロー関数の逆関数に相当する**逆フロー関数**（reverse flow function）[6),7)] を使って定義される．完備束 L と単調なフロー関数 $h : L \to L$ とすると，逆フロー関数 h^r は，y を，$y \sqsubseteq h(x)$ を満たす最小元 x へ写像する関数として，次のように定義される．

$$h^r(y) = \sqcap\{x \in L : y \sqsubseteq h(x)\}$$

そのような元 x が存在しない場合は，$h^r(y) = \top$ となる．また，h が $\sqcap$ に関して分配則を満たす場合，h と h^r は，次の関係をもち，要求駆動型データフロー解析の結果が，網羅型データフロー解析の結果に一致することが知られている．

$$y \sqsubseteq h(x) \Longleftrightarrow h^r(y) \sqsubseteq x$$

プログラム点 n におけるフロー関数 h_n の逆フロー



(a) CFG とループ
CFG and its loop regions

(b) 変形後
Transformed CFG

図 3 仮想辺と短絡辺の付加
Fig. 3 Inserting virtual edges and short-cutting edges

関数を h_n^r とすると，前向き網羅型データフロー解析に相当する要求駆動型データフロー解析の方程式は，次のように示すことができる．

$$< y, \mathrm{s} > \leftrightarrow \begin{cases} true & \text{if } y = \bot \\ false & \text{otherwise} \end{cases} \quad (1)$$

$$< y, n > \leftrightarrow \bigwedge_{m \in pred(n)} \begin{cases} false & \text{if } h_m^r(y) = \top \\ true & \text{if } h_m^r(y) = \bot \\ < h_m^r(y), m > & \text{otherwise} \end{cases} \quad (2)$$

方程式 (2) の右辺に示すように，プログラム点 n の結果が $true$ あるいは $false$ と決定できない場合は，先行節へのクエリ伝播の結果による．

例： 図 1 (a) について**利用可能性**（availability）を解析する場合，各節の逆フロー関数は次のようになる．

節 n	逆フロー関数
2,5,10 11,14	$h_n^r(y) = \begin{cases} \bot & \text{if } y = avail(\mathtt{x+y}) \\ y & \text{otherwise} \end{cases}$
9	$h_n^r(y) = \begin{cases} \top & \text{if } y = avail(\mathtt{x+y}) \\ y & \text{otherwise} \end{cases}$
その他	$h_n^r(y) = y$

`x+y` が出現する節 2，5，10，11，14 では，$h_n^r(y) = \bot$ となり，データフロー方程式上のクエリが $true$ となる．また，`x=` によって，`x+y` のオペランドが変更される節 9 では，$h_n^r(y) = \top$ となり，クエリが $false$ になることが分る． ∎

求めたいクエリの結果は，一般に複数のプログラム点におけるクエリの結果の論理積として得られるので，伝播先の少なくとも 1 つのプログラム点でのクエリが $false$ になるか，あるいは伝播先が開始節であったときは，そこで伝播を終了させ，元のクエリを $false$ に決定することができる．また，伝播先のすべてのクエリが $true$ であることが分った時点で，元のクエリを $true$ と決定できる．

4. ランクの計算

本手法では，クエリを CFG 節 v から v の支配節 d へ直接伝播させることを考える．支配節 d は，開始節から v へのすべての実行パス上に存在しているので，v から伝播してきたクエリは，途中で伝播を止めない限り，必ず d に到達する．すなわち，d から v に到達するすべての実行パス上で，データフロー情報に変化が生じないのであれば（以降，**短絡条件**と呼ぶ），先行節を通って d に伝播されるクエリは v におけるクエリと同じであることが保証できる．

ここで，節 v のランクを $rank(v)$ と表すことにすると，d から v への実行パス上の任意の節 v' の $rank(v')$ が，$rank(d)$ よりも大きく，$rank(v)$ よりも小さい場合，短絡条件は，データフロー情報に変化を生じる節のランク集合を R として，$rank(d) < r < rank(v)$ を満たす r が R に含まれないことを検査するだけでよい．以降，データフロー情報に変化を生じる節のランクを記録した表を**ランク表**（rank table）と呼ぶ．

CFG が非循環グラフ G_{DAG} の場合には，次の単純な計算法によって，ランクを決定することができる．

定義 1（ランク計算法）：

(1) v が開始節ならば，$rank(v) = 0$ である．

(2) そうでなければ，$pred(v)$ の中で最大のランクをもつ節を v_{max} として，$rank(v) = rank(v_{max}) + 1$ である． ∎

一方，CFG に循環が存在する場合，上記のランク計算法では，節のランクを一意に決定することができない．そこで，CFG から上昇辺を取り除いたグラフに，ランク計算法を適用することを考える．しかしながら，この場合にはさらに，次に示す問題が生じて正しいランク付けを行うことができない．

(1) ループ外の節 v_{outer} がループ内の節 v_{inner} を支配する場合，v_{outer} から v_{inner} への実行パス上のすべての節 v' が次の関係を満たすとは限らない．

$$rank(v_{outer}) < rank(v') < rank(v_{inner})$$

例： 図 1 (a) において，節 2 は節 5 を支配しているが，節 2 から節 5 への実行パス上に存在する $v' \in \{6, 7, 8, 9\}$ は，$rank(2) < rank(v') < rank(5)$ を満たさない． ∎

(2) ループの出口節を v_{exit} とし，ループ外にある v_{exit} の後続節を $v_{postExit}$ とすると，$v_{postExit}$ の支配節 v_d から $v_{postExit}$ への実行パス上のすべての節 v' が次の関係を満たすとは限らない．

$$rank(v_d) < rank(v') < rank(v_{postExit})$$

例： 図 1 (a) において，クエリを節 14 から 1 に直節伝播させようとした場合，節 9 でデータフロー情報が変化するので，その伝播は避けなければならない．しかし，上記の計算法によってランクを計算すると，$rank(1) = 0$，$rank(9) = 6$，$rank(14) = 6$ となり，$rank(1) < rank(9) < rank(14)$ の関係が満たされない． ∎

本手法では，ループ外の節からループ内の節へは短絡辺を付加しないことによって，問題 (1) を解決する．短絡辺を付加する方法については，次の章で詳しく述べる．

問題 (2) については，ループ出口の後続節がループ外にあるとき，そのランクを上昇辺の出元節の最大ランクより大きく設定することによって解決する．ループ内で最も長い実行パスは，上昇辺の出元の節 v_{upSrc} を含むので，ループ内で最大のランクをもつ v_{upSrc} が少なくとも 1 つ存在する．すなわち，ループ外に存在するループの出口の後続節 $v_{postExit}$ を，上昇辺の出元節 v_{upSrc} の最大のランクより大きく設定すればよい．

本手法では，各ループについて，すべての上昇辺の出元節とすべての出口のループ外にある後続節の組を考え，それらの間に，CFG 辺が存在しなければ，仮想の辺（以降**仮想辺**と呼ぶ）を生成して，任意の $v_{postExit}$ のランクが，v_{upSrc} のランクより大きくなるように強制する．

例： 図 1 (a) に対して，ランクの順序を強制するために付加する仮想辺を，図 3 (b) の太い矢印で示す．図 3 (b) に対して，CFG 辺と仮想辺を使ってランク付けを行うと，左に示すように，各節の高さに対応したランクを計算することができる． ∎

仮想辺の付加は，ループの入れ子構造を表す**ループ木**（loop tree）[12),14)] を作成するのと同様の方法で行うことができる．本手法では，ループ木の作成アルゴリズムとして Morgan の手法[12)] を用いる．ループ木の作成手順は，次のとおりである．

(1) 解析対象となる CFG G について，そのコピー CFG G' を用意する．以降，ループ木の作成は，G' 上で行う．

(2) CFG 節を深さ優先順序の逆順で訪問し，上昇辺 (v, h) の指し先 h を見つける．

(3) 節 v と h の共通の支配節で，h に最も近い節 z を見つける．以降，z は，節 v と h を含むループ l_z を代表するループ木の節とする．また，z を l_z の**代表節**と呼ぶ．

(4) v から z まで後向きに各節 v_l を訪問し，ループ l_z に含まれる節として，ループ木の辺 (z, v_l) を生成する．そして，v_l を CFG 節から取り除く．その際，v_l から l_z に含まれない節 v' への辺 (v_l, v') を，z からの辺 (z, v') で置き換える．

ステップ (3) で，$z = h$ であれば，ループは可約ループとなり，そうでなければ既約ループとなる．

例： 図 3 (a) に，ループ木が表す入れ子構造を点線で囲んで示す．節 1，2，10 は，各ループの代表節である． ∎

本手法では，ステップ (4) の各節を後向きに訪問する過程で，節 x を訪問したのち，すでに発見済みのループの代表節 y に訪問した場合，CFG G において，y のすべての上昇辺の出元節と x の組のうち CFG 辺が存在しない組に対して仮想辺を生成する．

$$< y, \mathrm{s} > \leftrightarrow \begin{cases} true & \text{if } y = \bot \\ false & \text{otherwise} \end{cases}$$

$$< y, n > \leftrightarrow \bigwedge_{m \in \begin{cases} parent(n) & \text{if } check(rank(parent(n)), rank(n)) = 0 \\ pred(n) & \text{otherwise} \end{cases}} \begin{cases} false & \text{if } h^r_m(y) = \top \\ true & \text{if } h^r_m(y) = \bot \\ < h^r_m(y), m > & \text{otherwise} \end{cases}$$

図 4 疎な要求駆動型データフロー解析のクエリ伝播
Fig. 4 Query propagation for sparse demand-driven dataflow analysis

例： 図 3 (a) において，点線で囲まれた部分が発見済みのループであり，CFG 全体を除くそれぞれの代表節は，節 2 と 10 である．ステップ (4) で，節 14 から先行節を辿ると，付け替えた辺によって，節 2 と 10 に到達する．節 2 と 10 は，代表節なので，上昇辺の出元節 9，13 からそれぞれ節 14 へ仮想辺 (9,14) と (13,14) が必要である．辺 (13,14) は，既に存在しているので，仮想辺 (9,14) を付加する． ∎

5. 短絡辺の付加

本手法では，節 v の短絡節 d を明示すために，CFG に対して短絡辺 (d, v) を付加する．本節では，直接伝播が可能な節としてどのような節を用い，短絡辺を付加するかを述べる．

要求駆動型データフロー解析は，伝播させた先のクエリが，1 つでも $false$ になることが分ると，そこで解析を終了させ，最終的な結果を $false$ とする．また，伝播されたすべてのクエリが $true$ になることが分った場合は，そこで解析を終了させ，最終的な結果を $true$ とする．このような場合には，伝播の範囲が局所化され，網羅型より効率の良い解析が期待できる．

一方，クエリの解がすぐには得られず，伝播される他のクエリも同様な状況にあったり，一部の結果が $true$ であったりした場合は，クエリが CFG の開始節まで伝播されてしまう．要求駆動型データフロー解析の効率向上をはかるためには，開始節までの伝播を効率良く行うことが重要である．そこで，本手法では，ループ外の節からループ内の節へ短絡辺をもたない短絡節のうち，開始節にできるだけ近い節を選んで短絡辺を付加する．具体的には，節 v と v の短絡節 d とからなる短絡辺 (d, v) を，次のような場合分けによって付加する．

(1) 節 v がループに含まれない場合：
開始節 s との間に (s, v) を付加する．

(2) 節 v が可約ループ l に直接含まれる場合：l の代表節 d との間に (d, v) を付加する．

(3) 節 v が既約ループ l によって直接含まれる場合：l の代表節 cd の支配木上の子供節 $\{c \mid cd \text{ は } c \text{ の即支配節}\}$ のうち，v を支配するものを d として，(d, v) を付加する．このとき，$d = v$ ならば，短絡辺は付加しない．

以上の処理は，前節で述べたループ木作成の際に同時に行うことができる．ここで，場合 (2) の代表節 d は，ループヘッダであり，ループ l に含まれる．これに対して，場合 (3) の節 cd は，ループ l の外部に存在する．すなわち，場合 (3) では，ループ l 内の節で，cd に最も近い v の支配節 d を短絡節としている．

d がループ内の節であることは，背理法を用いて次のように示すことができる．もし，場合 (3) の節 d が，ループ l の外に存在する節であったとすると，d はループ内の節 v を支配することから，l のすべての入口も支配する．同時に l の入口節を指す上昇辺の出元も支配するので，d は l の代表節か cd の支配節かのいずれかである．これは，d の即支配節が cd であることに反する．

本手法の短絡辺に関して，ランクに基づいく短絡条件の検査が健全であることは，次のように示すことができる．

定理 1 (ランクに基づく短絡条件) ：データフロー情報に変化を生じる節のランク集合を R_{flow} とすると，ランク集合 $R_{chk} = \{r \mid rank(d) < r < rank(v)\}$ が R_{flow} のランクを含まないなら，短絡辺 (d, v) について短絡条件が成り立つ．

証明: CFG から上昇辺を除いたグラフ G_{DAG} において，定義 1 から，d から v に到達する実行パス上のすべての節のランクは，R_{chk} に含まれる．次に，d から v への実行パス上に含まれる上昇辺 (s, t) は，上昇辺と G_{DAG} の関係から次の 3 とおりが考えられる．

(1) G_{DAG} における d から v への実行パス上に 節

s と 節 t の両方が存在する場合

(2) G_{DAG} における d から v への実行パス上に 節 s だけが存在する場合

(3) G_{DAG} における d から v への実行パス上に 節 t だけが存在する場合

(4) v が t である場合

(1) の場合，仮想辺を用いたランクの計算によって，上昇辺を通るパス上の節は，そのランクが R_{chk} に含まれる．(2) の場合，節 d が節 v の支配節であることから，節 t から節 v への実行パス上に d が存在する．従って，v のクエリは，節 t に伝播される前に，節 d へ伝播されるので，d から v への実行パスにこの上昇辺は含まれない．(3)，(4) の場合，(d,v) は，ループ内部からループ外部への辺に相当し，短絡辺は存在しないので，考慮する必要はない．

以上から，短絡辺 (d,v) について，R_{chk} が R_{flow} のランクを含まなければ，短絡条件が成り立つ． ∎

6. 短絡辺経由のクエリ伝播可能性の検査

要求駆動型データフロー解析を，短絡辺に関する短絡条件の検査と，短絡辺経由のクエリ伝播とによって拡張したデータフロー方程式を図 4 に示す．最終的に，要求されるデータフロー解析に合わせて逆フロー関数を決定し，短絡辺を付加した CFG に適用することによって疎な要求駆動型データフロー解析を実現することができる．

ここで，$parent(v)$ は，v の短絡節を表す関数であり，$check(r,r')$ は，ランク r と r'（$r < r'$）の間に，データフロー情報を変化させる節のランク集合 R の要素が含まれているかどうかを判定する関数である．

$check$ は，ランク表をビットベクタで表現することによって，効率の良い実装が可能である．ランクをビットベクタ B のビット位置に対応させ，R に含まれる各ランクに対応する位置のビットを 1 に，他を 0 に割り当てる．B に対して，ビット位置が r 以下の部分と，r' 以上の部分をマスクしたビットベクタ B' とすると，$check$ は，B と B' のビット積 $B\&B'$ が 0 でないかどうかを検査する関数として実現できる．以降，$check$ は，単に $B\&B'$ を計算する関数とする．

この $check$ を利用すると，$check(r,r')=0$ のとき，r と r' の間には，R の要素が含まれないことが保証できる． 一方，$check(r,r')\neq 0$ は，必ずしも，r と r' の間に R の要素が含まれることを保証するわけではない．この意味で，$check$ は，**保守的**（conservative）な判定関数である．

例： 図 1 (b) に示す CFG において，`x+y` の利用可能性に関するランク表は，$B=[1,1,0,0,1,1,1,0]$ と表現できる．節 8 から節 5 への短絡辺に沿った伝播を考えると，ビットマスクを $M=[0,0,0,1,0,0,0,0]$ として，ビット積 $B\&M$ が 0 になることから，短絡辺に沿った伝播が可能であることが分る． ∎

ランクをビットベクタで管理することによって，データフロー情報を変化させる CFG 節の更新はビットの反転によって実現できるので，インクリメンタルにデータフロー情報を反映した疎な解析が実現できる．

例： 図 1 (b) に示す CFG に対して，節 14 の `x+y` の解析を行う前に，節 11 の `x+y` の解析を行ったとすると，節 10 に同一の式があることから節 11 で `x+y` は利用可能であることが分る．従って，節 11 の `x+y` は共通部分式として除去することができる．このとき，同時に，ビットベクタ B のビット位置 2 のビットを 0 に更新することによって，式が除去されたことをランクに反映させることができる．結果として，節 14 におけるクエリは，節 13 に伝播させたのち，短絡辺に沿って節 10 に伝播できる． ∎

7. 評　　価

本手法の効果を調べるために，共通部分式の除去を例にとって，本手法と従来法とでそれぞれ実現し，両者の解析の実行時間を比較した．実現には，**並列コンパイラ向け共通インフラスト ラクチャ**（a COmpiler INfrastructure project，以降 COINS と呼ぶ）[4] から提供されている C コンパイラを使用した．COINS コンパイラは，すべて Java を用いて実現されており，Java 仮想機械上で動作する．入力されたソースコードは，高水準中間表現から低水準中間表現（以降，LIR と呼ぶ）に変換され，LIR を基にして目的コードが生成される．この実験に用いたハードウェアは Pentium 4 CPU 2.4GHz である．

本評価では，共通部分式の除去に必要な利用可能性の計算を要求駆動型データフロー解析で実現する際に，簡単な拡張を加えている．すなわち，式 $x+y$ に関するクエリが，式のオペランド x を定義しているコピー代入 $x=z$ に伝播されたとすると，そこで解析を終了するのではなく，オペランド x をコピー代入の右辺 z で置き換えることによって，新たに $z+y$ に関するクエリを生成し，それをさらに伝播させるようにしている．要求駆動型データフロー解析におけるこのような拡張は，容易に行えることが知られている[6]．

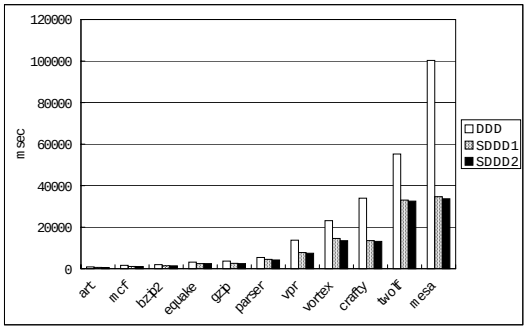


図 5 解析時間の比較
Fig. 5 Analyzing costs (msec.)

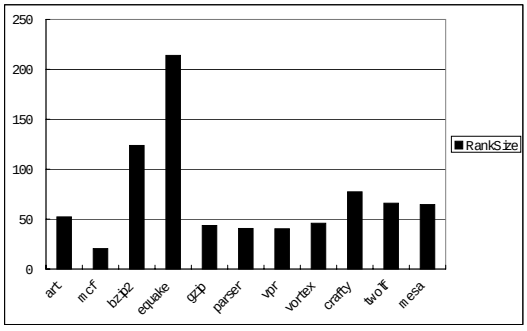


図 6 ランクサイズの比較
Fig. 6 Rank sizes

この実験では，サンプルプログラムに出現するすべての式に対して次の 2 つの手法を適用し，実行時間を比較した．いずれの手法も，LIR 上の最適化パッケージの 1 つとして実現した．

(1) 従来法：**利用可能式**（available expression）を計算する要求駆動型データフロー解析
(2) 本手法：利用可能式を計算する疎な要求駆動型データフロー解析．ランク表は，Java における `long`（64 ビット）の配列によって実現し，各式のパターンごとにこの配列を 1 つ用意する．

評価に使用したプログラムは，SPEC CINT2000 ベンチマークから 8 つのプログラム（mcf，bzip2，gzip，parser，vpr，vortex，crafty，twolf）と SPEC CFP2000 ベンチマークから 3 つのプログラム（art，equake，mesa）である．実行効率の比較には，プログラム中の各式の解析に要した CPU 時間の合計を用いた．

図 5 に，各手法の解析時間（ミリ秒）をグラフにして，プログラムサイズの小さい順に示す．各解析時間の内容は，次のとおりである．

DDD ：従来法のデータフロー解析を行ったときの時間
SDDD1 ：本手法において，支配木の作成，ループ木の作成，ランク付け，ランク表の初期化，およびデータフロー解析に要した時間
SDDD2 ：本手法において，ランク表の初期化とデータフロー解析に要した時間

DDD と SDDD1 を比較すると，サンプルプログラムのサイズが大きくなるほど，本手法の効果が増していることが分る．すべてのプログラムに要した解析時間を平均すると，本手法の従来法に対する割合は，47.8% であり，半分以下のコストで実現できている．本手法で作成した短絡辺と各 CFG 節のランクは，データフロー解析を何回か繰り返して行うときにも使用することができる．そこで，2 回目以降のデータフロー解析の適用を考慮して，支配木の作成，ループ木の作成，ランク付けの時間を除いた結果を SDDD2 として示す．ランク表の初期化は，データフロー解析の種類によって異なるので，SDDD2 に含めてある．SDDD1 と同様，プログラムサイズが大きくなるほど効果が増しており，すべてのプログラムに要した解析時間の平均は従来法の 46.2% となっている．

次に，各サンプルプログラムについてランクの最大値であるランクサイズを求め，それらの平均を計算した結果を，図 6 に示す．ランクサイズとプログラムサイズとの間に相関関係は認められないが，多くのプログラムにおけるランクサイズの平均は 64 を下回っており，ランク表は，64 ビットアーキテクチャのマシンであれば 1 ワードに収まることが期待できる．すべてプログラムについて，ランクサイズの平均を計算した結果は，71.7 であった．

本手法で採用した短絡辺の付加方法は，ループだけに着目した単純なものであり，短絡辺が短絡条件を満たす場合，クエリの伝播が不要な最も多くの節への訪問を回避する方法である．一方，短絡条件が満たされなかった場合，回避するはずであった多くの節に対してクエリを伝播させなければならない．この問題は，関連研究で述べるような他の短絡辺の付加方法を選択することによって，改善できる可能性があるが，本手法が最も多くの節への訪問を回避する方法である点を

考慮すると，改善の方法として許されるのは，本手法の短絡辺を複数の短絡辺に分割することに限られる．この意味で，本手法で採用した方法が，今後の研究における指針を与えるものと考える．

8. 関連研究

本手法が基にしている要求駆動型データフロー解析は，Duesterwald[6] らによって提案された．それ以前にも，Rosen らが冗長な式を除去するために導入した**質問伝播**（question propagation）[15] のようにクエリを伝播する手法は存在したが，定式化したのは，Duesterwald らが最初である．元々，要求駆動型データフロー解析は，手続き間解析を含んでおり，この点で特に解析コストの低減が期待できることが知られている．本稿では，本手法を手続き内だけに適用しているが，手続き間に適用することも容易であり，本手法の疎な解析がさらに効果的に働くと考えられる．

疎な解析には，従来から定義-使用チェインや使用-定義チェインを用いる手法が利用されてきた[1]．これらは，変数の定義と使用を直接関連付けた手法であり，近年では，複数の定義の結合点を明示した静的単一代入（static single assignment，以降 SSA）形式も利用されるようになった．また，データフロー情報を，変数から式に拡張し，同じ式を関連付けた疎な解析によって，部分冗長除去[10),11] を高速化した手法に，Weise らの**値依存グラフ**（value dependence graph）[16] がある．これらの手法は，特定のデータフロー情報に限定している点で制限があった．

SSA 形式を，変数以外の疎なデータフロー解析に一般化したものに，Choi らの**評価グラフ**（evaluation graph）[3] がある．評価グラフは，扱うデータフロー情報が変化するプログラム点を直接関連付けるばかりでなく，グラフの作成時に，可能な伝播を行ってしまい，さらに高速化を実現している．しかしながら，後に異なる解析を行ったり，プログラムの変形を行ったりした場合は，グラフの再計算が必要である．

Johnson らの**プログラム構造木**（program structure tree）に基づく**高速伝播グラフ**（quick propagation graph）[8] は，入口と出口がそれぞれ 1 つしかない（single entry and single exit, SESE）領域の入れ子構造に基づいており，本手法を応用することができるが，高速伝播グラフ自体は，評価グラフ同様，個々の解析に対して用意しなければならないものである．

本手法は，すべての解析に対してグラフ構造が共通であり，異なるのはランク表だけなので，グラフ作成コストが小さいという特徴をもつ．また，解析中にプログラムが変形された場合でも，ランク表の更新を行うだけで，グラフの再計算は必要ない．

9. まとめ

本稿では，要求駆動型データフロー解析を疎な解析によって実現する発見的手法を示した．本手法の特徴は，すべてのデータフロー解析に共通の短絡辺を CFG に付加しておき，解析の際には，短絡辺に沿ったクエリの伝播が可能であるかどうかをランクに基づいて検査しながら，有効な伝播先を動的に選択する点にある．

本手法の効率を評価するために，C コンパイラに本手法を組み入れた処理系を実現し，解析時間を測定して従来法と比較した．その結果，本手法は，プログラムサイズに比例して効率が向上し，平均でも従来法の半分以下の実行時間で解析できることを示した．

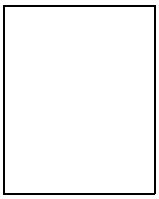
ここで提案した疎な解析を用いることによって，要求駆動型データフロー解析のより広範囲への応用が期待できる．

参考文献

1) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*, Addison Wesley (1986).
2) Appel, A. W.: *Modern Compiler Implementation in ML*, Cambridge University Press (1998).
3) Choi, J. D., Cytron, R. and Ferrante, J.: Automatic Construction of Sparse Data Flow Evaluation Graphs, *Proc. Principles of Programming Languages (POPL'91)*, ACM, pp.55–66 (1991).
4) COINS: *A Compiler Infrastructure Project*, http://www.coins-project.org/.
5) Dhamdhere, D. M., Rosen, B. K. and Zadeck, F. K.: How to Analyze Large Programs Efficiently and Informatively, *Proc. Programming Language Design and Implementation (PLDI'92)*, ACM, pp. 212–223 (1992).
6) Duesterwald, E., Gupta, R. and Soffa, M. L.: A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis, *ACM Trans. Prog. Lang. Syst.*, Vol. 19, No. 6, pp. 992–1030 (1997).
7) Hughes, J. and Launchbury, J.: Reversing Abstract Interpretations, *Proc. 4th European Symposium on Programming (ESOP '92)*, LNCS, Vol. 582, Springer-Verlag, pp. 269–286 (1992).
8) Johnson, R., Pearson, D. and Pingali, K.: The Program Structure Tree, *Proc. Programming Language Design and Implementation (PLDI)*,

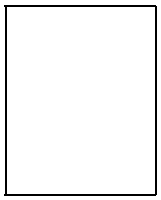
ACM, pp. 171–185 (1994).
9) Khedker, U. P. and Dhamdhere, D. M.: A Generalized Theory of Bit Vector Data Flow Analysis, *ACM Trans. Prog. Lang. Syst.*, Vol. 16, No. 5, pp. 1472–1511 (1994).
10) Knoop, J., Rüthing, O. and Steffen, B.: Lazy Code Motion, *Proc. Programming Language Design and Implementation (PLDI'92)*, ACM, pp. 224–234 (1992).
11) Morel, E. and Renvoise, C.: Global Optimization by Supperssion of Partial Redundancies, *Comm.ACM*, Vol. 22, No. 2, pp. 96–103 (1979).
12) Morgan, R.: *Building an Optimizing Compiler*, Bufferworth-Heinemann (1998).
13) Muchnick, S. S.: *Advanced Compiler Design Implementation*, Morgan Kaufmann (1997).
14) Ramalingam, G.: On Loops, Dominators, and Dominace Frontier, *Proc. Programming Language Design and Implementation (PLDI)*, ACM, pp. 233–241 (2000).
15) Rosen, B. K., Wegman, M. N. and Zadeck, F. K.: Global Value Numbers and Redundant Computations, *Proc. Principles of Programming Languages (POPL'88)*, ACM, pp. 12–27 (1988).
16) Weise, D., Crew, R. F., Ernst, M. and Steensgaard, B.: Value Dependence Graphs: Representation with Taxation, *Proc. Principles of Programming Languages (POPL'94)*, ACM, pp. 297–310 (1994).

(平成 2004 年 12 月 21 日受付)
(平成 ? 年 ? 月 ? 日採録)



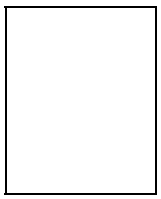
滝本 宗宏（正会員）

1994 年慶応義塾大学大学院理工学研究科計算機科学専攻修士課程修了．1999 年東京理科大学理工学部情報科学科助手．現在，東京理科大学理工学部情報科学科講師．工学博士．プログラミング言語およびその処理系に興味をもつ．ACM，IEEE，日本ソフトウェア科学会各会員．



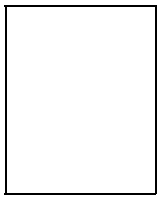
福岡 岳穂（正会員）

1999 年成蹊大学工学部経営工学科卒業．2001 年電気通信大学大学院情報システム学研究科卒．現在株式会社 管理工学研究所研究員．コンパイラ，並列／分散アルゴリズム，計算機アーキテクチャに興味を持つ．日本ソフトウエア科学会会員



佐々 政孝（正会員）

1970 年東京大学理学部物理学科卒業．1974 年同大学院博士課程中退，東京工業大学理学部情報科学科助手．1981 年筑波大学電子・情報工学系．1992 年東京工業大学理学部．現在同大学大学院情報理工学研究科数理・計算科学専攻教授．理学博士．プログラミング言語，コンパイラ，プログラミング環境に興味を持つ．著書「プログラミング言語処理系」（岩波書店）．日本ソフトウェア科学会，ACM，IEEE 各会員．



原田 賢一（正会員）

1966 年慶応義塾大学大学院工学研究科管理工学専攻修士課程終了．1967 年同大学工学部助手．1970〜1989 年同大学情報科学研究所助手，専任講師，助教授，教授．1989 年 4 月より同大学理工学部計測工学科教授．1996 年 4 月からは同学部情報工学科教授，同大学院開放環境科学専攻所属．この間，1973〜1975 年米国メリーランド大学計算機科学センター訪問研究員．工学博士．ソフトウェア工学，プログラミング言語およびその処理系の研究に従事．ACM, IEEE, ソフトウェア科学会員．