情報処理学会論文誌　プログラミング　Vol. 2　No. 5　15–27 (Nov. 2009)

質問伝播に基づく投機的部分冗長除去

滝　本　宗　宏[†1]

コンパイラが行うコード最適化の 1 つである部分冗長除去は，冗長な式を除去するとともに，ループ不変式をループの外に移動する強力な手法である．部分冗長除去に基づいて行うプログラム変形は，いずれの実行経路上にも式を増加させないことを保証しており，その意味で，安全な最適化であるといわれる．一方，ループ内に存在する計算のように，頻繁に実行されることが予想される式は，たとえ実行経路上の式の数を増加させても，ループの外に移動させる投機的な移動を行う方が，プログラムの効率的な実行に貢献する場合がある．本研究では，ループ内の式についてだけ，投機的な移動によって，ループの外に移動させる部分冗長除去法を提案する．従来，部分冗長除去において，一部の実行経路上で冗長な部分冗長な式とループ不変式とを見分けることは困難であった．本手法では，質問伝播という要求駆動型の解析法を用いることによって，任意の制御フローグラフに対して，ループ不変式だけを投機的にループ外に移動させることができる．本手法の効果を示すために，本手法を C コンパイラに実装し，評価を行った．その結果，従来法と比べ，実行効率が 17%以上向上する場合があることを確認した．

Speculative Partial Redundancy Elimination Based on Question Propagation

Munehiro Takimoto[†1]

Partial redundancy elimination (PRE) is the code optimization which not only removes redundant expressions but also moves loop-invariant expressions out of a loop, and therefore it is known as one of the strongest optimizations. Since the program transformation performed by PRE guarantees that it never makes the number of expressions increase on any execution path, it is a safe code optimization. On the other hand, frequently executed expressions such as ones inside a loop may contribute to efficient execution of the program due to moving them out of it even if they make the number of expressions increase on some execution paths. In this paper, we propose a new PRE moving only loop-invariant expressions out of the loop based on speculation. Previously, PRE could not distinguish loop-invariant expressions from the partially redundant expressions which are redundant on some execution paths. Our approach achieves such a speculative loop-invariant code motion on any control flow graphs using demand-driven method which is called question propagation. We implemented our approach on a C compiler, and evaluated it for some benchmarks. As a result, we found that some programs were more than 17% improved in efficiency.

1. はじめに

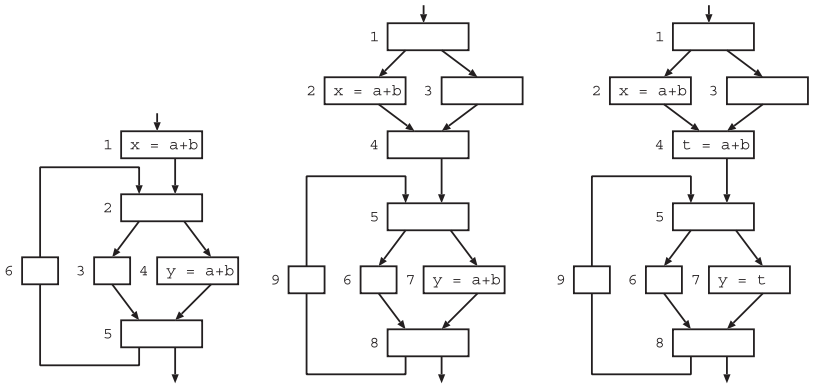
冗長な式を除去する手法（以降，**冗長除去**と呼ぶ）は，コンパイラのコード最適化において強力な手法であり，多くの最適化コンパイラで実現されてきた．特に，**部分冗長除去**（partial redundancy elimination，以降 PRE と呼ぶ）[15] は，**共通部分式**（common subexpressions）として知られる，すべての実行経路で冗長である**全冗長な**（totally redundant）式を除去するばかりでなく，いくつかの実行経路で冗長であるが，すべての実行経路では冗長でない**部分冗長な**（partially redundant）式を除去することができる．また，一部の**ループ不変式**（loop-invariant code）は，部分冗長な式と見なすことができるので，PRE は，ループ不変式をループの外に移動する**ループ不変コード移動**（loop-invariant code motion，以降 LICM と呼ぶ）の効果を含んでいる．

一方，PRE は，任意の実行経路上に新しい計算を導入しないという性質（以降，**下向き安全性**，down-safety と呼ぶ）を持つので，いくつかの実行経路に新しく計算を導入するような**投機的な移動**（speculation）は，難しいことが知られている．たとえば，非常に多くの繰返しが期待できるループに対して，投機的な LICM を適用すると，目的コードの実行効率を向上させる可能性があるが，そのような移動を PRE で扱うことはできない．

例：**図 2** (a) の**制御フローグラフ**（control flow graph，以降 CFG と呼ぶ）における節 9 の式 `a+b` は，ループ不変式なので，ループ外に移動すると，目的コードの実行効率を向上させる可能性があるが，そのような移動は下向き安全でないので，PRE では行われない． ∎

より単純な例として，while ループのように，その本体が 1 度も実行されない可能性がある **0 繰返しループ**（0-trip loop）をあげることができる．0 繰返しループ内の任意の式は，PRE によって，ループ外に移動させることはできない．while ループのような定型なループに対しては，if 文と do-while 型ループとの組合せに変形することによって，ループ外へ

†1 東京理科大学
Tokyo University of Science



(a) PRE の効果 Effects of PRE　(b) LICM の効果 (1) Effects of LICM (1)　(c) LICM の効果 (2) Effects of LICM (2)

図 1　PRE と投機的 LICM の副次的効果
Fig. 1　Second order effects between PRE and speculative LICM.

の移動を実現できるが，そのような変形は，一般には難しい．

これらの投機的移動に関する PRE の問題は，投機的な LICM と組み合わせることによって改善することができる．しかしながら，PRE が任意の CFG に適用できるのに対して，投機的な LICM は，**既約**（irreducibule）な CFG を直接扱うことができないという問題がある．既約な構造は，ユーザが故意に導入することは稀であるが，いくつかの**コード複写**（code replication）をともなう最適化[3),18),19)] によって導入される可能性がある．また，PRE と投機的な LICM には，相互に副次的効果が存在することが知られている．

例：**図 1** (a) の節 4 の式 `a+b` は，ループ不変式であるが，ループの外への移動は投機的に行う必要がある．しかしながら，節 4 の `a+b` は，節 1 の `a+b` に対して冗長なので，PRE によってループ内から除去できる．すなわち，LICM において，本当に必要な投機的移動のコストを正確に見積もるためには，PRE を適用しておくことが有用であることが分かる．次に，図 1 (b) では，節 7 の `a+b` が節 2 の `a+b` に対して部分冗長なので，`a+b` を節 3 へ挿入することによって，節 7 の `a+b` 除去できる．しかしながら，このような PRE は，今まで式が存在しなかった節 1，3，4，5，6，8 を含む実行経路に，`a+b` を導入することになるので，許されない．ここで，前処理として，投機的な LICM を適用すると，図 1 (c) のようになり，節 4 の `a+b` は，節 3 へ `a+b` を挿入する PRE によって，安全に除去できる．∎

例に示したような，PRE と投機的な LICM が互いに生じる副次的効果を反映するためには，PRE と投機的な LICM を複数回適用する必要があり，高いコストが必要になる．

本研究では，任意のループから，ループ不変式をループ外に投機的に移動させることができる PRE を提案する．本手法は，式 e の出現 e_o が冗長であるかどうかを検査するために，「e は冗長か」というクエリを後向きに伝播させる．クエリに対する解が得られると，その解を，クエリの伝播した経路を逆向きに戻していく．プログラム点 v に複数の解が戻ってくる場合，そのすべてが false であれば，そのプログラム点の解を false にする．このとき，v において下向き安全で，解に true と false の両方が含まれるなら，false が戻ってきたプログラム点に e を挿入し，解を true にすることによって，部分冗長性を扱うことができる．

本手法では，クエリが，クエリの生成元の式の出現に再び到達した場合に，その true の解を**自己生成解**として区別する．複数の解が戻ってくる際に，自己生成解と false が含まれているなら，false が戻ってきたプログラム点に，下向き安全性を検査せずに式を挿入する．最終的に，クエリの解が true になり，e_o を除去すると，投機的な LICM が実現できる．

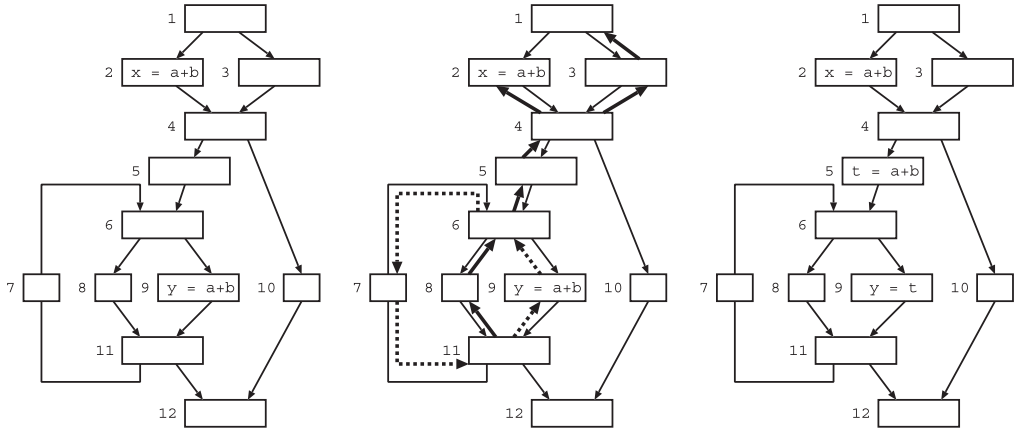
例：図 2 (b) の太矢印は，節 9 の `a+b` に対するクエリの伝播を表現している．太線矢印のうち，点線は，自己生成解が戻る伝播を表し，実線は，それ以外の伝播を表している．節 6 へ伝播したクエリは，さらに，節 5 と節 7 に伝播する．その後，節 5，4 と伝播したクエリは，次の節 2 への伝播で `a+b` の出現に到達するので，true の解が戻る．一方，節 3，1 と伝播したクエリは，開始節 1 で false の解が戻る．ここで，節 4 に true と false の解が戻るので，節 3 の出口に，式の挿入を試みる．しかし，節 3 への挿入は，実行経路 1，3，4，10，12 に `a+b` の計算を付加することになる（下向き安全でない）ので，節 3 から戻る解は false のままにする．結果として，false になった節 4 の解は，さらに節 5，6 と戻る．

節 6 から，節 7，11 と伝播したクエリは，その後，節 8，6 と伝播し，同じクエリの伝播済み節に到達すると，true の解を戻す．一方，節 9 へ伝播したクエリは，クエリの生成元の式に再び到達するので，自己生成解として true を戻す．この後，自己生成解は，節 11，7，6 へと戻る．すなわち，節 6 に false と自己生成解の true が戻るので，false が戻ってきた節 5 の出口に，`a+b` を挿入する．最終的に，節 9 に true の解が戻るので，節 9 の `a+b` を除去することができる．

図 2 (b) の解析結果を基に，一時変数 `t` を用いてプログラムを変形すると，図 2 (c) のプログラムを得る．∎

本手法の特徴をまとめると次のとおりである．

(1)　ループ内からループ外へ投機的な移動を行う．



(a) 元のプログラム Original code　(b) クエリの伝播 Query propagaton　(c) 結果のプログラム Results

図 2　投機的 LICM を行う PRE
Fig. 2　PRE with Speculative LICM.

(2)　ループ内で投機的な挿入を行わない.
(3)　ループ外では，投機的な移動を行わない.
(4)　任意の制御構造を持つプログラムに適用することができる.
(5)　**静的単一代入**（static single assignment，以降 SSA と呼ぶ）形式のプログラムを対象とし，結果として SSA 形式のプログラムを生成する.

以降の構成は，次のとおりである．まず，2 章で，本手法が仮定する，プログラムの構造と表現について述べ，3 章で，PRE について概説する．次に，4 章で，質問伝播を概説し，質問伝播に基づく共通部分式除去法を示す．5 章では，4 章で述べた共通部分式除去を拡張することによって，本手法の振舞いと詳細なアルゴリズムを与える．6 章で，本手法の効果を，実験結果によって示す．7 章で，関連研究を述べ，最後に，まとめを述べる．

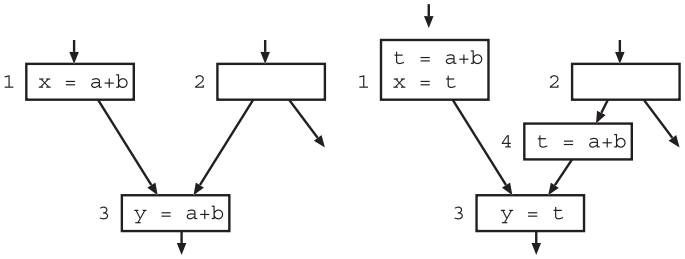
2. 入力プログラム

本手法では，原始プログラムで定義された各関数について，CFG がすでに作成されているものとする．CFG は，途中に分岐を持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 $E \subset N \times N$，そして，特別な節である開始節 $\mathbf{s}$ と終了節 $\mathbf{e}$ とからなる 4 つ組 $(N, E, \mathbf{s}, \mathbf{e})$ である．任意の CFG 節は，$\mathbf{s}$ から $\mathbf{e}$ までの実行経路上に存在するものと仮定する．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，CFG の開始節からある節 w へ至るすべての実行経路が節 v を含むとき，v は w を**支配する**（dominate）といい，$v \geq_{dom} w$ で表す[1),2),17)]．

CFG の各文は，SSA 形式に変換されていることを前提とする．SSA 形式では，すべての変数は唯一の定義を持つ．複数の定義が到達する場合は，それらを，ϕ 関数という仮想関数の引数にし，その返戻値によって新しい変数を定義する．ϕ 関数は，到達する定義点が初めて支配できなくなる点（以降，**支配辺境**，dominance frontier と呼ぶ）に置かれる．さらに，すでに求まっている支配辺境のさらなる支配辺境を付加して得られるプログラム点の集合を考慮する必要がある．この集合は，**反復支配辺境**（iterated dominance frontiers）と呼ばれる．ϕ 関数は，反復支配辺境に属するすべてのプログラム点に配置される必要がある．

任意の原始プログラムに対する CFG から SSA 形式への変換については，効率的なアルゴリズム[2),7),22)] が知られている．

本手法は，他のコード移動に基づく手法と同様に，**図 3** (a) のような，2 つ以上の後続節



(a) クリティカル辺
Critical edges

(b) クリティカル辺の除去
Removing critical edges

図 3　クリティカル辺の扱い
Fig. 3　Critical edges and their elimination.

を持つ節 2 から 2 つ以上の先行節を持つ節 3 への辺（以降，**クリティカル辺**，critical edge と呼ぶ）[11),12)] があると，効果が制限される可能性がある．このようなクリティカル辺は，図 3 (b) のように，節を挿入することによって取り除かれているものとする．

3. PRE の基本

プログラム点 r に式 e が存在し，r からプログラム点 q に到達する実行経路 $\mathcal{P}$ 上で，e のオペランドが変更されないとき，e は，$\mathcal{P}$ 上で，**利用可能**（available）であるという．q に到達する実行経路のうち少なくとも 1 つの経路上で，e が利用可能であるなら，e は，q において**部分利用可能**（partially available）であるという．e が q で部分利用可能である場合，特に，q に到達するすべての実行経路上で，e が利用可能であるなら，e は q で利用可能であるという．式 e が，プログラム点 q に存在し，q の直前で利用可能である場合，e は全冗長であるといい，利用可能な式の値を保持する変数で置き換えることによって除去することができる．一方，式 e が，プログラム点 q に存在し，q の直前で部分利用可能である場合，e は部分冗長であるといい，直接除去することはできない．

また，プログラム点 r に式 e が存在し，プログラム点 q から r へ到達する実行経路 $\mathcal{P}$ 上で，e のオペランドが変更されないとき，e は，実行経路 $\mathcal{P}$ 上のプログラム点 q で，**予期可能**（anticipated）であるという．q から始まる少なくとも 1 つの実行経路上で，e が予期可能である場合，e は，q において，**部分予期可能**であるという．e が q で部分予期可能である場合，特に，q から始まるすべての実行経路上で，e が予期可能であるとき，e は q で予期可能であるという．式 e が，プログラム点 q で予期可能である場合，q は，e に関して下向き安全であるという．

PRE は，式を挿入することによって，部分冗長な式を除去する．このとき，利用可能でなく，下向き安全なプログラム点に挿入することによって，いずれの実行経路上の式の実行も増すことなく，部分冗長な式を全冗長にし，除去できるようにする．

PRE で，実際に式を除去するには，利用可能な式の値を一時変数に格納し，その一時変数によって，冗長な式を置き換える必要がある．式を除去する際に導入される一時変数は，PRE の効果を低減させないように，レジスタに割り付けることが重要である．そこで，PRE は，**レジスタ圧力**（register pressure）を抑えるために，冗長除去の効果が得られない不要なコード移動を行わない．

4. 質問伝播に基づく共通部分式除去

この章では，まず，質問伝播を定義し，次に，以降の理解を容易にするために，**質問伝播に基づく共通部分式除去**（common subexpression elimination based on question propagation，以降，CSEQP と呼ぶ）を示す．本手法のアルゴリズムは，CSEQP から容易に拡張することができる．

4.1 質 問 伝 播

質問伝播は，Rosen らによって提案された冗長性を検査する手法[20)] であるが，本研究では，SSA 形式上で利用可能性を検査するために質問伝播を用いる．

CFG 節 v_o に含まれる式 e の利用可能性は，「e が利用可能か」というクエリ $av(e)$ を，v_o の先行節から始めて後向きに伝播させることによって検査することができる．この際，クエリは，次の規則に従って伝播させる．ここで，各規則は，上から優先して適用するものとし，v 中の冗長性は取り除かれているものとする．

(1)　CFG 節 v にクエリが伝播されたとき，v が開始節 **s** なら，v におけるクエリの解は $false$ になる．
(2)　CFG 節 v にクエリが伝播されたとき，v に，$av(e)$ と同じクエリが伝播済みなら，v におけるクエリの解は $true$ になる．
(3)　CFG 節 v にクエリが伝播されたとき，v に，$av(e)$ と異なるクエリ $av(e')$ が伝播済みなら，v におけるクエリの解は $false$ になる．
(4)　CFG 節 v にクエリが伝播されたとき，v に式 e が含まれているなら，v におけるクエリの解は $true$ になる．
(5)　CFG 節 v にクエリが伝播されたとき，v に，式 e のオペランドを変更する ϕ 関数以

外の計算が含まれるなら，v におけるクエリの解は $false$ になる．

(6)　規則 (1)–(5) 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，CFG 節 v に，オペランド x を持つ式 $e(x)$ のクエリ $av(e(x))$ が伝播されたとすると，v に，x を定義する ϕ 関数 $x = \phi(a_0, a_1, \ldots, a_i)$ が含まれるなら，クエリを各引数に基づいて $av(e(a_0))$，$av(e(a_1)), \ldots, av(e(a_i))$ に変更し，その引数に対応する先行節にそれぞれ伝播させる．

質問伝播では，規則 (1)–(5) のいずれかを満たす場合に，各節において局所的に解を得ることができる．もし，局所的に解が得られない場合は，(6) によって，クエリを伝播し，先行節に対して，一連の規則を適用する．ここで，(2) において，伝播先に同じクエリが伝播済みなら，解を $true$ にする点に注意が必要である．この規則によって，網羅型データフロー解析同様，データフロー方程式の最大解を求めることができる．最終的に，クエリの解が $false$ にならずに，すべての伝播が終了すると，v_o におけるクエリの解は $true$ になり，v_o の式 e が冗長であることが分かる．

上記の質問伝播は，伝播の方向を前向きに変更し，依存関係による計算順序に基づいて規則 (4) と (5) の順序を入れ換え，(6) を次のように変更することによって，予期可能性の検査として用いることができる．以降，「e が予期可能か」というクエリは，$ant(e)$ で表す．

(6)　CFG 節 v から v の後続節 v_s に，オペランド x を持つ式 $e(x)$ のクエリ $ant(e(x))$ が伝播されるとき，v に対応する引数として x を持つ ϕ 関数 $y = \phi(\ldots, x, \ldots)$ が v_s に含まれるなら，$ant(e(y))$ に変更して伝播させる．

4.2　共通部分式除去の冗長性検査

質問伝播の応用として，共通部分式除去法の CSEQP を示す．CSEQP は，次の 2 段階で実現する．

(1)　冗長性検査

(2)　プログラム変形

冗長検査では，質問伝播の実現に加えて，クエリの解および，利用可能式の値を保持する一時変数の定義点を戻す操作が重要である．得られた定義点は，後のプログラム変形において，一時変数を SSA 形式で表現するのに役立つ．アルゴリズム 1 に，冗長検査のアルゴリズムを示す．

節 v の式 e について冗長性を調べたい場合，$propagate(e, v)$ の呼び出しを行う．$local$ と $propagate$ は，それぞれ節 v のクエリ $av(e)$ に対する解を局所的に求める関数と，クエリを伝播させて解を求める関数を示している．クエリは，これらの関数の相互再帰呼び出しによって伝播され，その解は，利用可能式の節と組みにして，返戻値として戻す．利用可能式の節は，プログラム変形で一時変数を導入する際に，その定義点となる節を示す．

$local$ の 16，18，21，24，26 の各行は，それぞれ，質問伝播の規則 (1)–(5) に対応している．ここで，$query[v]$ は，v に到着済みのクエリの式を記録する配列であり，$isComp(e, v)$ は，節 v に式 e の出現がある場合に true を返す関数である．規則 (2) に対応する 18–20 行目では，解がすでに求まっている場合，19 行目でその解を戻すようにしている．解が求まっていない場合，20 行目で，現在訪問中の節を利用可能式の仮の節として戻す．この仮の節は，プログラム変形において，本当の利用可能式の節へリンクを提供する役割をする．

$propagate$ は，各先行節 p に，e_p のクエリを伝播させる．e_p は，節 v に ϕ 関数による e のオペランドの定義が存在しなければ，e 自身であり，さもなければ，規則 (6) の変形を適用したものになる．変形は，4 行目の関数 $transPhi$ によって行う．次に，$local$ を呼び出し，局所的な解 $(isAvail_p, v_p)$ を得る（5 行目）．最終的に，8 行目で，すべての先行節の解 $isAvail_p$ が true だった場合，v の解も true になる．このとき，初期値を v とする v' を用いて返戻値 $(true, v')$ を返す．さもなければ，14 行目で $(false, \bot)$ を返す．利用可能式の節は，7 行目で $link[v]$ にリストとして保持し，後のプログラム変形で，v に ϕ 関数を挿入するために利用する．

ここで，利用可能式の節の中に，v を支配する節 v_p があり，e_p が e と等しいならば，v' として v_p を採用してよい（9 行目）．この支配関係の検査は，不要な ϕ 関数の挿入を避ける効果がある．

アルゴリズム 1（冗長性検査）
input：SSA 形式の CFG，支配関係情報
output：配列 $query$，$link$

```
1: Function propagate(e, v)
2:   let linkedCand := ∅ and v' := v
3:   foreach p ∈ pred(v)
4:     e_p := transPhi(e, v, p)
5:     let (isAvail_p, v_p) := local(e_p, p)
6:     if (isAvail_p)
7:       add v_p to link[v]
8:   if (∏_{p∈pred(v)} isAvail_p)
9:     if (∃v_p ∈ link[v].v_p ≥_dom v ∧ e_p = e)
10:      link[v] := {v_p}
```

```
11:       v' := v_p
12:     return (true, v')
13: else
14:     return (false, ⊥)
15: Function local(e, v)
16: if (v = s) return (false, ⊥)
17: if (query[v] ≠ ⊥)
18:     if (query[v] = e)
19:         if (answer[v] ≠ ⊥) return answer[v]
20:         else return (true, v)
21:     else return (false, ⊥)
22: query[v] := e
23: let rlt := ⊥
24: if (isComp(e, v))
25:     rlt := (true, v)
26: else if (isMod(e, v))
27:     rlt := (false, ⊥)
28: else
29:     rlt := propagate(e, v)
30: answer[v] := rlt
31: return rlt
```

4.3　プログラム変形

冗長性検査で，$propagate(e, v)$ の結果が，$(true, v')$ であった場合，v' の情報を用いて，プログラムを変形する．

アルゴリズム 2 に，必要な ϕ 関数を挿入してプログラムを変形する関数 $insert$ を示す．

アルゴリズム 2（プログラム変形）
input：SSA 形式の CFG，配列 $query$，$link$
output：SSA 形式の CFG

```
1: Function insert(v)
2:   if (var[v] ≠ ⊥)
3:     return var[v]
4:   else if (isComp(query[v], v)
5:     var[v] := lhs(query[v], v)
6:   else if (|link[v]| = 1)
7:     var[v] := insert(link[v])
8:   else
9:     var[v] := createNewVar()
10:    let args := ∅
11:    foreach v'_i ∈ link[v]
12:      let var_i := insert(v'_i)
13:      add var_i to args
14:    add [var[v] "=" "φ" "(" args ")" ]
         to the entry of v
15: return var[v]
```

v に対する $insert$ の呼び出しは，$link[v]$ 中の節に対して，さらに再帰呼び出しを行って，利用可能節まで到達する．利用可能式に到達すると，4 行目で，その式を含む文の左辺 $lhs(query[v], v)$ を取り出し，$var[v]$ に記録したのち，15 行目で返戻値として返す．$var[v]$ は，2 行目に示すように，訪問済みの節に到達した際の返戻値になる．8 行目で，$link[v]$ の要素が複数であった場合，各要素に $insert$ を適用して（12 行目）得られた変数を引数とした ϕ 関数を v の入口 $in[v]$ に挿入する（14 行目）．代入先は，新しい一時変数を $createNewVar$ 関数で生成し，割り当てる（9 行目）．この ϕ 関数の代入先も，$var[v]$ に記録し，15 行目で返戻値として返す．ここで，$link[v]$ の要素が 1 つの場合は，6，7 行目に示すように，ϕ 関数を生成しない．

5. 質問伝播に基づく投機的部分冗長除去

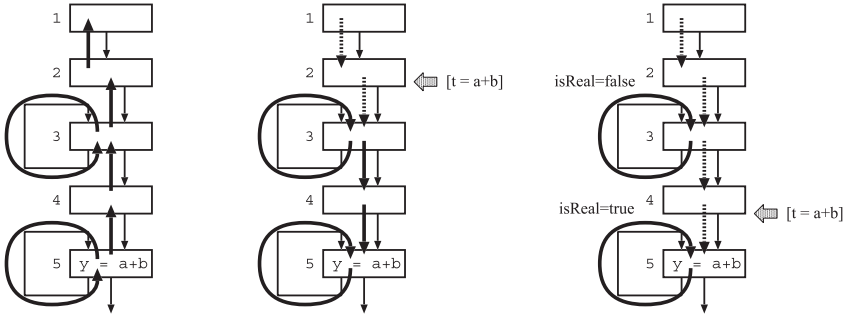
この章では，前節で定義した CSEQP を拡張して，**質問伝播に基づく投機的 PRE**（speculative PRE based on question propagation，以降 SPREQP と呼ぶ）を定義する．まず，CSEQP の冗長性検査を，PRE 用に拡張し，さらに SPREQP に拡張する．次に，SPREQP の冗長検査の結果を基にしたプログラム変形を示す．最終的に，本手法の適用方法と，期待される効果について述べる．

5.1　PRE への拡張

CSEQP を PRE に拡張するためには，次の点を実現する必要がある．

(1)　式の挿入：複数の解が戻る場合，その解に true と false の両方が含まれるなら，false が戻ってきた先行節の出口に，クエリの式を挿入する．
(2)　不要な移動の回避：クエリが生成元の式の出現に到達したという事実を，解と一緒に戻す．

拡張 (1) で式を挿入しようとする場合，下向き安全性を確認する必要がある．下向き安全性は，前章で述べたように，逆向きにクエリ $ant(e)$ を伝播させる質問伝播 $antqp$ を用いて



(a) クエリの伝播
Query propagations
(b) 利用可能情報だけの挿入
Insertions based on only available information
(c) *isReal* 情報による挿入
Insertions based on *isReal*

図 4　不要な移動の回避
Fig. 4　Suppressing needless code motions.

検査する．本手法では，各節にクエリを伝播させるたびに，$antqp$ の結果を $isDownSafe$ に設定し，後の利用可能条件の計算に用いる．

拡張 (2) で扱う不要な移動は，利用可能な式を含まないループで生じる．

例：**図 4** (a) は，節 5 の `a+b` に対するクエリの伝播を示し，図 4 (b) は，true を実線矢印，false を点線矢印として，クエリの解の戻りを示している．図 4 (b) に示すように，クエリの解として，利用可能性の情報だけを用いると，式は，節 2 の出口に挿入される．しかしながら，節 3 からなるループの上への挿入は，冗長な式を除去することなく，このループを通過して移動させるだけである．このような不要な移動は，利用可能式が定義する一時変数の生存期間を長くし，レジスタに割り付けられる可能性を低減させる．この例の場合，式は，節 4 の出口に挿入されるべきである．∎

不要な移動の問題は，利用可能な式が存在することを保証することによって，解決できる．そこで，クエリが式の出現に到達した事実を示すために，関数 $propagate$ および $local$ の返戻値に，「式の出現に到達したどうか」を示す要素 $isReal$ を付加する．

例：図 4 (c) は，クエリの解を $isReal$ の情報で区別して示している．クエリの解が true になるとき，少なくとも先行節の 1 つは，$isReal = true$ でなければならないという条件を付加すると，挿入点は，節 4 の出口になる．∎

以上の点から CSEQP を拡張するために，まず，$propagate$ と $local$ の返戻値 $(isAvail, v)$ を，$(isAvail, isReal, v)$ のように，3 つ組みにする．$isReal$ は，関数 $local$ で $isComp(e, v) = true$ のとき true になる．さらに，先行節 p から戻ってくる結果を $(isAvail_p, isReal_p, v_p)$ として，関数 $propagate$ の節 v における利用可能性の条件（8 行目）を次のように変更する．

$$\prod_{p \in pred(v)} isAvail_p \vee isDownSafe \wedge \sum_{p \in pred(v)} isReal_p \qquad (1)$$

この条件が true のとき，$propagate$ の返戻値は，$(true, \sum_{p \in pred(v)} isReal_p, v')$ となり，そうでなければ，$(false, false, \bot)$ である．

5.2　投機的移動への拡張

前節で PRE へ拡張したものを，さらに投機的移動について拡張する．投機的移動の拡張点は，次のとおりである．

拡張：先行節から複数の解が戻ってくる際に，自己生成解の true と，$isAvail$ の false が含まれているなら，下向き安全性を検査せずに，false の先行節に式を挿入する．

下向き安全性を無視すれば，移動は投機的になる[14]．さらに，自己生成解が戻るという条件は，「ループ内に投機的な挿入を行わない」ことを保証する．この自己生成解の性質を**ループ安全性**と呼ぶ．ループ安全性は，次の補助定理を用いて証明できる．

補助定理 1（ループ非変更性）　節 v の式 e に対するクエリが，自己生成解を得るとき，v を囲むループ L は，v に到達する任意の経路上に，e のオペランドの変更を含まない．

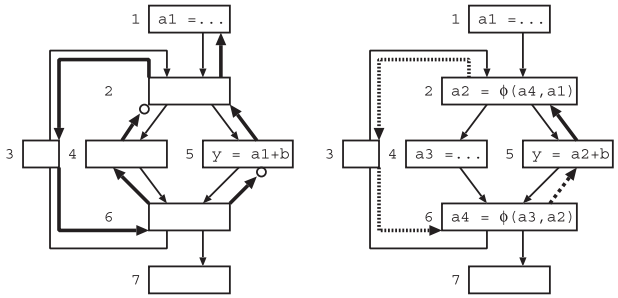
証明：自己生成解が得られたので，クエリの式 e を含む v から，CFG の閉路を通って，v に戻る経路のうち，e のオペランドを変更しない経路 P が，少なくとも 1 つ存在する．このとき，v に到達する L 内の任意の経路上に，e のオペランドへの変更が存在しないことを背理法で示す．

v に到達する経路 P' が，e のオペランドを変更する L 内の節 n を含むとする．ここで，n が，v を支配するなら，P は n を含むことになるので，仮定に反する．一方，n が，v を支配しないなら，P は，n の反復支配辺境を含む．すなわち，P は，e のオペランドを定義する ϕ 関数を含むことになるので，仮定に反する．∎

定理 1（ループ安全性）　節 v の式 e に対するクエリが，自己生成解を得るとき，v を囲むループ L に，e が投機的に挿入されることはない．

証明：自己生成解が得られているので，補助定理 1 から，v に到達する任意の経路は，L 内に e のオペランドへの変更を含まない．したがって，L 中のクエリは，必ず，e の出現か，伝播済みの節に到達し，$false$ の局所解を生ずることがないので，式の挿入を生じない．∎

例：**図 5** は，自己生成解が得られる場合と，そうでない場合を示している．(a) は，式 e の



(a) オペランドの変更なし
No modification

(b) オペランドの変更あり
Some modifications

図 5　ループのクエリ伝播
Fig. 5　Query propagation within a loop.

クエリの伝播について，ループ内に e のオペランドの変更が存在しない場合を示しており，節 5，2，3，6，5 の経路 P によって，自己生成解が得られる．節 5 で生じたクエリは，伝播の後，節 2 と 5 の出口（○で示す）ですべて true になる．

一方 (b) は，クエリの式を変更する節 4 が，節 5，2，3，6，5 の経路 P 上にないにもかかわらず，P 上の節 2，6 に配置される ϕ 関数によって，自己生成解は得られない．∎

「自己生成解が得られたかどうか」を示すために，述語 $isSelf$ を用いると，PRE のために拡張した利用可能性の条件は，さらに次のように拡張する必要がある．

$$\prod_{p\in pred(v)} isAvail_p \vee (isDownSafe \vee \sum_{p\in pred(v)} isSelf_p) \wedge \sum_{p\in pred(v)} isReal_p \qquad (2)$$

最終的に，アルゴリズム 1 を投機的 PRE 用に拡張して得られるアルゴリズムを，アルゴリズム 3 に示す．アルゴリズム 1 に対する拡張部分は，次のとおりである．

(1)　*propagate* と *local* の返戻値に自己生成解 $isSelf$ を加えて，$(isAvail, isReal, isSelf, v)$ の 4 つ組みにする．

(2)　安全性検査：3 行目の $antqp(e, v)$ の呼び出しで行う．結果は，$isDownSafe$ に記録し，後に，利用可能性の条件で使用する．

(3)　式の挿入：10，11 行目で挿入候補を $insertCand[p]$ に記録し，利用可能式の出現場所として，p を $link[v]$ に加える．

(4)　式の出現に到達したかどうかの判定：12 行目で計算し，結果を $isReal$ に記録する．

(5)　自己生成解かどうかの判定：13 行目で計算し，結果を $isSelf$ に記録する．

(6)　利用可能性の計算：14，15 行目で，条件式 (2) を計算する．

(7)　自己生成解の局所的判定：30 行目で，関数 $isComp$ を拡張し，式の出現が存在するかどうかだけでなく，その出現が，クエリの生成元と一致しているかどうかを返すようにする．

アルゴリズム 3（投機的部分冗長性検査）
input：SSA 形式の CFG，支配関係情報
output：配列 $query$，$link$，$answer$，$insertCand$

```
1: Function propagate(e, v)
2:   let linkedCand := ∅ and v' := v
3:     and isDownSafe := antqp(e, v)
4:   foreach p ∈ pred(v)
5:     e_p := transPhi(e, v, p)
6:     let (isAvail_p, isReal_p, isSelf_p, v_p) := local(e_p, p)
7:     if (isAvail_p)
8:       add v_p to link[v]
9:     else
10:      insertCand[p] := e_p
11:      add p to link[v]
12: let isReal := ∑_{p∈pred(v)} isReal_p
13:   and isSelf := ∑_{p∈pred(v)} isSelf_p
14: if (∏_{p∈pred(v)} isAvail_p
15:       ∨isReal ∧ (isDownSafe ∨ isSelf)
16:   if (∃v_p ∈ link[v].v_p ≥_dom v ∧ e_p = e)
17:     link[v] := {v_p}
18:     v' := v_p
19:   return (true, isReal, isSelf, v')
20: else
21:   return (false, false, false, ⊥)
22: Function local(e, v)
23: if (v = s) return (false, false, false, ⊥)
24: if (query[v] ≠ ⊥)
25:   if (query[v] = e)
26:     if (answer[v] ≠ ⊥) return answer[v]
27:     else return (true, false, false, v)
28:   else return (false, false, false⊥)
29: query[v] := e
30: let rlt := ⊥ and (isAvail, isSelf) := isComp(e, v)
```

```
31: if (isAvail)
32:     rlt := (true, true, isSelf, v)
33: else if (isMod(e, v))
34:     rlt := (false, false, false, ⊥)
35: else
36:     rlt := propagate(e, v)
37: answer[v] := rlt
38: return rlt
```

5.3　プログラム変形への拡張

SPREQP のプログラム変形は，アルゴリズム 2 に PRE を実現するための拡張を加えたものになる．その拡張点は，次の 2 点である．

(1)　節の出口への式の挿入

(2)　クエリの解に対する一貫性の検査

SPREQP のプログラム変形を，アルゴリズム 4 に示す．

アルゴリズム 4（プログラム変形）

input：SSA 形式の CFG，配列 *query*，*link*，*answer*，*insertCand*
output：SSA 形式の CFG

```
1: Function insert(v)
2:   if (var[v] ≠ ⊥)
3:     return var[v]
4:   else if (isComp(query[v], v))
5:     var[v] := lhs(query[v], v)
6:   else if (insertCand[v] ≠ ⊥)
7:     var[v] := createNewVar()
8:     add [var[v] " = " insertCand[v]] to the exit of v
9:   else if (#1(answer[v]) = false)
10:    return ⊥
11: else if (|link[v]| = 1)
12:    var[v] := insert(link[v])
13: else
14:    var[v] := createNewVar()
15:    let args := ∅
16:    foreach v'_i ∈ link[v]
17:      let var_i := insert(v'_i)
18:      if (var_i = ⊥) return ⊥
19:      add var_i to args
```

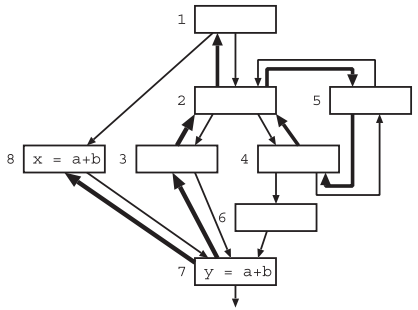


図 6　一貫性のない解
Fig. 6　Inconsistent answers.

```
20:    add [var[v] " = " "φ" "(" args ")"]
         to the entry of v
21: return var[v]
```

式の挿入は，6 行目にあるように，$insertCand[v]$ に挿入候補の式が記録されている場合に行う．その際，8 行目で，新しい一時変数を $var[v]$ に生成し，$var[v]$ `"="` $insertCand[v]$ という文を節 v の出口に挿入する．

PRE で気をつけなければならないのは，9 行目で判定する解の一貫性である．共通部分式では，クエリが 1 つでも false になると，プログラム変形は行わないが，PRE では，クエリが false になっても，式を挿入して，true になる場合がある．すなわち，クエリがどのプログラム点で false になったかが重要になる．

例：図 6 は，節 7 から，節 8 と節 6 に伝播したクエリの解が戻ってきた状況を示している．解の $isAvail$ と $isReal$ に着目して，$(isAvail, isReal)$ で表現すると，節 8 から $(true, true)$，節 3 から $(false, false)$ の解がそれぞれ得られる．ここで，節 2 から，節 5，4，2 と伝播したクエリの解は，$(true, false)$ である．この後，節 7 から節 6，4 へと伝播したクエリは，節 4 で，解 $(true, false)$ を得るので，節 6 からも $(true, false)$ を得る．最終的に，冗長性検査の結果は，節 3 へ式 `a+b` を挿入することになる．しかし，この結果は，節 6 から 7 へ到達する経路上に利用可能式が存在しないことになり正しくない．これは，節 2 の最終的な解 $(false, false)$ が，節 4 から 2 に伝播したクエリの解 $(true, false)$ と異なり，一貫性が損なわれたからである．∎

一貫性が損なわれたかどうかは，冗長性検査の段階で決定した挿入点以外で，クエリがfalseになるかどうかを判定すればよい．アルゴリズム 4 では，$\#1(answer[v]) = false$ を判定し，判定が $true$ であれば，$\perp$ を返す．ここで，$\#1$ は，組みの最初の要素を取り出す関数である．プログラム変形の返戻値が $\perp$ であった場合，行った ϕ 関数や式の挿入をキャンセルしなければならない．

一貫性判定によって，式 e の出現が除去される場合，この出現に到達するすべての実行経路上に利用可能式が存在することが保証される．

5.4　プログラム全体への適用

SPREQP は，個々の式の出現に対して適用するコード最適化なので，プログラム中のすべての式の出現に対して，適切な順序で適用することによって，網羅的な冗長除去を実現することができる．

冗長除去では，ある式が，冗長な式として除去されると，その式に依存している後続の式の冗長性が，新たに明らかになる**副次的効果**（second order effects）が生じる場合がある．PRE において，この副次的効果を反映させるためには，PRE を再度適用する必要がある．一般に，このような副次的効果をすべて反映させるためには，PRE の繰返し適用が必要であることが知られている．

各式の出現ごとに冗長性の除去を考える場合，開始節に近い式の出現から，順に適用していくのが効果的である．たとえば，CFG 節を**ポストオーダの逆順**（reverse postorder）で訪問し，基本ブロック内を入口から出口に向かって到達した式に対して，順に SPREQP とコピー伝播を適用すると，多くの副次的効果を反映できる可能性がある．

6. 評　　価

本手法の効果を調べるために，本手法と従来法とをそれぞれ実現し，両者の解析時間と，目的コード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）[6] 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．入力された原始プログラムは，高水準中間表現（以降，HIR と呼ぶ）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．また，実験には，ハードウェアとして Intel Pentium 4，2.4 GHz CPU の Linux マシンを使用した．

まず，本手法を含めて次の 2 つの手法を，サンプルプログラムに適用し，解析時間と目的コードの実行時間を比較した．評価に使用したプログラムは，SPEC ベンチマークのうち，CFP2000 の 4 つのプログラム（mesa，art，equake，ammp）と，CINT2000 の 7 つのプログラム（gzip，vpr，mcf，crafty，parser，gap，bzip2）である．

PRE*2：コピー伝播，PRE，コピー伝播，PRE，コピー伝播，定数伝播および畳込み，無用コード除去の順に最適化を行う方法．ここで用いた PRE は，データフロー解析でコストが高い初期化を効率化した手法である[16]．

SPREQP：コピー伝播，本手法とコピー伝播のすべての式への適用，定数伝播および畳込み，無用コード除去の順に最適化を行う方法．SPREQP とコピー伝播の適用は，CFG 節をポストオーダの逆順で訪問し，基本ブロック内を入口から出口に向かって到達した式に対して順に行った．

SPREQP は，PRE の副次的効果を反映しやすい特徴を持つので，比較対象として，PRE の 2 回適用 PRE*2 を用いた．両者とも，実践的な評価が行えるように，多くの最適化コンパイラで用いられるコピー伝播，定数伝播，定数畳込み，無用コード除去と組み合わせた．

図 7 に，PRE*2 に対する本手法の解析時間の割合を示す．本手法の解析コストは，parser 以外のプログラムで，PRE*2 を下回った．最大で 69%，平均で 33.2% 解析効率が良いことが分かる．ここで，解の一貫性が損なわれたプログラムは存在しなかった．

次に，目的コードの実行時間の比較を**図 8** に示す．本手法は，mesa，ammp，gzip，vpr，mcf，crafty，parser，gap の 8 プログラムにおいて，PRE*2 を上回る実行効率を得た．特に，

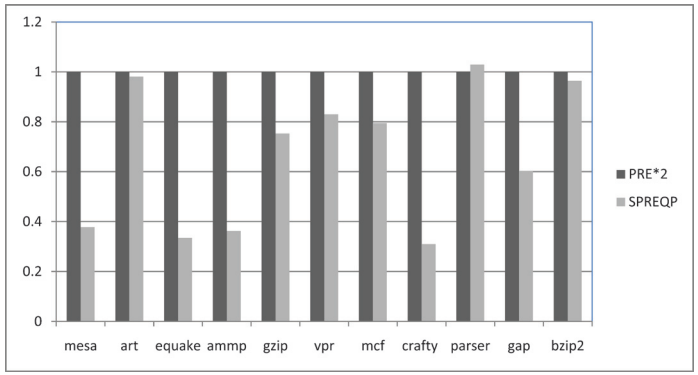


図 7　解析時間（ミリ秒）の比較
Fig. 7　Execution time (msecond).

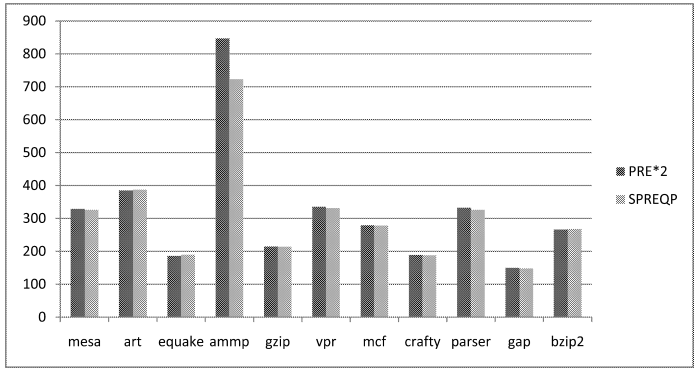


図 8　実行時間（秒）の比較
Fig. 8　Execution time (second).

ammp では，17.2%の向上が得られた．一方，実行効率の低減は，最大で，2.1%であった．

本手法の効果は，投機的な移動によって生じた実行コストの増加を，ループのコスト低減が上回ることによって得られる．すなわち，ループ本体の繰返し回数が多いほど，目的コードの実行効率が向上する．評価結果から分かるように，必ず実行効率の向上が期待できるわけではないので，本手法を実践的に用いる場合，ループの繰返し回数が十分多いことを確認する必要がある．

7. 関 連 研 究

ループ不変コード移動については，不変コードをループの直前の基本ブロックに移すためのアルゴリズムが Aho らによって示されている[1]．このアルゴリズムは SSA 形式を使用していないので，1 つの使用に対して，複数の定義が到達する計算式を移動することができない．

PRE は，コード移動によって効果がある場合に有効な手法であり，ループ不変コード移動も含まれる．Morel ら[15] によって初めて示され，後に多くの改良がなされた．その中には，双方向のデータフロー解析を単方向の組合せにすることによって，データフロー方程式の双方向依存をなくし，ビットベクトルを用いたデータフロー解析の計算量を単方向と同等にする手法[9] や，同様の方法で，不要なコード移動を行わないことを保証する手法[11),12] などがある．また，PRE が式を対象にしているのに対して，代入文自体を移動する手法も提案されている[8),13]．これらが除去の対象としているのは，字句形式が一致する計算だけである．PRE とコピー伝播を組み合わせて複数回の適用を行うと，同じ値を生成する式が，順次，字句形式が一致するものに変わるという副次的効果を反映させることができるが，コストが高いことが知られている[20]．

基本ブロック単位のハッシュ表を用いた値番号付け[1] を大域的な範囲に拡張した Rosen らの手法（以降，RWZ 法と呼ぶ）[20] は，SSA 形式の性質を利用して，同じ値を生成する式どうしを冗長除去の対象にできるので，PRE の副次的効果を反映できる．しかしながら，RWZ 法は，ループの抽出が必要であり，入力プログラムの構造に大きく依存する．

Steffen らの手法[23] は，計算式を DAG（directed acyclic graph）で表現し，CFG に沿って，伝播させることで各計算点の計算式表現を作成する．この方法は，ループを含まないプログラムに対して，RWZ 法と同じ最適化効果を保証しながら，RWZ 法のように，特定のプログラム構造を必要としない利点を持つ．しかしながら，解析コストが大きいという問題を持つ．

最近の値番号付けの研究では，PRE に忠実に式の挿入と冗長な式の除去を行い，SSA 形式を出力する手法に，Kennedy らの SSAPRE[10] がある．SSAPRE は，まず，式の出現位置についての支配辺境を計算することによって，式の値を保持する変数に対する**要素化冗長グラフ**（factored redundancy graph，以降 FRG と呼ぶ）を作成する．そのうえで，SSAPRE は，FRG 上で PRE を行う．FRG は疎なグラフなので，解析コストを低減させる効果がある．

SSAPRE と同様に SSA 形式を出力する手法に，VanDrunen らの GVN-PRE がある[24]．GVN-PRE は，SSA 形式上の同じ値を持つ，異なる変数や，字句形式が異なる式を，同じ“値”に写像することによって，コピー伝播なしに，PRE の副次的効果を反映することができる．特殊なグラフ構造を作成することなしに，フロー依存な変数をオペランドに持つ式の冗長性も除去することができる．しかしながら，GVN-PRE は，PRE に忠実なデータフロー方程式を使用していないので，PRE と同じ結果を得るには，複数回の適用が必要である．

本手法が行う PRE は，各式の出現ごとに適用する方法を，プログラムの開始点に近い方から適用していくので，特殊なグラフ構造を必要とせずに，副次的効果を反映していくことができる．また，ϕ 関数によるクエリの変更を行うので，フロー依存なオペランドを持つ式の冗長性も除去できる．解析コストも，データフロー解析 1，2 回程度である．

ループ不変コードの投機的移動を行うものには，RWZ 法を効率化した，Click の手法[5]

や，Lo らの保守的な投機的移動[14] がある．これらは，まずループの抽出を行わなければならず，本手法のように，任意のループを対象にした手法ではない．

また，プロファイル情報を用いる投機的な PRE として，Lo らのプロファイル駆動の投機的移動[14] や，投機的 PRE における最適コードを生成する Cai ら[4] や，Scholz らの手法[21] がある．本手法も，クエリの伝播の際に，プロファイル情報を用いることによって，同様な拡張が可能であると考えられる．

投機的移動を用いずに，同等な効果を得る PRE 法として，Bodik らの手法[3] がある．元の制御構造では，下向き安全でない挿入を，コードを複写することによって，下向き安全に行う．この手法は，コードを複写することでプログラムサイズを増大させるので，実践的には，プログラムの一部に適用することが想定されている．すなわち，従来法を補間的に用いることが効果的であるが，コードの複写は，プログラム構造を既約にする可能性があり，従来の LICM の適用は難しい．既約なプログラム構造を可約にする手法も提案されているが，さらなるコードの複写が必要になり，以降の最適化コストを増大させる．本手法は，任意のプログラム構造に適用可能なので，このような構造変形をともなう手法を補間する手法としても有効である．

8. ま と め

本稿では，質問伝播を投機的 PRE に応用し，各式の出現に対して，ループ不変式を投機的にループ外に移動できる PRE 手法を提案した．また，本手法を，プログラム中のすべての式の出現に適用することによって，いくつかのプログラムの実行効率を向上させることができることを実験で示した．その際の解析時間も，PRE を 1，2 回適用するのに要する程度である．

本手法は，命令スケジューリングのように，1 命令ごとに処理を行うコード最適化と，容易に組み合わせることができる．特に，本手法の投機的移動の性質を用いた投機的命令スケジューリングの実現は，今後の重要な課題である．

謝辞　本研究の一部は，文部科学省科学研究費補助金の補助を受けた．

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) Bodik, R., Gupta, R. and Soffa, M.L.: Complete Removal of Redundant Expressions, *Proc. Programming Language Design and Implementation* (*PLDI'98*), pp.1–14, ACM (1998).
4) Cai, Q. and Xue, J.: Optimal and Efficient Speculation-Based Partial Redundancy Elimination, *Proc. IEEE/ACM Int. Symposium on Code Generation and Optimization* (*CGO 2003*), San Francisco, pp.91–102, IEEE Computer Society (2003).
5) Click, C.: Global Code Motion Global Value Numbering, *Proc. Programming Language Design and Implementation* (*PLDI'95*), pp.246–257, ACM (1995).
6) COINS：並列化コンパイラ向け共通インフラストラクチャ. http://www.coins-project.org/
7) Cytron, R., Ferrante, J., Rosen, B.K. and Wegman, M.N.: Efficiently Computing Static Single Assignment Form and Control Dependence Graph, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.4, pp.451–490 (1991).
8) Dhamdhere, D.M.: Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.2, pp.291–294 (1991).
9) Dhamdhere, D.M. and Patil, H.: An Elimination Algorithm for Bidirectional Data Flow Problems Using Edge Placement, *ACM Trans. Prog. Lang. Syst.*, Vol.15, No.2, pp.321–336 (1993).
10) Kennedy, R., Chan, S., Liu, S., R.Lo, P.T. and Chow, F.: Partial redundancy Elimination in SSA Form, *ACM Trans. Prog. Lang. Syst.*, Vol.21, No.3, pp.627–676 (1999).
11) Knoop, J., Rüthing, O. and Steffen, B.: Lazy Code Motion, *Proc. Programming Language Design and Implementation* (*PLDI'92*), pp.224–234, ACM (1992).
12) Knoop, J., Rüthing, O. and Steffen, B.: Optimal Code Motion: Theory and Practice, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.4, pp.1117–1155 (1994).
13) Knoop, J., Rüthing, O. and Steffen, B.: The Power of Assignment Motion, *Proc. Programming Language Design and Implementation* (*PLDI'95*), pp.233–245, ACM (1995).
14) Lo, R., Chow, F., Kennedy, R., Liu, S. and Tu, P.: Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores, *Proc. Programming Language Design and Implementation* (*PLDI'98*), pp.26–37, ACM (1998).
15) Morel, E. and Renvoise, C.: Global Optimization by Suppression of Partial Redundancies, *Comm. ACM*, Vol.22, No.2, pp.96–103 (1979).
16) Morgan, R.: *Building an Optimizing Compiler*, Digital Press (1998).
17) Muchnick, S.S.: *Advanced Compiler Design and Implementation*, Morgan Kaufmann (1997).

18) Mueller, F. and Whalley, D.B.: Avoiding Uncoditional Jumps by Code replication, *Proc. Programming Language Design and Implementation* (*PLDI'92*), pp.322–330, ACM (1992).
19) Mueller, F. and Whalley, D.B.: Avoiding Coditional branches by Code replication, *Proc. Programming Language Design and Implementation* (*PLDI'95*), pp.56–66, ACM (1995).
20) Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Global Value Numbers and Redundant Computations, *Proc. Principles of Programming Languages* (*POPL'88*), pp.12–27, ACM (1988).
21) Scholz, B., Horspool, N. and Knoop, J.: Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination, *Proc. ACM SIGPLAN / SIGBED Int. Conf. Languages, Compilers and Tools for Embedded Systems* (*LCTES'04*), Washington, pp.221–230, ACM (2004).
22) Sreedhar, V.C. and Gao, G.R.: A Linear Time Algorithm for Placing ϕ-Nodes, *Proc. Principles of Programming Languages* (*POPL'95*), pp.62–73, ACM (1995).
23) Steffen, B., Knoop, J. and Rüthing, O.: The Value Flow Graph: A Program Representation for Optimal Program Transformations, *Proc. Int. Conf. European Symposium on Programming* (*ESOP'90*), Copenhagen, Denmark, pp.389–405, Springer-Verlag (1990).
24) VanDrunen, T. and Hosking, A.L.: Value-Based Partial Redundancy Elimination, *Proc. Int. Conf. Compiler Construction* (*CC'04*), LNCS, Berlin, pp.167–184, Springer-Verlag (2004).

(平成 21 年 5 月 8 日受付)
(平成 21 年 7 月 29 日採録)

滝本　宗宏（正会員）

1994 年慶應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持つ．ACM，IEEE，日本ソフトウェア科学会各会員．