関数型言語の並列要求駆動型計算における副作用解析†

鶴岡行雄†† 小野諭††

本論文では，構造データの破壊的更新を利用して関数型プログラムを効率的に実行するための新たな副作用解析法を提案する．本方法は，先に提案した計算経路解析をデータ共有性について拡張したものを基礎にしており，関数のノンストリクト性に起因する計算の排他性や順序性，引数/値間のデータ共有関係を，可能な計算経路ごとに明らかにできる．このため，ひとつの関数間解析のみで，ユーザ定義関数とノンストリクトプリミティブ関数とを統一した枠組みで処理できるようになった．対象は入れ子を許さない配列を扱う一階の関数型プログラムとし，計算規則としては，要求駆動型計算に対し並列性を引き出すようデマンド波及などを最適化した最適化要求駆動型計算を用いた．この解析の結果，無駄なコピーを伴うデータ更新操作を破壊的更新に置き換える最適化が可能となる．

1. はじめに

関数型言語は宣言型言語の一つであり，プログラムの意味はその実行方法とは独立に定まっている．そのため検証や変換が容易なほか，並列処理に適しているなどの利点がある．しかし，関数型プログラムの実行を考えたとき，効率の面で以下のような問題がある．

ひとつは，ユーザ定義関数のノンストリクト性を正しく実現する要求駆動型計算では，プログラムに内在する並列性を十分に引き出せないことである．また，代入やループ構造を持つ手続き型言語に比べて，対応する関数型プログラムの計算時間，メモリ使用量のオーダが高くなる場合があることも問題となる．

前者に対しては，安全性と並列性を両立させるための様々な広域解析技術や最適化アルゴリズムが提案されている[1)~3)]．また後者の原因には，同一計算の繰り返し，ループに対する再帰計算のオーバヘッド，構造データ更新時の無駄なコピーなどがある．同一計算の繰り返しに対しては，タビュレーション[4)]，スーパコンビネータ[5)]などによる最適化が適用できる．再帰計算のオーバヘッドに対しては，末尾再帰の除去[6)]，プログラム変換による補助変数の導入[7)]などの手法が知られている．

本論文で扱う構造データ更新の問題についても，無駄なコピーを伴う更新を破壊的更新に置き換える最適化が可能である．ただし，置き換えた破壊的更新が，副作用でプログラムの意味が変わることのない安全な更新であることを保証する必要がある．副作用解析は破壊的更新を行った場合の波及可能性を調べ，更新が安全であることを保証するために用いられる．

副作用解析においては，複数の値が同一のデータ構造を共有していること（共有性），複数の値の計算のうちいずれかひとつしか起きないこと(排他性)，ある値の計算は他の値の計算終了後に開始されること（順序性）などを検出することが重要である．共有性は，ある値を破壊した影響がデータ構造を経由して波及する範囲を求めるのに使用される．また，ある値の破壊的更新が，その値と排他的な関係にある値集合へ与える影響は無視できる．一般に引数が破壊されることの影響は，その引数を参照する計算の結果が求まる前後では異なる．したがって，計算の順序性を知ることにより副作用の範囲を限定することができる．

従来より関数型プログラムのこれらの性質を求める解析が提案されている．文献2)では，構造データを同一のメモリ領域に格納する最適化のため，排他性・順序性を検出する方法を提案している．文献8)では，共有性を含む解析法を提案している．しかし，これらはいずれも計算経路情報（結果を求めるために計算された引数・値の集合）の共通部分を求める必須引数解析[9)]の枠組みを基本としているため，ユーザ定義関数のノンストリクト性に起因する共有性，排他性，順序性などの検出が不十分であった．このため，必須引数以外の値の順序性に関する解析能力が低下するほか，条件関数（if-then-else）について特別な処理が必要になる，合成関数の性質が，参照している関数の性質のみからは計算できない，などの問題点があった．

本論文で提案する方法は，最適化要求駆動型計算[10)]の計算順序を定める関数間広域解析技術である計算経路解析[11),12)]をもとに，共有性を処理できるよう拡張

† Side Effect Analysis for Parallel Demand-Driven Computation of Functional Programs by YUKIO TSURUOKA and SATOSHI ONO (NTT Software Laboratories).

†† 日本電信電話(株)ソフトウェア研究所

した解析に基づいている．このため，計算規則が定める値計算の順序性を正確に解析することができる．また，この解析はユーザ定義関数についても可能な計算経路を独立に追跡するため，値の排他性が自然に把握でき，共有性も計算経路ごとに求めることができる．したがって，ひとつの関数間解析のみで，ユーザ定義関数とノンストリクトプリミティブ関数とを統一した枠組みで処理できるようになり，上記問題点を解決している．なお，解析の対象として入れ子を許さない配列を扱う一階の関数型プログラムを考え，計算規則は最適化要求駆動を前提とした．

2. 破壊的更新とコンフリクトセット

2.1 破壊的更新による副作用

関数型プログラムの実行においては，データ構造更新の副作用でプログラムの意味を変えることのない安全な更新を行う必要がある．

［例1］ 配列の二つの要素を交換する関数，

$swap(a, i, j) \equiv upd(upd(a, i, sel(a, j)), j, sel(a, i))$

を考える（図1）．また説明のため各式の結果に名前をつけて表すことにする．ここでは，

$s = sel(a, i)$, $t = sel(a, j)$, $u = upd(a, i, t)$,

$r = upd(u, j, s)$

と定義する．$sel(a, i)$ は配列 a の i 番目の要素を返す関数とする．また，$upd(a, i, v)$ は，i 番目の要素が v であり，それ以外の要素はすべて配列 a と同じであるような配列を返す関数とする．

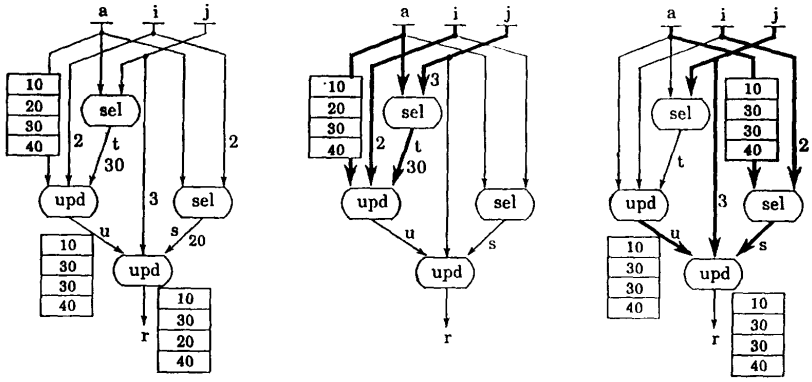
配列更新関数 $upd(a, i, v)$ の計算法として，配列 a の i 番目に直接 v を代入する方法（破壊的更新）と，配列 a をコピーし，それに対して更新を行う方法（コピーを伴う更新）があるが，このそれぞれについて $swap(a, 2, 3)$ の計算を考えてみる．ただし a は，要素が {10, 20, 30, 40} である配列とする．

upd に対してコピーを伴う更新を適用した場合には，更新によって upd の第一引数となる配列が破壊されることはなく，正しい結果を返す（図1（a））．

他方，破壊的更新を適用し，値 s より先に値 u が計算された場合を（図1(b-i)）に示す．この場合には，値 u の計算の副作用により引数 a が破壊される．このため，$sel(a, i)$ の結果である s に副作用が及び，値 r は誤った結果となる（図1(b-ii)）．したがって，破壊的更新による u の計算は安全ではない．破壊的更新の安全性を保証するためには，副作用の波及範囲を調べる必要がある．この解析の概要について次節で述べる．

2.2 コンフリクトセット

副作用解析は，破壊的更新に対するコンフリクトセットを求めるものである．コンフリクトセットとは，直観的には，更新の影響を受ける可能性のある値の集合のうち，更新時に計算が終了しているとは限らないものである．例1では，u を破壊的更新で計算した場合，u に対して {s} となる．コンフリクトセットを求めるためには，副作用の波及の要因となる，引数や値の共有性，排他性，順序性を関数間に渡って解析する必要がある．副作用解析は図2に示すように，関数間計算経路解析，関数内計算経路解析，データ共有解析，計算順序解析，コンフリクトセット解析から構成される．関数間計算経路解析では，すべての関数を同時に解析し，各関数の計算経路，関数の結果と引数間のデータ共有，関数適用による引数破壊などを求める



（a） コピーを伴う更新の場合　　（b） 破壊的更新の場合

図1 $swap(a, 2, 3)$ の計算例
Fig. 1 Examples of $swap(a, 2, 3)$.

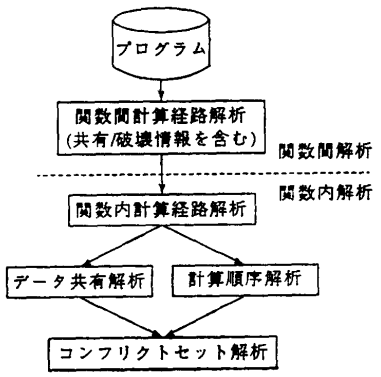


図 2 副作用解析の構成
Fig. 2 Structure of side-effect analysis.

ものである．関数内計算経路解析では，上記の関数間解析の結果をもとに，関数内の計算経路やデータ共有解析の基礎となる計算経路ごとの値の参照関係を求めるものである．データ共有解析は，関数内計算経路解析の結果から，関数内の値の共有関係により副作用が波及する可能性のある範囲を求める．計算順序解析は，関数内計算経路解析の結果より，値計算の順序関係を求める解析である．コンフリクトセット解析は，データ共有解析，計算順序解析の結果より，各破壊的更新のコンフリクトセットを導く．

2.3 副作用解析によるコンフリクトセットの計算例

コンフリクトセットと破壊的更新の安全性の関係について二つの例を用いて述べる．まず，例1で用いた $swap(a, i, j)$ について値 u のコンフリクトセットを考えてみる．

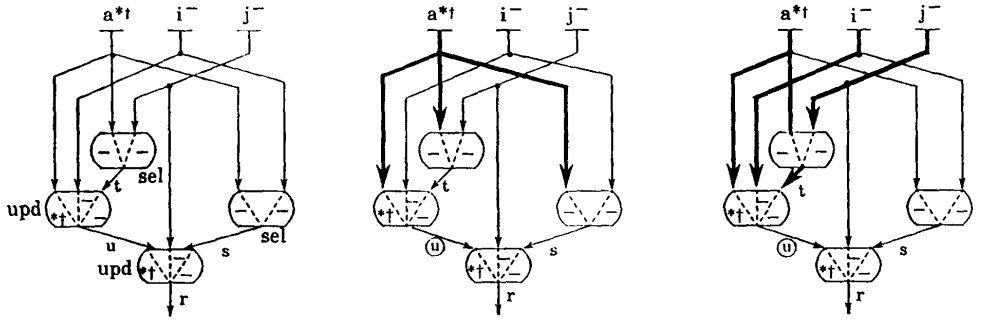
図 3(a)は関数間・関数内計算経路解析の結果を表したものである．関数適用によって引数と関数適用の結果がデータを共有する場合，その引数参照を透過モードの参照と呼び，引数に記号＊を付加して表す．また，関数適用により引数が破壊される引数参照を破壊モードの参照と呼び，引数に↑を付加して表す．非透過・非破壊モードの参照は，－を付加して表す．たとえば，$u=upd(a, i, t)$ を破壊的に計算したとき，引数 a は破壊され(↑)，同時に upd の引数と結果が共有により同一のデータをさすようになる(＊)．また，引数 i, t はともに，結果とデータを共有することはなく破壊されない(－)．

図3(b)はデータ共有解析の結果を表したものである．太線矢印で示された部分は，u の計算による値の破壊が波及する範囲を示している．これは一般に破壊的更新によって直接破壊される値から透過モードの参照をたどって到達できる値である．図3(c)は計算順序解析の結果であり，太線矢印は，値 u の計算開始時に計算が終了している値を表している．

これらより，u の計算開始時に，必ずしも計算は終了せず，かつ u の計算により破壊される値を参照するため $\{s\}$ が u のコンフリクトセットとなることがわかる．同様にして，値 r のコンフリクトセットは空集合となることがわかる．関数 $swap(a, i, j)$ では，ストリクト関数のみから定義されるため関数結果に対する計算経路はひとつであった．計算経路が複数ある場合には，データ共有・計算順序の解析は各計算経路ごとに行う．次の例は，複数の計算経路を持つノンストリクトな再帰関数の場合である．

[例2] $f(a, i, v)$

$\equiv if(i<1, a, f(upd(a, i, v), i-1, v))$



(a) 最適化要求駆動によるデータフローグラフと各関数の参照モード(計算経路解析結果)

(b) 値 u の計算時に破壊される値 a の副作用波及範囲(データ共有解析結果)

(c) 値 u の計算開始時に計算が終了している値の範囲(計算順序解析結果)

図 3 関数 $swap(a, i, j)$ におけるコンフリクトセット解析結果
Fig. 3 Results of conflict-set analysis for $swap(a, i, j)$.

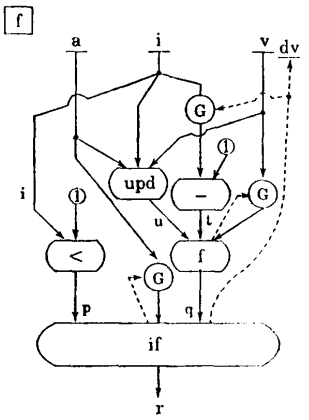


図 4 再帰関数 $f(a,i,v)$ のデータフローグラフ (最適化要求駆動)
Fig. 4 Dataflow-graph of $f(a,i,v)$ by optimized demand driven computation.

この例についても，各値に以下のように名前をつける．

$r=if(p,a,q)$, $p=i<1$, $q=f(u,t,v)$,
$u=upd(a,i,v)$, $t=i-1$

関数 f を最適化要求駆動で計算する場合のデータフローグラフを**図 4** に示す．図においてGで表されるゲート命令は，値に対するデマンドが発生するまで，その値を参照する計算を止めておくものである．また，点線はデマンドを表しており，デマンドの発生により止められていた計算が開始される．例では，最終結果となる *if* の *then* 部分と *else* 部分に対して，それぞれデマンドが付加される．*then* 部分に対するデマンドは，引数 a と *if* の第二引数の間のゲート命令に接続され，*then* 部分が必須となったときに *if* に a の値を渡すものである．*else* 部分が必須となったときは，デマンドは引数 i と値 t の間のゲート命令，および f の第三引数 v を計算するよう f の外側に波及する．最適化要求駆動の詳細は，文献 10) を参照されたい．

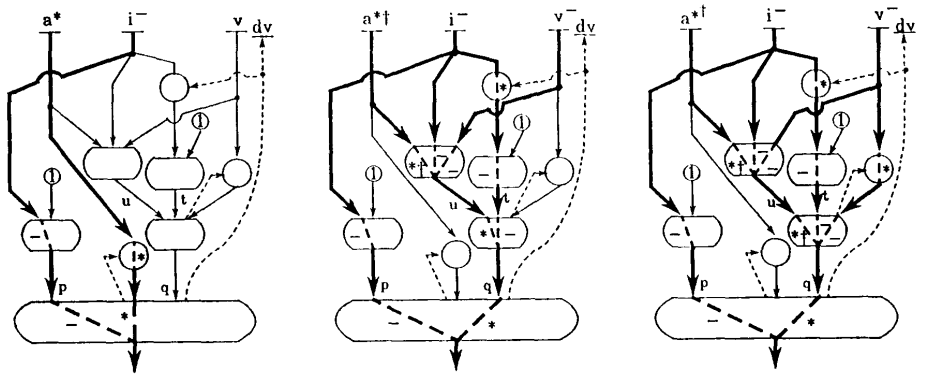
この関数 f は関数間・関数内計算経路解析により，**図 5** に示すように三とおりの計算経路を持っていることがわかる．ここでは，$i\geq 2$ のときの計算経路 (図 5 (c)) について値 q に対するコンフリクトを考えることにする．このとき破壊される値 u と共有関係にある値は，データ共有解析により $\{u,a\}$ となる (**図 6** (a))．また，値 q の計算開始時に計算が終了している値は，計算順序解析により，$\{p,i,u,a,v,t\}$ となる．(同期のためのゲート命令の計算結果は除いて考えている．) したがって，破壊的更新時に，破壊される値と共有関係にある値の計算はすべて終了しているため，この計算経路ではコンフリクトは生じない．このほかの計算経路でも q に対するコンフリクトはないため，この関数では q の計算で u が破壊されても安全であることがわかる．

3. 副作用解析の詳細

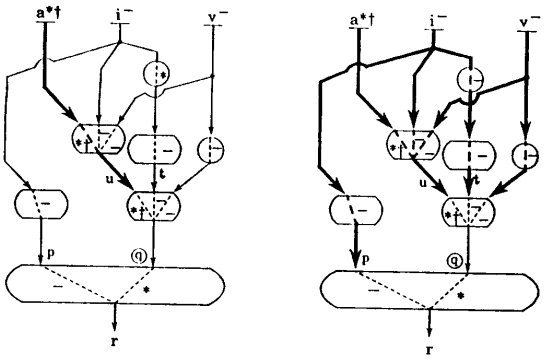
3.1 諸 定 義

データを共有するとは，同一の値を，複数の演算から (または同一の演算から異なる引数として) 参照する場合，また，演算の引数と結果がポインタにより同一のデータをさす場合である．

値の計算開始時とは，値が組み込み関数の演算結果である場合には，その関数の必須引数の計算がすべて終了し関数本体の計算を開始した時点とする．たとえば，$+(x,y)$ (加算) では引数 x,y が共に必須であるので，x と y の値の計算が終了し加算の処理を開始した時点である．また，$if(x,y,z)$ では，必須引数で



(a) $i\leq 0$ に対応する計算経路 (b) $i=1$ に対応する計算経路 (c) $i\geq 2$ に対応する計算経路
図 5 再帰関数 $f(a,i,v)$ に対する関数内計算経路解析結果
Fig. 5 Intra-functional computation paths of $f(a,i,v)$.



(a) 値 q の計算時に破壊される値 u の副作用波及範囲（データ共有解析）　(b) 値 q の計算開始時に計算が終了している値の範囲（計算順序解析）

図 6 関数 $f(a,i,v)$ の $i \geqq 2$ の場合におけるコンフリクトセット解析結果

Fig. 6 Results of conflict set analysis for $f(a,i,v)$ $(i \geqq 2)$.

ある x を計算し，真偽の判定を開始する時点である．これは *then* 部分/*else* 部分にあたる引数 y や z の計算を開始する前である．ユーザ定義関数の計算開始時は，関数適用を定義に基づき展開した時点とする．計算終了時とは関数の結果を返す時点である．値 s の計算終了後に値 u の計算を開始する場合を $s \rightarrow u$ と表し，値 s,u 間の計算順序と呼ぶ．たとえば，式 $if(x,y,z)$ では，x の計算終了後，y あるいは z の計算を開始するので $x \rightarrow y$ あるいは $x \rightarrow z$ である．

演算の結果から引数に向かって繰り返したどることにより u から s に到達できる場合，u は s に対してデータ依存関係にあるという．

値 s, u が以下の三つの条件を同時に満たす場合，s は u に対してコンフリクトしているという．

コンフリクト条件-1　u の計算開始時に，s の引数が u の引数とデータを共有する．

コンフリクト条件-2　$s \rightarrow u$ でない．

コンフリクト条件-3　u は s に対してデータ依存関係にない．

u に対してコンフリクトする値の集合を u のコンフリクトセットと呼び，$C(u)$ と表す．

3.2 関数間計算経路解析の計算法

関数間計算経路解析では，ユーザ定義関数の引数参照モードを関数間に渡って解析するものである．

解析結果は，関数の結果が求まる引数参照モードの組み合わせの集合で表される[13]．たとえば，$if(x,y,z)$ に対しては，$\{\{x^-,y^*\}\{x^-,z^*\}\}$ となる．これは，関数 $if(x,y,z)$ では，結果が求まる引数の組み合わせは，$\{x^-,y^*\}$ と $\{x^-,z^*\}$ の二つの場合があることを表している．(それぞれ x が *true* の場合と *false* の場合に対応する．) また，関数 $+(x,y)$ については，$\{\{x^-,y^-\}\}$ となる．結果が求まる引数の組み合わせは，$\{x^-,y^-\}$ のみであり，x と y は共に加算結果と値を共有せず，かつ破壊されないことを示している．$upd(a,i,x)$ を破壊的に実行した場合，$\{\{a^{*\dagger},i^-,x^-\}\}$ となる．また，$sel(a,i)$ は，$\{\{a^-,i^-\}\}$ となる．以上の性質を表す引数の組み合わせの集合を，PDPS (Parameter Dependency Property Set) と呼び，関数 f に対して，$D(f)$ と表す．

プリミティブ関数の PDPS は，あらかじめ定義されているものとする．ユーザ定義関数の PDPS は，関数定義において用いられる関数の PDPS を合成し簡約化することにより計算できる．ユーザ定義関数の PDPS の計算法は以下のとおりである．まず，PDPS を集合表現から代数式を λ 抽象した関数表現に変換する．変換は次の手順で行う．引数の組み合わせ集合どうしは，二項演算子＋で結合する．組み合わせ集合中の引数どうしは，二項演算子 $\times$ で結合する．参照モード $(*,\dagger)$ については単項演算子と考える．また，$\{x^{*\dagger}\}$ は $x^* \times x^\dagger$ と考える．そして得られた代数式を引数で λ 抽象したものを，PDPS の関数表現とする．たとえば，$if(x,y,z):\{\{x^-,y^*\}\{x^-,z^*\}\}$ に対しては，

$$\lambda(x,y,z).\ x^- \times y^* + x^- \times z^*$$

となる．簡約化規則は以下のとおりである．ただし $\alpha \in \{-,*,\dagger\}$ とする．

$$x \times x = x \qquad x + x = x$$
$$x \times y = y \times x \qquad x + y = y + x$$
$$(x \times y) \times z = x \times (y \times z) \qquad (x+y)+z = z+(y+z)$$
$$x \times (y+z) = x \times y + x \times z$$
$$(x+y)^\alpha = (x)^\alpha + (y)^\alpha \qquad (x \times y)^\alpha = (x)^\alpha \times (y)^\alpha$$

以下の規則は一つの値に対する複数の異なるモードの参照の簡約化を表す．

$$x^{*\dagger} = x^* \times x^\dagger \qquad x^- \times x^* = x^* \qquad x^- \times x^\dagger = x^\dagger$$

以下の規則は参照モードの合成を表す．

$$(x^-)^\alpha = x^- \qquad (x^\dagger)^\alpha = x^\dagger \qquad (x^*)^\alpha = x^\alpha$$

例として，三引数の逐次論理和である $or_3(a,b,c) \equiv if(a,a,if(b,b,c))$ の PDPS の計算を示す．

$$D(or_3)$$
$$\Rightarrow D(\lambda(a,b,c).\ if(a,a,if(b,b,c)))$$
$$\Rightarrow \lambda(a,b,c).\ D(if)aa(D(if)bbc)$$
$$\Rightarrow \lambda(a,b,c).\ (\lambda(x,y,z).x^- \times y^* + x^- \times z^*)aa$$

$$((\lambda(x,y,z).x^-\times y^*+x^-\times z^*)bbc)$$
$$\Rightarrow\lambda(a,b,c).a^-\times a^*$$
$$+a^-\times((\lambda(x,y,z).x^-\times y^*+x^-\times z^*)bbc)^*$$
$$\Rightarrow\lambda(a,b,c).a^-\times a^*+a^-\times(b^-\times b^*+b^-\times c^*)^*$$
$$\Rightarrow\lambda(a,b,c).a^*+a^-\times(b^*+b^-\times c^*)$$
$$\Rightarrow\lambda(a,b,c).a^*+a^-\times b^*+a^-\times b^-\times c^*$$
$$\Rightarrow\{\{a^*\}\ \{a^-,b^*\}\ \{a^-,b^-,c^*\}\}$$

となる．

また再帰関数の場合は，同様な計算がすべて収束するまで繰り返す．二章で用いた関数 $f(a,i,v)$（図4）の例については，$\{\{a^*,i^-\}\ \{a^{*\dagger},i^-,v^-\}\}$ となる．詳細は，文献 9)，11) を参照されたい．

3.3 関数内計算経路解析の計算法

ある値を計算するために参照された値と参照モードを演算の結果側から引数側に向かって連接したものを**参照連鎖**と呼ぶ．関数内計算経路解析では，計算経路ごとの参照連鎖を求めるものである．これを，RCP (Reference Chain Property) と呼び，値 v に対して $R(v)$ と表す．以下に，RCP の計算法を示す．値 v に対して，

（i）v が着目する関数定義の引数である場合は，$R(v)=1$ とする．

（ii）v が引数でない場合は，v が参照する値の RCP の合成・簡約化で計算する．

合成法は以下のとおりである．まず，$R'(v)=v\cdot R(v)$ と定義する．ただし "・" は右結合の二項演算子とする．v を与える関数を f，その引数を $a_1,a_2\ldots a_n$ としたとき，$R(v)=D(f)(R'(a_1),R'(a_2),\ldots,R'(a_n))$ となる．これは，$R'(a_1),R'(a_2),\ldots,R'(a_n)$ への関数 $D(f)$ の適用である．また，簡約化は，関数間解析に用いた規則と以下の規則により行う．

$$x\cdot 1=x \qquad (x\cdot y)^\alpha=x^\alpha\cdot y$$
$$x\cdot(y+z)=x\cdot y+x\cdot z$$
$$x\cdot(y\times z)=x\cdot y\times x\cdot z$$

関数 *swap*（図3）については，$R(r)=\{\{j^-,\ u^{*\dagger}\cdot i^-,\ u^{*\dagger}\cdot a^{*\dagger},\ u^{*\dagger}\cdot t^-\cdot a^-,\ u^{*\dagger}\cdot t^-\cdot j^-,\ s^-\cdot a^-,\ s^-\cdot i^-\}\}$ となる．結果は，ひとつの要素からなる集合であり，*swap* の計算経路はただ一つであることがわかる．また，計算経路中の各要素は，最終結果 r から各引数へ至る参照連鎖である．たとえば二番目の参照連鎖 $u^{*\dagger}\cdot i^-$ を考えると，値 r を生じる演算が値 u を透過・破壊モードで参照し，その u を生じる演算が値 i を非透過・非破壊モードで参照することを表している．したがって，この参照経路では r の計算で u は破壊されるが i は透過モードで参照されないため副作用は i には及ばないことなどがわかる．同様に，関数 $f(a,i,v)$ については，$R(r)=\{\{p^-\cdot i^-,\ a^*\}\ \{p^-\cdot i^-,\ q^*\cdot u^*\cdot v^-,\ q^*\cdot u^*\cdot a^{*\dagger},\ q^*\cdot u^*\cdot i^-,\ q^*\cdot t^-\cdot i^-\}\ \{p^-\cdot i^-,\ q^*\cdot v^-,\ q^*\cdot u^{*\dagger}\cdot v^-,\ q^*\cdot u^{*\dagger}\cdot a^{*\dagger},\ q^*\cdot u^{*\dagger}\cdot i^-,\ q^*\cdot t^-\cdot i^-\}\}$ となる．

3.4 データ共有解析の計算法

データ共有解析は，最終結果に対する RCP を用いて計算経路ごとに行う．ある値と共有関係にある値は，その値から引数側に向かって透過モードの参照が続くかぎりたどっていき，たどる途中の値からさらに結果側に向かって透過モードの参照をたどることで求める．たとえば，関数 *swap*（図3）では値 u について $\{a,t,s\}$ となり，関数 $f(a,i,v)$（図6（a））において $i\geq 2$ のとき，q について $\{u,a\}$ となる．

3.5 計算順序解析の計算法

最適化要求駆動では，値の必須性に基づいて計算が進められる．計算順序が生じる要因としては，データの依存性によるもの，デマンドによるものがある．

データ依存性による計算順序とは，演算に必須な値と演算結果間の計算順序である．関数 *swap*（図3）の例では，値 u の計算に必須な値 $\{a,i,t,j\}$ に対して，$\{a,i,t,j\}\rightarrow u$ となる．同様に関数 $f(a,i,v)$（図6（b），$i\geq 2$）の例では，値 q の計算に必須な値 $\{a,i,v,u,t\}$ に対して，$\{a,i,v,u,t\}\rightarrow q$ となる．

デマンドによる計算順序とは，デマンド発生時に計算が終了している値と，デマンドにより計算が開始される値の間の計算順序である．並列処理環境では一般に，デマンドはデマンド発生以前に必須であった計算の終了を待たずに発生するが，本論文では簡単のため，デマンド発生以前に必須であった値のうち可能な計算がすべて終了した時点でデマンドが発生するものとする．関数 $f(a,i,v)$（図6（b），$i\geq 2$）の例では，値 q に対するデマンド以前に計算が終了する値 $\{a,i,p\}$ と，q に対するデマンド以降に必須となる値 $\{v,u,t,q\}$ の間の計算順序 $\{a,i,p\}\rightarrow\{v,u,t,q\}$ である．

これらの計算順序は以下の手順で求める．まず，前節で述べた RCP に対して簡約化規則，$x^\alpha\cdot y^\beta=x\times y$（$\alpha,\beta\in\{*,\dagger,-\}$）を適用することにより，計算経路情報のみを取りだす．これを値 v に対して $V(v)$ と表す．たとえば，関数 $f(a,i,v)$ の値 r に対する計算経路は，$V(r)=\{\{a,i,p\}\ \{a,i,v,p,q,u,t\}\}$ となる．これは，値 r を計算するために必要となる値の組み合わせの集合である．値 r に対して必須な値は，この

組み合わせに共通な値であり（$\cap_{m \in V(r)} m$），上記の例では，$\{a, i, p\} \cap \{a, i, v, p, q, u, t\} = \{a, i, p\}$ となる．これよりデータ依存性による計算順序 $\{a, i, p\} \rightarrow r$ が得られる．

デマンド以前に必須となる値と，デマンド以降に新たに必須となる値を，計算経路情報から解析する方法については，文献10)を参照されたい．

3.6 コンフリクトセットの計算法

コンフリクトセットは，各計算経路ごとにデータ共有解析，計算順序解析の結果から求める．例として，関数 *swap*（図3）における値 u のコンフリクトセットについて考える．データ共有解析の結果から，u に対して 3.1 節で述べたコンフリクト条件-1 を満たす値は $\{a, t, s\}$ となる．計算順序解析の結果から，u に対してコンフリクト条件-2 を満たさない値は $\{a, i, j, t\}$ となる．また，u に対してコンフリクト条件-3 を満たさない値は $\{a, i, j, t\}$ であることから，$C(u) = \{a, t, s\} - \{a, i, j, t\} - \{a, i, j, t\} = \{s\}$ となる．同様に，$f(a, i, v)$ の $i \geq 2$ の場合において，q についてのコンフリクトは以下のようになる．コンフリクト条件-1 を満たす値は $\{a, u\}$ である．コンフリクト条件-2 を満たさない値，すなわち q の計算開始時に計算が終了している値は，3.5 節で述べたデータ依存性，デマンドによる計算順序より $\{a, i, v, u, t, p\}$ となる．コンフリクト条件-3 を満たさない値は $\{a, i, v, u, t\}$ である．したがって，$i \geq 2$ のとき $C(q) = \{a, u\} - \{a, i, v, u, t, p\} - \{a, i, v, u, t\} = \phi$ となる．

4. 破壊的更新による最適化および解析システム

4.1 破壊的更新による最適化法

二章で用いた例について，破壊的更新による最適化を考える．関数 $f(a, i, v)$（図4）では，図5に示す三とおりの計算経路のうち，*if* の計算で else 部分を選択する二つの計算経路において u が参照され，このときともに $C(u) = \phi$ である．したがって，u の計算に破壊的更新を適用できる．

関数 $swap(a, i, j)$（図3）では，計算経路はひとつであり，値 r に対して $C(r) = \phi$ となる．したがって破壊的更新が可能である．値 u に対しては $C(u) = \{s\}$ である．この場合には破壊的更新は安全でない．

一般に，$C(u)$ に含まれるすべての s に対して $s \rightarrow u$ となるよう（すなわちコンフリクト条件-2 を満たさないよう）計算を逐次化することにより，u の計算に破壊的更新を適用できる．関数 *swap* の例では，$s \rightarrow u$ となるよう逐次化することで u を破壊的に計算できる．このように逐次化が必要となる最適化では，構造データのコピーの手間と逐次化による計算の遅延とがトレードオフの関係となる．たとえば，逐次化による計算の遅延時間や遅延される計算量が少なく，更新する構造データが大きい場合には，逐次化して破壊的更新を適用する最適化は有効である．

また，計算経路ごとのコンフリクトセットの解析から，以下の詳細度の異なる最適化が考えられる．ふたつの更新方法を計算経路ごとに静的に決定し，実行時の計算経路情報を用いて動的に更新方法を切り替える最適化では，自由度は大きいが実行コードは複雑になる．また，更新演算が必須となるすべての計算経路においてコンフリクトが解消できる場合のみ，それらの計算経路すべてについて静的に破壊的更新を適用する最適化が考えられる．この場合には実行コードの単純な置き換えで実現できるが前者に比べ最適化の機会は限られる．

4.2 解析システムによる解析例

関数間・関数内計算経路解析システムを Common Lisp にて作成した．解析プログラムの規模は 600 行程度である．データ共有解析，計算順序解析については現在は作成中である．

再帰関数 $f(a, i, v)$（図4）について，関数間計算経路解析の結果（$D(f)$）と，値 r についての関数内計算経路解析の結果（$R(r)$）を**図7**に示す．$D(f)$ については三度目の繰り返しで収束している．なお，参照モードの † は，＋で表している．$R(r)$ より，*Path-a*, *Path-b*, *Path-c* の三とおりの関数内計算経路がある

```
>(analyze-path 'f)

loop=1 : D(F) = ((*A -I))
loop=2 : D(F) = ((*A -I) (*+A -V -I))
loop=3 : D(F) = ((*A -I) (*+A -V -I))
PDPS converged.

R(R) =
Path_a : ((-P -I) (*A))
Path_b : ((-P -I) (*Q *U -V) (*Q *U *+A) (*Q *U -I) (*Q -T -I))
Path_c : ((-P -I) (*Q -V) (*Q *+U -I) (*Q *+U *+A) (*Q *+U -V) (*Q -T -I))

>
```

図7 解析システムによる $f(a, i, v)$ の計算経路解析結果
Fig. 7 Analysis for $f(a, i, v)$ by analyzer system.

ことがわかる．これらは，図5の(a)，(b)，(c)にそれぞれ対応する．

5. むすび

本論文では，構造データの破壊的更新操作を利用して関数型プログラムを効率的に実行するための新たな副作用解析法を提案した．従来の副作用解析では，計算経路情報の共通部分を求める必須引数解析の枠組みを基本としていたため，ノンストリクト性を持つユーザ定義関数について，共有性，排他性，順序性などの検出が不十分になっていた．本論文で提案した方法は，最適化要求駆動型計算の計算順序を定める計算経路解析をもとに，構造データの共有性を検出できるよう拡張した解析を基本にした．また，関数の性質を求める際には，可能な計算経路それぞれを独立に追跡するようにした．この結果ユーザ定義関数についてもノンストリクトプリミティブ関数の場合と同様な解析力が得られ，また，ひとつの関数間解析に基づく統一した枠組みの中で，議論できるようになった．なお，解析の対象は入れ子を許さない配列を扱う一階の関数型言語とし，計算規則としては，最適化要求駆動を用いた．

本論文で提案した解析法は，共有性，排他性に関して以下に示す限界を持つ．データ共有解析では配列の要素に立ち入った解析を行わないため，たとえば $upd(a, 2, v)$ と $sel(a, 3)$ が異なる要素をアクセスするため実際はコンフリクトしないことを検出できない．ただし，解析結果が真に共有関係にある値を必ず含むという意味で安全な近似となっている．また計算経路解析の結果が到達不能な経路を含む場合などがある[3]．たとえば $f(x, y)=if(x>0, 1, if(x>1, y, x))$ に対して，実際には到達不能な y を返す経路が解析結果に含まれる．(真の計算経路は決定不能である．詳細は文献3)を参照されたい．) この場合も，解析結果が真の計算経路を必ず含むため安全な近似となっている．最適化要求駆動では計算経路解析の結果により計算順序を定めるため，順序性については正確に解析できる．

配列の入れ子に対しては，配列の要素への参照を追跡するためのタイプ推論と組み合わせることで解析が可能となる．この場合には，どの深さまで入れ子を追跡するかによって解析の精度が決まる．また，リストの第一層に対する解析など，実行時に解析対象の構造が決まる場合には，解析結果は無限の構造となり解析アルゴリズムが停止しないという問題が生じる．このためには無限の構造を有限表現に抽象化するなどのアプローチが必要である．今後は，リストやストリームなど動的データ構造への適用法を考えていきたい．

謝辞 本研究を進めるにあたり御指導いただいた塚本克治ソフトウェア基礎技術研究部長，武末勝グループリーダ，御助言をいただいた高橋直久主任研究員，小川瑞史研究主任をはじめ NTT ソフトウェア基礎技術研究部の諸氏に感謝します．

参考文献

1) Clack, C. D. and Peyton Jones, S. L.: *Strictness Analysis—A Practical Approach*, *Functional Programming and Computer Architecture*, LNCS 201, pp. 35-49, Springer-Verlag (1985).
2) 関，井上，谷口，嵩：関数型言語 ASL/F のコンパイル時における最適化，信学論(D)，Vol. J67-D, No. 10, pp. 1115-1122 (1984).
3) 小野，高橋：再帰関数系における依存属性集合の計算法，信学論(D)，Vol. J69-D, No. 5, pp. 714-723 (1986).
4) Bird, R. S.: Tabulation Techniques for Recursive Programs, *ACM Comput. Surv.*, Vol. 12, No. 4, pp. 403-417 (1980).
5) Hughes, R. M. J.: Super-combinators, A New Implementation Method for Applicative Languages, *1982 ACM Symp. on Lisp and Functional Programming*, pp. 1-10 (1982).
6) Aho, A. V., Hopcroft, J. E. and Ullman, J. D.: *Data Structures and Algorithms*, pp. 66-69, Addison-Wesley (1983).
7) Burstall, R. M. and Darlington, J.: A Transformation System for Developing Recursive Programs, *J. ACM*, Vol. 24, pp. 44-67 (1977).
8) Hudak, P. and Bloss, A.: The Aggregate Update Problem in Functional Programming Systems, *12th ACM Symp. on Prin. of Prog. Lang.*, pp. 300-314 (1985).
9) 小野：関数型プログラムのストリクト性関連解析と最適化技術，情報処理，Vol. 29, No. 8, pp. 862-871 (1988).
10) 小野，高橋：関数型言語における要求駆動型評価の並列処理向き最適化，信学論(D)，Vol. J70-D, No. 2, pp. 259-268 (1987).
11) Ono, S.: Computation Path Analysis: Toward an Autonomous Global Dataflow Analysis, *2nd France-Japan Artificial Intelligence and Computer Science Symposium*, INRIA, pp. 45-68 (1987).
12) 小川，小野：広域データフロー解析に基づく関数型プログラムの変則性検出，信学論(D)，Vol.

J71-D, No. 10, pp. 1949-1959 (1988).
13) 鶴岡，小野：並列要求駆動型計算における副作用の解析，並列処理シンポジウム JSPP '89, pp. 41-48 (1989).

(平成元年6月 8 日受付)
(平成元年9月 12日採録)

鶴岡 行雄（正会員）

1962 年生．1985 年電気通信大学電子工学科卒業，1987 年同大学院応用電子工学科修士課程修了．同年日本電信電話(株)入社．以来，関数型言語，並列処理の研究に従事．現在，NTT ソフトウェア研究所に勤務．電子情報通信学会会員．

小野 諭（正会員）

1954 年生．1977 年東京大学工学部電子工学科卒業，1979 年同大学院修士課程，1982 年博士課程修了．同年日本電信電話公社入社．以来，データフロー計算機，関数型言語，並列処理の研究に従事．現在，日本電信電話(株)NTT ソフトウェア研究所に勤務．電子情報通信学会，ACM 各会員．