一般化 max 演算子の自動抽出

佐藤 重幸　森畑 明昌

max 演算子は，プログラム記述を簡潔にし，計算機の上で効率良く実行され，コンパイラによるプログラム変換を助ける，大変有益な言語要素である．しかし，その重要性をプログラマが認識しているとは限らず，max 演算相当の計算が if 文で記述されることも珍しくない．そこで本研究では，max 演算に気付かないプログラマにも max 演算子の恩恵をもたらすために，複雑な if 文のネストから，max 演算子を抽出して，if 文を簡単化する手法を開発する．そして，より一般的な max 演算子も抽出できるようにする拡張も与える．提案手法は，特にリダクションループ並列化への応用において，既存手法に対する優位性がある．

1 導入

より大きいオペランドを返す 2 項演算子として max 演算子（及びより小さい方を返す min 演算子）がある．多くのプログラミング言語に，組込み関数や標準ライブラリ関数の形で提供されている．その理由は，ひとえに記述が簡潔で明快だからである．例えば，文字列 a と b のレーベンシュタイン距離 lev(a,b,len(a),len(b)) を求める関数 lev は，次のように可変長引数関数の max と min を使って簡潔且つ明快に定義できる．

```
def lev(a, b, i, j):
  if min(i,j) == 0:
    return max(i,j)
  else:
    c = int(s[i] != t[j]) # 0 or 1
    return min(lev(a, b, i-1,   j) + 1,
               lev(a, b,   i, j-1) + 1,
               lev(a, b, i-1, j-1) + c)
```

しかし，もしもすべての max と min を if 文で記述すると，例えば，次のように冗長になる．

```
def lev(a, b, i, j):
  if i < j and i == 0:
    return j
  elif j < i and j == 0:
    return i
  else:
    c = int(s[i] != t[j]) # 0 or 1
    r1 = lev(a, b, i-1,   j) + 1
    r2 = lev(a, b,   i, j-1) + 1
    r3 = lev(a, b, i-1, j-1) + c
    if r1 < r2:
      if r2 < r3:
        return r3
      else:
        return r2
    else:
      if r1 < r3:
        return r3
      else:
        return r1
```

max 演算子（以降では，min 演算子も暗黙に含める）を使った方が，人間にやさしい（human-friendly）のは明らかである．

実は，max 演算子は，単に記述が人間にやさしいだけではなく，計算機にやさしい（machine-friendly）

---

* Automatically Extracting Generalized Max Operators.

Shigeyuki Sato, 東京大学 大学院 情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.
Akimasa Morihata, 東京大学 大学院 総合文化研究科, Graduate School of Arts and Sciences, The University of Tokyo.

性質も持つ．例えば，x86 命令セットには，cmov 命令（条件付きコピー）があり，max 演算子は，直接的に cmov 命令に対応する．cmov 命令は，分岐命令ではないので，命令パイプラインにとって都合が良く，分岐予測ミスによる性能低下のリスクが無い．したがって，効率良く計算機で実行する上で，回路レベルの複雑さが if 文とは大きく異なる．また，現代の命令セットアーキテクチャには，SIMD 命令として広く max 命令があり，様々なアーキテクチャでレイテンシが小さい [4] ことが知られている．

更に，max 演算子の持つ代数的性質（結合性や可換性）は，プログラム変換にとって大変都合が良い（compiler-friendly）．max 演算子による畳み込みは，分割統治による並列リダクションが即座に可能であるし，より複雑な max-plus 半環を用いた並列リダクション [8,10,13] も，max 演算子があれば簡単に導出可能になる．

これほど嬉しいことの多い max 演算子を，プログラマがきちんと利用するかというと，残念ながらそうではない．そもそもプログラマは，max 演算を行っているとは認識せずに，if 文で計算を考えて記述することが珍しくない．特に命令型プログラミングでは，

```
x = max(x, a)
```

と記述するよりも，

```
if x < a:
  x = a
```

と記述する方が，素直であるという見方すらできる．いくら max 演算子を使うと簡潔になるとしても，それに気付けないなら，プログラマにとって max 演算子を使うことがやさしいとは言えない．特に，比較演算を一般化した max 演算子（例えば目的関数を最大化する argmax）ともなってくると，それが if 文で書かれていたときに，max 演算子の一種であると認識するのは，より非自明になる．加えて，当初は max 演算ではなかった if 文に，プログラミングの過程で結果的に max 演算が紛れ込むこともある．複雑な if 文の入れ子の中に紛れ込んでいる max 演算を峻別して括り出すリファクタリングは，仮にできたとしても間違えやすいものである．したがって，max 演算子を明示的に使われない場合でも，その恩恵を受け取れるようにすることには価値がある．

そこで本研究では，SMT ソルバによる検証に基づいて，if 文の入れ子から max 演算子を自動的に抽出する方法を示す．提案手法は，max 演算子の抽出によって，入力の if 文が確実に単純化されるため，リファクタリングの自動化として機能する．そして，より一般化された max 演算子についても対応できるように，提案手法を拡張する方法も示す．提案手法の応用として特に有望なのは，リダクションループ並列化である．本論文では，リダクションループ並列化の文脈における提案手法の恩恵も示す．

本研究の貢献は次の通りである．

- max 演算子を SMT ソルバを用いて if 文から抽出する方法を開発した（3 節）．提案手法は，複雑な if 文の入れ子から，max 演算部分を抽出して，より簡単なプログラムを構成する．
- より一般的な max 演算子を抽出するための拡張を提案手法に加えた（4 節）．提案手法は，目的関数を最大化する所謂 argmax や，順序関係が一般化された max 演算子も抽出可能である．
- 提案手法をリダクションループ並列化への応用したときの有用性を示した（5 節）．提案手法は，既存手法 [10] よりも，他のプログラム変換 [7] と組み合わせ易く，モジュラーである．

2 入力言語

入力言語は，Python 構文を借りた命令型言語を仮定する．構文はほぼ Python[†1] だが，意味論の基本として，次の 3 点を仮定する．

- 暗黙的な静的型付き
- 基底型は真理値型と数値型
- 関数や代入の作用は静的に追跡可能

これらは，提案手法が直接要求するものというよりも，入力プログラムの式を SMT ソルバに解釈させるときに，一般的に要求される制約である．別の言い方をすれば，入力言語の式は，ほぼそのまま SMT ソルバで検証可能なものであると仮定する．尚，本論文の

†1 記述の簡潔のために，仮引数のパターンマッチ（例えば lambda (x,y): x + y）を加えて拡張している．

以降では，SMT ソルバのことを単に検証器と呼ぶ．

以上に加えて，提案手法の説明を簡単にするために，次の 2 点を仮定する．

- 代入文の左辺が変数
- ブロックスコープの変数

これらは，入力プログラムを正規化したり，書換えたりするときに都合が良いものである．提案手法の技術的要点に関わるものではない．

尚，左辺が変数以外の代入文や関数呼出しについては，その作用の影響範囲を考える必要がある．print 関数のような計算作用は，それが max 演算には関わらないので，即座に無視できる．配列要素に値が代入された場合でも，その作用を静的に追跡して，変数のように配列要素を扱える場合には問題無い．

提案手法は，loop-free プログラム断片に適用することを前提としている．入力言語がより複雑でも，適用対象のプログラム断片について，上記の条件が満たされているなら問題は無い．

3 提案手法：max 演算子の抽出

提案手法の骨子は，次の 3 ステップである．

1. オペランドの見積り
2. 条件式の検証
3. max 式の構成と if 文の書換え

このうち，技術的要点は，ステップ 1 と 2 である．

提案手法の説明を簡単にするために，与えられた loop-free プログラム断片に対して，次の前処理を与えて正規化する．

- 変数の単一代入化．全ての代入文について，その右辺式を後続の右辺式へ記号的に代入．
- if 文の条件部の正規化．then/else 節を複製することで，and や or を伴わない形に変換．
- if 文の巻き上げ．if 文の連接は，ネストした if 文に変換．
- else 節の完備化．自己代入文を含んだ else 節を用いて，全ての if 文に else 節を導入．

尚，入力プログラムを正規化しなくても，プログラムポイント毎の記号的環境を（例えば [5] の要領で）準備すれば，提案手法の適用に過不足は無い．

正規化の結果，例えば次の if 文の塊を得たとする．

```
if x + 1 < a:
  if x > 0:
    x = a
 else:
    x = 0
else:
  if x > 0:
    x = x + 1
  else:
    x = 0
```

本節の以降では，ここから，max 演算子を抽出することを考える．

3.1 ステップ 1：オペランドの見積り

ステップ 1 では，適当な if 文を設定して，その中に max 演算子のオペランド候補を見つける．具体的には，

```
if x + 1 < a:
  if x > 0:
    x = a
  else:
    x = 0
else:
  if x > 0:
    x = x + 1
  else:
    x = 0
```

この下線部が，max 演算に**置換されうる**ことを検証する．

ここで思い出すべきことは，max 演算子を抽出した後に，それに対応する if 文を max 式に変換することが本来の目的であるということである．対象の if 文と代入文から，それらと等価な max 式を抽出しても，対象の if 文が置換できないなら目的を満たさない．例えば，次のプログラムにおいて，

```
if x + 1 < a:
  if x > 0:
    x = a
  else:
    x = 0
else:
  if x > 0:
    x = 0
  else:
    x = x + 1
```

下線部について max 演算子を抽出して正規化すると

次を得る．

```
if x + 1 < a:
  if x > 0:
    x = max(x + 1, a)
  else:
    x = 0
else:
  if x > 0:
    x = 0
  else:
    x = max(x + 1, a)
```

これは，単純化しても `x + 1 < a` による条件分岐を除去できず，むしろ max 演算子が除去されてしまう．このようなオペランドの選び方を排除して，もし max 演算子が抽出できたとしたら，対象 if 文を除去できるような候補を発見する．方法としては，then/else 節のそれぞれのオペランド候補となる右辺式を，適当な共通変数 t で置換した上で，then/else 節の等価性を検証する．最初の例の場合では，

```
if x + 1 < a:
  if x > 0:
    x = t
  else:
    x = 0
else:
  if x > 0:
    x = t
  else:
    x = 0
```

を構成し，この then 節と else 節の等価性を検証する．

3.2 ステップ 2：条件式の検証

ステップ 2 では，対象 if 文が，オペランド候補に対する max 演算を行っているかを検証する．基本的なアイデアは，対象 if 文の条件部の比較式と，オペランド候補の順序比較との等価性検証であるが，その検証方法に一工夫を加える．

さて，then 節内のオペランド候補を if 式でまとめた e_t と else 節内のオペランド候補を if 式でまとめた e_e は，抽出する max 演算子のオペランド式となる．したがって，対象 if 文の条件部が $e_1 < e_2$ であるとしたとき，

$$(e_e < e_t) = (e_1 < e_2) \qquad (1)$$

であることが検証できれば，対象 if 文の条件部が max 演算と同等の条件式であることがわかる．

このように比較演算の結果を比較する手法は，対応するオペランドを直接を比較するよりも，記号的差異に対して頑健である．例えば，最初に挙げた例では，

$$e_1 = \texttt{x + 1}$$
$$e_2 = \texttt{a}$$
$$e_t = \texttt{a}$$
$$e_e = \texttt{x + 1}$$

であり，条件 (1) を満たす．ここでは，オペランド候補の `a` と `x + 1` が，条件部に直接現れるので，対応するオペランドの直接比較でも問題ない．一方，

$$e_1 = \texttt{x + 2}$$
$$e_2 = \texttt{a + 1}$$

の場合には直接比較では等価性検証に失敗する．しかし，このような場合でも，条件 (1) は満たされるので，max 演算子が抽出できる．

オペランド候補でない右辺式を無視して e_t と e_e を構成することは，then/else 節単独では等価ではないが，それぞれの事前条件の下で等しくなる場合を自然に対処する．例えば，

```
if x + 1 < a:
  x = a
else:
  if x + 1 < a:
    x = 0
  else:
    x = x + 1
```

を考える．もし，この対象 if 文の else 節全体から e_e を構成すると，$e_t - e_e$ = `a - (0 if x + 1 < a else x + 1)` となり，条件 (1) を満たさない．しかし，下線部のみを残すように else 節内の if 文を枝刈りすると，$e_t - e_e$ = `a - (x + 1)` となるので，条件 (1) を満たす．この枝刈りは，一般にはプログラムの意味を変える．しかし今，ステップ 1 で対象 if 文内のオペランド候補以外の右辺式（及びその代入）は，then/else 節で等価であることが検証されているので，この枝刈りは安全である．

3.3 ステップ 3：max 式の構成と if 文の書換え

ステップ 3 では，ステップ 2 で構成したオペランド式 e_t と e_e を元に max 式を実際に構成して，ステップ 1 で構成したオペランド部分が抽象化された if 文を元に，対象 if 文を書換える．具体的には，最初に挙げた例の場合，

```
t = max(x + 1, a)
if x + 1 < a:
  if x > 0:
    x = t
  else:
    x = 0
else:
  if x > 0:
    x = t
  else:
    x = 0
```

と一時変数 `t` を用いて，max 式を挿入する．このとき，対象 if 文は，ステップ 1 で then/else 節の等価性検証を行った if 文と等しいことに注目してほしい．したがって，その等価性検証の過程で構成した，then/else 節に対応した式を利用すれば，

```
t = max(x + 1, a)
x = 0 if x > 0 else t
```

が得られる．更に max 演算子の抽出を続けるために，改めて正規化すると，次のプログラムを得る．

```
if x > 0:
  x = max(x + 1, a)
else:
  x = 0
```

この後，max 演算子が抽出できる限り，ステップ 1–3 を再帰的に繰り返す．この例では，これ以上 max 演算子を抽出できないので，終了である．

3.4 補遺：単純化に基づく場合

Z3[†2] などの検証器は，式の単純化の機能を提供する．この単純化を用いると，もう少し簡単且つ一般的に max 演算子抽出を実装できるようになる．

まず，ステップ 1 では，等価性検証の代わりに，単純化を実行して，オペランド候補が抽象化された対象 if 文（下線部）が除去されるか調べればよい．

ステップ 2 では，適当な変数 C を利用して，対象 if 文の条件部を含む形で then/else 節にそれぞれに対応する式 e'_t と e'_e を構成する．具体的には，

```
if x + 1 < a:
  x = a
else:
  if x + 1 < a:
    x = 0
  else:
    x = x + 1
```

の場合，

$$e'_t = \texttt{a if x + 1 < a else } C$$
$$e'_e = C \texttt{ if x + 1 < a else (0 if x + 1 < a else x +1)}$$

を構成し，それぞれを単純化することで

$$e'_t = \texttt{a if x + 1 < a else } C$$
$$e'_e = C \texttt{ if x + 1 < a else x + 1}$$

を得る．そして e'_t と e'_e から，C のある分枝を削除することで，オペランド式 e_t と e_e を得る．

ステップ 3 では，max 式の直後に，ステップ 1 で得た単純化された if 文を，そのまま置けばよい．

この単純化に基づく手法は，単に実装が簡単になるだけでなく，扱える if 文の対象が少し拡張される．例えば，次の if 文を考える．

```
if x + 1 < a:
  x = a
else:
  if x + 1 > a:
    x = x + 1
  else:
    x = 0
```

この下線部は，max 演算を成す．しかし，オペランド候補を t で置換した時，

```
if x + 1 < a:
  x = t
else:
  if x + 1 > a:
    x = t
  else:
    x = 0
```

対象 if 文の then/else 節は等価ではない．一方，次のように単純化できる．

```
if x + 1 == a:
  x = 0
else:
  x = t
```

†2 https://github.com/Z3Prover/z3

したがって，単純化に基づく手法では，最終的に

```
if x + 1 == a:
  x = 0
else:
  x = max(x + 1, a)
```

が得られる．

ただし，この単純化に基づく手法は，単純化の内部実装の影響を実装レベルで受けることに注意されたい．例えば，ステップ 1 で，対象 if 文が除去されたと判定するときに，単純化で予想もしない式が手に入ると，判定不能で失敗するかもしれない．この場合，if 文の数が減っているかどうかを調べるなど，除去判定にヒューリスティクスを実装する必要が生じるかもしれない．同様に，ステップ 2 で，単純化した e'_t と e'_e の if 式の分枝に，C が直接現れないかもしれない．この場合，C の出現位置に合わせて削除方法を変えるヒューリスティクスを実装する必要が生じるかもしれない．このように，単純化に基づく手法は，等価性判定に基づく手法に対して，絶対的に優れているとは言い切れない．したがって，両方を提案手法のインスタンスとして想定すべきである．

4 拡張：一般化された max 演算子の抽出

本節では，前節で述べた通常の max 演算子の抽出法を拡張することで，より一般的な max 演算子が抽出可能であることを示す．

4.1 argmax 演算子の抽出

argmax 演算子とは，オペランドを目的関数 f で lift した定義域の上で順序比較し，大きい方を返す演算子である．f を恒等関数とすれば，argmax は max と等しくなるので，argmax は max の一般形である．例えば，次の if 文には，

```
if x > 0:
  if (x - c)*(y - c) < (a - c)*(b - c):
    x, y = a, b
  else:
    x, y = x, y
else:
  x, y = 0, 0
```

次のように argmax 演算子が内在している．

```
if x > 0:
  f = lambda w: (w[0] - c)*(w[1] - c)
  x, y = argmax(f, (x,y), (a,b))
else:
  x, y = 0, 0
```

ここでは，通常の max 演算子の抽出法を拡張することで，argmax 演算子を抽出することを考える．

まず，ステップ 1 は，そのまま利用できる．先に挙げた例のように，複数の変数に対する代入文をひとまとまりに扱う必要が生じる可能性はあるが，右辺値を選択して，変数毎に適当な変数で抽象化すれば，オペランドになりうるかどうかの判定には影響しない．同様に，ステップ 2 にて，オペランド式 e_t と e_e を構成する部分も，そののまま利用できる．異なる変数への代入が then/else 節にあった場合には，先の例のようにタプルにして 1 つの右辺式を構成すれば良い．そして，もし目的関数 f が特定できていれば，ステップ 3 にて `argmax(`f`,`e_t`,`e_e`)` を構成でき，対象 if 文の書換えも同様にできる．したがって，argmax 演算子を抽出する上での技術的要点は，ステップ 2 にて，

$$(f(e_e) < f(e_t)) = (e_1 < e_2) \tag{2}$$

を満たす目的関数 f を発見できるかどうかである．尚，右辺式のタプルで e_t と e_e を構築するとき，構築方法は複数通りあるが，先の例ではどれでも f が構築できるので，ここでは問題にならない．

目的関数 f の発見には，原則としてヒューリスティクスに頼る必要がある．我々が検証器に仮定できるのは，ある具体的な f が条件 (2) を満たすかどうかを判定可能だということだけである．仮に，存在することが保証できた場合でも，具体的な f の候補は，一般に際限なくある．その上，条件 (2) を満たすならなんでも良いというわけではない．プログラムを簡潔化や効率化に役立つ有望な f を構成できないなら，argmax 演算子を抽出する意義は乏しい．したがって，アルゴリズム（停止性が保証された手続き）の設計にはヒューリスティクスが必要になる．これは，プログラム合成 [6] の難しさとして，広く知られている．

ここで，f を構成するヒューリスティクスとして，2 つのアプローチを与える．

1 つは，条件部のオペランド e_1 と e_2 への構造的探索に基づく手法である．e_2 と e_1 は，それぞれ $f(e_t)$

と $f(e_e)$ に対応する．そこで，e_2 と e_1 の部分構造に，それぞれ e_t と e_e が存在していると仮定し，それらと等価な部分構造を変数で抽象化することで，f を構成する．この手法は，e_1 と e_2 のサイズが有限であるために，brute-force 探索を使っても停止する上に，式の構造を利用した枝刈り [1] も導入しやすい．特に，e_t や e_e が変数である場合，部分構造の等価性判定が不要になるので高速である．しかし，このアプローチは，入力の式の構造に依存した探索なので，記述の影響を受けやすく，等価な部分式を記号的に括り出す必要がある場合（例えば，`x + 2` を `(x + 1) + 1` と変換して `x + 1` を抽象化）には機能しない．

もう 1 つは，テンプレートに基づくプログラム合成 [11,12] である．例えば，f として，配列参照を行う関数 `lambda i: a[i]` や，自乗和をとる関数 `lambda xy: xy[0]**2 + xy[1]**2` が典型的によく用いられる．このような関数を抽象化したテンプレート（穴空きの関数）の集合を元に，テンプレートの具体化（穴を適当な式で埋めること）と検証を繰り返す列挙的探索で f を発見する．このアプローチは，停止性が保証されないものの，既定義のテンプレート集合に含まれる典型的な場合については，即座に発見できるという特徴を持つ．

これら 2 つは，どちらもヒューリスティクスであるため，得意な場面では上手くいくが，そうでないときには機能しない．ただし，両者は対立しているわけではなく，相補的に組み合わせることができる．例えば，構造的探索をする中で，部分式についてテンプレートに基づくプログラム合成に帰着して等価な部分式を抽出したり．テンプレートに基づくプログラム合成をする中で，e_1 や e_2 の構造を元に，テンプレートを具体化する式を選んだり．それぞれの得意な場面で上手く機能するように組み合わせるのは，実装上の論点である．

4.2 一般化 max 演算子の抽出

一般化 max 演算子（gmax と名付ける）とは，オペランドを順序関数 *ord* で比較して，大きい方を返す演算である．$ord = \lambda x, y.\, f(x) < f(y)$ と定義すれば，gmax は argmax と等しくなるので，gmax は argmax の一般形である．ただし，*ord* に任意の関数クロージャを許可すると，あらゆる if 式が gmax 演算子にエンコードできてしまい，gmax を考える意味が無い．特に，結合性がない gmax は，プログラム変換にとってあまり有益ではない．したがって，gmax 演算子に結合性をもたらすように，*ord* は，推移律 $ord(x,y) \wedge ord(y,z) \rightarrow ord(x,z)$ が成り立つものに限定する．例えば，次の if 文は，

```
if x < a:
  x, y = a, b
else:
  if x == a:
    if y < b:
      x, y = a, b
    else:
      x, y = x, y
  else:
    x, y = x, y
```

次のような 1 つの gmax 演算である．

```
ord = lambda v, w: v[0] < w[0] \
        or (v[0] == w[0] and v[1] < w[1])
x, y = gmax(ord, (x,y), (a,b))
```

ここでは，argmax 演算子の抽出法を拡張することで，gmax 演算子の抽出を考える．

max 演算子や argmax 演算子を抽出する場合と比べて，上の例が異なるのは，対象 if 文が 1 つではない点である．つまり，gmax の順序関数 `ord` が，複数の if 文の条件部にまたがって出現している．その `ord` がまたがっているらしき範囲を特定することが必要になる．これはステップ 1 を，次のように拡張することで対処できる．

ステップ 1 の要領で特定されたオペランド候補の集合を，2 つの真部分集合に分割して，その分割を与えるように適当に if 文をまとめる．先の例の場合，

```
if x < a or (x == a and y < b):
  x, y = a, b
else:
  x, y = x, y
```

と if 文をまとめる．次に，この条件部における比較演算子の等価なオペランドを適当な変数で抽象化することで，順序比較式 $e_{\sqsubset}$ を抽出する．上の例の場合，$e_{\sqsubset}$ = `v1 < w1 or (v1 == w1 and v2 < w2)` である．$e_{\sqsubset}$ を本体に持ち，`v1`，`v2`，`w1`，`w2` のペアを引数

に持つ関数の中で，推移律を持つもの $ord_{\sqsubset}$ を発見する．ここでは，$ord_{\sqsubset}$ = `lambda (v1,v2),(w1,w2): e`$_{\sqsubset}$ である．この様な，推移的な順序関数 $ord_{\sqsubset}$ を抽出できたときに，先の if 文のまとめ方が，gmax の順序関数 `ord` を**与えうる**と特定し，それを対象 if 文と見做す．

ステップ 2 では，ステップ 1 で得た $ord_{\sqsubset}$ を元に，`ord` を構成する．具体的には，対象 if 文の then 節のオペランド候補のタプル e_t，else 節のオペランド候補のタプル e_e，条件部を e_c として，

$$ord_{\sqsubset}(f_2(e_e), f_1(e_t)) = e_c \tag{3}$$

を満たすような f_1 と f_2 を発見する．これは argmax 演算子抽出における目的関数の発見と同じ問題であるので，同様の手法で抽出する．抽出できたならば，gmax の順序関数 `ord` は，

$$\texttt{ord} = \lambda v, w.\, ord_{\sqsubset}(f_1(v), f_2(w))$$

と構成できる．尚，e_t と e_e の構成法は複数想定される．具体的には，上の例の場合，4 通り想定できる．

- e_1 = `(a,b)`，e_2 = `(x,y)`
- e_1 = `(b,a)`，e_2 = `(x,y)`
- e_1 = `(a,b)`，e_2 = `(y,x)`
- e_1 = `(b,a)`，e_2 = `(y,x)`

その中で条件 (3) を満たしうるのは，最初と最後の構成法である．そして，最初の構成法を選んだ場合，

$$f_1 = \texttt{lambda x: x[0]}$$
$$f_2 = \texttt{lambda x: x}$$

となる．

ステップ 3 は，通常の max 演算子の抽出と同様に，対象 if 文を gmax 式で置換すればよい．

ここで述べた gmax 演算子の抽出法は，`ord` の部分構造である $ord_{\sqsubset}$ を，構造的探索によって構成しているものと見做すことができる．したがって，前小節で述べた構造的探索の弱みを持つことになる．しかし，構成する順序関係を，入力に含まれる順序関係からなる論理式に限定するなら，構造的探索を選択するのは合理的である．

4.3 限界

提案手法は，入力が loop-free プログラムであることに強く依存している．argmax の目的関数や gmax の順序関数が，ループを含んだ計算であるとき，提案手法をそのまま適用しても argmax や gmax は抽出できない．そのような場合は，目的関数や順序関数らしい部分を事前に見つけて，loop-free 且つ検証可能な式に翻訳してから，提案手法を適用することになる．

5 リダクションループ並列化における有用性

本節では，提案手法をリダクションループの並列化に応用したときの有用性について述べる．

まず，既存研究 [3, 10] に基づいた用語を導入する．

リダクション変数 ループ繰越依存がある（反復中に書かれた値が後の反復で読まれる）変数

記号的定数 リダクション変数の値を利用しない式

既存研究 [3, 8, 10] で，条件部が記号的定数であるような if 文は，並列化の障害にならないことが知られている．つまり，条件部にリダクション変数の値を利用する if 文こそが，リダクション並列化の障害になる．このような if 文のことを**有害な** if 文と呼ぶことにする．

リダクション変数をオペランドに含むような max 演算子を，if 文で記述したときには，それが有害なものになる．しかし，例えば次のように max 演算子で記述されていれば，

```
for a in as:
  x = max(x, a)
```

max 演算子の結合性から，効率良く並列リダクション `x = bigmax(as)` に帰着できる．ここで，`bigmax` とは，$\sum$ のような演算子であり，与えられたリストの最大要素を返す．これは，max 演算子に限らず，argmax や gmax 演算子でも同様に，それぞれに対応する `bigargmax` や `biggmax` 演算子に帰着できる．

有害な if 文への対処法としては，ループ分割とスカラ拡張の組合わせ（diffusion [7] とも呼ばれる）が用いられる場合もある．例えば，

```
for a in as:
  if x + a < 0:
    x = 0
  else:
    x = x + a
  if y < x:
    y = x
```

この 2 つの if 文は，どちらも有害である．このルー

プ分割して，スカラー変数 x を配列 xs に拡張すると，次を得る．

```
xs = [None] * (len(as) + 1)
xs[0] = x
for i, a in enumerate(as):
  x = xs[i]
  if x + a < 0:
    x = 0
  else:
    x = x + a
  xs[i+1] = x
for i, a in enumerate(as):
  if y < xs[i]:
    y = xs[i]
```

これによって，後続のループ内の if 文は有害ではなくなった．加えて，先行するループも，max 演算子を抽出できれば，max による並列スキャン（prefix sum）に帰着できる．

この diffusion 変換は，並列化を促進するものの，追加のループと中間データを導入するため，オーバーヘッドが大きい．実装レベルでは，必要最小限の適用に留める設計が望ましい．一方，よりオーバーヘッドが小さい並列化が適用可能な場合もある．例えば，上の例では，

```
for a in as:
  x = max(0, a + x)
  y = max(x, y)
```

と max 演算子が抽出できる．このループ本体は，max-plus 半環の行列ベクトル積に相当するので，行列積の結合性から，1 つの並列リダクションに帰着できる [8,10]．

この様な max-plus 半環の行列乗算に基づく並列化を促進するために，if 文から max 演算子を自動抽出する手法 [10] が開発されている．これは，先の例を，提案手法と同様に正規化し，

```
for a in as:
  if x + a < 0:
    if y < x:
      x, y = 0, x
    else:
      x, y = 0, y
  else:
    x = x + a
    if y < x:
      x, y = x + a, x + a
    else:
      x, y = x + a, y
```

この代入文の右辺式を利用して

```
for a in as:
  x, y = bigmax([0, 0, x + a, x + a]), \
         bigmax([x, y, x + y, y])
```

を検証して抽出する．この手法は，技術的に言えば，2 項演算子の max を抽出するのではなく，有害な if 文全体を bigmax 演算子に帰着するものである．

この bigmax 抽出法は，有害な if 文全体が bigmax 演算子であると仮定するため，部分的に max 演算子が含まれている有害な if 文には対応できない．したがって，bigmax 抽出を行う前に，max 演算子に帰着できない有害な if 文を，diffusion 変換によって潰しておく必要がある．これは，diffusion 変換を最小限に留めようとする設計と相反するので，この bigmax 抽出法と diffusion 変換の相性は悪い．

一方，提案手法は，細粒度に可能な限り max 演算子を事前に抽出することが可能なので，その後に並列化に必要な部分だけを，依存グラフを用いて diffusion 変換で潰すことが容易にできる．つまり，提案手法は diffusion 変換と相性が良い．この意味で，提案手法はモジュラーであり，並列化を含めたコンパイラ最適化との親和性が高い．

6 関連研究

リダクション並列化のための bigmax 演算子抽出 [10] は，本研究と最も技術的に関係する仕事である．前節で述べたように，提案手法の方がモジュラーで，他のプログラム変換と組合わせやすいという利点がある．もう 1 つの重要な点は，bigmax 演算子抽出 [10] は，argmax や gmax 演算子を抽出するような拡張が設計上困難であることである．これは，[10] の手法が，与えられた if 文の条件部を，単に実行パスの事前条件と見做してオペランド候補の順序を検証するためである．したがって，提案手法は，[10] の手法を，多面的に一般化した手法である．

効率面としては，[10] の手法は，通常の max 演算子に特化している分，手法が単純で，検証器への問合せ回数も抑えられている．一方，提案手法は，最悪時の検証器への問合せ回数は大きくなるものの，ステッ

プ 1 でオペランド候補を特定できれば，大幅に効率化（検証器への問合せ回数が削減）される．したがって，提案手法は，実装レベルのヒューリスティクスで高速化する余地が大いにあり，効率面でも [10] の手法に劣後するとは限らない．

Fisher と Ghuloum [3] は，関数合成に基づくリダクションループ並列化を提案した．これは，ループ本体を 1 つの関数として定式化し，その合成関数を，合成の下で閉じる形に変形することで並列化する手法である．彼らの手法では，合成関数内の if 文を，Fourier–Motzkin 消去法を用いて単純化する．これは，閉じた形を得る上で重要な変換である．一方，我々の提案手法は，（ステップ 1 で実質的に）if 文の単純化を用いることで，max 演算子を抽出している．この意味では，Fisher と Ghuloum の手法が有害な if 文を含んだループを並列化ときには，max 演算子相当の計算を抽出していると見ることができる．そして，提案手法は，その max 演算子抽出に対応する部分を，リダクション並列化から切り離して整理したものであると言える．

Morihata と Matzuaki [9] は，Fisher と Ghuloum の並列化手法 [3] を，限定記号除去を使って定式化した．彼らは，限定記号除去アルゴリズムに広く使われる除去集合を用いることで，並列リダクションの結合的 2 項演算子が，具体的に構成できることを示した．彼らの手法も，Fisher と Ghuloum [3] と同様に，max 演算子相当の計算を結果的に抽出している．一方，我々の提案手法は，max 演算子のオペランドを見積もって，それを除去集合のようなものと見做して，具体的なプログラムを構成している．提案手法は，限定記号除去という難しい一般的問題を避けて，if 文からの max 演算子の抽出に問題設定を限定しているため，既存の検証器を用いて，軽量且つモジュラーに実装できる．

Farzan と Nicolet [2] は，Fisher と Ghuloum の並列化手法 [3] を，探索に基づくプログラム合成のアプローチで実装した．我々の提案手法も，argmax や gmax 演算子を構成する時には，探索に基づくプログラム合成を用いている点で類似している．一方，彼らは，並列リダクションの結合的 2 項演算子を，入力プログラムから直接合成している．この意味で，彼らはより難しい問題をプログラム合成に帰着させており，本質的により広い探索空間の中でプログラム候補を発見する必要がある．したがって，より限定的な範囲の探索で済ますことができる我々の提案手法の方が，入力プログラムのサイズに対してより頑健である．そして，通常の max 演算子の抽出に関して言えば，より効率が良く，停止性も保証できている．

我々の提案手法は，入力を loop-free プログラムに限定している．プログラム合成では，入力を loop-free プログラムに還元して扱うことは，簡潔だが非自明なプログラムを合成するときの常套手段 [5] である．

7 まとめと今後の課題

本論文では，検証器を利用して if 文から max 演算子を抽出する方法を提案した．そして，順序比較が一般化された max 演算子を抽出する拡張も提案した．提案手法は，既存の max 演算子の抽出法 [10] と比べて，細粒度に max 演算子を抽出するため，他のプログラム変換とモジュラーに組み合わせ易い．類似するプログラム合成ないしプログラム導出の手法 [2,3,9] と比べて，問題設定を限定しているので，軽量に実装可能である．

現段階で，提案手法は未実装である．Z3 ソルバを使って，抽出するための条件が検証可能であることは手作業で確認している．リダクションループの並列化への応用を含めて提案手法を実装する予定である．

謝辞 本研究は JSPS 科研費 18K18032 の助成を受けたものである．

参 考 文 献

[1] Alpern, B., Wegman, M. N., and Zadeck, F. K.: Detecting Equality of Variables in Programs, *Proc. the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, ACM, 1988, pp. 1–11.

[2] Farzan, A. and Nicolet, V.: Synthesis of Divide and Conquer Parallelism for Loops, *Proc. the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, ACM, 2017.

[3] Fisher, A. L. and Ghuloum, A. M.: Parallelizing Complex Scans and Reductions, *Proc. the ACM*

SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94, ACM, 1994, pp. 135–146.

[4] Fog, A.: Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, `https://www.agner.org/optimize/instruction_tables.pdf`, September 2018.

[5] Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R.: Synthesis of Loop-Free Programs, *Proc. the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementationw*, PLDL '11, ACM, 2011, pp. 62–73.

[6] Gulwani, S., Polozov, O., and Singh, R.: Program Synthesis, *Foundations and Trends in Programming Languages*, Vol. 4, No. 1–2(2017), pp. 1–119.

[7] Hu, Z., Takeichi, M., and Iwasaki, H.: Diffusion: Calculating Efficient Parallel Programs, *Proc. the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '99, BRICS, 1999, pp. 85–94.

[8] Matsuzaki, K., Hu, Z., and Takeichi, M.: Towards Automatic Parallelization of Tree Reductions in Dynamic Programming, *Proc. the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, ACM, 2006, pp. 39–48.

[9] Morihata, A. and Matsuzaki, K.: Automatic Parallelization of Recursive Functions using Quantifier Elimination, *Functional and Logic Programming (Proc. FLOPS '10)*, Lecture Notes in Computer Science, Vol. 6009, Springer, 2010, pp. 321–336.

[10] Sato, S. and Iwasaki, H.: Automatic Parallelization via Matrix Multiplication, *Proc. the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementationw*, PLDL '11, ACM, 2011, pp. 470–479.

[11] Srivastava, S., Gulwani, S., and Foster, J. S.: From Program Verification to Program Synthesis, *Proc. the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, ACM, 2010, pp. 313–326.

[12] Srivastava, S., Gulwani, S., and Foster, J. S.: Template-based program verification and program synthesis, *Int. J. Softw. Tools Technol. Transfer*, Vol. 15, No. 5–6(2013), pp. 497–518.

[13] Xu, D. N., Khoo, S.-C., and Hu, Z.: PType System: A Featherweight Parallelizability Detector, *Programming Languages and Systems (Proc. APLAS '04)*, Lecture Notes in Computer Science, Vol. 3302, Springer, 2004, pp. 197–212.