

プログラム変換における変換履歴再利用の支援

3L-4

齋藤 博 西谷 泰昭

群馬大学 工学部 情報工学科

1 概要

ソフトウェアの生産性、信頼性を向上させるために、ソフトウェアの再利用技術が研究されている。プログラムの再利用の他に、プログラムの作成過程を再利用することが提案されている[1]。これは、プログラムの作成過程に終端再帰や二分法のようなプログラミング手法が含まれていると考え、そのようなプログラミング手法を形式的に抽出、一般化して保存し、それを同様なプログラムを作るときに再利用するものである。

作成過程の再利用には、個々の問題に固有な知識が必要となり、その知識をどのように与えるかが問題点として指摘されている。本発表では、その解決法について述べる。

2 プログラム変換と変換履歴

プログラムの作成過程として、仕様からプログラムへ、ルールの適用を繰り返しながら変換を行なうプログラム変換を考える。そして、その変換の過程を記録したものを変換履歴といい、それを一般化したものを変換履歴スキーマという。変換履歴スキーマは、プログラミング手法を表現していると考えられる。鈴木[1]は、2つの変換履歴を二階のパターンマッチングにより比較し、異なっている関数や定数をスキーマ変数に置き換えることにより、変換履歴を自動的に一般化する手法を与えている。

[例 1] 変換履歴と変換履歴スキーマの例

```
(multi A B) ≡ (fold + 0 (map (λ.x A) (mklist 1 B))) 仕様
||| 略
(if (= B 1) A
  (if (oddp B) (+ A (multi A (1- B)))
    (fold + 0
      (append
        (map (λ.x A) (mklist 1 (mid 1 B)))
        (map (λ.x A) (mklist (1+ (mid 1 B)) B))))
      ルール：< (fold + 0 (append l1 l2)) →
        (+ (fold + 0 l1) (fold + 0 l2)) >を適用
    ||| 略
    (+ (multi A (mid 1 B)) (multi A (mid 1 B)))
    ルール：< (+ a a) → (double a) >を適用
    (double (multi A (mid 1 B)))
  ))
```

これは、乗算を、'+'を用いて対数時間で計算するプログラム multi を作成したときの変換履歴である。また、この変換履歴中の関数'multi'や'+', 'double', 定数'0'をスキーマ変数 (SV1 ~ SV4) に置き換えて、変換履歴を一般化した変換履歴スキーマを次に示す。

```
(SV1 A B) ≡ (fold SV2 SV3 (map (λ.x A) (mklist 1 B))) 仕様
||| 略
(if (= B 1) A
  (if (oddp B) (SV2 A (SV1 A (1- B)))
    (fold SV2 SV3
      (append
        (map (λ.x A) (mklist 1 (mid 1 B)))
        (map (λ.x A) (mklist (1+ (mid 1 B)) B))))
      ルール：< (fold SV2 SV3 (append l1 l2)) → ... (1)
        (SV2 (fold SV2 SV3 l1) (fold SV2 SV3 l2)) >を適用
      ||| 略
      (SV2 (SV1 A (mid 1 B)) (SV1 A (mid 1 B)))
      ルール：< (SV2 a a) → (SV4 a) >を適用 ..... (2)
      (SV4 (SV1 A (mid 1 B)))
    ))
```

次の変換履歴スキーマは、二分探索のプログラミング手法を表している。

```
(SV1 A ARR M N)
≡
(fold (λ.xy (if (SV2 A (SV3 ARR x)) x y)) nil (mklist M N))
||| 略 ↑仕様
(if (> M N)
  nil
  (if (SV2 A (SV3 ARR (mid M N)))
    (mid M N)
    (if (SV4 A (SV3 ARR (mid M N)))
      (fold (λ.xy (if (SV2 A (SV3 ARR x)) x y))
        (fold (λ.xy...) nil (mklist (1+ (mid M N)) N))
        (mklist M (1- (mid M N)))) ..... (3)
      ルール：< (fold (λ.xy (if (p x) x y))
        init (mklist m n)) → init ... (4)
        適用条件：∀k m ≤ k ≤ n (p k) = nil >を適用
      (fold (λ.xy...) nil (mklist (1+ (mid M N)) N))
      |仕様でたみ込み
      (SV1 A ARR (1+ (mid M N)) N)
    (SV1 A ARR M (1- (mid M N)))
  ))
```

3 変換履歴スキーマの再利用

仕様が与えられた時に、その仕様と似ている仕様を持つ変換履歴スキーマを再利用することができる。まず、両者の仕様を二階のパターンマッチングにより比

較し、それらが等しくなるようなスキーマ変数への代入を求める。次に、その代入を変換履歴スキーマ全体に行なうことによって、与えられた仕様をプログラム変換したときの変換履歴を得ることができる。しかし、ここで問題が生じる。その問題点と解決法を以下に示す。

問題点

1. スキーマ変数を含んでいるルールの扱い。

例1の(1)のルールのようにスキーマ変数を含んだルールは、具体化時に正しいルールとなる保証はないので、正当性の証明が必要である。

2. すべてのルールの適用条件を満たしているか。

ルールのパターン変数に対応する式がスキーマ変数を含んでいる場合、具体化時にルールの適用条件を満たすとは限らないので、証明が必要である。

3. 仕様に現れないスキーマ変数の具体化。

例1のSV4のように、仕様に現れないスキーマ変数への代入は、自動的に決定できない。

解決法

1. ルール自身の証明をプログラム変換で行ない、その変換履歴をルールに持たせ、変換履歴スキーマ作成時に、これも一般化する。
2. プログラム変換時に、ルールの適用条件を満たすことの証明もプログラム変換で行ない、変換履歴の中に、その証明を変換履歴として記録する。そして、変換履歴スキーマ作成時に、この証明も同時に一般化する。

これにより、問題点は次のように改善される。

問題点1について：スキーマ変数を含んだルールは、具体化時にプログラム変換によって正当性の証明を行なう。しかし、解決法の1より、そのルールが、それ自身の一般化された証明として変換履歴スキーマを持つので、それを正しく具体化すればより簡単に証明を得られる。

例えば、例1の(1)のルールは、具体化されたときの証明が非常に複雑であるが、一般化された証明として変換履歴スキーマを持つので、それを正しく具体化すれば、証明を変換履歴として得ることができる。しかし、このときも次のようなスキーマ変数を含むルールが存在する。

$\langle (SV2\ SV3\ a) \rightarrow a \rangle$
 $\langle (SV2\ (SV2\ a\ b)\ c) \rightarrow (SV2\ a\ (SV2\ b\ c)) \rangle$

これは、(1)のルールを証明するときに、上記の2つのルール、すなわち、SV3がSV2の単位元であることと、SV2が結合律を満たすことを証明すればよいことを示している。

問題点2について：適用条件を満たすことの証明も同様に、プログラム変換によって行なう。このとき、解決法の2より、変換履歴スキーマの中に適用条件の一般化された証明が変換履歴スキーマとして記録されているので、それを具体化することによって、より簡単に証明できる。

例1の(3)の式で、(4)のルールのパターン変数pに対応する部分は、 $(\lambda x.(SV2\ A\ (SV3\ ARR\ x)))$ であり、スキーマ変数を含むので、変換履歴スキーマの具体化時に適用条件を満たすことを証明しなければならない。一般化された証明である変換履歴スキーマを正しく具体化すれば適用条件の証明が得られる。そのとき、スキーマ変数を含むルールが現れ、それらは次のことを意味している。したがって、具体化時にこれらのルールの正当性さえ証明すればよい。

$(SV5\ (SV3\ ARR\ i)\ (SV3\ ARR\ j)) = True$
 $(SV4\ a\ b) \Rightarrow (SV5\ b\ a)$
 $(SV4\ a\ b) \wedge (SV4\ b\ c) \Rightarrow (SV4\ a\ c)$
 $(SV4\ a\ b) \Rightarrow (not\ (SV2\ a\ b))$ } (*)

問題点3について：仕様に現れないスキーマ変数への代入は人間が与えなければならない。しかし、そのスキーマ変数を含むルールを集めることによって、代入を決定するためのヒントを得ることができる。証明を保持することにより、より多くのヒントを得られるようになった。

例1の2つの変換履歴スキーマに含まれるSV4は、いずれも仕様に現れないスキーマ変数である。対数時間での計算を表す変換履歴スキーマでは、(2)のルールが代入を決めるためのヒントとなり、もし、SV2が'*'に具体化されたならば、SV4は自乗(square)に具体化するのが適当であることが予想できる。また、二分探索の変換履歴スキーマでも、(*)で示したように、SV4を含むルールが存在し、SV2が'='に、SV5が'<'に具体化されたならば、SV4は'>'に具体化すればよいことが予想できる。

4 まとめ

変換履歴スキーマを具体化するとき、問題点として述べたような個々の問題に固有な知識が必要となる。そこで、ルール自身や、ルールの適用条件の証明を変換履歴スキーマに記録することにより、そのような知識をより簡単に与えることができることについて述べた。

参考文献

- [1] 鈴木秀明: “プログラム変換履歴の一般化”, 情報ソフトウェア工学研究会, SE-90, pp. 81-88, 1993.
- [2] G. Heute and B. Lang: “Proving and applying program transformations expressed with second-order patterns”, *Acta Informatica*, 11, pp. 31-55, 1978.