プログラミング 12−2
（1997. 3. 28）

tactic からの プログラム抽出

戸田 洋三
千葉大学 総合情報処理センター
〒263 千葉市稲毛区弥生町 1–33
E-mail: yozo@ipc.chiba-u.ac.jp

萩谷 昌己
東京大学 大学院 理学系研究科
〒113 文京区本郷 7–3–1
E-mail: hagiya@is.s.u-tokyo.ac.jp

あらまし
定理証明系の証明記述言語である tactic からのプログラム抽出における問題について議論する．tactic は証明の構成手順を記述するための制御構造を持っており，プログラムの制御構造にあてはめて考えることができる．本稿ではいくつかの事例をとおして tactic からプログラムを抽出するために次の 2 点が重要であることを指摘する．

- tactic が扱う goal のパターンを求めること
- goal のパターンを論理体系のなかで表現すること

キーワード　　定理証明, タクティク, プログラム抽出

Program Extraction from Tactics

Yozo TODA
Information Processing Center, Chiba Unviersity
1-33 Yayoicho, Inage-ku, Chiba 263 JAPAN
E-mail: yozo@ipc.chiba-u.ac.jp

Masami HAGIYA
Graduate School of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113 JAPAN
E-mail: hagiya@is.s.u-tokyo.ac.jp

ABSTRACT
We argue about problems on program extraction from tactics; tactics are used in many theorem proving systems to describe how to construct proofs. Each control structure of tactics can be easily mapped to that of programs. Examining a few cases, we claim that the following are important points.

- obtaining goal patterns which tactics accept and generate
- representing goal patterns inside the logical systems

KEYWORD　　theorem proving, tactics, program extraction

1 tactic からのプログラム抽出

1.1 プログラム抽出と 定理証明系

構成的な証明からプログラムを合成する手法として種々の研究が行なわれている [1,2,5]. この手法の魅力は, プログラムを書くと同時にプログラムがなぜ正しいものであるのかということの説明までをも含めることによって, (プログラムが動き出すまでの労力は増えるが) 思い違い・訂正ミスなどといった人為的なエラーを減らすことができる, というところにある. このような目的のためにはプログラムの仕様を命題として表現し, その証明をつくるという作業が中心となる. 証明を作成する環境はこの活動を援助するために重要な要素となる.

また, ハードウェア設計・ソフトウェア検証・数学の形式化などといった分野においても同様に, 証明作成を支援する環境は重要であり, すでにさまざまなシステムが開発・利用されている [1,2,4,5,7,8]. 計算機を用いて定理を証明するこれらのシステムでは, 高階論理あるいは型付き λ 計算にもとづいて命題や証明を表現する.

現在広く利用されている定理証明系 (証明支援系) としては COQ[2], HOL[3] などがある. COQ は型つき λ 計算の体系の一種である Calculus of (Inductive) Constructions を利用したシステムであり, 証明 (λ 項) から ML プログラムを抽出する機能が用意されている [6]. これに対して, HOL は Church の simple type theory をもとにした高階論理を利用するシステムであり, おもにハードウェア設計検証の分野で広く使われている.

1.2 定理証明系と tactics

定理証明系において証明を記述する手段としては, 決められた形の自然言語で直接証明を書くもの, 型付き λ 項を書くもの, tactic と呼ばれる一種のプログラミング言語により証明の構築手順を記述するものなどがある.

HOL では高階論理の証明を, COQ では型つき λ 項を構成するが, どちらも tactic と呼ばれるプログラミング言語の一種を用いてその構成手順を表現する. このうち HOL では, tactic によって記述された証明は内部で検証され, 証明自体は利用者には見えない. このようなシステムを使ってプログラム抽出を行なうには, tactic を直接扱う手法が有効であると考えられる.

以下, tactic による証明について簡単に紹介したあと, tactic からプログラムを合成するアプローチについて述べる. GCD の計算と co-inductive な述語を例として取り上げる.

2 tactic による 証明記述

"tactic" という概念は定理証明系 LCF においてはじめて導入された. Nuprl, HOL, Coq など現在あるシステムの多くは LCF の影響を受けており共通点も多い. なかでも HOL は LCF の直系の子孫であり, もっとも広く使われている.

HOL を使って対話的に証明をつくっていく際には, まず証明すべき命題を "goal" として設定する. goal は構築すべき証明図の根に相当する部分である. この goal に対して tactic を適用し, 証明図を「下から」構成していく. これが tactic の基本的な使い方である. (実際には「上から」証明を構成する機能もあり, 両方をうまく使って証明をつくっていくことになる.) 例えば,

```
⊢? A ==> (B ==> (A /\ B))
```

を goal とする[1]. この命題の最後の推論規則は implication 導入であると考えて, 対応する tactic(DISCH_TAC) を適用する. これによって A は前提条件となり残りの部分が新しく goal となる. 同様にもう一度 DISCH_TAC を 適用して残る goal は

```
A,B ⊢? A /\ B
```

となる. 次に $\wedge$ の導入規則に対応する tactic (CONJ_TAC) を適用すると A および B に対応する 2 つの subgoal がつくられる.

```
A,B ⊢? A          A,B ⊢? B
```

最後に前提の 1 番目, 2 番目が goal 自身であるとして (ACCEPT_TAC) 証明が完了する. (goal が空になる.)

このような基本的な tactic を 組み合わせてより複雑な tactic をつくるために "tactical" が用意されている. 例えば, ORELSE という tactical がある. tactic では goal が一定の形をしているものと想定しているので適用が失敗することがあるが, tactical ORELSE によってつくられた (T_1 ORELSE T_2) という tactic では, まず T_1 を goal に適用してみて成功すればそのまま次に進み, 失敗したらそこでもとに戻って T_2 の適用を試みる. どちらも失敗した場合には, この tactic 全体が失敗したことになる.

このように選択的実行 (ORELSE) を行なう tactical の他に, tactic を 順番に実行したり (THEN), 失敗するか goal がすべて証明されるまで繰り返したり (REPEAT) といった動的な仕掛けが用意されている (利用者が新しい tactic や tactical をつくることもできる). tactic による証明作成の利点はこのように tactical によって動的な実行制御を行なえるところにある.

[1] ここでは $\Gamma \vdash A$ で定理を, $\Gamma \vdash_? A$ で証明すべき goal を表すこととする. また, HOL における論理記号の表記, /\, \/, ==>, ~, !, ? などをそのまま 使うことにする. ! は $\forall$, ? は $\exists$, ~ は否定, (\x.A) は λ 項を表している.

先の例を一つの tactic にまとめてみると以下のようになる:

```
REPEAT DISCH_TAC THEN
CONJ_TAC THENL
[(POP_ASSUM_LIST
    (fn asl => ACCEPT_TAC (el 2 asl))),
 (POP_ASSUM_LIST
    (fn asl => ACCEPT_TAC (el 1 asl)))]
```

3 tactic からのプログラム抽出

tactic は goal に応じて動的な制御をしつつ証明を作成する「プログラム」であり, とくに tactical が制御構造を表していると見ることができる. そこで, tactic の構造をそのままプログラムに対応づけて考える. ここで重要なことは, goal に適用された tactic の実行が成功 (あるいは失敗) する条件がプログラムでの条件分岐や 繰り返しに現れることである.

具体例としてユークリッドの互除法による最大公約数の計算を考える. 述語 GCD を以下のように定義する[2].

```
(GCD x y z) =
  ((DIVIDES z x) /\ (DIVIDES z y) /\
   (!w.((DIVIDES w x) /\ (DIVIDES w y))
        ==> (DIVIDES w z)))
```

gcd を計算するアルゴリズムの仕様は gcd の存在を表す命題として表現される.

⊢? ?z.(GCD x y z)

この命題に対し, ユークリッドの互除法にしたがって以下の 3 つの lemma を用いた証明を構成する.

(A) ⊢ ?z.(GCD 0 y z)

(B) ⊢ (x<=y) /\ (GCD x (y-x) z)
==> (GCD x y z)

(C) ?z.(GCD x y z) ⊢ ?z.(GCD y x z)

3 つの lemma を証明する tactic をそれぞれ A_TAC, B_TAC, C_TAC とすると[3], 存在定理を証明する tactic は 以下のように書くことができる.

```
REPEAT
  (A_TAC ORELSE B_TAC ORELSE C_TAC)
```

この tactic ではまず A_TAC を 試す. A_TAC が失敗するとかわりに B_TAC, これも失敗すると C_TAC と順に試す (ORELSE). そしてこの動作を繰り返す (REPEAT). ⊢? ?z.(GCD 10 4 z) という goal に対し tactic は以下のようにして "証明" を行なう.

(C_TAC の適用): ⊢? ?z.(GCD 4 10 z)
(B_TAC の適用): ⊢? ?z.(GCD 4 6 z)
(B_TAC の適用): ⊢? ?z.(GCD 4 2 z)
(C_TAC の適用): ⊢? ?z.(GCD 2 4 z)
(B_TAC の適用): ⊢? ?z.(GCD 2 2 z)
(B_TAC の適用): ⊢? ?z.(GCD 2 0 z)
(C_TAC の適用): ⊢? ?z.(GCD 0 2 z)
(A_TAC の適用): (z=2) goal proved!

この tactic 全体は 以下のようなプログラムを実行していることになる:

```
gcd(x,y) =
  if (x=0) then y
    else if (x<=y)
      then gcd(x,(y-x))
      else gcd(y,x)
```

このプログラム gcd は, 上の tactic と同様の再帰構造を有している. REPEAT の本体である

(A_TAC ORELSE B_TAC ORELSE C_TAC)

という tactic は ⊢? ?z.(GCD x y z) という形の goal に対して適用され, 同じ形の subgoal を生成する. すなわち REPEAT の実行中に現れる goal は常に上の goal のパターンを具体化したものになっており, gcd はこのパターンの変数部分を引数とする関数になっている. gcd(x,y) という関数呼び出しは ⊢? ?z.(GCD x y z) という形の goal に対して上の tactic が実行されたことに対応する. また, 関数 gcd の本体の最初の条件 (x=0) は, tactic A_TAC が成功する条件, 次の条件 (x<=y) は tactic B_TAC が成功する条件に対応している.

⊢? ?z.(GCD x y z) という goal に対して B_TAC は ⊢? ?z.(GCD x (y-x) z) という subgoal を生成する. これに対応して, プログラムでは再帰呼び出し gcd(x,(y-x)) を行なう. 同様に tactic C_TAC が生成する subgoal に対応して再帰呼び出し gcd(y,x) を行なう. それぞれ, 関数の引数は subgoal の形に応じて決まる.

関数 gcd の返り値は, goal の中の存在変数 z に具体化される値に対応する. すなわち, 上の tactic は最終的に⊢ ?z.(GCD x y z) という形の定理を証明するが, これによって得られる証明は ∃ の導入規則である EXISTS によって始まるものである. HOL では証明が失われてしまうが, COQ のように証明がデータ構造として構築される場合は, 存在変数に具体化される値を証明から抽出することができる. また, Isabelle[8] や論理型の定理証明系 (例えば λPROLOG[3]) のようにいわゆる論理変数を有する定理証明系では⊢? ?z.(GCD x y z) という goal

[2]述語 DIVIDES はすでに定義されているものとする

[3]より正確にいうと B_TAC は⊢?(GCD x y z) を⊢?(GCD x (y-x) z) に, C_TAC は⊢??z.(GCD x y z) を⊢??z.(GCD y x z) に '書き換える' tactic としてつくってある.

の代わりに⊢? (GCD x y Z) という goal を tactic によって証明する. Z はいわゆる論理変数であり, tactic の実行中に適当な項によって具体化される. 論理変数に対する具体化は tactic の出力と考えられ, そのような tactic から抽出される関数は, 論理変数に具体化される値をその返り値とする.

tactic の出力としては, 存在変数もしくは論理変数への具体化の他に, 次節の例で述べるように, disjunctive な goal に対してどちらの disjunct が成り立つか (右か左か) に関するものがある. この場合, tactic から抽出される関数は bool 値を返すことになる.

上の関数 gcd(x,y) は, どのような自然数の組 (x,y) に対しても停止する. これは, 自然数の組の辞書式順序に対して, 引数 (x,y) が再帰呼び出しの過程で常に減少するからである. tactic から抽出された関数の停止性を示すことは, tactic が対象とする goal が常に定理として成り立っていることを意味する. gcd の例の場合, 関数 gcd の停止性が成り立っているならば, x と y をどのような自然数によって具体化しても, ⊢∃z.(GCD x y z) という定理が成り立つ.

さらに, 定理証明系が対象としている論理体系の中で, 関数の停止性を示すことができるならば, goal のパターンの中の変数を論理体系の中で全称化することができるはずである. gcd の例の場合は

⊢ ∀x.∀y.∃ z.(GCD x y z)

という定理が成り立つ. このアプローチでは, 停止性の議論に対して別の独立した形式体系を必要としないという利点がある. (これは証明からのプログラム抽出の手法と等価なものになっている.)

以上の考察より, tactic からプログラムを抽出するためには次の二点が重要な課題であると我々は考えた.

1. tactic が対象とする goal のパターンを適切に求める. tactic が繰り返し (REPEAT) などの制御構造を含む場合は, tactic の実行過程で現れる goal のパターンも求める.

2. 得られた goal のパターンを, 定理証明系の論理体系の中で表現する.

最終的には, 以上の二点を自動的に行う手法を求めたいと考えている.

本研究では, 上の課題を明確なものとするためにより non-trivial な事例について考察した. これについて次節で報告する.

4 co-induction の事例

本節では, tactic からのプログラム抽出の non-trivial な事例として, co-inductive に定義された述語の扱いを考えてみる. co-induction はデータ型や述語を最大不動点によって定義する手法で, その用途は広い. 例えば, 関数プログラミングなどで利用される無限のストリームは, co-inductive なデータ型として定義される. また, 並行計算の理論で用いられる bisimulation は, co-inductive に定義される述語と考えられる.

いま, 自然数上の述語 Q が次のように co-inductive に定義されているとする.

$$(q1)\ \frac{(Q\ 0)\quad (Q\ 1)}{(Q\ 0)} \qquad (q2)\ \frac{(Q\ 1)}{(Q\ 1)}$$

$$(q3)\ \frac{}{(Q\ 2)} \qquad (q4)\ \frac{(Q\ 1)\quad (Q\ 3)}{(Q\ 4)}$$

以下では, 定数 a が与えられているとき (Q a) であるか, あるいは ¬(Q a) であるかを 判定する tactic を定義する.

具体例として (Q 0) を示すことを考えよう. 規則 (q1) より,

```
(Q 0) /\ (Q 1) ==> (Q 0)
```

が成り立つ. また, 規則 (q2) より,

```
(Q 1) ==> (Q 1)
```

が成り立つ. 従って (Q 0) と (Q 1) が成り立っていると仮定すると, (Q 0) と (Q 1) 自身がその帰結として得られる. すなわち, (Q 0) と (Q 1) という仮定は上の co-inductive な定義と矛盾していない. 従って (Q 0) と (Q 1) は成り立つ. これが co-induction の原理であるが, 形式的には以下のようになる.

自然数上の述語 P に対して,

```
\x. ((x=0) /\ (P 0) /\ (P 1)) \/
    ((x=1) /\ (P 1)) \/
    ((x=2) /\ T) \/
    ((x=4) /\ (P 1) /\ (P 3)) \/ F
```

という自然数の述語を (TQ P) と書く. 述語 P が

```
!x. (P x) ==> (TQ P x)
```

を満たしているならば,

```
!x. (P x) ==> (Q x)
```

が導かれる (co-induction).

上の例の場合,

```
P = \x.(x=0) \/ (x=1)
```

とおくと, P は !x.(P x)==>(TQ P x) を満たすので, !x.(P x)==>(Q x) が導かれる. 明らかに P(0) は成り立つので Q(0) が得られる.

一般に, 定数 a に対して (Q a) を示したいときには,

```
P = \x.(x=a) \/ ...
```

のようにPを定義しておき, `!x.(P x)==>(TQ P x)` が成り立つように ... の部分を具体化してやればよい. 上の例のように, ... の部分は`(x=`a_i`)` という等式の disjunction になる.

`(Q 0)` を示す場合, `(Q 0)` を結論に持つ規則 (q1) の前提部分である `(Q 1)` によって`(x=1)` という disjunct が新たに加わる. この disjunct により `(Q 1)` を結論に持つ規則を探すが, これは規則 (q2) である. ところが規則 (q2) の前提からは新たな disjunct は得られないので, P の定義が完成する.

逆に `~(Q` a`)` を示すにはどうすればよいだろうか. 例えば `~(Q 4)` を示す場合, `(Q 4)` を結論に持つ規則は (q4) しかないので,

```
(Q 4) ==> (Q 1) /\ (Q 3)
```

が成り立つ. `(Q 1)` は規則 (q2) により成り立つが, `(Q 3)` を結論に持つ規則がないので, `(Q 3)` は成り立たない. すなわち, `~(Q 3)` である. 従って `~(Q 4)` も成り立つ.

ここで仮に `(Q 4)` を示そうとしたとき, 述語 P はどのように定義されるだろうか. `(Q 0)` の場合と同様に考えると,

```
P = \x.(x=4) \/ (x=1) \/ (x=3)
```

という定義が得られる. この定義の求め方から,

```
(Q 4) ==> !x. (P x) ==> (Q x)
```

が成り立っており, これと `~(Q 3)` より`~(Q 4)` を導くことができる.

以上の議論により,

```
⊢? !x.((P x)==>(TQ P x)) \/ ~(Q a)
 where  P = \x.(x=a)\/(P' x)
```

という形の goal を, P' の部分を具体化しながら証明する tactic を求めることが可能ではないかと考えられる. まず, 上の goal を

```
(Q a),((x=a)\/(P' x)) ⊢? (TQ P x)
  where P = \x.(x=a)\/(P' x)
```

と書き換えると, 次のような tactic T_0, T_1, T_2 を書くことができる. どの tactic も,

```
Q(b1),Q(b2)...,Q(bn),
((x=b1)\/(x=b2)\/...\/(x=bn)\/(P' x))
   ⊢? (TQ P x)
 where P =
   \x.((x=a1)\/...\/(x=am)\/
       (x=b1)\/(x=b2)\/...\/(x=bn)\/
       (P' x))
```

という形の goal[4]に対して適用されるもので, 次のように REPEAT と ORELSE によって組み合わされて実行される.

REPEAT (T_0 ORELSE T_1 ORELSE T_2)

なお, 本研究の最終目標の一つは, このような goal のパターンを tactic のコードから正確に求めることにある.

(tactic T_0)

$n = 0$ のときに呼び出される. `P'=\x.F` という具体化を行って証明を終了する. この場合 `(Q` a`)` が成り立っていることになる.

(tactic T_1)

`(TQ P` b_1`)` の各 disjunct を調べる. どの disjunct においても `(x=...)` の部分が成り立たなければ, `~(Q` b_1`)` と最初の仮定 `(Q` b_1`)` とを矛盾させて証明を終了する. この場合 `~(Q` a`)` が成り立っていることになる.

(tactic T_2)

まず `(TQ P` b_1`)` の各 disjunct に対して`(x=...)` の部分が成り立つかどうかを調べる. 成り立つ disjunct が存在すればそれを

`(x=`b_1`)/\P(`d_1`)/\.../\P(`d_l`)`

としたとき, `P(`d_j`)` が即座に証明できない, すなわち d_j が$[a_1,\ldots,a_m,b_1,\ldots,b_n]$ に含まれないものを $[c_1,\ldots,c_k]$ とする. そして,

```
P' = \x.(x=c1)\/.../\(x=ck)\/(P'' x)
```

という代入を行って次のような subgoal を返す.

```
Q(b2),...,Q(bn),Q(c1),...,Q(ck),
((x=b2)\/.../\(x=bn)\/
 (x=c1)\/.../\(x=ck)\/(P'' x))
      ⊢? (TQ P x)
  where  P =
  \x.((x=a1)\/.../\(x=am)\/
      (x=b2)\/.../\(x=bn)\/
      (x=c1)\/.../\(x=ck)\/
      (P'' x))
```

以上の議論では

```
(Q b1) ==> (Q d1)/\.../\(Q dl)
```

が成り立つことを仮定している.

前節で述べたように,

1. tactic が対象とする goal のパターンを適切に求めること
2. 定理証明系の論理体系の中で表現すること

が我々の最終目標であるが, 実際の場合は容易なことではない. 上の場合,

```
(Q b1),(Q b2)...,(Q bn),
((x=b1)\/(x=b2)\/.../\(x=bn)\/(P' x))
   ⊢? (TQ P x)
 where  P =
   \x.((x=a1)\/.../\(x=am)\/
       (x=b1)\/(x=b2)\/.../\(x=bn)\/
       (P' x))
```

[4]ちなみに, 最初の goal は $n=1$, $b_1=a$, $m=0$ となっている.

という goal パターンを求めなければならない．このパターンは，次の二つの自然数のリストを引数としている．

$[a_1,\ldots,a_m]$, $[b_1,\ldots,b_n]$

$[b_1,\ldots,b_n]$ の方は，パターンの二ヶ所に分散して，しかも異なる構造で出現している．このようなパターンを tactic のコードから適切に求める必要がある．さらに二つのリストは，それぞれ重複を含まず，しかも共通の要素を含まない，という制約を満足している．この制約は，tactic もしくは tactic から抽出されたプログラムの停止性を示す際に必要である．以上のパターンが得られれば，二つの自然数のリストを引数とする関数を上述の tactic から抽出することができるだろう．

次のステップでは，求められたパターンを定理証明系の論理体系の中で表現したい．上の例の場合は，自然数のリストが体系の中で定義されており，リスト上の再帰的な関数によって上の goal のパターンが体系の中で定義可能でなければならない．このような定義が可能な場合，co-inductive な定義の結論を満たす自然数が有限個であれば，上の tactic の停止性を体系の中で示すことができる．なぜならば，結論を満たす自然数の個数が N のとき，$N-m$ が T_2 によって常に減少するからである．

5 まとめ

tactic からのプログラム抽出について議論した．tactic は証明を構成する手順を表現するための制御構造を持っており，tactic の再帰的・反復構造をそのままプログラムの再帰的・繰り返し処理として考えることができる．また，tactic の実行，さらには抽出されるプログラムの実行，が停止するという保証をするために，tactic によって書き換えられていく goal がたどるパターンをいかにして見い出すか，また，そのパターンを体系のなかで表現し，tactic による処理の停止性を議論すること，が今後追求すべき課題である．

参考文献:

[1] R.L.Constable et al., *Implementing Mathematics with the Nuprl proof Development System*, Prentice-Hall, 1986

[2] The Coq Proof Assistant Reference Manual, V6.1, 1996

[3] A.Felty, Implementing tactics and tacticals in a higher-order logic programming language, *Journal of Automated Reasoning*, 1993

[4] M.Gordon and T.Melham (ed.), *Introduction to HOL*, Cambridge University Press, 1993

[5] S.Hayashi and H.Nakano, *PX: A Computational Logic*, MIT press, 1988

[6] C.Paulin-Mohring, *Extraction de programmes dans le Calcul des Constructions*, PhD thesis, Université Paris 7, 1989

[7] L.Paulson, *Logic and Computation*, Cambridge University Press, 1987

[8] L.Paulson, *Isabelle: A Generic Theorem Prover*, LNCS828, Springer-Verlag, 1994