ソフトウェア基礎論 31 － 6
（１９８９．９．２０）

$\mathcal{SST}/\Lambda$ に基づく構成的プログラミング

佐藤雅彦、亀山幸義

東北大学 電気通信研究所

構成的論理体系 $\mathcal{SST}$ とその中に含まれるプログラム言語 Λ を用いて、構成的プログラミングを行う試みについて述べる。構成的論理では、仕様を論理式で表現してそれを証明することにより、正しさの保証されたプログラムを得ることができる。このように、証明の形でプログラミングをすることを構成的プログラミングとよぶ。Λ は、$\mathcal{SST}$ に含まれる言語であるが、$\mathcal{SST}$ の証明システムを実現するための言語でもある。ここでは、$\mathcal{SST}$ の証明チェッカー・プログラム抽出システムの Λ による実現、および、それによるプログラムの抽出について論じる。

Constructive Programming based on $\mathcal{SST}/\Lambda$

Masahiko Sato　Yukiyoshi Kameyama

Research Institute of Electrical Communication

Tohoku University

2-1-1, Katahira, Aoba-Ku, Sendai 980, Japan

This paper examines the idea of constructive programming in the framework of the constructive logical system $\mathcal{SST}$, and the programming language Λ. In constructive logic, we can synthesize correct programs by expressing the specification as a formula, and proving it. We call this style of programming constructive. We, then, argue about the implementation of the $\mathcal{SST}$proof system and the program extration system on top of Λ, and about the extraction of programs.

1 初めに

構成的論理（直観主義論理）における証明と計算の対応は良く知られている。有名なMartin-Löfの型理論[5]は、Curry-Howardの同型対応に基づいて論理式と型、証明とその型に属する項をそれぞれ同一視し、項の計算に対応する等式をいれたものである。この場合、証明はそのままプログラムと見ることができる。型のない構成的体系においても、Fefermanの体系T_0[2]や林のPX[4]のように、多くの体系において、$\exists x.A(x)$という論理式の証明から、実際に$A(t)$を満たす項tを抽出できるという性質（term existence property）が成立する。この性質は、排中律のある古典論理では成立し得ないことである。従って、構成的論理における証明を作るという行為は、正しさの保証付きのプログラムを合成する行為と見なすことができる。この立場でのプログラミングを構成的プログラミング(Constructive Programming)とよぶ。

構成的論理を基礎におき、計算機の上に、証明、プログラミングシステムを作る動きは、PXやCornell大学のNuprl[1]を初めとして、近年盛んである。

本稿では、型のない構成的数学体系$\mathcal{SST}$(Symbolic Set Theory)[6]とプログラム言語Λを使って、この構成的プログラミングを実践する。

$\mathcal{SST}$とΛのもうひとつの目的は、形式化した超数学を計算機上で実行することである。論理体系の中で成立する定理のみならず、論理体系に関する超定理の証明を計算機上で行う。この目的のために、Λにより$\mathcal{SST}$の証明チェックシステム、証明からのプログラムの抽出システムを実現した。

2 Λの構文

Λは、$\mathcal{SST}$の項をプログラムと見なすことにより得られる関数型言語である。$\mathcal{SST}$は、その名の通り全ての構文要素をS式（Symbolic Expression）で表現している。言いかえると、変数や定数などあらゆるものが、ただひとつのアトム*から、対を作る操作(_ . _)によって構成される二分木である。ここでは、その表現の詳細には立ちいらないが、全ての構文要素がS式で一様に表現されている、ということは$\mathcal{SST}$自身を取り扱うプログラム、たとえば、構文をチェックするプログラムを書く場合において、有用なことである。もし、S式の代わりに自然数を基礎においたとすると、Gödel numberingのような面倒な符号化が必要になる。

(* . _)の形のS式をアトムとよび、そうでないS式をリストとよぶ。特に、(* . *)というアトムを()と書き、リストに対して、$(a\ b)\equiv(a\ .\ (b\ .\ ()))$等の省略形を用いる。アトムには、シンボル、オブジェクトなどがあり、それぞれ、適当にS式に符号化されている。シンボルは変数名などに利用され、オブジェクトは、真偽値（$t, $f）、自然数（0, 1, ...）、関数オブジェクトなどの基本的なデータタイプを含む。S式sに対して、(* quote s)の形のS式を定数とよび、'sと略記する。定数はλ-calculusの正規形に相当し、定数全体の集合が、計算の値全体の集合となる。以下では、シンボルをl、定数をv、オブジェクトをoであらわす。

変数(x)、項(t)、パターン(p)を以下の通り定義する。

$$x \ ::= \ l \mid \#x$$

変数xから#を全て取り去ったシンボルを$purepart(x)$と書く。

$$t \ ::= \ () \mid x \mid ! \mid v \mid o$$

$$
\begin{array}{ll}
| & \texttt{(cons } t\ t\texttt{)} \mid \texttt{(car } t\texttt{)} \mid \texttt{(cdr } t\texttt{)} \\
| & \texttt{(lambda } p\ t\texttt{)} \mid \texttt{(mu } p\ t\texttt{)} \mid \texttt{(apply } t\ t\texttt{)} \\
| & \texttt{(let ((}p\ t\texttt{)} \cdots \texttt{(}p\ t\texttt{))} \ t\ t\texttt{)} \\
| & \texttt{(case } t\ \texttt{(}p\ t\texttt{)} \cdots \texttt{(}p\ t\texttt{))} \\
| & \texttt{(eq } t\ t\texttt{)} \mid \texttt{(if } t\ t\ t\texttt{)} \ \texttt{(or } t \cdots t\texttt{)}
\end{array}
$$

(apply t (cons s_1 ($\cdots$ (cons s_n ()) $\cdots$)))を、(t s_1 $\cdots$ s_n)と省略する。lambdaは通常のλ抽象で、muは、再帰的関数に対するλ抽象である。

$$
\begin{array}{rl}
p & ::= \ \texttt{()} \mid x \mid \texttt{\_} \mid v \mid o \\
& \quad \mid \ p\ \texttt{:}\ t \ \mid p\ \texttt{=}\ p \ \mid \texttt{(}p\ \texttt{.}\ p\texttt{)}
\end{array}
$$

3 Λの意味論

Λは、値呼びの純粋な関数型言語であり、簡潔な意味論を持つ。ただし、$\mathcal{SST}$の構文を取り扱うという目的からパターンマッチを強力にしている。

$x_1, ..., x_n$をシンボルまたは!、$v_1, ..., v_n$を定数とするとき、((x_1 v_1) $\cdots$ (x_1 v_1))という項を環境とよぶ。

環境から、変数または!の値を取り出す関数assocは以下のように定義される。

1. $e \equiv$()のとき、$assoc(x, e) =$()
2. $e \equiv$((x v) . e')のとき、$assoc(x, e) = v$
3. $e \equiv$((y v) . e'), $x \equiv$ #z, $purepart(x) \equiv y$ のとき $assoc(x, e) = assoc(z, e')$
4. $e \equiv$((y v) . e'), $purepart(x) \not\equiv y$ のとき $assoc(x, e) = assoc(x, e')$

パターン$p_1, \cdots, p_n$と項$t_1, \cdots, t_n$に対し、((p_1 t_1) $\cdots$ (p_1 t_1))の形の項を宣言とよび、パターンマッチをあらわす。外部環境e_1、内部環境e_2のもとでの宣言dのパターンマッチ$match_{e_1}(d, e_2)$は以下のように定義される。(以下、添字e_1は省略する。)

1. $e_2 \equiv$'$f のとき、$match(d, e_2)=$'$f$

   以下、$e_2 \not\equiv$'$fとする。

2. $match($(), $e_2) = e_2$
3. $d \equiv$((s t) . d') (sは()またはオブジェクト) のとき、
   (a) $t \triangleright_{e_1} s$ ならば、$match(d, e_2) = match(d', e_2)$
   (b) $t \triangleright_{e_1} v$, $v \not\equiv s$ ならば、$match(d, e_2) =$'f
4. $d \equiv$((x t) . d')のとき、

(a) $assoc(x, e_2) = v$, $t \triangleright_{e_1} v$ ならば $match(d, e_2) = match(d', e_2)$

(b) $assoc(x, e_2) = v$, $t \triangleright_{e_1} v'$, $v \neq v'$ ならば $match(d, e_2) =$ '$f

(c) $assoc(x, e_2) = ()$, $t \triangleright_{e_1} v$ ならば $match(d, e_2) = match(d', ((x\ t)\ .\ e_2))$

5. $d \equiv ((v\ t)\ .\ d')$のとき、

(a) $t \triangleright_{e_1} v$ ならば $match(d, e_2) = match(d', e_2)$

(b) $t \triangleright_{e_1} v'$, $v \neq v'$ ならば $match(d, e_2) =$ '$f

6. $d \equiv ((p\ :\ s\ \ t)\ .\ d')$のとき、

(a) $match(((p\ t)\ .\ d'), e_2) =$ '$fならば $match(d, e_2) =$ '$f

(b) $match(((p\ t)\ .\ d'), e_2) = e_3$, $(s\ t) \triangleright_{e_1}$ '$tならば $match(d, e_2) = match(d', e_3)$

(c) $match(((p\ t)\ .\ d'), e_2) = e_3$, $(s\ t) \triangleright_{e_1}$ '$fならば $match(d, e_2) =$ '$f

7. $d \equiv ((p$ = $q\ \ t)\ .\ d')$ のとき $match(d, e_2) = match(((p\ t)\ .\ ((q\ t)\ .\ d')), e_2)$

8. $d \equiv (((p\ .\ q)\ t)\ .\ d')$のとき

(a) $t \triangleright_{e_1}$ '$(s_1\ .\ s_2)$ ならば $match(d, e_2) = match(((p$ '$s_1)\ .\ ((q$ '$s_2)\ .\ d')), e_2)$

(b) $t \triangleright_{e_1}$ 'v （v はアトム）ならば $match(d, e_2) =$ '$f

関数 $pmatch$ を、$pmatch_e(d) \equiv match_e(d, ())$ と定義する。

$p\ :\ s$ というパターンは、関数 s に適用して真という値が返るもののみとマッチする。また、p = q というパターンは、同じ定数を p および q とパターンマッチさせる。ペアに関しては、たとえば、$pmatch_{()}$((((x y) '(1 2)))) =((x '1) (y '2)) となる。

Λ の計算は、項 t、環境 e、定数 v の間の三項関係で、t を e のもとで評価したとき v を得ることを、$t \triangleright_e v$ と書く。

1. $t \equiv ()$ あるいは $t \equiv o$ （o はオブジェクト）のとき、$t \triangleright_e$ 't

2. $t \equiv x$ (x は変数) あるいは $t \equiv$! で、assoc(t, e)=v のとき、$t \triangleright_e v$

3. $t \equiv$(eq t_1 t_2)、$t_1 \triangleright_e v$、$t_2 \triangleright_e v$ のとき、$t \triangleright_e$'$t

4. $t \equiv$(eq t_1 t_2)、$t_1 \triangleright_e v_1$、$t_2 \triangleright_e v_2$、$v_1 \neq v_2$ のとき、$t \triangleright_e$'$f

5. $t \equiv$(cons t_1 t_2)、$t_1 \triangleright_e$ 's_1、$t_2 \triangleright_e$ 's_2 のとき、$t \triangleright_e$ '$(s_1\ .\ s_2)$

6. $t \equiv$(car t_1)、$t_1 \triangleright_e$ '$(s_1\ .\ s_2)$のとき、$t \triangleright_e$ 's_1

7. $t \equiv$(cdr t_1)、$t_1 \triangleright_e$ '$(s_1\ .\ s_2)$のとき、$t \triangleright_e$ 's_2

8. $t \equiv$(lambda p t_1)のとき、$t \triangleright_e$ '(obj lambda (p t_1 c))

ただし、c は t 中の自由変数に対し、e のもとでの値を対にした環境。

9. $t \equiv$(mu p t_1)のとき、$t \triangleright_e$'(obj mu (p t_1 c))

10. $t \equiv$(apply t_1 t_2)、$t_1 \triangleright_e$'(obj lambda (p t_3 c))、(let ((p t_2)) t_3 ()) $\triangleright_e v$ のとき、$t \triangleright_e v$

11. $t \equiv$(apply t_1 t_2)、$t_1 \triangleright_e$'(obj mu (p t_3 c))、$e' \equiv$((! t) . e)、(let ((p t_2)) t_3 ()) $\triangleright_{e'}$ v のとき、$t \triangleright_e v$

12. $t \equiv$(if t_1 t_2 t_3)、$t_1 \triangleright_e$'$t、$t_2 \triangleright_e v$ のとき、$t \triangleright_e v$

13. $t \equiv$(if t_1 t_2 t_3)、$t_1 \triangleright_e$'$f、$t_3 \triangleright_e v$ のとき、$t \triangleright_e v$

14. $t \equiv$(let d s_1 s_2)、$pmatch_e(d) = e'$ (e' は環境)、$append(e', e) = e''$、$s_1 \triangleright_{e''} v$ のとき、$t \triangleright_e v$

15. $t \equiv$(let d s_1 s_2)、$pmatch_e(d) =$'$f、$s_2 \triangleright_e v$ のとき、$t \triangleright_e v$

16. $t \equiv$(case t) のときエラー

17. $t \equiv$(case t (p_1 t_1) $\cdots$ (p_n t_n))、$pmatch_e(((p_1\ t))) = e'$ (e' は環境)、$append(e', e) = e''$、$t_1 \triangleright_{e''} v$ のとき $t \triangleright_e v$

18. $t \equiv$(case t (p_1 t_1) $\cdots$ (p_n t_n))、$pmatch_e(((p_1\ t))) =$ '$f,
(case t (p_2 t_2) $\cdots$ (p_n t_n)) $\triangleright_e v$ のとき $t \triangleright_e v$

Lisp の計算では、'$s \triangleright s$ となるため、normal form に相当する概念を定義できない。従って、Lisp の計算をそのまま等式論理に埋めこむことはできない。一方、Λ の計算においては、計算の結果は必ず定数になり、定数はそれ以上評価しても変わらないので、意味論を明確に定義することができる。

eq はいわゆる intensional equality である。すなわち、関数としての extensional な等しさではなく、S 式としても syntactical な等しさを見る関数である。lambda による関数を評価すると、関数閉包を生成する。この閉包には、自由変数のその時の値がはいり、たとえば、$e \equiv$((y '1)) のとき、(lambda (x) (cons x y)) $\triangleright_e$ '(obj lambda ((x) (cons x y) ((y '1)))) mu については、lambda とほぼ同じであるが、apply で適用するときに、! にその関数閉包を束縛した環境で評価する。これによって、再帰的関数の計算ができる。たとえば、

```
(mu (x y) (if (eq x ()) y (cons (car x) (! (cdr x) y))))
```

により、関数 *append* を定義することができる。let, case は上で定義したパターンマッチを使うためのものである。

Λ の計算について以下の定理が成立する。

定理 **1** (計算結果の一意性)

t を項、u と v を定数、e を環境とするとき、

(1) $a \triangleright_e u$ かつ $a \triangleright_e v$ ならば、$u \equiv v$

(2) $u \triangleright_e u$

Λ のプログラムは型を持たないので、Y-combinator に相当するものを作って再帰的関数を定義することができる。従って、mu などは、Λ のプリミティブとしては不要であるが、実際に、Λ で大きなプログラムを書いて、それについての推論を行う際には、再帰的関数を直接扱うことができる方が便利であるので、最初からいれることとした。

4 $\mathcal{SST}$

この章では、$\mathcal{SST}$ について簡単に説明する。$\mathcal{SST}$ は、一階の直観主義論理に基づいた集合論である。

F ::= `$t`
| `$f`
| `X`
| `(eq` t t`)` $(t = t)$
| `(set` t`)` $(Set(t))$
| `(mem` t t`)` $(t \in t)$
| `(and` F F`)` $(F \wedge F)$
| `(or` F F`)` $(F \vee F)$
| `(if` t F F`)` $(if\ t\ then\ F\ else F)$
| `(imp` F F`)` $(F \supset F)$
| `(apply (lambda (x)` F`)` t`)` $(\lambda x.F)(t)$
| `(apply (mu (X) (lambda (x)` F`))` t`)` $(\mu X.\lambda x.F)(t)$
| `(all (lambda (x) F))` $(\forall x.F)$
| `(ex (lambda (x) F))` $(\exists x.F)$
| `(case` t $(p\ F) \cdots (p\ F)$`)`

かっこ内は、通常の論理における表現である。X は、mu で束縛される述語変数で、論理式の中に自由に現れてはいけない。

この定義から明らかに全ての論理式は項である。

$\mathcal{SST}$ の論理は、通常の論理と違い、部分項の論理 (logic of partial terms) である。$t = t$ という論理式は、t の計算が止まる時のみ成立する。$t = t$ を $t\downarrow$、$t\downarrow \wedge s\downarrow \supset s = t$ を $s \simeq$ と省略する。変数は計算が止まるもののみを表すので、$\exists$ の導入や $\forall$ の除去の規則は、以下のような形になる。

$$\frac{A(t) \quad t\downarrow}{\exists x.A(x)} \qquad \frac{\forall x.A(x) \quad t\downarrow}{A(t)}$$

and 導入などの他の論理の規則は普通の直観主義論理と同じである。このほか、
t↓ ⊃ (car (cons s t)) ≃ s などの計算に関する公理がある。

if や case という論理式は、項としての if や case をそのまま論理式に導入したものである。これらの論理式は or や ex と異なり計算の情報を持たないので、再帰的述語や集合を定義する論理式の中に書くことができる。そのように定義された論理式の証明から抽出されるプログラムは、(項としての) if や case を使ったものになる。

μ オペレータは、述語を再帰的に定義するために用いられる。直観的には、$\mu X.\lambda x.F(X,x)$ は、$F(X,x) \equiv X(x)$ となる最小の X を表す。ただし、F として許されるのは、Harrop 論理式（$\exists$ と $\vee$ がたかだか $\supset$ の前半部分にしかこない論理式）であって、X が positive にあらわれるものである。Harrop 論理式は計算の意味を持たないので、再帰的に定義される述語も計算の意味を持たず、realizer をつけるときに定義が簡単になる。

たとえば、自然数をあらわす述語 Nat は、以下のように定義できる。

$Nat \equiv$
```
(mu (X) (lambda (x)
    (case x
          ((* 'obj 'num ()) $t)
          ((* 'obj 'num y) x) (X y)))
```

ここでは、'(* obj num n)を n の successor としている。通常、自然数は orを使って定義されるが、ここでは再帰的定義に orを使うことはできないので、case を使う。

$M \equiv \mu X.\lambda x.F(X,x)$ とおくと、μ オペレータにともなう帰納法は以下の通りである。

$$\forall x.(F(A,x) \supset A(x)) \supset \forall x.(M(x) \supset A(x))$$

集合は、論理式から elmentary comprehension の推論規則により作られる。

$$\frac{Set(A_1) \ \ldots \ Set(A_k)}{Set(\{x|H\})}$$

{x | H} は、(lambda (x) H) の省略形である。任意の論理式 H に対して、この推論規則が成立するわけではなく、H が Harrop かつ elementary (H 中の $\in$ の右側には $A_1, \cdots, A_n$ のいずれかの項しかこない) でなければならない。この規則を使って実数の集合などを定義することにより、Bishop 流の構成的数学を展開することができる。

$\mathcal{SST}$ と Λ の関係を示すものとして、以下の定理が成立する。

定理 2　t を項、u を定数、e を環境、t の自由変数は全て e で値を持っているとき、

$$t \triangleright_e u \quad iff \quad E(e) \vdash_{\mathcal{SST}} t = u$$

ただし、$E(e)$ は、環境 e から作られる等式を並べたものである。たとえば、
$E(((x\ '1)\ ((y\ z)\ '(3\ 4))))$ は、$x =$'1, (cons y z)='(3 4) である。

5　Λ による $\mathcal{SST}$ の証明システムの実現

Λ 上で、$\mathcal{SST}$ を取り扱うためには、まず、$\mathcal{SST}$ の項、論理式、証明であるかどうか判定するプログラムが必要である。我々の目的は、単に Λ 上の証明チェッカーを書くことではなく、そのプログラム自身の性質（たとえば、任意の入力に対して、計算が停止すること）を $\mathcal{SST}$ によって推論することである。従って、このようなプログラムが比較的簡潔に書けるように、Λ に強力なパターンマッチを導入した。

Λ による論理式および証明の定義例

```
; sort is a sublist of (harrop postive negative)
(defmu form1 (F sort)
```

```
    (let ((form (lambda (F) (! F sort))))
 (case F
   (('eq _:term _:term) $t)
   (('mem _:term _:term) $t)
   (('set _:term) $t)
   (('and _:form _:form) $t)
   (('or _:form _:form) (not (member 'harrop sort)))
   (('imp F _:form) (form1 F pv (negate sort)))
          ...
(defun formula (F) (form1 F '(positive negative)))
(defun harrop (F) (form1 F '(harrop)))
(defun assump (L) (and . (map (lambda (F) (formula F)) L)))

;; P is (conclusion assumption-list last-rule subproof ... subproof)
(defmu proof (P)
    (case P
       ((F:formula (F) 'assume) $t)
       ((F:formula () 'axiom) (axiom F))
       ((('and F G) L:assump 'and-intro (F M . _):! (G N . _):!)
        (equal-list L (merge-list M N)))
       ((F:formula L:assump 'and-elim1 (('and F _)  L . _):!) $t)
       ((F:formula L:assump 'and-elim2 (('and _ F)  L . _):!) $t)
       ((('or F _):formula L:assump 'or-intro1 (F L . _):!) $t)
       ((('or _ F):formula L:assump 'or-intro2 (F L . _):!) $t)
       ...
```

defun, defmu はそれぞれ、lambda, mu の関数閉包を作って現在の環境に付加する関数である。関数 proof は、P が証明に'をつけたものであれば'$tを、そうでなければ '$fを返す。

6 証明からのプログラム抽出

$\mathcal{SST}$ の証明からプログラムを抽出するためには、realizability interpretation を用いる。「項 t が論理式 F を realize する」ことを表す論理式を $F[t]$ を書くことにする。realizer t は、F の証明の構造にほぼ対応する。たとえば、

$(A \vee B)[t] \equiv$ (car t) $\downarrow \wedge$ *if* (car t) *then* A[(cdr t)] *else* B[(cdr t)]

であり、この realizer の中に、A と B のどちらが本当に成立するかの情報がはいっている。ここでは、詳しい realizer の定義は省略して、[6] を参照するにとどめる。

定理 3 $\Gamma \vdash F$ の証明から、$\Gamma, \Gamma[\gamma] \vdash e \downarrow \wedge F[e]$ となる項 e を得られる。

系 1 $\vdash \forall x.\exists y.F(x,y)$ の証明から、$\vdash e \downarrow \wedge \forall x.F(x,(e\ x))$ となる項 e を得られる。

```
(defmu extract (P RV NV)
  (case P
```

```
((_:harrop . _) dummy)
((F _ 'assume) (cdr (assoc F RV)))
((_ _ 'axiom) dummy)
((_ _ 'top-intro) dummy)
((_ _ 'bottom-elim Q) dummy)
(('and _ _) _ 'and-intro Q R) (cons (! Q RV NV) (! R RV NV)))
((_ _ 'and-elim1 Q) (car (! Q RV NV)))
((_ _ 'and-elim2 Q) (cdr (! Q RV NV)))
((_ _ 'or-intro1 Q) (cons $t (! Q RV NV)))
((_ _ 'or-intro2 Q) (cons $f (! Q RV NV)))
...
```

ここで、RVは、仮定にくる論理式とそのrealizerの対のリスト、NVは、新しい変数のリストである。

extractは、上の定理の証明に忠実に、realizerを抽出するプログラムである。たとえば、Pにおける最後に適用された推論規則が、orの導入規則であれば、上で述べたようなrealizerの定義に従って、$tまたは$fを、残りの部分のrealizerと対にする。また、Harrop論理式は計算情報がないので、dummyのコードを生成する。

例: 割り算プログラムの抽出

割り算の仕様は、

$$\forall m.Nat(m) \supset \forall n.Nat(n) \wedge n > 0 \supset .\exists r.Nat(r) \wedge \exists s.Nat(s) \wedge (m = n*r+s) \wedge (0 \leq s < n)$$

である。この論理式を証明するためには、

$$(A(0) \wedge (\forall p.Nat(p) \wedge p < m \supset A(p)) \supset A(m)) \supset \forall m.Nat(m) \supset A(m)$$

(ただし、$A(m) \equiv \exists r.Nat(r) \wedge \exists s.Nat(s) \wedge (m = n*r+s) \wedge (0 \leq s < n)$)

という累積帰納法を用いるのが自然である。累積帰納法は、論理的には通常の自然数に対する帰納法から導くことができるが、その証明から得られるプログラムは、最終的に自然数の帰納法に頼るので効率が悪いことが多い。そこで、通常の帰納法から累積帰納法を導く代わりに、まず、累積帰納法を直接適用できる述語Nat'を定義する。

$Nat' \equiv$
```
(mu (X) (lambda (x)
    (case x
        (('* 'obj 'num ()) $t)
        (('* 'obj 'num y)
          (all (lambda (z) (imp (< z y) (X z))))))))
```

ここで、(< z y)≡(eq (− (+ z 1) y) 0)である。Nat'に対する帰納法は、上の累積帰納法になる。

次に、$(Nat\ x) \equiv (Nat'\ x)$を言えば、$Nat'$に対する帰納法を普通の自然数に対して適用することができる。

```
(defmu div (m n)
   (if (< m n) (cons 0 m)
     (let (((r . s) (! (- m n) n))) (cons (+ r 1) s))))
```

この部分の証明はプログラムに反映しないので、抽出されたプログラムは、累積帰納法を直接適用したように見えることである。この割り算のような単純な例では両者の差ははっきりしないが、たとえば、quick-sort プログラムを抽出する場合は、ここで述べたような方法が必要になる。

プログラム抽出の際の問題として、実際に抽出されるプログラムに、不要なコードがいくつかはいっていることが多く、それをいかに実際に効率の良いプログラムとするか、ということがある。

7 終わりに

論理体系とプログラム言語の関係という意味で、$\mathcal{SST}$ と Λ の関係に似ているものは、Milner らの Edinburgh LCF と ML の関係 [3] である。LCF は、Scott 理論に基づいて、PPλ という pure な関数型言語についての推論を行う。一方、LCF の証明系は、ML という強力な言語によって実現されていた。この場合、ML は PPλ よりはるかに複雑であるので、LCF の証明系の中で ML のプログラム（特に、この証明系自身）の性質を推論することはできない。ML のプログラムの性質について論じるためには、LCF よりはるかに複雑な体系が必要であろう。

一方、$\mathcal{SST}$ と Λ は、最初からこのことを目的として設計されている。すなわち、$\mathcal{SST}$ によって Λ のプログラムの性質を推論することができ、一方、Λ によって $\mathcal{SST}$ の証明系を実現することができる。これが完成すれば、Λ で書かれた $\mathcal{SST}$ の証明系の性質について、その証明系自身を用いて計算機上で推論することが可能になり、他の体系に頼らない閉じた世界を作ることができる。本稿で述べた $\mathcal{SST}$ の証明システムやプログラム抽出システムは、最終的には、この証明システム自身でその性質を推論する予定である。

このほか、将来の課題としては、$\mathcal{SST}$ の内部の定理でなく、$\mathcal{SST}$ に関するメタ定理（たとえば、第6章定理3）や、Gödel の第二不完全性定理などを $\mathcal{SST}$ で表現して計算機上で証明することがあげられる。

参考文献

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

[2] Feferman, S., 1979: Constructive theories of functions and classes, pp. 159-224 in Boffa, M., van Dalen, D. and McAloon, K. (eds.), *Logic Colloquium '78*, North-Holland.

[3] Gordon, M. J., Milner, R., and Wadsworth, C. P., 1979: *Edinburgh LCF*, Lecture Notes in Computer Science, Vol. 78, Springer-Verlag.

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

[5] Martin-Löf, P., 1982: Constructive mathematics and computer programming, pp. 153-179 in Cohen, L. J., Pfeiffer, H., and Podewski, K. P. (eds.), *Logic, Methodology, and Philosophy of Science VI*, North-Holland.

[6] Sato, M., to appear: Symbolic Set Theory, in Mathematical Logic and its Application: Proceedings.