数理解析研究所講究録
第516巻 1984年 12-21

Constructive mathematics and program synthesis

Susumu HAYASHI

The Metropolitan College of Technology
Asahigaoka 6-6, Hino, Tokyo

林　晋

Connection between constructive mathematics and "computing" has been recognized by some logicians Scott, Martin-Löf etc. and by some computer scientists Constable, Goto, Takasu, Sato etc.. Some leading logicians, e.g. Scott and Martin-Löf, seem to believe that constructive logic is "a logic of computing". Martin-Löf [5] showed that his theory of types can be regarded as a programming language with verification. (Indeed, an interpreter of Martin-Löf's theory of type has been implemented on computer by Nordström [4].) On the other hand, Dana Scott suggested to try to see the possibility of using realizability for interpreting a "logic of computability" in his lecture in Logic Colloquium '83.

In Hayashi [3], the author introduced a formal theory **LM**, which is a Feferman type formal theory of class and algorithm (cf. Feferman [1]). In **LM**, algorithms are described by (pure) **LISP** programs, and the author showed how to extract **LISP** programs from proofs of **LM** by using realizability based on **LISP**. A system based on **LM** had been implemented, but the author recognized some defects of **LM** through using it. One of them is the lack of inductive definition as its fundamental facility. Inductive definitions make possible to define some kinds of data types such as lists and trees in a natural way. A new system **PX** (Program eXtractor) remedies this defect by introducing a sort of *deterministic* inductive definition **CIG**. In the following, we will sketch **PX**, and show some examples in **PX**.

PX has been implemented by the author on VAX/UNIX at Computer Center of University of Tokyo. The author would like to express his hearty thanks to Prof. Masahiko Sato for discussions on the subjects.

PX as a package of Franz Lisp

PX can be thought as a package or an extension of **Franz Lisp**. **PX** extracts and verifies (pure) **Franz Lisp** functions. Its **PX** top-level is a modification of **CMULisp** top-level (cf. Foderaro & Sklower[2]). It has a history mechanism, where users can edit events by **ex**, **vi** or the Lisp Editor.

Syntax of PX

constant ::= individual constant | class constant

variable (var) ::= individual variable (a,b,..)

| class variable (X,Y,..)

| term variable (?1,?2,..)

term (tm) ::= variable | constant (function term ... term)

function ::= system function

| user-defined function

| (lambda (var ... var)) tm)

formula ::= implicative | implicative <-> implicative

implicative ::= disjunctive | disjunctive -> implicative

disjunctive ::= conjunctive | conjunctive + disjunctive

conjunctive ::= monadic | monadic & conjunctive

monadic ::= atomic | parenthetic | quontative | - monadic

parenthetic ::= (formula)

quontative ::= (UN cvl) pq-formula | (EX cvl) pq-formula

pq-formula ::= parenthetic | quontative

atomic ::= tm = tm | tm : tm

| CL tm | E tm

| gp (tm ... tm) | dp (tm ... tm)

| TRUE | FALSE

gp ::= generic predicate (predicate free variable)

dp ::= defined predicate (Users can abbreviate formulae by dp.)

cvl ::= var ... var : tm | var ... var

Examples of formulae.

(UN a b : N) (EX q r : N)
(- b = 0 -> a = (add (product b q) r) & (lessp r b) = t)

(UN a : (List X)) (a = nil + (car a) : X & (cdr a) : (List X))

Logic of PX (logic of partial terms)

$$\frac{\mathbf{P(tm)} \quad \mathbf{tm} : C}{(EX\ x : C)\ \mathbf{P(X)}}, \quad \frac{\mathbf{P(?1)} \quad [?1 : C]}{(UN\ x : C)\ \mathbf{P(x)}}, \quad \frac{\mathbf{P(?1)} \quad [\mathbf{ASP}(?1)]}{\mathbf{P(tm)} \quad [\mathbf{ASP(tm)}]}, \quad \frac{\mathbf{P(x)} \quad \mathbf{E\ tm} \quad [\mathbf{ASP}(x)]}{\mathbf{P(tm)} \quad [\mathbf{ASP(tm)}]}$$

Axioms of PX

1. axioms on primitives.
?1 : Dp -> E (car ?1), E (add1 ?1) <-> ?1 : N, E (car ?1) -> ?1 : Dp,

?1 : Atm <-> (atom ?1) = t, ?1 : Dp <-> (dtpr ?1) = t,

(list ?1 ?2 ..) = (cons ?1 (list ?2 ..)),

?1 : T -> (cond (?1 ?2) (?3 ?4) ..) = ?2, x : T <-> - x = nil,

E ((lambda (x y ..) ?1) ?2 ?3 ..) -> E ?2 E ?3 ..,

etc...
These axioms are available through a function "so" as follows:

```
1.(assume '{?1 : Dp})  ; A formula must be enclosed by braces.
{.]- ?1 : Dp}
2.(so '{E (car ?1)} it)
{.]- E (car ?1)}
3.(asp it)
({?1 : Dp})
```

2. CIG (conditional inductive generation).

This is a special case of Feferman's inductive generation, but much more practical than the general one. E.g., the class of lists of elements of a class A is declared as follows:

```
1.(deCIG {x : (List A)}
        ((dtpr x) {(car x) : A} {(cdr x) : (List A)})
        (t {x = nil}))
List
```

CIG can be used to define classes simultaneously. See Appendix 1 for an example. For each **CIG**-declaration, an induction principle is associated. See Appendix 2 for an example of such an induction.

3. $\mathbf{ECA}^{(-)}$

Negative elementary comprehension axiom $\mathbf{ECA}^{(-)}$ (cf.[1],[3]) is available as a class declaration as follows:

```
1.(deECA {x : (Cartesian X Y)} {(car x) : X & (cdr x) : Y})
Cartesian  ; Cartesian is decalred.
2.so {CL (Cartesian Atm Dp)}  ; Cartesian is class-valued.
{}- CL (Cartesian Atm Dp)}
3.axiom Cartesian  ; the axiom of Cartesian.
{}- x : (Cartesian X Y) <-> (car x) : X & (cdr x) : Y}
```

$\mathbf{ECA}^{(-)}$ can be derived from **CIG**, so $\mathbf{ECA}^{(-)}$ is defined as a macro in **PX**.

4. Join.

This is the same as Feferman [1]. This corresponds to coproduct. So it is useful to define a data type such as *record* of **PASCAL**. (See Martin-Löf [5].)

5. Induction for N and V.

N is the class of natural numbers and **V** is the domain of **PX**, i.e. the class of S-expressions. See Hayashi [3] and Appendix 3 for such induction principles.

6. Using extracted functions.

Extracted functions can be named and used as follows:

```
1.(setq tm '(cons (cons a b) (cons c d)))
(cons (cons a b) (cons c d))
2.(exI '{(EX z) (z = ,tm)} (so '{,tm = ,tm}]
{}- (EX z) (z = (cons (cons a b) (cons c d)))}
3.(deEXFUN foo (b c a d) it)
foo
4.axiom foo
{}- (EX z) ((foo b c a d) = (list z) & z = (cons (cons a b) (cons c d)))}
```

7. Structural induction rule is not included.

Structural induction rule **SIR** of Hayashi[3] is not included. Practically, **SIR** will be substituted by its weak form $\mathbf{SIR}_0$ of Hayashi [3]. Furthermore, $\mathbf{SIR}_0$ will be substituted by the induction principle associated to **CIG**. So we will not make use of **SIR** in construction of actual proofs of specifications. But the system with **SIR** or full inductive generation (**IG**) is proof theoretically stronger than **PX**. So these might be included optionally in the future version of **PX**.

References

[1] Feferman, S., *Constructive theories of functions and classes,* Logic Colloq. 78, D. van Dalen et al. eds., 1978

[2] Foderaro, J. K. & Sklower K. L., *The FRANZ LISP Manual, ,University of California, Berkeley, 1982*

[3] Hayashi, S., *Extracting Lisp Programs from Constructive Proofs: A Formal Theory of Constructive Mathematics Based on Lisp,* Publications of RIMS, Vol. 19, No. 1 , 1983

[3] Nordström, B., *Programming in Constructive Set Theory: Some Examples,* Proceedings of 1981 Conference on Functional Programming Languages and Computer Architecture, 1981

[4] Martin-Löf, P., *Constructive Mathematics and Computer Programming,* Logic, Methodology and Philosophy of Science VI, Studies and the Foundations of Mathematics 104, North-Holland, 1982

Appendix 1.

The following are examples of **CIG**.

```
1.% cat test    ; the contents of the file "test".
(setq tree
  '(deCIG {a : (Tree A)}
          ((atom a) {a : A})
          (t {(car a) : (Tree A)} {(cdr a) : (Tree A)})))

(setq even-odd
  '(deCIG ({x : Even} (in N)
             ((equal x 0) {TRUE}) (t {(sub1 x) : Odd}))
          ({x : Odd}  (in N)
             ((equal x 1) {TRUE}) (t {(sub1 x) : Even}))))

0
2.sload test    ; loading the test file.
tree
even-odd
nil
3.(eval tree)    ; Tree is defined.
(Tree A)
4.axiom Tree     ; the axiom of Tree
{}- (atom a) : T & a : A + (atom a) = nil & (car a) : (Tree A)
& (cdr a) : (Tree A) <-> a : (Tree A)}
5.(eval even-odd)     ; Even and Odd are defined simultaneously.
(Even Odd)
6.axiom Even          ; the axiom of Even
{}- x : N & ((equal x 0) : T + (equal x 0) = nil & (sub1 x) : Odd)
<-> x : Even}
```

Appendix 2.

The following is an example of the induction principle associated to **CIG** declaration of List.

```
1.axiom List    ;the axiom of List
}]- (dtpr x) : T & (car x) : A & (cdr x) : (List A) +
(dtpr x) = nil & x = nil <-> x : (List A)}
2.% cat cig.test      ; the contents of the file "cig.test"
 (setq
   mj (assume '{a : (List X)})
   lemma1 (so '{x : Dp} (assume '{(dtpr x) : T}))
   lemma2 (assume '{x = nil})
   lemma3 (disjI '{x = nil} lemma1)
   lemma4 (disjI lemma2 '{x : Dp})
   th (cigIND 'x mj lemma3 lemma4))    ;the induction associated to List
(lp mj lemma1 lemma2 lemma3 lemma4 th)  ;listing proved theorems.
0
3.load cig.test    ; loading cig.test
[load /usr/usr1/a6647/px/cig.test]
mj.
  {a : (List X)}
    from
  [1] {a : (List X)}

lemma1.
  {x : Dp}
    from
  [1] {(dtpr x) : T}

lemma2.
  {x = nil}
    from
  [1] {x = nil}

lemma3.
  {x = nil + x : Dp}
    from
  [1] {(dtpr x) : T}

lemma4.
  {x = nil + x : Dp}
    from
  [1] {x = nil}

th.
  {a = nil + a : Dp}
    from
  [1] {a : (List X)}
```

Appendix 3.

The following displays a proof of the Euclidian division theorem., which is based on mathematical induction derived from the following definition of N.

```
(deCIG {x : N} (in Atm) ((zerop x) {TRUE}) (t {(sub1 x) : N}))

; type declarations
(setq TYa (assume '{a : N})
      TYb (assume '{b : N})
      TYq1 (assume '{q1 : N})
      TYr1 (assume '{r1 : N}))

; abbreviations
(setq
  b*q1+r1
  '(plus (times b q1) r1)
  hyp1
  '{(sub1 a) =  ,b*q1+r1 & (lessp r1 b) = t}
  HYP1 (assume hyp1)
  case1
  '{(lessp (add1 r1) b) = t}
  case2
  '{(lessp (add1 r1) b) = nil})

(defun division (a) '{(EX q r : N) (,a = (plus (times b q) r) & (lessp r b) = t)})


; lemmata
(setq
  LEMMA1        ; ..]- {a = (plus (times b q1) (add1 r1))}
  (eval
  '[eqchain
    a = (add1 (sub1 a))              (assume '{(sub1 a) : N})
      = (add1 ,b*q1+r1)              (conjE HYP1 1)
      = (plus (times b q1) (add1 r1))  ])

  TMP
   (so '{(add1 r1) = b}
     (so '{(lessp b (add1 r1)) = nil} (conjE HYP1 2)) (assume case2))

  TMP1
   (so '{(times b (add1 q1)) : N} TYb (so '{(add1 q1) : N} TYq1))

  LEMMA2        ; .....]- {a = (plus (times b (add1 q1)) 0)}
   [eqchain
    a = (plus (times b q1) (add1 r1))  LEMMA1
      = (plus (times b q1) b)          TMP
```

```
      = (plus b (times b q1))
      = (times b (add1 q1))
      = (plus (times b (add1 q1)) 0)          TMP1 ])

(setq
 BASIS   ; the case {(zerop a) : T}  (T is the class of non-nil objects)
 (exI (division 'a)
   (conjI
        (trns
         (so '{a = 0} (assume '{(zerop a) : T}))
         (Equal '{0 = (plus (times b 0) 0)}
                (so '{0 = (plus 0 0)}) (so '{0 = (times b 0)} TYb)))
        (assume '{(lessp 0 b) = t}))))

(setq
 CASE1
 (exI (division 'a) (conjI LEMMA1 (assume case1)) TYq1 TYr1)
 CASE2
 (exI (division 'a) (conjI LEMMA2 (assume '{(lessp 0 b) = t})) TYq1))

(setq
 IND-STEP  ; the case {(zerop a) = nil}
 (exE
  (assume
   '{(EX q1 r1 : N) ((sub1 a) =  (plus (times b q1) r1) & (lessp r1 b) = t)})
  (disjE (so '{_case1 + _case2}
             (so '{(add1 r1) : N} TYr1) TYb)
         CASE1
         CASE2)))

(setq DIVISION (cigIND '{a : N} BASIS IND-STEP))
The conclusion and assumptions of DIVISION is as follows:
 {(EX q r : N) (a = (plus (times b q) r) & (lessp r b) = t)}
   from
 [1] {a : N}
 [2] {b : N}
 [3] {(lessp 0 b) = t}
```

The following are the extracted realizers of the above proof. (<rec0> a b) realizes DIVISION, and <rec0>-aux is an auxiliary function for <rec0> with a *global* variable <sv>00186. If <rec0> is defined by a recursion directly, *b* is pushed on a stack at each call of <rec0>. But it is not necessary and wastes spaces and times.

```
(def <rec0>
 (lambda (a b)
  (let ((<sv>00186 b)) (<rec0>-aux a))))
```

```
(def <rec0>-aux
 (lambda (a)
  (cond ((zerop a) (list 0 0))
        (t
         (let (((@1:1 @1:2) (<rec0>-aux (sub1 a))))
           (cond ((lessp (add1 @1:2) <sv>00186)
                  (list @1:1 (add1 @1:2)))
                 (t (list (add1 @1:1) 0))))))))
```