Programming language theory

Stanislav Sumec

Abstract

This article deals with elementary knowledge of the programming language theory. It is divided into two parts. There is shown how we can formal prove program by Floyd-Hoare Logic in the first one. At the beginning is defined a little programming language with basic commands and further are shown some axioms and rules of Floyd-Hoare Logic. At the end of the first part is introduced an example of proving simple program. The second part attends to λ-calculus. There is explained its notation and using for representing functions and data objects as number or list. Further there is shown which classes of functions can be in λ-calculus represented.

1. Proving programs correct

1.1. A little programming language

Foremost will be defined a little programming language whereat will be shown how prove the program correct. The following notation of symbols is used:

- V, V_1, …, V_n – arbitrary variables (X, Y, …)
- E, E_1, …, E_n – arbitrary expression (X + Y)
- S, S_1, …, S_n – arbitrary statements, conditions (X < Y)
- C, C_1, …, C_n – arbitrary commands which are described below

There can be used these commands:

- Assignments – $V := E$ – the state is changed by assigned the value of E to variable V
- Sequence – *C*1; …; *C*n – command are executed in that order
- Blocks – BEGIN VAR V_1; …; VAR V_n; C END – C is executed and values of V_1, …, V_n are restored to the values they had before the block was entered. So the variables V_1, …, V_n are local for the block. Its initial values are unspecified.
- One-armed conditionals – IF S THEN C – if the statement S is true then C is executed
- Two-armed conditionals – IF S THEN C_1 ELSE C_2 – if the statement S is true then C_1 is executed, if the statement S is false then C_2 is executed
- WHILE-commands – WHILE S DO C – if the statement S is true then C is executed and then WHILE-command is repeated.
- FOR-commands – FOR $V := E_1$ UNTIL E_2 DO C – let initial values of E_1, E_2 are e_1, e_2 then command C is executed $(e_2 – e_1) + 1$ if $e_1 <= e_2$

1.2. Hoare's notation

Hoare's notation $\{P\}$ C $\{Q\}$ enable specify what a program does. This expression is called **a partial correctness specification** and we say that is true if program C executes with initial state of variables specified by condition P and terminate then condition Q hold. P is called precondition and Q postcondition. The partial correctness specification does not say that program terminate. Stronger kind of specification is **a total correctness specification**, which can be written $[P]$ C $[Q]$. It says if a program C is executed in a state satisfying P, then C terminate and after termination Q hold. Relationship between partial and total correctness is

Total correctness = Termination + Partial correctness. Partial correctness is used because it is easier to prove it and termination can be established separately.

1.3. Floyd-Hoare Logic

Floyd-Hoare logic provides axioms and rules that make possible construct formal proof of partial correctness specification. A proof in Floyd-Hoare logic is a sequence of lines, each of which is either an axiom of the logic or follows from earlier lines by a rule of inference of the logic. It is important to use formal proofs to ensure that only sound methods of deduction are used. If any of axioms or rules is unsound then we can proof false conclusion.

The inference rules of Floyd-Hoare logic will be specified with a notation of the form:

$$\frac{\mathbf{a}\ S_1, \ldots, \mathbf{a}\ S_n}{\mathbf{a}\ S}$$

Notation $\mathbf{a}\ S$ means that statement *S* has a proof. So the inference rule means that the conclusion $\mathbf{a}\ S$ can be deduced from the hypotheses $\mathbf{a}\ S_1, \ldots, \mathbf{a}\ S_n$, which can be theorems of Floyd-Hoare logic or theorems of mathematics.

One of axioms of Floyd-Hoare Logic is **the assignment axiom** that represents the fact that value of a variable V after executing an assignment command $V := E$ equal the value of expression E in the state before executing it. Formally we can say if statement *P* is true after the assignment, then the statement obtained by substituting *P*[*E*/*V*] must be true before executing it. So the assignment axiom has form:

$$\mathbf{a}\ \{P[E/V]\}V := E\{P\}.$$

An example instance if assignment axiom is $\mathbf{a}\ \{X+1=n+1\}X := X+1\{X=n+1\}$. According to the assignment axiom *V* is *X* and *E* is $X+1$. So if *P*[*E*/*V*] is equal to $X+1=n+1$ then *P* must be $X=n+1$.

Useful rules of Floyd-Hoare logic are rules so-called rules of consequence, which enable simplify preconditions and postconditions. The first of these is rule called **precondition strengthening** with form:

$$\frac{\mathbf{a}\ P \Rightarrow P',\ \mathbf{a}\ \{P'\}C\{Q\}}{\mathbf{a}\ \{P\}C\{Q\}}.$$

This rule says that if precondition *P* implies condition *P'* then the conclusion can be deduced from this implication and $\mathbf{a}\ \{P'\}C\{Q\}$. So if in the example of the assignment axiom hold $X+1=n+1 \Rightarrow X=n$ then precondition can be simplified to form $X=n$. The second of rules of consequence is **postcondition weakening** that has form:

$$\frac{\mathbf{a}\ \{P\}C\{Q'\},\ \mathbf{a}\ Q' \Rightarrow Q}{\mathbf{a}\ \{P\}C\{Q\}}.$$

The rule enable deduced conclusion so that we can say $\{P\}C\{Q\}$ is true if $\{P\}C\{Q'\}$ and *Q'* implies original condition *Q*.

Next rules are **conclusion**

$$\frac{\mathbf{a}\ \{P_1\}C\{Q_1\},\ \mathbf{a}\ \{P_2\}C\{Q_2\}}{\mathbf{a}\ \{P_1 \wedge P_2\}C\{Q_1 \wedge Q_2\}}$$

and **disjunction**

$$\frac{\mathbf{a}\ \{P_1\}C\{Q_1\},\ \mathbf{a}\ \{P_2\}C\{Q_2\}}{\mathbf{a}\ \{P_1 \vee P_2\}C\{Q_1 \vee Q_2\}}.$$

They enable to combine different specifications about the same commands. This means that we can for example split proving of command to two ways with separate conditions.

The sequencing rule is useful for a partial correctness specification a sequence of commands so that commands can be proved separately. Its form is

$$\frac{\mathbf{a}\ \{P\}C_1\{Q\},\ \mathbf{a}\ \{Q\}C_2\{R\}}{\mathbf{a}\ \{P\}C_1;C_2\{R\}}\ .$$

You can see that postcondition of command C_1 is equal to precondition of command C_2. It is possible concatenate several commands.

Speed up of deduction enables **the derived sequencing rule**. It is combination of sequencing and consequence rules. This rule consists from sequence of command where postcondition of every command implies precondition of following command.

$$\frac{\begin{array}{c} \mathbf{a}\ P \Rightarrow P_1 \\ \mathbf{a}\ \{P_1\}C_1\{Q_1\}\ \ \mathbf{a}\ Q_1 \Rightarrow P_2 \\ \mathbf{a}\ \{P_2\}C_2\{Q_2\}\ \ \mathbf{a}\ Q_2 \Rightarrow P_3 \\ \mathrm{M} \\ \mathbf{a}\ \{P_n\}C_n\{Q_n\}\ \ \mathbf{a}\ Q_n \Rightarrow Q \end{array}}{\mathbf{a}\ \{P\}C_1;...;C_n\{Q\}}$$

Similar to the sequencing rule is **the block rule**. In addition this rule takes care of local variables. Its form is

$$\frac{\mathbf{a}\ \{P\}C\{Q\}}{\mathbf{a}\ \{P\}\mathrm{BEGIN\ VAR}\,V_1,\ldots,\ \mathrm{VAR}\,V_n, C\ \mathrm{END}\{Q\}}\ ,$$

but variables V_1, …, V_n do not occur in P or Q.

To specifying blocks with several command sequence can be used **the derived block rule**. This rule is similarly to the derived sequencing rule but there can be used local variables as in previous rule and enable speed up in deduction too.

$$\frac{\begin{array}{c} \mathbf{a}\ P \Rightarrow P_1 \\ \mathbf{a}\ \{P_1\}C_1\{Q_1\}\ \ \mathbf{a}\ Q_1 \Rightarrow P_2 \\ \mathbf{a}\ \{P_2\}C_2\{Q_2\}\ \ \mathbf{a}\ Q_2 \Rightarrow P_3 \\ \mathrm{M} \\ \mathbf{a}\ \{P_n\}C_n\{Q_n\}\ \ \mathbf{a}\ Q_n \Rightarrow Q \end{array}}{\mathbf{a}\ \{P\}\mathrm{BEGIN\ VAR}\,V_1,\ldots,\ \mathrm{VAR}\,V_n, C_1;...;C_n\ \mathrm{END}\{Q\}}$$

Necessarily for proving programs are rules for conditional and cycles. There are two **conditionals rules** for one-armed

$$\frac{\mathbf{a}\ \{P \wedge S\}C_1\{Q\},\ \mathbf{a}\ \{P \wedge \neg S\}C_2\{Q\}}{\mathbf{a}\ \{P\}\mathrm{IF}\ S\ \mathrm{THEN}\ C_1\ \mathrm{ELSE}\ C_2\{Q\}}$$

and two-armed conditions

$$\frac{\mathbf{a}\ \{P \wedge S\}C_1\{Q\},\ \mathbf{a}\ \{P \wedge \neg S\}C_2\{Q\}}{\mathbf{a}\ \{P\}\mathrm{IF}\ S\ \mathrm{THEN}\ C_1\ \mathrm{ELSE}\ C_2\{Q\}}\ .$$

Before the rules for cycles can be defined must be defined what is an invariant of command. It is condition that is the same before and after executing if command. More precisely defined if we have partial correct specification $\mathbf{a}\ \{P \wedge S\}C\{P\}$ then P is invariant of C whenever S holds. So **WHILE-rule** says if P is invariant of body of WHILE-command then P is invariant for whole WHILE-command, whenever the test condition holds independently on numbers of executing of body. It also says that test condition is after executing of WHILE-command always false, otherwise it wouldn't have terminated. The WHILE-rule can be written following

$$\frac{\mathbf{a}\ \{P \wedge S\} C_1 \{P\}}{\mathbf{a}\ \{P\}\text{WHILE } S \text{ DO } C\{P \wedge \neg S\}}.$$

Much harder is defining the **FOR-rule**, because there it is a lot of problems. If we suppose the semantic of FOR-command that was defined in first paragraph the FOR-rule has form

$$\frac{\mathbf{a}\ \{P \wedge (E_1 \le V) \wedge (V \le E_2)\} C \{P[V+1/V]\}}{\mathbf{a}\ \{P[E_1/V] \wedge (E_1 \le E_2)\}\text{FOR } V := E_1 \text{ UNTIL } E_2 \text{ DO } C\{P[E_2/V]\}},$$

where neither *V*, nor any variable in E_1 or E_2, is assigned to in the command *C*. Useful can be the FOR-axiom for the case that $E_2 < E_1$. This axiom says that if $E_2 < E_1$ then the FOR-command has no effect $\mathbf{a}\ \{P \wedge (E_2 < E_1)\}\text{FOR } V := E_1 \text{ UNTIL } E_2 \text{ DO } C\{P\}$.

1.4. Example

On a short example will be demonstrated using of Floyd-Hoare Logic to prove program. Following code multiplies two numbers x and y and stores the result to variable S.

```
   12. a  {X=x ∧ Y=y}
    BEGIN
   11. {0=(x-X)*y}
      S:=0;
   10. {S=(x-X)*y}
      WHILE ¬(X=0) DO
      BEGIN
        VAR R;
        R:=0;
    5. {S=(x-X)*y+R}
        WHILE ¬(R=Y) DO
        BEGIN
    4. {S+1=(x-X)*y+R+1}
          S:=S+1;
    3. {S=(x-X)*y+R+1}
          R:=R+1;
    2. {S=(x-X)*y+R}
        END
    1. {S=(x-X)*y+R}
        X:=X-1;
 6.-9. {S=(x-X)*y}
      END
   10. {S=(x-X)*y ^ X=0}
    END
   13. {S=x*y}
```

At first will be proved invariant for second WHILE. Process will be following.

1. We will suppose that invariant of WHILE is S=(x-X)*y+R.
2. According to WHILE-rule the invariant of cycle is invariant for commands in body
3.-4. Application of assignment axiom
5. S+1=(x-X)*y+R+1 ⇒ S=(x-X)*y+R.

So you can see that hypothesis was correct. In addition it is clearly that R=Y after cycle terminate. In the next steps will be identified invariant of first WHILE.

6. The precondition of assignment X:=X+1 is S=(x-X)*y+R ∧ R=Y ⇒ S=(x-X)*y+Y
7. If we use and assignment axiom the postcondition will be S=(x-(X+1))*y+Y
8. S=(x-(X+1))*y+Y ⇒ S=(x-X-1)*y+Y ⇒ S=(x-X)*y-y+Y
9. Because holds Y=y then this postcondition can be simplified to form S=(x-X)*y
10. According to WHILE-rule this condition is invariant of first WHILE cycle, further holds that X=0 after this WHILE terminate.
11. Application of assignment axiom
12. X=x ∧ Y=y ⇒ 0=(x-X)*y precondition strengthening
13. S=(x-X)*y ∧ X=0 ⇒ **S=x*y**

2. The l-calculus

2.1. Introduction

The λ-calculus is a theory of functions that take functions as arguments or return functions as results. It has inspired the design of functional programming languages e.g. LISP. It was shown that the λ-calculus is universal computing system.

The λ-calculus is a notation for defining functions. All expression in the λ-calculus denotes functions and they are called λ-expressions. With this notation can be also represented data objects like number, lists etc.

There are three kinds of λ-expressions:

a) **Variables:** The functions denoted by variables are determined by what the variables are bound to in the environment. Binding is done by abstraction. We use V, $V_{1,\ \ldots}$ for arbitrary variables.
b) **Functions applications or combinations:** Application (E_1 E_2) denotes the result of applying the function denoted by E_1 to the function denoted by E_2.
c) **Abstractions:** If V is a variable and E is a λ-expression, then $\lambda V.E$ is an abstraction with bound variable V ad body E. More specifically, the abstraction $\lambda V.E$ denotes a function which takes an argument E' and transform it by rule $E[E'/V]$ (the result of substituting E' for V in E).

If we use symbol V for variables and E for λ-expressions, λ-expressions can be denoted by BNF: $E ::= V \mid (E_1\ E_2) \mid l\,V.E.$

Notes:

- Free and bound variables – an occurrence of a variable V in a λ-expression is free if it is not within the scope of a λV, otherwise it is bound.
- Applications are left associative and abstractions are right associative. So the expression $E_1\ E_2\ E_3$ means $((E_1\ E_2)\ E_3)$ and $\lambda xy.E$ means $(\lambda x(\lambda y.E))$.

2.2. Conversion rules

The process of conversing one λ-expression to other λ-expression is called λ-conversion. An example can be evaluation of numerical expression, where numbers are represented by λ-expression and result is λ-expression too. This process can be also called reduction.

There are three kinds of λ-conversions:

a) ***a*-conversion:** Any abstraction of the form $\lambda V.E$ can be converted to $\lambda V'.E[V'/V]$ provided the substitution is valid. This mean that bound variables can be renamed provided no name-clashes occur.
b) ***b*-conversion:** Any application of the form $(\lambda V.E_1)E_2$ can be converted to $E_1[E_2/V]$, provided the substitution is valid. This rule is like then evaluation of a function in programming language, where the body E_1 of the function $(\lambda V.E_1)$ with formal parameter V is evaluated with parameter E_2.
c) ***h*-conversion:** Any abstraction of the form $\lambda V.(E\ V)$ in which V has no free occurrence in E can be reduced to E.

Notes:

- The substitution is valid if and only if no free variable *E'* becomes bound in *E*[*E'/V*].
- For conversions is used following notation $E_1 \xrightarrow[a]{} E_2$, $E_1 \xrightarrow[b]{} E_2$, $E_1 \xrightarrow[h]{} E_2$ which means that E_1 is converted to E_2 by *a*, *b* or *h* conversion.
- A λ-expression to which *a*-conversion, *b*-conversion or *h*-conversion can be applied is called an *a*-redex (necessarily an abstraction), *b*-redex (necessarily an application) or *h*-redex (necessarily an abstraction).
- The conversion is called generalized, if E_2 is reduced from E_1 by α, β, or η converting any subterm.
- Conversion rules preserve the meaning of λ-expressions.

2.3. Equality of l-expression and the $\longrightarrow$ relation

Two λ-expressions are equal if they can be transformed into each other by a sequence of forwards or backwards λ-conversions. In contrast two λ-expressions are identical if they consist of the same sequence of characters. For equal expressions we use symbol = and for identical expressions ≡.

Equality is reflexive, symmetric and transitive so it is equivalence relation. Equality has property called Leibnitz's law. This means that if $E_1 = E_2$ then $E_1' = E_2'$ if E_1' contain E_1 and E_2' contain E_2.

Similarly to equality is defined the $E_1 \longrightarrow E_2$ relation. E_2 is obtained from E_1 by using a sequence of only forward λ-conversions.

2.4. Representing Things in the l-calculus

Instead functions can be in the λ-calculus represented data objects as number, list, strings, etc. In the following section will be described how can be these object defined. There is used this notation: LET ~ = λ-expression. It means that object ~ (number, …) is represented by λ-expression. For conditions is used more complicated notation: $(E \rightarrow E_1 \mid E_2)$. It means if *E* is true then result expression is E_1 and if E is false then E_2 is result.

The expressions for representing truth-values and not function have to match followed requests:

$(true \rightarrow E_1 \mid E_2) = E_1$

$(false \rightarrow E_1 \mid E_2) = E_2$

$not\, true = false$

$not\ false = true$.

For truth-values can be used these expressions.

LET true = λ *x y. x*

LET false = λ *x y. y*

If we used this notation than for not function and condition expression we can use

LET not = λ *t. t* false true

LET $(E \rightarrow E_1 \mid E_2)$ = $(E\, E_1\, E_2)$.

It is possible check up whether holds what must hold for not function.

```
not true = (λ t. t false true) true   - not definition
         = true false true            - b-conversion
         = (λ x y. x) false true      - true definition
```

= false - *b*-conversion

not false = (λ *t*. *t* false true) false - not definition

= false false true - *b*-conversion

= (λ *x y*. *y*) false true - false definition

= true - *b*-conversion

You can see that λ-expressions for truth-values and not function are useful. It is possible define other Boolean functions as and, or, etc. For example can be defined following:

LET and = λ *x y*. *x y* false

LET or = λ *x y*. *x* true *y*.

Useful can be defining pairs and tuples. Pair is data structure with two components and tuple with n components (called n-tuple).

LET (E_1, E_2) = λ *f*. *f* E_1 E_2 - pair (2-tuple)

LET (E_1, E_2, ..., E_n) = (E_1, (E_2, (...(E_{n-1}, E_n)...))) - n-tuple

If we will use previous defined truth-values we can define functions to obtain first and second element from pair:

LET fst = λ *p*. *p* true

LET snd = λ *p*. *p* false.

Extracting i component from n-tuple if i < n can be done:

LET $E \overset{i}{\downarrow} n$ = fst(snd(snd(…(snd *E*)…))), where number of snd is equal to *i*-1

and if i = n:

LET $E \overset{n}{\downarrow} n$ = snd(snd(…(snd *E*)…)), where number of snd is equal to *n*-1.

One possible way to represent numbers by λ-calculus is using Church's notation. At first we have to define $f^n x$. It means n applications of *f* to x. Number that can be defined:

LET $\underline{0}$ = λ *f x*. *x*

LET $\underline{1}$ = λ *f x*. *f x*

LET $\underline{2}$ = λ *f x*. *f*(*f x*)

LET $\underline{n}$ = λ *f x*. f^n *x*

Now we can define primitive functions for these numbers as successor, predecessor, addition and predicate if a number is zero.

LET suc = λ *n f x*. *n f*(*f x*)

LET add = λ *m n f x*. *m f*(*n f x*)

LET iszero = λ *n*. *n* (λ *x*. false) true

If these definition are correct can be demonstrated on some examples.

suc $\underline{5}$ = (λ *n f x*. *n f*(*f x*)) $\underline{5}$ - suc definition

= λ *f x*. $\underline{5}$ *f*(*f x*) - *b*-conversion

= λ *f x*. (λ *f x*. f^5 *x*) *f*(*f x*) - $\underline{5}$ definition

= λ *f x*. f^5 (*f x*) - *b*-conversion

= λ *f x*. f^6 *x* = $\underline{6}$ - Church's notation

Useful can be function identity that does not change value of number.

LET id = λ *n f x*. *n f x*

Its property can be demonstrated on example of application to number $\underline{4}$.

id $\underline{4}$ = (λ *n f x*. *n f x*) $\underline{4}$ - id definition

= λ *f x*. $\underline{4}$ *f x* - *b*-conversion

= λ *f x*. ((λ *f x*. f^4 *x*) *f x*) - $\underline{4}$ definition

= λ *f x*. f^4 *x* = $\underline{4}$ - *b*-conversion

Definition function predecessor is harder and have to be used auxiliary function.

LET prefn = $\lambda\, p.$ (false, (fst $p \rightarrow$ snd p | (f(snd p)))

LET pre = $\lambda\, n\, f\, x.$ (snd(n (prefn f)(true, x))

2.5. Definition by recursion

Sometime is possible that we want to use in definition function its name. For example if we define multiplication following

mult $m\, n$ = (iszero $m \rightarrow \underline{0}$ | add n (mult (pre m) n)).

We can't define λ-expression for this function, because we can't use in definition function that has not been yet defined. However exist one technique, which can help with this problem. First we define λ-expression **Y** that, for any expression E, has the following property:

$\mathbf{Y}\, E = E\, (\mathbf{Y}\, E)$

This property means that **Y** *E* is unchanged when the function *E* is applied to it. Functions that satisfy this property are called fixed-point operators and **Y** is one of them.

LET $\mathbf{Y} = \lambda\, f.\, (\lambda\, x.\, f(x\, x))\, (\lambda\, x.\, f(x\, x))$

On the practice you can see that Y satisfy fixed-point operator property.

$\mathbf{Y}\, E = (\lambda\, f.\, (\lambda\, x.\, f(x\, x))\, (\lambda\, x.\, f(x\, x)))\, E$

$= (\lambda\, x.\, E(x\, x))\, (\lambda\, x.\, E(x\, x))$

$= E\, ((\lambda\, x.\, E(x\, x)\, (\lambda\, x.\, E(x\, x))$

$= E\, (\mathbf{Y}\, E)$

Now it is possible to define mult function with auxiliary function and fixed point operator.

LET multfn = $\lambda\, f\, m\, n.$ (iszero $m \rightarrow \underline{0}$ | add n (f (pre m) n))

LET mult = **Y** multfn

Similarly we can in λ–calculus define function

sub $m\, n$ = (iszero $n \rightarrow m$ | sub(pre m)(pre n)) as

LET subfn = $\lambda\, f\, m\, n.$ (iszero $n \rightarrow m$ | f (pre m)(pre n))

LET sub m n = **Y** subfn.

It is possible find out correctness of this declaration so that we derivate expression using definition of fixed point operator.

sub $m\, n$ = (**Y** subfn) $m\, n$	- definition of sub
= subfn (**Y** subfn) $m\, n$	- using of fixed point **Y**
= subfn sub $m\, n$	- definition of sub
= ($\lambda\, f\, m\, n.$ (iszero $n \rightarrow m$ | f (pre m)(pre n))) sub $m\, n$	- definition of subfn
= (iszero $n \rightarrow m$ | sub (pre m)(pre n))	- β-conversion

2.6. Representing lists

List can be in λ-calculus represented as pair, where first component is truth-value that says if list is empty and second component is pair that contains first element and rest of elements in other list. So list is defined:

LET $\perp$ = $\mathbf{Y}(\lambda\, f\, x.\, f)$	- undefined value
LET [] = (true, $\perp$)	- empty list
LET [E] = (false, (E, [ ]))	- list with elemetn E
LET [E_1; E_2] = (false, (E_1, [E_2]))	
LET [E_1; ...; E_n] = (false, (E_1, [E_2; ...; E_n]))	- list with n elemetns

It is possible easy define in λ-calculus basic operations over list, which use this notation.

LET null = fst

Function null return true if the list is empty otherwise return false. It is clearly that if function null get the first element of pair that represent list it obtains this value.

LET car = λ *l*. (null *l* → ⊥ | fst(snd *l*))

Function car read the first element from the list. You can see that it is correct because it get the first element of pair that represent list values and this pair is second element of pair that represent whole list. Similarly function cdr returns tail of list so that return second element of second element of pair that represent list.

LET cdr = λ *l*. (null *l* → ⊥ | snd(snd *l*))

Function cons insert new element on the beginning of list.

LET cons = λ *x l*. (false, (*x*, *l*))

It is also possible to represent more complex functions for example function that compute length of list or joining two list into one.

LET length = λ *l*. (null *l* → $\underline{0}$ | add $\underline{1}$ (length (cdr *l*)))

LET append = λ l_1 l_2. (null l_1→ l_2 | cons (car l_1) (append (cdr l_1) l_2))

2.7. Representing the recursive functions

The recursive functions are important class of numerical functions. It was shown that every recursive function could be represented in λ-calculus.

First class of recursive functions is called **the primitive recursive functions**. Initially is necessarily defining used conceptions. The successor function *S* is defined as $S(x) = x + 1$ and projection function $U_n^i(x_1, x_2, ..., x_n) = x_i$. Functions $\underline{0}$, *S* and U_n^i are called initial functions. Next function substitution can be explained that if we have function *g* of *r* arguments and functions $h_1, \ldots, h_r$ of *n* arguments then function *f* is defined from *g* and $h_1, \ldots, h_r$ by substitution if $f(x_1,...,x_n) = g(h_1(x_1,...,x_n),...,h_r(x_1,...,x_n))$. Primitive recursion is defined from function *g* of n-1 arguments and function *h* of n+1 arguments. Function *f* is defined from *g* called base function and *h* called step function by primitive recursion if:

$$f(0, x_2,...,x_n) = g(x_2,...,x_n)$$

$$f(S(x_1), x_2,...,x_n) = h(f(x_1, x_2,...,x_n), x_1, x_2,...,x_n).$$

Function is called primitive recursive if it can be constructed from $\underline{0}$ and the successor function and projection function by a finite sequence of substitutions and primitive recursions.

Initial functions can be easy represented in λ-calculus as was shown earlier. If we use function *f* that was defined from functions *g* and *h* by substitution $f(x_1,...,x_n) = g(h_1(x_1,...,x_n),...,h_r(x_1,...,x_n))$ we can define its representation in λ-calculus as $\mathrm{f} = l(x_1,...,x_n).g(h_1(x_1,...,x_n),...,h_r(x_1,...,x_n))$. Primitive recursion can be represented using fixed point operator with following λ-expression **Y**(λ *f* x_1, …, x_n. (iszero x_1 → g(x_2, …, x_n) | h(*f*(pre x_1, x_2,…, x_n), pre x_1, x_2,…, x_n))). Every primitive recursive function can be represented by λ-calculus, because we can represent initial function $\underline{0}$, *S*, U_n^i and substitution and primitive recursion can be represented too.

Function is called **recursive** if it can be constructed from $\underline{0}$ and the successor function and projection function by a finite sequence of substitutions and primitive recursions and minimizations. Function *f* is defined by minimization from *g* (used notation is *f* = MIN(*g*)) if:

$f(x_1, x_2, \ldots, x_n)$ = 'the smallest y such that $g(y, x_2, \ldots, x_n) = x_1$' .

Function defined by minimization can be undefined for some arguments. λ-expression for minimization can have form min *x* f (x_1, …, x_n) which represents the smallest number *y* greater then *x* such that f(y, x_2, …, x_n) = x_1. It is possible write the expression with this property using fixed point operator **Y**(λ *m x f*(x_1, x_2, …, x_n). (eq(*f*(*x*, x_2, …, x_n)) x_1 → *x* | *m* (suc

x) $f(x_1, x_2, \ldots, x_n)$))). With this expression g can be rewritten into λ-calculus with following expression g = λ($x_1, x_2, \ldots, x_n$). min $\underline{0}$ f ($x_1, x_2, \ldots, x_n$). Recursive functions can be represented in λ-calculus because primitive recursive functions can be represented in λ-calculus and minimization can be represented too.

A partial function is function that is not defined for all arguments. An example can be dividing. If partial function can be constructed from $\underline{0}$ and the successor function and projection function by a sequence of substitutions and primitive recursions and minimizations, it is called **partial recursive function**. A partial function f can be represented by a λ-expression $\underline{f}(\underline{x}_1, \ldots, \underline{x}_n)=\underline{y}$ if $f(x_1, \ldots, x_n)=y$ and if $f(x_1, \ldots, x_n)$ is undefined then $\underline{f}(\underline{x}_1, \ldots, \underline{x}_n)$ has no normal form. So partial recursive function can be represented in λ-calculus.

3. Conclusion

This article recalls groundwork for programming language theory. There was defined a little programming language and introduced Floyd-Hoare Logic as an instrument for proving program in this language. Definitely it is also possible to use Floyd-Hoare Logic for proving other program languages but it is necessary to adapt used rules for commands of these languages. Proving programs correct is useful because it is important not only for life critical system to work correct. Next there was introduced λ-calculus as one possible way to represent universal computing system. There was shown its notation and operations with λ-expressions. Also there was demonstrated how it is possible to represent various data objects and how can be defined functions by recursion. At the end there was shown which classes of function can be represented in λ-calculus.

Bibliography

Gordon, M.J.C., *Programming Language Theory and its Implementation*