ICOT Technical Report: TM-0029

TM-0029

Higher Order: Its Implications to Programming Languages and Computational Models

by

T. Ida, M. Sato, S. Hayashi,
M. Hagiya, T. Kurokawa
T. Hikita, K. Futatsugi,
K. Sakai, T. Toyama, T. Matsuda

October, 1983

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Higher order: its implications to programming languages and computational models

Contents

1. Introduction

T. Ida

This report constitutes a summary of discussions of the meetings of WG 5 of Fifth Generation Computer System Project, which were held in May 19 and 23, 1983.

The importance of clarifying the notion of "higher-order" used in various fields of mathematics and computer science is pointed out by K. Hirose, chief investigator of WG 5, in pursuing the goals of the FGCS project. These goals, when broken down and presented as a task of WG 5 take the form of investigating (i) the usage of the notion of "higher-order" in mathematical systems, and (ii) the ways that the notion is applied in the design of programming languages and computer systems.

Although the notion of "higher order" is intuitively clear to many mathematicians and computer scientists, there seems to be differences in perceptions, if not wide. Therefore we start by defining the meaning of "higher-order". In section 2, the general usage of term "higher-order" in mathematics is explained and the notion seems to be fairly well understood among the members of WG 5, at least the usage of the term in this report. Then interpretation of the notion of "higher-order" is given in various theories such as Λ calculus and the constructive set theory.

The notion of "higher-order" in programming languages is rather extensive. Our attempt here is to collect examples from various programming paradigms which are of prime importance to FGCS project. Section 3 is the accumulation of such examples both in logic, functional programming and software specifications. Although the examples are not exhaustive, they serve as the illustration in making the efforts of understanding, as a whole, the objects of computing systems in general, such as what we call rather loosely programs, data, objects etc..

From the beginning of the discussions, some members of WG 5 expressed the hope that the understanding of logic and functional programming methodologies will be gained on the common basis, or altogether, to say the least, and sought to bridge the gap, if exist, between the two streams of researches.

The article in section 4 is the first step towards such a goal. In this section an attempt is made to classify the notions now prevailing among FGCS researchers such as reduction, resolution, unification and so on. The explanation is brief as the fuller exploration of this subject is under way. It is hoped that the classifica tion will guide the directions towards which researches in the related fields will progress.

The notion of "higher-order" in mathematics leads us to proof systems, since the proof systems treat mathematical (sub-) systems as an object of study. Section 5 covers the present state of the art in the development of systems whose implementations are being attempted or are consummated. A discussion on Martin-Löf's constructive set theory is included in section 6, as it is an attempt to view notions of mathematics and computer science such as set, proof, program and problem in a unified way.

In section 7 we describe the incorporation of the notion of "higher-order" in combinatory systems. How the notion of "higher-order" in programming languages is transformed to "higher-order" in these computational systems is described. In section 8 an approach is described to extend term rewriting systems to the typed systems.

Finally, unification is briefly discussed from the view point "higher-order". Topics on unification in general will be treated in a separate report.

2. Notion of higher order

Susumu HAYASHI

Introduction

The notion of "higher order" is very popular in modern mathematics, although its explicit introduction into mathematics is not very new. Various higher-order formal systems are precisely defined in the area of logic, but it seems that there is no general definition of the notion. Hence, it may be worthwhile and necessary to clarify the notion from the point of view of programming languages, because higher-order notions will play importants role both theoretically and practically, not only in mathematics but programming languages in the future. As the actual aspects of higher-order notions in certain programming languages will be discussed extensively in the following sections, we will restrict ourselves here a general discussion on the subject.

Higher-order notions in mathematics

At the base level, predicates describe properties of objects in the universe of discourse, and functions describe mappings of objects to objects. Higher-order predicates, then describe properties of such predicates and functions, and higher-order functions describes mappings of such predicates and functions to objects or functions, or other higher-order objects. The effect of the introduction of higher-order concept is to elevate our position in the universe of discourse in the upward direction, and we may continue this process of elevation indefinetely. To clarify our point of view, let us explain the terminology of a <u>one-step elevation of order</u>. The forming of power set of a set is a typical example of a one-step elevation of order. By <u>iteration</u> of this one-step elevation we arrive at such types as $P^n(A)=P...P(A)$. Similarly, we arrive at such types as $A^{A^{\cdot^{\cdot^{\cdot A}}}}$ by <u>iteration</u> of the one-step elevation forming B^A. In the usual formulation of higher-order theory using <u>arbitrary</u> functions and <u>arbitrary</u> sets, the type of resulting higher order objects <u>must</u> be distinguished from the type of objects of the lower orders, because the higher levels have many more elements. However, it is sometimes possible to "identify" higher types with lower types, provided the meaning of the notion of "function" is <u>not</u> the same as that of current mathematics. For example, in λ-calculus all objects may be thought as functions at the same level. In this case, the one step elevation of order does <u>not</u> result in a new type, because the notion of function in λ-calculus is different from the notion of function in current mathematics. Scott's mathematical models of λ-calculus exist in the category of continuous lattices and continuous functions, so functions in the models are restricted to be <u>continuous</u> and they can be reflexive. Dana Scott stresses "type-free" systems are special kinds of typed systems and significance of types (cf. Scott[5,6]). A model of λ-calculus is a typed system with morphisms i and j between types D and D -> D such that

$$D \underset{i}{\overset{j}{\rightleftarrows}} (D \to D), \quad j \circ i = id \quad \text{(cf. Koymans[4])}.$$

Each object of D, say a, represents a function j(a), on the other hand, each function of D -> D, say f, can be represented by an object i(f). In this way, the elements of D can be regarded as both "objects" and "functions", and D looks like "type-free". However, "type-free" systems should be distinguished from the other typed systems as a <u>special</u> kind of typed systems. In this respect, a system of functions in which a "function" is applicable to itself will be said <u>self-applicative</u>. In a self-applicable system of functions, we may identify higher-order objects with some lower-order objects. So there is no rigid distinction between "functions" and "objects" in such a system. On the other hand, systems of functions in which higher-order objects must be distinguished from lower-order objects will be said <u>well-typed</u>.

<u>Intensional versus extensional</u>

The introduction of higher-order leads to the problem of "intensional vs. extensional". In this subsection, we will discuss this problem. Let σ be a type of "concrete" objects such as integers, finite strings of characters etc.. Then it is clear what means "two objects a and b of type σ are equal". Assume that f and g be functions from σ to σ , we say f and g are extensionally equal, notation $f =_{ext} g$, iff f(x)=g(x) holds for every object x of type σ. We call a function F of type $(\sigma\text{->}\sigma)\text{->}\sigma$ extensional iff F(f)=F(g) for every f, g such that $f =_{ext} g$. (This is a definition of extensionality for a special type $(\sigma\text{->}\sigma)\text{->}\sigma$, and extensionality of functions for other types can be defined similarly.) In current mathematics based on set theory, all functions are assumed to be <u>extensional</u>, in another word, a "function" is identified with its "extension" (graph of the function). But, in the programming languages, a "function" is often identified with "intension" (algorithm or content of program or program itself). E.g., we might say "a function f is extensionally equal to a function g, but f is quicker than g". Let us explain extensionality in programming languages by some examples. MAPCAR of LISP is extensional with respect to its functional argument, because it uses only some pairs of input and output of the functional argument in computation. On the other hand, if the function EQUAL of LISP takes into account an equality between functions, then it is intensional, because equality implied by EQUAL can only be checked by examining the definition (intension) of functions. Similarly, Church's combinator δ (cf. Barentregt [1]) is intensional. In LISP, a functions is an algorithm represented by a code (an S-expression) and such a code is available to users at any level, so LISP is a typical intensional language. On the other hand, ML of Edinburgh LCF is an extensional language. (Of course, you can imagine a version of ML in which intensional equailty on higher types is availabe, then ML turns to be non-extensional. It seems that there is no clear attitude to the problem in Gordon et. al [2].)

Intensional functionals seem to be seldom used in actual programs. Extensional systems will be much better than intensional systems in some mathematical aspects. (E.g., it will provide for an algebra of functions (programs). Bohm [2] gives an interesting foundation of FP by using a system of extensional combinatory algebra. In his work, extensionality (including η-rule) plays an important role to simplify the axioms of the system and proofs of equivalences of programs.) However, it may not be the right way to neglect intensional aspects of functions completely in general purpose programming languages. A solution might be an intensional language with a subset which is purely extensional.

Higher-order notions in programming languages

Contrary to current mathematics, "functions" in the usual programmong languages are restricted to be "computable" or "definable", so that they may be coded by appropriate data of lower types. Hence distinction of higher types from lower types may not be desirable. A typical example is LISP. In LISP, "functions" are partial recursive functions on S-expressions, hence they can be coded by S-expressions as is well known. Generally speaking, self-application is not unnatural in programming languages. This is contrary to current mathematics. However, this does not mean that distinctions of higher types and lower types are meaningless in programming languages. Typing enables us to classify objects in the universe of discourse. (Some designers and users of typed languages stress that type assignment is useful for debugging.) And, it is sometimes useful to forget that a "function" is given as intension (definition). By supposing a "function" as an "abstract" object, we can suppose a "function" in programming languages as an actual function in mathematics, that is, it may be thought as extension. And, by such an abstraction, we can grasp "mathematical" contents of an algorithm. Then abstract types of "functions" must be distinguished from types of "concrete" lower objects such as integers, (finite) strings of concrete objects etc.. In this respect too, "well-typed" programming languages are meaningful. A typical example of well-typed programming language is ML of Edinburgh LCF.

So even in programming languages, there are (at least) two kinds of higher-order systems as in mathematics, although the portions of these two kinds of higher-order structure in the general higher-order structure are very different between programming languages and mathematics. For including higher-order notions in future programming languages, it will be useful to note the distinctions between "well-typed" systems and "self-applicative" systems.

Acknowledgment

The author of this section would like to express his hearty thanks to Prof. Dana Scott and Prof. Michael Beeson for their criticism on the author's opinion on higher-order notions. Especially, Prof. Scott's suggestions made clear the author's thought on higher-order notions.

REFERENCES

[1] Barentregt,H.P., The Lambda Calculus, its syntax and semantics, 1981, North-Holland, Amsterdam.

[2] Bohm,C., Combinatory Foundation of Functional Programming, 1982.

[3] Gordon,M.J.,Milner,A.J.& Wadsworth,C.P., Edinburgh LCF, Lecture Notes in Computer Science 78, 1979.

[4] Koymans,C.P.J., Models of the Lambda Calculus, Information and Control 52, 306-332 (1982).

[5] Scott,D.S., Lambda calculus; some models, some philosophy, in The Kleene Symposium, (Eds. J.Barwise, H.J.Keisler & K.Kunen), North-Holland, Amsterdam, 1980, pp. 223-265.

[6] Scott,D.S., Relating Theories of the λ-calculus,in To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, (Eds. J.P.Seldin & J.R.Hindley), Academic Press, 198[illegible], pp.403-450.

3. Programming experiences with higher-order notion

Here, we want to collect examples of higher-order notion from various programming paradigms: logic programming, functional programming, and software specification.

3.1 Logic programming languages

Toshiaki Kurokawa

In theory, the universe of logic programming is the universe of terms, i.e. Herbrand universe. However, this comes from the logical interpretation of logic programming.

In practice, i.e. from the procedural interpretation of logic programming, we handle variables, functions, and programs (i.e. logical formula), which are not included in the narrow sense of the Herbrand universe.

So, in general, "higher-order" notions in logic programming fall into the following categories: checking the variable if it is not instantiated, handling programs as data, additon/deletion of programs, using the set of terms to interface logical formula and function, and applying function to data.

In the broader sense of "higher-order", we can include the control of the logic programming, and the explanation of how the system executes the program. LOGLISP includes this kind of "higher-order" facilities.

We will describe the usage of "higher-order" notion in three logic programming languages: DEC-10 PROLOG, LOGLISP, and PARLOG.

DEC-10 PROLOG provides the typical features of the so-called PROLOG. LOGLISP and PARLOG cast some insights on other (and experimental) features to incorporate the "higher-order" facilities. It is not, however, our intention to limit the "higher-order" notions in the points listed below.

DEC-10 PROLOG [Bowen 81]

DEC-10 PROLOG has mainly two kinds of higher order facilities: handling programs as data and handling set of terms.

a) checking the variable if it is not instantiated

o DEC-10 PROLOG can check the variable if it is already instantiated or not. The built-in predicate is 'var'.

```
Example. var(X)   ---> success if X is not instantiated, fail otherwise.
         var(a)   ---> fail.
         var(13)  ---> fail.
```

b) handling programs as data

o DEC-10 PROLOG can convert the clause to the list, and also the list to the clause. The built-in procedure (also called as evaluable predicate) is '=..'. Note that this is the bi-directional conversion.

```
Example. product(0,N,N-1) =.. [product,0,N,N-1]
         N-1 =.. [-,N,1]
         product =.. [product]
```

o DEC-10 PROLOG can execute the body of the clause using the built-in procedure, 'call'. The body must be in the form of the term.

Example. call(append([a,b], [c,d], Y))

o DEC-10 PROLOG has the database of programs (i.e. clauses). There are following procedures to access and to modify the database:

```
clause  -- get the clause having the designated head.
assert  -- add the clause.
asserta -- add the clause at top.
assertz -- add the clause at bottom.
retract -- delete the clause.
abolish -- delete the clauses having the principal functor with the arities.
```

c) handling set of terms

DEC-10 PROLOG has procedures to make a set of terms.

```
o setof(X,P,S) -- unify S with the set of all instances of X such that P is
                  provable, where that set is non-empty.  S will be the list
                  with the elements ordered.

o bagof(X,P,S) -- same as setof except that the instances are not ordered and
                  the same instances are not eliminated.
```

LOGLISP [Robinson 80, 81a, 81b, 82a, 82b]

LOGLISP is a language implemented on top of Lisp by J. Robinson and E. Sibert.

a) LOGLISP can handle the set, which interfaces the logic part and the Lisp part of the system.

```
(ALL X C1 ... Cn) -- returns a list of X which satisfy C1 ... Cn.
(ANY K X C1 ... Cn) -- returns K instances of X.
(THE X C1 ... Cn)   -- returns the sole instance of X.
```

b) LOGLISP has a deduction control mechanism.

o Controlling the deduction tree size through its depth and length.
o Selection between a Prolog-like LUSH resolution mode and the heuristic search mode.

c) LOGLISP has a deduction explaining facility.

```
(EXPLAIN N1 ... Nk) -- explains the N1st, ..., Nkth answers.
```

PARLOG [Clark 83]

PARLOG is an experimental language now underway at Imperial College, University of London.

a) PARLOG has the following main features:

Including both the and-parallel and the or-parallel. (They call it don't care non-determinism and don't know non-determinism, respectively.)

The relation declared as and-parallel (default case) are executed concurrently. The information are handed using the input-output mode declaration and the ^-producer annotation.

```
Example. relation c(?, ^)
         c(w,x) :- (p(w,x)&q(x,y)), (r(y^)&s(y)).
```

The relation declared as or-parallel enables the concurrent search which is used as database query.

PARLOG includes functions. The set of PARLOG interfaces between functional and relational execution.

b) PARLOG has four types of the top level execution:

: <relational expression>
This is almost same as DEC-10 PROLOG. There will be no output, however, unless the output is explicitly programmed in the <relational expression>.

<term> : <relational expression>
This is almost same as DEC-10 PROLOG. Final output is the value of the <term> which satisfies the <relational expression>.

<term>
This is the functional execution of the <term>.

{ <term> : <sequential expression> }
This will return the set of <term> which satisfies the sequential expression.

```
Example. {<x,y>: parent(x) & y={z : child-of(*x, z) }
         where *x denotes the global variable.
```

c) PARLOG has also a special execution feature, i.e. the lazy evaluation.

```
Example. :printlist(primes!())
         where ! is the annotation for the lazy evaluation.

         s = { T : L & SE }!
         where T is a term, L is a literall and SE is a sequential
         expression. It means each time the set element is accessed,
         a term is selected which satisfies L and SE.
```

d) As for the higher order facilities, except for the set facilities, PARLOG has the function application and the adding and deleting the clause.

o function application -- using @

```
Example. function f_maplist(?, ?)
         f_maplist(f, []) = []
         f_maplist(f, [u|x]) = [f@(u)|f_maplist(f,x)]
```

o adding or deleting the clause to the module.
addcl -- add clause
delcl -- delete clause
Using these procedures, the user must explicitly denote the module where the clauses belong.

```
Example. : addreln(Personnel, 'salary') &
                addcl(/salary(Smith,8560)\) &
                addcl(/salary(Jones,9800)\) &
                addcl(/salary(Brown,10150)\) &
                addcl(/salary(Green,12010)\).

         : delcl('employee', 2).
         : delcl(/employee(D1,Jones)\).
```

REFERENCES

[Bowen 81] Bowen, D.L. "DECsystem-10 PROLOG USER'S MANUAL", Version 3.43, Edinburgh Univ. (Dec. 1981)

[Robinson 80] Robinson, J. and Sibert, E.E. "Logic Programming in Lisp", Syracuse Univ., (Dec. 1980)

[Robinson 81a] Robinson, J. and Sibert, E.E. "LOGLISP Implementation Notes", Syracuse Univ., 22, (Dec. 1981)

[Robinson 81b] Robinson, J.A., E.E. Sibert, "THE LOGLISP USER'S MANUAL", Syracuse University, (Dec. 1981)

[Robinson 82] Robinson, J.A. and Sibert, E.E. "LOGLISP: MOTIVATION, DESIGN AND IMPLEMENTATION" in Clark,K.L.and S.A.Tarnlund eds.,Logic programming, Academic press, 299-313 (1982)

[Robinson 82b] Robinson, J.A. and Sibert, E.E. "LOGLISP: an alternative to PROLOG" in Hayes, J.E., Michie, D. and Pao, Y-H, (eds.) "Machine Intelligence 10", 399-419, Ellis Horwood, (1982)

[Clark 83] Clark, K.L. and Gregory, S. "PARLOG: A PARALLEL LOGIC PROGRAMMING LANGUAGE", RR DOC 39, (Mar. 1983)

3.2 Functional programming

T. Ida

We will describe the usage of the notion of higher-order in functional programming languages FP and KRC (and its predecessor SASL). The choice of the two languages reflects our recognition that there exist two distinct views on higher-order functions embedded in the design philosophy of the two functional programming languages. In this section 'higher-order' is exclusively applied to functions.

FP

FP is a functional programming language advocated by J. Backus[1]. The universe of FP may be viewed as consisting of two distinct universes of $\mathcal{O}$ (for object) and $\mathcal{F}$ (for function). $\mathcal{F}$ is a set of functions which operate on objects and deliver an object as a result of application of a function to an object. FP is thus a typed system in that its universe consists of two separate (sub-)universes $\mathcal{O}$ and $\mathcal{F}$.
$\mathcal{O}$ is inductively defined as follows:
$x \in \mathcal{O}$ if x is an atom
$<> \in \mathcal{O}$ ($<>$ is called *nil*)
$\perp \in \mathcal{O}$ ($\perp$ is called bottom)
$<x_1, x_2, \ldots, x_n> \in \mathcal{O}$ if $x_i \in \mathcal{O}$ for $i=1,2,\ldots,n$
($< , \ldots, >$ is called a sequence.)
$< >$ in $<x_1, x_2, \ldots, x_n>$ can be regarded as a sequence forming operator. The universe $\mathcal{O}$ is closed with respect to the operator $<>$.

$\mathcal{F}$ is a set of functions $f_1, f_2, \ldots f_n, \ldots$ Corresponding to the sequence forming operator, a set of operators called PFO's (Programming Forming Operator) are introduced. They are used to combine functions and to create new functions. For example, '$\cdot$', '$[\,]$', '$\rightarrow\ ;$' are PFO's. We can construct a self-applicative universe using PFO's from primitive (or may be called atomic) functions. Similar to $\mathcal{O}$, $\mathcal{F}$ is inductively defined as follows:
$f \in \mathcal{F}$ if f is a primitive function.

$f_1 \cdot f_2 \in \mathfrak{F}$ (composition) for f_1, $f_2 \in \mathfrak{F}$

$[f_1, f_2, \ldots, f_n] \in \mathfrak{F}$ (construction) for $f_1, f_2, \ldots, f_n \in \mathfrak{F}$

$(f_1 \rightarrow f_2; f_3) \in \mathfrak{F}$ (conditional) for f_1, f_2, $f_3 \in \mathfrak{F}$

and so on.

$\mathfrak{F}$ and $\mathfrak{O}$ are connected via apply (:). $f{:}x$ denotes the result of applying f to x, where $f \in \mathfrak{F}$, $x \in \mathfrak{O}$, hence $f{:}x \in \mathfrak{O}$. Backus's intention here is to construct FP-algebra $(\mathfrak{F}_0, \wp)$ where $\mathfrak{F}_0$ is a set of primitive functions and $\wp$ is a set of PFO's.

PFO's can also be regarded as a function taking functions as arguments in the following way.

Example:

(A) $f_1 \cdot f_2$ => $<comp, f_1, f_2>$

(B) $<x_1, x_2>$ => $apndl{:}<x_1, apndl{:}<x_2, nil>>$

"$\cdot$" on the left hand side of (A) is a notation when viewed as an operator in FP-algebra and "*comp*" in the right hand side is a notation viewed as a functional. Similar analogy can be observed in the case of universe $\mathfrak{O}$ (see (B)). "$\cdot$" may not be considered as a higher-order function in FP-algebra, but the corresponding "*comp*" in the right hand side of (A) can be regarded as a higher-order function in a system which defines operational semantics (Backus called this system $\overline{\text{FFP}}$). Similar arguments hold for conditional, construction and so on.

"*bu*" (binary to unary) is an exception to the above explanation. "*bu*" is a PFO which converts a binary function to a unary function by supplying a constant. For example, $add1 = (bu, plus, 1)$. If we adhere to the argument that FP should be considered as a typed system in which PFO should take only functions as arguments, "*bu*" should be defined as taking two functions as arguments, i.e. (bu, f, g) where f is a binary function and g is a constant function, such as $\overline{1}$ in $(bu, plus, \overline{1})$.

From the view point of programming, we perceive the following hierarchy;

object --> function --> PFO.

Hence, in this sense PFO is higher-order.

KRC (Kent Recursive Calculator)

KRC is a functional programming language designed by D. Turner [3]. KRC is a type-free system and is based on a combinatory system (SASL[2] in particular). Therefore, functions are also objects. Let O be the universe of KRC and O_0 be the subset of O, consisting of atomic objects.

Following illustrates the notion of higher-order in this type-free system:
Examples $f_1 : O_0 \rightarrow O_0$ for function f_1 (unary),
and higher-order functions f_2 and f_3:
$f_2 : [\, O_0 \rightarrow O_0 \,] \rightarrow O_0$, or
$f_3 : [\, O_0 \rightarrow O_0 \,] \rightarrow [\, O_0 \rightarrow O_0 \,]$,
etc.
and $f_1, f_2, f_3 \in O_0$.
For the function of arity n, we have

$f_4 : [O_{(1)} \rightarrow [O_{(2)} \rightarrow [\; .. [O_{(n-1)} \rightarrow O_{(n)}] \, ..]$

Curried functions such as

$f_5 : [O_{(2)} \rightarrow [\, .. [O_{(n-1)} \rightarrow O_{(n)}] \, ..]$ where $x \in$ O and $f_4 x \in$ O can be introduced naturally.

A type-free system such as KRC can express certain algorithms more naturally than FP. It can be argued, however, that the algebraic structure of O would become more complex than that of FP.

The following example illustrates the use of higher-order functions in KRC:
intersection [*x*] = *x*
intersection (*x*:*y*) = *filter* (*member x*) (*intersection y*)
"*intersection*" is a function to take an intersection of lists,
e.g. *intersection* [[1,2] , [5,2] , [2,3,4]] = [2] . Here "*filter*" is a higher-order function which takes as arguments a predicate and a list. "*member x*" is a Curried function which tests whether x is in the given list. This construction is not possible in original FP since it cannot handle a Curried function such as "*member*

x" above. In FP, a higher-order function "*filter*" cannot naturally be defined at FP-level. "*filter*" should either be provided as a primitive PFO or be defined in FFP system [1]. The limitations of FP may be remedied by Böhm's combinatory based system [4]. Another approach is reported in [5] to improve the power of FP by a modest extension.

Comparison of the two systems

It would be interesting to view the two systems in terms of the combinatory system. In KRC, conditional expression "if p then q else r" usually given in the form of guard in the equation) may be translated into **S** (**S** (**B** *cond* p) q) r, where **S** and **B** are combinators. Functions *cond*, p, q, and r are Curried.
In FP, the form equivalent to the above is $p \rightarrow q; r$. This form is translated into FFP form <*cond*, p, q, r>. In FP, p, q, r cannot be variables. That is, functions should be specified at the time of definition and furthermore "*cond*" requires full arguments at the time of definition.

References

[1] J. Backus. Can programming be liberated from von Neuman style?
Functional style and its algebra of programs. CACM 21(8), 613 641, Aug. 1978

[2] D. Turner A new implementation technique for applicative
languages, Software practice and experiences, Vol. 9, 1979

[3] D. Turner The semantic elegance of applicative languages
Proc. 1981 ACM Conf. on Functional programming languages and computer architecture, Oct. 18-22, 1981, Portsmouth, New Hampshire

[4] C. Böhm, Combinatory foundation of functional programming,
Conf. Record of ACM symposium on LISP and functional programming languages
Aug. 1982

[5] T. Ida, Some FP algebra with Currying operations
to appear in Information Processing Letters

3.3 Software Specification

Kokichi Futatsugi

Higher order concepts in specification/programming would be summarized as follows.

Meta Function(MF): the ability to manipulate specifications/programs texts themselves as data. LISP's S-expression is the most notable vehicle of this function.

Higher Order Operators(HOO): the feasibility of higher order functions/predicates in writing specifications/programs. The systems based on the combinatory systems (e.g. FP, KPC, etc., see 3.2) are considered to be most promising in this respect.

Modularization/Parameterization(M/P): the ability to give specifications/programs the "good" structures. The "good" structures should imply the "open-endedness" [Abrial80] (for defining new entity) of the resulting specifications/programs. Object Orientedness in Smalltalk and Frame Structures in AI languages are typical as vehicles of M/P. M/P would be broken down to the following two features [Abrial80].

Generic Definition(GD), whereby certain statements can be parameterized in such a way that the same "schema" may subsequently be instantiated in a variety of different contexts.

Abstract Structures/Named Classes(AS/NC), which permit the definition of particular concepts to be "encapsulated" so that the corresponding properties may be applied in constructing other entities.

In the following we briefly survey how the higher order concepts have been treated in formal specification languages with respect to the above mentioned points. The specification languages considered are, Z, Clear, and HISP. The survey will be done mainly around the M/P (GD and AS/NC) for these are crucial properties in software specification.

I. Z [Abrial 79,80]

Z is a formal specification language based on set theory.

I-1. MF in Z

There is no Meta Function in Z. Z is not running on machines and the designers of Z do not seem to be aware of the necessity of MF.

I-2. HOO in Z

Z is based on set theory and functions/relations(predicates) are all considered as sets which satisfies certain conditions. In Z, one can define power sets as well as products and sums (unions) of sets. As a result, one can freely uses the higher order functions and relations in Z.

I-3. GD and AS/NC in Z

One of the main design aims of Z is to provide "open-ended formal framework for defining new theories". For this purpose Z provides two constructs which correspond to GD and AS/NC. As a result one can easily achieve M/P in Z.

[Examples]

(a) Generic Definition:

The set union operator op(U) is defined generically as follows in Z.

```
op(U)[X] = func S1,S2 -> S3 for
             S1,S2,S3 : subset(X)
           /* op(U) is a function from S1 x S2 to S3
              where S1,S2,S3 are subsets of of X  */
           then
             S3 = set x for x : X where
                    x e S1 or x e S2
                    /* x is a element of S1 or S2 */
                  end
           end;
```

This definition is generic because the X in the definition stands for any set. X can be associated to any kind of set afterwards.

(b) Abstract Structures/Named Classes:

In Z, to define "monoid", one can firstly define a class "semi-group", and then define "monoid" as a subclass of the "semi-group".

```
semi-group [S] = class
                   oper : S x S -> S
                 where
                   oper o (oper prod ident[S])
                          = oper o (ident[S] prod oper)
                   /* oper is associative */
                 end;

monoid [S] = subclass semi-group [S] class
               u : S
             where
               oper o (const(S,u) & ident[S]) = ident[S];
               oper o (ident[S] & const(S,u)) = ident[S]
               /* u is the unit of monoid */
             end;
```

Note also that the definitions of "semi-group" and "monoid" themselves be generic with respect to set S.

II. Clear [Burstall 77,80,81]

Clear is a formal specification language based on algebraic specification techniques and its semantics has been defined using category theory.

II-1. MF in Clear

There seems to be no Meta Function in Clear.

II-2. HOO in Clear

One can not use the Higher Order Operators in Clear. All operators in Clear are restricted to first order. That is, the operator in Clear is declared as,

```
appendText: Text, File -> File
```

and the declaration like,

```
mapCar: List, (Element -> Element) -> List
```

is not permitted.

II-3. GD and AS/NC in Clear

In Clear, GD and AS/NC are supported by the parameterized theories(theory procedures) and enrichment operations respectively.

[Examples]

(a) GD with Parameterized Theories:

The generic (parameterized) theory which represents "sorting" on general partial ordered set (poset) would be written as follows in Clear.

```
procedure Sorting(P: Poset) = ...
```

"Sorting" is defined to be theory procedure, and so one can easily define "sorting" on any kind of "poset" by specifying a specific "poset" as follows.

```
Sorting(Natleq[element is nat, <= is leq])
```

(b) AS/NC with Enrichment:

In Clear class-subclass relation could be realized by enriching a theory (class) and geting a more detailed another theory (subclass). For example, the theory Rat (rational number) could be defined by enriching the theory Interger as follows.

```
const Rat =  /* Rat is a constant theory,
                not a theory procedure */
   Integer enriched by
   sorts rat
   opns  0,1: rat
         +,-,*,/: rat,rat -> rat
         rational: int -> rat
     ....
   enden
```

III. HISP [Futatsugi 80,82a,82b]

HISP is a specification language/system based on algebraic specification techniques and term rewriting systems.

III-1. MF in HISP

HISP system provide an operation for transforming the specification texts in HISP into the HISP's basic data structures (i.e. terms). HISP has MF in this respect.

III-2. HOO in HISP

One could not use the Higher Order Operators in HISP. Presently, however, higher order operators have been implemented into HISP and the declaration like,

```
   mapCar: List, (Element -> Element) -> List
```

is now permitted in HISP.

III-3. GD and AS/NC in HISP

In HISP, any software system are embodied as hierarchical structures of modules, and these structures are constructed using predefined module constructing operations. GD and AS/NC are supported by the HISP's substitution and refinement operations respectively.

[Examples]

(a) GD with Substitution Operation:

HISP provides the substitution operation for replacing base-module(s) of any module with another module(s). And so, all HISP's modules are generic in this sense. For example, "sorting" theory in II-3 can be written as follows in HISP.

```
POSET ::
 ...
  sort Element
  op   _<=_: Element,Element -> Bool
  ...
 end

NATLEQ ::
 ...
  sort Nat
  op   _leq_: Nat,Nat -> Bool
  ...
 end

SORTING ::
 create
  sub  POSET
       /* POSET is one of the base modules of SORTING */
   ...
 end

SORTINGonNAT ::
  SORTING (* POSET <- NATLEQ; Element <- Nat; <= <- leq *)
       /* (* ... *) is a HISP's substitution construct */
```

(b) AS/NC with Refinement Operation:

In HISP, class-subclass relation could be realized by refinement operation in HISP. HISP's refinement operation is almost the same as Clear's enrichment operation. For example, the rational number could be defined as a subclass of integer using HISP's refinement operation as follows (cf. II-3).

```
RAT ::
 refine INT/*INTeger*/
  sort Rat
  op   0,1: Rat
       _+_: Rat,Rat -> Rat
       _-_: Rat,Rat -> Rat
       _*_: Rat,Rat -> Rat
       _/_: Rat,Rat -> Rat
       rational: Int -> Rat
       ....
 end
```

REFERENCES

Z

[Abrial 79] J.R.Abrial, S.A.Schuman, and B.Meyer: Specification language Z, Proc. of Summer School on Program Construction, Belfast, Sep. 1979; Cambridge Univ. Press,

[Abrial 80] J.R.Abrial: The specification language Z: syntax and "semantics", Oxford Univ. Computing Lab., Programming Research Group, (Apr. 1980)

Clear

[Burstall 77] R.M.Burstall and J.A.Goguen: Putting theories together to make specifications, Proc. of Fifth IJCAI, pp.1045-1058, Carnegie-Mellon Univ., Pittsburgh, (1977).

[Burstall 80] R.M.Burstall and J.A.Goguen: The semantics of CLEAR, a specification language, Proc. of advanced course on abstract software specifications, Lecture Note in Comp. Sci. 86, Springer-Verlag, Berlin, (1980).

[Burstall 81] R.M.Burstall and J.A.Goguen: An informal introduction to specifications using Clear, in "The correctness problem in computer science", Eds. R.S.Boyer and J.S.Moore, pp.185-213, Academic Press,(1981).

HISP

[Futatsugi 80] K.Futatsugi and K.Okada: Specification writing as construction of hierarchically structured clusters of operators, Proc. IFIP Congress 80, pp.287-292, North Holland, (Oct.1980)

[Futatsugi 82a] K.Futatsugi: Hierarchical software development in HISP, "Computer Science & Technologies 1982", Ed. Kitagawa T., Japan Annual Reviews in Electronics, Computers & Telecommunications Series, OHMSHA/North Holland, pp.151-174 (1982)

[Futatsugi 82b] K.Futatsugi and K.Okada: A hierarchical structuring method for functional software systems, Proc. 6th International Conf.Software Eng., pp.393-402 IEEE Press, (1982)

4. Reduction and Resolution

Yoshihito TOYAMA

We will consider substitution operations appearing in various computation models. It is shown that, from the viewpoint of the substitution operations, computation models are classified into four types: Formal Language type, Data Base type, Reduction type, and Resolution type.

First, we explain our computation model. The computation model is defined by C=<O,R>, where O is the set of objects and R is the set of rewriting rules (i.e., computation rules). In this model, the computation is defined by a binary relation $\rightarrow$ on the set of objects. This relation is determined by using the rewriting rule set R and substitutions. For example, we consider a reduction system having the following rule,

$$f(x) \triangleright if(zero(x),1,x*f(x-1)).$$

Let f(3)+1 be a object. Then we first take an instance of the rule, $f(3) \triangleright if(zero(3),1,3*f(3))$, by using the substitution [x:=3]. Next, we replace the term f(3) occurring in the object f(3)+1 by the right hand side of the rule instance. Thus, we can get a computation,

$$f(3)+1 \rightarrow if(zero(3),1,3*f(3-1))+1.$$

Continuing the above process, we finally get the following computation:

$$f(3)+1 \rightarrow if(zero(3),1,3*f(3-1))+1 \overset{*}{\rightarrow} 6+1 \rightarrow 7,$$

where $\overset{*}{\rightarrow}$ is the transitive reflexive closure of $\rightarrow$. The above example shows that, in reduction systems, the computation $\rightarrow$ is defined by the following way. Let M be an object term and P be a subterm of M, denoted by $M\equiv\ldots P\ldots$ where $\equiv$ is syntactical equality. If there are a rule $A \triangleright B$ in R, and a substitution θ such that $A\theta \equiv P$, then the occurrence P in M, i.e., $A\theta$, is replaced by $B\theta$. Thus, we obtain the following computation from M to N:

$$M\equiv\ldots A\theta\ldots \quad \rightarrow \quad N\equiv\ldots B\theta\ldots \quad .$$

Note that the substitution is applied to only the rule $A \triangleright B$.

On the other hand, in resolution systems, substitutions usually appear in both rules and objects. A resolution system, such as Programming Logic system, has rules denoted by $A \triangleright B1, B2, \ldots, Bn$. Let clause $M \equiv M1, \ldots, Mi, \ldots, Mm$ be an object in this system. If there are substitution θ and $\widetilde{\theta}$, such that $A\theta \equiv Mi\widetilde{\theta}$, then $Mi\widetilde{\theta}$ occurring in $M\widetilde{\theta} \equiv M1\widetilde{\theta}, \ldots, Mi\widetilde{\theta}, \ldots, Mm\widetilde{\theta}$ can be replaced by $B1\theta, \ldots, Bn\theta$, and we obtain the following computation:

$$M \equiv M1, \ldots, Mi, \ldots, Mm \rightarrow N \equiv M1\widetilde{\theta}, \ldots, B1\theta, \ldots, Bn\theta, \ldots, Mm\widetilde{\theta}.$$

Note that, in this case, substitutions appear also in the object. Thus, we can get the next table:

Type	Rule	Object
Reduction	○	×
Resolution	○	○
	Substitution	

.

Next, we consider the rest of combination, i.e.,

Type	Rule	Object
?	×	×
?	×	○
	Substitution	

.

First, we give an example for the case <×, ○>, that is,

Rule: friend(YAMADA) $\triangleright$ OGAWA,

friend(OGAWA) $\triangleright$ HAYASHI,

Object: friend(friend(x)).

This object means the question who is connected to someone by using the friendship relation twice. By using the substitution [x:=YAMADA] for the object and applying the above rules, we get the following computation:

$$\text{friend(friend(x))} \xrightarrow{*} \text{HAYASHI.}$$

In data base systems, we usually use only relations between concrete objects to answer queries, hence, the above computation may be said Data Base type.

Finally, we consider the pair $\langle \times , \times \rangle$. This situation appears in Formal Language Theory. For example, context free language $\{a^n b^n\}$ is defined by:

Rule: $S \triangleright ab$,

$S \triangleright aSb$,

Object: S.

Then, we obtain a computation $S \xrightarrow{*} a^n b^n$ without substitutions.

From the above discussion, we obtain the following table representing four computation models. Here, the Data Structure column shows from what each object is constructed.

Type	Rule	Object	Research Area	Data Structure
Formal Language	×	×	Class of Languages, Automata, Complexity of computation.	String
Data Base	×	○	Decomposition of a relation, Consistency.	Table
Reduction	○	×	Reduction process, Semantics, Theory of computation.	Tree
Resolution	○	○	Logic, Proof method.	Set
	Substitution			

Acknowledgment. I wish to thank Dr. S. HAYASHI who suggested us to call Data Base type for the computation model defined by the pair <×, ○>.

5. Proof Systems for Logics with Types

Masahiko Sato

Summary

We have collected here a small bibliography that will, to some extent, cover existing implementations of proof systems for logics with some type structures.

Martin-Löf's Type Theory

See Hagiya's memo for more information on Martin-Löf's theory. We note that type structures are very similar among Martin-Löf's system, Edinburgh LCF (PPλ) and AUTOMATH although underlying logics are different.

[1] Kent Petersson: *A Programming System for Type Theory*, LPM Memo 21, March 1, 1982, Laboratory for Programming Methodology, University of Göteberg Chalmers University of Technology.

[2] Martin Löf, P.: *An Intuitionistic Theory of Types: Predicative Part*, Logic Colloquium '73, North-Holland, 1975.

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

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

Edinburgh LCF

In Edinburgh LCF, the polymorphically typed programming language ML is used as the meta language for the object language PPλ. The type structures between ML and PPλ is very similar and Constable [8] claims that it is possible to design a language which is at the same time its own meta language.

[5] Gordon, M., Milner, R. and Wadsworth, C.: Edinburgh LCF, Lect. Notes in Comp. Sci. 78, Springer 1978.

PL/CV2

PL/CV2 is a proof checking system based on a natural deduction style first order classical logic. However, Constable has an intention of extending it to a system with types like that of Martin-Löf's and LCF.

[6] Constable, R., Johnson, S., Eichenlamb, C.: Introduction to the PL/CV2 Programming Logic, Lect. Notes in Comp. Sci. 135, Springer 1982.

[7] Constable, R.: *Intensional Analysis of Functions and Types*, Dept. of Computer Science, Univ. of Edinburgh 1982.

[8] Constable R.: *Formalizing Metamathematics in Type Theory*, Dept. of Computer Science, Univ. of Edinburgh 1982.

AUTOMATH

AUTOMATH is a proof checking system that has successfully verified a text book on a theory of real numbers.

[9] de Bruijn, J.G.: *The mathematical language AUTOMATH, its usage and some of its extensions*, Symposium on Automated Demonstration, Lect. Notes in Math. 125, Springer 1970, 29-61.

[10] Jutting, L.S.: Checking Landau's "Grundlagen" in the AUTOMATH System, Math. Centre Tracts 83, Amsterdam 1979.

Feferman's System, Hayashi's System and Sato's System

These systems all treat Lisp's domain of symbolic expressions or its variants. The choice of the domain is based on the fact that encoding of algorithms and proofs can be done quite naturally into their respective domains.

[11] Feferman, S.: *Inductively presented system and formalization of meta-mathematics*, Logic Colloquium '80, North-Holland, 1982.

[12] Hayashi, S.: *Extracting Lisp programs from constructive proofs: A formal theory of constructive mathematics based on Lisp*, Publ. RIMS, Kyoto Univ., **19** (1983), 169-191.

[13] Sato, M.: *Theory of Symbolic Expressions, I*, Theoretical Computer Science, **22** (1983), 19-55.

FOL and EKL

FOL is a first order system, but its reflection mechanism is powerful enough to make the system a kind of meta-circular one. EKL is extended to finite-order predicate logic with typed λ-calculus.

[14] Weyhrauch, R.: *Prolegomena to a theory of mechanized formal reasoning*, Artificial Intelligence, **13**, 1980, 133-170.

[15] Ketonen, J. and Weening, S.: EKL — An Interactive Proof Checker, User's Reference Manual, Dept. of Computer Science, Stanford University 1982.

Meta reasoning systems

These systems can reason about themselves.

[16] Boyer, R.S. and Moore, J.S.: *Metafunctions: proving them correct and using them efficiently as new proof procedures*, The correctness problem in computer science, Academic Press, 1981.

[17] Bowen, K.A. and Kowalski, R.A.: *Amalgamating Language and Metalanguage in Logic Programming*, Logic Programming, Academic Press, 1982.

TPS

Unlike the systems mentioned above, TPS is a theorem proving system for type theory with λ-abstraction.

[18] Miller, D., Cohen, E., Andrews, P.: *A Look at TPS*, Lect. Notes in Comp. Sci. 138, 6th Conference on Automated Deduction, Springer 1982.

[19] Huet, G.: A Unification Algorithm for Typed λ-calculus, Theoretical Computer Science, **1** (1975), 27-57.

[20] Jensen, D., Pietrzykowski, T.: *Mechanizing ω-Order Type Theory through Unification*, Theoretical Computer Science, **3** (1976), 123-171.

6. Constructive Set Theory

Masami Hagiya

6.1. Introduction

Constructive Set Theory (CST) was formulated by Per Martin-Löf in [1] as a logical theory to develop constructive mathematics in general. The study was started purely from the logical and metamathematical interest, but as was discussed in [2], almost all the constructs of CST had their natural counterpart in the field of programming, so the possibility to directly implement CST as a programming language began to be recognized.

At least two projects are now running in the University of Göteborg to implement CST, which will be discussed later in this note.

When viewed as a programming language, CST has the following features:

* If a program is correctly typed, it terminates for any legal input, which means that only total functions are definable in CST.
* The type structure of CST is so rich that when a type of a program is written in sufficient detail, it can serve as a specification of that program.

From the purely operational point of view, one can say that CST is a typed lambda-calculus with a lambda-term as a type. But it is far more complicated than the usual typed lambda-calculus, since the terms and their types --- which are also lambda-terms --- are generated simultaneously (i.e. co-recursively).

6.2. Outline of CST

Since CST is a logical theory, its fundamental operation is to prove theorems. A theorem of CST is called a judgement. There are two kinds of judgement in CST. The first one is of the form

$$a \in A$$

and read in one of the following ways:

* a is an element of the set (type) A.
* a is a proof (construction) of the proposition (formula) A.
* a is a program for the task A.
* a is a solution of the problem A.

The second reading is obtained by regarding a proposition as the set of all the constructions that prove the proposition. This identity of sets and propositions is called the formulae-as-types notion and is inherent in intuitionistic logic. The third reading is derived by expressing a task as a proposition i.e. in a form of a logical formula, and this is what we mean when we say that the specification of a program is written as the type of the program.

The judgement

$$A \text{ set}$$

asserts that A is a set. It is a judgement one level higher than the above kind of judgement, since it says that A is an element of the class of all the sets.

The second kind of judgement is of the form

$$a = b \in A$$

which says that a and b denote the same element of the set A. In CST, the equality expresses just the convertibility between two terms, i.e. a = b means that a is convertible to b by the conversion rules, or a and b have the same canonical (normal) form. The usual reduction sequence from a to b by the reduction rules can be interpreted in CST as a derivation of the judgement $a = b \in A$, since a reduction is just a one-way conversion.

The inference rules to derive a new judgement from already established judgements are divided into the following four kinds:

* rule of set formation
* introduction rule
* elimination rule
* equality rule

The rule of set formation defines how to make a new set form the existing ones. Its conclusion is of the form A set. The introduction rule determines how to construct a canonical (normal) element of that new set. The elimination rule gives the way to break down the element in the set i.e. gives the selectors of the set. The equality rule defines the conversion rules between terms in the set as was discussed above.

To each set forming operator correspond four inference rules, one from each of the above four kinds. Here, let us list the fundamental set forming operations.

* finite set
* natural numbers
* lists
* function set (implication)
* direct product of two sets (conjunction)
* direct sum of two sets (disjunction)
* general direct product (universal quantification)
* general direct sum (existential quantification)

Each operator can be read as in the corresponding parentheses, when sets are interpreted as propositions.

6.3. Projects to Run CST

The first project to "run" CST was that of Bengt Nordström [3], in which was implemented the reducer i.e. interpreter of CST in Franz Lisp. [3] gives many example programs written in CST, which show the expressive power of CST as a higher order functional programming language. Let us give an example from [3], the summation operator,

```
sum(l, h, e) = rec h-l of
                  0: e(l),
                succ(x): from p to
                   p + e(succ(x)+1)
             endrec.
```

(sum(1, h, e) expresses $\sum_{i=1}^{h}$ e(i).) [3] uses what is called concrete syntax to increase the readability and writability of the programs.

The second project is that of Kent Petersson [4], in which the proof checker of CST is being built on ML of LCF [5], where CST takes the role of PPlambda of Edinburgh LCF. The project tries to evaluate the feasibility of CST for the program verification problems.

It is said that the movement to unify the two projects by incorporating Nordström's reducer into the proof checking system has been completed.

[1] Per Martin-Löf, An intuitionistic theory of types: predicative part, Logic Colloquium '73, H. E. Rose and J. C. Shepherdson eds., North-Holland, Amsterdam, 1973.

[2] Per Matin-Löf, Constructive mathematics and computer programming, Logic, Methodology and Philosophy of Science VI, North-Holland, Amsterdam, 1982.

[3] Bengt Nordström, Programming in constructive set theory: some examples, Proceedings of the ACM Conference on Functional Programming language and Computer Architecture, 1981.

[4] Kent Petersson, A programming system for type theory, LPM MEMO 21, Laboratory for Programming Methodology, Department of Computer Sciences, University of Göteborg, Chalmers University of Technology, 1982.

[5] M. Gordon, R. Milner and C. Wadsworth, Edinburgh LCF, LNCS 78, Springer Verlag, Berlin, 1979.

7. Combinatory systems

We will describe higher-order concepts in combinatory systems. Section 7.1 shows how we can treat higher-order functions in combinatory systems. In Section 7.2 combinatory systems with higher-order concepts, abstraction mechanism, are explained.

7.1 Higher-order functions and their implementation by combinators

T. Hikita

We will first examine higher-order concepts in applicative languages SASL (St. Andrews Static Language [7], 1979) and KRC (Kent Recursive Calculator [8], [9]), both developed by Turner. The concepts treated here are higher-order functions and ZF set abstractions, although the latter may not particularly be "higher-order". We then explain the combinator implementation of applicative programs in SASL and other variations of implementations.

Higher-order functions in SASL and KRC

First we give an example of a higher-order function written in KRC [9]. The following higher-order function "fold" distributes a binary operation "op" among elements of a list. The definition takes a form of recursion equations.

```
fold op s [ ]     =  s
fold op s (a:x)   =  op a (fold op s x)
```

Here [ ] is an empty list, and "a" and "x" are the head and tail parts of a list, respectively. Note that all the functions in KRC are considered as those with just one argument by so-called Currying, as follows: a + b as (+ a) b. The association of application is always to the left when parentheses do not appear. Some applications of "fold" are

```
sum      =  fold plus 0
product  =  fold times 1
```

Now we turn to the concept of ZF set abstraction. A simplest form of set abstraction in KRC is

$$\{x \ ; \ x \leftarrow y\}$$

where y is a (possibly infinite) list and x is assigned each element of the list successively. An example of the use of set abstraction: the function "perms" generates all permutations of a given list.

```
          perms [ ]  =  [[ ]]
          perms x    =  {a:p ; a ← x ; p ← perms(x--[a])}
```

Here "--" means the set difference operation for lists.

Implementation of functional programs by combinators

Higher-order functions, that is, functions receiving as arguments functions or those returning functions as result values, etc., are very naturally realized in the lambda calculus. But the direct implementation of substitution (ß-reduction) is inefficient in general. Turner proposed that the use of combinators could remedy this difficulty while preserving the naturalness of the treatment of higher-order functions [7]. (It seems that the language KRC is not implemented by combinators.)

Turner's implementation of functional programs by combinators proceeds in the following way. First a user's program is translated to a combinator expression by successively eliminating bound variables. This is a rather old and well-known fact in combinatory logic [2], but Turner's invention is a new base set of combinators which prevents translated expressions from combinatorial explosion of sizes. In the second phase a graph which the translated expression represents is transformed (reduced) to a result value when supplied with actual input arguments.

Following is an example of the implementation process. The function "pick" selects the "n"-th element of a list "s" (written in SASL).

```
          def pick n s  =  n = 1 → hd s ; pick (n-1) (tl s)
```

This function is translated to a combinator expression by abstracting the variables "s" and "n" in this order.

```
          def pick  =  S' S (C' (B' cond) (C eq 1) hd)
                               (C' (B' pick) (C minus 1) tl)
```

This is the object program for graph transformation. The definitions of the combinators by lambda expressions are:

$$S = \lambda xyz.xz(yz), \qquad C = \lambda xyz.xzy,$$
$$S' = \lambda wxyz.w(xz)yz, \quad B' = \lambda wxyz.wx(yz), \quad C' = \lambda wxyz.w(xz)y.$$

One drawback of this combinator implementation is the difficulty at debugging because of the long-winding object code.

Several actual implementations using combinators have been reported besides SASL, both in software and hardware, e.g. [1]. Comparison between the usual and combinator implementations have been done in [6], showing that the combinator implementation is better in speed for programs with higher-order functions.

Other combinator implementations

Since the work of Turner there appeared other similar but somewhat different ways of combinator implementations for functional programs. Jones and Muchnick [4] gave a (usual von Neumann-type) stack machine model and presented a translation method of combinator expressions to fixed programs on the machine. Kennaway and Sleep [5] studied a distributed processing model for evaluating combinator expressions. Finally Hughes [3] proposed an interesting implementation method using "super-combinators", which is, in a sense, a top-down implementation of functional programs by a non-prefixed base set of combinators, in contrast with Turner's bottom-up representation of programs by a prefixed base set of combinators.

References

[1] T. J. W. Clarke, P. J. S. Gladstone, C. D. MacLean and A. C. Norman : SKIM - The S, K, I reduction machine, 1980 LISP Conference, pp. 128-135.

[2] H. B. Curry and R. Feys : Combinatory Logic, Vol. I, North-Holland, 1958.

[3] R. J. M. Hughes : Super-combinators - a new implementation method for applicative languages, 1982 ACM Symp. on LISP and Functional Programming, pp. 1-10.

[4] N. D. Jones and S. S. Muchnick : A fixed-program machine for combinator expression evaluation, 1982 ACM Symp. on LISP and Functional Programming, pp. 11-20.

[5] J. R. Kennaway and M. R. Sleep : Expressions as processes, 1982 ACM Symp. on LISP and Functional Programming, pp. 21-28.

[6] S. L. Peyton Jones : An investigation of the relative efficiencies of combinators and lambda expressions, 1982 ACM Symp. on LISP and Functional Programming, pp. 150-158.

[7] D. A. Turner : A new implementation technique for applicative languages, Softw. Pract. Exper., 9 (1979), 31-49.

[8] D. A. Turner : The semantic elegance of applicative languages, ACM Conf. on Functional Programming Languages and Computer Architecture, 1981, pp. 85-92.

[9] D. A. Turner : Recursion equations as a programming language, in "Functional Programming and Applications", eds. J. Darlington et al., pp. 1-28, Cambridge Univ. Press, 1982.

7.2 Combinatory Reduction System

Yoshihito TOYAMA

The concept of the Combinatory Reduction System (CRS) is introduced by J. W. Klop [1]. Roughly speaking, the Combinatory Reduction System is a Combinatory System having abstraction mechanism, in other words, the union of a Combinatory system and the Lambda Calculus. Let C be the set of constant symbols and V be the set of variable symbols. The set of terms of CRS is defined by

(1) $C \cup V \subset$ Term,

(2) $[x]A \in$ Term if $x \in V$ and $A \in$ Term (abstraction),

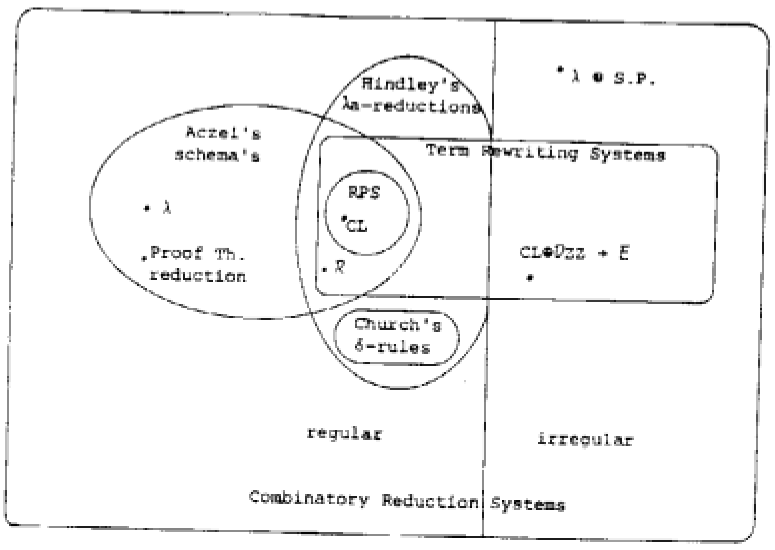
(3) $(AB) \in$ Term if $A, B \in$ Term (application),

where abstraction $[x]A$ is analogous to $\lambda x.A$ in the Lambda calculus.

The Reduction rule in CRS is a pair $\langle H, H' \rangle$ of meta-terms, written as $H \rightarrow H'$, where a meta-term is a term containing meta-variables (i.e., function variables) denoted by Z1, Z2,... . For example, β-reduction is written as $([x]Z1(x))Z2 \rightarrow Z1(Z2)$.

If any reduction rule in a CRS is non-ambiguous and left-linear, then the CRS is called regular. In a regular CRS hold the basic syntactical theorems such as the Church-Rosser theorem, the standard reduction theorem, the finite development theorem, and so on. Thus, the concept of CRS gives a natural higher-order extension of Combinatory Systems.

The following diagram shows the relation of various notations of reduction.



by Klop

Reference

[1] Klop,J.W. :"Combinatory Reduction Systems ", Dissertation, Univ. of Utrecht (1980).

8. Term Rewriting Systems

In this section the attempts for defining Term Rewriting Systems with higher order functions (operators) are sketched. 8.1 is a brief explanation of such TRS based on Combinatory System with Type (TCS). 8.2 is a preliminary attempt to define such TRS using S-expressions.

8.1 Higher-Order Term Rewriting System

Yoshihito TOYAMA

We will attempt to explain higher-order Term Rewriting System (HTRS) by using Combinatory System with type (TCS). HTRS may have function variables in the following way.

Examples:

(1) $H(f,x) \rhd f(f(x))$,

(2) $\begin{cases} G(g,0,x) \rhd x, \\ G(g,s(y),x) \rhd g(G(g,y,x)), \end{cases}$

where H and G are higher-order function symbols, and f and g are function variables. Rule(2) gives a reduction $G(g,n,x) \xrightarrow{*} \underbrace{g(g(\ldots g}_{n}(x)\ldots))$ for $n \geq 0$.

First, we will explain TCS. Type is defined by:

(1) $\mathrm{Type0} \subset \mathrm{Type}$ where Type0 is an elementary type,

(2) $\alpha\beta \in$ Type if $\alpha, \beta \in$ Type.

Let C be the set of constant symbols and V be the set of variable symbols, and each symbol $s \in C \cup V$ has a type denoted by $\tau(s)$. Then, term set T is defined by:

(1) $M \in T$ if $M \in C \cup V$,

(2) $MN \in T$ and $\tau(MN) = \beta$ if $M, N \in T$, $\tau(M) = \alpha\beta$, and $\tau(N) = \alpha$.

The set of terms with type is denoted by $T^{\sigma} = \{M : \tau(M) = \sigma\}$, hence $T = \bigcup_{\sigma} T^{\sigma}$. A substitution θ is a mapping from T to T such that

(1) $f\theta \equiv f$ if $f \in C$,

(2) $v\theta \in T^{\tau(v)}$ if $v \in V$,

(3) $(MN)\theta \equiv (M\theta)(N\theta)$,

where $\equiv$ is syntactical equality. A rewriting rule is a binary relation on T, written as $A \triangleright B$ (type), where type$\in$Type. If $A \triangleright B$ (type), then $\tau(A)=\tau(B)=$type. The rewriting rule defines a reduction relation $\rightarrow$ as follows:

$M \rightarrow N$ iff $M \equiv C[A\theta]$, $N \equiv C[B\theta]$, and $A \triangleright B$ (type), for some A, B, type, context C[ ], and θ.

HTRS R is defined by TCS. Let TCS have an elementary type Type0={0}. If HTRS R has a function symbol f with arity(f)=n, then we put a constant symbol f into TCS, where (f)=$\underbrace{0\ldots0}_{n}$ (association of type is to the right). Then, the previous examples of HTRS are denoted in TCS as follows:

(1) Let H be a constant symbol with $\tau(H)=(00)00$, and f, x be a variable symbol with $\tau(f)=00$ and $\tau(x)=0$, respectively. Then we obtain the following rules on TCS:

$$Hfx \triangleright f(fx) \quad (0) .$$

(2) Let G, s, 0 be a constant symbol with $\tau(G)=(00)000$, $\tau(s)=00$, $\tau(0)=0$, respectively, and let g, x be a variable symbol with $\tau(g)=00$, $\tau(x)=0$, respectively. Then we can get

$$\begin{cases} Gg0x \triangleright x \quad (0), \\ Gg(s(y))x \triangleright g(Ggyx) \quad (0). \end{cases}$$

Reference

[1] Hindley,J.R., Lercher,B., and Seldin,J.P.: " Introduction to combinatory logic", Cambridge University Press (1972).

[2] Strend,S.: " Combinators, Lambda-terms and proof theory ", D.Readel Publishing Company (1972).

8.2 CTRS: Combinatory Term Rewriting Systems

[extended abstract]

Kokichi Futatsugi

In this note we will propose term rewriting systems in which the terms are allowed to be "combinatory terms". This extension implies that we can freely uses higher order operators in the term rewriting systems.

We will use S-expressions (S-exp) for expressing terms in CTRS.

NOTATIONS

(1) The following symbols are meta symbols in BNF notation.

```
'::=' '[' ']'  '¦' '...'  '{'  '}'  '--'
```

(2) <SyntaxCategory>:"s-exp" denotes the "s-exp" which belongs to the SyntaxCategory.

TERMS

```
Atom ::= Name ¦ Variable
Variable ::= { any LISP's Atom starting with the character '*' }
Name ::= { any LISP's Atom which is not Variable }

SortName ::= Name

SortDecl ::= (SORT SortName...)

(1)  Type ::= SortName
(2)         ¦ (PROD Type...)   -- Cartesian product
(3)         ¦ (SUM  Type...)   -- disjoint union
(4)         ¦ (LIST Type)
(5)         ¦ (Type . Type)    -- function type

<Type>:typeA "matches" <Type>:typeF if and only if
(1)  typeF = <SortName>:sort0 and typeA = sort0
(2)  typeF = (PROD typeF1 typeF2 ... typeF(n)),
     typeA = (PROD typeA1 typeA2 ... typeA(n)) and
     typeA(i) matches typeF(i) for i = 1, 2, ... , n
(3)  typeF = (SUM typeF1 typeF2 ... typeF(n)) and
       typeA matches typeF(i) for some i = 1, 2, ..., n,
       or,
       typeA = (SUM typeA1 typeA2 ... typeA(m)) and
       any typeA(i) (i = 1, 2, ... ,m) matches typeF(j)
       for some j = 1, 2, ... ,n
(4)  typeF = (LIST typeF1), typeA = (LIST typeA1) and
     typeA1 matches typeF1
(5)  typeF = (typeF1 . typeF2),
     typeA = (typeA1 . typeA2) and
     typeA(i) matches typeF(i) for all i = 1, 2
```

```
OpName ::= Name

OpDecl ::= (OPER (OpName Type)...)  -- operator declaration
```

When
```
  (OPER (op1 type1) (op2 type2) ... (op(n) type(n)))
```
has been declared, we say that the op(i)'s type is type(i) for i = 1,2,...,n.

```
(1)     Term ::= (Variable Type)
(2)            | OpName
(3)            | (# Term...)          -- # is a reserved Name
(4)            | ($ Term...)          -- $ is a reserved Name
(5)            | (Term . OpName)      -- OpName applies to Term
(6)            | (Term . Term)        -- the second Term acts as a operator
```

The map:
```
        typeOf_: Term -> Type
```
assigns some specific <Type>:type to every <Term>:term as follows.

```
(1)       typeOf <Term>:(<Variable>:var <Type>:type)
                = type
(2)   If <OpName>:op's type be <Type>:type then
          typeOf op = type.
(3)   If
         typeOf term(i) = type(i)  for i = 1,2,...n
      then
         typeOf (# term1 term2 ... term(n))
          = (PROD type1 type2 ... type(n)).
(4)   If
         typeOf term(i) = type  for i = 1,2,...n
      then
         typeOf (& term1 term2 ... term(n))
          = (LIST Type)
(5)   If <OpName>:op's type be <Type>:
          (<Type>:typeF1 <Type>:typeF2 ... <Type>:typeF(n))
      and,
          typeOf <Term>:termF1 = <Type>:typeA  and
          typeA matches typeF1,
      then
          typeOf (termA . op) = (typeF2 ... typeF(n)).
(6)   If
          typeOf <Term>:term
           = <Type>:(<Type>:typeF1 <Type>:typeF2 ... <Type>:typeF(n))
      and,
          typeOf <Term>:termA = <Type>:typeA and
          typeA matches typeF1,
      then
          typeOf (termA . term) = (typeF2 ... typeF(n)).
```

Every terms must be "well-formed" in the sense that the typeOf map can be defined for that terms.

EQUATIONS AND TERM REWRITING

Eq ::= (EQ Term Term)

In any <Eq>:
          (EQ <Term>:termL <Term>:termR),

(i) typeOf termR must match typeOf termL, and

(ii) the <Term>:term of the form (Variable Type) which appears in
     <Term>:termR must also appeare in <Term>:termL.

EqDef ::= (DEFEQ Eq...)

When <EqDef>:
       (DEFEQ <Eq>:eq1 <Eq>:eq2 ... <Eq>:eq(n))
has been declared, we say that <Eq>:eq(i) (i = 1, 2, ..., n) have been declared.

If no <Term>:term of the form (Variable Type) appeares in <Term>:gTerm, gTerm is said to be "ground".

In the environment where the <Eq>:eq(j) (j = 1, 2, ..., n) are declared, we define the relation:
           <Term>:term1 => <Term>:term2

of "term1 is rewritten to term2" on the set of "ground" terms, if there exist some <Eq>:eq(j) (j = 1, 2, ..., n) such that term1 can be rewritten to term2 using eq(j). (Precise definition of rewriting is omitted here.)

If there exist no <Term>:term2 such that <Term>:term1 => <Term>:term2, then the term1 is called "canonical". Let =*> denote the reflexive and transitive closure of =>. If <Term>:term1 =*> <Term>:term2 and term2 is canonical, then the term2 is said to be a "canonical term of term1". Sometimes it is convenient to consider the case where every term has at most one canonical term.

In CTRS, same as ordinary TRS, construction of <SortDecl>, <OpDecl>, and <EqDef> corresponds to program construction, and running that program implies obtaining the the canonical term(s) of some given ground term.

9. HIGHER ORDER UNIFICATION

K. Sakai and T. Matsuda

9.1 HISTORY:

The unification problem for a formal language is the problem of determining whether any two terms of the language possesses a common instance or not. The study of unification was initiated by J.R.Guard [Guard 64] and J.A.Robinson [Robinson 65]. The unification problem for first order language is decidable and the latter paper includes an algorithm that finds a unique substitution, called the most general unifier, for two formulas of a first order language together with a complete inference rule, called resolution principle, for mechanizing first order logic. Existence of the most general unifier is of critical importance for the proof procedure currently used in automatic theorem proving. Basically it permits us to restrict the rule of substitution to the most general unifier in order to apply the cut rule (i.e. modus ponens).

[Robinson 69] suggested a system of higher order logic and discussed the implementation of the proof procedure for the system. P.B.Andrews [Andrews 71] has described a refutation system for Church's type theory. [Lucchesi 72] and [Huet 73] independently showed that there are two terms of third order for which it is not decidable whether they have a common instance or not. Their results imply that the third order k-adic unification problem is undecidable for k >= 4, because the proofs reduced the unification problem to the PCP (Post's correspondence problem) and there exists k >= 4 such that the PCP with k pairs of words is undecidable. [Baxter 78] refined the result to the third order dyadic unification problem by reducing it to the Hilbert's tenth problem. [Goldfarb 81] showed that the second order unification problem is undecidable by reducing it to Hibert's tenth problem again.

9.2 DEFINITIONS:

We will define types, terms, unifiers for higher order unification according to [Huet 73].

<types> Let T0 be a finite set of elementary types. The set T of types and the functions O, A, called the order and the arity of types respectively, from T into the set of positive integers are defined recursively as follows.

(1) If t is in T0, then t is in T and O(t) = 1, A(t) = 0.
(2) If t1, ..., tn is in T and t is in T0, then
 t' = (t1, ..., tn -> t) is in T
 and
 O(t') = max{O(ti)| 1 =< i =< n } + 1,
 A(t') = n.

<terms> The set of terms consists of atoms, applications and abstractions. Every term e possesses a type T(e), given by the following definition.

(1) Atoms: There exists a denumerable set of variables of each type and arbitrary number of constants of any type. We shall usually denote variables by strings headed by a lower case letter and constants by strings headed by a upper case letter.
(2) Applications: If e is a term of type

```
            T(e) = (t1, ..., tn -> t)
        and if e1, ..., em are terms of types
            T(ei) = ti       0 =< i =< m =< n,
        then we define the application
            e' = e(e1, ..., em)
        as a term of type
            T(e') = (t m+1, ..., tn -> t)        if n > m
                  = t                            if n = m
    (3) Abstractions:  If e is a term and u1, ..., un are distinct
        variables with
            T(e) = t,
            T(ui) = ti       0 =< i =< n
        then we define the abstraction
            e' = (^u1 ... un).e
        as a term of type
            T(e') = (t1, ..., tn -> t)          if t is in T0
                  = (t1. ..., tn, t1', ..., tm' -> t')
                                          if t = (t1', ..., tm' -> t')

<Normal form> Iterative application of lambda conversion reduces a term
        to a unique normal form, which does not contain any subterm of
        of the forms:
            (^u1 ... um).((^v1 ... vn).e),
            (e(e1, ..., em))(f1, ..., fn),
            ((^u1 ... um).e)(e1, ..., en).
        We will consider two or more terms identical if they have the
        same normal form.

<Substitution> A substitution S is a set of finite ordered pairs
            {<ui,ei>| 1 =< i =< n }
        where ui's are distinct variables, ei's are terms and
            T(ui) = T(ei)            1 =< i =< n.
        The application of S to a term E is defined as the normal
        form of:
            S(E) = ((^u1 ... un).E)(e1, ..., en).

<Unification> Two terms e1 and e2 of the same type are said to be
        unifiable  if there exists a substitution S, called unifier,
        for e1 and e2, such that S(e1) = S(e2).
```

9.3 PERSPECTIVES

The undecidability results for second and higher order unification as well as the enormous proliferation of unifiers even for small problems have cast some shadows on earlier hopes for higher order theorem proving.

Unification is a fundamental process in all contemporary first order theorem provers, since it is embedded as the basic operation in rules such as resolution, factoring and paramodulation. It is not possible to extend these rules to higher order logic directly, because of the absence of a most general unifier.

J.A.Robinson [Robinson 69] proposed another approach to automating higher order logic, which does not require the process of unification. Unfortunately it would probably suffer the same disadvantages as Herbrand-base saturation methods for first order logic.

A new method trying to overcome the difficulties of unifier based method is presented in [Huet 72]. Basically, this procedure computes unifiers for a sequence of cuts leading to a refutation, rather than for an individual cut. The main advantage here is that, by delaying as much as possible the search for unifiers, we save computation time, because the cumulated information permits us to reject a lot of irrelevant cases. Unfortunately we still need to detect the existence of a unifier, and therefore some enumerating process is still necessary.

Given some equational theory T, the higher order unification problem for it is less complex than free higher order unification problem. For example the second order monadic unification problem closely resembles the string-unification problem. Now the string-unification problem is infinitary, it posed a very hard decidability problem and the known string-unification algorithms [Plotkin 72], [Siekmann 75], [Livesey 75], are almost useless for all practical proposes. However string-unification under commutativity is comparatively simple [Stickel 75], [Livesey 76], : it is finitary, decidability is easy to compute and the unification algorithms are not too far away from practical applicability.

9.4 REFERENCES

[Andrews 71] Andrews, P.B.
"Resolution in type theory"
J. Symbolic Logic 36, 414-432 (1971).

[Baxter 78] Baxter, L.D.
"The undecidability of third order dyadic unification problem,"
Information and Control, vol.38, No.2, 170-178 (1978)

[Goldfarb 81] Goldfarb, D.
"The undecidability of the second order unification problem,"
Theoretical Computer Science, 13, 225-230 (1981)

[Guard 64] Guard, J.R.
"Automated logic for semi-automated mathematics,"
Scientific report No.1, Air Force cambridge research lbs.,
64-411, AD 602 710, (1964)

[Huet 73] Huet, G.
"The undecidability of unification in the third order logic"
Information and Control 22 (3), 257-267, (1973)

[Livesey 75] Livesey, M., Siekmann, J.,
"Termination and decidability results for stringunification,"
Univ. of Essex, Memo CSM-12, (1975)

[Livesey 76] Livesey, M., Siekmann, J.,
"Unification sets and multisets," Univ. Karlsruhe, Techn. Report, (1976)

[Lucchesi 72] Lucchesi, C.L.
"The undecidability of the unification problem for third order languages,"
rep. CSRR 2059, Dept. of Applied Analysis and Computer Science,
Univ. of Waterloo, (1972)

[Plotkin 72] Plotkin, G.D.
"Building in equational theories," Machine Intellighence, vol 7 (1972)

[Robinson 65] Robinson, J.A.
"A machine-oriented logic based on the resolution principle,"
J. ACM 12,(1965) 23-41.

[Robinson 69] Robinson, J.A.
"Mechanizing higher-order logic,"

Machine Intelligence 4, 151-170, (1969).

[Robinson 70] Robinson, J.A.
"A note on mechanizing higher order logic,"
Machine Intelligence 5, 123-133, (1970).

[Siekmann 75] Siekmann, J.
"Stringunification," Essex University, memo CSM-7 (1975)

[Stickel 75] Stickel, M. E.
"A complete unification algorithm for associative-commutative functions," 5th-IJCAI75, 71-76, (1975)

LIST OF CONTRIBUTORS

Kokichi	Futatsugi	Electrotechnical laboratory
Masami	Hagiya	RIMS, Kyoto University
Susumu	Hayashi	The Metropolitan College of Technology
Teruo	Hikita	Tokyo Metropolitan University
Tetsuo	Ida	Institute of Chemical and Physical Research
Toshiaki	Kurokawa	3rd laboratory, ICOT
Toshio	Matsuda	Science University of Tokyo
Ko	Sakai	3rd laboratory, ICOT
Masahiko	Sato	The University of Tokyo
Yoshihito	Toyama	Musashino ECL, NTT

Acknowledgement

We acknowledge active and helpful discussions of other members of WG. 5, in particular T. Yokoi of ICOT who has been instrumental in organizing the activities described in this report.