I C L
Defence
Systems

Defence Business Centre

Technology Development Department

Formal Methods

Ref: DBC/RBJ/076 **Issue:** 0.1 **Pages:** ?

Date: 1988-02-04 **Status:** Draft

Title: Combinatory Type Theory in HOL

Prepared by: R.B.Jones

Document Type: Formal Theory

Keywords: COMBINATORY-LOGIC TYPE-THEORY HOL

Sign-off Authority		
Name	Location	Signature

Summary:
This document contains a formal development of a combinatory type theory in Higher Order Logic (HOL).

Distribution: * = top sheet only

Combinatory Type Theory in HOL

R.B.Jones

Formal Methods Group,
ICL Defence Systems.

1. INTRODUCTION

The combinatory terms are the free algebra generated by the 2-ary operator Ap from the 0-ary constants K,S,Ξ and a countable set of 0-ary variables (Va n), for n∈num.

Pure combinators are combinatory terms in which there occurs no 0-ary consitituents except K and S.

The combinators are our primitive language in which we assert propositions. The subject matter of these propositions is the pure combinators, and ranges, pacs, types, and type_assignments which may be represented, or approximated by pure combinators.

ML
```
new_theory(‘076‘);;
```

2. REPRESENTING COMBINATORS BY NUMBERS

In order to use HOL as a metalanguage we must determine some way to represent the objects in the language as objects in HOL.

This is done using the type "num" consisting of the natural numbers 0,1,2... 0,2 and 4 represent respectively K, S, Ξ, and the remaining even numbers are used to represent variables. Odd numbers are used to represent applications, using the bijection: (n,m)->(n+m)(n+m)+(2nm)+n+3m+1. Projections are defined, pro-tem, using the choice function μ.

ML
```
new_definition(‘K_DEF‘,
        "K = 0"                                                        );;
new_definition(‘S_DEF‘,
        "S = 2"                                                        );;
new_definition(‘Ξ_DEF‘,
        "Ξ = 4"                                                        );;
```

```
ML
new_definition(‘Va_DEF‘,
        "(Va:num→num) n = 6+(2*n)"                                    );;
new_infix_definition(‘Ap_DEF‘,
        "(Ap:num→num→num) n m = (n*n)+(m*m)+(2*n*m)+n+(3*m)+1"        );;
```

```
ML
new_definition(‘is_atom_DEF‘,
        "(is_atom:num→bool) n = ∃x.n=2*x"                              );;
new_definition(‘is_Ap_DEF‘,
        "(is_Ap:num→bool) n = ¬ is_atom n"                             );;
```

```
ML
new_definition(‘fun_DEF‘,
        "(fun:num→num) n = μx.∃y.n=x Ap y"                             );;
new_definition(‘arg_DEF‘,
        "(arg:num→num) n = μy.∃x.n=x Ap y"                             );;
```

3. CONVERSION

The combinators are to be used to represent functions over combinators. The first step in showing how this is done consists in defining an equivalence relation over the combinatory terms. Operationally the semantics of combinators as representing functions is determined by a graph reduction process. If this graph reduction process is regarded as preserving equality in some way, then the strongest equality compatible with the normal reduction rule for K and S is the equivalence relation which we define below as ’conv’.

the basic reduction rules for K and S. We therefore first define an equivalence relation, then an applicative relation, the relation of immediate reducibility, and finally ’conv’.

3.1. Equivalence Relations

```
ML
let relation = ":*num→*num→bool";;
let numeric_relation = ":num→num→bool";;
```

ML
```
new_definition(‘reflexive_DEF‘,"(reflexive:^relation→bool) r =
        ∀x:*num.r x x");;
new_definition(‘symmetric_DEF‘,"(symmetric:^relation→bool) r =
        ∀x y.r x y ☞ r y x");;
new_definition(‘transitive_DEF‘,"(transitive:^relation→bool) r =
        ∀(x:*num)(y:*num)(z:*num).(r x y) ∧ (r y z) ☞ (r x z)");;
new_definition(‘equivalence_DEF‘,"(equivalence:^relation→bool) r =
        reflexive r ∧ symmetric r ∧ transitive r");;
new_infix_definition(‘contains_DEF‘,"($contains:^relation→^relation→bool) r1 r2 =
        ∀x y. r2 x y ☞ r1 x y");;
new_definition(‘equivalence_closure_DEF‘,
        "(equivalence_closure:^relation→^relation) r =
        µs. equivalence s ∧ s contains r ∧
        ∀t:^relation. equivalence t ∧ t contains r ☞ t contains s");;
```

3.2. Applicative Relations

ML
```
new_definition(‘applicative_DEF‘,"(applicative:^numeric_relation→bool) r =
        ∀x x’ y y’. (r x x’ ∧ (r y y’  (y=y’)))  ((x=x’) ∧ (r y y’))
        ☞ r (x Ap y) (x’ Ap y’)");;

new_definition(‘applicative_closure_DEF‘,
        "(applicative_closure:^numeric_relation→^numeric_relation) r =
        µs. applicative s ∧ s contains r ∧
        ∀t:^numeric_relation. applicative t ∧ t contains r ☞ t contains s");;
```

3.3. Immediate Reducibility and Conversion

ML
```
new_definition(‘prim_red_DEF‘,"(prim_red:num→num→bool) x y =
        (∃u. x = (K Ap y) Ap u)
        ∃u v w. x = ((S Ap u) Ap v) Ap w");;
new_infix_definition(‘im_red_DEF‘,"($im_red:num→num→bool)
        = (applicative_closure prim_red)");;
```

ML
```
new_infix_definition(‘conv_DEF‘,"($conv:num→num→bool)
        = (equivalence_closure $im_red)");;
```

4. SUBSTITUTION

```
ML
new_definition(‘subst_DEF‘,"(subst:num→num→num→num) =
        μf. ∀x y z.
                ((z = y) ☞ (f x y z = x)) ∧
                (¬(z = y) ☞
                        ((is_atom z) ☞ (f x y z = z)) ∧
                        ((is_Ap z) ☞ (f x y z = ((f x y (fun z)) Ap (f x y (arg z))))))");;
new_definition(‘value_DEF‘,"(value:num→num→num→bool) x y z =
        (subst y Ξ x) conv z");;
```

5. ENCODINGS

```
ML
new_definition(‘I_DEF‘,"I = (S Ap K) Ap K");;
new_definition(‘cabs_DEF‘,"(cabs:num→num→num) =
        μf.∀var body.
        ((Va var = body) ☞ (f var body = I)) ∧
        (¬(Va var = body) ☞
                ((is_atom body ☞ (f var body = body)) ∧
                (is_Ap body ☞
                        (f var body = (f var (fun body)) Ap (f var (arg body))))))");;
new_prim_rec_definition(‘cabsn_DEF‘,"
        (cabsn 0 n = n)      ∧
        (cabsn (SUC m) n = cabs m (cabsn m n))");;
```

```
ML
new_definition(‘ctrue_DEF‘,
                        "ctrue = K");;
new_definition(‘cfalse_DEF‘,
                        "cfalse = cabsn 1 (Va 0)");;
new_definition(‘cif_DEF‘,
                        "cif x y z = ((x Ap y) Ap z)");;
new_definition(‘cpair_DEF‘,
                        "cpair x y = cabs 0 (cif (Va 0) x y)");;
new_definition(‘cfst_DEF‘,
                        "cfst x = x Ap ctrue");;
new_definition(‘csnd_DEF‘,
                        "csnd x = x Ap cfalse");;
```

```
ML
new_definition(‘encode_DEF‘,"encode:num→num =
        μ f.∀x.
                ((x=K)☞ (f x = cpair ctrue ctrue)) ∧
                ((x=S)☞ (f x = cpair ctrue cfalse)) ∧
                (is_Ap x ☞ (f x = cpair cfalse (cpair (f (cfst x))(f (csnd x)))))");;
```

6. INTERPRETATIONS

I now want to define the acceptable interpretations of Ξ. Informally Ξ should be understood as a function which takes two encodings of combinators and returns true if the range represented by the first is contained in the range represented by the second, otherwise it returns false. This function is not recursive, so we have to accept an approximation. An approximation is a function which never yields an incorrect truth value, but may sometimes yield no truth value.

We may use combinators to represent and number of different things.

6.1. Ranges

First a combinator may represent a range, by which should be understood a range of quantification.

```
ML
let range = ":num→bool";;
```

```
ML
new_definition(‘comb_to_range‘,"(comb_to_range:num→^range) x y =
        (x Ap (encode y)) conv ctrue");;
```

A fundamental relation over ranges is inclusion:

```
ML
new_definition(‘rnge_incl‘,"(rnge_incl:^range→^range→bool) r s =
        (∀x. r x ☞ s x)");;
```

Ξ is intended to approximate ‘rnge_incl‘ in the object language. Since ‘rng_incl‘ is not recursive (or recursively enumerable), we have to accept computable approximations.

Second a combinator may represent a partial characteristic function, which we abbreviate, pac.

```
ML
let pac = ":num→(bool#bool)";;
```

The type of an approximation is a partial relation over ranges:

```
ML
let approxim = ":^range→^range→(bool#bool)";;
```

And the perfect ’approximation’ to Ξ is:

```
ML
new_definition(‘poxΞ‘,"(poxΞ:^approxim) r s =
        let b = rnge_incl r s in (b,¬b)");;
```

Since every combinator represents a range this induces a partial relation over combinators as follows:

```
ML
let approximc = ":num→num→(bool#bool)";;
```

```
ML
new_definition(‘poxΞc‘,"(poxΞc:^approximc) x y =
        poxΞ (comb_to_range x) (comb_to_range y)");;
```

We now define a partial ordering over these approximations:

```
ML
new_infix_definition(‘better_than‘,"($better_than:^approximc→^approximc→bool) a b =
        ∀(x:num) (y:num).        let a_yes = FST (a x y) in
                        let a_no = SND (a x y) in
                        let b_yes = FST (b x y) in
                        let b_no = SND (b x y) in
                        ((b_yes ☞ a_yes) ∧ (b_no ☞ a_no)) ");;
```

A combinator when regarded as a partial characteristic function over encoded terms determines a ’partial relation’ over combinators as follows:

```
ML
new_definition(‘comb_to_prel‘,"(comb_to_prel:num→^approximc) n x y =
        ((n Ap (encode x) Ap (encode y)) conv ctrue,
         (n Ap (encode x) Ap (encode y)) conv cfalse)");;
```

We can now define the notion of approximation to Ξ as any combinator which poxΞc is ’better_than’:

```
ML
new_definition(‘approxΞ‘,"(approxΞ:num→bool) x =
        poxΞc better_than (comb_to_prel x)");;
```

```
ML
new_infix_definition(‘$better_Ξ_than‘,"($better_Ξ_than:num→num→bool) x y =
        approxΞ x
        ∧ (comb_to_prel x) better_than (comb_to_prel y)");;
new_infix_definition(‘true_for‘,"($true_for:num→num→bool) x y =
        (subst y Ξ x) conv ctrue");;
new_definition(‘valid‘,"(valid:num→bool) x =
        ∃y. approxΞ y ∧
        (∀z. z better_Ξ_than y ☞ x true_for z)");;
```

7. PACS

A pac is a partial characteristic function, or a pair of disjoint recursively enumerable sets.

8. TYPES

The next objective is to define what a type is in this system, and then to define a useful set of type constructors. Informally a type is an ordered pair. The first element is a PAC, the second an equivalence relationship defined over the RANGE of the PAC.

9. LISTING OF THEORY

```
The Theory 076
Parents --  HOL
Constants --
  K ":num"     S ":num"     Ξ ":num"     Va ":num → num"
  is_atom ":num → bool"     is_Ap ":num → bool"
  fun ":num → num"     arg ":num → num"
  reflexive ":(*num → (*num → bool)) → bool"
  symmetric ":(*num → (*num → bool)) → bool"
  transitive ":(*num → (*num → bool)) → bool"
  equivalence ":(*num → (*num → bool)) → bool"
  equivalence_closure
    ":(*num → (*num → bool)) → (*num → (*num → bool))"
  applicative ":(num → (num → bool)) → bool"
  applicative_closure
    ":(num → (num → bool)) → (num → (num → bool))"
  prim_red ":num → (num → bool)"
  subst ":num → (num → (num → num))"
  value ":num → (num → (num → bool))"     I ":num"
  cabs ":num → (num → num)"     cabsn ":num → (num → num)"
  ctrue ":num"     cfalse ":num"
  cif ":num → (num → (num → num))"
  cpair ":num → (num → num)"     cfst ":num → num"
  csnd ":num → num"     encode ":num → num"
  comb_to_range ":num → (num → bool)"
  rnge_incl ":(num → bool) → ((num → bool) → bool)"
  poxΞ ":(num → bool) → ((num → bool) → bool # bool)"
  poxΞc ":num → (num → bool # bool)"
  comb_to_prel ":num → (num → (num → bool # bool))"
  approxΞ ":num → bool"     valid ":num → bool"
Curried Infixes --
  Ap ":num → (num → num)"
  contains
    ":(*num → (*num → bool)) → ((*num → (*num → bool)) → bool)"
  im_red ":num → (num → bool)"     conv ":num → (num → bool)"
  better_than
    ":(num → (num → bool # bool)) →
      ((num → (num → bool # bool)) → bool)"
  better_Ξ_than ":num → (num → bool)"
  true_for ":num → (num → bool)"
Definitions --
  K_DEF   K = 0
  S_DEF   S = 2
  Ξ_DEF   Ξ = 4
  Va_DEF   Va n = 6 + (2 * n)
  Ap_DEF
     n Ap m =
       (n * n) + ((m * m) + ((2 * (n * m)) + (n + ((3 * m) + 1))))
```

```
is_atom_DEF   is_atom n = (∃x. n = 2 * x)
is_Ap_DEF   is_Ap n = ¬is_atom n
fun_DEF   fun n = (µx. ∃y. n = x Ap y)
arg_DEF   arg n = (µy. ∃x. n = x Ap y)
reflexive_DEF   reflexive r = (∀x. r x x)
symmetric_DEF   symmetric r = (∀x y. r x y ☞ r y x)
transitive_DEF   transitive r = (∀x y z. r x y ∧ r y z ☞ r x z)
equivalence_DEF
   equivalence r = reflexive r ∧ symmetric r ∧ transitive r
contains_DEF   r1 contains r2 = (∀x y. r2 x y ☞ r1 x y)
equivalence_closure_DEF
   equivalence_closure r =
     (µs.
       equivalence s ∧
       s contains r ∧
       (∀t. equivalence t ∧ t contains r ☞ t contains s))
applicative_DEF
   applicative r =
     equivalence r ∧
     (∀x x’ y y’. r x x’ ∧ r y y’ ☞ r(x Ap y)(x’ Ap y’))
applicative_closure_DEF
   applicative_closure r =
     (µs.
       applicative s ∧
       s contains r ∧
       (∀t. applicative t ∧ t contains r ☞ t contains s))
prim_red_DEF
   prim_red x y =
     (∃u. x = (K Ap y) Ap u)  (∃u v w. x = ((S Ap u) Ap v) Ap w)
im_red_DEF   $im_red = applicative_closure prim_red
conv_DEF   $conv = equivalence_closure $im_red
subst_DEF
   subst =
     (µf.
       ∀x y z.
        ((z = y) ☞ (f x y z = x)) ∧
        (¬(z = y) ☞
         (is_atom z ☞ (f x y z = z)) ∧
         (is_Ap z ☞ (f x y z = (f x y(fun z)) Ap (f x y(arg z))))))
value_DEF   value x y z = (subst y Ξ x) conv z
I_DEF   I = (S Ap K) Ap K
cabs_DEF
   cabs =
     (µf.
       ∀var body.
        ((Va var = body) ☞ (f var body = I)) ∧
        (¬(Va var = body) ☞
```

```
          (is_atom body ☞ (f var body = body)) ∧
          (is_Ap body ☞
           (f var body = (f var(fun body)) Ap (f var(arg body))))))
 cabsn_DEF_DEF
   cabsn = PRIM_REC(λn. n)(λg00012 m n. cabs m(g00012 n))
 ctrue_DEF   ctrue = K
 cfalse_DEF   cfalse = cabsn 1(Va 0)
 cif_DEF   cif x y z = (x Ap y) Ap z
 cpair_DEF   cpair x y = cabs 0(cif(Va 0)x y)
 cfst_DEF   cfst x = x Ap ctrue
 csnd_DEF   csnd x = x Ap cfalse
 encode_DEF
   encode =
     (μf.
       ∀x.
        ((x = K) ☞ (f x = cpair ctrue ctrue)) ∧
        ((x = S) ☞ (f x = cpair ctrue cfalse)) ∧
        (is_Ap x ☞ (f x = cpair cfalse(cpair(f(cfst x))(f(csnd x))))))
 comb_to_range   comb_to_range x y = (x Ap (encode y)) conv ctrue
 rnge_incl   rnge_incl r s = (∀x. r x ☞ s x)
 poxΞ   poxΞ r s = (let b = rnge_incl r s in b,¬b)
 poxΞc   poxΞc x y = poxΞ(comb_to_range x)(comb_to_range y)
 better_than
   a better_than b =
     (∀x y.
       let a_yes = FST(a x y)
       in
       let a_no = SND(a x y)
       in
       let b_yes = FST(b x y)
       in
       let b_no = SND(b x y) in (b_yes ☞ a_yes) ∧ (b_no ☞ a_no))
 comb_to_prel
   comb_to_prel n x y =
     (n Ap ((encode x) Ap (encode y))) conv ctrue,
     (n Ap ((encode x) Ap (encode y))) conv cfalse
 approxΞ   approxΞ x = poxΞc better_than (comb_to_prel x)
 $better_Ξ_than
   x better_Ξ_than y =
     approxΞ x ∧ (comb_to_prel x) better_than (comb_to_prel y)
 true_for   x true_for y = (subst y Ξ x) conv ctrue
 valid
   valid x =
     (∃y. approxΞ y ∧ (∀z. z better_Ξ_than y ☞ x true_for z))
```

Theorems --

```
 cabsn_DEF   (cabsn 0 n = n) ∧ (cabsn(SUC m)n = cabs m(cabsn m n))
```