Combinatory Logic

We have seen that variables in expressions, especially those involving functions (the λ-calculus) can be a nuisance since we have binding occurrences of variables, bound occurrences of variables and free occurrences of the variables. Technically, one can take an abstract syntax tree, and from a variable occurrence (always at a leaf position), trace the part to the root of the tree, and if it crosses a λ node, which has a binding occurrence of the same variable (i.e., the same name), then the leaf is a bound occurrence, bound to the *nearest* λ-binding occurrence. It no such λ node is encountered, then the variable occurrence at the leaf is "free". Of course, this definition is not the most efficient way to check if a variable occurrence is binding, bound, or free (hint: a top-down method, accumulating the binding occurrences in a LIFO list is the better way to associate a bound occurrence to the binding occurrence).

Example: Recall that in the following term, the occurrences of variable x are (respectively), binding, bound, and free: $(\lambda x\,.\,x)\ x$.

bounded to

There is a formal theory, called α-equivalence, where we can systematically rename binding occurrences of variables and all their corresponding bound occurrences (those connected to this binding occurrence). So we can rename the above expression to a *semantically equivalent* form, $(\lambda y\,.\,y)\ x$, where we avoid binding and bound occurrences of variable x. Note that we cannot rename free occurrences of variables ("Once free, always free"). Free variables are the target of substitution. Substitution in the presence bound and binding occurrences is a subtle operation to define, since one must replace *all* free occurrences of designated free variables, but in the process not accidently capture (i.e., make bound) any free variables.

Hence there is always a desire to somehow present a calculus of functions and function call that avoids using binding- and bound occurrences of variables. This seems almost paradoxical, since formal parameters (the binding occurrences) and their use in the body of the function (the bound occurrences) seem fundamental when defining and using functions.

Nonetheless, there are various ways of encoding functions (especially in an untyped calculus), where we can and do avoid using binding- and bound occurrences of variables. One such is called Combinatory Logic (CL), and as all such calculi, was developed as a syntax for manipulating proof terms. CL was proposed by Moses Schönfinkel around 1921 to be able to encode proofs in Hilberts proof systems. However, they are a rather nice "target language" to which we can translate the λ-calculus. We study it here in that spirit.

2.0 Abstract Syntax of Combinatory Logic

Expressions in Combinatory Logic are given by:

$$P, Q, R \in CL \ ::= \ x \mid \mathbf{I} \mid \mathbf{K} \mid \mathbf{S} \mid (P_1\ P_2)$$

As usual, we have variables x — but only free variables. Then we have three functional constants, called combinators, namely **I**, **K**, **S**, whose meaning we will specify below.

Exercise: Define a data type `comb` in OCaml for terms in CL. (Note that you will need a different constructor each for variables and application).

Equational Theory of CL

(I) $\mathbf{I}\,P =_{CL} P$

(K) $\mathbf{K}PQ =_{CL} P$

(S) $\mathbf{S}PQR =_{CL} (PR)(QR)$

$KPQ = P$

$K\,KP\,Q =$

These rules define the meaning of the combinators in terms of an equational relation $=_{CL} \subseteq CL \times CL$

For $=_{CL}$ to behave as an equality, we require it to be reflexive, symmetric and transitive.

(Refl) $P =_{CL} P$

(Symm) $P =_{CL} Q$ implies $Q =_{CL} P$

(Transitivity) $P =_{CL} Q$ and $Q =_{CL} R$ implies $P =_{CL} R$

And $=_{CL}$ is also posited to be a congruence with respect to the constructs of CL (i.e., $=_{CL}$ is preserved both in the operator and argument position):

(Op) $P =_{CL} Q$ implies $PR =_{CL} QR$

(Arg) $P =_{CL} Q$ implies $RP =_{CL} RQ$

Exercise: Define a function `wnf: comb -> comb` that implements simplification of CL terms by orienting the equations (I,K,S) left to right, and incorporates the recursive calls in (Op) and (Arg). Take a leftmost-outermost approach to apply the equational rules.

Exercise: Prove that the combinator **I** is redundant — i.e., for all CL terms P, $(\mathbf{SK}\mathbf{K})P =_{CL} P =_{CL} \mathbf{I}P$

$SKKP$

Substitution

Since CL has no bound and binding occurrences of variables, the notion of substitution is very simple (and coincides with the treatment we gave earlier for abstract syntax trees), i.e., the unique homomorphic extension of a valuation from variables to CL terms.

Let $P, R \in CL$ and let x be a specified variable. The following table shows the definition of substitution as a recursive function (i.e., the unique homomorphic extension of the first case). The second case, i.e., when $x \notin vars(P)$, covers all the subcases of the combinatory constants **I**, **K**, **S** as well as variables $y \not\equiv x$

P	$P[R/x]$	**Condition**
x	R	
P	P	$x \notin vars(P)$
P_1P_2	$P_1[R/x]\ P_2[R/x]$	

Exercise: Define a function `substc: comb -> comb -> string -> comb` that implements substitution on CL terms, i.e., $P[R/x]$.

CL is as expressive as the λ-calculus

Lemma: For every $P \in CL$ and variable x, there exists a $Q \in CL$ with $x \notin vars(Q)$, such that for all $R \in CL$, $QR \ =_{CL} \ P[R/x]$.
That is, we can abstract all occurrences of variable x in P, and obtain a term Q which when applied to any argument, behaves the same (in terms of $=_{CL}$) as $P[R/x]$.
Proof: Define a meta-operation of abstraction $[x]P$ wrt a given variable x of P

P	$[x]P$	**Condition**
x	$\mathbf{I}$	
P	$\mathbf{K}P$	$x \notin vars(P)$
P_1P_2	$\mathbf{S}\ ([x]P_1)\ ([x]P_2)$	

Note how this meta-operation introduces the combinators **I**, **K**, **S** in *operator* positions. Define $Q \ \equiv \ [x]P$.
We verify that $([x]P)R \ =_{CL} \ P[R/x]$, using induction on the structure of P and case analysis:

- Case $P \equiv x$: $([x]x)R \equiv \mathbf{I}R \ =_{CL} \ R \equiv x[R/x]$
- Case $x \notin vars(P)$: $([x]P)R \equiv (\mathbf{K}P)R \ =_{CL} \ P \equiv P[R/x]$ (because $x \notin vars(P)$).
- Case $P \equiv P_1P_2$: $([x](P_1P_2))R \equiv (\mathbf{S}([x]P_1)([x]P_2)R \ =_{CL} \ (([x]P_1)R)(([x]P_2)R)$ $=_{CL} \ (P_1[R/x])(P_2[R/x]) \equiv (P_1P_2)[R/x]$ (by IH applied on P_1, P_2 respectively).

Exercise: Define a function `abstc: comb -> string -> comb` that implements abstraction $[x]P$ over a variable in CL terms .

Translating from λ-calculus to CL

We define the translation $\lceil e \rceil$ from λ-calculus terms e to CL terms, based on the inductive structure of the λ-calculus expressions:

e	$\lceil e \rceil$
x	x
$\lambda x . e_1$	$[x](\lceil e_1 \rceil)$
$(e_1\ e_2)$	$(\lceil e_1 \rceil)\ (\lceil e_2 \rceil)$

Exercise: Define a function `trans2CL: exp -> comb` that implements the translation $\lceil e \rceil$ from the λ-calculus to CL terms.

Translating from CL to λ-calculus

We define the translation $\lfloor P \rfloor$ from CL terms P to λ-calculus terms, based on the inductive structure of the CL terms in the table below.

P	$\lfloor P \rfloor$
x	x
$\mathbf{I}$	$\lambda x \, . \, x$
$\mathbf{K}$	$\lambda x \, . \, \lambda y \, . \, x$
$\mathbf{S}$	$\lambda x \, . \, \lambda y \, . \, \lambda z \, . \, ((x\ z)\ (y\ z))$
$P_1 P_2$	$(\lfloor P_1 \rfloor)\ (\lfloor P_2 \rfloor)$

Exercise: Define a function `trans2LC: comb -> exp` that implements translation $\lfloor P \rfloor$ from CL terms to the λ-calculus .