Lecture 11
Theory of Combinators

- Combinators are an alternative theory of functions to the $\lambda$-calculus

- Originally introduced by logicians as a way of studying the process of substitution

- More recently, Turner has argued that combiners provide a good ‘machine code’ into which functional programs can be compiled

- Several experimental computers have been built based on Turner’s ideas

- Combinators also provide a good intermediate code for conventional machines
  - several of the best compilers for functional languages are based on them
Formulations of theory of combinator

- Two equivalent ways of formulating the theory of combiners:
  
  (i) within the $\lambda$-calculus, or

  (ii) as a completely separate theory.

- approach (i) taken here

- approach (ii) was the original one

- It will be shown that any $\lambda$-expression is equal to an expression built from variables and two particular expressions, $K$ and $S$, using only function application

- This is done by mimicking $\lambda$-abstractions using combinations of $K$ and $S$

- $\beta$-reductions can be simulated by simpler operations involving $K$ and $S$

  - it is these simpler operations that combinator machines implement directly in hardware
S and K

- The definitions of K and S are:
  \[
  \text{LET } K = \lambda x \ y. \ x
  \]
  \[
  \text{LET } S = \lambda f \ g \ x. \ (f \ x) \ (g \ x)
  \]

- By β-reduction, for all \(E_1, E_2\) and \(E_3\):
  \[
  K \ E_1 \ E_2 = E_1
  \]
  \[
  S \ E_1 \ E_2 \ E_3 = (E_1 \ E_3) \ (E_2 \ E_3)
  \]
Combinators

- Any expression built by application (i.e. combination) from \( K \) and \( S \) is called a \textit{combinator}
  - \( K \) and \( S \) are the \textit{primitive combinators}

- Combinators have the following syntax:
  \[
  \langle \text{combinator} \rangle \ ::= \ K \ |
  S \ |
  (\langle \text{combinator} \rangle \ \langle \text{combinator} \rangle)
  \]

- A \textit{combinatory expression} is an expression built from \( K \), \( S \) and zero or more variables
  - a combinator is a combinatory expression not containing variables

- Syntax of combinatory expressions:
  \[
  \langle \text{combinatory expression} \rangle
  ::= \ K \ |
  S \\
  | \ \langle \text{variable} \rangle \\
  | \ (\langle \text{combinatory expression} \rangle \ \langle \text{combinatory expression} \rangle)
  \]
The identity combinator $I$

- The identity function $I$ is often taken as a primitive combinator, but this is not necessary as it can be defined from $K$ and $S$

- Define $I$ by:

\[
\text{LET } I = \lambda x. x
\]

- Then $I = S \ K \ K$
  - exercise!
Combinator reduction

- If $E$ and $E'$ are combinatory expressions then $E \xrightarrow{c} E'$ means:
  - $E \equiv E'$
  - or $E'$ can be got from $E$ by a sequence of rewritings of the form:
    (i) $K E_1 E_2 \xrightarrow{c} E_1$
    (ii) $S E_1 E_2 E_3 \xrightarrow{c} (E_1 E_3) (E_2 E_3)$
    (iii) $I E \xrightarrow{c} E$

- Example: for any $E$

\[
S K K E \xrightarrow{c} K E (K E) \quad \text{by (ii)} \\
\xrightarrow{c} E \quad \text{by (i)}
\]

- thus (iii) is derivable from (i) and (ii)

- Any sequence of combinatory reductions can be expanded into a sequence of $\beta$-conversions

  - $K E_1 E_2 \rightarrow E_1$
  - $S E_1 E_2 E_3 \rightarrow (E_1 E_3) (E_2 E_3)$
Functional completeness

- Every $\lambda$-expression is equal to some combinatory expression
  - called the functional completeness of combinators
  - basis for compilers for functional languages to the machine code of combinator machines

- Key idea:
  - for variable $V$ and combinatory expression $E$ a combinatory expression $\lambda^*V. E$ will be defined
  - $\lambda^*V. E$ uses $K$ and $S$ to simulate adding ‘$\lambda V$’ to an expression
  - $\lambda^*V. E = \lambda V. E$
Bracket abstraction $\lambda^*V. E$

- If $V$ a variable and $E$ a combinatory expression, then $\lambda^*V. E$ is defined inductively on the structure of $E$ as follows:

  (i) $\lambda^*V. V = I$

  (ii) $\lambda^*V. V' = K V'$ (if $V \neq V'$)

  (iii) $\lambda^*V. C = K C$ (if $C$ is a combinator)

  (iv) $\lambda^*V. (E_1 E_2) = S (\lambda^*V. E_1) (\lambda^*V. E_2)$

- Note that $\lambda^*V. E$ is a combinatory expression not containing $V$

- Example: if $f$ and $x$ are variables and $f \neq x$, then:

\[
\lambda^*x. f \ x = S (\lambda^*x. f) (\lambda^*x. x) = S (K f) I
\]
Proof of functional completeness

• **THEOREM:**
  
  • \((\lambda^* V. E) = \lambda V. E\)

• **PROOF:**
  
  • show \((\lambda^* V. E) \ V = E\)
  
  • follows immediately that \(\lambda V. (\lambda^* V. E) \ V = \lambda V. E\)
  
  • and hence by \(\eta\)-reduction that \(\lambda^* V. E = \lambda V. E\)
Proof that \((\lambda^*V. E) \, V = E\)

• Mathematical induction on the ‘size’ of \(E\):
  
(i) if \(E = V\) then:

\[
(\lambda^*V. E) \, V = I \, V = (\lambda x. \, x) \, V = V = E
\]

(ii) if \(E = V'\) where \(V' \neq V\) then:

\[
(\lambda^*V. E) \, V = K \, V' \, V = (\lambda x\, y. \, x) \, V' \, V = V' = E
\]

(iii) if \(E = C\) where \(C\) is a combinator, then:

\[
(\lambda^*V. E) \, V = K \, C = (\lambda x\, y. \, x) \, C \, V = C = E
\]

(iv) if \(E = (E_1 \, E_2)\) then we can assume by induction that:

\[
(\lambda^*V. \, E_1) \, V = E_1
\]
\[
(\lambda^*V. \, E_2) \, V = E_2
\]

and hence

\[
(\lambda^*V. \, E) \, V = (\lambda^*V. \, (E_1 \, E_2)) \, V
\]
\[
= (S \, (\lambda^*V. \, E_1) \, (\lambda^*V. \, E_2)) \, V
\]
\[
= (\lambda f\, g\, x. \, f\, x\, (g\, x)) \, (\lambda^*V. \, E_1) \, (\lambda^*V. \, E_2) \, V
\]
\[
= (\lambda^*V. \, E_1) \, V \, ((\lambda^*V. \, E_2) \, V)
\]
\[
= E_1 \, E_2 \quad \text{(by induction assumption)}
\]
\[
= E
\]
Translation to combinators

- The notation

\[ \lambda^* V_1 \ V_2 \ \cdots \ V_n \ E \]

is used to mean

\[ \lambda^* V_1. \ \lambda^* V_2. \ \cdots \ \lambda^* V_n. \ E \]

- Define the translation of \( \lambda \)-expression \( E \) to a combinatory expression \( (E)_c \):

  (i) \( (V)_c = V \)

  (ii) \( (E_1 \ E_2)_c = (E_1)_c \ (E_2)_c \)

  (iii) \( (\lambda V. \ E)_c = \lambda^* V. \ (E)_c \)
\[ E = (E)_c \]

- **THEOREM:**
  - for every \( \lambda \)-expression \( E \) we have: \( E = (E)_c \)

- **PROOF:** induction on the size of \( E \)
  1. **(i)** If \( E = V \) then \( (E)_c = (V)_c = V \)
  2. **(ii)** If \( E = (E_1 \ E_2) \) we can assume by induction that
     \[
     E_1 = (E_1)_c \\
     E_2 = (E_2)_c
     \]
     hence
     \[
     (E)_c = (E_1 \ E_2)_c = (E_1)_c \ (E_2)_c = E_1 \ E_2 = E
     \]
  3. **(iii)** If \( E = \lambda V. \ E' \) then we can assume by induction that
     \[
     (E')_c = E'
     \]
     hence
     \[
     (E)_c = (\lambda V. \ E')_c \\
     = \lambda^*V. \ (E')_c \quad \text{(by translation rules)} \\
     = \lambda^*V. \ E' \quad \text{(by induction assumption)} \\
     = \lambda V. \ E' \quad \text{(by previous theorem)} \\
     = E
     \]
Consequences of last theorem

- Every $\lambda$-expression is equal to a $\lambda$-expression built up from $K$ and $S$ and variables by application
  - the class of $\lambda$-expressions $E$ defined by:
    $$ E ::= V \mid K \mid S \mid E_1 E_2 $$
    is equivalent to the full $\lambda$-calculus

- A collection of $n$ combinators $C_1, \ldots, C_n$ is called an $n$-element basis
  - if every $\lambda$-expression $E$ is equal to an expression built from $C_i$s and variables by function applications
  - theorem above shows $K$ and $S$ form a 2-element basis

- There exists a 1-element basis!

  Exercise
  Find a combinator, $\chi$ say, such that any $\lambda$-expression is equal to an expression built from $\chi$ and variables by application.

  Hint: Let $\langle E_1, E_2, E_3 \rangle = \lambda p.\ p\ E_1\ E_2\ E_3$ and consider $\langle K, S, K \rangle\ \langle K, S, K \rangle\ \langle K, S, K \rangle$ and $\langle K, S, K \rangle\ \langle \langle K, S, K \rangle\ \langle K, S, K \rangle \rangle$
Examples

- Part of $Y$:

\[ \lambda^* f. \lambda^* x. f \ (x\ x) \]
\[ = \lambda^* f. (\lambda^* x. f \ (x\ x)) \]
\[ = \lambda^* f. (S (\lambda^* x. f) (\lambda^* x. x\ x)) \]
\[ = \lambda^* f. (S (Kf) (S(\lambda^* x. x) (\lambda^* x. x))) \]
\[ = \lambda^* f. (S (Kf) (S\ S\ I\ I)) \]
\[ = S (\lambda^* f. S (Kf)) (\lambda^* f. S\ S\ I\ I) \]
\[ = S (S (\lambda^* f. S (\lambda^* f. K f)) (K (S I I))) \]
\[ = S (S (K S) (S (\lambda^* f. K) (\lambda^* f. f))) (K (S I I)) \]
\[ = S (S (K S) (S (K K) I)) (K (S I I)) \]

- $Y$:

\[ (Y)_c = (\lambda f. (\lambda x. f(x\ x)) (\lambda x. f(x\ x)))_c \]
\[ = \lambda^* f. (((\lambda x. f(x\ x)) (\lambda x. f(x\ x)))_c \]
\[ = \lambda^* f. (((\lambda x. f(x\ x))_c (\lambda x. f(x\ x)))_c \]
\[ = \lambda^* f. (\lambda^* x. (f(x\ x))_c) (\lambda^* x. (f(x\ x))_c) \]
\[ = \lambda^* f. (\lambda^* x. f(x\ x)) (\lambda^* x. f(x\ x)) \]
\[ = S (\lambda^* f. \lambda^* x. f(x\ x)) (\lambda^* f. \lambda^* x. f(x\ x)) \]
\[ = S(S(S(KS)(S(KK)I))(K(SII))))(S(S(KS)(S(KK)I))(K(SII))) \]
Reduction machines

- Represent combinatory expressions by trees

- Example: \( S (f \ x) (k \ y) \ z \) represented by:

- Such trees are represented as pointer structures in memory
  - special hardware or firmware can then be implemented to transform such trees according to the rules of combinator reduction defining \( \xrightarrow{c} \)
Examples of tree reduction

- The tree:

  ![Tree Diagram]

  Could be transformed to:

  ![Transformed Tree Diagram]

  Using the transformation:

  ![Transformation Diagram]

- Implements: $S E_1 E_2 E_3 \xrightarrow{c} (E_1 E_3) (E_2 E_3)$
Graph reduction

- Tree transformation for $S$ just given duplicates a subtree
  - wastes space
  - a better transformation would be to generate one subtree with two pointers to it:

- Generates a graph rather than a tree
Using combinators for evaluation

- Valid way of reducing $\lambda$-expressions is:
  
  (i) translating to combinators
  
  • i.e. $E \mapsto (E)_c$

  (ii) applying the rewrites

  \[
  K \quad E_1 \quad E_2 \xrightarrow{c} E_1 \\
  S \quad E_1 \quad E_2 \quad E_3 \xrightarrow{c} (E_1 \ E_3) \ (E_2 \ E_3)
  \]

  until no more rewriting is possible

- If $E_1 \longrightarrow E_2$ in the $\lambda$-calculus

  • then not necessarily $(E_1)_c \xrightarrow{c} (E_2)_c$

  • for example, take

  \[
  E_1 = \lambda y. \ (\lambda z. \ y) \ (x \ y) \\
  E_2 = \lambda y. \ y
  \]
A combinatory expression is in combinatory normal form if it contains no subexpressions of the form $K E_1 E_2$ or $S E_1 E_2 E_3$.

Normalization theorem holds for combinatory expressions:
- i.e. always reducing the leftmost combinatory redex will find a combinatory normal form if it exists.

If $E$ is in combinatory normal form, then it does not necessarily follow that it is a $\lambda$-expression in normal form:
- $S K$ is in combinatory normal form, but it contains a $\beta$-redex, namely:

$$(\lambda f. (\lambda g. x. (f \ x \ (g \ x))) \ (\lambda x. y. x))$$
Improving translation to combinators

• Simple λ-expressions can translate to complex combinatory expressions

• To make the ‘code’ executed by reduction machines more compact, various optimizations have been devised

• Let $E$ be a combinatory expression and $x$ a variable not occurring in $E$
  
  • then:
  
  \[
  S (K E) \ I \ x \xrightarrow{c} (K E \ x) (I \ x) \xrightarrow{c} E \ x
  \]

  • hence $S (KE) \ I \ x = E \ x$ (because $E_1 \xrightarrow{c} E_2$ implies $E_1 \longrightarrow E_2$)

  • so by extensionality:
  
  \[
  S (K E) \ I = E
  \]

• Whenever $S (K E) \ I$ is generated
  
  • it can be ‘peephole optimized’ to just $E$
Another optimisation

- Let $E_1$, $E_2$ be combinatory expressions and $x$ a variable not occurring in either of them
  - then:
    $S (K E_1) (K E_2) x \xrightarrow{c} K E_1 x (K E_2) x \xrightarrow{c} E_1 E_2$
  - thus
    $S (K E_1) (K E_2) x = E_1 E_2$
  - now
    $K (E_1 E_2) x \xrightarrow{c} E_1 E_2$
  - hence $K (E_1 E_2) x = E_1 E_2$
  - thus
    $S (K E_1) (K E_2) x = E_1 E_2 = K (E_1 E_2) x$
  - it follows by extensionality that:
    $S (K E_1) (K E_2) = K (E_1 E_2)$

- Whenever $S (K E_1) (K E_2)$ is generated
  - it can be optimized to $K (E_1 E_2)$
Example optimisation

- Example: showed earlier that:
  \[
  \lambda^* f. \lambda^* x. f(x \ x) = S (S (K \ S) (S (K \ K \ I))) (K (S \ I \ I))
  \]

- Using the optimization
  \[
  S (K \ E) \ I = E
  \]

- This simplifies to:
  \[
  \lambda^* f. \lambda^* x. f(x \ x) = S (S (K \ S) K) (K (S \ I \ I))
  \]
More combinators

- Easy to recognize applicability of optimization
  \( S (\mathsf{K} E) \mathsf{I} = E \) if \( \mathsf{I} \) has not been expanded to \( S \mathsf{K} \mathsf{K} \)
    - i.e. if \( \mathsf{I} \) is taken as a primitive combinator

- Other combinators similarly useful

- Define \( \mathsf{B} \) and \( \mathsf{C} \) by:

  \[
  \text{LET } \mathsf{B} = \lambda f \ g \ x. \ f \ (g \ x) \\
  \text{LET } \mathsf{C} = \lambda f \ g \ x. \ f \ x \ g
  \]

- These have the following reduction rules:

  \[
  \begin{align*}
  \mathsf{B} \ E_1 \ E_2 \ E_3 \xrightarrow{c} & E_1 \ (E_2 \ E_3) \\
  \mathsf{C} \ E_1 \ E_2 \ E_3 \xrightarrow{c} & E_1 \ E_3 \ E_2
  \end{align*}
  \]

- It follows that:

  \[
  \begin{align*}
  S \ (\mathsf{K} \ E_1) \ E_2 & = \mathsf{B} \ E_1 \ E_2 \\
  S \ E_1 \ (\mathsf{K} \ E_2) & = \mathsf{C} \ E_1 \ E_2 \\
  (E_1, E_2 \text{ are any two combinatory expressions})
  \end{align*}
  \]
Curry’s algorithm

- Combining the various optimizations yields Curry’s algorithm for translating λ-expressions to combinatory expressions

- Use the definition of $(E)C$

- Whenever an expression of the form $S \ E_1 \ E_2$ is generated, try to apply the following rewrite rules:
  1. $S \ (K \ E_1) \ (K \ E_2) \rightarrow K \ (E_1 \ E_2)$
  2. $S \ (K \ E) \ I \rightarrow E$
  3. $S \ (K \ E_1) \ E_2 \rightarrow B \ E_1 \ E_2$
  4. $S \ E_1 \ (K \ E_2) \rightarrow C \ E_1 \ E_2$

- Always use earliest applicable rule

- $S \ (K \ E_1) \ (K \ E_2)$ is translated to $K \ (E_1 \ E_2)$

- $Y$ is translated to $S \ (C \ B \ (S \ I \ I)) \ (C \ B \ (S \ I \ I))$