Combinatory Logic

April 15, 2002
CS 131: Programming Languages

Syntax

• Pure Combinatory Logic

\[ a, b, c, d ::= K \quad a \text{ constant} \]
\[ | S \quad \text{another constant} \]
\[ | a \ b \quad \text{application} \]

• That’s it!

- Random term: \( K (SKK) KS \)
  \[ = ( (K ((SK)K))K)S \]

One-step Reduction

• The relation \( \rightarrow_{\text{CL}} \) is defined by:

\[
(K \ a) \ b \rightarrow_{\text{CL}} a \\
(S \ a \ b \ c) \rightarrow_{\text{CL}} (a \ c) (b \ c) \\

a \rightarrow_{\text{CL}} a' \quad \frac{a \ b \rightarrow_{\text{CL}} a \ b'}{a \ b \rightarrow_{\text{CL}} a' \ b} \\

b \rightarrow_{\text{CL}} b' \quad \frac{a \ b \rightarrow_{\text{CL}} a \ b'}{a \ b \rightarrow_{\text{CL}} a' \ b} \\

\]

One-step Reduction

• Using the left-associativity of application

\[
K \ a \ b \rightarrow_{\text{CL}} a \\
S \ a \ b \ c \rightarrow_{\text{CL}} (a \ c) (b \ c) \\

K \ a \ b \rightarrow_{\text{CL}} a \quad \frac{a \ b \rightarrow_{\text{CL}} a \ b'}{a \ b \rightarrow_{\text{CL}} a' \ b} \\
S \ a \ b \ c \rightarrow_{\text{CL}} a \ c \quad \frac{a \ b \rightarrow_{\text{CL}} a \ b'}{a \ b \rightarrow_{\text{CL}} a' \ b} \\

\]
Correspondence with \(\lambda\)-Calculus

\[ K \ a \ b \rightarrow_{\text{CL}} a \]

\[ S \ a \ b \ c \rightarrow_{\text{CL}} (a \ c) \ (b \ c) \]

\[
K = \lambda x.\lambda y.\ x
\quad = \lambda x.\ (\lambda y.\ x)
\]

\[
S = \lambda x.\lambda y.\lambda z.\ (xz) \ (yz)
\quad = \lambda x.\ (\lambda y.\ (\lambda z.\ ((xz) \ (yz))))
\]

Exercises

1. What does \(SKKS\) reduce to?

2. And \(S(KK)S\)?

3. How about \(SKKa\)?

Combinatory Completeness

- Claim: For every \(\lambda\)-term, there are terms in combinatory logic with the "same meaning"
  - For example, \(SKK\) acts like the identity function:
    \[ SKKa \rightarrow_{\text{CL}}^* a \]
  - \(SII = S(SKK)(SKK)\) acts like \(\lambda x.\ xx\)
    \[(S(SKK)(SKK)) \ a \rightarrow_{\text{CL}}^* aa\]

- Thus combinatory logic is as powerful as the \(\lambda\)-calculus...even though there are no variables!

Extending CL with variables

\[
a, b, c, d ::= \ x \mid y \mid \ldots \mid K \mid S \mid a \ b
\]

- Typical term: \(K(SKxK)KyS\)
- No bound variables
  - All variables are free
  - Substitution is really easy
- Evaluation rules unchanged.
Bracket Abstraction

- For every extended-CL term $a$ and every variable $x$, there is an extended-CL term abbreviated $[x]a$ such that
  1. $x$ is not free in $[x]a$.
  2. $([x]a) b \rightarrow_{cl^*} [x \rightarrow b]$

- For example, $([x]xx)(SK) \rightarrow_{cl^*} (SK)(SK)$

Examples

- $[x](xx) =$

- $[x](SKx) =$

Bracket Abstraction

$[x]K =$

$[x]S =$

$[x]x =$

$[x]y = (x \neq y)$

$[x](ab) =$

Combinatory Completeness

- We can then translate every $\lambda$-term into an equivalent extended CL-term.

  $CL(x) := x$
  $CL(\lambda x.e) := [x](CL(e))$
  $CL(e_1 e_2) := (CL(e_1))(CL(e_2))$

- Every closed $\lambda$-term translates into a variable-free CL-term.
Examples

\[ CL(\lambda x.\lambda y.x) = \]

\[ CL(\lambda x.\lambda y.y) = \]

Implementing Combinators

• David Turner (1979):
  - Compile programs into combinatory logic
  - In practice, extend S and K with combinators like + and \textit{eq} and \textit{cond}, numeric constants, Y and I, etc.

\[
\text{fact} = S \ (S \ (S \ (K \ \text{cond}) \ (S \ (S \ (K \ \text{eq}) \ (K \ 0)) \ I)) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (K \ 1)) \ (S \ (S \ (K \ \text{times}) \ I) \ (S \ (K \ \text{fact}) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (S \ (S \ (K \ \text{minus}) \ I) \ (K \ 1))))
\]

Graph Reduction

• Nice implementation of call-by-need (lazy evaluation)
  - Evaluates each expression at most once

• Represent terms as \textit{graphs} instead of trees
  - Overwrite sub-graphs with their values
  - Expresses sharing of delayed computations
    • As soon as it’s evaluated once, everyone referring to this computation sees the resulting value.
Potential Advantages

• Resulting program has no variables
  - Don’t have to worry about substitution or environments
• Very simple execution strategy
  - Just a handful of combinators
  - Could even implement S and K in hardware, e.g., SKIM
• Parallel graph reduction possible
  - Processors work on disjoint parts of graphs

Problems

• $\text{CL}(\lambda x. \lambda y. \lambda z. (xz)(yz)) =$
  
  $S (S (KS) (S (KS) (S (KK) (KS)))) (S (S (KS) (S (KS) (S (K K) (KK)))) (S (S (KS) (S (KK) (KK)))) (S (S (KK) (SKK))) (S (S (KK) (KK)))) (S (S (KK) (KK)))) (S (S (KK) (KK)))) (S (S (KK) (KK)))) (S (S (KK) (KK)))) (S (S (KK) (KK))))$ 

  • A better translation would be?
  • In general, translation can cause exponential blowup.

Improvements

1. Add new combinator constants
   - $I \ a \rightarrow_{\text{CL}} a$
   - $B \ a \ b \ c \rightarrow_{\text{CL}} a\ (b\ c)$
   - $C \ a \ b \ c \rightarrow_{\text{CL}} a\ (c\ b)$

2. Improve the translation: $[x]x = I$

3. Apply optimizations to the output
   - $S\ (K\ a)\ (K\ b) \rightarrow K\ (a\ b)$
   - $S\ (K\ a)\ I \rightarrow a$
   - $S\ (K\ a)\ b \rightarrow B\ a\ b$
   - $S\ a\ (K\ b) \rightarrow C\ a\ b$

Other Improvements

• More complex primitive combinators

• Program-specific combinators
  - Any closed lambda term can be made into a new constant

• Avoid graph updates of unshared terms