Computer Science 131
Programming Languages

November 16, 2000
Combinatory Logic
Syntax

• Pure Combinatory Logic

\[ a, b, c, d ::= \begin{align*}
K & \quad \text{a constant} \\
S & \quad \text{another constant} \\
a \ b & \quad \text{application}
\end{align*} \]

• That's it!

  - Random term: \( K (SKK) KS \)
One-step Reduction

- The relation $\rightarrow_{CL}$ is defined by:

\[
(K \ a) \ b \rightarrow_{CL} \ a
\]

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

\[
a \rightarrow_{CL} \ a' \\
\hline
a \ b \rightarrow_{CL} \ a' \ b
\]

\[
b \rightarrow_{CL} \ b' \\
\hline
a \ b \rightarrow_{CL} \ a \ b'
\]
One-step Reduction

- Using the left-associativity of application

\[
\begin{align*}
K & \ a \ b \ \rightarrow_{CL} \ a \\
S & \ a \ b \ c \ \rightarrow_{CL} \ (a \ c) \ (b \ c)
\end{align*}
\]
Correspondence with $\lambda$-Calculus

\[
\begin{align*}
K & \approx \lambda x. \lambda y. x \\
& = \lambda x. (\lambda y. x) \\
S & \approx \lambda x. \lambda y. \lambda z. (xz)(yz) \\
& = \lambda x. (\lambda y. (\lambda z. ((xz)(yz)))
\end{align*}
\]

\[
\begin{align*}
K\ a\ b & \rightarrow_{CL} a \\
S\ a\ b\ c & \rightarrow_{CL} (a\ c)\ (b\ c)
\end{align*}
\]
Exercises

1. What does \texttt{skks} reduce to?

2. And \texttt{s(kk)s}?

3. How about \texttt{skka}? 

4. Put \texttt{i := skk}. 
   How does \texttt{sii(sii)} reduce?
Combinatory Completeness

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

• Thus combinatory logic is as powerful as the $\lambda$-calculus
This slide intentionally left blank
Extending CL with variables

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

- Variables
- A constant
- Another constant
- Application

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

- For every extended-CL term $a$ and every variable $x$, there is an extended-CL term $[x]a$ such that

1. $x$ is not free in $[x]a$.

2. $([x]a)b \rightarrow_{CL}^* a[x\rightarrow b]$.

- For example, $([x]xx)(SK) \rightarrow_{CL}^* (SK)(SK)$. 
Bracket Abstraction

\([x]K = \)

\([x]S = \)

\([x]x = \)

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

\([x](ab) = \)
Examples

• \([x](xx) = \]

• \([x](SKx) = \]
Combinatory Completeness

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

$$\begin{align*}
\text{CL}(x) & := x \\
\text{CL}(\lambda x. e) & := [x](\text{CL}(e)) \\
\text{CL}(e_1 e_2) & := (\text{CL}(e_1))(\text{CL}(e_2))
\end{align*}$$

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

\[ \text{CL}(\lambda x. \lambda y. x) = \]

\[ \text{CL}(\lambda x. \lambda y. y) = \]
Implementing Combinators

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

\[
\text{fact} = S \left( S \left( S \left( K \text{cond} \right) \left( S \left( S \left( K \text{eq} \right) \left( K \ 0 \right) \right) \ I \right) \right) \left( K \ 1 \right) \right) \left( S \left( S \left( K \text{times} \right) \ I \right) \left( S \left( K \text{fact} \right) \left( S \left( S \left( K \text{minus} \right) \ I \right) \left( K \ 1 \right) \right) \right) \right)
\]
Graph Reduction

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

• Represent terms as graphs instead of trees
  - Replace 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.
Claimed 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 easy
  – Processors work on disjoint parts of graphs
Problems

• \( \text{CL}(\lambda x.\lambda y.\lambda z. \ (xz) \ (yz)) = \)

\[
S \ (S \ (KS) \ (S \ (KS) \ (S \ (KK) \ (KS)) \ )) \ (S \ (S \ (KS) \ (S \ (KS) \ (S \ (KK) \ (KK)) \ )) \ (S \ (KK) \ (SKK)) \ )) \ ) \ ) \ ) \ ) \ ) (S \ (KK) \ (K \ (SKK))))
\]

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

1. Add new combinator constants

\[ I \ a \rightarrow_{CL} \ a \]
\[ B \ a \ b \ c \rightarrow_{CL} \ a \ (b \ c) \]
\[ C \ a \ b \ c \rightarrow_{CL} \ (a \ c) \ b \]

2. Improve the translation:

\[ [x]x = I \]

3. Apply optimizations to the output

\[ S \ (K \ a) \ (K \ b) \ == \ K \ (a \ b) \]
\[ S \ (K \ a) \ I \ == \ a \]
\[ S \ (K \ a) \ b \ == \ B \ a \ b \]
\[ S \ a \ (K \ b) \ == \ C \ a \ b \]
Other Improvements

• Even more complicated standard combinators

• Program-specific combinators
  - Any closed lambda term can be thought of as a new constant

• Avoiding graph updates when unshared