Combinatory Logic

April 15, 2002

CS 131: Programming Languages

Syntax

- Pure Combinatory Logic

$$
\begin{array}{rcll}
a,b,c,d & ::= & \mathsf{K} & \textit{a constant} \\
 & | & \mathsf{S} & \textit{another constant} \\
 & | & a\ b & \textit{application}
\end{array}
$$

- That's it!
  - Random term: $\mathsf{K(SKK)KS}$
    $= \mathsf{((K((SK)K))K)S}$

One-step Reduction

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

$$\frac{}{(\mathsf{K}\ a)\ b \rightarrow_{\mathsf{CL}} a}$$

$$\frac{}{((\mathsf{S}\ a)\ b)\ c \rightarrow_{\mathsf{CL}} (a\ c)(b\ c)}$$

$$\frac{a \rightarrow_{\mathsf{CL}} a'}{a\ b \rightarrow_{\mathsf{CL}} a'\ b} \qquad \frac{b \rightarrow_{\mathsf{CL}} b'}{a\ b \rightarrow_{\mathsf{CL}} a\ b'}$$

One-step Reduction

- Using the left-associativity of application

$$\frac{}{\mathsf{K}\ a\ b \rightarrow_{\mathsf{CL}} a}$$

$$\frac{}{\mathsf{S}\ a\ b\ c \rightarrow_{\mathsf{CL}} (a\ c)(b\ c)}$$

$$\frac{a \rightarrow_{\mathsf{CL}} a'}{a\ b \rightarrow_{\mathsf{CL}} a'\ b} \qquad \frac{b \rightarrow_{\mathsf{CL}} b'}{a\ b \rightarrow_{\mathsf{CL}} a\ b'}$$

Correspondence with λ-Calculus

$$\frac{}{\mathsf{K}\ a\ b \rightarrow_{\mathrm{CL}} a}$$

$$\frac{}{\mathsf{S}\ a\ b\ c \rightarrow_{\mathrm{CL}} (a\ c)(b\ c)}$$

$$\begin{aligned}
\mathsf{K} &\approx \lambda x.\lambda y.x \\
&= \lambda x.(\lambda y.x) \\
\mathsf{S} &\approx \lambda x.\lambda y.\lambda z.(xz)(yz) \\
&= \lambda x.(\lambda y.(\lambda z.((xz)(yz))))
\end{aligned}$$

Exercises

1. What does SKKS reduce to?
2. And S(KK)S ?
3. How about SKKa?

Combinatory Completeness

- Claim: For every λ-term, there are terms in combinatory logic with the "same meaning"
  - For example, SKK acts like the identity function:

    $$\mathsf{SKK}a \rightarrow_{\mathrm{CL}}^{*} a$$

  - SII = S(SKK)(SKK) acts like λx.xx

    $$(\mathsf{S(SKK)(SKK)})a \rightarrow_{\mathrm{CL}}^{*} aa$$

- Thus combinatory logic is as powerful as the λ-calculus...even though there are no variables!

Extending CL with variables

a,b,c,d	::=	x | y | ...	*variables*
	|	K	*a constant*
	|	S	*another constant*
	|	$a\ b$	*application*

- 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. $([x]a)b \to_{CL}^{*} a[x \to b]$

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

Bracket Abstraction

[x]K =

[x]S =

[x]x =

[x]y = (x≠y)

[x](ab) =

Examples

- [x](xx) =

- [x](SKx) =

Combinatory Completeness

- We can then translate every λ-term into an equivalent extended CL-term.

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

- Every *closed* λ-term translates into a variable-free CL-term.

Examples

`CL(λx.λy.x) =`

`CL(λx.λy.y) =`

Implementing Combinators

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

```
fact = S (S (S (K cond) (S (S (K eq) (K 0)) I))
         (K 1)) (S (S (K times) I) (S (K fact)
         (S (S (K minus) I) (K 1))))
```

Graph Reduction

- Nice implementation of call-by-need (lazy evaluation)
  - Evaluates each expression at most once
- Represent terms as *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

- `CL(λx.λy.λz.(xz)(yz)) =`

```
S(S(KS)(S(S(KS)(S(KK)(KS)))(S(S(KS)(S(S(KS)(S(K
K)(KS)))(S(S(KS)(S(KK)(KK)))(S(KK)(SKK)))))(S(K
K)(K(SKK))))))(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(S(
KS)(S(KK)(KK)))(K(SKK)))))(S(KK)(K(SKK))))
```

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

Improvements

1. Add new combinator constants

$$\texttt{I a} \rightarrow_{CL} \texttt{a}$$
$$\texttt{B a b c} \rightarrow_{CL} \texttt{a (b c)}$$
$$\texttt{C a b c} \rightarrow_{CL} \texttt{(a c) b}$$

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

- More complex primitive combinators
- Program-specific combinators
  - Any closed lambda term can be made into a new constant
- Avoid graph updates of unshared terms