Trampolining

Dark Arts Edition Part 2

🔮 🪄

Combinatory Logic, and an aside to GHC

🚫 🔮 🪄 🚫

Motivation

```
double :: [a] -> [a]
double [] = []
double (hd : tl) = hd : hd : double tl
```

Von Neumann Machine

First-order logic

Motivation

```
double :: [a] -> [a]
double [] = []
double (hd : tl) = hd : hd : double tl
```

Von Neumann Machine

First-order logic

Doesn’t know what a “function” is

Recap

Task: Take a list of size 1*e*7, double it, and calculate its length.

```
$ scala-cli run scala-list.sc
// DNF: stack overflow

```

```
$ scala-cli run scala-defunc.sc
// 1240 ms

```

```
$ sbtn nativeLink && ./target/scala-3.3.3/scaladefunc
// defunctionalized version, no code changes
// 6046 ms (237 ms for 1e6)

```

```
$ stack ghc --resolver nightly haskell-list.hs && ./haskell-list
// 67 ms

```

Sources

Discussions with Shardul, Simon, and Viktor.

Several papers, added at the end.

First-order logic, optional recap

A first-order structure:

- A domain / universe U
- A set of relations R
- A set of functions F

From the logic:

- A set of variables V
- Logical connectives $\wedge, \neg$
- Binder $\forall$

Notably, the relations and functions are *not* first-class objects, only elements of the universe are. We may write $f(a)$ but never simply f.

Functions as data: slightly more formal

$$\lambda x.\ \lambda y.\ x\ (y\ z)$$

- Free variable, z
- Bound variables x, y

Can we simply write $f = \lambda x.\ \lambda y.\ x\ (y\ z) \implies f\ a = \lambda y.\ a\ (y\ z)$?

Functions as data: slightly more formal

$$\lambda x.\ \lambda y.\ x\ (y\ z)$$

- Free variable, z
- Bound variables x, y

Can we simply write $f = \lambda x.\ \lambda y.\ x\ (y\ z) \implies f\ a = \lambda y.\ a\ (y\ z)$?

Could we write the LHS as $\forall x.\ \forall y.\ f(x, y) = x(y(z))$?

Combinatory Logic

"Combinatory logic is a notation to eliminate the need for quantified variables in mathematical logic."

Combinatory Logic

"Combinatory logic is a notation to eliminate the need for quantified variables in mathematical logic."

Definition (Baren)
A combinator is a lambda expression which contains no occurrences of a free variable.

Standard example, the S, K, and I combinators:

$$S\ x\ y\ z = x\ z\ (y\ z) \qquad \text{(Verschmelzungsfunktion)}$$

$$K\ x\ y = x \qquad \text{(Konstanzfunktion)}$$

$$I\ x = x \qquad \text{(Identitätsfunktion)}$$

Combinatory Logic

> *"Combinatory logic is a notation to eliminate the need for quantified variables in mathematical logic."*

Definition (Baren)
A combinator is a lambda expression which contains no occurrences of a free variable.

Standard example, the S, K, and I combinators:

$$S\ x\ y\ z = x\ z\ (y\ z) \qquad \text{(Verschmelzungsfunktion)}$$
$$K\ x\ y = x \qquad \text{(Konstanzfunktion)}$$
$$I\ x = x \qquad \text{(Identitätsfunktion)}$$

Older than the lambda calculus, Schönfinkel (1924) and Curry (1930).

‘Compiling’ to combinatory logic

Starting from the innermost abstraction:

$$\begin{aligned}
\lambda x.\ x &\mapsto I \\
\lambda x.\ c &\mapsto K\ c \qquad (x \notin fv(c), bv(c) = \emptyset) \\
\lambda x.\ e_1\ e_2 &\mapsto S\ (\lambda x.\ e_1)\ (\lambda x.\ e_2)
\end{aligned}$$

‘Compiling’ to combinatory logic

Starting from the innermost abstraction:

$$\begin{aligned} \lambda x.\ x &\mapsto I \\ \lambda x.\ c &\mapsto K\ c \qquad (x \notin fv(c), bv(c) = \emptyset) \\ \lambda x.\ e_1\ e_2 &\mapsto S\ (\lambda x.\ e_1)\ (\lambda x.\ e_2) \end{aligned}$$

$$\lambda x.\ \lambda y.\ x\ (y\ z)$$

$$\mapsto$$

‘Compiling’ to combinatory logic

Starting from the innermost abstraction:

$$\lambda x.\ x \mapsto I$$
$$\lambda x.\ c \mapsto K\ c \qquad (x \notin fv(c), bv(c) = \emptyset)$$
$$\lambda x.\ e_1\ e_2 \mapsto S\ (\lambda x.\ e_1)\ (\lambda x.\ e_2)$$

$$\begin{aligned} & \lambda x.\ \lambda y.\ x\ (y\ z) \\ \mapsto\ & \lambda x.\ S\ (\lambda y.x)\ (\lambda y.y\ z) \\ \mapsto\ & \end{aligned}$$

‘Compiling’ to combinatory logic

Starting from the innermost abstraction:

$$\lambda x.\ x \mapsto I$$

$$\lambda x.\ c \mapsto K\ c \qquad (x \notin fv(c), bv(c) = \emptyset)$$

$$\lambda x.\ e_1\ e_2 \mapsto S\ (\lambda x.\ e_1)\ (\lambda x.\ e_2)$$

$$
\begin{aligned}
& \lambda x.\ \lambda y.\ x\ (y\ z) \\
\mapsto\ & \lambda x.\ S\ (\lambda y.x)\ (\lambda y.y\ z) \\
\mapsto\ & \lambda x.\ S\ (K\ x)\ (\lambda y.y\ z) \\
\mapsto\ &
\end{aligned}
$$

‘Compiling’ to combinatory logic

Starting from the innermost abstraction:

$$\begin{aligned}
\lambda x.\ x &\mapsto I \\
\lambda x.\ c &\mapsto K\ c \qquad (x \notin fv(c), bv(c) = \emptyset) \\
\lambda x.\ e_1\ e_2 &\mapsto S\ (\lambda x.\ e_1)\ (\lambda x.\ e_2)
\end{aligned}$$

$$\begin{aligned}
& \lambda x.\ \lambda y.\ x\ (y\ z) \\
\mapsto\ & \lambda x.\ S\ (\lambda y.x)\ (\lambda y.y\ z) \\
\mapsto\ & \lambda x.\ S\ (K\ x)\ (\lambda y.y\ z) \\
\mapsto\ & \lambda x.\ S\ (K\ x)\ (S\ (\lambda y.\ y)(\lambda y.\ z)) \\
\mapsto\ &
\end{aligned}$$

‘Compiling’ to combinatory logic

Starting from the innermost abstraction:

$$\lambda x.\ x \mapsto I$$

$$\lambda x.\ c \mapsto K\ c \qquad (x \notin fv(c), bv(c) = \emptyset)$$

$$\lambda x.\ e_1\ e_2 \mapsto S\ (\lambda x.\ e_1)\ (\lambda x.\ e_2)$$

$$\begin{aligned}
& \lambda x.\ \lambda y.\ x\ (y\ z) \\
\mapsto\ & \lambda x.\ S\ (\lambda y.x)\ (\lambda y.y\ z) \\
\mapsto\ & \lambda x.\ S\ (K\ x)\ (\lambda y.y\ z) \\
\mapsto\ & \lambda x.\ S\ (K\ x)\ (S\ (\lambda y.\ y)(\lambda y.\ z)) \\
\mapsto\ & \lambda x.\ (S\ (K\ x))\ (S\ (I\ (K\ z))) \\
\mapsto\ &
\end{aligned}$$

‘Compiling’ to combinatory logic

Starting from the innermost abstraction:

$$
\begin{aligned}
\lambda x.\ x &\mapsto I \\
\lambda x.\ c &\mapsto K\ c \qquad (x \notin fv(c), bv(c) = \emptyset) \\
\lambda x.\ e_1\ e_2 &\mapsto S\ (\lambda x.\ e_1)\ (\lambda x.\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
& & \lambda x.\ \lambda y.\ x\ (y\ z) \\
&\mapsto & \lambda x.\ S\ (\lambda y.x)\ (\lambda y.y\ z) \\
&\mapsto & \lambda x.\ S\ (K\ x)\ (\lambda y.y\ z) \\
&\mapsto & \lambda x.\ S\ (K\ x)\ (S\ (\lambda y.\ y)(\lambda y.\ z)) \\
&\mapsto & \lambda x.\ (S\ (K\ x))\ (S\ (I\ (K\ z))) \\
&\mapsto & S\ (\lambda x.\ S\ (K\ x))\ (\lambda x.\ S\ (I\ (K\ z))) \\
&\mapsto &
\end{aligned}
$$

‘Compiling’ to combinatory logic

Starting from the innermost abstraction:

$$\lambda x.\ x \mapsto I$$

$$\lambda x.\ c \mapsto K\ c \qquad (x \notin fv(c), bv(c) = \emptyset)$$

$$\lambda x.\ e_1\ e_2 \mapsto S\ (\lambda x.\ e_1)\ (\lambda x.\ e_2)$$

$$\begin{aligned}
& && \lambda x.\ \lambda y.\ x\ (y\ z) \\
&\mapsto && \lambda x.\ S\ (\lambda y.x)\ (\lambda y.y\ z) \\
&\mapsto && \lambda x.\ S\ (K\ x)\ (\lambda y.y\ z) \\
&\mapsto && \lambda x.\ S\ (K\ x)\ (S\ (\lambda y.\ y)(\lambda y.\ z)) \\
&\mapsto && \lambda x.\ (S\ (K\ x))\ (S\ (I\ (K\ z))) \\
&\mapsto && S\ (\lambda x.\ S\ (K\ x))\ (\lambda x.\ S\ (I\ (K\ z))) \\
&\mapsto && S\ ((K\ S)\ ((K\ K)\ I))\ (K\ (S\ (I\ (K\ z))))
\end{aligned}$$

Takeaways

- Eliminated bound variables (yay!)

Takeaways

- Eliminated bound variables (yay!)
- Evaluation rules are like rewrites

Takeaways

- Eliminated bound variables (yay!)
- Evaluation rules are like rewrites
- In particular, cannot partially evaluate

Takeaways

- Eliminated bound variables (yay!)
- Evaluation rules are like rewrites
- In particular, cannot partially evaluate
- But the generated term is huge (possibly cubic)

Takeaways

- Eliminated bound variables (yay!)
- Evaluation rules are like rewrites
- In particular, cannot partially evaluate
- But the generated term is huge (possibly cubic)
- The term we generated was not minimal, but minimal terms are not unique and hard to find

Motivation[1]

$$(\lambda x.\ \lambda y.\ -\ y\ x)\ 3\ 4$$

- We gain much: no intermediate term, one less step

[1] Simon Peyton Jones (1987). The Implementation of Functional Programming Languages.

Supercombinator compilation

Motivation[1]

$$(\lambda x.\ \lambda y.\ -\ y\ x)\ 3\ 4$$

- We gain much: no intermediate term, one less step
- We lose nothing wrt computation: the intermediate term is useless

[1] Simon Peyton Jones (1987). The Implementation of Functional Programming Languages.

Motivation[1]

$$(\lambda x.\ \lambda y.\ -\ \ y\ x)\ 3\ 4$$

- We gain much: no intermediate term, one less step
- We lose nothing wrt computation: the intermediate term is useless
- Issue: our computation model now needs to accommodate arbitrary arity reductions

[1] Simon Peyton Jones (1987). The Implementation of Functional Programming Languages.

Supercombinators

Solution: arbitrary arity reductions *are* the computation model.

Definition
A supercombinator $\$S$ of arity n is a lambda expression of the form:

$$\lambda x_1.\ \lambda x_2.\ \ldots \lambda x_n.\ E$$

where

- E is not a lambda abstraction
- any lambda abstraction in E is a supercombinator itself
- $n \geq 0$

Solution: arbitrary arity reductions *are* the computation model.

Definition
A supercombinator $\$S$ of arity n is a lambda expression of the form:

$$\lambda x_1.\ \lambda x_2.\ \ldots \lambda x_n.\ E$$

where

- E is not a lambda abstraction
- any lambda abstraction in E is a supercombinator itself
- $n \geq 0$

Each supercombinator is paired with a *reduction* acting on a redex. A *supercombinator redex* is a fully applied supercombinator.

Compilation

$$\$XY = \lambda x.\ \lambda y.\ -\ \ y\ x$$

Compilation

$$\$XY = \lambda x.\ \lambda y.\ -\ \ y\ x$$

$$\$XY\ x\ y = -\ y\ x$$

Compilation

$$\$XY = \lambda x.\ \lambda y.\ -\ y\ x$$

$$\$XY\ x\ y = -\ y\ x$$

Rewrite rules:

$\$XY\ x\ y = -\ y\ x$

Expression to be evaluated:

$\$XY\ 3\ 4$

Compilation: Example

$$(\lambda x.\,(\lambda y.\ +\ y\ x)\ x)\ 4$$

Notably, no supercombinators right now.

Compilation: Example

$$(\lambda x.\,(\lambda y.\ +\ y\ x)\ x)\ 4$$

Notably, no supercombinators right now.

```
$YX y x = + y x
$X x = $YX x x
---------------
$X 4
```

Recursion, case analysis

$$\frac{\$L\ \ xs = \text{if}\ \ (=\ \ xs\ \ nil)\ \text{then}\ \ 0\ \text{else}\ \ 1 + (\$L\ \ (tail\ \ xs))}{\$L\ \ (cons\ \ 1\ \ (cons\ \ 2\ \ (cons\ \ 3\ \ nil)))}$$

2

[2]Treatment of recursion from Johnsson, Augustsson (1985).

Combinator Families

Schönfinkel's combinators $\{S, K, I, B, C\}$ where:

$$B\ x\ y\ z = x\ (y\ z)$$
$$C\ x\ y\ z = x\ z\ y$$

Combinator Families

Schönfinkel's combinators $\{S, K, I, B, C\}$ where:

$$B\ x\ y\ z = x\ (y\ z)$$
$$C\ x\ y\ z = x\ z\ y$$

Define a family Φ indexed by trees with words as leaves. Here $w \in \{b, c\}^*$.

$$\begin{aligned}
\Phi_\epsilon &= I \\
\Phi_{b \cdot w}\ x\ x_1 \ldots x_{|w|+1} &= x\ (\Phi_w\ x_1\ \ldots\ x_{|w|+1}) \\
\Phi_{c \cdot w}\ x\ x_1 \ldots x_{|w|+1} &= (\Phi_w\ x_1\ \ldots\ x_{|w|+1})\ x \\
\Phi_{(t_1, t_2)}\ x_1 \ldots x_{|t_1|}\ y_1\ \ldots\ y_{|t_2|}\ z &= (\Phi_{t_1}\ \bar{x}\ z)(\Phi_{t_2}\ \bar{y}\ z)
\end{aligned}$$

Combinatory logic in GHC

- Was *maybe* used as the foundation earlier?

- Was *maybe* used as the foundation earlier?
- Definitely switched by 2004
  Simon Marlow and Simon Peyton Jones. **“Making a fast curry: push/enter vs. eval/apply for higher-order languages.”** In: *SIGPLAN Not.* 39.9 (Sept. 2004), pp. 4–15. ISSN: 0362-1340. DOI: `10.1145/1016848.1016856`. URL: `https://doi.org/10.1145/1016848.1016856`

Combinatory logic in GHC

- Was *maybe* used as the foundation earlier?
- Definitely switched by 2004
  Simon Marlow and Simon Peyton Jones. **"Making a fast curry: push/enter vs. eval/apply for higher-order languages."** In: *SIGPLAN Not.* 39.9 (Sept. 2004), pp. 4–15. ISSN: 0362-1340. DOI: `10.1145/1016848.1016856`. URL: `https://doi.org/10.1145/1016848.1016856`
- But the core ideas remain

Combinatory logic in GHC

- Was *maybe* used as the foundation earlier?
- Definitely switched by 2004
  Simon Marlow and Simon Peyton Jones. **"Making a fast curry: push/enter vs. eval/apply for higher-order languages."** In: *SIGPLAN Not.* 39.9 (Sept. 2004), pp. 4–15. ISSN: 0362-1340. DOI: `10.1145/1016848.1016856`. URL: `https://doi.org/10.1145/1016848.1016856`
- But the core ideas remain
- This is why the GHC assembly *looked* familiar before but was still a bit alien

Combinatory logic in GHC

- Was *maybe* used as the foundation earlier?
- Definitely switched by 2004
  Simon Marlow and Simon Peyton Jones. **"Making a fast curry: push/enter vs. eval/apply for higher-order languages."** In: *SIGPLAN Not.* 39.9 (Sept. 2004), pp. 4–15. ISSN: 0362-1340. DOI: `10.1145/1016848.1016856`. URL: `https://doi.org/10.1145/1016848.1016856`
- But the core ideas remain
- This is why the GHC assembly *looked* familiar before but was still a bit alien
- I understand it a bit more now

Combinatory logic in GHC

- Was *maybe* used as the foundation earlier?
- Definitely switched by 2004
  Simon Marlow and Simon Peyton Jones. **"Making a fast curry: push/enter vs. eval/apply for higher-order languages."** In: *SIGPLAN Not.* 39.9 (Sept. 2004), pp. 4–15. ISSN: 0362-1340. DOI: `10.1145/1016848.1016856`. URL: `https://doi.org/10.1145/1016848.1016856`
- But the core ideas remain
- This is why the GHC assembly *looked* familiar before but was still a bit alien
- I understand it a bit more now
- You can spot blocks of "rewrites"

Combinatory logic in GHC

- Was *maybe* used as the foundation earlier?
- Definitely switched by 2004
  Simon Marlow and Simon Peyton Jones. **"Making a fast curry: push/enter vs. eval/apply for higher-order languages."** In: *SIGPLAN Not.* 39.9 (Sept. 2004), pp. 4–15. ISSN: 0362-1340. DOI: `10.1145/1016848.1016856`. URL: `https://doi.org/10.1145/1016848.1016856`
- But the core ideas remain
- This is why the GHC assembly *looked* familiar before but was still a bit alien
- I understand it a bit more now
- You can spot blocks of "rewrites"
- Every piece of data is a function, and every function is data

On history of combinators and of Schönfinkel

Stephen Wolfram (2020), ”Combinators and the Story of Computation,” Stephen Wolfram Writings. `https://writings.stephenwolfram.com/2020/12/combinators-and-the-story-of-computation`.

Stephen Wolfram (2020), ”Where Did Combinators Come From? Hunting the Story of Moses Schönfinkel,” Stephen Wolfram Writings. `https://writings.stephenwolfram.com/2020/12/where-did-combinators-come-from-hunting-the-story-of-moses-schon`

Stephen Wolfram (2021), ”A Little Closer to Finding What Became of Moses Schönfinkel, Inventor of Combinators,” Stephen Wolfram Writings. `https://writings.stephenwolfram.com/2021/03/a-little-closer-to-finding-what-became-of-moses-schonfinkel-inve`

In interpretations of intuitionistic logic

nLab authors. *combinatory logic.* `https://ncatlab.org/nlab/show/combinatory+logic`. Revision 13. June 2024

nLab authors. *partial combinatory algebra.* `https://ncatlab.org/nlab/show/partial+combinatory+algebra`. Revision 25. June 2024

nLab authors. *realizability topos.* `https://ncatlab.org/nlab/show/realizability+topos`. Revision 23. June 2024

Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Chapters also by: Philip Wadler, Programming Research Group, Oxford; Peter Hancock, Metier Management Systems, Ltd.; David Turner, University of Kent, Canterbury. Prentice Hall Internaltional (UK) Ltd., Apr. 1987. URL: `https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages-2/`

David A Turner. "A new implementation technique for applicative languages." In: *Software: Practice and Experience* 9.1 (1979), pp. 31–49

Sabine Broda and Luís Damas. "Compact bracket abstraction in combinatory logic." In: *Journal of Symbolic Logic* 62.3 (1997), pp. 729–740. DOI: `10.2307/2275570`

Implementation details

SASL:

David A Turner. **"A new implementation technique for applicative languages."** In: *Software: Practice and Experience* 9.1 (1979), pp. 31–49

Eventually became the G-Machine:

Lennart Augustsson and Thomas Johnsson. **"Parallel graph reduction with the (v , G)-machine."** In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 202–213. ISBN: 0897913280. DOI: `10.1145/99370.99386`. URL: `https://doi.org/10.1145/99370.99386`

Peyton Jones, Simon L, and Simon Peyton Jones. **"Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine."** In: *Journal of Functional Programming* 2 (July 1992), pp. 127–202. URL: `https://www.microsoft.com/en-us/research/publication/implementing-lazy-functional-languages-on-stock-hardware-the-spineless-tagless-`

On comparison of implementation models

Simon Marlow and Simon Peyton Jones. "**Making a fast curry: push/enter vs. eval/apply for higher-order languages.**" In: *SIGPLAN Not.* 39.9 (Sept. 2004), pp. 4–15. ISSN: 0362-1340. DOI: `10.1145/1016848.1016856`. URL: `https://doi.org/10.1145/1016848.1016856`

Rémi Douence and Pascal Fradet. "**A systematic study of functional language implementations.**" In: *ACM Trans. Program. Lang. Syst.* 20.2 (Mar. 1998), pp. 344–387. ISSN: 0164-0925. DOI: `10.1145/276393.276397`. URL: `https://doi.org/10.1145/276393.276397`

Thanks! Questions?

A summary:

- SKI combinators
- Supercombinator compilation
  - Lambda-lifting
  - Recursion
  - Combinator families
- A look at remnants in GHC

References

[1] Simon Marlow and Simon Peyton Jones. **"Making a fast curry: push/enter vs. eval/apply for higher-order languages."** In: *SIGPLAN Not.* 39.9 (Sept. 2004), pp. 4–15. ISSN: 0362-1340. DOI: 10.1145/1016848.1016856. URL: https://doi.org/10.1145/1016848.1016856.

[2] nLab authors. ***combinatory logic.*** https://ncatlab.org/nlab/show/combinatory+logic. Revision 13. June 2024.

[3] nLab authors. ***partial combinatory algebra.*** https://ncatlab.org/nlab/show/partial+combinatory+algebra. Revision 25. June 2024.

[4] nLab authors. ***realizability topos.*** https://ncatlab.org/nlab/show/realizability+topos. Revision 23. June 2024.

[5] Simon Peyton Jones. ***The Implementation of Functional Programming Languages.*** Chapters also by: Philip Wadler, Programming Research Group, Oxford;

Peter Hancock, Metier Management Systems, Ltd.; David Turner, University of Kent, Canterbury. Prentice Hall Internaltional (UK) Ltd., Apr. 1987. URL: `https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages-2/`.

[6] David A Turner. **"A new implementation technique for applicative languages."** In: *Software: Practice and Experience* 9.1 (1979), pp. 31–49.

[7] Sabine Broda and Luís Damas. **"Compact bracket abstraction in combinatory logic."** In: *Journal of Symbolic Logic* 62.3 (1997), pp. 729–740. DOI: `10.2307/2275570`.

[8] Lennart Augustsson and Thomas Johnsson. **"Parallel graph reduction with the (v , G)-machine."** In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 202–213. ISBN: 0897913280. DOI: `10.1145/99370.99386`. URL: `https://doi.org/10.1145/99370.99386`.

[9] Peyton Jones, Simon L, and Simon Peyton Jones. "Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine." In: *Journal of Functional Programming* 2 (July 1992), pp. 127–202. URL: `https://www.microsoft.com/en-us/research/publication/implementing-lazy-functional-languages-on-stock-hardware-the-spineless-tagless-g-machine/`.

[10] Rémi Douence and Pascal Fradet. "A systematic study of functional language implementations." In: *ACM Trans. Program. Lang. Syst.* 20.2 (Mar. 1998), pp. 344–387. ISSN: 0164-0925. DOI: `10.1145/276393.276397`. URL: `https://doi.org/10.1145/276393.276397`.