3.36pt

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Applied Functional Programming

USCS 2017

Jan Rochel and Andres Löh and Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

Aug 27-31, 2018

B1. Haskell and the λ-Calculus

An Example

Program definition:

$$
\begin{aligned}
&\mathsf{main} && = \mathsf{print}\ (\mathsf{gcd}\ 15\ 12)\\
&\mathsf{print}\ \mathsf{x} && = \mathsf{putStrLn}\ (\mathsf{show}\ \mathsf{x})\\
&\mathsf{gcd}\ \mathsf{x}\ \mathsf{y} && = \mathsf{gcd}'\ (\mathsf{abs}\ \mathsf{x})\ (\mathsf{abs}\ \mathsf{y})\\
&\mathsf{gcd}'\ \mathsf{x}\ 0 && = \mathsf{x}\\
&\mathsf{gcd}'\ \mathsf{x}\ \mathsf{y} && = \mathsf{gcd}'\ \mathsf{y}\ (\mathsf{rem}\ \mathsf{x}\ \mathsf{y})\\
&\ldots
\end{aligned}
$$

Evaluation (naive):

$$
\begin{aligned}
\mathsf{main} &\rightarrow \mathsf{print}\ (\mathsf{gcd}\ 15\ 12)\\
&\rightarrow \mathsf{putStrLn}\ (\mathsf{show}\ (\mathsf{gcd}\ 15\ 12))\\
&\rightarrow \mathsf{putStrLn}\ (\mathsf{show}\ (\mathsf{gcd}'\ (\mathsf{abs}\ 15)\ (\mathsf{abs}\ 12)))\\
&\rightarrow \ldots\\
&\rightarrow 3
\end{aligned}
$$

Term Rewriting

Definition: A **term rewriting system** (TRS) consists of a

- signature Σ: function symbols $\{F, G, \dots\}$ of fixed arity
- set of variables $V = \{a, b, c, \dots\}$
- set of **terms** $Ter(\Sigma)$ over Σ and V.
  Example: $F(a, G(G(b, c), d), H)$
- set **rewriting rules** of the form $l \to r$ with $l, r \in Ter(\Sigma)$
  constraint: variables in r must also occur in l

Example as a TRS

Rewrite rules:

$$\begin{array}{lcl} \mathsf{Main} & \rightarrow & \mathsf{Print}\ (\mathsf{Gcd}\ (15, 12)) \\ \mathsf{Print}\ (\mathsf{x}) & \rightarrow & \mathsf{PutStrLn}\ (\mathsf{Show}\ (\mathsf{x})) \\ \mathsf{Gcd}\ (\mathsf{x}, \mathsf{y}) & \rightarrow & \mathsf{Gcd}'\ (\mathsf{Abs}\ (\mathsf{x}), \mathsf{Abs}\ (\mathsf{y})) \\ \mathsf{Gcd}'\ (\mathsf{x}, \mathsf{y}) & \rightarrow & \ldots \\ \mathsf{Abs}\ (\mathsf{x}) & \rightarrow & \ldots \end{array}$$

A **reduction** (also: **rewriting sequence**) to a normal form:

$$\begin{array}{lcl} \mathsf{Main} & \rightarrow & \mathsf{Print}\ (\mathsf{Gcd}\ (15, 12)) \\ & \rightarrow & \mathsf{PutStrLn}\ (\mathsf{Show}\ (\mathsf{Gcd}\ (15, 12))) \\ & \rightarrow & \mathsf{PutStrLn}\ (\mathsf{Show}\ (\mathsf{Gcd}'\ (\mathsf{Abs}\ (15), \mathsf{Abs}\ (12)))) \\ & \rightarrow & \ldots \\ & \rightarrow & 3 \end{array}$$

Some Terminology and Notation in Rewriting

- **reducible expression** (redex): a term that matches the left-hand side of a rewriting rule
- reduction/rewriting step: application of a rule to a redex.
  Main $\rightarrow$ Print (gcd $(15, 12)$)
  Print (gcd $(15, 12)$) $\leftarrow$ Main
  Main $\rightarrow^*$ PutStrLn (Show (Gcd′ (Abs (15), Abs (12))))
- normal form: term that does not contain a redex
- strong normalisation: every reduction sequence is finite
- unique normalisation: strong normalisation to a unique normal form

Literature: *Term Rewriting Systems* by Terese

Higher-Order Functions

Program definition in Haskell:

$$
\begin{aligned}
\mathsf{main} &= \mathsf{print}\ (\mathsf{flip}\ \mathsf{map}\ [1\,..]\ \mathsf{inc})\\
\mathsf{print}\ \mathsf{x} &= \mathsf{putStrLn}\ (\mathsf{show}\ \mathsf{x})\\
\mathsf{flip}\ \mathsf{f}\ \mathsf{x}\ \mathsf{y} &= \mathsf{f}\ \mathsf{y}\ \mathsf{x}\\
\mathsf{inc}\ \mathsf{x} &= \mathsf{x} + 1\\
\mathsf{map} &= \ldots
\end{aligned}
$$

As a rewriting system (attempt):

$$
\begin{aligned}
\mathsf{Main} &\rightarrow \mathsf{Print}\ (\mathsf{Flip}\ (\mathsf{Map}, [1\,..], \mathsf{Inc}))\\
\mathsf{Print}\ (\mathsf{x}) &\rightarrow \mathsf{PutStrLn}\ (\mathsf{Show}\ (\mathsf{x}))\\
\mathsf{Flip}\ (\mathsf{f}, \mathsf{x}, \mathsf{y}) &\rightarrow \mathsf{f}\ (\mathsf{y}, \mathsf{x})\\
\mathsf{Inc}\ (\mathsf{x}) &\rightarrow \mathsf{x} + 1\\
\mathsf{Map}\ (\mathsf{f}, \mathsf{xs}) &\rightarrow \ldots
\end{aligned}
$$

Problem: higher-order functions require partial application

The λ-Calculus

- introduced by Church in 1932
- rewriting system and simplistic programming language
- supports higher-order functions naturally
- Turing complete

λ-Calculus: A Higher-Order Function

In Haskell: flip f x y = f y x

Desired behaviour: flip a b c $\rightarrow^*$ a c b

In λ-calculus:

$$\begin{array}{ll} & (\lambda f\ x\ y.\ f\ y\ x)\ a\ b\ c \\ \rightarrow & (\lambda x\ y.\ a\ y\ x)\ b\ c \\ \rightarrow & (\lambda y.\ a\ y\ b)\ c \\ \rightarrow & a\ c\ b \end{array}$$

Properties:

- arguments are consumed one by one
- functions are gradually destroyed when applied
- function definitions do not live in a separate space

λ-Calculus: Grammar

λ-terms are of the form:

$$
\begin{array}{lll}
e ::= x & & \text{(variables)} \\
\quad\mid & e\ e & \text{(application)} \\
\quad\mid & \lambda x.\ e & \text{(abstraction)}
\end{array}
$$

Examples:

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

- application associates to the left: $a\ b\ c = (a\ b)\ c$
- Observation: only unary functions and unary application

λ-Calculus: flip

$$\text{flip f x y} = \text{f y x}$$

$$
\begin{array}{ll}
 & (\lambda f\ x\ y.\ f\ y\ x)\ a\ b\ c \\
\rightarrow & (\lambda x\ y.\ a\ y\ x)\ b\ c \\
\rightarrow & (\lambda y.\ a\ y\ b)\ c \\
\rightarrow & a\ c\ b
\end{array}
$$

Representation with unary functions:

$$
\begin{array}{ll}
 & (\lambda f.\ \lambda x.\ \lambda y.\ f\ y\ x)\ a\ b\ c \\
\rightarrow & (\lambda x.\ \lambda y.\ a\ y\ x)\ b\ c \\
\rightarrow & (\lambda y.\ a\ y\ b)\ c \\
\rightarrow & a\ c\ b
\end{array}
$$

λ-Calculus: β-Reduction

A term of the form $\lambda x.\, e$ is called an **abstraction** or **lambda binding**; e is called the abstraction's **body**.

The central rewrite rule of the λ-calculus is β-reduction:

$$(\lambda x.\, e)\ a \ \rightarrow_\beta \ e\,[x \mapsto a]$$

$[x \mapsto a]$:= substitution of all *free* occurrences of variable x by a

$$\begin{aligned}
& (\lambda f.\, \lambda x.\, \lambda y.\, f\ y\ x)\ a\ b\ c \\
\rightarrow_\beta\ & (\lambda x.\, \lambda y.\, a\ y\ x)\ b\ c \\
\rightarrow_\beta\ & (\lambda y.\, a\ y\ b)\ c \\
\rightarrow_\beta\ & a\ c\ b
\end{aligned}$$

Bound and free variables

- An abstraction $\lambda x.\, e$ **binds** variable x in its body e.
- An occurrence of a variable that is not bound is called **free**.

Examples:

- x occurs free in $\lambda y.\, y\ (\lambda z.\, x)$
- $(\lambda x.\, x\ z)\ y\ x$ has one bound and one free occurrence of x, therefore $(\lambda x.\, (\lambda x.\, x\ z)\ y\ x)\ a \rightarrow_{\beta} ((\lambda x.\, x\ z)\ y\ a)$

A term without free variables is called a **closed** term.

λ-Calculus: Name Capturing and α-conversion

$$
\begin{aligned}
& \lambda y.\, (\lambda x.\, \lambda y.\, x\ y)\ y \\
\rightarrow_\beta\ & \lambda y.\, ((\lambda y.\, x\ y)\ [x \mapsto y]) \\
=^?\ & \lambda y.\, \lambda y.\, y\ y
\end{aligned}
$$

Problem: y is **captured** by the innermost lambda binding!
$[x \mapsto y]$ must be a **capture-avoiding** substitution which renames the abstraction variable:

$$
\begin{aligned}
\rightarrow_\beta\ & \lambda y.\, ((\lambda y.\, x\ y)\ [x \mapsto y]) \\
\rightarrow_\alpha\ & \lambda y.\, ((\lambda z.\, x\ z)\ [x \mapsto y]) \\
=\ & \lambda y.\, \lambda z.\, y\ z
\end{aligned}
$$

α-conversion: $\lambda x.\, e \rightarrow_\alpha \lambda y.\, e\ [x \mapsto y]$

λ-Calculus: Convertibility

When are two λ-terms equivalent?

Every rewrite rule r induces a relation on terms $\rightarrow_r$. The equivalence/convertibility relation (symmetric, reflexive, transitive closure) induced by $\rightarrow_r$:

$$=_r \quad := \quad \leftrightarrow_r^* \quad \equiv \quad (\leftarrow_r \cup \rightarrow_r)^* \quad \equiv \quad (\leftarrow_r \cup = \cup \rightarrow_r)^+$$

- $\lambda x.\, \lambda y.\, y\ x =_\alpha \lambda y.\, \lambda z.\, z\ y$
  - $\lambda x.\, \lambda y.\, y\ x \rightarrow_\alpha \lambda y.\, \lambda z.\, z\ y$
- $(\lambda y.\, a\ y)\ b =_\beta (\lambda x.\, x\ b)\ a$
  - $(\lambda y.\, a\ y)\ b \rightarrow_\beta a\ b \leftarrow_\beta (\lambda x.\, x\ b)\ a$
- $(\lambda y.\, \lambda s.\, a\ s\ y)\ b =_{\alpha\beta} \lambda t.\, (\lambda x.\, t\ x\ b)\ a$
  - $(\lambda y.\, \lambda s.\, a\ s\ y)\ b \rightarrow_\beta (\lambda s.\, a\ s\ b) =_\alpha$
    $(\lambda x.\, a\ x\ b) \leftarrow_\beta \lambda t.\, (\lambda x.\, t\ x\ b)\ a$

λ-Calculus: Convertibility and η-Conversion

> $\lambda x.\,(\mathsf{putStrLn} \circ \mathsf{show})\ x \quad \neq_{\alpha\beta} \quad \mathsf{putStrLn} \circ \mathsf{show}$

even though if applied to the same argument they are β-equivalent.

η-conversion: $\lambda x.\, e\ x \ \rightarrow_\eta\ e \qquad$ (x does not occur free in e)

> $(\lambda x.\, e\ x)\ z\ \rightarrow_\beta\ e\ z$
> $(\lambda x.\, e\ x)\ z\ \rightarrow_\eta\ e\ z$

> $\lambda x.\,(\mathsf{putStrLn} \circ \mathsf{show})\ x \ =_{\alpha\beta\eta}\ \mathsf{putStrLn} \circ \mathsf{show}$

$\alpha\beta\eta$-equivalence is one possible criterion for function equivalence. Point-free style programming is essentially the application of η-conversion.

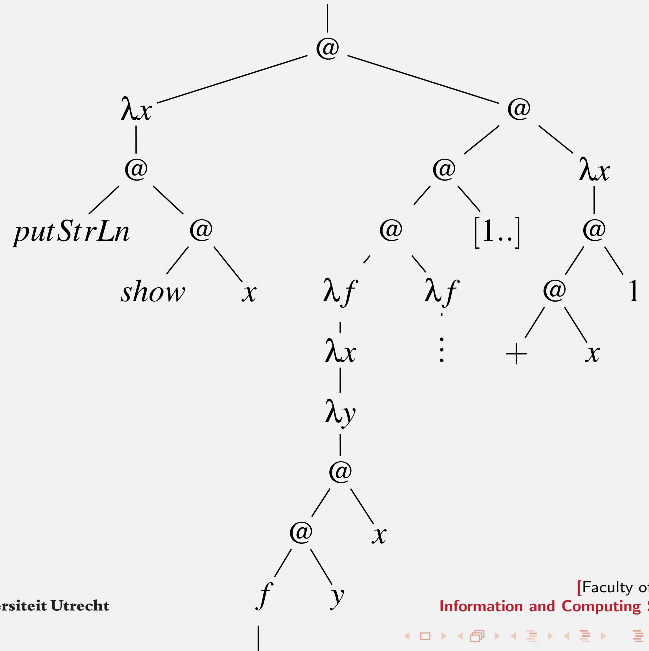
Example

$$\begin{aligned}
\mathsf{main} &= \mathsf{print}\ (\mathsf{flip}\ \mathsf{map}\ [1\,..]\ \mathsf{inc}) \\
\mathsf{print}\ x &= \mathsf{putStrLn}\ (\mathsf{show}\ x) \\
\mathsf{flip}\ f\ x\ y &= f\ y\ x \\
\mathsf{inc}\ x &= x + 1 \\
\mathsf{map}\ f &= \ldots
\end{aligned}$$

$$\begin{aligned}
\mathsf{main} &= \mathsf{print}\ (\mathsf{flip}\ \mathsf{map}\ [1\,..]\ \mathsf{inc}) \\
\mathsf{print} &= \lambda x.\, \mathsf{putStrLn}\ (\mathsf{show}\ x) \\
\mathsf{flip} &= \lambda f.\, \lambda x.\, \lambda y.\, f\ y\ x \\
\mathsf{inc} &= \lambda x.\, x + 1 \\
\mathsf{map} &= \lambda f.\ \ldots
\end{aligned}$$

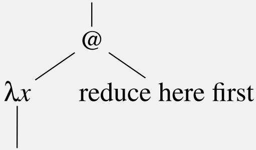
$$\begin{aligned}
&(\lambda x.\, \mathsf{putStrLn}\ (\mathsf{show}\ x))\ ((\lambda f.\, \lambda y.\, \lambda x.\, f\ y\ x) \\
&\quad (\lambda f.\, \lambda x.\ \ldots)\ [1\,..]\ (\lambda x.\, x + 1))
\end{aligned}$$

Example as a Syntax-Tree



Reduction Strategies

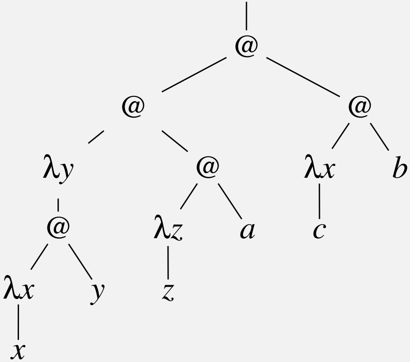
- Strict languages use call-by-value reduction: arguments have to be fully evaluated before a function is applied

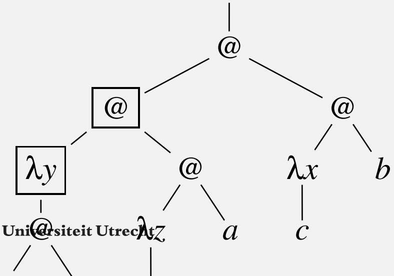


- Non-strict (lazy) evaluation: no reductions take place within the argument of a redex, for instance
- Haskell uses call-by-name reduction: always contract
  - the 'leftmost outermost' redex
  - on the spine (never in arguments)
  - not under lambda

leading to **weak head normal form** (WHNF).

Example: Lazy Evaluation





Simply-Typed λ-calculus

$$
\begin{array}{rcll}
e & ::= & x & \text{variables} \\
 & | & e\ e & \text{application} \\
 & | & \lambda x : t.\ e & \text{lambda abstraction} \\
t & ::= & \tau & \text{type variable} \\
 & | & t \to t & \text{function type}
\end{array}
$$

Function types nest to the right:

$\tau \to \sigma \to \rho = \tau \to (\sigma \to \rho)$

Closed terms are typed as follows (next lecture):

- Every abstraction $\lambda x : \tau.\, e$ assigns a type τ to its variable x. All free occurences of x in e have type τ. If the type of e is σ then $\lambda x : \tau.\, e$ is of type $\tau \to \sigma$.
- In an application f x the function f must have a function type $(\tau \to \sigma)$ and the argument x must have the input type of the function (τ). The type of f x then is σ.

Recursion and Turing Completeness

The simply-typed λ-calculus is strongly normalising
$\Longrightarrow$ A program in simply-typed λ-calculus always terminates.
$\Longrightarrow$ The simply-typed λ-calculus is not Turing complete.

A **fixed-point combinator** is a combinator fix with the property that for any f:

$$\mathsf{fix}\ \mathsf{f} = \mathsf{f}\ (\mathsf{fix}\ \mathsf{f})$$

and thus:

$$\begin{aligned} \mathsf{fix}\ \mathsf{f} &= \mathsf{f}\ (\mathsf{fix}\ \mathsf{f}) \\ &= \mathsf{f}\ (\mathsf{f}\ (\mathsf{f}\ (\mathsf{f} \ldots))) \end{aligned}$$

We say $\mathsf{fix}\ \mathsf{f}$ is a fixed point of f.

Recursion and Turing Completeness

There are many fixed-point combinators in the λ-calculus. One of the smallest and most famous ones is called Y:

$$Y \equiv \lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x))$$

$$\begin{aligned} Y\,f &\rightarrow_\beta (\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)) \\ &\rightarrow_\beta f\,((\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x))) \\ &\leftarrow_{\beta/\eta} f\,((\lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)))\,f) \equiv f\,(Y\,f) \end{aligned}$$

Fixed-point combinators can be used to express recursion:

$$fac = Y\,(\lambda fac.\,\lambda n.\ \textbf{if}\ n == 0\ \textbf{then}\ 1\ \textbf{else}\ n * fac\,(n-1))$$

However: fixed-point combinators are not typeable in the simply-typed λ-calculus! Recursion in Haskell:

$$\textbf{let}\ fac = \lambda n.\ \textbf{if}\ n == 0\ \textbf{then}\ 1\ \textbf{else}\ n * fac\,(n-1)\ \textbf{in}\ fac$$

Haskell vs. the simply-typed λ-Calculus

syntactic sugar	desugars to
operators	functions
function parameters	lambda abstractions
pattern matching	case discrimination
guards	case discrimination
if-**then**-**else**	case discrimination on Bools
list comprehensions	map, concat, filter
do notation	$(\gg=)$ and lambda abstractions
where	**let**
top-level-bindings	**let**
class polymorphism	higher-order functions

Haskell is ..

- λ-calculus
- extended with syntactic sugar
- a very powerful type system
- a type inferencer
- mechanisms to "fill in holes in your program" based on inferred type information

Definitions

The **let** and **where** constructs can be drscribed by lambda expressions

> **let** v = e1 **in** e2 $\rightsquigarrow$ $(\lambda x \circ e2)$ e1

If e1 refers to v then first use Y to remove recursion.

Scott-Encoding of Data Types and Case Discrimination

Haskell: **data** D $= C_1\ x_1 \ldots x_{A_1} \ \mid\ \ldots\ \mid\ C_n\ x_1 \ldots x_{A_n}$

Scott-Enconding: $C_i = \lambda x_1 \ldots \lambda x_{A_i}.\lambda c_1 \ldots \lambda c_n.c_i\ x_1 \ldots x_{A_i}$

```
data [a] = [] | a : [a]
```

```
map f [      ] = []
map f (x : xs) = f x : map f xs
```


Universiteit Utrecht
[Faculty of Science
Information and Computing Sciences]

Scott-Encoding of Data Types and Case Discrimination

Haskell: **data** D $= C_1\ x_1 \ldots x_{A_1} \ \mid\ \ldots\ \mid\ C_n\ x_1 \ldots x_{A_n}$

Scott-Enconding: $C_i = \lambda x_1 \ldots \lambda x_{A_i}.\lambda c_1 \ldots \lambda c_n.c_i\ x_1 \ldots x_{A_i}$

```
data [a] = [] | a : [a]
```

```
map f list = case list of
    [      ] → []
    (x : xs) → f x : map f xs
```

Scott-Encoding of Data Types and Case Discrimination

Haskell: **data** D $= C_1\ x_1 \ldots x_{A_1} \ \mid\ \ldots\ \mid\ C_n\ x_1 \ldots x_{A_n}$

Scott-Enconding: $C_i = \lambda x_1 \ldots \lambda x_{A_i}.\lambda c_1 \ldots \lambda c_n.c_i\ x_1 \ldots x_{A_i}$

```
data List a = Nil | Cons a (List a)
```

```
map f list = case list of
  Nil       → Nil
  Cons x xs → Cons (f x) (map f xs)
```

Scott-Encoding of Data Types and Case Discrimination

Haskell: **data** D $= C_1\ x_1 \ldots x_{A_1}\ \mid\ \ldots\ \mid\ C_n\ x_1 \ldots x_{A_n}$

Scott-Enconding: $C_i = \lambda x_1 \ldots \lambda x_{A_i}.\lambda c_1 \ldots \lambda c_n.c_i\ x_1 \ldots x_{A_i}$

```
Nil  =             λnil. λcons. nil
Cons = λx. λxs. λnil. λcons. cons x xs
```

```
map = λf. λlist. list
   Nil
   (λx. λxs. Cons (f x) (map f xs))
```

Recap

We have seen how most Haskell constructs can be desugared to the lambda calculus:

- constructors of datatypes using the Church encoding,
- non-recursive **let** using lambda abstractions,
- general recursion using a fixed-point combinator,
- pattern matching using possibly nested applications of case functions.

Further examples

$$
\begin{aligned}
\mathsf{pair} &= \lambda x\ y\ f \rightarrow f\ x\ y \\
\mathsf{first} &= \lambda p \rightarrow p\ (\lambda x\ y \rightarrow x) \\
\mathsf{second} &= \lambda p \rightarrow p\ (\lambda x\ y \rightarrow y)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{true} &= \lambda t\ f \rightarrow t \\
\mathsf{false} &= \lambda t\ f \rightarrow f \\
\mathbf{if}\ c\ t\ e &= c\ t\ e
\end{aligned}
$$

Recap – contd.

Many other Haskell constructs can be expressed in terms of the ones we have already seen – for instance:

- **where**-clauses can be transformed into **let**
- **if**-**then**-**else** can be expressed as a function
- list comprehensions can be transformed into applications of map, concat and **if**-**then**-**else**
- monadic **do** notation can be transformed into applications of a limited number of functions

Even Simpler

A straightforward implementation of the lambda calculus may give rise to abitrary large reduction steps. We can represent all lambda expressions using only three combinators:

$$
\begin{aligned}
&\mathsf{S}\ f\ g\ x = f\ x\ (g\ x)\\
&\mathsf{K}\ y\quad x = y\\
&\mathsf{I}\qquad x = x
\end{aligned}
$$

Translation follows the structure of lambda expressions.

$$
\begin{aligned}
&\lambda x.\,x &&\rightsquigarrow \mathsf{I}\\
&\lambda x.\,y &&\rightsquigarrow \mathsf{K}\ y\\
&\lambda x.\,\lambda y.\,e &&\rightsquigarrow \lambda x.\,e' \ \textbf{where}\ \lambda y.\,e \rightsquigarrow e'\\
&\lambda x.\,e1\ e2 &&\rightsquigarrow \lambda x.\,(\lambda x.\,e1)\ x\ ((\lambda x.\,e2)\ x)\\
& &&\rightsquigarrow \mathsf{S}\ (\lambda x.\,e1)\ (\lambda x.\,e2)
\end{aligned}
$$

Graph-reduction

Note that in the application of S the argument x is doubled. By using a graph representation of our expression we only neeed to duplicate the pointers to the expression x.

When we overwrite an expression by its result computations are shared, and each argument is evaluated at most once!

Optimisations

The combinator S sends the argument x into both subexpressions which have K c and I's at the leave. In case we have larger subtrees in which no x's are needed, we can use:

$$\begin{array}{ll} \mathsf{B}\ f\ g\ x = f\ \ (g\ x) & \texttt{-- function composition} \\ \mathsf{C}\ f\ g\ x = f\ x\ g & \texttt{-- happens to be the same as flip} \end{array}$$

which are used in the transformations:

$$\begin{array}{ll} \mathsf{S}\ (\mathsf{K}\ f)\ (\mathsf{K}\ g)\ x & \leadsto (\mathsf{K}\ f\ x)\ (\mathsf{K}\ g\ x) \leadsto \mathsf{K}\ (f\ g)\ x \\ \mathsf{S}\ (\mathsf{K}\ f)\ g & \leadsto \mathsf{B}\ f\ g \\ \mathsf{S}\ f\ (\mathsf{K}\ g) & \leadsto \mathsf{C}\ f\ g \end{array}$$

Specialised version to avoid expression blowup

Expressions soon beome veru complitaed, exponential blow-up in size: Better to use:

$$
\begin{aligned}
S' \; c\; f\; g\; x &= c\; (f\; x)\; (g\; x) \\
B' \; c\; f\; g\; x &= c\; f \quad\;\; (g\; x) \\
C' \; c\; f\; g\; x &= c\; (f\; x)\; g
\end{aligned}
$$

This leads for each ap[plication node to a string of S', B' and C's ending in a S, B or C. Each next element tells in which directions the next argument should flow.

Even Simpler 2

The combinator I is superfluous:

$$\mathsf{S}\ \mathsf{K}\ \mathsf{K}\ \mathsf{x} \leadsto (\mathsf{K}\ \mathsf{x})\ (\mathsf{K}\ \mathsf{x}) \leadsto \mathsf{x}$$

and hence

$$\mathsf{I} = \mathsf{S}\ \mathsf{K}\ \mathsf{K}$$

Even Simpler 2

In 1989 Jeroen Fokker (UU) invented:

$$
\begin{array}{l}
\mathsf{X} = \lambda \mathsf{f}.\, \mathsf{f}\ \mathsf{S}\ \mathsf{f3} \\
\mathsf{f3} = \lambda \mathsf{p}\ \_\ \_.\, \mathsf{p}\texttt{-- first of three}
\end{array}
$$

with which

$$
\begin{array}{llllll}
\mathsf{K\ y\ x} & \rightsquigarrow \mathsf{X} & \mathsf{X} & & & \mathsf{y\ x} \\
 & \rightsquigarrow \mathsf{X} & \mathsf{S} & \mathsf{f3} & & \mathsf{y\ x} \\
 & \rightsquigarrow \mathsf{S} & \mathsf{S} & \mathsf{f3} & \mathsf{f3} & \mathsf{y\ x} \\
 & \rightsquigarrow \mathsf{S} & \mathsf{f3} & (\mathsf{f3\ f3}) & \mathsf{y} & \mathsf{x} \\
 & \rightsquigarrow \mathsf{f3} & \mathsf{y} & (\mathsf{f3\ f3\ y}) & \mathsf{x} & \\
 & \rightsquigarrow \mathsf{y} & & & &
\end{array}
$$

And:

$$\mathsf{S} = \mathsf{X}\ (\mathsf{X}\ \mathsf{X}) = \mathsf{X}\ \mathsf{K} = \mathsf{K}\ \mathsf{S}\ \mathsf{f3} = \mathsf{S}$$

Conclusion

If we denote X by () all functions can be expressed using parentheses only.