Universiteit Utrecht

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

Advanced Functional Programming

2010-2011, periode 2

Jan Rochel

Department of Information and Computing Sciences
Utrecht University

November 24, 2011

4. Haskell and the λ-Calculus

An Example

Program definition:

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

Evaluation:

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

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:

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

A **reduction** to a normal form:

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

Some Terminology and Notation in Rewriting

- reducible expression (redex): a term that matches the left-hand side of a rewriting rule
- reduction step: application of a rule to a redex.
  $\mathsf{Main} \rightarrow \mathsf{Print}\ (\mathsf{gcd}\ (15, 12))$
  $\mathsf{Print}\ (\mathsf{gcd}\ (15, 12)) \leftarrow \mathsf{Main}$
  $\mathsf{Main} \rightarrow^* \mathsf{PutStrLn}\ (\mathsf{Show}\ (\mathsf{Gcd}'\ (\mathsf{Abs}\ (15), \mathsf{Abs}\ (12))))$
- 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

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

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

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

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

$$\text{flip a b c} \rightarrow^* \text{a c b}$$

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

Observations:

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

λ-Calculus: Grammar

λ-terms are of the form:

$$
\begin{array}{rcll}
e & ::= & x & \text{variables} \\
 & \mid & e\ e & \text{application} \\
 & \mid & \lambda x.\, e & \text{lambda abstraction}
\end{array}
$$

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

> flip f x y = f y x

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

Representation with unary functions:

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

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 rule of the λ-calculus is β-reduction:

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

An abstraction applied to an argument reduces to the abstraction's body with all *free* occurrences of the abstraction variable substituted by the argument.

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

Bound and free variables

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

Examples:

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

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

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:

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

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

λ-Calculus: Function Equivalence and η-Conversion

When are two λ-terms equivalent?

Every rewrite rule $\rightarrow_r$ is a relation on terms and every relation induces an equivalence relation (symmetric, reflexive, transitive closure):

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

- $\lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathsf{y}\ \mathsf{x}$ and $\lambda \mathsf{y}.\, \lambda \mathsf{z}.\, \mathsf{z}\ \mathsf{y}$ are α-equivalent because they can be transformed into another by α-conversion.
- $(\lambda \mathsf{y}.\, \mathsf{a}\ \mathsf{y})\ \mathsf{b} =_\beta (\lambda \mathsf{x}.\, \mathsf{x}\ \mathsf{b})\ \mathsf{a}$
  since $(\lambda \mathsf{y}.\, \mathsf{a}\ \mathsf{y})\ \mathsf{b} \rightarrow_\beta \mathsf{a}\ \mathsf{b} \leftarrow_\beta (\lambda \mathsf{x}.\, \mathsf{x}\ \mathsf{b})\ \mathsf{a}$
- $(\lambda \mathsf{y}.\, \lambda \mathsf{s}.\, \mathsf{a}\ \mathsf{s}\ \mathsf{y})\ \mathsf{b} =_{\alpha\beta} \lambda \mathsf{t}.\, (\lambda \mathsf{x}.\, \mathsf{x}\ \mathsf{t}\ \mathsf{b})\ \mathsf{a}$

λ-Calculus: Function Equivalence 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: $\lambda x.\, e\ x \ \rightarrow_\eta\ e$ (x does not occur free in e)

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

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

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

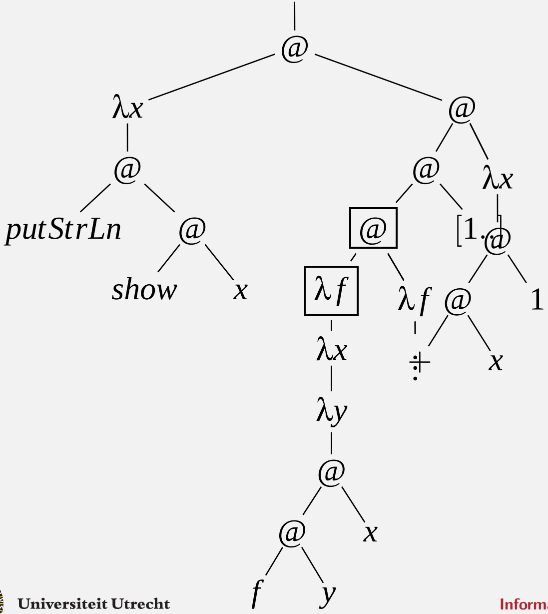
Example

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

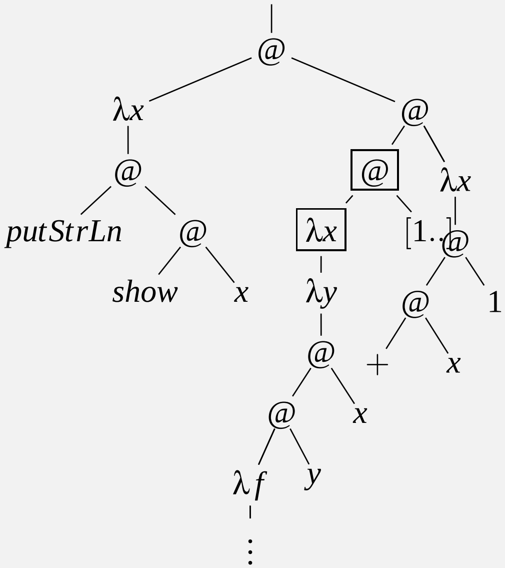
$$
\begin{aligned}
\mathsf{main} &= \mathsf{print}\ (\mathsf{flip}\ \mathsf{map}\ [1\,..]\ \mathsf{inc}) \\
\mathsf{print} &= \lambda \mathsf{x}.\ \mathsf{putStrLn}\ (\mathsf{show}\ \mathsf{x}) \\
\mathsf{flip} &= \lambda \mathsf{f}.\ \lambda \mathsf{y}.\ \lambda \mathsf{x}.\ \mathsf{f}\ \mathsf{y}\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

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

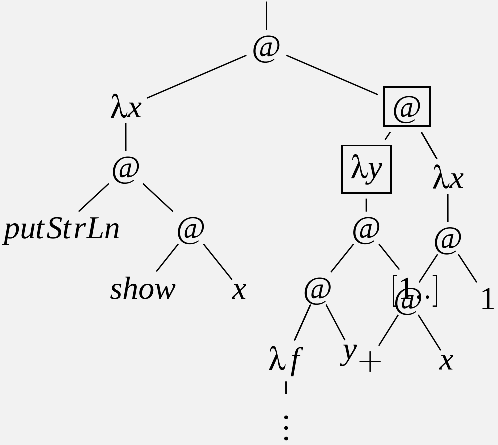
Example in Syntax-Tree Notation



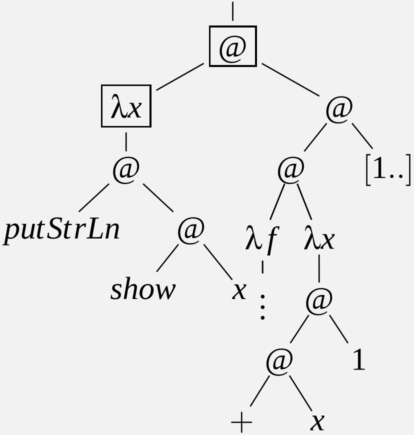
Example in Syntax-Tree Notation



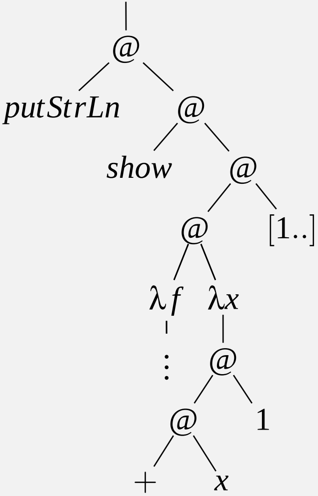
Example in Syntax-Tree Notation



Example in Syntax-Tree Notation

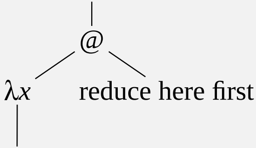


Example in Syntax-Tree Notation



Reduction Strategies

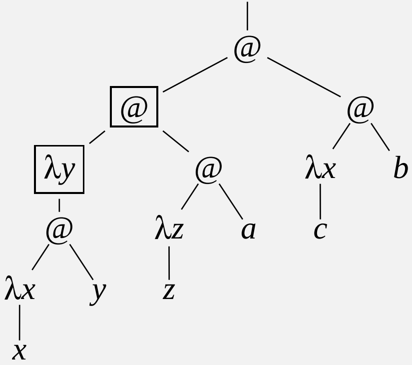
- Strict languages use call-by-value reduction: arguments have to be fully evaluated before a function is applied



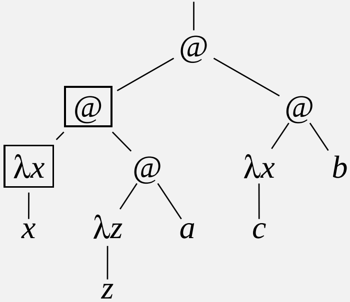
- Non-strict (lazy) evaluation: no reductions take place within the argument of a redex, for instance
- Haskell uses call-by-name reduction: the 'leftmost outermost' redex is reduced[1], leads to **weak head normal form** (WHNF)[2].

[1] also: no reductions under lambda take place
[2] otherwise reduction leads to head normal form (HNF)

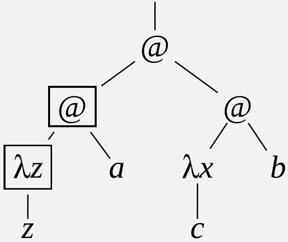
Example: Non-Strict Evaluation



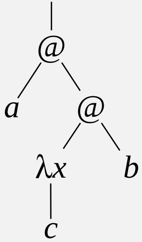
Example: Non-Strict Evaluation



Example: Non-Strict Evaluation



Example: Non-Strict Evaluation



Term is in WHNF but not in normal form

Simply-Typed λ-calculus

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

Function types nest to the right: $\tau \to \sigma \to \rho = \tau \to (\sigma \to \rho)$

Closed terms are typed as follows:

- 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 type of x must be 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 halts
$\Longrightarrow$ The simply-typed λ-calculus is not Turing complete

There are lambda terms (**fixed-point combinators**) that can be used to express recursion, like the Y-combinator:

$$\mathsf{Y} \equiv \lambda f.\,(\lambda x.\,f\ (x\ x))\ (\lambda x.\,f\ (x\ x))$$

but they are not typeable in the simply-typed λ-calculus.

Recursion and Turing Completeness

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

Homework: evaluate fac 3

Haskell features a (more flexible) **let** construct for recursion:

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

Haskell vs. the simply-typed λ-Calculus

Haskell is essentially λ-calculus extented by **let**, data types, case discrimination, and a richer type system.

syntactic sugar	desugares 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

Next lecture

Doaitse!