The lambda calculus
(An introduction)

Ralf Lämmel
Software Language Engineering Team
University of Koblenz-Landau
http://www.softlang.org/
Dependencies:
- Tree-based abstract syntax
- Operational semantics
- Type systems
Illustrative lambda expressions

Lambda abstraction, i.e., variables abstract over lambda expressions

// Identity function
\( \lambda x. x \)

// Function composition
\( \lambda f \ g \ x. \ f \ (g \ x) \)

N.B.: The \( \lambda \)-calculus is essentially a very simple functional programming language.
What’s the $\lambda$-calculus?

- The core of functional programming
- A Turing-complete notion of computability
- A great example of a calculus in language design
- A popular starting point in programming language theory

- let id = $\lambda x \to x$
- let (.) = $\lambda f g x \to f (g x)$
- id 42
- 42
- (not . not) True
- True

N.B.: Haskell is based on the $\lambda$-calculus, quite obviously.
Call-by-value
small-step operational semantics of $\lambda$-calculus

Reduction of lambda expressions

\[
\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2} \quad [\text{app1}]
\]

\[
\frac{e_2 \rightarrow e_2'}{v_1 e_2 \rightarrow v_1 e_2'} \quad [\text{app2}]
\]

Substitution

\[
(\lambda x. e) v \rightarrow [v/x]e \quad [\text{beta}]
\]

Values

\[
\lambda x. e \in \text{value} \quad [v-\lambda\text{lambda}]
\]

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Substitution as needed for beta-reduction

Free variables in a lambda expression

\[
\text{FREE}(x) = \{x\}
\]
\[
\text{FREE}(e_1 e_2) = \text{FREE}(e_1) \cup \text{FREE}(e_2)
\]
\[
\text{FREE}(\lambda x. e) = \text{FREE}(e) \setminus \{x\}
\]

Substitution of \(x\) by \(e\) within an expression

\[
[e/x]x = e
\]
\[
[e/x]y = y, \text{if } x \neq y
\]
\[
[e/x]e_1 e_2 = ([e/x]e_1) ([e/x]e_2)
\]
\[
[e/x]\lambda x. e' = \lambda x. e'
\]
\[
[e/x]\lambda y. e' = \lambda y. [e/x]e', \text{if } x \neq y \text{ and } y \notin \text{FREE}(e)
\]
Adding predefined values and operations

11.1 The untyped lambda calculus

Exercise 11.1

(Alpha equivalence)

Implement alpha equivalence in a declarative program. That is, given two expressions, return true, if they can be alpha-converted into each other; otherwise return false.

Exercise 11.2

(Substitution as a total operation)

Implement substitution as a total operation in a declarative program. In order to achieve totality, you need to implement the last case of Specification 11.3 such that alpha conversion is applied, if necessary. A robust scheme is needed to identify a 'fresh' variable identifier to be used for alpha conversion.

11.1.4 Predefined values and operations

We extend the untyped lambda calculus to incorporate the expression language BTL which essentially provides predefined values and operations for natural numbers and Boolean values. The result is also referred to as an applied lambda calculus. This extension is supposed to provide us with a practically more useful calculus. We summarize abstract and concrete syntax for the resulting calculus.

<table>
<thead>
<tr>
<th>Abstract syntax</th>
<th>Concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>var((x))</td>
<td>(x)</td>
</tr>
<tr>
<td>lambda((x, e))</td>
<td>(\lambda x. e)</td>
</tr>
<tr>
<td>apply((e_1, e_2))</td>
<td>(e_1 e_2)</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>zero</td>
<td>zero</td>
</tr>
<tr>
<td>succ((e))</td>
<td>succ (e)</td>
</tr>
<tr>
<td>pred((e))</td>
<td>pred (e)</td>
</tr>
<tr>
<td>iszero((e))</td>
<td>iszero (e)</td>
</tr>
<tr>
<td>if((e_1, e_2, e_3))</td>
<td>if (e_1) (\text{then } e_2 \text{ else } e_3)</td>
</tr>
</tbody>
</table>

The small-step operational semantics of the resulting applied lambda calculus is also just the trivial composition ('concatenation') of the semantics of the contributing languages. Arguably, this style of composing lambda calculus and BTL leads to some redundancy because the rules for the constructs succ, pred, iszero, and if involve some elements of function application which should be taken care of by rules for the lambda calculus.

N.B.: Thus, **BTL** — Basic TAPL Language — (referring again to Pierce’s textbook ‘Types and programming languages’) is added to the \(\lambda\)-calculus.
If we only had recursive functions ...

\[
\begin{align*}
\text{add} & \quad = \lambda n. \lambda m. \text{if iszero } n \text{ then } m \text{ else succ } (\text{add} (\text{pred } n) \ m) \\
\text{mul} & \quad = \lambda n. \lambda m. \text{if iszero } n \text{ then zero else add } m \ (\text{mul} (\text{pred } n) \ m) \\
\text{factorial} & \quad = \lambda n. \text{if iszero } n \text{ then succ zero else mul } n \ (\text{factorial} (\text{pred } n))
\end{align*}
\]

N.B.: Lambda-abstractions allow us to abstract over \( \lambda \)-expressions (i.e., functions), but there is no form of self-reference.
Recursion by means of a fixed-point combinator

\[
\begin{align*}
\text{add} & = \text{fix} (\lambda f. \lambda n. \lambda m. \text{if iszero } n \text{ then } m \text{ else succ } (f (\text{pred } n) m)) \\
\text{mul} & = \text{fix} (\lambda f. \lambda n. \lambda m. \text{if iszero } n \text{ then zero else add } m (f (\text{pred } n) m)) \\
\text{factorial} & = \text{fix} (\lambda f. \lambda n. \text{if iszero } n \text{ then succ zero else mul } n (f (\text{pred } n)))
\end{align*}
\]

\[
\begin{align*}
\frac{e \rightarrow e'}{\text{fix } e \rightarrow \text{fix } e'} \quad \text{[fix1]} \\
\frac{\text{fix } \lambda x. e \rightarrow [\text{fix } \lambda x. e/x]e}{\text{[fix2]}}
\end{align*}
\]

N.B.: We parametrize in the recursive reference (i.e., \(f\) above) and we tie the recursive knot with the \textbf{fix}-construct.
Turing completeness

<table>
<thead>
<tr>
<th>Value/Operation</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>( \lambda t. \lambda f. t )</td>
</tr>
<tr>
<td>false</td>
<td>( \lambda t. \lambda f. f )</td>
</tr>
<tr>
<td>if</td>
<td>( \lambda b. \lambda v. \lambda w. b , v , w )</td>
</tr>
<tr>
<td>0</td>
<td>( \lambda s. \lambda z. z )</td>
</tr>
<tr>
<td>1</td>
<td>( \lambda s. \lambda z. s , z )</td>
</tr>
<tr>
<td>2</td>
<td>( \lambda s. \lambda z. s , (s , z) )</td>
</tr>
<tr>
<td>3</td>
<td>( \lambda s. \lambda z. s , (s ,(s , z)) )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>succ</td>
<td>( \lambda n. \lambda s. \lambda z. s ,(n , s , z) )</td>
</tr>
</tbody>
</table>

\[ \text{fix} = \lambda f. (\lambda x. f((\lambda v.((x \, x) \, v))) \,(\lambda x. f((\lambda v.((x \, x) \, v)))) \]

N.B.: None of the extensions discussed thus far are ‘essential’.
A realization of the $\lambda$-calculus: ULL — Untyped Lambda Language

\[
\text{add} = \text{Fix} (\text{Lambda} \ "f" (\text{Lambda} \ "n" (\text{Lambda} \ "m" \\
(\text{If} (\text{IsZero} (\text{Var} \ "n")) \\
(\text{Var} \ "m") \\
(\text{Succ} (\text{Apply} (\text{Apply} (\text{Var} \ "f") (\text{Pred} (\text{Var} \ "n"))) (\text{Var} \ "m")))))))
\]

--- Convert Haskell Int to expression such that isNat is True
\[
\text{fromInt} :: \text{Int} \to \text{Expr} \\
\text{fromInt} \ i \mid i \geq 0 = \text{if} \ i = 0 \text{ then Zero else Succ} \ (\text{fromInt} \ (i-1))
\]

--- Convert expression such that isNat is True to Haskell Int
\[
\text{toInt} :: \text{Expr} \to \text{Int} \\
\text{toInt} \ \text{Zero} = 0 \\
\text{toInt} \ (\text{Succ} \ e) = \text{toInt} \ e + 1
\]

\[\text{let (Just } r) = \text{steps} (\text{Apply} (\text{Apply} \ \text{add} \ (\text{fromInt} \ 20)) (\text{fromInt} \ 22))\]
\[\text{tolInt} \ r\]
42
Type system of the simply typed $\lambda$-calculus

\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{[t-var]}
\]

\[
\frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1 \, e_2 : T_2} \quad \text{[t-apply]}
\]

\[
\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1. \, e : T_1 \rightarrow T_2} \quad \text{[t-abstr]}
\]

N.B.: An extra type context maintains the types of variables as declared by the annotations of the $\lambda$-abstractions.
A realization of the simply typed $\lambda$-calculus:

**TLL — Typed Lambda Language**

add = Fix (Lambda "f" (FunType NatType (FunType NatType NatType)))
    (Lambda "n" NatType)
    (Lambda "m" NatType)
    (If (IsZero (Var "n"))
        (Var "m")
        (Succ (Apply (Apply (Var "f") (Pred (Var "n"))) (Var "m"))))))))

▶ typeof empty add
Just (FunType NatType (FunType NatType NatType))
A trivial form of type erasure

N.B.: As a consequence, we can leverage directly the semantics of the untyped calculus. We could also aim at ‘inferring’ the types of the variables instead of relying on type annotations.
Online resources

YAS’ GitHub repository contains all code.
YAS (Yet Another SLR (Software Language Repository))
http://www.softlang.org/yas
See Haskell-based implementations of languages ULL and TLL.

The Software Languages Book
http://www.softlang.org/book
The book discusses also polymorphism.