barkhauseninstitut.org

Foundations of Certified Programming Language and Compiler Design

Dr.-Ing. Sebastian Ertel

Composable Operating Systems Group, Barkhausen Institute

Outline

Lecture	Logic	Formalisms	PL
1	Propositional and first-order logic		
2			Functional programming
3		Syntax and Semantics	
4			The untyped lambda calculus
5		Types	
6			The typed lambda calculus
7			Polymorphism
8		Curry-Howard	
9			Higher-order types
10			Dependent types

Goals

Let’s understand the foundation of programming languages

- as a mathematical system
- that allows for proving theorems.

History

1920s/30s – Alonzo Church The (untyped) lambda calculus =

- A formal system in which all operations are reduced to
- function definition and
- function application.

1960s – Peter Landin A complex language =

- a tiny core calculus (that captures the language essentials) with
- a collection of derived forms (that can be translated into this core).

  The core language of Landin was the lambda calculus.

1970s – John McCarthy Lisp is based on the lambda calculus.

The lambda calculus is both

- a simple programming language *in which* compuations can be described and
- a mathematical object *about which* rigorous statements can be proved.

The Essence of Programming

Abstraction

- Consider this expression: `(5*4*3*2*1) + (7*6*5*4*3*2*1) + (3*2*1)`
- A programmer would write: `factorial 5 + factorial 6 + factorial 3`
- and define: `factorial n = if n=0 then 1 else n * factorial (n-1)`
- where `factorial =` λ `n. if n=0 then 1 else n * factorial (n-1)`
- is a function/*abstraction* that yields . . . for each n.

Application

- When stating `factorial 0`, we *apply*
- the function λ `n. if n=0 then 1 else n * factorial (n-1)`
- to the argument `0`, i.e.,
- *variable* `n` is replaced by `0` such that
- `if 0=0 then 1 else 0 * factorial (0-1)`
- to compute the result `1`.

The Untyped Lambda Calculus
Syntax

- The lambda calculus captures exactly this essence of programming in its purest form:

$$
\begin{array}{lll}
t ::= & & \text{terms:} \\
 & \mid\ x & \text{variable} \\
 & \mid\ \lambda x.t & \text{abstraction} \\
 & \mid\ t\ t & \text{application}
\end{array}
$$

- Scope of variables: A variable x is said to be

  bound if it occurs inside the body t of an abstraction $\lambda x.t$.
  (λx is a *binder* whose scope is t.)

  free if it occurs in a position where it is not bound by an abstraction.

- A term with no free variables is called a *closed term* or *combinator*.

```
id = λx.x
```

The Untyped Lambda Calculus
Operational Semantics

In its pure form, the lambda calculus

- contains **no** built-in constants or primitives
- captures the sole means of computation: application of functions to arguments.

Each step in the computation

- *rewrites* an application with an abstaction on the left-hand side by
- *substituting* the term on the right-hand side for the variable in the abstraction's body:

$$(\lambda x.t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12}$$

- where $[x \mapsto t_2]t_{12}$ means the term obtained by "replacing all **free** occurences of x in t_{12} by t_2."

Beta Reduction

- According to Church:
  redex (“reducible expression”) is a term of the form $(\lambda x.t_{12})\ t_2$
  beta reduction is the operation of *rewriting* a redex according to the substitution rule.
- Consider this term with 3 redexes:

$$(\lambda x.x)\ ((\lambda x.x)\ (\lambda z.(\lambda x.x)\ z)) \equiv \underline{\mathtt{id}\ (\underline{\mathtt{id}\ (\lambda z.\underline{\mathtt{id}\ z})})}$$

- Several evaluation strategies exist:
  full beta reduction where any redex may be reduced
  normal order where the leftmost, outermost redex is reduced
  call by name where reductions inside abstractions are not allowed
  call by value where only outermost redexes are reduced *and* a redex is reduced only when its right-hand side is a value

$$\mathtt{id}\ (\mathtt{id}\ (\lambda z.\mathtt{id}\ z)) \longrightarrow \mathtt{id}\ (\lambda z.\mathtt{id}\ z) \longrightarrow \lambda z.\mathtt{id}\ z \not\longrightarrow$$

Beta Reduction

- According to Church:

  redex ("reducible expression") is a term of the form $(\lambda x.t_{12})\ t_2$

  beta reduction is the operation of *rewriting* a redex according to the substitution rule.
- Consider this term with 3 redexes:

$$(\lambda x.x)\ ((\lambda x.x)\ (\lambda z.(\lambda x.x)\ z)) \equiv \underline{\texttt{id}\ (\underline{\texttt{id}\ (\lambda z.\underline{\texttt{id}\ z})})}$$

- Several evaluation strategies exist:

  full beta reduction where any redex may be reduced

  normal order where the leftmost, outermost redex is reduced

  call by name where reductions inside abstractions are not allowed

  call by value where only outermost redexes are reduced *and* a redex is reduced only when its right-hand side is a value
- Evaluation strategies can be classified as

  strict where all arguments are evaluated

  lazy where only used arguments are evaluated
- The strategy is mostly irrelevant for the typed lambda calculus. (Let's stick with call by value.)

Operational Semantics

Syntax

t ::=		terms:
	x	variable
	$\lambda x.t$	abstraction
	$t\ t$	application

v ::=		values:
	$\lambda x.t$	abstraction value

Evaluation Rules

$$\boxed{t \longrightarrow t'}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{E-APP1} \qquad \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}\ \text{E-APP2} \qquad \frac{}{(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}}\ \text{E-APPABS}$$

Substitution Formally

- Let's define a function $[x \mapsto s]$ inductively over terms t.
- A naive solution:

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.[x \mapsto s]t_1 & \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$

- But what about $[x \mapsto y](\lambda x.x) = \lambda x.y$?!

Substitution Formally

- Let's define a function $[x \mapsto s]$ inductively over terms t.
- Distinguish between the *free* and *bound* occurences of variables in a term.

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \begin{cases} \lambda y.t_1 & \\ \lambda y.[x \mapsto s]t_1 & \end{cases} & \begin{array}{l}\text{if } y = x \\ \text{if } y \neq x\end{array} \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$

- But what about $[x \mapsto z](\lambda z.x) = \lambda z.z$?!
- This problem is called *variable capture*.

Substitution Formally

- Let's define a function $[x \mapsto s]$ inductively over terms t.
- *Capture-avoiding substitution:*

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \begin{cases} \lambda y.t_1 \\ \lambda y.[x \mapsto s]t_1 \end{cases} & \begin{array}{l} \text{if } y = x \\ \text{if } y \neq x \text{ and } y \notin FV(s) \end{array} \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$

- This function is only partial! Consider $[x \mapsto y\ z](\lambda y.x\ y)$

Substitution Formally

- Let's define a function $[x \mapsto s]$ inductively over terms t.
- Common fix: working with terms up to renaming of bound variables (Church: alpha conversion).

Convention

Terms that differ only in the names of bound variables are interchangable in all contexts.

- Example: $[x \mapsto y\ z](\lambda y.x\ y) \xrightarrow{\alpha} [x \mapsto y\ z](\lambda w.x\ w) = \lambda w.y\ z\ w$

Definition (Substitution)

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.[x \mapsto s]t_1 & \text{if } y \neq x \text{ and } y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$

Encoding Computation

- Functions with multiple arguments via higher-order functions: $\lambda x.\lambda y.t$

  Currying (named in honor to Haskell Curry) is a transformation from multi-argument functions such as $\lambda(x, y).t$ to higher-order functions $\lambda x.\lambda y.t$

- Booleans, Conditionals and Logical Connectives

  Church booleans are defined as $\texttt{tru} = \lambda t.\lambda f.t$ and $\texttt{fls} = \lambda t.\lambda f.f$

  Conditional test can be encoded as $\texttt{test} = \lambda l.\lambda m.\lambda n.l\ m\ n$

  Logical $\texttt{and}$ follows as

  - $\texttt{and} = \lambda b.\lambda c.\texttt{test}\ b\ (\texttt{test}\ c\ \texttt{tru}\ \texttt{fls})\ \texttt{fls}$
  - or shorter $\texttt{and} = \lambda b.\lambda c.b\ c\ \texttt{fls}$

- Pairs based on booleans: $\texttt{pair} = \lambda f.\lambda s.\lambda b.b\ f\ s$ where $\texttt{fst} = \lambda p.p\ \texttt{tru}$ and $\texttt{snd} = \lambda p.p\ \texttt{fls}$

Recursion

- Not all terms reduce to a normal form!
- Consider this combinator: $\texttt{omega} = (\lambda x.x\ x)\ (\lambda x.x\ x)$
- Its generalization gives rise to the *fix-point combinator*[1]:

$$\texttt{fix} = \lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y))$$

- $\texttt{fix}$ embodies recursion.
- Consider our $\texttt{factorial}$ example (with Church numerals) again:

$$\begin{array}{l}
\texttt{if realeq}\ n\ c_0\ \texttt{then}\ c_1 \\
\texttt{else times}\ n\ (\\
\quad \texttt{if realeq}\ (\texttt{prd}\ n)\ c_0\ \texttt{then}\ c_1 \\
\quad \texttt{else times}\ (\texttt{prd}\ n)\ (\\
\quad\quad \texttt{if realeq}\ (\texttt{prd}\ (\texttt{prd}\ n))\ c_0\ \texttt{then}\ c_1 \\
\quad\quad \texttt{else times}\ (\texttt{prd}\ (\texttt{prd}\ n))\ (\\
\ldots)))
\end{array}$$

$$\begin{array}{l}
\texttt{g} = \lambda fct.\lambda n.\texttt{if realeq}\ n\ c_0\ \texttt{then}\ c_1 \\
\qquad\qquad\qquad \texttt{else times}\ n\ (fct\ (\texttt{prd}\ n)) \\
\texttt{factorial} = \texttt{fix}\ g
\end{array}$$

[1] Also known as the *call by value Y-combinator*. The call by name version is simpler: $\texttt{fix} = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$

Encoding data

Coq	Church	Scott
	$c_0 = \lambda s.\lambda z.\, z$ $c_1 = \lambda s.\lambda z.\, s\, z$ $c_2 = \lambda s.\lambda z.\, s\,(s\, z)$ $c_3 = \lambda s.\lambda z. s\,(s\,(s\, z))$ etc.	
`Inductive ℕ = O` `  | S: ℕ -> ℕ`	$\texttt{zero} = \lambda s\, z.\, z$ $\texttt{succ} = \lambda n\, s\, z.\, s\,(n\, s\, z)$	$\texttt{zero} = \lambda z\, s.\, z$ $\texttt{succ} = \lambda n\, f\, g.\, g\, n$

`Inductive List a :=`
`  | nil:  List a`
`  | cons:  a -> List a -> List a`
	$\texttt{nil} = \lambda\, c\, x.\, x$
	$\texttt{cons} = \lambda\, h\, t\, c\, n.\, c\, h\,(t\, c\, n))$
		$\texttt{nil} = \lambda\, c\, x.\, x$
		$\texttt{cons} = \lambda\, h\, t\, c.\, c\, h\, t$

`head :  list a -> a`	$\texttt{head} = \lambda\, l.\, l\,(\lambda\, x\, xs.\, x)\ \texttt{undef}$	$\texttt{head} = \lambda\, l.\, l\,(\lambda\, x\, xs.\, x)\ \texttt{undef}$

[0] Pieter Koopman, Rinus Plasmeijer, and Jan Martin Jansen. "Church encoding of data types considered harmful for implementations". In: *26th Symposium on Implementation and Application of Functional Languages (IFL)*. 2014