

Semantics and Analysis of Programs for Security

Operational semantics

Thomas Jensen

PAS/SdL, M2R Informatique, U. Rennes 1

October 2011



Outline

1 Semantics of the lambda calculus



The lambda calculus

We define a minimalistic procedural programming language: the lambda calculus extended with booleans and integers.

One basic computation: **substitution** (β -reduction).

The calculus is minimalistic but as a computational formalism it is Turing complete.

References

- ▶ A. Church: The Calculi of Lambda-Conversion, Princeton University Press, 1941.
- ▶ H.P. Barendregt. The Lambda Calculus: Its Syntax and Semantics, Studies in Logic and the Foundations of Mathematics, 103, North Holland, 1984.
- ▶ Barendregt and Barendsen: Introduction to Lambda Calculus, available on-line at <ftp://ftp.cs.ru.nl/pub/CompMath.../lambda.pdf>



Syntax of the lambda calculus

Expressions :

$$\begin{array}{l}
 e ::= \mathbf{x} \\
 \quad | \lambda \mathbf{x}. e \\
 \quad | e e
 \end{array}$$

The current version is **untyped**.

In the lecture on type systems, we'll see how to define *the simply-typed lambda calculus*.



Examples

The identity function $\text{id} = \lambda x. x$

Constant function: $\lambda y. x$

Projection: $\lambda x. \lambda y. x$

Double: $\lambda f. \lambda x. f (f x)$

Omega: $(\lambda x. x x) (\lambda x. x x)$

Alpha equivalence

Variables are **bound** by lambdas.

The variables bound by lambdas become “placeholders” and can be renamed without changing the term

Example:

$\lambda x. x$ and $\lambda y. y$

represent the same function.

Lambda terms that are equal up to renaming of bound variables are said to be **alpha-equivalent**

Free variables of a lambda term

Definition The set $FV(t)$ of **free variables** of a term t .

$$\begin{aligned}
 FV(\mathbf{x}) &= \{\mathbf{x}\} \\
 FV(t_1 t_2) &= FV(t_1) \cup FV(t_2) \\
 FV(\lambda \mathbf{x}.t) &= FV(t) \setminus \{\mathbf{x}\} \\
 FV(op(t_1, t_2, \dots)) &= FV(t_1) \cup \dots \cup FV(t_2)
 \end{aligned}$$

Substitution

The (only) computation step in the pure lambda calculus is substitution: replacing a variable by a lambda term.

Substitution is **tricky** to define in presence of bound variables.

First and **wrong** attempt:

$$\begin{aligned}
 \mathbf{x}[t/\mathbf{x}] &= t \\
 \mathbf{y}[t/\mathbf{x}] &= \mathbf{y} \\
 t_1 t_2[t/\mathbf{x}] &= t_1[t/\mathbf{x}] t_2[t/\mathbf{x}] \\
 (\lambda \mathbf{y}.t_1)[t/\mathbf{x}] &= \lambda \mathbf{y} t_1[t/\mathbf{x}]
 \end{aligned}$$

Works fine for $\lambda \mathbf{y}. \mathbf{x}[\lambda \mathbf{z}. \mathbf{w}/\mathbf{x}]$

But not for $\lambda \mathbf{x}. \mathbf{x}[\mathbf{y}/\mathbf{x}]$.

Substitution - next try

Second but still **wrong** attempt:

$$\begin{aligned}
 \mathbf{x}[t/\mathbf{x}] &= t \\
 \mathbf{y}[t/\mathbf{x}] &= \mathbf{y} \\
 t_1 t_2[t/\mathbf{x}] &= t_1[t/\mathbf{x}] t_2[t/\mathbf{x}] \\
 (\lambda \mathbf{y}.t_1)[t/\mathbf{x}] &= \lambda \mathbf{y} t_1[t/\mathbf{x}] && \mathbf{x} \neq \mathbf{y} \\
 (\lambda \mathbf{y}.t_1)[t/\mathbf{x}] &= \lambda \mathbf{y} t_1 && \mathbf{x} = \mathbf{y}
 \end{aligned}$$

Works for $\lambda \mathbf{x}.\mathbf{x}[y/\mathbf{x}]$.

But not for $\lambda \mathbf{z}.\mathbf{x}[z/\mathbf{x}]$: a free variable becomes bound.



Substitution and beta reduction

Definition Substitution of a term for a variable.

$$\begin{aligned}
 \mathbf{x}[t/\mathbf{x}] &= t \\
 \mathbf{y}[t/\mathbf{x}] &= \mathbf{y} \\
 t_1 t_2[t/\mathbf{x}] &= t_1[t/\mathbf{x}] t_2[t/\mathbf{x}] \\
 (\lambda \mathbf{y}.t_1)[t/\mathbf{x}] &= \lambda \mathbf{y} t_1[t/\mathbf{x}] && \mathbf{y} \notin FV(t) \text{ and } \mathbf{y} \neq \mathbf{x} \\
 op(t_1, t_2, \dots)[t/\mathbf{x}] &= op(t_1[t/\mathbf{x}], t_2[t/\mathbf{x}], \dots)
 \end{aligned}$$

using **alpha conversion** to avoid problems such as $\lambda \mathbf{y}.\mathbf{x}[y z/\mathbf{x}]$.

Definition Beta (β) reduction

$$(\lambda \mathbf{x}.e) t \rightarrow e[t/\mathbf{x}]$$

Example:

$$(\lambda \mathbf{x}.\mathbf{x} \mathbf{x}) (\lambda \mathbf{x}.\mathbf{x} \mathbf{x}) \rightarrow (\lambda \mathbf{x}.\mathbf{x} \mathbf{x}) (\lambda \mathbf{x}.\mathbf{x} \mathbf{x})$$



Evaluation order

A **redex** is a sub-term of the form $\lambda x.e_1 e_2$.

This is where β -reduction is done.

A term may have **more than one** redex.

For example, there are three redexes in the term $\text{id } (\text{id } (\lambda z.\text{id } z))$ where $\text{id} = \lambda x. x$.

$$\text{id } (\text{id } (\lambda z.\underline{\text{id } z})) \quad \text{id } (\underline{\text{id } } (\lambda z.\text{id } z)) \quad \underline{\text{id } } (\text{id } (\lambda z.\text{id } z))$$

There are thus different **strategies** for evaluating a lambda term.

Normal order reduction and call-by-name

Normal order reduction.

Reduce the left-most outer-most redex first.

Call-by-name (and call-by-need)

Normal order reduction but never under a λ -abstraction.

Normal order will reduce

$$\underline{\text{id } (\text{id } (\lambda z.\text{id } z))} \rightarrow \underline{\text{id } } (\lambda z.\text{id } z) \rightarrow \lambda z.\underline{\text{id } z} \rightarrow \lambda z. z$$

Call-by-value

Intuitively: evaluate the arguments to functions before applying the function.

Call-by-value.

Evaluate outermost redex, but argument (right term) must be in normal form and no evaluation under λ -abstractions.

In our example, call-by-value will start with

$$\text{id } (\text{id } (\lambda z.\text{id } z))$$

Most programming languages use this strategy.

Syntax of lambda calculus with constants

Expressions :

$$\begin{array}{l}
 e ::= \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \\
 \quad \mid \mathbf{0} \mid \text{succ } e \mid \text{pred } e \mid \text{iszero } e \\
 \quad \mid \mathbf{x} \\
 \quad \mid \lambda \mathbf{x}. e \\
 \quad \mid e e
 \end{array}$$

So we can now write terms like

$$\lambda f . \lambda x . \text{if } x < 2 \text{ then } 1 \text{ else } x * f(x-1)$$

The current version is **untyped**.

In the lecture on type systems, we'll see how to define *the simply-typed lambda calculus*.

Operational semantics of the lambda calculus

The definition of the (small-step) operational semantics has two parts:

Values :

$$v ::= nv \mid \text{true} \mid \text{false} \mid \lambda x.e \quad \mathbf{with} \quad nv ::= 0 \mid \text{succ}(nv)$$

Transition relation:

$$\begin{array}{c} \text{pred } 0 \rightarrow 0 \quad \text{pred}(\text{succ } nv) \rightarrow nv \\ \text{iszero } 0 \rightarrow \text{true} \quad \text{iszero}(\text{succ } e) \rightarrow \text{false} \\ \frac{e \rightarrow e'}{\text{succ } e \rightarrow \text{succ } e'} \quad \frac{e \rightarrow e'}{\text{pred } e \rightarrow \text{pred } e'} \quad \frac{e \rightarrow e'}{\text{iszero } e \rightarrow \text{iszero } e'} \\ \text{if true then } e_1 \text{ else } e_2 \rightarrow e_1 \quad \text{if false then } e_1 \text{ else } e_2 \rightarrow e_2 \\ \frac{e \rightarrow e'}{\text{if } e \text{ then } e_1 \text{ else } e_2 \rightarrow \text{if } e' \text{ then } e_1 \text{ else } e_2} \\ \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{e_2 \rightarrow e'_2}{v e_2 \rightarrow v e'_2} \quad (\lambda x.e) v \rightarrow e[v/x] \end{array}$$

◀ ▶

Recursion

Recursively defined functions are usually written

$$\text{letrec fac } x = \text{if } x < 2 \text{ then } 1 \text{ else } x * \text{fac}(x-1)$$

We introduce a **recursion operator** `fix`

$$\text{fac} = \text{fix } \lambda f . \lambda x . \text{if } x < 2 \text{ then } 1 \text{ else } x * f(x-1)$$

Expressions $e ::= \dots \mid \text{fix } e$

Values $v ::= \dots$

Evaluation

$$\text{fix } (\lambda x.e) \rightarrow e[\text{fix } (\lambda x.e)/x] \quad \frac{e \rightarrow e'}{\text{fix } e \rightarrow \text{fix } e'}$$

◀ ▶

Recursion - example

Let

$$\text{fac} = \text{fix } \lambda f . \lambda x . \text{if } x < 2 \text{ then } 1 \text{ else } x * f(x-1)$$

Compute $\text{fac } 3$.

$$(\text{fix } \lambda f . \lambda x . \text{if } x < 2 \text{ then } 1 \text{ else } x * f(x-1)) \ 3 \rightarrow$$

$$(\lambda x . \text{if } x < 2 \text{ then } 1 \text{ else } x * f(x-1))[\text{fac} / f] \ 3 \rightarrow$$

$$(\text{if } 3 < 2 \text{ then } 1 \text{ else } 3 * f(3-1))[\text{fac} / f] \rightarrow$$

$$(3 * (\text{fix } \lambda f . \lambda x . \text{if } x < 2 \text{ then } 1 \text{ else } x * f(x-1)) (2))) \rightarrow$$

...



Fix as a lambda term

The recursion operator can be defined as a lambda term!

Set fix to the lambda term

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

Notice the similarity with the diverting term Ω .

Exercise: Show that fix is a recursion operator, i.e.,

$$\text{fix } (\lambda x.e) \rightarrow e[\text{fix } (\lambda x.e) / x]$$

