

Comp 205: Comparative Programming Languages

Semantics of Functional Languages

- Term- and Graph-Rewriting
- The λ -calculus

Lecture notes, exercises, etc., can be found at:
www.csc.liv.ac.uk/~grant/Teaching/COMP205/

Term Rewriting

A straightforward way of implementing a functional programming language is to implement **term-rewriting**.

The Haskell interpreter evaluates expressions (terms) by "**substituting equals for equals**".

For example, given the following definitions:

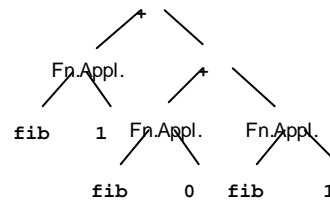
```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2), if n>1
```

An evaluation might proceed as follows:

```
fib 3
P (fib 2) + (fib 1)
P (fib 1) + (fib 0) + (fib 1)
P 1 + (fib 0) + (fib 1)
P 1 + 1 + (fib 1)
P 1 + 1 + 1
P 3
```

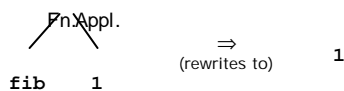
Terms and Trees

A standard way of representing terms is as trees:



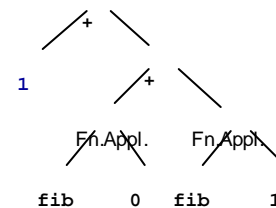
Term Rewriting

Term rewriting replaces (sub)trees:



Rewriting Trees

... giving the new tree:



Side-Effects

In the imperative paradigm, all evaluation can change the current state (e.g., by side-effects).

```
int funnyCount=0;

int funny(int i)
{
  return funnyCount++ * i;
}
```

```
funny(2) + funny(2) // might be 0+2
```

Functional Expressions

In the functional paradigm there is no state, so an expression always denotes the same value, and evaluation simply converts an expression to its value.

This important property of functional languages is referred to as "referential transparency".

Referential Transparency

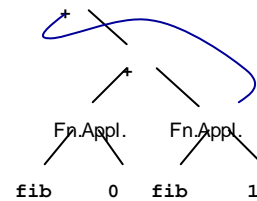
Referential Transparency: any expression denotes a single value, irrespective of its context.

Consequently, (sub)expressions can be replaced by their values without changing the behaviour of a program.

(Referential transparency = no side-effects)

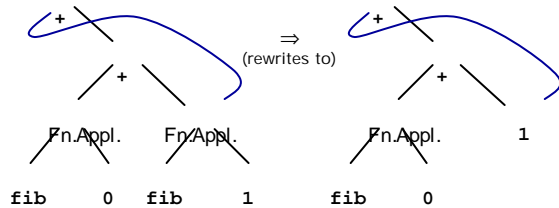
Graphs

Expressions can also be represented by **graphs**:



Graph Rewriting

This allows identical subexpressions to be rewritten **"in one go"**:



The λ -Calculus

The λ -calculus was developed by the mathematician Alonzo Church as a tool to study functions and computability.

The λ -calculus provides a very simple model of computable function, and inspired the designers of the first functional programming language, LISP.

The λ -calculus also provides an operational semantics for functional languages.

Computability

Alan Turing developed an abstract machine (the Turing Machine) and showed that it provided a universal model of computation. He showed that the **Turing-computable functions** were exactly the **general recursive functions**.

Church showed that the Turing-computable functions were exactly those that could be represented in the λ -calculus (the **λ -computable functions**).

Church-Turing Hypothesis

Both Turing and Church conjectured that their model of computability formalised the intuitive notion of mathematical computability.

The **Church-Turing Hypothesis** states that the equivalent notions of Turing- and λ -computability capture precisely "every function that would naturally be regarded as computable".

λ -Terms

Church designed the λ -calculus as a tool to study the fundamental notion of computable function. He therefore sought to strip away all but the bare essentials.

The syntax for λ -terms is accordingly very simple. λ -terms are built from:

- **variables**
- **function application**
- **λ -abstraction** (declaring formal parameters)
- (and **brackets**)

Syntax of λ -Terms

The set \mathcal{T} of λ -terms is defined by:

$$\mathcal{T} ::= \text{Var} \mid \lambda x. \mathcal{T} \mid \mathcal{T} \mathcal{T}$$

where **Var** is a set of variables.

Looking at each clause in turn:

- **variables**
Typically, we use **a, b, c, ..., x, y, z** for variables; if we run out of variable names, we can use **x', x'', x'''** , etc.

Syntax of λ -Terms

- **function application**

If M and N are λ -terms, so too is $M N$, which represents the application of M to N . (All λ -terms can be considered functions.)

- **λ -abstraction**

If M is a λ -term, so too is $\lambda x.M$, which can be thought of as a function with formal parameter x , and with body M .

Some Examples

- x
- $x y$
- $\lambda y.(x y)$
- $\lambda x.\lambda y.x y$
- **brackets**
 λ -abstraction binds less tightly than function application, so the last example above should be read as $\lambda x.(\lambda y.(x y))$ not $(\lambda x.(\lambda y.x)) y$.
Also, $x y z$ should be read as $(x y) z$.

Evaluation of λ -Terms

A λ -term $\lambda x.M$ represents a function whose formal parameter is x and whose body is M . When this function is applied to another λ -term N , as in the λ -term

$$(\lambda x.M) N,$$

evaluation proceeds by replacing x with N in the body M . For example,

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

In order to make this precise, we need the notions of **free** and **bound variables**.

Free and Bound Variables

Given a λ -term $\lambda x.M$, we say that λ **binds occurrences of x in M** .

We also say that x is **bound in $(\lambda x.M)$** .

A **free variable** is one that is not bound by a λ -abstraction.

A λ -term is **closed** if it has no free variables.

Free Variables

Given a λ -term M , we write $FV(M)$ for the set of free variables in M . This set is defined as follows:

- $FV(x) = \{x\}$
In a λ -term consisting of just a variable, that variable is free.
- $FV(MN) = FV(M) \cup FV(N)$
Function application doesn't bind variables.
- $FV(\lambda x.M) = FV(M) - \{x\}$
If x is a bound variable, then x is not free.

Evaluation of λ -Terms

Given a function application of the form

$$(\lambda x.M) N,$$

evaluation proceeds by replacing all **free** occurrences of x in the body M with N .

For example,

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

Here, the argument $\lambda z.z$ replaces the variable x , which is free in the body $\lambda y.x$ of the function being applied.

Evaluation of λ -Terms

However,

$$(\lambda x.(\lambda x.x)) \lambda z.z \rightarrow \lambda x.x$$

Here, there is no substitution, since x is not free in the body $\lambda x.x$.

β -Conversion

In general, we write

$$(\lambda x.M) N \rightarrow_{\beta} M[x \mapsto N],$$

where $M[x \mapsto N]$ denotes the result of replacing all **free** occurrences of x in M with N .

This form of evaluation is referred to as **β -conversion**, for which we use the symbol \rightarrow_{β} .

Example

$$\begin{aligned}
 & ((\lambda x. \lambda y. y x) (\lambda z z)) (\lambda u. \lambda v. u) \\
 \beta & \\
 & (\lambda y. y x)[x \leftarrow (\lambda z z)] (\lambda u. \lambda v. u) \\
 = & \\
 & (\lambda y. y (\lambda z z)) (\lambda u. \lambda v. u) \\
 \beta & \\
 & ((\lambda y (\lambda z z))[x \leftarrow (\lambda u. \lambda v. u)]) \\
 = & \\
 & (\lambda u. \lambda v. u) (\lambda z z) \\
 \beta & \\
 & \lambda v. \lambda z. z
 \end{aligned}$$

α -Conversion

Just as with functions, formal parameters simply serve as place-holders, representing possible arguments.

In the λ -calculus, formal parameters (i.e., bound variables) can be renamed. This is referred to as **α -Conversion** (written \Rightarrow_α).

For example,

$$\lambda x. \lambda y. y \quad \alpha \quad \lambda x. \lambda v. v \quad \beta \quad \lambda u. \lambda v. v$$

β -Conversion Again

In general, we write

$$(\lambda x.M) N \quad \beta \quad M[x \leftarrow N],$$

where $M[x \leftarrow N]$ denotes the result of replacing all **free** occurrences of x in M with N , **provided that no free variables in N become bound as a result of this substitution.**

For example,

$$(\lambda x. \lambda y. y x) (\lambda z y) \quad \beta \quad \lambda y. y (\lambda z y)$$

is **not** allowed.

α - and β -Conversion

Sometimes it is necessary to apply α -conversion before β -conversion can be applied.

For example,

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

η -Conversion

A final reduction relation on λ -terms is **η -conversion**. This applies to λ -terms of the form $\lambda x. M x$, where x is not free in M . Such a function takes an argument and applies M to that argument; it is therefore equal to M itself.

This is the idea behind η -conversion (β_η):

$$\lambda x. M x \quad \beta_\eta \quad M$$

provided that x is not free in M .

Example

- $\lambda y. (\lambda x. y x) \quad \beta_\eta \quad \lambda y. y$

But η -conversion is **not** applicable to

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

Computing with λ -Terms

How does the λ -calculus relate to real programming languages such as Haskell?

Summary

Key points:

- λ -terms
- β -conversion
- α -conversion
- η -conversion

Next: Computing with λ -terms