Session 10: λ-calculus and summary

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

λ-calculus

Introduction

- λ-calculus: rule system to describe computations solely via function abstraction and application
- Inspired functional programming languages
- Simplest known Turing-complete programming language

$\rightarrow$ Proof of Turing-completeness:
`https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-11`

Definition of λ-expressions

- Inductive definition to build all λ-expressions:
  - Variable v is a λ-expression.
    - $\rightarrow$ Variables represented by lower-case letters
  - If M is a λ-expression, then $(\lambda v.M)$ is a λ-expression.
    - $\rightarrow$ Abstraction aka definition of a function with parameter v and body M
    - $\rightarrow$ λ denotes start of function definition
  - If M and N are λ-expressions, then (MN) is a λ-expression.
    - $\rightarrow$ Application of M to N
- Note: Functions take exactly one argument, *currying* used to model multi-argument functions

Examples

Valid λ-expressions:

- $x \rightarrow$ a variable
- $(\lambda x.x) \rightarrow$ the identity function
- $((\lambda x.x)e) \rightarrow$ the identity function applied to an expression e
- $(\lambda x.(\lambda y.(xy))) \rightarrow$ nested function, i.e. currying
- $(((\lambda x.(\lambda y.(xy)))a)b) \rightarrow$ nested function applied to expressions a and b

Conventions to avoid ambiguity

- Application is left-associative, e.g., MNP is equivalent to ((M N) P); instead of (M (N P))
- Abstraction is right-associative
- Application has precedence over abstraction, e.g. $\lambda x.\lambda y.xy$ is equivalent to $\lambda x.(\lambda y.(xy))$; instead of $\lambda x.(\lambda y.x)y$

Free vs. bound variables

Bound variables

A variable is *bound* if it occurs in a function that takes a variable of the same name as input. For instance, x in $\lambda x.x$ is bound. A variable binds to the closest function argument considering its enclosing functions.

Free variables

A variable is *free* if it is not bound. For instance, x in $\lambda y.x$ is free.

Transformation rules

β-Reduction to evaluate expressions

β-reduction allows to substitute the argument of an abstraction with the value of an application $((\lambda x.M[x])N) \to (M[x := N])$.

Example 1:

$$(\lambda x.x)a = a$$
$$(\lambda x.y)a = y$$

Example 2:

$$\begin{aligned} ((\lambda x.(\lambda y.(xy)))a)b &= \\ (\lambda y.(ay))b &= \\ ab & \end{aligned}$$

- Normalform: A λ expression is in β-normalform if no β-reduction is possible, i.e., if the expression cannot be reduced any further.

Transformation rules: α-Conversion

α-Conversion to rename variables

α-conversion allows to resolve name conflicts by renaming bound variables as such $(\lambda x.M[x]) \to (\lambda y.M[y])$.

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

- Prevent capturing free variables when β-reducing expressions

$\to$ For instance, β-reduction would change the semantics of the inner function in $(\lambda x.(\lambda y.xy))y$ without prior α-conversion

Summary

Summer exam

- Closed book in-person exam, tests knowledge, comprehension, application and synthesis
- Format: reading and writing Haskell code + conceptual and theory questions

$\Rightarrow$ Practice programming in Haskell

$\Rightarrow$ Think about functional paradigms, look for them elsewhere. Has your mindset changed?

Relevant past paper questions

2023 all

2022 Q1 (not (d)) and Q2

2021 Q1 (not (e))

2020 Q1 and Q2

2019 Q2 (the only Haskell question)

2018 Q1 (b–e, g) (not (a), (f))

Content overview

- Functional programming paradigm
- Types system: built-in types, type checking, polymorphism, type classes, algebraic data types
- Functions: currying, λ-expressions and higher-order functions
- Lists: pattern matching, comprehensions
- Recursion: structure, classification, and complexity
- Evaluation: expression graphs, lazy vs. eager
- Abstractions for computational patterns: functors, foldables, and monads
- λ-calculus: syntax and reduction rules

Functional programming paradigm

- Programming *paradigm* where the building block of computation is the *application of functions* to arguments
- Functional programs specify a data-flow to describe *what* computations should proceed (instead of *how* they should proceed)
- Algebraic programming style dominated by function application and composition

$\Rightarrow$ a functional language is one that *supports* and *encourages* programming in this style

Data types

Type: collection of values

Haskell built-in types

- Int, Integer, Char, String, ...
- Lists `[1,2,3]`
- Tuples `(1,2,3)`

Haskell custom data types

- `type` keyword for synonyms
- `data` keyword for new algebraic types (sum and product types)

Polymorphism

- Polymorphism: functions that are defined generically for many types.
- Types of polymorphism: parametric, ad-hoc, subtype polymorphism
  - Type variables: `length :: [a] -> Int` "a" is a type variable, length is generic over the type of the list.
  - Haskell uses *parametric polymorphism* "generic functions"
- Constraining polymorphic functions: type classes
  - `(+) :: Num a => a -> a -> a` "+ works on any type a as long as that type is numeric"
  - Relevant type classes: `Num` "numeric", `Eq` "equality", `Ord` "ordered", `Functor`, `Foldable`, `Monad`

  ⇒ Include class constraints in type definitions when appropriate

Lists

- Pattern matching: can match literal values but also match a list pattern, and bind the values

```haskell
sumTwo :: Num a => [a] -> a
sumTwo (x:y:_) = x + y
```

- List comprehensions: construct new lists based on generator and guard expressions

```haskell
[ x | x <- [1..5], even x]
```

Recursion

Recursion: a function that calls itself until it reaches a base case.

Definition (Tail recursion)

A function is *tail recursive* if the *last result of a recursive call* is the result of the function itself.

Definition (Linear recursion)

The recursive call contains only a *single* self reference.

Definition (Multiple recursion)

The recursive call contains *multiple* self references.

Definition (Direct recursion)

The function calls *itself* recursively.

Definition (Mutual/indirect recursion)

Multiple functions call *each other* recursively.

Functions

- Saw nameless or anonymous functions (λ-expressions), and syntax
- Formalises idea of functions defined using currying

```haskell
add x y = x + y
-- Equivalently
add = \x -> (\y -> x + y)
```

Definition (Higher order function)

A function that does at least one of the following

- take one or more functions as arguments
- returns a function as its result

- Due to currying, every function of more than one argument is higher-order in Haskell

Generalizing computational patterns

- `Functor` for mappable types
- `Foldable` for types that can be reduced
- `Monads` for types that describe actions to compose and structure computations
- Instances must obey some equational laws

Evaluation strategies

- Lazy evaluation
  - Infinite data structures are fine (as long as we don't try and consider all of their elements)
- Call by name (lazy, outermost) vs. call by value (eager, innermost) $\rightarrow$ contrast with imperative languages
- Think about expression as a graph of computations: multiple different evaluation orders possible

λ-calculus

- λ-calculus: set of rules to transform expressions of the following form
  - v (Variables; lower case letters)
  - (MN) (Application of M to N)
  - $(\lambda v.M)$ (Abstraction aka function with parameter v and body M)
  - with M and N being expressions of the same form
- α-conversion: solving name conflicts by renaming variables
- β-reduction: reducing expressions by applying functions to arguments

Thank you!