Overview

Beginning Steps

Overview

Functional programming in Haskell

- Examine and use existing functions (from prelude)
- How Haskell functions are being evaluated
- Types
- Literate programming
- Write simple Haskell scripts

Overview (cont.)

Functional programming in Haskell --- and -- formal methods

- Mathematical logic and formal methods
- Logical languages
- Specify properties rigorously using a logical language
- Well-formed formulas and their meanings

The End

Function Evaluation and Types

Function Evaluation

Function Application

In **mathematics**, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

f(a,b) + c d

Apply the function f to a and b, and add the result to the product of c and d.

In **Haskell**, function application is denoted using space, and multiplication is denoted using.*

```
f a b + c*d
```

As previously but in Haskell syntax

Moreover, function application is assumed to have **higher priority** than all other operators.

f a + b

Means (f a) + b rather than f (a + b)

Examples

Mathematics	Haskell
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g x)
f(x,g(y))	f x (g y)
f(x)g(y)	f x * g y

Function Evaluation and Types: Function Evaluation

The End

Function Evaluation and Types

Haskell Functions and Types

What Is a Type?

- A **type** is a name for a collection of related values; for example, in Haskell the basic type

Bool

- Contains the two logical values:

False True

Type Errors

Applying a function to one or more arguments of the wrong type is called a **type error**.

```
> 1 + False
error ...
```

1 is a number and False is a logical value, but + requires two numbers.

Types in Haskell

- If evaluating an expression e would produce a value of type t, then e **has type** t, written

$$e :: t$$

- Every well-formed expression has a type, which can be automatically calculated at compile time using a process called **type inference**

- All type errors are found at compile time, which makes programs **safer and faster** by removing the need for type checks at run time
- In GHCi, the **:type** command calculates the type of an expression, without evaluating it:

```
> not False
True

> :type not False
not False :: Bool
```

Basic Types

Haskell has a number of **basic types**, including:

Bool - logical values

Char - single characters

String - strings of characters

Int - fixed-precision integers

Integer - arbitrary-precision integers

Float - floating-point numbers

List Types

- A **list** is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a', 'b', 'c', 'd'] :: [Char]
```

- In general:
  - [t] is the type of lists with elements of type t

Note:

- The type of a list says nothing about its length:

```
[False,True] :: [Bool]

[False,True,False] :: [Bool]
```

- The type of the elements is unrestricted; for example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

Tuple Types

- A **tuple** is a sequence of values of **different** types:

```
(False,True) :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

- In general:
  - (t1,t2,…,tn) is the type of n-tuples whose ith components have type ti for any i in 1…n

Note:

- The type of a tuple encodes its size:

```
(False,True) :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b']) :: (Bool,[Char])
```

Function Types

- A **function** is a mapping from values of one type to values of another type:

```
not :: Bool → Bool

even :: Int → Bool
```

- In general:
  - t1 → t2 is the type of functions that map values of type t1 to values to type t2

Note:

- The arrow $\rightarrow$ is typed at the keyboard as ->
- The argument and result types are unrestricted; for example, functions with multiple arguments or results are possible using lists or tuples:

```
add :: (Int,Int) → Int
add (x,y) = x+y

zeroto :: Int → [Int]
zeroto n = [0..n]
```

Lambda Expressions

Functions can be constructed without naming the functions by using **lambda expressions**.

$$\lambda x \rightarrow x + x$$

The nameless function that takes a number x and returns the result x + x

Why Are Lambdas Useful?

- Lambda expressions can be used to give a formal meaning to functions defined using **currying**
- For example:

$$\text{add } x\ y = x + y$$

means

$$\text{add} = \lambda x \to (\lambda y \to x + y)$$

- Lambda expressions are also useful when defining functions that return **functions as results**
- For example:

```
const :: a → b → a
const x _ = x
```

is more naturally defined by

```
const :: a → (b → a)
const x = λ_ → x
```

- Lambda expressions can be used to avoid naming functions that are only **referenced once**
- For example:

```
odds n = map f [0..n-1]
         where
            f x = x*2 + 1
```

can be simplified to

```
odds n = map (λx → x*2 + 1) [0..n-1]
```

Operator Sections

- An operator written **between** its two arguments can be converted into a curried function written **before** its two arguments by using parentheses
- For example:

```
> 1+2
3

> (+) 1 2
3
```

- This convention also allows one of the arguments of the operator to be included in the parentheses
- For example:

```
> (1+) 2
3

> (+2) 1
3
```

- In general, if ⊕ is an operator, then functions of the form (⊕), (x⊕), and (⊕y) are called **sections**

Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections; for example:

(1+)	- successor function
(1/)	- reciprocation function
(*2)	- doubling function
(/2)	- halving function

Function Evaluation and Types: Haskell Functions and Types

The End

Haskell Scripts

Literate Programming and Haskell

Literate Programming

"Literate programming is a methodology that combines a programming language with a documentation language, thereby making programs more robust, more portable, more easily maintained, and arguably more fun to write than programs that are written only in a high-level language."

—Donald E. Knuth

Literate Programming (cont.)

According to Knuth, the main idea is:

- To treat a program as a piece of literature, addressed to human beings rather than to a computer
- That the program is also viewed as a hypertext document
- An example is:
  - *Jon Bentley, Communications of the ACM, Volume 29, Issue 601 June 1986, pp. 471–483*

Literate Haskell Script

- In Haskell, a program is either with suffix *.hs* or *.lhs.*
- Scripts with the suffix *.lhs* can be written in a style that provides native features to support literate programming
- Regarding literate Haskell script, the guideline can be found in:
  - Literate programming: HaskellWiki

The End

Functional Programs

A Beginning Example: Numbers into Words

Number to Words, Part I

Problem statement (Bird, Section 1.4, pp. 7–12)

- Define a Haskell function such that:
  - *Given a nonnegative number less than one million, it returns a string that represents the number in words*

Number to Words, Part II

Understanding the problem

- Formulate a list of questions that can help you to better understand the given programming problem; answer them (as precisely as you can) if possible
- For example: What are the inputs and their outputs, notations, conditions etc.? Can you write them down?

Number to Words, Part III

Devising a plan

- *Have you seen it seen the problem in a slightly different form? Do you know some utility functions that could be useful? Could you restate the problem? Could you imagine a more accessible related problem, a more specific one, a simpler one? A more general problem? A more special problem? Could you solve a part of the problem?*

Number to Words, Part IV

Carrying out your plan

- *When you have a plan of the solution, implement it. Check each step carefully. Can you check if the step has no obvious errors? Can you prove (or provide convincing evidence) that it is correct?*

Number to Words, Part V

Examining the solution obtained

- When you are totally convinced that your implementation is correct, ask if you can implement the solution differently, or if you can use it for some other problems. Is there a need to refactor your code? That is, to restructure/modify its internal structure without changing its external behavior?

Functional Programs: A Beginning Example (Numbers into Words)

The End

Functional Programs

Numbers to Words: Design and Specification

Design and Specification

- To devise a plan for the program can be divided into two areas
  1. Design
  2. Specification
- And we often work on them together

Design and Specification (cont.)

- Program design in functional programming
  - Program construction emphasizes functions and their application
  - Design a single function using a collection of functions
  - Function composition/decomposition
  - This program that “convert number to words” has a simple pipeline structure

A Pipeline Structure

Pipeline structure: convert6 = combine6 . dights6

(h = convert6; f = dights6; g = combine6)

$h: A \rightarrow C$ (write $h = g \circ f$)

$f : A \rightarrow B$ $\qquad\qquad$ $g : B \rightarrow C$

$$x \rightarrow \boxed{\quad f \quad} \rightarrow f(x) \rightarrow \boxed{\quad g \quad} \rightarrow h(x) = g(f(x))$$

$x \in A$ $\qquad\qquad$ $f(x) \in B$ $\qquad\qquad$ $h(x) = g(f(x)) \in C$

Convert :: Int → String

x   digits6   combine6   name of

(m, n)   of that Integer

Design and Specification

From understanding the problem to devising a solution

- Create and examine input-output examples.
- Use simple mathematical notation to specify the problem precisely and concisely.
- Test and solve small but representative special cases.

Design and Specification (cont.)

From understanding the problem to devising a solution (cont.)

- Apply function decomposition and identify partial functions and helpful utility functions. mod div.
- Combine the partial solutions into the final solution.

Functional Programs: Numbers to Words (Design and Specification)

The End

The Design of a Logical Language

The Language of Propositional Logic

Introduction

- We use a familiar example (propositional logic) to introduce the three areas of knowledge that one should consider when designing/choosing a logical language. They are the:
  1. Syntax
  2. Semantics
  3. Proof theory

  of the logic.

The Language of Propositional Logic

- We describe the underlying world by well-formed formulas.
- Well-formed formulas are built from propositional symbols and logical symbols.
- The meaning of these formulas is specified by the truth tables of the formulas.
- The syntax of the language is the rigorous definitions that specify what well-formed formulas are.

The Language of Propositional Logic

1. Symbols for *atomic propositions*: $p, q, r, s, \ldots$
2. Logical *symbols*: $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (if then)
3. Well-formed formulas (abbrev. as *wff*): They are constructed (via *recursion*) by the following rules:

		Construction Rule
1.	p :	Every propositional atom is a well-formed formula
2.	$\neg$:	if ϕ is a wff, then so is $(\neg\phi)$.
3.	$\wedge$:	if ϕ, ψ are wffs, then so is $(\phi \wedge \psi)$
4.	$\vee$:	if ϕ, ψ are wffs, then so is $(\phi \vee \psi)$
5.	$\rightarrow$:	if ϕ, ψ are wffs, then so is $(\phi \rightarrow \psi)$

Syntax of Propositional logic

1. (Base case) p, q, r, s

2 (Rules) By recursion.

Backus Naur Form

1. Backus Naur Form (BNF) refers to a compact way to formally specify a logical language.
2. BNF for well-form formula (*wff*) in propositional logic:

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi)$$

   where Rule: 1 2 3 4 5

   - p stands for any atomic proposition
   - Each occurrence of ϕ to the right of $::=$ stands for any already constructed formula.
3. The BNF notation are used when specifying the grammar for a context free language (e.g. specifying the syntax of a programming language.)

Semantics

Suppose that ϕ is a well-formed formula. When do we say that is true?

- We can construct a truth table for ϕ according to the definition of the logical connectives. Each row represents a possible world (a model) where ϕ is true.
- We can also compute the truth value ϕ for each model by using recursion (the construction rule given previously).

Proof Theory

Natural deduction

- Formulate inference rules as the basis for deduction—for example, the modus ponens rule.
  - Hypotheses: H1: ϕ, H2: $\phi \rightarrow \phi'$
  - Conclusion: C : ϕ'

$$\frac{H_1 \quad H_2}{C}$$

Modus Ponens

- Apply inference rules to carry out reasoning, showing what can be inferred from a list of hypotheses.

Proof

1. H_1

2. H_2

3. C

⋮

n.

(modus ponens, 1, 2)

⋮

$\left(\underbrace{\qquad}_{\text{rules used}}, \underbrace{\text{previous}}_{\text{cases}} \right)$

The Design of a Logical Language: The Language of Propositional Logic

The End

The Design of a Logical Language

Other Logical Languages

First-Order Logic, Part I

- What is first-order logic? Did you use it before?
- Express the following property
  - An integer *n* greater than one is a *prime* if its *only positive factors* are one and the number itself… (*)
  - Formally, as *prime* (*n*), that is:

$$\overbrace{\textit{Prime}}^{\text{predicate}}\ (n) \text{ is true if and only if } n \text{ satisfy } (*)$$

First-Order Logic, Part II

- Another name: predicate logic
- We often use the language when specifying mathematical (or formal) properties, such as specifying when a positive integer is a prime
- Their language (syntax, semantics, proof theory) can be formulated rigorously (but beyond the score of our class)
- You often use it informally as in many computer science texts when we specify properties (e.g., being a prime) and code it up

First-Order Logic, Part III

- Their languages have:
  - Quantifiers
    - $\forall$ (for any, for all etc.), $\exists$ (there exists, for some etc.)
  - Usual logical symbols, such as $\neg$ (not), $\wedge$ (and), $\rightarrow$ (if-then), $\vee$ (or), etc.
  - Nonlogical symbols, such as constants, functions, and predicate symbols

Prime (n)

The Property Prime (n)

$$\forall\ n\ [(n > 1) \rightarrow (\exists\ r)\ [\text{factor}\ (r, n) \rightarrow ((r = 1) \vee (r = n))]]$$

- Note that:
  - The factor is a predicate “symbol” to represent the property “*r* is a factor of *n*”
  - We can translate this property as Haskell code easily

The End

Access-Control Logic: The Language

Definition

What Do We Need?

We need a language that lets us describe precisely and reason about:

- Principals
- Requests principals make
- Access-control policies
- Statements principals make
- Authorities and their jurisdiction
- Certificates
- Credentials
- Trust assumptions

Principal Names

Principal names (the set **PName**): refer to simple principals

- *Alice*
- *Bob*
- The key K_{Alice}
- The PIN 1234
- The userid–password pair $\langle alice, bAdPsWd! \rangle$

Compound principals also possible ("Alice and Bob together", "Alice quoting Bob"), but we won't talk about them right now.

Logical Formulas

- Relevant primitive sets (and meta-variables)
  - Principal expressions: $A, B, P, Q \in \textbf{Princ}$ (= **PName** for now)
  - Propositional variables: $p, q, r \in \textbf{PropVar}$
- Logical formulas ($\varphi \in \textbf{Form}$) given by:

$$
\begin{aligned}
\varphi \quad ::= \quad & p \mid \neg\varphi \\
& \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \mid (\varphi_1 \supset \varphi_2) \mid \varphi_1 \equiv \varphi_2 \\
& \mid (P \Rightarrow Q) \mid (P \text{ says } \varphi) \mid (P \text{ controls } \varphi)
\end{aligned}
$$

rules for prop. logic

(a) (b) (c)

- Sample well-formed formulas:

$$r \qquad ((\neg q \wedge r) \supset s) \qquad (\mathit{Alice} \text{ says } (r \vee (p \supset q)))$$

- **Not** well-formed formulas:

$$\neg \mathit{Alice} \qquad (\mathit{Alice} \Rightarrow (p \wedge q)) \qquad (\mathit{Alice} \text{ controls } \mathit{Bob})$$

$\neg$ Alice

$\neg$ ($\uparrow$)

" Form"

But

Alice is a principal.

principal is Not a ~~pr~~ "Form" expression

Access-Control Statements

The symbols says, controls and, $\Rightarrow$ are introduced to formulate access requests.

- P says ϕ.
- P controls ϕ.
- P $\Rightarrow$ Q (P speaks for Q).

Principal Expressions

- BNF specification
  - Princ ::= PName / Princ & Princ / Princ | Princ
- P, Q are principal expressions
  - P & Q : P in conjunction with Q
  - P | Q : P quotes Q

① single principal

②

Interpreting Principal Expressions

- P & Q denotes the abstract principal “P in conjunction with Q”
- P | Q denotes the abstract principal “P quoting Q”; for example:

President & Congress denotes the abstract principal
“the President together with Congress.”

Reporter | Source denotes the abstract principal
“the reporter quoting her source.”

A Beginning Example

- P: Macy (a principal); Q: Al (a principal)
- ϕ: the action ‘read file foo’
- The following are well-formed access-control statements:
  - P says ϕ: Macy’s request to read the file foo
  - P controls ϕ: Macy’s entitlement to read the file foo
  - P $\Rightarrow$ Q: Macy speaks for Al

Access-Control Logic: The Language (Definition)

The End

Access-Control Logic: The Language

Examples

Revisit the Beginning Example

- P: Macy (a principal); Q: Al (a principal)
- ϕ: the action ‘read file foo’
- The following are well-formed access-control statements:
  - P says ϕ: Macy’s request to read the file foo
  - P controls ϕ: Macy’s entitlement to read the file foo
  - P $\Rightarrow$ Q: Macy speaks for Al

Logical Formulas

- Relevant primitive sets (and meta-variables)
  - Principal expressions: $A, B, P, Q \in \textbf{Princ}$ (= **PName** for now)
  - Propositional variables: $p, q, r \in \textbf{PropVar}$
- Logical formulas ($\varphi \in \textbf{Form}$) given by:

$$
\begin{array}{rcl}
\varphi & ::= & p \mid \neg\varphi \\
 & & \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \mid (\varphi_1 \supset \varphi_2) \mid \varphi_1 \equiv \varphi_2 \\
 & & \mid (\underline{P} \Rightarrow \underline{Q}) \mid (\underline{P} \text{ says } \varphi) \mid (\underline{P} \text{ controls } \varphi)
\end{array}
$$

principal expression is NOT Logical form. ($\notin$ Form)

- Sample well-formed formulas:

$$r \qquad ((\neg q \wedge r) \supset s) \qquad (Alice \text{ says } (r \vee (p \supset q)))$$

- **Not** well-formed formulas:

$$\neg Alice \qquad (Alice \Rightarrow (p \wedge q)) \qquad (Alice \text{ controls } Bob)$$

Well-Formed Access-Control Statements

An access control statement	Derivation
(Jill says (r ⊃ (p ∨ q))) Note: A principal expression can be a part of a well-formed access-control statement, but it is not a well-formed access-control statement.	$\mathbf{Form} \leadsto (\mathbf{Princ} \text{ says } \mathbf{Form})$ $\leadsto (\mathbf{PName} \text{ says } \mathbf{Form})$ $\leadsto (\mathit{Jill} \text{ says } \mathbf{Form})$ $\leadsto (\mathit{Jill} \text{ says } (\mathbf{Form} \supset \mathbf{Form}))$ $\leadsto (\mathit{Jill} \text{ says } (\mathbf{PropVar} \supset \mathbf{Form}))$ $\leadsto (\mathit{Jill} \text{ says } (r \supset \mathbf{Form}))$ $\leadsto (\mathit{Jill} \text{ says } (r \supset (\mathbf{Form} \vee \mathbf{Form})))$ $\leadsto (\mathit{Jill} \text{ says } (r \supset (\mathbf{PropVar} \vee \mathbf{Form})))$ $\leadsto (\mathit{Jill} \text{ says } (r \supset (p \vee \mathbf{Form})))$ $\leadsto (\mathit{Jill} \text{ says } (r \supset (p \vee \mathbf{PropVar})))$ $\leadsto (\mathit{Jill} \text{ says } (r \supset (p \vee q)))$

Non-Well-Formed Statements

Examples

Non-well-formed access-control statement	**Reason**
Orly & Mitch	A principal expression, not an access-control formula
$\neg$ Orly	The negation operator : must precede an access-control formula
(Orly) $\Rightarrow$ (p $\wedge$ q))	Because (p $\wedge$ q) is not a principal expression, the speaks-for operator $\Rightarrow$ must appear between two principal expressions

Text's Convention

- Distinguish between principal names and propositional variables through capitalization. Specifically, we will use capitalized identifiers—such as Josh and Reader—for simple principal names. We will use lowercase identifiers—such as *r*, *write,* and *rff*—for propositional variables.
- Use parentheses (see last paragraph in ACST 2.2.1 and last paragraph in ACST 2.2.2 before the exercises).

The End

Weekly Summary

Beginning Steps

Functional Programming

- Be familiar with the programming environment.
- Run simple Haskell functions (programs) using Glasgow Haskell Compiler (interactive mode).
- Avoid type errors. Use basic, list and tuple types, prelude functions to write and compose Haskell programs.
- Examine the guiding example “number to words” to develop good habits for program development.

Secured Systems

- Mathematical logic from formal methods provides a firm foundation of access control in secured systems.
- The language of propositional logic and access-control logic can be specified rigorously to capture the intended meanings of the access-control properties

Weekly Summary

The End