Overview

Formal Methods and Programming

Overview

- Formal methods for program verification and testing
- Case studies: sorting and Caesar ciphers
- Data declarations in Haskell
- Haskell representation of access-control formulas and their implementation

Overview

The End

Program Verification and Testing

Introduction

Introduction

Motivations

- Develop rigorous methods to verify the correctness of computer systems: software, hardware, or a combination.
- Systems can either be safety critical, commercially critical, or mission critical.
- Verification methods can establish whether a description of a system satisfies a specification.

Formal Verification

Techniques typically include:

- A description language for modelling systems
- A specification language for describing the properties to be verified
- A verification method to test if the description of a system property satisfies a specification

Formal Verification (cont.)

Approaches to verification

- Proof-based or model-based
- Automatic or semi-automatic
- Intended domain of application (e.g., software or hardware, sequential or concurrent, etc.)
- Stage in program development: pre-development or post-development

Program Verification and Testing

The End

Case Study: Sorting

Sorting Methods

Sorting Methods

Interested in the following comparison-based sort methods

- Insertion sort
- Merge sort
- Quick sort

4-4-2.1hs

Insertion Sort

- Should work for any list (type [a]) where the elements are ordered (a member of the type class Ord)
  - Implement the insert operation as a function insert
  - Implement insertion sort as a function isort
- Use recursion in both cases

The Functions Insert and Isort

Steps	The function insert	The function isort
Define the type.	Int -> [Int] -> [Int]	[Int] -> [Int]
Enumerate the cases.	insert x [ ] Insert x (y:ys)	isort [ ] isort (x:xs)
Define the simple cases.	insert x [ ] = [ ]	isort [ ] = [ ]

Define the other cases.	insert x (y:ys)
	    | x <= y = x: y:ys
	    | otherwise = y: insert x ys
		isort (x:xs) = insert x (isort xs)

Generalize and simplify.	Ord a => a -> [a] -> [a]	Ord a => a -> [a] -> [a]

Merge Sort

- Should work for any list (type [a]) where the elements are ordered (a member of the type class Ord)
  - Implement the merge operation as a function merge
  - Implement merge sort as a function msort
- Use recursion in both cases

The Sorted Function

Property: Given any input list (elements are ordered), each of the sorting functions will always return a sorted list.

1. ```
   pairs :: [a] -> [(a,a)]
   pairs xs = zip xs (tail xs)
   ```
2. ```
   sorted :: Ord a => [a] -> Bool
   sorted xs = and [x <= y | (x,y) <- pairs xs]
   ```

Other Properties

Discussions

→ After sorting the lists,

the number of elements remain

unchanged.

Case Study: Sorting

The End

Case Study: Caesar Ciphers

Caesar Ciphers: Introduction

Background

Cipher: an encryption method that transform a string (plaintext) to another string (ciphertext) to disguise its contents

- Caesar cipher: a classical cipher
- This case study: implement Caesar cipher and show how to crack it

Cipher.hs

- Demonstrate string comprehensions (use functions from the library Data.Char in the implementation)

Caesar Cipher :-

Shift function.

shift 2 "ABCD" → encrypt

A B C D

A B C D E . .

shift (-2)

← reverse direction

Implementation Highlights

Utility functions	Encoding and decoding methods	Frequency analysis
Data.Char: ord chr isLower **Prelude:** fromInteger drop take length sum	Note the use of string comprehensions let2int character → integer int2let integer → character shift → Encode → main function to support encryption & decryption	Analyze from a large volume of text to obtain approximate percentage frequencies of the 26 letters. Given an encoded string, but not the shift factor that was used to encode it, determine the shift factor in order that we can decode the string. Use the position of the minimum chi-square value as the guess for the shift factor.

Case Study: Caesar Ciphers

The End

Data Declarations in Haskell

Introduction

- Type declarations can be used to make other types easier to read; for example, given

```
type Pos = (Int,Int)
```

we can define

```
origin :: Pos
origin = (0,0)

left :: Pos → Pos
left (x,y) = (x-1,y)
```

- Like function definitions, type declarations can also have **parameters;** for example, given

```
type Pair a = (a,a)
```

we can define

```
mult :: Pair Int → Int
mult (m,n) = m*n

copy :: a → Pair a
copy x = (x,x)
```

- Type declarations can be nested:

```
type Pos = (Int,Int)

type Trans = Pos → Pos
```

✓

- However, they cannot be recursive:

```
type Tree = (Int,[Tree])
```

✗

Data Declarations in Haskell

The End

Data and Type Declarations in Haskell

Data Declarations

4-6-3.lhs

Data Declarations

A completely new type can be defined by specifying its values using a **data declaration**.

```
data Bool = False | True
```

Bool is a new type with two new values: False and True.

Note:

- The two values False and True are called the **constructors** for the type Bool.
- Type and constructor names must always begin with an uppercase letter.
- Data declarations are similar to context-free grammars. The former specifies the values of a type; the latter the sentences of a language.

- Values of new types can be used in the same ways as those of built in types; for example, given

```
data Answer = Yes | No | Unknown
```

we can define

```
answers :: [Answer]
answers = [Yes,No,Unknown]

flip :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

- The constructors in a data declaration can also have parameters; for example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define

```
square :: Float → Shape
square n = Rect n n

area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Note:

- Shape has values of the form Circle r, where r is a float, and Rect x y, where x and y are floats
- Circle and Rect can be viewed as **functions** that construct values of type Shape:

```
Circle :: Float → Shape

Rect :: Float → Float → Shape
```

- Not surprisingly, data declarations themselves can also have parameters; for example, given

```
data Maybe a = Nothing | Just a
```

we can define

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be **recursive**.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat $\rightarrow$ Nat.

Note:

- A value of type Nat is either Zero, or of the form Succ n where n :: Nat; that is, Nat contains the following infinite sequence of values:

```
Zero

Succ Zero

Succ (Succ Zero)

⋮
```

- We can think of values of type Nat as **natural numbers**, where Zero represents 0 and Succ represents the successor function 1+.
- For example, the value

Succ (Succ (Succ Zero))

represents the natural number

1 + (1 + (1 + 0)) = 3

Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int :: Nat → Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int → Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

- Two naturals can be added by converting them to integers, adding, and then converting back:

```
add :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

- However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

For example:

```
  add (Succ (Succ Zero)) (Succ Zero)
=
  Succ (add (Succ Zero) (Succ Zero))
=
  Succ (Succ (add Zero (Succ Zero))
=
  Succ (Succ (Succ Zero))
```

Note:

- The recursive definition for add corresponds to the laws 0+n = n and (1+m)+n = 1+(m+n).

Type Classes

- A *type class* is a collection of types that support certain overloaded operations called *methods*
- Haskell provides several basic classes that are built into the language; the most used are:
  - Num, Eq, Ord, Show, Read, Integral

Recap: Overloaded Functions

A polymorphic function is called **overloaded** if its type contains one or more class constraints.

(+) :: Num a ⇒ a -> a -> a

For any numeric type a, (+) takes two values of type a and returns a value of type a.

Note:

- Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> ‘a’ + ‘b’
ERROR
```

a = Int

a = Float

Char is not a numeric type

Haskell has a number of type classes, including:

Num	- Numeric types
Eq	- Equality types
Ord	- Ordered types

For example:

```
(+) :: Num a ⇒ a → a → a

(==) :: Eq a ⇒ a → a → Bool

(<) :: Ord a ⇒ a → a → Bool
```

Type Classes

- A *new class* can be declared using a mechanism.
- The declaration states that, for a type a to be an instance of the class, it must support a list of operators of the specified types (see the example of Eq in Section 8.5 for details).
- When new types are declared, it is usually appropriate to make them into instances of a few built-in classes in the form of the deriving mechanism (see the example of type Bool in Section 8.5 for details).

Data and Type Declarations in Haskell

The End

Access-Control Formulas: Specification

Principal Expressions and Access-Control Formulas

Principal Expressions

Mathematical definition	Haskell definition

$Princ ::= PName \ / \ Princ \ \& \ Princ \ / \ Princ \mid Princ$	`data Prin = Name String`
	`           | Together Prin Prin`
	`           | Quote Prin Prin`
	`   deriving (Show, Eq)`

Principal

data Prin = .. | .. | ..

deriving (Show, Eq)

(See 4-8-1.lhs)

Access-Control Formulas

Mathematical definition	Haskell definition

$\mathbf{Form} ::= \mathbf{PropVar} \;/\; \neg\,\mathbf{Form} \;/\; (\mathbf{Form} \vee \mathbf{Form}) \;/\; (\mathbf{Form} \wedge \mathbf{Form}) \;/\; (\mathbf{Form} \supset \mathbf{Form}) \;/\; (\mathbf{Form} \equiv \mathbf{Form}) \;/\; (\underbrace{\mathbf{Princ}} \Rightarrow \underbrace{\mathbf{Princ}}) \;/\; (\mathbf{Princ}\ \mathsf{says}\ \mathbf{Form}) \;/\; (\mathbf{Princ}\ \mathsf{controls}\ \mathbf{Form})$
 Part 2
 Where:
 $\mathbf{Princ} ::= \mathbf{PName} \;/\; \mathbf{Princ}\ \&\ \mathbf{Princ} \;/\; \mathbf{Princ}\ \vert\ \mathbf{Princ}$
 (define principal expressions)
	`data Form = Var Char`
	`| Not Form`
	`| Or Form Form`
	`| And Form Form`
	`| Imply Form Form`
	`| Equiv Form Form`
	`| Says Prin Form`
	`| Contr Prin Form`
	`| For Prin Prin`
	`deriving Show`
	part 1
	 part 2
	
	`data Prin = Name String`
	`| Together Prin Prin`
	`| Quote Prin Prin`
	`deriving (Show, Eq)`

Access-Control Formulas: Specification

The End

Weekly Summary

Formal Methods and Programming

Summary

Examples of applying formal methods in programming

- Implement sorting methods by following its specification.
- Create tests for verifying the implementation of Caesar ciphers.
- Use the data declaration mechanisms in Haskell to define new data types (including recursive data types).

Summary (cont.)

Examples of applying formal methods in programming

- Use the data declaration mechanisms in Haskell to translate the specification of principal expressions and access-control formulas to Haskell code.
- Create tests to verify the Haskell implementation of principal expressions and access-control formulas.

Weekly Summary

The End