

Chapter 1: Problem Solving, Programming, and Calculation

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

August 17, 2007

SoE Chapter 1: Problem Solving, Programming, and Calculation (revised 2007-08-17)

Computation by Calculation

In “ordinary” programming languages, computation alters state by writing new values to program variables:

State before: $x = 3, y = 4$
 $x = 17 + x + y$
State after: $x = 24, y = 4$

This means you perform computations in *sequence*

There is a *control flow* that decides this sequence:

```
for(i=0,i++,i<17)
{ if (j > i) then x++ else y++;
  x = y + j;
}
```

Overview

Basic concepts of functional programming: computation by calculation, values, expressions, types, declarations

First glance of Haskell: introduction, basic language elements

Some simple functional programming examples/exercises

SoE Chapter 1: Problem Solving, Programming, and Calculation (revised 2007-08-17)

In (pure) functional languages, computation is simply *calculation*:

$$\begin{aligned} 3 * (9 + 5) &\implies 3 * 14 \\ &\implies 42 \end{aligned}$$

Forget about assignments, sequencing, loops, if-then-else, ...

Forget everything you know about programming! (Well, almost)

You might wonder how to do useful stuff just by calculating, we'll see later ...

Functions

In functional languages we have:

- *functions*
- *recursion* (instead of loops)
- *expressive data types* (much more than numbers)
- *advanced data structures*

and calculation on top of that

Functions, mathematically: sets of pairs where no two pairs can have the same first component:

$$f = \{(1, 17), (2, 32), (3, 4711)\}$$

$$f(2) = 32$$

Or, given the same argument the function always returns the same value (cannot have $f(2) = 32$ and $f(2) = 33$ at the same time)

Functions model *determinism*: that outputs depend predictably on inputs

Something we want to hold for computers as well . . .

Recursion

Defining a function by writing all pairs can be very tedious

Often defined by simple *rules* instead

$$\text{simple}(x, y, z) = x \cdot (y + z)$$

These rules express *abstraction*: that a similar pattern holds for many different inputs

(“For all x, y, z , $\text{simple}(x, y, z)$ equals $x \cdot (y + z)$ ”)

Abstraction makes definitions shorter and easier to grasp

A good property also for software, right?

Mathematical functions are often specified by *recursive* rules

Recursion means that a defined entity refers to itself in the definition

This seems circular, but can make sense

Example: the factorial function “!” on natural numbers

$$0! = 1$$

$$n! = n \cdot (n - 1)!, \quad n > 0$$

Recursion corresponds to loops in ordinary programming

Pure Functional Languages

Pure functional languages implement mathematical functions

A functional language is pure if there are no *side-effects*

A side effect means that a function call does something more than just return a value

An example in C:

```
int f(int x)
{
    n++;
    return(n+x);
}
```

In pure functional languages, functions are specified by side-effect free rules (declarations)

In Haskell:

```
simple x y z = x*(y+z)
```

Each rule defines a calculation for any actual arguments:

```
simple 3 9 5  =>  3 * (9 + 5)
              =>  3 * 14
              =>  42
```

Just put actual arguments into the right-hand side and go!

Compare this with an execution model that must account for side-effects

Side effect for f : global variable n is incremented for each call

This means that f returns different values for different calls, even when called with the same argument

Much harder to reason mathematically about such functions: for instance,

$$f(17) + f(17) \neq 2 * f(17)$$

Side effects requires a more complex model, and thus makes it harder to understand the software

Exercise

Calculate `simple (simple 2 3 4) 5 6`

Note that we can do the calculation in different order

Do we get the same result?

More on this later ...

Expressions, Values, and Types

The mathematical view makes it possible to *prove* properties of programs

When calculating, all intermediate results are *mathematically equal*:

$$\text{simple } 3 \ 9 \ 5 = 3 * (9 + 5) = 3 * 14 = 42$$

For instance, prove that $\text{simple } x \ y \ z = \text{simple } x \ z \ y$ for all x, y, z :

$$\begin{aligned} \text{simple } x \ y \ z &= x * (y + z) \\ &= x * (z + y) \\ &= \text{simple } x \ z \ y \end{aligned}$$

We cannot do this for functions with side-effects

Calculation is performed on *expressions*:

$$\text{simple } (\text{simple } 2 \ 3 \ 4) \ 5 \ 6$$

Expressions are calculated into *values*:

$$\text{simple } (\text{simple } 2 \ 3 \ 4) \ 5 \ 6 \implies 154$$

Values are also expressions, which cannot be calculated any further

We said a calculation always ends in a value

But what about x defined below?

$$x = x+1$$

$$x \implies x+1 \implies (x+1)+1 \implies ((x+1)+1)+1 \implies \dots$$

Calculating x yields a never-ending calculation!

No value will ever be returned

(In reality, the Haskell system will break the computation when it is out of memory.)

We denote this “no-value” by \perp

\perp is called “bottom” (for mathematical reasons), think of it as “no information”

Types

Many programming languages have *types*

Types help avoiding certain programming errors, like adding an integer with a character

A programming language can be:

- *strongly typed*: every program part must have a legal type
- *weakly typed*: every program part can have a legal type, but need not have
- *untyped*: no types exist

A First Introduction of Haskell

Haskell is a strongly typed, purely functional language

It uses *lazy evaluation* (don't compute anything before it's needed)

It is *higher order* (functions are ordinary data)

It has a *polymorphic* type system, with *type inference*

It has *type classes* (somewhat similar to classes in object-oriented languages)

It has a lot of *syntactic facilities* to help write clear and understandable programs

Haskell implementations

Three maintained public domain implementations exist:

- hugs, an interpreter
- GHC, an optimizing compiler
- nhc, a compiler producing compact code

All are freely available from www.haskell.org

In this course we use hugs, but you are free to try the others

(The graphics library for the course book requires hugs, though)

Basic Language Elements of Haskell

A first introduction:

- Values
- Types (atomic values, composed values)
- Operators and predefined functions
- Other syntactic forms

Numerical Types

Haskell has a number of numerical types:

<code>Integer</code>	Integers of arbitrary size (0, -3, 5487357384578349545, ...)
<code>Int</code>	Integers of limited size (1, -3, 54873, ...)
<code>Float</code>	Single precision floats (1, 1.0, 3.14159, 3.2E3, ...)
<code>Double</code>	Double precision floats (1, 1.0, 3.14159, 3.2E3, ...)
<code>Rational</code>	Exact rational numbers (1, 3.5, 7 % 2, 3.2E3, ...)

Functions and operators on numeric types are the usual ones:

+, -, * (all numeric types), / (not integer types), ** (floating-point exponentiation), ^ (exponentiation with integer)

Most “typical” numerical functions

Numerical expressions look like in most languages:

`x + 7*y`

`3.14159/(x + 1.0) - 33`

(`x + 7*y` is the same as `x + (7*y)`, since `*` binds stronger than `+`)

Note that integer constants like `17` can assume any numeric type, depending on context

So in `x+17`, `17` will get the same type as `x`

Booleans

Type `Bool` for the two booleans values `True`, `False`

Boolean operators and functions: `&&` (and), `||` (or), `not`

Relational operator returning a boolean value: `==`, `/=`, `<`, `<=`, `>=`, `>`

Can compare elements from any “comparable” type (more on this later)

Characters

Type `Char` for characters

Syntax: `'a'`, `'b'`, `'\n'` (newline), ...)

Characters are elements in *strings*

More on strings later

Conditional

Haskell has a conditional if-then-else expression:

`if True then x else y` \implies `x`

`if False then x else y` \implies `y`

So we can write expressions like

`if x > 0 then x else -x`

However, the two branches must have the same type

Thus, `if x > 0 then 17 else 'a'` is illegal

Functions

Functions take a number of arguments and return a result

Some predefined ones, and you can define your own

A little unusual syntax: no parentheses around arguments

```
sqrt 17
```

```
mod 17 3
```

(There are good reasons for this syntax, more on this later)

Declarations

You can define your own entities

Entities of any type can be defined by a *declaration*

```
pi = 3.14159 defines pi to be a floating-point constant
```

```
simple x y z = x*(y+z) defines simple to be a function in three arguments
```

The space between function and argument can be seen as a special operator: *function application*

Function application binds harder than any other operator

Thus, $f\ x + y$ means $(f\ x) + y$, not $f\ (x + y)$

(Common beginner's mistake to forget this)

Types of Defined Entities

Note that we did not give any types in the definitions

Haskell manages to find types automatically!

This is called *type inference*

However, we can also give explicit types (sometimes useful)

```
e :: t declares e to have type t
```

For instance:

```
pi :: Float declares pi to be a single precision float
```

```
pi :: Double declares pi to be a double precision float
```

What about `pi :: Char`?

Recursive Definitions

As a first exercise, recall the factorial function:

$$0! = 1$$

$$n! = n \cdot (n - 1)!, \quad n > 0$$

Define it in Haskell!

(Answer on next slide)

Factorial in Haskell

```
fac n = if n == 0 then 1 else n * fac (n-1)
```

Exercise:

Calculate `fac 3`

```
fac 3 => if 3 == 0 then 1 else 3 * fac (3-1)
      => if False then 1 else 3 * fac (3-1)
      => 3 * fac (3-1)
      => 3*(if (3-1) == 0 then 1 else (3-1) * fac ((3-1)-1))
      => 3*(if 2 == 0 then 1 else 2 * fac (2-1))
      => 3*(if False then 1 else 2 * fac (2-1))
      => 3*2 * fac (2-1)
      ...etc...
      => 3*2*1 * fac (1-1)
      => 3*2*1*1
      => 6
```

Eventually we'll reach the "base case" `fac 0`. This is what makes recursion work here!

Pattern Matching in Declarations

Another way to write the factorial function: use *patterns*

Haskell allows to define functions case-by-case, for different forms of the argument

Every case has a *pattern* for the argument

Cases are checked in order, the first that matches the argument is selected

For "atomic" types like numeric types, the possible patterns are constants

So we can define `fac` like this instead:

```
fac 0 = 1
fac n = n * fac (n-1)
```

Nontermination and Errors

Now what about `fac (-1)`?

`fac(-1) ⇒ (-1)*fac(-2) ⇒ (-1)*(-2)*fac(-3) ⇒ ...`

Infinite recursion! Will never terminate. (Same problem as with `x = x+1`)

Thus, `fac(-1) = ⊥`

Remember `fac` is really just defined for natural numbers, not for negative numbers

It's good practice to have controlled error handling of out-of-range arguments

The Haskell Error Function

Haskell has an `error` function, that when executed prints a string and stops the execution

E.g.,

```
error "You cannot input this number to this function"
```

(Strings in Haskell are written within quotes, like `"Hello world"`)

Accumulating Arguments

A version of `fac` with error handling:

```
fac 0 = 1
fac n = if n > 0 then n * fac (n-1)
       else error "Negative argument to fac"
```

Would we be able to write this case-by case (`n > 0, n == 0, n < 0`)?

Not by pattern-matching, but there are other ways. More on this later

Another way to define the factorial function in Haskell:

```
fac n      = fac1 1 n
fac1 acc 0 = acc
fac1 acc n = fac1 (n*acc) (n-1)
```

This solution uses a help function `fac1` with two arguments

The second argument is an *accumulating argument*, where we successively collect the result

Note similarity with a loop with two variables:

```
while (n \= 0)
{
  acc = n*acc;
  n = n-1;
}
```

Exercise: calculate `fac 3` with this new definition

`let`-expressions are ordinary expressions and return values, can be used wherever ordinary expressions can be used:

```
17 + let x = fac 3 in x + 3*x ==>
17 + let x = 6 in x + 3*x ==> 17 + (6 + 3*6) ==> 41
```

Also note that the defined entity (`x` here) only needs to be computed once – saves work!

So `let` can be used also to save work when the same result is to be used many times

Local Definitions

The `fac` version with accumulating argument uses a help function `fac1`

This function is globally defined

However, only used by `fac`

We may want to *hide* it in the definition of `fac`

Haskell has a `let`-construct for *local* definitions:

```
fac n      = let fac1 acc 0 = acc
              fac1 acc n = fac1 (n*acc) (n-1)
              in fac1 1 n
```

Defines `fac1` locally in the expression after “`in`”, which is the right-hand side in the declaration

Function Types

A function that takes an argument of type `a` and returns a value of type `b` has the *function type* `a -> b`

For instance, `fac :: Integer -> Integer`

(Note resemblance with mathematical notation)

What about functions with several arguments?

```
simple x y z = x*(y+z)
simple :: Integer -> Integer -> Integer -> Integer
```

Last type is result type, preceding types are the respective argument types

(We'll explain later why multi-argument function types look this way)

Types for Data Structures

Haskell has two predefined types for data structures: *tuples* and *lists*

Lists are very important data structures in functional programming

They are sequences of elements

Lists can be arbitrarily long, but (in Haskell) *all elements must be of the same type*

If a is a type, then $[a]$ is the type “list of a ”

E.g. $[Integer]$, $[Char]$, $[[Integer]]$, $[Float \rightarrow Integer]$

A list can contain elements of *any* type (as long as all have the same type)

Constructing Lists

Lists are built from:

- the *empty list*: $[]$
- the “*cons*” operator, which puts an element in front of a list: $:$

Example: $1 : (2 : (3 : []))$

$:$ and $[]$ are called *constructors*: they “construct” data structures

(Constants like 17 and $'x'$ are also constructors, but they only construct themselves)

$1 : 2 : 3 : []$ is same as $1 : (2 : (3 : []))$ (“ $:$ ” is *right-associative*)

$[1, 2, 3]$ is another shorthand for $1 : (2 : (3 : []))$

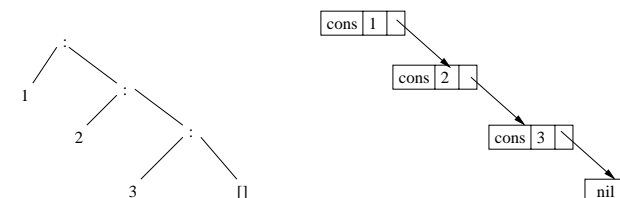
The first element of a nonempty list is the *head*:

$\text{head } [1, 2, 3] \Rightarrow 1$

The list of the remaining elements is the *tail*

$\text{tail } [1, 2, 3] \Rightarrow [2, 3]$

To the left is a graphical picture of $[1, 2, 3]$ as an expression tree:



Underneath, there is a linked data structure (shown to the right)

In conventional languages you’d have to manage the links yourselves. Functional programming systems handle them automatically

Some List Functions

Haskell has many builtin functions on lists. (See Ch. 23 in the book.) Let's look at some and try to program them, as an exercise

`length xs`, computes the length of the list `xs`

`length ['a','b','c']` \Rightarrow 3

`take n xs`, returns list of first `n` elements from the list `xs`

`take 2 ['a','b','c']` \Rightarrow ['a','b']

`sum xs`, sums all the numbers in `xs`

`sum [1,2,3]` \Rightarrow 6

(Solutions on next slide)

```
length [] = 0
length (x:xs) = 1 + length xs
```

```
take 0 xs = []
take n [] = error "taking too many elements"
take n (x:xs) = x : take (n-1) xs
```

```
sum [] = 0
sum (x:xs) = x + sum xs
```

Note the pattern-matching!

The pattern `(x:xs)` matches *any list that is constructed with a cons*

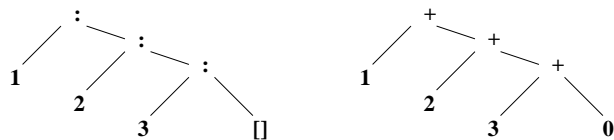
`x` gets bound to the head, and `xs` to the tail

An Observation on `sum`

We have:

`sum [1,2,3] = sum 1:(2:(3:[]))` \Rightarrow `1+(2+(3+0))` \Rightarrow 6

Note the similarity between tree for list `1:(2:(3:[]))` and sum expression for `1+(2+(3+0))`:



`sum` basically replaces `:` with `+` and then calculates the result

This is a common pattern! We'll get back to this when we treat higher-order functions

Also: in a sense, data structures (like lists) in Haskell are just expressions where the "operators" (= constructors) cannot be calculated: thus, the constructors are left in the result (where they build the data structure)

Tuples

Tuples are similar to records, or objects

A tuple is like a container for data with a fixed number of slots

An example: `('a', 17, 3.14159)`

This is a three-tuple whose first component is a character, the second an integer, and the third a floating-point number

It has the *tuple type* `(Char, Integer, Float)`

Tuples can contain *any* type of data, for instance:

`(fac, (17, 'x')) :: (Integer -> Integer, (Integer, Char))`

Thus, there are really infinitely many tuple types

An Example of the Use of Tuples

Use tuples with two floats to represent 2D-vectors

Define functions `vadd`, `vsub`, `vlen` to add, subtract, and compute the length of vectors:

```
vadd, vsub :: (Float, Float) -> (Float, Float) -> (Float, Float)
vlen :: (Float, Float) -> Float
```

(Solutions on next slide)

```
vadd (x1, y1) (x2, y2) = (x1+x2, y1+y2)
vsub (x1, y1) (x2, y2) = (x1-x2, y1-y2)
vlen (x, y) = sqrt (x**2 + y**2)
```

Note the pattern-matching to get the components of the argument tuples

Some More on Recursion and Modularity of Programs

A basic idea of functional programming is to define small but general functions, than can be used as building blocks in many applications

An example is the set of predefined list functions in Haskell

If you store your data with lists, then you can often use these functions to achieve what you want

An example: say we want to sum the lengths of a number of vectors

v_1, \dots, v_n

Assume the vectors are stored in a list

We'll give two solutions to this on the next two slides

Direct Solution

A direct solution:

```
sumvecs [] = 0
sumvecs (v:vs) = vlen v + sumvecs vs
```

Note the similarity with `sum`

Somehow we're duplicating work here

Let's try another solution where we use `sum`

A Solution that Reuses `sum`

Idea: first create list of vector lengths, then sum the elements in this list

```
sumvecs vs = let vlengths [] = []
              vlengths (v:vs) = vlen v : vlengths vs
              in sum (vlengths vs)
```

Note how we create the list of vector lengths by applying `vlen` to each element in the list of vectors

This is a common pattern! More on this when we talk about higher-order functions

Modules and Data Type Declarations

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

August 17, 2007

Modules and Data Type Declarations (revised 2007-08-17)

Modules and Data Types

Haskell code is packaged in *modules*

A module contains a number of declarations

The *scope* of the declarations is the module (normally not visible outside)

Module = software component containing functions, data types, ...

Good for packaging libraries to be reused in other Haskell programs

Declarations in a module are made visible in another module by an *import declaration*

```
import Shape
```

Typically one Haskell file ~ one module

Modules and Data Type Declarations (revised 2007-08-17)

1

Syntax of modules:

```
module XXX(...exported names...) where  
  
...declarations...
```

Modules and Data Type Declarations (revised 2007-08-17)

2

A Haskell compiler needs a special module `Main` with a function `main` to create an executable program:

```
module Main where  
  
...declarations...
```

```
main = ....
```

`main` has a special I/O-data type, more on this later

In Hugs you don't need the `Main` module, you just `:load` a module and then its exported declarations become visible

Modules and Data Type Declarations (revised 2007-08-17)

3

A Simple Module Example

A simple module `Vector` with our vector operations on tuples:

```
module Vector where

vadd,vsub :: (Float,Float) -> (Float,Float) -> (Float,Float)
vlen :: (Float,Float) -> Float

vadd (x1,y1) (x2,y2) = (x1+x2,y1+y2)
vsub (x1,y1) (x2,y2) = (x1-x2,y1-y2)
vlen (x,y) = sqrt (x**2 + y**2)
```

Note: no list of exported names \implies *all* declared names are exported. A version exporting only `vadd`, `vlen` would look like this:

```
module Vector(vadd,vlen) where
....
```

Another module can now import the `Vector` module and use the operations:

```
module Main where

import Vector

v1 = (1,3)
v2 = (3,2)

main = print (vadd v1 v2)
```

(The `print` function generates an *action* that prints a value to stdout (typically screen).)

Data Type Declarations

In Haskell you can define your own *data types*

A first, simple example:

```
data Color = Black | Blue | Green | Cyan | Red | Magenta
           | Yellow | White
```

Here, `Color` is a *type* (Just like `Bool`, `Integer`, `[Integer]`)

`Black`, `Blue` etc. are *constructors* (just like `True`, `17`, `[]`)

The elements of `Color` are the values `Black`, `Blue` etc.

We can write functions that use the `Color` values:

```
f :: Color -> Integer
f Black = 17
f Blue  = f Black + 2
...
```

Pattern-matching works as usual on user-defined constructors.

(User-defined types are no different from predefined types!)

Example: Geometrical Shapes

The previous example was quite limited

Haskell can do more than types with a small number of given elements

We can for instance define types whose elements are structured data (like tuples)

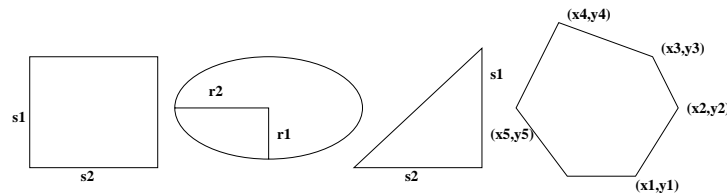
I'll show an example from the book on next page

Say we want to represent some kinds of *geometrical shapes*

(Later, we may want to do things with them like computing their areas, or displaying them graphically, or composing them into more complex shapes)

We want to represent *rectangles*, *ellipses*, *right triangles* (90 degree angle), and general *polygons*

Rectangles, ellipses, and right triangles are characterized by two numbers, and polygons by a number of 2D-coordinates:



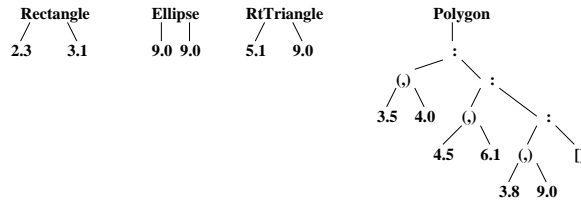
Here's the data type declaration:

```
data Shape = Rectangle Float Float
           | Ellipse Float Float
           | RtTriangle Float Float
           | Polygon [(Float,Float)]
  deriving Show
```

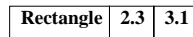
So, for instance, `Rectangle 2.3 3.1` represents a rectangle with sides of length 2.3 and 3.1, respectively

("deriving Show" gives a default way to print values of type `Shape`. More on this later)

The constructors `Rectangle` etc. take arguments and build data structures containing these arguments



You can also think of them as unique tags:



So `Rectangle 2.3 3.1` is basically the same as the tuple `(2.3, 3.1)` plus a tag telling that this tuple represents a rectangle

Type Synonyms

In Haskell, we can define *type synonyms*

A type synonym has the same information as the original type, but the type system differs between them

This is useful since sometimes one uses the same data type to represent different things

For instance, we use floating point numbers to represent both sides of rectangles and radii of ellipses

If we use type synonyms, then the type system can catch errors where we use values in the wrong way

Type synonym declarations for our geometrical shapes:

```
type Radius = Float
type Side   = Float
type Vertex = (Float,Float)
```

New definition of the `Shape` data type:

```
data Shape = Rectangle Side Side
           | Ellipse Radius Radius
           | RtTriangle Side Side
           | Polygon [Vertex]
  deriving Show
```

Type synonyms are used just as ordinary types:

```
circle :: Radius -> Shape
circle r = Ellipse r r

square :: Side -> Shape
square s = Rectangle s s
```

Declaring, say, `square :: Float -> Shape` would give a type error

Functions on Shapes

Let's define a function `area :: Shape -> Float` that computes the area of a shape

Solution on the next few slides ...

`area` can be defined case by case by pattern-matching on different constructors

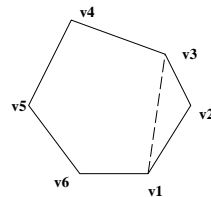
Easy cases first:

```
area (Rectangle s1 s2) = s1*s2
area (RtTriangle s1 s2) = s1*s2/2
area (Ellipse r1 r2)   = pi*r1*r2
```

(Assuming `pi` is defined in the module we're working in)

What about polygons?

Three corners or more: compute it by cutting a triangle, computing its area, and adding to area of rest of polygon (which is also a convex polygon)



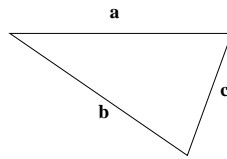
Recursive function, must terminate since one corner removed for each cut

Solution:

Assume for now a function `triArea` that compute the area of a triangle given its corners

```
area (Polygon (v1:v2:v3:vs))
  = (triArea v1 v2 v3) + area (Polygon v1:v3:vs)
area (Polygon _) = 0
```

triArea is computed with *Heron's formula*:



$$A = \sqrt{s(s-a)(s-b)(s-c)}, \text{ where } s = \frac{1}{2}(a+b+c)$$

(This is classical geometry)

We have the vertices but not the length of the sides between them

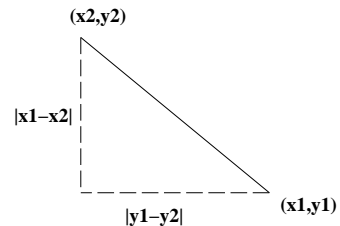
Assume for now a function `distBetween` that computes the distance between two vertices:

```
triArea      :: Vertex -> Vertex -> Vertex -> Float
triArea v1 v2 v3 = let a = distBetween v1 v2
                    b = distBetween v2 v3
                    c = distBetween v3 v1
                    s = 0.5*(a+b+c)
                    in sqrt(s*(s-a)*(s-b)*(s-c))
```

A Note on Programming Style

Finally,

```
distBetween      :: Vertex -> Vertex -> Float
distBetween (x1,y1) (x2,y2) = sqrt ((x1-x2)^2 + (y1-y2)^2)
```



In the polygon case, we used smaller functions (`triArea`, `distBetween`) to compute results needed to compute the whole area

This is a style of programming supported well by functional programming languages like Haskell: define (or use predefined) small, general functions to successively compose the desired solution

Chapter 3: IO Actions (Simple Graphics)

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

September 8, 2003

SoE Chapter 3: IO Actions (Simple Graphics)

A Haskell function can *compute* a sequence of actions

The `main` function is a sequence of actions. When computed the actions are *executed* in order

Actions may produce values, which can be used by the subsequent actions in scope (think of `getchar()` in C)

An action producing a value of type `a` has type `IO a`

However, *actions are not functions!* `IO a` is not the same type as `a`. Actions can only be put in sequence, not called as functions

Basic Input/Output in Haskell

We said functional programming is about calculating expressions

Gives a simple way of interacting: type an expression, obtain the calculated result

But often more complex ways of interaction are needed

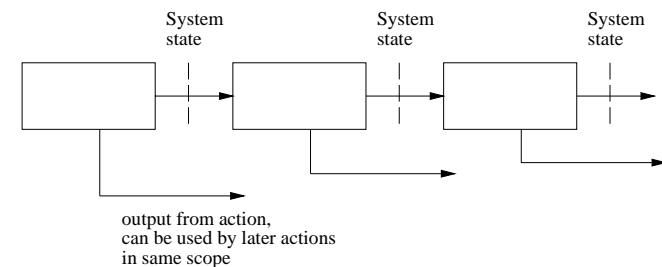
Haskell has a special kind of data called *actions*

Actions can interact with the environment (like performing I/O)

Actions can be sequenced, so they are performed in a certain order

SoE Chapter 3: IO Actions (Simple Graphics)

A Graphical View of Actions



Actions

Some functions returning actions:

```
putChar :: Char -> IO ()   writes character to standard output
getChar :: IO Char         reads character from standard input
putStr  :: [Char] -> IO ()  writes string to standard output
getline :: IO [Char]       reads a line from standard input
```

(Actions producing no useful value have type `IO ()`)

Strings are simply lists of characters in Haskell (`[Char]` is also called `String`)

Special syntax for strings: `"Hello" = ['H', 'e', 'l', 'l', 'o']`

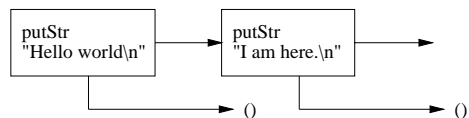
A simple Haskell program printing "Hello world":

```
module Main where
main = putStr "Hello world\n"
```

A Small Example

How to put actions in sequence:

```
do putStr "Hello world\n"
   putStr "I am here.\n"
```



Special keyword `do` denotes beginning of sequence

The layout rule decides when an action is considered to follow in sequence (don't put it to the left of previous action)

Let's write an action `echo` that reads a character and echoes it to the screen:

(See next slide for solution)

```
echo :: IO ()
```

```
echo = do c <- getChar  
        putChar c
```

`getChar` has type `IO Char`, thus “returns” a `Char`

In the code above, `c` is bound to the “returned” character

`echo` is executed when called in `main`:

```
module Main where  
main = echo
```

Composed Actions and what they “Return”

A sequence of actions is itself an action (a “composed” action)

A composed action “returns” the returned value (if any) of its *last* action

Example:

```
get2 = do getChar  
         getChar
```

action “returns” the last character read

Use as usual:

```
do c <- get2  
    putChar c
```

How to “Return” Special Values

Sometimes we would like to have more control over the “returned” values

Then use:

```
return :: a -> IO a
```

`return x` does nothing but “return” `x`

An example: an action that reads a character and “returns” the upper-case version (`toUpper :: Char -> Char` converts characters to upper case):

```
getUpper = do c <- getChar  
             return (toUpper c)
```

A Second Small Example

A composed action `echoLoop` that repeatedly reads characters, echoes them, and exits when a space is hit

(See next slide for solution)

We need some kind of “loop” to do this

Let's use recursion!

```
echoloop :: IO ()

echoloop = do c <- getChar
             if c == ' ' then return ()
                 else do putChar c
                         echoloop
```

File Handling Actions

```
writeFile :: FilePath -> String -> IO ()
```

FilePath synonym for String

```
writeFile "testFile.txt" "Hello File System"
```

```
readFile :: FilePath -> IO String
```

+ many other actions for file/system/user I/O, see Ch. 16 in the book

(Error handling: see Ch. 16 as well)

Two things to note:

- the use of recursion to get a “looping” effect
- returning the value `()`

`()` is the single value of the *unit* type `()` (same name for type and value!)

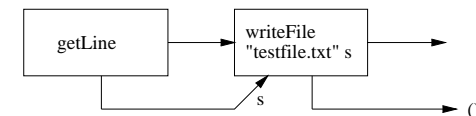
Used for values that are not important (like “return” values from actions that we don't care about)

Think of the value `()` as the empty tuple, and the type `()` as the empty tuple type

A Third Small Example

Read a string from standard input and write it to file `testFile.txt`:

```
do s <- getLine
   writeFile "testFile.txt" s
```



Actions are Ordinary Data

Actions are just like any other data, can for instance be stored in data structures and moved around

A list of actions:

```
actionList = [putStr "Hi\n",
              writeFile "file.txt" "Hello file",
              putStr "Ho Ho\n"]
```

However, must be put into a `do`-sequence in order to get executed

A Fourth Small Example

`PutChar :: Char -> IO ()` writes a single character

Define `putStr` in terms of `PutChar`

First a function converting a list of characters into a list of `PutChar` actions:

```
putCharList :: String -> [IO ()]
putCharList [] = []
putCharList (c:cs) = putChar c : putCharList cs
```

Then easy to define `putStr`:

```
putStr s = sequence_ (putCharList s)
```

A useful function that turns a list of actions into a `do`-sequence:

```
sequence_ :: [IO a] -> IO ()
```

```
main = sequence_ actionList
```

(`sequence_` is definable in Haskell itself. See the book, p. 259)

A good exercise is to define it. It is not hard!

Simple Graphics

The book defines a module `SimpleGraphics` for handling a graphics window

It contains a number of actions to open and close such windows, and to draw different kinds of graphics objects in it

It is used in the graphics library that is built up in the book

I will not cover it here, but you will use it in Lab 2. I recommend reading about it in Ch. 3.2

A String Programming Example

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

August 17, 2007

A String Programming Example (revised 2007-08-17)

Strings

Strings are simply lists of characters in Haskell

`[Char]` is also called `String`

Special syntax for strings: `"Hello" = ['H','e','l','l','o']`

This means all general list functions will work on strings

Makes it easy to write program for string processing since Haskell has a rich set of useful list functions

String Processing

String (or text) processing is important

Conversions between different formats: files, documents, XML, web/database, etc.

I think functional programming is good for this kind of application

We will look at a simple example here: how to break a text into a list of words, that can be used for various things like:

- counting the number of words in the text
- printing the text with a given maximal line length in characters (breaking lines when next word does not fit in)

A String Programming Example (revised 2007-08-17)

1

Breaking a String Into Words

Words are sequences of characters separated by one or more *whitespace* characters: space, newline, tab

(In Haskell: `' '`, `'\n'`, `'\t'`)

We want a function that converts a string into a list of its words:

```
string2words :: String -> [String]
```

For instance,

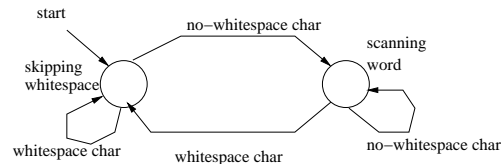
```
string2words "Allan tar      kanan i  
                28000 baud"
```

```
=>
```

```
["Allan", "tar", "kakan", "i", "28000", "baud"]
```

How code `string2words`?

We need a mental model. This is a simple parsing problem, which can be solved by a *finite automaton* with two states:



Common design pattern: one function per state. When new character read the function for the new state is called

A First Solution

Functions to count characters until next whitespace and next no-whitespace, respectively:

```
find_ws [] = 0
find_ws (c:cs) = if c == ' ' || c == '\n' || c == '\t'
                  then 0 else 1 + find_ws cs

find_nows [] = 0
find_nows (c:cs) = if c \= ' ' && c \= '\n' && c \= '\t'
                    then 0 else 1 + find_nows cs
```

We'll use a variation of this pattern: in each state we will *look ahead* and count the number of characters before changing to the other state:

- whitespace: count characters until non-whitespace char, then drop that number of characters and call the other function on rest of list
- word: count characters until whitespace char, then save that number of characters into string and call the other function on rest of list

We can then use the standard function `drop` to skip a number of characters:

```
drop 3 [1,4,2,5,6] ==> [5,6]
```

(`drop n s` returns the list remaining after `take n s`)

Functions `string2words` and `string2words1` corresponding to states “skipping whitespace” and “scanning word”, respectively:

```
string2words [] = []
string2words s = string2words1 (drop (find_nows s) s)

string2words1 [] = []
string2words1 s = let n = (find_ws s)
                  in take n s : string2words (drop n s)
```

Note how the words are collected into separate strings by `take`

Also note that “:” in `string2words1` puts the string as *element* into the list, so the returned list is a list of *strings* (not characters)

A More Elegant Slution

This solution works fine, but is a bit clumsy

In particular, `find_ws` and `find_nows` are very similar

They do precisely the same, but with negated conditions!

Can we “factor out” the common structure?

Yes, if we can make the condition a *parameter* to a more general function!

Let's see on next slide how to do this ...

A More General Character Count Function

Haskell has *higher order functions*

We can thus define a function `find` that takes a *predicate* `p` on characters as first arguments and counts the number of characters up to the first character `c` such that `p c == True`:

```
find :: (Char -> Bool) -> String -> Integer
```

```
find p [] = 0
```

```
find p (x:xs) = if p x then 0 else 1 + find p xs
```

Predicate to check for whitespace:

```
ws :: Char -> Bool
```

```
ws ' ' = True
```

```
ws '\n' = True
```

```
ws '\t' = True
```

```
ws _ = False
```

Then simply:

```
find_ws s = find ws s
```

For `find_nows`, we must have a negated whitespace-predicate:

```
not_ws c = not (ws c)
```

We get:

```
find_nows s = find not_ws s
```

Final Solution

```
module String2words where

ws ' ' = True
ws '\n' = True
ws '\t' = True
ws _ = False

find p [] = 0
find p (x:xs) = if p x then 0 else 1 + find p xs

find_ws s = find ws s

find_nows s = find (not_ws) s

string2words [] = []
string2words s = string2words1 (drop (find_nows s) s)
string2words1 [] = []
string2words1 s = let n = (find_ws s)
                  in take n s : string2words (drop n s)
```

A function `words2lines` `linelen ws`, where `linelen` is the line length and `ws` is a list of words to be printed

Idea: keep a current position on the line, check length of next word, if greater than `linelen` then start new line else output word on current line and update position

Current position passed as argument

Local function to do this, so `words2lines` does not need to have this extra argument

We will use the *append* operation “++” on lists:

`[1,2,3] ++ [4,2] ==> [1,2,3,4,2]`

Applications of `string2words`

Let's do the two applications mentioned before:

- counting the number of words in the text
- printing the text with a given maximal line length in characters (breaking lines when next word does not fit in)

The first you can do yourself in one line!

The second is more interesting ...

The Solution

```
words2lines linelen ws =
  let
    w2l [] pos = []
    w2l (w:ws) pos = if pos + length w < linelen
                     then w ++ [ ' ' ] ++ w2l ws (pos + length w + 1)
                     else '\n' : w ++ [ ' ' ] ++ w2l ws (length w + 1)
  in w2l ws 0
```

Not perfect. Leaves space at end of each line. Somewhat poor treatment of words longer than line length – always new line even if the long word is first in list

Exercise: write a new solution that handles these cases better

List Functions, Polymorphic Functions, and Higher-Order Functions

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

September 10, 2003

List Functions, Polymorphic Functions, and Higher-Order Functions

Append

(We've seen it in use before)

Here's the definition:

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Thus,

```
[1,2] ++ [3,4,5] = 1 : 2 : [] ++ 3 : 4 : 5 : []
=> 1 : (2 : [] ++ 3 : 4 : 5 : [])
=> 1 : 2 : ([] ++ 3 : 4 : 5 : [])
=> 1 : 2 : 3 : 4 : 5 : []
= [1,2,3,4,5]
```

Note that append takes time proportional to length of first argument

List Functions

We have seen some list functions already

There are some important ones left

We'll define *append* (++) and *zip* here

List Functions, Polymorphic Functions, and Higher-Order Functions

1

zip

zip takes two lists and returns a list of pairs of their respective elements (like closing a zipper):

```
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b):zip as bs
zip _ _ = []
```

Thus,

```
zip [1,2,3] ["allan","tar","kakan"] =>
  [(1,"allan"),(2,"tar"),(3,"kakan")]
```

So we can for instance use *zip* to put a number on each element in a list

Polymorphic types

Consider the following function (that computes the length of a list):

```
length [] = 0
length (x:xs) = 1 + length xs
```

What is the type of `length`?

It could be `[Integer] -> Integer`, or `[Char] -> Integer`, or even `[[Integer]] -> Integer`!

`length` should really work regardless of the type of the elements

It has type `[a] -> Integer`, where `a` is a *type variable*

This is a *polymorphic type*

`[a] -> Integer` is the *most general* type of `length`

Any other possible type for `length` can be obtained by replacing `a` with some other type

Haskell's type system gives the most general type, unless you give an explicit type declaration

Type inference is used to find this type

Some other polymorphic list functions (and lists):

```
head :: [a] -> a
tail :: [a] -> [a]
take :: Integer -> [a] -> [a]
drop :: Integer -> [a] -> [a]
(++): :: [a] -> [a] -> [a]
zip :: [a] -> [b] -> [(a,b)]
(:) :: a -> [a] -> [a]
[] :: [a]
```

Higher-Order Functions

Haskell is a *higher order* language

This means that functions are data just as data of any other “ordinary” type

They can be stored in data structures, passed as arguments, and returned as function values

Functions as arguments provides a way to *parameterize* function definitions, where common computational structure can be “factored out”

Functions that take functions as arguments are called *Higher-Order Functions*

Common computational patterns can be captured as higher order functions

We'll show some important examples here

A First Example: map

We have previously seen (in Chapter 3.1):

```
putCharList :: String -> [IO ()]
putCharList [] = []
putCharList (c:cs) = putChar c : putCharList cs
```

This is a common pattern: to apply a function to each element in a list

Computation pattern captured by a higher-order function `map`:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

`map` applies an *arbitrary* function `f` to the elements in a list

```
map :: (a -> b) -> [a] -> [b]
```

Note that the type of `map` is polymorphic, this is common for higher-order functions

We can now define

```
putCharList cs = map putChar cs
```

A Second Example: filter

`filter` removes all elements from a list that do not satisfy a given predicate:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x
                  then x : filter p xs
                  else filter p xs
```

For instance: if `even` returns `True` for exactly the even numbers, then

```
filter even [0,1,2,3,4,5] => [0,2,4]
```

Some Syntax: Guarded Definitions

This is really how `filter` is defined in Haskell:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

This definition uses *guards*: conditions that “filter out” different cases

They are tried in order (like pattern-matching)

`otherwise` is a “catchall”, often used at end

Guards are just syntactic sugar, can always be expressed with `if-then-else`

Folds

Rather than applying a function to each single member of a list, we might want to apply a function with two arguments successively to all elements

An instance of this is summing all numbers in a numeric list, recall `sum`:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

Applies `+` successively to all elements, “collecting” them into their sum

All these functions are instances of `fold`:

```
fold :: (a -> b -> b) -> b -> [a] -> b
fold op init [] = init
fold op init (x:xs) = x `op` (fold op init xs)
```

We can now define:

```
sum xs = fold (+) 0 xs
product xs = fold (*) 1 xs
and xs = fold (&&) True xs
```

Can you think of any other functions that can be defined with `fold`?

Now consider *multiplying* the numbers in a list:

```
product [] = 1
product (x:xs) = x * product xs
```

Or, ANDing together a list of booleans:

```
and [] = True
and (x:xs) = x && and xs
```

There’s something in common here!

Haskell actually defines *two* folds:

`foldr`: same as `fold`

`foldl`, defined as:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl op init [] = init
foldl op init (x:xs) = foldl op (init `op` x) xs
```

`foldr` = “fold from the right”

`foldl` = “fold from the left”

Note the accumulating argument for `foldl`, where the “sum” is collected

Why two Folds?

Why two folds? Sometimes, one can be more efficient than the other (see Ch. 5.4.2)

Also, they have slightly different types, there are cases where one will work but not the other

However, under some conditions they will compute the same answer (more on this in Ch. 11.3)

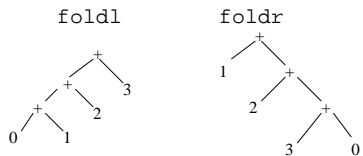
How do the Folds Work?

Let's compare the evaluation of `foldr (+) 0 [1,2,3]` and `foldl (+) 0 [1,2,3]`:

```
foldr (+) 0 [1,2,3] => 1 + foldr (+) 0 [2,3]
                    => 1 + (2 + foldr (+) 0 [3])
                    => 1 + (2 + (3 + foldr (+) 0 []))
                    => 1 + (2 + (3 + 0))
                    => 6
```

```
foldl (+) 0 [1,2,3] => foldl (+) (0 + 1) [2,3]
                    => foldl (+) ((0 + 1) + 2) [3]
                    => foldl (+) (((0 + 1) + 2) + 3) []
                    => (((0 + 1) + 2) + 3)
                    => 6
```

Note how `foldl` and `foldr` builds the expression tree in different ways:



Since `+` is associative these give the same result

If the operator is not associative, then `foldl` and `foldr` can yield different results

Lazy Evaluation, and More on List Notation

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

October 13, 2004

Lazy Evaluation (revised 2004-10-12)

Lazy evaluation can change the meaning of programs:

```
inf = inf
f x y = if x == 0 then 1 else y
```

In language with eager evaluation: $f\ 0\ inf = \perp$ (it does not terminate)

In Haskell:

```
f 0 inf => if 0 == 0 then 1 else inf
        => if True then 1 else inf
        => 1
```

Most languages have *eager* evaluation of function arguments (call by value)

That is: evaluate all arguments in full before a call is made

Haskell uses *lazy evaluation* (call by need)

This means: delay evaluation of anything until it is needed

(Never do today what you can postpone until tomorrow. . .)

Lazy Evaluation (revised 2004-10-12)

1

Lazy evaluation can save work, but is costly to implement (more complex evaluation mechanism)

Its great advantage is that it can be used with potentially infinite data structures

This enables a different style of programming, with cleaner control

Consider

```
ones = 1:ones
```

Try to print it!

Nonterminating, however its value is *not* \perp (no lack of information here)

Rather, infinite list of ones

However, if we only need a finite part of it, only that part is computed.

For instance, if we only need the first five elements:

```
take 5 ones = [1,1,1,1,1]
```

How Does it Work?

Consider again take 5 ones:

```
take 5 ones → take 5 (1:ones)
            → 1:(take 4 ones)
            → 1:(take 4 (1:ones))
            → 1:1:(take 3 ones)
            → 1:1:(take 3 (1:ones))
            → 1:1:1:(take 2 ones)
            → 1:1:1:(take 2 (1:ones))
            → 1:1:1:1:(take 1 ones)
            → 1:1:1:1:(take 1 (1:ones))
            → 1:1:1:1:1:(take 0 ones)
            → 1:1:1:1:1:[] = [1,1,1,1,1]
```

An Example: Line Numbering

Let's see how we can put line numbers on each line in a text

Assume the text is formatted as a list of lines (each line a string)

Design:

- for each line, make a pair of line number and line
- then convert into string (with number as text in front)

Some Help Functions

A way to compute the infinite list of positive integers:

```
posints = let posints1 n = n : posints1 (n+1)
          in posints1 1
```

A function that converts a pair (integer,string) into a string:

```
pair2string (i,s) = show i ++ ". " ++ s
```

(show converts values into printable strings)

The Solution

We use higher order functions to produce a short solution:

```
number_lines s = map pair2string (zip posints s)
```

Note that `zip` returns a list as long as its shortest argument. Thus, only the initial segment of `posints` with the same length as `s` is used

`zip` produces a list of pairs, then trivial to `map` the conversion function over its elements

Convenient List Notations

Haskell has a number of convenient list notations:

`[m,m+step..n]` is shorthand for `[m,m+step,m+2*step,...,n]`

E.g., `[1,3..7] = [1,3,5,7]`

If the step is one it can be omitted:

`[1..5] = [1,2,3,4,5]`

Omitted upper limit means infinite list:

`[1..] = [1,2,3,...]`

Thus, our line numbering function can be rewritten as

```
number_lines s = map pair2string (zip [1..] s)
```

List Comprehensions

Sometimes one wants to generate lists whose elements are functions of elements of other lists

List Comprehensions provide a convenient notation for this

Example:

```
[(x,y) | x <- [0,1,2], y <- ['a','b']] =>
  [(0,'a'),(0,'b'),(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Here, `x <- [0,1,2]` and `y <- ['a','b']` are called *generators*

They generate “indices” (`x`, `y` above), much like in a nested loop

It is also possible to have a “guard” – a condition filtering the list:

```
[(x,y) | x <- [0,1,2], y <- [1,2], x < y] => [(0,1),(0,2),(1,2)]
```

An Example

A classical sorting algorithm: *quicksort*

Idea:

- select an element (say, the first)
- move all smaller elements to the left
- move all greater (or equal) elements to the right
- apply quicksort recursively to the moved sequences

(Solution on next slide)

Quicksort in Haskell, with list comprehensions:

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++
               [x] ++
               qsort [y | y <- xs, y >= x]
```

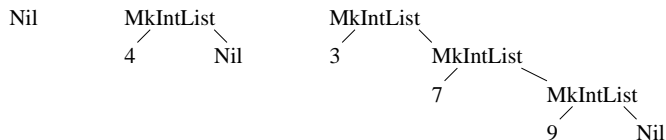
Chapter 7: Trees (and Recursive Data Types)

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

September 2, 2004

Some `IntList` examples:



Recursive Data Types

So far, we have defined data types with a number of cases, each of fixed size

How do we define data types for data like lists, which can have an arbitrary number of elements?

By making the data type definition *recursive*:

```
data IntList = Nil | MkIntList Int IntList
```

`IntList` can be either `Nil`, or a data structure that contains an `Int` and an `IntList`

Note similarity between data type declaration and context-free grammar

Polymorphism

Haskell's own data type for list is polymorphic

We can roll our own polymorphic list data type:

```
data List a = Nil | MkList a (List a)
```

This data type is precisely the same as Haskell's list data type, except that the constructor names are different!

Data type declarations can be recursive and polymorphic

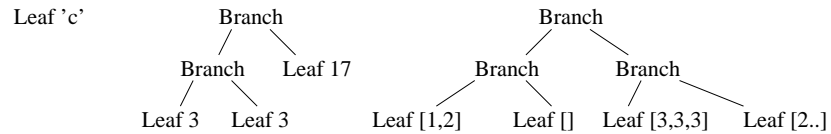
Haskell's built-in data types can in principle be declared in the language itself

Data Types for Trees

We can easily make our own data types for *trees*, like:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

A data type for trees with data stored in the leaves



Many other variations possible, see examples in the book

Let us use this type for now

Operations on Trees

Let us define some useful operations over our trees:

- a “map” for trees, that applies a function to all data stored in a tree,
- a function to put the elements in a tree into a list,
- a function to compute the *size* (number of leaves) of a tree, and
- a function to compute the *height* of a tree.

(Code on next two slides)

Map on trees:

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree (Leaf x) = Leaf (f x)
mapTree (Branch t1 t2) = Branch (mapTree f t1) (mapTree f t2)
```

(Hmmm, quite similar to `map` on lists, right? Is there some common underlying structure here? In the field of *generic programming*, such connections are investigated.)

To put the elements in a tree into a list:

```
fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch t1 t2) = fringe t1 ++ fringe t2
```

Size (number of leaves):

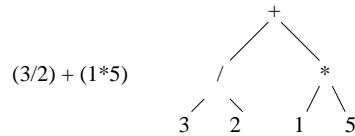
```
treeSize :: Tree a -> Integer
treeSize (Leaf x) = 1
treeSize (Branch t1 t2) = treeSize t1 + treeSize t2
```

Height:

```
treeHeight :: Tree a -> Integer
treeHeight (Leaf x) = 0
treeHeight (Branch t1 t2) = 1 + max (treeHeight t1) (treeHeight t2)
```

A Different Example: Arithmetic Expressions

Arithmetic expressions are really trees:

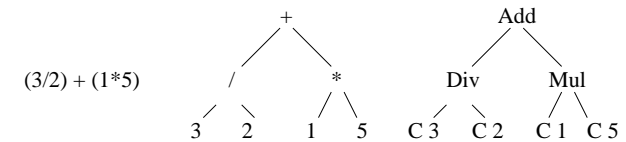


Let us define a data type for arithmetic (floating-point) expressions! We can then use it for various symbolic manipulations of such expressions

(Data type declaration on next slide)

```
data Expr = C Float | Add Expr Expr | Sub Expr Expr |
           Mul Expr Expr | Div Expr Expr
```

Each tree now represents an arithmetic expression:



An alternative data type declaration:

```
data Expr = C Float | Expr :+: Expr | Expr :- Expr |
           Expr :* Expr | Expr :/ Expr
```

Demonstrates *infix constructors* – such must begin with “:” (canonical example is cons)

So $(3/2) + (1 * 5)$ is represented by `(C 3 :/ C 2) :+: (C 1 :* C 5)`

Evaluating Expressions

One operation is to *evaluate* expressions

```
eval :: Expr -> Float
eval (C x) = x
eval (e1 :+: e2) = eval e1 + eval e2
eval (e1 :- e2) = eval e1 - eval e2
eval (e1 :* e2) = eval e1 * eval e2
eval (e1 :/ e2) = eval e1 / eval e2
```

```
eval ((C 17) :+: ((C 3) :- (C 1))) = 19.0
```

`eval` is a simple *interpreter* for our expression trees

Exercise (mini-project): extend `Expr` with *variables*. Then define a small *symbolic algebra package* for manipulating and simplifying expressions, for instance:

- evaluate constant subexpressions
- simplify as far as possible using algebraic identities
- symbolic derivation
- etc. . .

Chapter 9: More About Higher-Order Functions

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

August 17, 2007

SoE Chapter 9: More About Higher-Order Functions (revised 2007-08-17)

`simple x y z` means `((simple x) y) z` (function application is left associative)

`Integer -> Integer -> Integer -> Integer` means
`Integer -> (Integer -> (Integer -> Integer))`

Thus, `simple` is a function in one argument, returning a function of type

`Integer -> (Integer -> Integer)`

which returns a function of type

`Integer -> Integer`

which returns an `Integer`!

Encoding functions with several arguments like this is called `currying` (after Haskell B. Curry, early logician)

Currying (what Functions of Several Arguments Really are)

Remember `simple`?

A function of three variables, we said:

```
simple :: Integer -> Integer -> Integer -> Integer
simple x y z = x*(y + z)
```

But in Haskell, a function *only takes one argument!*

What's up?

SoE Chapter 9: More About Higher-Order Functions (revised 2007-08-17)

We *could* have defined:

```
simple :: (Integer,Integer,Integer) -> Integer
simple (x,y,z) = x*(y + z)
```

Another way to represent a function of three arguments, as a function taking a 3-tuple

But it is not the same function – it has different type!

This version may seem more natural, but the curried form has some advantages

Currying and Syntactical Brevity

What is `simple 5`?

A function in two variables (say x, y), that returns $5 * (x + y)$

We can use `simple 5` in *every place where a function of type*
`Integer -> (Integer -> Integer)` *can be used*

Direct Function Declarations

A declaration

`f x = g x`

where `g` does not contain `x`, can be written

`f = g`

“The function `f` equals the function `g`”, not stranger than “scalar”
declarations like `pi = 3.154159`

A First Example

Recall `sum` (and all the other functions defined by folds):

```
sum xs = foldl (+) 0 xs
```

Same as

```
sum xs = (foldl (+) 0) xs
```

Both `sum` and `foldl` have `xs` as last argument (and nowhere else)

It can then be “cancelled”:

```
sum = foldl (+) 0
```

A Second Example

A function that reverses a list

We first make a recursive definition, then redo it using higher order
functions, and finally we make it as terse as possible

(Recursive solution on next slide)

Recursive reverse

We use the “stack the books” principle:

```
reverse xs = revl [] xs
  where revl acc []      = acc
        revl acc (x:xs) = revl (x:acc) xs
```

(Try it on a few arguments to see how it works!)

Note `where`, an alternative to `let` when making local definitions. Some subtle differences but mostly interchangeable with `let`

Higher-Order reverse

The main operation of `reverse` is to put an element in a list, which is accumulated in an argument

Can we define a binary operation and use, say, `foldl` to define `reverse` (or `revl`)?

Let’s line up their definitions:

```
revl acc []      = acc
revl acc (x:xs) = revl (x:acc) xs
```

```
foldl op init []      = init
foldl op init (x:xs) = foldl op (init `op` x) xs
```

Hmmm, an operation `revOp` such that `acc `revOp` x = x:acc`?

Here’s the result:

```
reverse xs = foldl revOp [] xs
  where revOp acc x = x:acc
```

Can we proceed to break down the definition into smaller, more general building blocks?

Consider `revOp`. It is really just a “cons” (`:`), but with switched arguments

A general function that switches (or flips) arguments:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

(So `flip f` is a function that performs `f` but with flipped arguments)

Then

```
revOp acc x = flip (:) acc x
```

Nameless Functions

```
revOp acc x = flip (:) acc x
```

Can be simplified to

```
revOp = flip (:)
```

Finally, we obtain

```
reverse = foldl (flip (:)) []
```

Simple? Obfuscated? It's much a matter of training to appreciate this style

Functions don't have to be given names

We can write *nameless functions* through λ -abstraction:

$\lambda x \rightarrow e$ stands for function with formal argument x and function body e

(Comes from λ -calculus, where we write $\lambda x.e$)

$\lambda x \rightarrow x + 1$, an increment-by-one function

Can be used freely provided the type is OK

`map (\x -> x + 1) xs` returns list with all elements incremented by one

Syntactical conveniences:

$\lambda x y \rightarrow e$ shorthand for $\lambda x \rightarrow (\lambda y \rightarrow e)$

Pattern matching as in ordinary definitions, like $\lambda(x,y) \rightarrow x + y$

Currying can be defined through λ -abstraction:

```
simple 5 = \x y -> simple 5 x y
```

Also note:

```
f x = ....
```

is precisely the same as

```
f = \x -> (....)
```

Sections

Sections generalize curried syntax to binary operators

Really just syntactic sugar, but quite convenient. . .

Using `+` as example:

`(x+)` same as $\lambda y \rightarrow x+y$

`(+y)` same as $\lambda x \rightarrow x+y$

`(+)` same as $\lambda x y \rightarrow x+y$

Example:

```
posInts :: [Integer] -> [Bool]
posInts xs = map test xs where test x = x > 0
```

can be written

```
posInts xs = map (> 0) xs
```

or even, through “curry-cancelling”

```
posInts = map (> 0)
```

Very concise! Easy to understand? You judge.

Function Composition

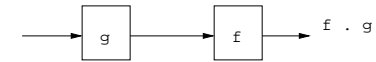
A well-known operation in mathematics, there defined thus:

$$(f \circ g)(x) = f(g(x)), \quad \text{for all } x$$

Haskell definition:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

Think of pipes in unix: unix command ~ function producing stream of characters, pipe between commands ~ function composition. Or functions as boxes:



A simple example:

Function converting string with newlines to list of lines (through Standard Prelude function `lines :: String -> [String]`), then computing the lengths of each line:

```
(map length) . lines
```

(A good exercise is to write `lines` yourself, and try to reuse as much as possible from `string2words`)

SoE Chapter 8, 10 (2, 3.2, 4): An Overview of the Graphics Library for Static Pictures (Shapes, Graphics, Regions)

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

August 17, 2007

SoE Chapter 8, 10 (2, 3.2, 4): An Overview of the Graphics Library for Static Pictures (Shapes, Graphics, Regions) (revised :

We have defined *shapes*, data representations for simple geometrical objects (no position information, no colour information)

You have also tried the *Graphics Library* for graphics windows actions

The book defines two more data types for graphics objects:

- *Regions*, adds information for positioning and scaling + set operations
- *Pictures*, adds colour information for regions + composition of several regions into 2D-scenes

Overview given here (including refresh of shapes and graphics windows)

SoE Chapter 8, 10 (2, 3.2, 4): An Overview of the Graphics Library for Static Pictures (Shapes, Graphics, Regions) (revised 2007-08-17) 1

Graphics Windows

Actions to handle a graphics windows (open, close etc.) and draw `Graphic` values in it

They all use a *window handle* of type `Window` to identify the window where the action should go

`Graphic` is a data type for low-level representations of graphics objects

Need not be concerned with the details of `Graphic` here

Will build on this by mapping the other data types into `Graphic`

SoE Chapter 8, 10 (2, 3.2, 4): An Overview of the Graphics Library for Static Pictures (Shapes, Graphics, Regions) (revised 2007-08-17) 2

A Graphics Windows Example

```
main0
= runGraphics (
  do w <- openWindow
      "My First Graphics Program" (300,300)
  drawInWindow w (text (100,200) "Hello Graphics World")
  k <- getKey w
  closeWindow w
)
```

(`text :: Point -> String -> Graphic` creates graphic value for `text`)

`getKey :: Window -> IO Char` reads keystrokes

SoE Chapter 8, 10 (2, 3.2, 4): An Overview of the Graphics Library for Static Pictures (Shapes, Graphics, Regions) (revised 2007-08-17) 3

Shapes

Recall the Shape data type:

```
type Radius = Float
type Side   = Float
type Vertex = (Float,Float)

data Shape = Rectangle Side Side
          | Ellipse Radius Radius
          | RtTriangle Side Side
          | Polygon [Vertex]
  deriving Show
```

We'll see how to use shapes as elements in regions and pictures

Regions

Regions extend shapes with:

- *set operations* (union, intersection, complement, empty set), and
- *scaling and translation*

Furthermore:

- translation function into `Graphics` (so regions can be drawn), and
- predicate checking if a coordinate belongs to a region (good for user interaction with regions)

Wrapped in module `Region`

The Region Data Type

```
data Region = Shape Shape -- primitive shape
           | Translate Vector Region -- translated region
           | Scale Vector Region -- scaled region
           | Complement Region -- inverse of region
           | Region 'Union' Region -- union of regions
           | Region 'Intersect' Region -- intersection of regions
           | Empty -- empty region
  deriving Show
```

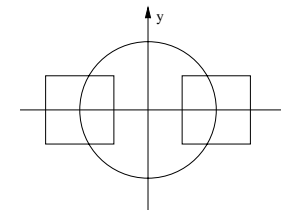
```
type Vector = (Float,Float)
```

Constructor and type names can be same

OK to use backquotes for constructors just as for functions

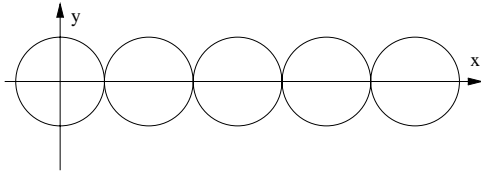
An Example

```
let c = Shape (Ellipse 0.5 0.5)
    s = Shape (Rectangle 1 1)
in (Scale (2,2) c) 'Union' (Translate (1,0) s)
   'Union' (Translate (-1,0) s)
```



Another Example

```
oneCircle = Shape (Ellipse 1 1)
manyCircles = [Translate (x,0) oneCircle | x <- [0,2..]]
fiveCircles = foldr Union Empty (take 5 manyCircles)
```



SoE Chapter 8, 10 (2, 3.2, 4): An Overview of the Graphics Library for Static Pictures (Shapes, Graphics, Regions) (revised 2007-08-17) 8

A set can be represented by its *characteristic function*: a predicate telling whether a point is in the set or not

We define translations from shapes and regions to characteristic functions:

```
containsS :: Shape -> Coordinate -> Bool
containsR :: Region -> Coordinate -> Bool
```

The characteristic function can give an implementation (two-color graphics):

- create array (matrix) of pixels
- test each pixel, set on/off depending on function
- Transfer array to graphics memory

In practice too heavy though, better to use other graphics interfaces

SoE Chapter 8, 10 (2, 3.2, 4): An Overview of the Graphics Library for Static Pictures (Shapes, Graphics, Regions) (revised 2007-08-17) 10

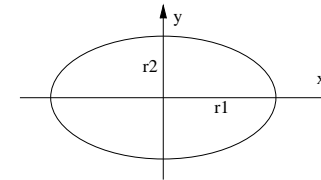
The Meaning of Shapes and Regions

What does a shape, or a region *mean*?

They represent sets of points in space

For an ellipse with radii r_1 and r_2 :

$$\left\{ (x, y) \mid \left(\frac{x}{r_1}\right)^2 + \left(\frac{y}{r_2}\right)^2 \leq 1 \right\}$$



SoE Chapter 8, 10 (2, 3.2, 4): An Overview of the Graphics Library for Static Pictures (Shapes, Graphics, Regions) (revised 2007-08-17) 9

More interesting to use `containsS` and `containsR` to test if coordinates belong to a shape (or region) or not

Can be used to test in which shape (region) a mouse click hits

This is useful for interactive applications

We will not define the characteristic functions here, see the book Ch. 8 for details

SoE Chapter 8, 10 (2, 3.2, 4): An Overview of the Graphics Library for Static Pictures (Shapes, Graphics, Regions) (revised 2007-08-17) 11

Pictures

Regions represent sets of points

Each Region, when given a colour, can be seen as a graphical object

Pictures are formed from different graphical objects

We'll check out the `Picture` module, the `Picture` data type, how to draw Pictures (briefly) and an interesting application (to move objects to front of picture by mouse clicks)

The Picture Module

```
module Picture (...lots of stuff...,
               module Region
               ) where
import Draw
import Region
import SOEGraphics hiding (Region)
import qualified SOEGraphics as G (Region)
```

What's this `hiding` and `qualified` stuff?

`SOEGraphics` also contains a data type `Region`

Name clash with our own `Region`

We can import all of `SOEGraphics` *except* `Region` using `hiding`

Still want access to the `Region` of `SOEGraphics`, but under different name

`import qualified` statement gives this under name `G.Region`

The Picture Data Type

```
data Picture = Region Color Region    -- Put colours on Regions
             | Picture 'Over' Picture  -- Must know which is over
             | EmptyPic                -- if overlap
             deriving Show             -- Empty picture
```

Color defined as before:

```
data Color = Black | Blue | Green | Cyan |
           Red | Magenta | Yellow | White
```

`Region Red (Shape (Ellipse 10 10))`: a red circle with radius 10

Drawing Pictures

First pictures, then regions (top-down approach)

Assume for now

```
drawRegionInWindow :: Window -> Color -> Region -> IO ()
```

It's defined in the book, we will not give the details here. Then

```
drawPic :: Window -> Picture -> IO ()
drawPic w (Region c r) = drawRegionInWindow w c r
drawPic w (p1 `Over` p2) = do drawPic w p2; drawPic w p1
drawPic w EmptyPic = return ()
```

Drawing Regions

How to draw regions (function `drawRegionInWindow`)?

We'll not give the full definition here (see the book Ch. 10), but rather a very brief overview

Simple shapes are easy, using predefined low-level primitives

Scaling, translation is easy (if the translated/scaled region is), but needs some consideration do obtain efficient solution

Union is easy (just draw the regions in any order)

But what about intersection? Complement?

A Simple Example

Intersection and complement are best computed pixel-by-pixel, with pixel sets represented by arrays

We could do it in Haskell, but would be inefficient

`SOEGraphics` has type `Region` for pixel arrays (using existing OS primitives and representations)

Will use this data type (renamed to `G.Region`)

`G.Region` provides operations to create graphics objects (rectangles, ellipses, polygons) and set operations on these (and, or, xor, set difference)

A function to draw a picture and close window when the space key is hit:

```
draw :: String -> Picture -> IO ()
draw s p = runGraphics (
    do w <- openWindow s (xWin,yWin)
       drawPic w p
       spaceClose w
    )
```

User Interaction

It is interesting not only to draw pictures, but also to manipulate them

This requires some kind of interaction:

- Capture mouse clicks and cursor positions
- Calculate which region is being marked
- Perform appropriate action on data structure
- Update screen accordingly

Move Regions to Top

A function for moving regions to the top by clicking on them

We can know the position of a mouse click (through an IO action returning the coordinates)

Idea:

1. Given a `Picture`, put its regions into a list sorted with the uppermost regions first
2. Go through the list to find the first region that is hit by the mouse click (`getLBP :: IO Coordinate`)
3. if there is such a region, then return the list where this region is moved to top, and redraw the image
4. otherwise do nothing

Creating a list of regions from a `Picture`:

```
picToList :: Picture -> [(Color,Region)]
picToList EmptyPic = []
picToList (Region c r) = [(c,r)]
picToList (p1 `Over` p2) = picToList p1 ++ picToList p2
```

Note similarity with `fringe`

How do we know that we get the list of regions in the right order?

Think about it!

A function `adjust` to move the “hit” region first

We must distinguish the case when a region is hit and when no region is

Use the `Maybe` data type for this (defined in `Standard Prelude`):

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

`adjust` uses standard prelude function

```
break :: (a -> Bool) -> [a] -> ([a],[a])
```

which splits a list in two, at the first element where the predicate becomes true

(If no such element, then first list = input list and second list = `[]`)

E.g. `break odd [2,4,3,4,5] = ([2,4],[3,4,5])`

(See Ch. 23 for a definition of `break`)

```
adjust :: [(Color,Region)] -> Coordinate
        -> (Maybe (Color, Region),[(Color,Region)])
adjust regs p =
  case (break (\(_,r)-> r `containsR` p) regs) of
    (top, hit:rest) -> (Just hit, top ++ rest)
    (_,[])          -> (Nothing, regs)
```

Note the case construct!

It gives the possibility to choose alternatives on pattern-matching anywhere in the code, not just on function arguments

A `loop` function that draws the list of regions, waits for mouse click, adjusts the list of regions, and repeats (if click not in any region, then exit loop):

```
loop :: Window -> [(Color, Region)] -> IO ()
loop w regs =
  do clearWindow w
     sequence_ [drawInWindow w c r | (c,r) <- reverse regs]
     (x,y) <- getLBP w
     case (adjust regs (pixelToInch (x - xWin2),
                               pixelToInch (yWin2 - y)) of
       (Nothing,_)      -> closeWindow w
       (Just hit, newRegs) -> loop w (hit : newRegs)
```

Note transformation of mouse coordinates from pixels to “Region” coordinates

(`clearWindow` does what the name suggests)

Finally, a function `draw2` that puts all together:

```
draw2 :: String -> Picture -> IO ()
draw2 s p =
  runGraphics (
    do w <- openWindow s (xWin, yWin)
       loop w (picToList p)
  )
```

Chapter 11: Proof by Induction

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

August 17, 2007

SoE Chapter 11: Proof by Induction (revised 2007-08-17)

Proofs by Induction for Properties of Natural Numbers

Goal: show that the property P is true for all natural numbers (whole numbers ≥ 0)

Proof by induction goes like this:

1. Show that P holds for 0
2. Show, for all natural numbers n , that if P holds for n then P holds also for $n + 1$
3. Conclude that P holds for all n

Formulated in formal logic:

$$[P(0) \wedge \forall n.P(n) \implies P(n+1)] \implies \forall n.P(n)$$

Induction

Have you ever performed proofs by induction? (You should have...)

Then you know induction proofs are to prove properties that hold for *all non-negative integers*

For instance, $\forall n. \sum_{i=1}^n i = n(n+1)/2$

Exercise: prove this property by induction!

SoE Chapter 11: Proof by Induction (revised 2007-08-17)

1

Why does Induction over the Natural Numbers Work?

The set of natural numbers \mathbf{N} is an *inductively defined set*

(A variation of) Peano's axiom:

- $0 \in \mathbf{N}$
- $\forall x.x \in \mathbf{N} \implies s(x) \in \mathbf{N}$
- $\forall x.0 \neq s(x)$
- $\forall x,y.x \neq y \implies s(x) \neq s(y)$

$s(x)$ "successor" to x , or $x + 1$

$$\begin{array}{ccccccc} 0 & \rightarrow & s(0) & \rightarrow & s(s(0)) & \rightarrow & s(s(s(0))) & \rightarrow & \dots \\ 0 & & 1 & & 2 & & 3 & & \dots \end{array}$$

Proofs by induction follow the structure of the inductively defined set!

The Inductively Defined Set of Lists

Inductively defined sets are typically sets of *infinitely* many *finite* objects

The set $[a]$ of (finite) lists with elements of type a :

- $[] \in [a]$
- $x \in a \wedge xs \in [a] \implies x:xs \in [a]$

Note similarity with the set of natural numbers!

An Induction Principle for Lists

Proof by induction *for finite lists* goes like this:

1. Show that P holds for $[]$
2. Show, for all finite lists $xs \in [a]$ and all possible list elements $x \in a$, that if P holds for xs then P holds also for $x:xs$
3. Conclude that P holds for all finite lists in $[a]$

Formulated in formal logic:

$$[P([]) \wedge \forall x \in a, xs \in [a].P(xs) \implies P(x:xs)] \implies \forall xs \in [a].P(xs)$$

Note: a proof by induction holds *only* for finite lists

Not for infinite lists, or the divergent list (\perp)

But very often this is good enough!

At least, it is better than not knowing anything . . . :-)

A Simple Example of Induction Over Lists

Prove that $\text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$ for all finite lists xs, ys

What induction hypothesis?

General rule: look at the function definitions and try to formulate the induction hypothesis so it matches the recursive structure!

A Number of Interesting Properties

Definitions of `length` and `++`:

```
length [] = 0
length (x:xs) = 1 + length xs
```

```
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Now formulate induction hypotheses and prove the result!

Can we extend the proof to infinite lists?

Some properties of `map`:

```
map id = id, where id = \x -> x
```

```
map (f . g) = map f . map g
```

```
map f . tail = tail . map f
```

```
map f . reverse = reverse . map f
```

```
map f (xs ++ ys) = map f xs ++ map f ys
```

(More properties in book, p. 138)

Another Proof by Induction

Let us prove $\text{map } (f \cdot g) = \text{map } f \cdot \text{map } g$!

(Proof on whiteboard)

A property of `fold`:

if op is associative and if e is left and right unit element for op , then, for all finite xs :

```
foldr op e xs = foldl op e xs
```

One can use properties of this kind to develop programs by *program transformations*

There is something called the *Bird-Meertens formalism*, which is a theory for functions over lists with many theorems like this

Let us prove a slightly simpler property:

That $\text{sum } xs = \text{sum1 } xs$ for all finite lists xS , where:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

```
sum1 xs = sum2 0 xs
```

```
sum2 a [] = a
sum2 a (x:xs) = sum2 (a+x) xs
```

Do you see how to generalize the proof to prove the property of `foldl` and `foldr` on the previous page?

Induction over Trees

Trees are also inductively defined, e.g., the tree data type in Ch. 7:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Corresponding, inductively defined set of finite trees:

- for any $x \in a$, $\text{Leaf } x \in \text{Tree } a$
- $t_1, t_2 \in \text{Tree } a \implies \text{Branch } t_1 t_2 \in \text{Tree } a$

Induction Principle for Trees

1. Show, for any $x \in a$, that P holds for $\text{Leaf } x$
2. Show, for any two finite trees $t_1, t_2 \in [a]$, that if P holds for t_1 and t_2 then P holds also for $\text{Branch } t_1 t_2$
3. Conclude that P holds for all finite trees in $\text{Tree } a$

A Proof by Induction over Trees

Show that `length (fringe t) = treeSize t` for all finite trees `t`, where

```
fringe (Leaf x) = [x]
fringe (Branch t1 t2) = fringe t1 ++ fringe t2

treeSize (Leaf x) = 1
treeSize (Branch t1 t2) = treeSize t1 + treeSize t2
```

Strictness

A function `f` is *strict* if `f ⊥ = ⊥`

Let

```
f x = 17
g x = x + 1
```

In Haskell, is `f` strict? `g`?

Theorem: *in a language with call-by-value, all user-defined functions are strict*

In a language with lazy evaluation, some user-defined functions can be non-strict

Strictness depends on whether the argument is needed or not

Example: consider definition of `&&` from Standard Prelude:

```
True && x = x
False && _ = False
```

The first argument *must* be evaluated to find out whether it is `True`, thus `&&` is strict in its first argument

But there are cases where the second argument is not needed, thus `&&` is not strict in its second argument

Some properties hold only for strict functions:

Theorem: *If `f` is strict, then*

```
f (if b then x else y) = if b then f x else f y
```

Can you prove the theorem?

A strict function in a lazy language can be evaluated with call-by-value!

This is interesting, since call-by-value often is more efficient than lazy evaluation

Strictness analysis is a program analysis that sometimes can detect if a function is strict

Good compilers for lazy languages have strictness analyzers

Is the following function strict or not?

```
f x = if x == 0 then 0 else x + f (x-1)
```

Chapter 12: Type Classes and Qualified Types

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

August 17, 2007

SoE Chapter 12: Type Classes and Qualified Types (revised 2007-08-17)

`Integer` is a member of `Num a` – an *instance*

So `(+) :: Integer -> Integer -> Integer` is still correct

But also

```
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
```

since `Int` and `Float` also are instances of `Num`

Type Classes

Haskell has more complex types than we have told you ...

`(+)` does not (only) have the type `Integer -> Integer -> Integer`

It has type `Num a => a -> a -> a`

Should be read “for all types `a` that are members of `Num`, `(+)` has type `a -> a -> a`”

`Num a` is a *type class*

Type class \sim set of types

SoE Chapter 12: Type Classes and Qualified Types (revised 2007-08-17)

1

`(Num a) => ...` is a *constraint* on the type variable `a`

Type constraints are propagated as part of the type inference

Example:

```
double x = x + x
```

Here, we know `(+) :: (Num a) => a -> a -> a`

Thus, we must have `x :: (Num a) => a` and
`x + x :: (Num a) => a`

It follows that `double :: (Num a) => a -> a` (since it takes `x` as argument and returns `x + x`)

Compare with object-oriented languages:

These have classes, with subclasses and inheritance

Objects of different classes can have methods with the same name, even if they do different things (since the objects are different)

For instance a `print` method can print different things depending on the object and its class

Thus, method names are *overloaded* – the same name stands for different things in different contexts

Same for operations like `+` in Haskell: means different things for `Integer`, `Int`, and `Float`

In Haskell, a class is a collection of types `+` a number of method names, where each type has an implementation of each function in the class

Declarations

Class declarations declare new classes

They declare the name of the class, possible subclass relations, and which methods the class has

Instance declarations make types members of a class

An instance declaration gives an *implementation* of each method for that particular type

Haskell does have conveniences for default implementations of methods, more on this later

The `Eq` class

`Eq`: class for all types where the elements can be compared for equality

Has two method names: `==` and `/=`

Any type in `Eq` must provide implementations of these

Which types are in `Eq`?

Almost all of Haskell's standard types!

Excluded are *function types*, *IO types*, and types containing such types (like `[a -> b]`)

(Can you find some good reason why these are excluded?)

So `Eq`-types are:

- “Basic” types (`Integer`, `Char`, `Bool`, `Float`, ...)
- Types that are formed from `Eq`-types (like `[Integer]`, `[[Integer]]`, `(Char, [Integer])`, ...)

Instance Declarations

Types are declared members of classes by *instance declarations*

Contains definitions for the *methods* in the class

Example 1: Eq-membership of Integer:

```
Instance Eq Integer where
  x == y = IntegerEq x y
```

(IntegerEq is a primitive operation comparing Integers)

No instance declaration for /=, let's get back to this later

Example 2: if a is a member of Eq , then $[a]$ is

Two lists are equal when:

- They are both the empty list, or
- their heads and tails are equal

In Haskell:

```
Instance Eq a => Eq [a] where
  [] == []      = True
  x:xs == y:ys = x == y && xs == ys
  _ == _       = False
```

Note constraint on a

Also note == being used on values of both types a and $\text{Eq } a$, not the same operation although same name!

Instance Declarations of User-defined Types

Exactly the same as for builtin types!

Example 3: Eq-membership of Tree a :

Similar to lists, trees are equal if:

- They are both leaves, and contain the same element, or
- They are both non-leaves, their left subtrees are equal, and their right subtrees are equal

In Haskell:

```
Instance Eq a => Eq (Tree a) where
  Leaf x == Leaf y      = x == y
  Branch l1 r1 == Branch l2 r2 = l1 == l2 && r1 == r2
  _ == _                = False
```

Class Declarations

Example: the class declaration for Eq :

```
Class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
```

Defines *class name*, *method names* with *type signatures*, and possible *default declarations* of methods

Default declarations are used if no explicit method definitions are given

For Eq , it thus suffices to define one of ==, /=

Inheritance

Haskell has *subclasses*, with *inheritance* of methods

A member of a subclass must also be a member of the superclass, and somehow define its methods as well

Example: Haskell has a predefined class `Ord` for comparison operations. This class is a subclass to `Eq`. Declaration:

```
Class Eq a => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

Note the constraint `Eq a => ...`, this gives the subclass relation

An instance declaration for `[a]`:

```
Instance Ord a => Ord [a] where
  [] < x:xs = True
  x:xs < y:ys = x < y || (x == y && xs < ys)
  _ < _ = False

... etc ...
```

This yields *lexicographic order* (“bokstavsordning”) on lists

Note subclass constraint necessary, since `==` is used

Quicksort

Recall quicksort:

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++
               [x] ++
               qsort [y | y <- xs, y >= x]
```

What type does it have?

Only constraint on list elements is that we must be able to compare them

Thus,

```
qsort :: (Ord a) => [a] -> [a]
```

This means it works on lists with elements of any “comparable” type!

A Selection of Haskell’s Standard Type Classes

The `Show` class

Class for types with printable values

These are essentially the same as `Eq`-types

Most important function:

```
show :: (Show a) => a -> String
```

E.g. `show [1,2,3] = "[1,2,3]"`

The actual methods in `Show` are more esoteric (for efficiency reasons), `show` defined in terms of these

Usually one does not have to define explicit instances of these methods, more on this later

Read

Class for types with “readable” values

“Inverse” to Show

Most important function:

```
read :: (Read a) => String -> a
```

E.g. `read "[1,2,3]" = [1,2,3]`

Often convenient to use for Haskell programs with simple user interactions

Num

Class for all numeric types (`Integer`, `Int`, `Float`, `Double`, ...)

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate      :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
```

Notably no division (there are subclasses to `Num` with division)

`fromInteger` is for overloading integer constants

```
fromInteger 4 :: Float = 4.0
```

“Coercion” and `fromInteger`

Some languages have *coercion* of numerical types:

For instance, if `x :: Float` and `y :: Int` then `x+y :: Float`, where a conversion from `Int` to `Float` is silently inserted for `y`

Haskell does not have this kind of coercion

The above would yield a type error

However, `fromInteger` provides a kind of “coercion” for integer constants into different numeric types

In Haskell, each integer constant `n` is parsed as `fromInteger n`

Thus `x + 17` means `x + fromInteger 17`, where `fromInteger 17` will assume the type of `x` in the type inference process

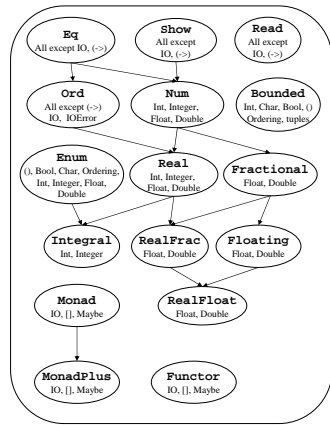
Exercise

Define the type `(Float,Float)` as an instance of `Num`!

(On the whiteboard only, I have no answer on next slide ...)

Can the instance declaration we will come up with be generalized to more general types of numerical pairs?

Haskell's Standard Class Hierarchy (selection)



Derived Instances

Haskell can derive *default implementations* for methods of the standard type classes Eq, Ord, Enum, Bounded, Ix, Read, Show

It then uses the “natural” way to define method instances of the type class from the structure of the values in the instance type

For instance, can automatically derive the implementation of methods in Eq for Tree a

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Eq
```

Often *very* convenient!

Uses underlying principles of generic programming

Chapter 13: A Module of Simple Animations

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

October 13, 2004

SoE Chapter 13: A Module of Simple Animations (revised 2004-10-12)

Thus, let us consider animations as *functions of time!*

The implementation can then sample the function with suitable intervals

For good reasons, we make the animation data type polymorphic:

```
type Animation a = Time -> a
type Time = Float
```

Can then also use the programming model for sound, control signals, ...

⇒ a general programming model for time-varying entities!

Critical though, how to link the model to the real world

Animations

An animation is a time-varying image

Implemented by a sequence of pictures (frames), that are shown by some time interval

How to program animations, and represent them?

We could use a list of `Picture` scenes, with some time interval:

```
type Animation = ([Picture], Float)
```

But this would tie us to a certain frequency. We would like to be more abstract, and leave the choice of frequency to the implementation

SoE Chapter 13: A Module of Simple Animations (revised 2004-10-12)

Some Simple Examples

A `Shape` for a “rubber ball” that pulsates, and a `Region` for a rotating ball:

```
rubberBall :: Animation Shape
rubberBall t = Ellipse (sin t) (cos t)

revolvingBall :: Animation Region
revolvingBall t = let ball = Shape (Ellipse 0.2 0.2)
                  in Translate (sin t, cos t) ball
```

A `Picture` of a yellow ball revolving around a red pulsating ball:

```
planets :: Animation Picture
planets t = let p1 = Region Red (Shape (rubberBall t))
              p2 = Region Yellow (revolvingBall t)
            in p1 'Over' p2
```

How to Animate (Real-world Connection)

A function

```
animate :: String -> Animation Graphic -> IO ()
```

that opens a window with a title and displays an animation in it

Note the type `Animation Graphic`: must use conversion functions to change animation type from `Picture`, `Region`, or `Shape`:

```
shapeToGraphic :: Shape -> Graphic
regionToGraphic :: Region -> Graphic
picToGraphic   :: Picture -> Graphic
```

Example: function `anim` to animate shapes

```
anim :: Animation Shape
shapeToGraphic . anim :: Animation Graphic
```

How `animate` Works

Programmed in Haskell, uses some nonstandard, OS-specific libraries to access graphics and timers

Basically a loop (a recursive action), which:

1. gets current time
2. draws animation (at current time) in window
3. waits for next clock tick (30 ms period)
4. loops to 1

Full definition at p. 169 in book

Lifting Picture Operations to Animations

We may want to “lift” pictures into constant animations, like the empty picture:

```
emptyA :: Animation Picture
emptyA t = EmptyPic
```

Or define a function that puts an animation on top of another:

```
overA :: Animation Picture -> Animation Picture -> Animation Picture
overA a1 a2 t = a1 t `Over` a2 t
```

But it is cumbersome to introduce new names all the time!

Would be nice to write `a1 `Over` a2` also when `a1, a2` are animations

How do this?

Type Classes for Lifting

Using the same name means overloading

Type classes can be used for this!

Two possible ways to use them:

- Make *pre-existing* operations work on animations: for instance, lift the numerical operations to “numerical animations” by making them instances of `Num`
- Lift new operations from static values to animations by defining a new type class with methods for them (for instance methods `empty`, `over` for pictures and picture animations)

Some technical issues:

`Animation a` is just type synonym for `Time -> a`: these cannot be made instances of a class! (For some technical reasons)

We can define a new but similar datatype instead (called `behavior` rather than `animation`, to stress generality):

```
data Behavior a = Beh (Time -> a)
```

But this gives unnecessarily costly implementation. For type with only one constructor one can use `newtype`:

```
newtype Behavior a = Beh (Time -> a)
```

Gives no overhead in representation! So go for this

Some useful *lifting operations*

To lift a *constant* into a behavior:

```
lift0 :: a -> Behavior a
lift0 x = Beh (\t -> x)
```

To lift a *unary function* into a unary function on behaviors:

```
lift1 :: (a -> b) -> (Behavior a -> Behavior b)
lift1 f (Beh a) = Beh (\t -> f (a t))
```

To lift a *binary function* into a binary function on behaviors:

```
lift2 :: (a -> b -> c) -> (Behavior a -> Behavior b -> Behavior c)
lift2 g (Beh a) (Beh b) = Beh (\t -> g (a t) (b t))
```

Etc.

A drawback of using `Animation a` as above rather than `Time -> a`:

Cannot use operations on functions, like function composition, directly

To compose two behaviours, one must first “pull out” the functions, then compose them, and finally build a new “box” for them:

```
compose (Beh a1) (Beh a2) = Beh (a1 . a2)
```

(Also possible to define a special infix operator for “behaviour composition”)

A function to animate behaviours:

```
animateB :: String -> Behavior Picture -> IO ()
animateB s (Beh pf) = animate s (picToGraphic . pf)
```

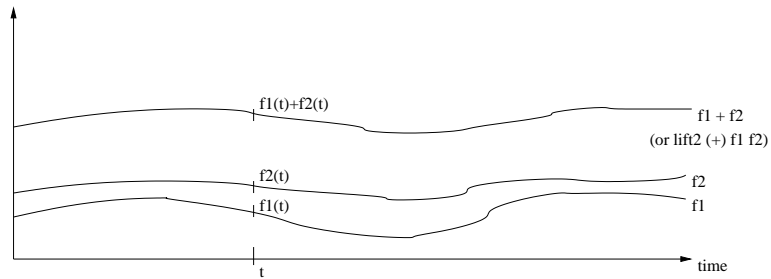
Some examples:

```
emptyB :: Behavior Picture
emptyB = lift0 EmptyPic -- (=> Beh (\t -> EmptyPic))

paintB :: Color -> Behavior Region -> Behavior Picture
PaintB c = lift1 (\r -> Region c r)
-- PaintB c (Beh anim) => Beh (\t -> (\r -> Region c r)(anim t))
--                               => Beh (\t -> Region c (anim t))

addB :: (Num a) => Behavior a -> Behavior a -> Behavior a
addB = lift2 (+)
-- addB (Beh b1) (Beh b2) => Beh (\t -> (+) (b1 t) (b2 t))
--                               => Beh (\t -> (b1 t) + (b2 t))
```

An Illustration of Lifting



Lifting Numerical Operations to Behaviors

Make `Behavior a` an instance of `Num` when `a` is!

Then we can indeed write `f1 + f2` rather than `lift2 (+) f1 f2`

How to do it? Here's the class declaration for `Num`

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate      :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
```

(Answer on next slide)

```
instance Num a => Num (Behavior a) where
  (+) = lift2 (+)
  (*) = lift2 (*)
  negate = lift1 negate
  abs = lift1 abs
  signum = lift1 signum
  fromInteger = lift0 . fromInteger
```

(Hmmm, these lifting functions were not a bad idea)

Lifting of some other numerical subclasses given in book, p. 174

For instance, all typical floating-point functions (trigonometric etc.)

However, we're not quite done yet ...

There is a subclass constraint on `Num`:

```
class (Eq a, Show a) => Num a where ...
```

Must make `Behavior a` an instance of `Eq a` and `Show a` first

No natural way to compare and show functions, so define methods as errors:

```
instance Eq (Behavior a) where
  a1 == a2 = error "Can't compare behaviors."

instance Show (Behavior a) where
  showsPrec n a1 = error "<< Behavior >>"
```

A Useful Behavior

Time itself can be seen as a behavior:

```
time :: Behavior Time
time = Beh (\t -> t)
```

A function that always returns the current time at the current time ...

What is `time + 5`?

```
time + 5 ==>
(lift2 (+)) (Beh (\t -> t)) (Beh (\t -> 5)) ==>
Beh (\t -> (\t -> t) t + (\t -> 5) t ) ==>
Beh (\t -> t + 5)
```

Aha. A function that always adds five to current time

New Type Classes for Behaviors

Let's define a class for `empty` and `over`, as suggested earlier:

```
class Combine a where
  empty :: a
  over  :: a -> a -> a
```

Now, this is quite general: intended for situations where `over` is associative and `empty` is unit element of `over` (that is, `empty `over` x = x `over` empty = x`)

Such structures are called *monoids* in algebra

Some possible (unexpected) instances:

```
instance Combine [a] where
  empty = []
  over  = (++)
```

```
instance Combine Integer where
  empty = 0
  over  = (+)
```

```
instance Combine a -> a where -- not valid Haskell 98,
  empty = \x -> x             -- but accepted by some implementations
  over  = (.)
```

Etc.

These are our intended instances:

```
instance Combine Picture where
  empty = EmptyPic
  over  = Over
```

```
instance Combine a => Combine (Behavior a) where
  empty = lift0 empty
  over  = lift2 over
```

A function to put a list of “combinable” values over each other:

```
overMany :: Combine a => [a] -> a
overMany = foldr over empty
```

So `overMany` works both on pictures and picture animations

Also, `sumInt = overMany :: [Integer] -> Integer`, etc.

More lifting

Some more examples of lifted operations and values:

```
reg      = lift2 Region
shape   = lift1 Shape
ell     = lift2 Ellipse
red     = lift0 Red
yellow  = lift0 Yellow
translate (Beh a1, Beh a2) (Beh r)
        = Beh (\t -> Translate (a1 t, a2 t) (r t))
```

(Could overload them by a new class, but maybe overkill)

Lifting of Predicates and Conditionals

This will prove quite useful. However,

We cannot overload lifted relational operations like numerical operations, since relational operations have types of form `a -> a -> Bool`

Thus, the result of comparing two behaviors would have to be a boolean value, not a boolean behavior

But we can introduce reasonably understandable names for the lifted operations:

```
(>*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool
(>*) = lift2 (>)
```

```
cond :: Behavior Bool -> Behavior a -> Behavior a -> Behavior a
cond = lift3 (\p c a -> if p then c else a)
```

The red revolving ball can now be written:

```
revolvingBallB :: Behavior Picture
revolvingBallB
  = let ball = shape (ell 0.2 0.2)
      in reg red (translate (sin time, cos time) ball)
```

Supports style of defining functions without mentioning any arguments

(Note: `sin`, `cos` are lifted floating-point operations, just as the operations in `Num`)

We can now define a “flashing”, time-varying color:

```
flash :: Behavior Color
flash = cond (sin time >* 0) red yellow
```

Will flash between red and yellow depending on the condition

Equivalently, for you who still have a hard time with this lifting stuff:

```
flash = Beh (\t -> if sin t > 0 then Red else Yellow)
```

Time Transformations

It can be interesting to *change the speed* of an animation, or part of it

We can do this by *time transformations*:

```
timeTrans :: Behavior Time -> Behavior a -> Behavior a
timeTrans (Beh f) (Beh a) = Beh (a . f)
```

Some examples:

```
timeTrans (2*time) anim -- speeding up an animation with factor 2
```

```
timeTrans (5+time) anim `over` anim
-- overlaying an animation with a copy delayed 5 time units
```

```
timeTrans (negate time) anim -- reversing an animation in time
```

More interesting examples in book

A Final Example

Firing a cannon ball with some initial velocity

This is a physical simulation problem

Movements of rigid bodies is determined by Newton's second law $F = ma$

This is a differential equation, acceleration a is second derivative of position w.r.t. time

To animate, we need to have the position as function of time

Thus, we need to solve the differential equation

Assume 2D-space with normal Earth gravity (9.81 m/s^2) in the negative y -direction

Assume no other forces affect the ball

Gives constant force $F = (0, -mg)$ where m is mass of ball

We want position $p = (p_x, p_y)$ as function of time

Acceleration $a = (a_x, a_y) = \left(\frac{d^2 p_x}{dt^2}, \frac{d^2 p_y}{dt^2}\right)$

$$F = ma$$

$$(0, -mg) = m(a_x, a_y)$$

$$(0, -g) = (a_x, a_y)$$

$$(0, -g)t + (v_{x0}, v_{y0}) = (v_x, v_y)$$

$$(0, -g)t^2/2 + (v_{x0}, v_{y0})t + (p_{x0}, p_{y0}) = (p_x, p_y)$$

Code

```
module Cannonball where

import Animation

import SOEGraphics

g :: Float
g = 9.81 -- acceleration caused by gravity on earth surface

pos :: (Float,Float) -> (Float,Float) ->
      (Behavior Float, Behavior Float)
pos (px,py) (vx,vy) = (Beh (\t -> vx*t + px),
                      Beh (\t -> -g*t^2/2 + vy*t + py))
```

```
circ r = ell r r

cannonball p0 v0 = reg (lift0 White)
                    (translate (pos p0 v0) (shape (circ 0.2)))

fire vel = animateB "Cannon shot" (cannonball (-1,-1) vel)

slomo = timeTrans (time/10)

slowfire vel = animateB "Slow cannon shot"
                  (slomo (cannonball (-1,-1) vel))
```

Note separation of aspects: colour, position, and shape

Chapter 14: Programming with Streams

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

October 13, 2004

Streams in Haskell

Streams can be modeled by *infinite lists* in a lazy language

Could define a data type for infinite lists in Haskell:

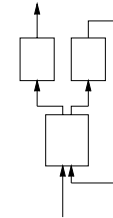
```
data Stream a = a :^ Stream a
```

However more convenient to use Haskell's builtin list data type (and ignore the finite lists)

Streams: a Common Model

Streams are infinite sequences of data

Common in hardware descriptions on architectural level: boxes connected with links, and a stream of data associated with each link



Also in operating systems (process communication, unix pipes)

Server-client communication

Streams to Save Memory

We can use streams to save work (and memory)!

Works for functions whose complexity can be decreased by saving the results of some previous function calls (*memoizing*)

Seems counterintuitive – how can an infinite list do that?

Sharing of results saves work

List elements without references become garbage

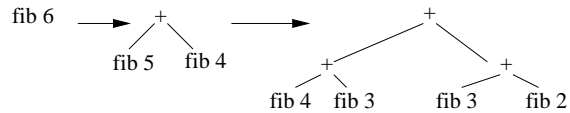
Thus, a function using a stream can sometimes run in constant space even if the stream is infinite

Example: the Fibonacci Function

Direct recursion for the Fibonacci Function (mathematical definition):

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Call tree for fib 6:



Exponential number of calls!

Anyone knows that one can implement fib with a loop, in linear time and constant space

Problem with recursive fib is lack of sharing: since $n - 2 = (n - 1) - 1$ one can share fib (n-2) and fib ((n-1)-1), but a compiler cannot reasonably understand that

But with streams we can introduce the sharing explicitly

A Fibonacci Stream

Idea:

```
1 1 2 3 5 8 13 21 34 55, ...
      =
1 1
  1 1 2 3 5 8 13 21 34 55, ...
    +
    1 2 3 5 8 13 21 34 55, ...
```

Haskell code:

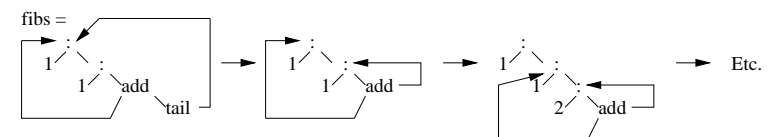
```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Let us simulate some steps. Define add = zipWith (+), or, essentially:

```
add (x:xs) (y:ys) = (x + y) : add xs ys
```

```
fibs
==> 1 : 1 : add fibs (tail fibs)
==> 1 : 1 : add (1 : 1 : add fibs (tail fibs))
                (1 : add fibs (tail fibs))
==> 1 : 1 : 2 : add (1 : add fibs (tail fibs))
                  (add fibs (tail fibs))
...etc...
```

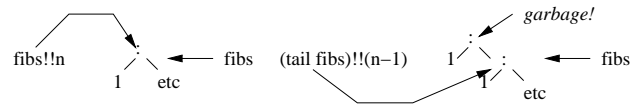
Graphically:



So exponential blowup of work can be avoided by storing previous Fibonacci values in list

But to compute `fib n` we need a list of length `n`, right? Well ...

```
fib n = fibs !! n
(x:xs)!!0 = x
(x:xs)!!n = xs!!(n-1) -- (skipped some conditions and cases...)
```



Producer-consumer situation, where only constant part of list needs to be kept in memory \implies constant space!

Client-Server Pattern

Servers and clients typically communicate through streams

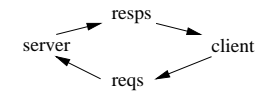
Can model server – single client communication with a simple list

(Modelling a server with several clients requires time-stamped streams as in Ch. 15)

Mutually recursive stream equation:

```
client :: [Response] -> [Request]
server :: [Request] -> [Response]
```

```
reqs = client resps
resps = server reqs
```



Example:

A server that returns value increased by 1

A client that initially sends 1, then echoes what comes back from server

```
client ys = 1 : ys
server xs = map (+1) xs
```

You may verify that `take 10 reqs = [1,2,3,4,5,6,7,8,9,10]`

This works since `xs` matches any argument. Thus `client ys \rightarrow 1 : ys` directly, without having to evaluate `ys` at all. This gets the recursion going.

Lazy Pattern Matching

Let us modify the client to test the first answer from the server:

```
client (y:ys) = 1 : if ok y then (y:ys)
                  else error "faulty server"
server xs = map (+1) xs
```

```
reqs = client resps
resps = server reqs
```

This will lead to nontermination, `reqs = resps = \perp !`

Reason: pattern `(y:ys)` forces evaluation of argument to check whether it matches, before "1" can be output. This leads to infinite recursion where nothing ever can be evaluated to match the pattern

Example: Modelling a Hardware Structure for Sorting

Two ways to circumvent the problem.

1: skip pattern-matching, using head:

```
client ys = 1 : if ok (head ys) then ys
           else error "faulty server"
```

2: *Lazy pattern-matching*:

```
client ~(y:ys) = 1 : if ok y then y:ys
           else error "faulty server"
```

The tilde defers the pattern-matching until the head or tail is actually needed

A mathematical model:

Cells numbered $0, \dots, n - 1$

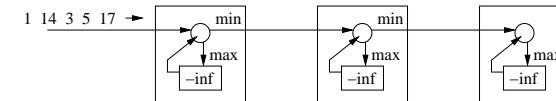
For input stream s_0 of length k , a system of *recursive equations* relating a number of *finite streams* of length k :

$\{s_{i,j}\}_{j=0}^{k-1}$: stream of input values (indexed $0, 1, \dots, k - 1$) to cell i
 ($\{s_{0,j}\}_{j=0}^{k-1} = s_0$)

$\{r_{i,j}\}_{j=0}^{k-1}$: stream of stored values in register of cell i

These streams are *indexed*: each element is numbered

A simple hardware structure to sort a sequence and store the n largest elements:



Operates in a synchronous, parallel fashion: every clock tick a new value is shifted in from left and compared with the value stored in the register

(This is a *systolic array*: a simple, efficient, special-purpose system-on-chip structure)

Equations:

$$r_{i,0} = -\infty, i = 0, \dots, n - 1$$

$$r_{i,j} = \max(r_{i,j-1}, s_{i,j-1}), i = 0, \dots, n - 1, j = 1, \dots, k$$

$$s_{i,j} = \min(r_{i-1,j}, s_{i-1,j}), i = 1, \dots, n - 1, j = 0, \dots, k - 1$$

Result: $\{r_{i,k}\}_{i=0}^{n-1}$ (contents of registers after k steps)

Haskell Model

Represent each stream by a list

We'll use two lists of lists: r for registers, and s for streams between registers

A complication: must represent $-\infty$

A solution: use type `Maybe a` rather than `a`, let `Nothing` represent $-\infty$

Can do this by making `Maybe a` an instance of class `Ord` whenever `a` is:

```
instance Ord a => Ord (Maybe a) where
  Nothing < Just _ = True
  Just x < Just y = x < y
  _ < _           = False
```

This is already done in Haskell! No need to declare yourself

Code:

```
ssort s0 n
= let k = length s0
    r = [Nothing : zipWith max (s!!i) (r!!i) | i <- [0..n-1]]
    s = map Just s0 :
        [zipWith min (s!!(i-1)) (r!!(i-1)) | i <- [1..n-1]]
  in map (!! k) r
```

Note indexing “!!” of lists, e.g. `[1,2,3]!!1 = 2`

(Indexing starts from 0)

Chapter 15: A Module of Reactive Animations

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

October 2, 2003

SoE Chapter 15: A Module of Reactive Animations (revised 2003-10-02)

What are Reactive Animations?

Animations are functions `Time -> a`

We can model the input (keystrokes, mouse clicks, ...) as a *stream of time-stamped atomic events*: `[(UserAction,Time)]`

Thus, reactive animations can be seen as be functions from the input stream to functions of time: `[(UserAction,Time)] -> Time -> a`

Assume the program executes in an environment with a *single, global stream* of input atomic events

Function `reactimate` executes reactive animations in this environment (implementation described in Ch. 17, not covered in course)

From Animations to Reactive Animations

We have seen animations as functions of time

But these animations were not interactive

Now, let's move to *reactive* animations, that can change according to input!

We then actually get a *general* high-level model for reactive real-time systems!

(Could be anything but animations: control systems, mobile phones, user interfaces, or whatever that can respond to inputs and depend on time)

As an example, we will build a little game

SoE Chapter 15: A Module of Reactive Animations (revised 2003-10-02)

Representation of Reactive Animations

We call them behaviors (just like animations)

The implementation does not use behaviors of type `[(UserAction,Time)] -> Time -> a`, however

Reason: would be very inefficient (see discussion in Ch. 15.2)

Instead,

```
newtype Behavior a = Behavior (([Maybe UserAction],[Time]) -> [a])
```

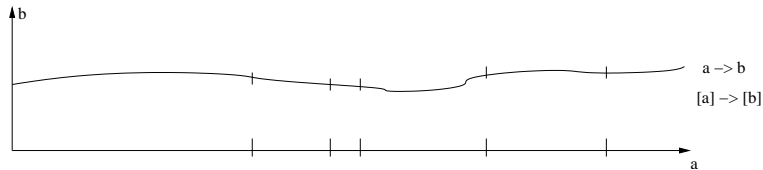
We can still think of behaviors as functions from atomic event streams and continuous time (will only use representation-independent primitives)

But it is good to understand the conversion (easier to read the book, for instance)

Understanding the Representation of Behaviors

Animations are not continuous in reality, but created from “samples” of the animated entity at certain times

The actual representation $([Maybe\ UserAction],[Time]) \rightarrow [a]$ depends on a conversion from continuous time to sampled time:



So a function $a \rightarrow b$ yields a sampled function $[a] \rightarrow [b]$ (list of inputs to list of outputs)

Behaviors are however also functions from streams of atomic events

The representation comes from merging this input stream with the stream of times at which we sample the animation

This means we must create a pair $(event, time)$ for each sample time, where *event* is an “empty” event (absence of real event)

Can use *Maybe* type and represent empty events by *Nothing*

See figure on next slide

Events

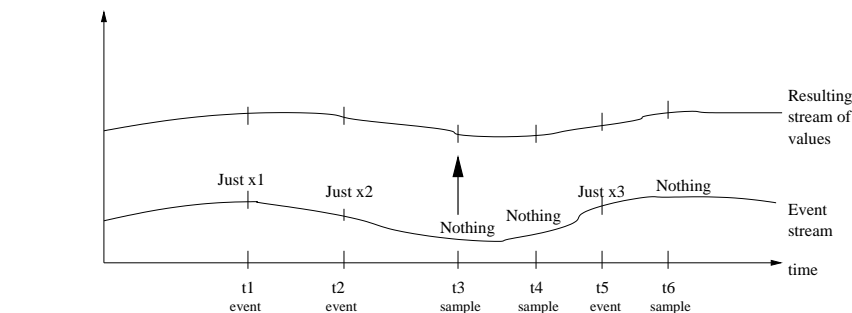
The reactive model also contains *event streams*

These are streams of atomic events that occur depending on input events and time

Do not mix them up with the atomic events!

We can think of them as functions from input events and time to time-stamped atomic events

```
type Event a = [(UserAction,Time)] -> Time -> [(a,Time)]
```



A representation $([Maybe\ UserAction,Time]) \rightarrow [a]$, can be “unzipped” to $([Maybe\ UserAction],[Time]) \rightarrow [a]$

View: global input stream of time-stamped events, some are “real” external events, some are events sampling the animation function

As for behaviors we convert to sampled time. The event stream is supposed to instantly produce an output event for every input event

`(Just x,t)` (external event) produces output of the form `(Just y,t)`

`(Nothing,t)` (sample event) produces output `(Nothing,t)`

So event streams are represented as

```
newtype Event a = ([Maybe UserAction],[Time]) -> [Maybe a]
```

which is really the same as `Behavior (Maybe a)`, only the name differs

Some Basic Behaviors

A number of primitives are the same as for the simpler animations in Ch. 13:

Time:

```
time :: Behavior Time
```

Behavior that yields current time, regardless of user input

Standard Lifting Functions

Lifting functions as for animations:

```
lift0 :: a -> Behavior a
lift1 :: (a -> b) -> (Behavior a -> Behavior b)
lift2 :: (a -> b -> c) -> (Behavior a -> Behavior b -> Behavior c)
```

Used to make `Behavior a` a member of numerical classes whenever `a` is, just as for animation

Also used for some standard liftings, see next page

Some Standard Liftings

```
paint = lift2 Region -- takes colour and region as args
```

```
red = lift0 Red -- similar for the other colours
```

```
shape = lift1 Shape
```

```
ell x y = shape (lift2 Ellipse x y)
```

```
rec x y = shape (lift2 Rectangle x y)
```

```
translate :: (Behavior Float, Behavior Float)
           -> Behavior Region -> Behavior Region
```

```
translate = ...see book, it does do what you'd expect...
```

```
over = lift2 Over
```

```
(<*) = lift2 (<) -- (also lifting >, &&, || to >*, &&*, ||*)
```

A Simple Example

A ball that circles the origin with distance 1, being painted with some (possibly time- and user-input-varying) color behavior `color1`:

```
ball1 :: Behavior Picture
ball1 = paint color1 circ
```

```
circ :: Behavior Region
circ = translate (cos time, sin time) (e11 0.2 0.2)
```

Just as in Ch. 13 but with new data types (this animation is not interactive, unless `color1` is)

Note that numerical constants `0.2` are lifted to constant behaviors

Reactive Behaviors

Behaviors that behave initially in some way, and then change to some other behavior when a certain event occurs

The new behavior is *carried in the event*: typical functional programming style!

```
untilB, switch :: Behavior a -> Event (Behavior a) -> Behavior a
```

`f `untilB` e`: Behave as `f` until time of first event in `e`, then change to behavior carried by that event

`switch` is a variation of `untilB` that loops, and on each new event switches to the event-carried behavior

Predefined Events

Different event streams can be defined by filtering out certain kinds of events from the global event stream

For instance, interesting to filter out different *button presses*, and *keystrokes*

Such predefined event streams in the library:

```
lbp :: Event ()
```

Returns an event containing `()` every time the left mouse button is pressed

```
key :: Event Char
```

Returns an event containing a character every time the corresponding key is pressed

How to Create Events Carrying Behaviors

Two (infix) operators:

```
(->>) :: Event a -> b -> Event b
```

`evs ->> x`: replace contents of every event in `evs` with `x`

Example: `lbp ->> 17` yields event containing 17 for each left button press

```
(=>>) :: Event a -> (a -> b) -> Event b
```

`evs =>> f`: transform contents of every event in `evs` with `f`

Example: `key =>> (\c -> toUpper c)` yields an event containing the upper-case character for each keystroke

Note that `evs ->> x = evs =>> (_ -> x)`

Some Simple Examples

A color behavior that starts out as red, and changes to blue when the left button is pressed:

```
color1 :: Behavior Color
color1 = red `untilB` lbp ->> blue)
```

A recursive color behavior that changes between red and blue when the left button is pressed:

```
color1r = red `untilB` lbp ->>
         blue `untilB` lbp ->>
         color1r
```

Predicate Events

```
when :: Behavior Bool -> Event ()
```

when b: a ()-event is inserted every time b changes from False to True

So it “triggers” an event every time a condition changes to true

Example:

```
when (time >* 2 &&* time <* 5) ||* (time >* 10)
```

Can be used for instance in animations to detect collisions between moving objects, or to define time-triggered behaviors

A color behavior that starts out as white, changes to red/blue/yellow when key R/B/Y is pressed, and changes back to white when any other key is pressed:

```
color6 = white `switch` (key ==> \c ->
                        case c of 'R' -> red
                                'B' -> blue
                                'Y' -> yellow
                                _   -> white)
```

Integration

```
integral :: Behavior Float -> Behavior Float
```

Integrates a floating-point behavior over time

(Implemented by approximating the integral with a sum, formed from the values of the behavior at sampled times)

Extremely useful for realistic animations, where a body is moving under the influence of forces (remember Newton's second law!)

Example

The x -coordinate of a moving body as a function of time, from some acceleration given by Newton's second law. Integrating the acceleration once gives velocity, and twice yields position

```
startpos = 0 :: Behavior Float
startvel = 10 :: Behavior Float

xacc = ...complex definition... :: Behavior Float -- acceleration
xvel = integral xacc + startvel -- velocity
xpos = integral xvel + startpos -- position
```

(Same can be done for the y -coordinate, to give full movement)

Snapshot

To “sample” a behavior when an event occurs:

```
snapshot :: Event a -> Behavior b -> Event (a,b)
```

The pair contains the value of the event and the current value of the behavior

So in a sense it “samples” the behavior for each input event

A variation that throws away the first component:

```
snapshot_ :: Event a -> Behavior b -> Event b
snapshot_ e b = (e `snapshot` b) ==>> snd
```

Example: `key `snapshot_` time`, a stream of events that contains the current time for each keystroke

Parallel Composition

“Merge” of two event streams:

```
(.|.) :: Event a -> Event a -> Event a
```

Will yield the “union” of the two event streams with the events in the right time order

Events of left stream go first if appearing at same time in both streams

Example:

```
color2 = red `untilB` (lbp ->> blue .|. key ->> yellow)
```

A color behavior that starts as red and switches to blue on left button press, or yellow on any key, whichever comes first

Thus, `b1 .|. b2` is similar to running `b1` and `b2` as parallel threads

Two Variations on Switch

Useful to have behaviors that “update” themselves when some event happen

```
step :: a -> Event a -> Behavior a
a `step` e = lift0 a `switch` (e ==>> lift0)
```

A behavior that starts off as `a` and then switches to the successive values of the events of `e`

Think of a “sample and hold”-circuit

`'A' `step` key`, starts as `'A'` and then switches to last pressed key

stepAccum successively applies functions carried by an event stream

```
stepAccum :: a -> Event (a->a) -> Behavior a
```

Example: counter = 0 `stepAccum` lbp ->> (+1)

A counter that increases every time the left mouse button is pressed

Think of a `stepAccum` function as a program variable, with an initial value of `a`, that for each event `f` in `funcevs` is updated by applying `f` to the current contents

Interesting application in interactive games: use it to generate a random number event stream, where the seed for the generator is updated at each event

Mouse Motion

We model mouse motion as a pair of behaviors, which give the `x`- and `y`-coordinates as functions of time:

```
mouse :: (Behavior Float, Behavior Float)
```

To have a ball, whose color changes with keyboard inputs, follow the mouse:

```
ball = paint color4 circ3
circ3 = translate mouse (ell 0.2 0.2)
color4 = white `switch` ((key `snapshot` color4) ==> \(c,old) ->
  case c of 'R' -> red
            'B' -> blue
            'Y' -> yellow
            _   -> lift0 old)
```

`stepAccum` can be defined in terms of our previously defined behavior primitives:

```
a `stepAccum` e = b
  where b = a `step` ((e `snapshot` b) ==> uncurry ($))
```

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y -- the opposite of currying
f $ x = f x           -- explicit operator for function application
```

(Thus, `uncurry ($) (f,x) = f $ x = f x`)

It takes some time to digest this. But note the recursive use of `b`! The `snapshot` will sample the *current* value of `b`, which is then used to compute its new value

Paddleball

Now we have finally reached the level of pong :-)

Paddleball - a game with three walls, a bouncing ball, and a mouse-controlled paddle to prevent the ball going off the fourth side

```
paddleball vel = walls `over` paddle `over` pball vel
walls = let upper = paint blue (translate ( 0,1.7) (rec 4.4 0.05))
         left  = paint blue (translate (-2.2,0) (rec 0.05 3.4))
         right = paint blue (translate ( 2.2,0) (rec 0.05 3.4))
       in upper `over` left `over` right
paddle = paint red (translate (fst mouse, -1.7) (rec 0.5 0.05))
```

```
ppball vel =
  let xvel    = vel `stepAccum` xbounce ->> negate
      xpos    = integral xvel
      xbounce = when (xpos >* 2 ||* xpos <* -2)
      yvel    = vel `stepAccum` ybounce ->> negate
      ypos    = integral yvel
      ybounce = when (ypos >* 1.5
        ||* ypos      `between` (-2.0,-1.5) &&*
          fst mouse `between` (xpos-0.25,xpos+0.25))
  in paint yellow (translate (xpos, ypos) (ell 0.2 0.2))

x `between` (a,b) = x >* a &&* x <* b
```

Chapter 16: Communicating With the Outside World

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

August 17, 2007

SoE Chapter 16: Communicating With the Outside World (revised 2007-08-17)

File Handling

Opening and closing files:

```
openFile :: FilePath -> IOMode -> IO Handle
hClose   :: Handle -> IO ()

Type FilePath = String
Data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Files Etc.

Haskell has a fairly conventional set of file operations

These are IO actions

Let's have a closer look at them!

SoE Chapter 16: Communicating With the Outside World (revised 2007-08-17)

1

Reading and Writing Files

Some file operations in write mode:

```
hPutChar :: Handle -> Char -> IO ()      -- write a character
hPutStr  :: Handle -> String -> IO ()   -- write a string
hPutStrLn :: Handle -> String -> IO ()  -- write a line
hPrint   :: Show a => Handle -> a -> IO () -- write a Showable value
```

Some file operations in read mode:

```
hGetChar :: Handle -> IO Char      -- read a character
hGetLine :: Handle -> IO String    -- read a line
hGetContents :: Handle -> IO String -- read full contents of file,
                                     -- lazily
```

File Handling Without Handles

Write to a file without handles:

```
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

But using these commands means opening and closing the file each time.
With explicit file handling we can avoid this inefficiency

Channels

Haskell also recognizes *channels* such as `stdin`, `stdout`, and `stderr`
(straightforward for those who know unix)

These can be used as handles

Some examples:

```
getChar = hGetChar stdin
putChar = hPutChar stdout
```

More Advanced File I/O Operations

Haskell has a standard library (module) for more advanced file handling and I/O in general

Importing this library gives access to the functions for this:

```
import System.IO
```

Exception Handling

Haskell has *exception handling* for IO actions

Errors (exceptions) can be generated for instance from attempting to open a non-existing file, or reading the end-of-file

We may want to recover from these errors

Exception handling can be used for this

Exceptions have type `IOError`

Catching Exceptions

A function to catch exceptions:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

First argument is action to be executed. If an exception is thrown, then it is handled by the second argument which reads the exception and returns an IO action to be executed

If no exception handling is provided, Haskell invokes a default handler that prints an error message and terminates the program

Example: variation of `getChar` returning newline if any exception is thrown:

```
getChar' :: IO Char
getChar' = catch getChar (\e -> return '\n')
```

A refinement: return newline if EOF is encountered, otherwise pass exception upwards (nested exception handling is possible):

```
getChar' :: IO Char
getChar' = catch getChar (\e -> if isEOFError e then return '\n'
                                else ioError e)
```

`isEOFError :: IOError -> Bool` tests for EOF exception

`ioError` throws the exception upwards

More examples in book!

Lambda Calculus and Type Inference

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
<http://www.idt.mdh.se/~blr/>

August 17, 2007

Lambda Calculus and Type Inference (revised 2007-08-17)

Lambda Calculus

Formal calculus

Invented by logicians around 1930

Formal syntax for functions, and function application

Gives a certain “computational” meaning to function application

Theorems about reduction order (which possible subcomputation to execute first)

This is related to call-by-value/call-by-need

Several variations of the calculus

Lambda Calculus and Type Inference (revised 2007-08-17)

2

The Topics

Lambda Calculus: a formal calculus for functions and how to compute with them

Type Inference: how to find the possible type(s) of expressions, without explicit typing

Lambda Calculus and Type Inference (revised 2007-08-17)

1

The Simple Untyped Lambda Calculus

A term in this calculus is:

- a *variable* x ,
- a *lambda-abstraction* $\lambda x.e$, or
- an *application* $e_1 e_2$

Some examples:

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

Any term can be applied to any term, no concept of (function) types

Syntax: function application binds strongest, $\lambda x.x y = \lambda x.(x y) \neq (\lambda x.x) y$

Lambda Calculus and Type Inference (revised 2007-08-17)

3

Untyped Lambda Calculus with Constants

We can extend the syntax with constants, for instance:

1, 17, +, [], :

We can then write terms closer to Haskell, like

$17 + x$ $\lambda x.(x + y)$ $\lambda l.\lambda x.(l : x)$

Every Haskell program can be translated into an intermediate form, which essentially is a lambda calculus with constants

Some constants (like +) can be given a computational meaning, more on this later

Equivalences

Some lambda-expressions are considered equivalent ($e_1 \equiv e_2$)

Rule 1: change of name of bound variable gives an equivalent expression (*alpha-conversion*)

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

Quite natural, right?

However, beware of *variable capture*:

$\lambda x.\lambda y.x \not\equiv \lambda y.\lambda y.y$

Renaming must avoid name clashes with locally bound variables

Note that $(\lambda x.\lambda y.x) 17 = \lambda y.17$, whereas $(\lambda y.\lambda y.y) 17 = \lambda y.y$. Different!

Beta-reduction

A lambda abstraction applied to an expression can be *beta-reduced*:

$(\lambda x.x + x) 9 \rightarrow_{\beta} 9 + 9$

Beta-reduction means substitute actual argument for symbolic parameter in function body

Works also with symbolic arguments:

$(\lambda x.x + x) (\lambda x.y z) \rightarrow_{\beta} (\lambda x.y z) + (\lambda x.y z)$

However, beware of *variable capture*:

$(\lambda x.\lambda y.(x + y)) y \not\rightarrow_{\beta} \lambda y.(y + y)$

The fix is to first rename the bound variable y :

$(\lambda x.\lambda y.(x + y)) y \equiv (\lambda x.\lambda z.(x + z)) y \rightarrow_{\beta} \lambda z.(y + z)$

Extending the Equivalence

We consider expressions equal if there is a way to convert them into each other, through beta-reductions or “inverse” beta-reductions

For instance, $(\lambda x.x) 17 \equiv (\lambda y.17) z$ since

$(\lambda x.x) 17 \rightarrow_{\beta} 17 \leftarrow_{\beta} (\lambda y.17) z$

Some Encodings

Many mathematical concepts can be *encoded* in the lambda-calculus

That is, they can be translated into the calculus

For instance, we can encode the *boolean constants*, and a *conditional* (functional if-then-else):

$$\begin{aligned} \text{TRUE} &= \lambda x.\lambda y.x \\ \text{FALSE} &= \lambda x.\lambda y.y \\ \text{COND} &= \lambda p.\lambda q.\lambda r.(p\ q\ r) \end{aligned}$$

Boolean connectives (and, or) can also be encoded

As well as lists, integers, . . .

Actually *anything you can do in a functional language!*

An example of how *COND* works:

$$\begin{aligned} \text{COND TRUE } A\ B &\rightarrow_{\beta} (\lambda p.\lambda q.\lambda r.(p\ q\ r)) (\lambda x.\lambda y.x)\ A\ B \\ &\rightarrow_{\beta} (\lambda q.\lambda r.((\lambda x.\lambda y.x)\ q\ r))\ A\ B \\ &\rightarrow_{\beta} (\lambda r.((\lambda x.\lambda y.x)\ A\ r))\ B \\ &\rightarrow_{\beta} (\lambda x.\lambda y.x)\ A\ B \\ &\rightarrow_{\beta} \lambda y.A\ B \\ &\rightarrow_{\beta} A \end{aligned}$$

Try evaluating *COND FALSE A B* yourself!

Nontermination

Consider this expression:

$$(\lambda x.x\ x)\ (\lambda x.x\ x)$$

What if we beta-reduce it?

$$(\lambda x.x\ x)\ (\lambda x.x\ x) \rightarrow_{\beta} (\lambda x.x\ x)\ (\lambda x.x\ x)$$

Whoa, we got back the same! Scary . . .

Clearly, we can reduce ad infinitum

The lambda-calculus thus contains *nonterminating* reductions

Recursion

Now consider this expression:

$$\lambda h.(\lambda x.h (x x)) (\lambda x.h (x x))$$

Let's call it Y

What if we apply it to a function f ?

$$\begin{aligned} Y f &= \lambda h.(\lambda x.h (x x)) (\lambda x.h (x x)) f \\ &\rightarrow_{\beta} (\lambda x.f(x x)) (\lambda x.f(x x)) \\ &\rightarrow_{\beta} f ((\lambda x.f (x x)) (\lambda x.f (x x))) \\ &= f (Y f) \end{aligned}$$

Hmm, we got back f applied to $Y f$

Y is called *fixed-point combinator*

It encodes *recursion*

To see why, consider the recursive definition

$$x = f x$$

The solution is

$$x = f f f \dots$$

Likewise,

$$Y f = f (Y f) = f f (Y f) = f f f \dots$$

Thus, $x = Y f!$

Note that *all* recursive definitions can be written on the form $x = f x$

Reduction Strategies

Any application of a lambda-abstraction in an expression can be beta-reduced

Each such position is called a *redex*

An expression can contain several redexes

Can you find all redexes in this expression?

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

Try reduce them in different orders!

Does the order of reducing redexes matter?

Well, yes and no:

Theorem: *if two different reduction orders of the same expression end in expressions that cannot be further reduced, then these expressions must be the same*

However, we can have potentially infinite reductions:

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

Reducing the “outermost” redex yields y

But the innermost redex can be reduced infinitely many times – nontermination!

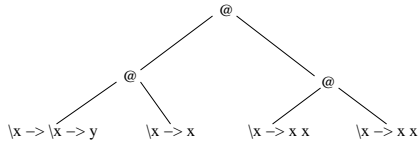
So the order *does* matter, as regards termination anyway!

Normal Order Reduction

Consider this slight adaptation of the previous example:

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

Let us draw it as a tree:



Reducing the “leftmost-outermost” redex twice yields y

Type Inference

You have seen that Haskell systems can find types for expressions:

$$\begin{aligned} y f &= f (y f) \\ y &:: (a \rightarrow a) \rightarrow a \end{aligned}$$

As we have mentioned, the *most general* type is always found

Some other reduction orders do not terminate

This is not a coincidence!

To reduce the leftmost-outermost redex in each step is called *normal order reduction*

Theorem: *if there is a reduction order that terminates, then normal order reduction terminates*

Normal order reduction corresponds to call-by-need in functional languages

There is an interesting theory behind Haskell-style type inference

To infer means “to prove”, or “to deduce”

A type system is a *logic*, whose statements are of form “expression e has type τ ”

To infer a type means to *prove* a statement like above

A *type inference algorithm* finds a type if it exists: it is thus a *proof search algorithm*

Such an algorithm exists for Haskell’s type system

Hindley-Milner's Type System

Haskell's type system extends a simpler type system known as *Hindley-Milner's type system* (HM)

HM does not have type classes, and it is defined over a simple lambda-calculus language

Here, we will briefly describe a subset of HM:

Language is lambda-calculus with constants (also known functions, like +)

Constants have typing given in advance

Recursion can be encoded through constant Y (fixed-point combinator)

A language of types:

- type constants (like Int)
- type variables α
- function types $\tau \rightarrow \tau'$ (\rightarrow is a *type constructor*)
- *type schemes* $\forall\alpha.\tau$

Other type constructors (for list types etc) can easily be added

Type schemes like $\forall\alpha.\alpha \rightarrow \alpha$ correspond to polymorphic Haskell types like $a \rightarrow a$

Statements

Statements of form $A \vdash e : \tau$, where e is expression and τ is a type

A is a set of *assumptions*, of form $x : \tau$ (read: "variable x has type τ ")

$A \vdash e : \tau$ is read "under the assumptions on typings of variables in A , the expression e can have type τ "

So in order to prove such a statement, we must "guess" some typings of variables in e and then check that we can give e a consistent type with these assumptions

The key in type inference is to make these "guesses" systematically

Inference Rules

Axioms and allowed proof steps are given as a set of *inference rules*

Each inference rule has a number of *premises* and a *conclusion*

Often written on the form

$$\frac{\text{premise 1} \ \dots \ \text{premise } n}{\text{conclusion}}$$

Example (modus ponens in propositional logic):

$$\frac{P \quad P \implies Q}{Q}$$

Axioms are inference rules without premises

Hindley-Milner Inference Rules

A selection of rules from the HM inference system:

$$\begin{array}{l} A \cup \{x : \tau\} \vdash x : \tau \quad [VAR] \\ \frac{A \cup \{x : \sigma\} \vdash e : \tau}{A \vdash \lambda x. e : \sigma \rightarrow \tau} \quad [ABS] \\ \frac{A \vdash e : \sigma \rightarrow \tau \quad A \vdash e' : \sigma}{A \vdash e e' : \tau} \quad [APP] \\ \frac{A \vdash e : \forall \alpha. \tau}{A \vdash e : \tau[\sigma/\alpha]} \quad [SPEC] \end{array}$$

$(\tau[\sigma/\alpha])$ stands for τ with σ replacing every occurrence of α

An Example

Best way to understand inference rules is to see a derivation

Let's infer a type for $(\lambda y. (tail\ y))\ nil$

Extend language of types with list types $[\tau]$

Assume given typings for constants:

$$A = \{nil : \forall \alpha. [\alpha], tail : \forall \alpha. [\alpha] \rightarrow [\alpha]\}$$

Derivation

$$\frac{\frac{T}{A \vdash \lambda y. (tail\ y) : [\gamma] \rightarrow [\gamma]} \quad \frac{A \vdash nil : \forall \alpha. [\alpha]}{A \vdash nil : [\gamma]}}{A \vdash (\lambda y. (tail\ y))\ nil : [\gamma]}$$

where

$$T = \frac{\frac{A \cup \{y : [\gamma]\} \vdash tail : \forall \alpha. [\alpha] \rightarrow [\alpha]}{A \cup \{y : [\gamma]\} \vdash tail : [\gamma] \rightarrow [\gamma]} \quad A \cup \{y : [\gamma]\} \vdash y : [\gamma]}{A \cup \{y : [\gamma]\} \vdash tail\ y : [\gamma]}$$

Inference Algorithm

There is a classical algorithm for type inference in the HM system

Called *algorithm* \mathcal{W}

Basically a systematic and efficient way to infer types like we did in the example

The algorithm uses *unification*, remember this when you learn logic programming!

It has been proved that algorithm \mathcal{W} always yields a *most general type* for any typable expression

“Most general” means that any other possible type for the expression can be obtained from the most general type by instantiating its type variables

A More Practical Type Inference Example

Define

```
length [] = 0
length (x:xs) = 1 + length xs
```

Derive the most general type for length!

For simplicity, assume that $0 :: \text{Int}$, $1 :: \text{Int}$, and
 $(+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

See next four slides for how to do it ...

Solving the equation for the first declaration:

```
length [] = 0
```

```
length :: c -> d (since it is applied to one argument)
```

```
c = [a] (from what we know about the type of [])
```

```
d = Int (since length [] :: d, 0 :: Int, and both sides of the
declaration must have the same type)
```

```
Thus, length :: [a] -> Int
```

Is this consistent with the second case in declaration of length?

Type inference can be seen as equation solving: every declaration gives rise to a “type equation” constraining the types for the untyped identifiers

These equations can be solved to find the types

In our example, we already know:

```
0 :: Int
1 :: Int
(+) :: Int -> Int -> Int
[] :: [a]
(:) :: b -> [b] -> [b]
```

Note different type variable names in types of `[]` and `(:)`, to make sure they're not mixed up

Second declaration

```
length (x:xs) = 1 + length xs
```

Must first find possible types for x , xs , $x:xs$

Assume $x :: e$, $xs :: f$

$e = b$, $f = [b]$ (or else $x:xs$ is not well-typed), then $x:xs :: [b]$

Left-hand side: OK if $[b] = [a]$ (so $xs :: [a]$ and $x:xs :: [a]$), and then $\text{length } (x:xs) :: \text{Int}$

Right-hand side: $xs :: [b]$, $\text{length} :: [a] \rightarrow \text{Int}$ and $xs :: [a]$ gives $\text{length } xs :: \text{Int}$

```
1 :: Int, length xs :: Int, (+) :: Int -> Int -> Int gives
1 + length xs :: Int
```

Applications of Type Inference

Thus type of LHS = type of RHS! We're done.

Result: `length :: [a] -> Int`

Must be a most general type since we were careful not to make any stronger assumptions than necessary about any types

(In the formal type system, we would obtain $length : \forall \alpha. [\alpha] \rightarrow Int$)

HM-style type systems used in some advanced functional languages, most notably ML and Haskell

Similar type inference systems can be used for *program analysis*

Types (not the usual ones) can stand for properties

$e : \tau$ then means “expression (program) e has property τ ”

Can be used in optimizing compilers

Another application: *dimensional analysis*, to find whether equations are sound w.r.t. physical dimensions