Types and Lazy Evaluation

Outline

◇ Types in programming languages
  - Type safety and strong typing
  - Polymorphism
  - Type inference
  - Case study: Haskell

◇ Lazy Evaluation
  - Lazyness vs. strictness; purity
  - Lazy evaluation examples

Types in Programming

- A type is a collection of computational entities that share some common property.

- There are 3 main uses:
  - Naming and organizing concepts.
  - Making sure that bit sequences in memory are interpreted consistently.
  - Providing information (e.g. size) to the compiler about data manipulated by the program

- Type error: when computational entity is used in an inconsistent manner.

Type Safety

- A PL is type safe is no program is allowed to violate type distinctions.

- More specifically, a data of a given type cannot be ”seen” as data of another type
  - *in-situ* casts are not type safe
  - pointer arithmetic is not safe
  - consequently C is not type-safe

- Compile-time vs Run-time type checking
  - Run-time checking: data is paired with its type during execution
    - type consistency is checked before every operation
    - type of data may change during execution
    - overhead incurred
  - Compile-time checking: type consistency is checked at compile time
    - Type is stripped from data during run-time
    - Data cannot change its type during execution
    - No type consistency checks at execution; no overhead

Type Inference

- Type safe languages:
  - Strongly typed: all type consistency can be checked at compile-time; there's no need for run-time checks.
  - Weakly typed: some type consistency checks must be done at run-time
- Some strongly typed languages infer (rather than just check) the types of their data
  - Haskell
  - Ocaml
- Type inference can be viewed as a type of semantics and can be defined via reasoning rules.

Polymorphism and Overloading

- Polymorphism: a symbol may have multiple types simultaneously
- Forms of polymorphism:
  - Parametric polymorphism: function may be applied to any arguments whose types match a type expression involving type variables – Haskell and Ocaml fall into this category.
  - Ad-hoc polymorphism: (also known as overloading): two or more implementations with different types are referred to by the same name
  - Subtype polymorphism: a *subtype* relation is defined between types; an expression with a given type can be used as argument anywhere where a subtype of the current type is expected – Haskell also has this form of polymorphism via *type classes* (not covered).

Haskell

- Functional, strongly typed, polymorphic, lazy (non-strict)
- Named after Haskell Curry – pioneer of lambda calculus
- Many implementations (some quite efficient), many extensions
- Elegant, theoretically clean
- Very well supported, see `www.haskell.org`
  – We shall be using the interpreter GHCi.

Syntax

- Expression based, `2+3` is a legal program
- Functional abstractions: `\ x -> x+1`
- Types
  - Rich, polymorphic type system
  - Java Generics was based on the same principle

Sample Interaction

```
Prelude> :set +t
Prelude> 2+3
5 :: Integer
Prelude> (\\x -> x+1) 3
4 :: Integer
Prelude> let
   factorial x = if x == 0
                 then 1
                 else x * (factorial (x-1))
in factorial 100
933262154439441526816992388562667004907
159682643816214685929638952175999932
299156089414639761565182862536979208272
237582511852109168640000000000000000000
000000 :: Integer
```

Syntax

- Operators are written infix
- Function application is treated as an invisible operator
  - `f x` is function `f` applied to `x`
  - `f x y` is evaluated as `(f x) y` (curried evaluation).
  - `f (g y)` means that `g` is applied to `y` first, and then `f` is applied to the result.
- Curried application:

```
Prelude> let f x y = x+y in let g = f 2 in g 3
5 :: Integer
```

Interactive Environment

◇ Files can be edited and loaded

◇ Full programming features in loaded files
  – Definition of new symbols
  – Operator declarations
  – Datatypes

◇ The shell only allows evaluation of expressions

◇ Open an editor window with `:edit`

Factorial

- Type in the editor window

```
factorial x =
  if x == 0 then 1
            else x * (factorial (x-1))
```

- Then load the file in Hugs and run it

```
Hugs> :load r:\\cs2104_lec09\\factorial.hs
Main> factorial 10
3628800 :: Integer
```

- The module name has changed to `Main`
  - That is the default module name, in case we don't define one in our file
  - The file may be reloaded every time we change it
  - The command is `:reload`

Alternative Factorial Definitions

Equational definitions

```
factorial2 0 = 1
factorial2 x | x > 0 = x * (factorial2 (x-1))

Main> factorial2 10
3628800 :: Integer
Main> factorial2 10.0
3628800.0 :: Double
Main> :type factorial2
factorial2 :: (Num a, Ord a) => a -> a
```

- Every symbol has a type, automatically inferred
- `factorial2` is a function type, takes type `a` into type `a`, where
  - `a` is a numeric type
  - `a` is also an ordered type

Types

- We can specify symbol types when we define a symbol
- Good practice, adds an extra layer of verification
- Useful sometimes to restrict the types of a function

```
factorial3 :: Integer -> Integer
factorial3 0 = 1
factorial3 n | n>0 = n*(factorial (n-1))
Main> factorial3 10
3628800 :: Integer
*Main> factorial3 10.0

<interactive>:1:12:
    No instance for (Fractional Integer)
      arising from the literal ‘10.0’
    Possible fix: add an instance declaration for (Fractional Integer)
    In the first argument of ‘factorial3’, namely ‘10.0’
    In the expression: factorial3 10.0
    In an equation for ‘it’: it = factorial3 10.0
```

Types

```
factorial4 :: Int -> Int
factorial4 0 = 1
factorial4 n | n>0 = n*(factorial4 (n-1))

Main> factorial4 10
3628800 :: Int
Main> factorial4 100
0 :: Int
```

If the type is not specified, the most general type is inferred.

Pattern Matching and Equations

- Recursive functions can be defined with equations
- Left-hand side of an equation uses pattern-matching, similar to Prolog
- It doesn't work in the reverse direction
  - the append of lists will not subtract
- Very powerful, we can match complex datatypes

Infix Operators and Sections

- Section: partial application of an operator
- Comes in handy with infix operators
- New infix operators can be defined by user
- Infix operators can be used in prefix form if quoted

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

Main> 3**4
25 :: Integer
Main> (**) 3 4
25 :: Integer
Main> let f = (3**) in f 4
25 :: Integer
Main> let f = (**4) in f 3
25 :: Integer
Main> 3 ** 4+1
26 :: Integer
Main> 3**(4+1)
34 :: Integer
Main> let f x y = x ** 2 ** y in 3 `f` 4
ERROR - Ambiguous use of operator "(**)" with "(**)"
Main> let f x y = (x ** 2) ** y in 3 `f` 4
185 :: Integer
```

Associativity of Infix Operators

```
infixl 9 ***
(***) :: Integer -> Integer -> Integer
x *** y = (x-y)*(x-y)

Main> 5 *** 2 *** 1
64 :: Integer
Main> (5***2)***1
64 :: Integer
Main> 5 *** ( 2 *** 1)
16 :: Integer
Main> 1+5 *** 2 *** 1
225 :: Integer
Main> 1+5 *** 2 *** 1
65 :: Integer
Main>
```

- precedence 9 is highest
- `infixl` : left associative
- `infixr` : right associative

Lists

- ◇ Simple colon `:` is the list constructor.
- ◇ Empty list: `[]`
- ◇ List type: `[a]`, where `a` is the type of elements in the list.
- ◇ `head l` is the head of the list
- ◇ `tail l` is the tail of the list
- ◇ enumeration of elements: `[1,2,3]`
- ◇ list append: `++`
  – `[1,2,3]++[4,5,6]` evaluates to `[1,2,3,4,5,6]`

Higher Order Programming

```
Main> map (1+) [1,2,3]
[2,3,4] :: [Integer]

Main> foldl (*) 1 [2,3,4,5]
120 :: Integer

Main> filter (>0) [1,-1,2,-2,3,-3]
[1,2,3] :: [Integer]

Main> foldl (++) [] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,4,5,6,7,8,9] :: [Integer]

Main> foldr (++) [] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,4,5,6,7,8,9] :: [Integer]

Main> foldr (++) [100,200,300] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,4,5,6,7,8,9,100,200,300] :: [Integer]

Main> foldl (++) [100,200,300] [[1,2,3],[4,5,6],[7,8,9]]
[100,200,300,1,2,3,4,5,6,7,8,9] :: [Integer]
```

Higher Order Programming

```
Main> zip [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')] :: [(Integer,Char)]

Main> zipWith (+) [1,2,3] [10,20,30]
[11,22,33] :: [Integer]

Main> take 5 [1,2,3,4,5,6,7,8,9]
[1,2,3,4,5] :: [Integer]

Main> drop 5 [1,2,3,4,5,6,7,8,9]
[6,7,8,9] :: [Integer]
```

List Comprehensions

```
Main> [1..10]
[1,2,3,4,5,6,7,8,9,10] :: [Integer]

Main> [x | x <- [1..10]]
[1,2,3,4,5,6,7,8,9,10] :: [Integer]

Main> [x | x <- [1..10] , x `mod` 2 == 0]
[2,4,6,8,10] :: [Integer]

Main> [x | x <- [2,4..10]]
[2,4,6,8,10] :: [Integer]

Main> [x+y| x <- [1,3..10] , y<-[100,130..140]]
[101,131,103,133,105,135,107,137,109,139] :: [Integer]

Main> foldr (*) 1 [1..10]
3628800 :: Integer

Main> let fact x =
        let prod = foldr (*) 1
        in prod [1..x]
      in fact 10
3628800 :: Integer
```

Polymorphic Types

```
Main> map (1+) [2,3,4]
[3,4,5] :: [Integer]

Main> :type map
map :: (a -> b) -> [a] -> [b]

Main> :type map (1+)
map (1 +) :: Num a => [a] -> [a]

Main> :type (+)
(+) :: Num a => a -> a -> a

Main> foldr (+) 0 [1..5]
15 :: Integer

Main> :type foldr
foldr :: (a -> b -> b) -> b -> [a] -> b

Main> :type foldr (+)
foldr (+) :: Num a => a -> [a] -> a

Main> length ['a','b','c']
3 :: Int

Main> :type length
length :: [a] -> Int
```

```
Main> :type \x -> x
\x -> x :: a -> a

Main> :type \x y -> x
\x y -> x :: a -> b -> a

Main> :type \x y -> y
\x y -> y :: a -> b -> b

Main> :type \f g -> g (f g)
\f g -> g (f g) ::
((a -> b) -> a) -> (a -> b) -> b

Main> :type \f g x -> g (f g)
\f g x -> g (f g) ::
((a -> b) -> a) -> (a -> b) -> c -> b

Main> :type \x f g -> f g (x g)
\x f g -> f g (x g) ::
(a -> b) -> (a -> b -> c) -> a -> c

Main> :type \x y f -> f (x (\w -> f w)) (y f x)
\x y f -> f (x (\w -> f w)) (y f x) ::
((a -> b -> c) -> a) ->
((a -> b -> c) ->
((a -> b -> c) -> a) -> b) ->
(a -> b -> c) -> c
```

Type Language

$$
\begin{array}{rcll}
< type > & ::= & < typeconst > & (a) \\
 & | & < typevar > & (b) \\
 & | & < type > \rightarrow < type > & (c) \\
< typeconst > & ::= & \texttt{Int} \mid \texttt{Boolean} \mid \ldots & (d) \\
< typevar > & ::= & < upper\_case\_letter > & (e) \\
 & & & \\
< expr > & ::= & < const > & (f) \\
 & | & < var > & (g) \\
 & | & (< expr > < expr >) & (h) \\
 & | & (\backslash < var > \rightarrow < expr >) & (i) \\
< const > & ::= & \mathsf{0} \mid \mathsf{1} \mid \cdots & (j) \\
 & | & \texttt{true} \mid \texttt{false} \mid \texttt{plus} \mid \cdots & \\
< var > & ::= & < lower\_case\_letter > & (k) \\
 & & & \\
< type\_assignment > & ::= & < expr > :: < type > & (l)
\end{array}
$$

Typing Judgements

$$\frac{\textit{premise}_1 \quad \textit{premise}_2 \quad \ldots \textit{premise}_k}{\Gamma, \Delta \vdash e :: T}$$

Γ is a type environment, Δ is a set of type unification constraints.

Typing Rules

$$\frac{}{\{x :: T\}, \emptyset \vdash x :: T} \quad (\textsc{const})$$

$$\frac{}{\emptyset, \emptyset \vdash \texttt{true} :: \texttt{Boolean}} \qquad \frac{}{\emptyset, \emptyset \vdash \texttt{false} :: \texttt{Boolean}}$$

$$\frac{}{\emptyset, \emptyset \vdash \texttt{plus} :: \texttt{Int}- > \texttt{Int}}$$

$$\frac{}{\emptyset, \emptyset \vdash \texttt{0} :: \texttt{Int}} \qquad \frac{}{\emptyset, \emptyset \vdash \texttt{1} :: \texttt{Int}} \qquad \frac{}{\emptyset, \emptyset \vdash \texttt{2} :: \texttt{Int}}$$

Typing Rules

$$\frac{\Gamma_1, \Delta_1 \vdash e_1 :: T_1 \quad \Gamma_2, \Delta_2 \vdash e_2 :: T_2}{\Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2 \cup \{T_1 = T_2 \rightarrow T_3\} \vdash (e_1\, e_2) :: T_3} \quad (\textsc{app})$$

T_3 is a new type variable

$$\frac{\{x_1 :: T_1\}, \emptyset \vdash x :: T_1 \quad \Gamma \cup \{x :: T_1'\}, \Delta \vdash e :: T_2}{\Gamma, \Delta \cup \{T_1 = T_1'\} \vdash \backslash x \rightarrow e :: T_1 \rightarrow T_2} \quad (\textsc{abs})$$

Typing Example

- A type judgement is valid as long as its set of type unification constraints is satisfiable.
- Simple example: the identity function:

$$\dfrac{\dfrac{\dfrac{}{\{x:T_1\},\phi \vdash x:T_1} \qquad \dfrac{}{\{x:T_2\},\phi \vdash x:T_2}}{\phi,\{T_1=T_2\} \vdash \backslash x \to x : T_1 \to T_2} \qquad \dfrac{}{\phi,\phi \vdash 3:\text{int}}}{\phi,\{T_1=T_2, T_1=\text{int}\} \vdash (\backslash x \to x)\,3 : T_2}$$

In a typing tree, every horizontal line must be a valid typing judgement.

Example: Composition Function

$$\dfrac{\dfrac{}{\vdash f : T_2} \qquad \dfrac{\dfrac{}{\vdash g : T_1} \qquad \dfrac{\dfrac{}{\{x : C\}, \phi \vdash x : C} \qquad \dfrac{\dfrac{}{\{f : T_2'\}, \phi \vdash f : T_2'} \qquad \dfrac{\dfrac{}{\{g : T_1'\}, \phi \vdash g : T_1'} \qquad \dfrac{}{\{x : C'\}, \phi \vdash x : C'}}{\{g : T_1', x : C'\}, \{T_1' = C' \to A\} \vdash (g\ x) : A}}{\{f : T_2', g : T_1', x : C'\}, \{T_2' = A \to B, T_1' = C' \to A\} \vdash (f\ (g\ x)) : B}}{\{f : T_2', g : T_1'\}, \{C' = C, T_2' = A \to B, T_1' = C' \to A\} \vdash \backslash x \to (f\ (g\ x)) : C \to B}}{\{f : T_2'\}, \{T_1 = T_1', C' = C, T_2' = A \to B, T_1' = C' \to A\} \vdash \backslash g \to \backslash x \to (f\ (g\ x)) : T_1 \to (C \to B)}}{\{T_2 = T_2', T_1 = T_1', C' = C, T_2' = A \to B, T_1' = C' \to A\} \vdash \backslash f \to \backslash g \to \backslash x \to (f (g\ x)) : T_2 \to T_1 \to (C \to B)}$$

After solving the unification equations, the type effectively becomes

$$(A \to B) \to (C \to A) \to (C \to B)$$

Type Inference Failures

- `\ f -> ( f f )`

- `\ f -> (f (x+f))`

Prolog type-checker demo

Lazy Evaluation

- Nothing is evaluated before it is actually needed

```
Main> let f x = f x in f 1
{Interrupted!}

Main> let f x = f x in [1+2,f 1]
[3,{Interrupted!}

Main> let f x = f x in head [1+2,f 1]
3 :: Integer

Main> let f x = f x
      in (\ x a b ->
              if x == 0 then a else b
         ) 1 (f 1) 2
2 :: Integer

Main> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10] :: [Integer]
```

Lazy Evaluation

- Nothing is evaluated before it is actually needed

```
Main> let f x = f x in f 1
{Interrupted!}

Main> let f x = f x i
[3,{Interrupted!}

Main> let f x = f x i
3 :: Integer

Main> let f x = f x
      in (\ x a b ->
             if x == (
         ) 1 (f 1) 2
2 :: Integer

Main> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10] :: [Integer]
```

- Also known as on-demand evaluation
- The opposite: strict
- Strict languages are the norm
  - strict evaluation more efficient and easier to implement
- Haskell: most well-known lazy language
  - Elegant abstract concepts can be imported from math due to lazy evaluation

Lazy Evaluation Implementation

- Haskell uses *memoized call by name*
- Argument to function is not computed before call; rather it is substituted for the formal argument as an expression.
- Substitution may occur in multiple places; upon the first evaluation, the value of the expression is *memoized* (i.e. stored for later use), and all subsequent references to the expression will access the memoized value, rather than recompute
- An expression that appears as actual argument may never be computed.
- Infinite computations, or exceptional conditions such as division by zero become less dangerous

Purity

- Functions with side effect: when called multiple times with same arguments, returns different results
  - Requires assignment
  - Do not mix well with lazy evaluation, since every expression is evaluated only once – value is memoized, and re-used in subsequent occurrences of same expression.

- Pure function: Function without side-effect.
  - Preferred in a lazy evaluation setting

- Pure language: Language where it is impossible to write functions with side-effects.
  - Usually assignment is removed
  - Haskell is a pure language

Infinite Lists

- Due to lazyness, we can *specify* a list without end
  - Ok as long as we *don't use all the list*
  - Specification is simpler and more elegant as compared to finite lists.
- The list comprehension `[k..]` denotes the infinite list that starts at `k` and contains all the numbers greater than `k` in increasing order.
- Useful only if we only take a finite number of elements in the list
- Using recursion we can define infinite lists containing any series
- Also called streams.
- Lead to simple, elegant programs, all due to lazy evaluation

Fibonacci, etc...

```
Main> let fib = 0:1:
                (zipWith (+) fib (tail fib))
      in take 10 fib
[0,1,1,2,3,5,8,13,21,34] :: [Integer]

Main> let pow2 = 1:map (2*) pow2
      in take 10 pow2
[1,2,4,8,16,32,64,128,256,512] :: [Integer]

Main> let sqrt2 = 1:map
                    (\x ->(x+2.0/x)/2.0)
                    sqrt2
      in take 6 sqrt2
[1.0,1.5,1.41666666666667,1.41421568627451,
1.41421356237469,1.41421356237309] :: [Double]
```

Prime Numbers

```
Main> let primes = sieve [2..]
          where sieve (p:xs) =
             p : sieve [x | x<-xs,
                           x `mod` p /= 0]
      in take 20 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,
47,53,59,61,67,71] :: [Integer]
```

Hamming Numbers

```
hamming = 1 :
          map (2*) hamming
           `merge`
          map (3*) hamming
           `merge`
          map (5*) hamming
  where
  merge (x:xs) (y:ys)
    | x < y = x : xs `merge` (y:ys)
    | x > y = y : (x:xs) `merge` ys
    | otherwise = x : xs `merge` ys
```