L6: Functional Programming

Haskell: Type Checking and Type Inference.
Racket: Simulating lazy evaluation

Mircea Marin
West University of Timişoara
mircea.marin@e-uvt.ro

Preliminary remarks

In Haskell, every expression has a type, which might be monomorphic (without type variables), polymorphic, or with one or more type class constraints in a context.

Examples

```
'w' :: Char
map :: (a->b)->[a]->[b]
elem :: (Foldable t,Eq a)=>a->t a->Bool
```

- `Char` are monomorphic types
- `(a->b)->[a]->[b]` and `(Foldable t,Eq a)=>a->t a->Bool` are polymorphic types
  - `a`,`b` are type variables
  - `Eq` and `Foldable` are type classes

Strong typing

Strong typing = a way to check if an expression is well typed, without any evaluation taking place.

- Haskell is strongly typed. Racket is not strongly typed.

```
> :type map (\x -> x+1) [1,2,3]
map (\x -> x+1) [1,2,3]::Num b=>[b]

> :type [("abc",True),("bob",False)]
[("abc",True),("bob",False)]::[([Char],Bool)]

> :type 1+'2'
<interactive>:1:1: error:
    • No instance for (Num Char) arising from a use of '+'
    • In the expression: 1 + '2'
```

Remark

The benefit of strong typing is obvious: we can catch a lot of type errors before we run a program.

Type inference

Type inference = a way to compute types from expressions and definitions.

- Haskell has a built-in type inference system ⇒ it is possible never to write a type declaration.

Example

```
> prodFun f g = \x -> (f x,g x)
> :type prodFun
prodFun :: (t->a)->(t->b)->t->(a,b)
```

Type inference

Type inference = a way to compute types from expressions and definitions.

- Haskell has a built-in type inference system $\Rightarrow$ it is possible never to write a type declaration.

Example

```
> prodFun f g = \x -> (f x,g x)
> :type prodFun
prodFun :: (t->a)->(t->b)->t->(a,b)
```

Remarks

1. Haskell programmers almost never write type declarations.
2. Writing type declarations is a good idea: it is the *most important single piece of documentation* of an object, since it tells us how it can be used
   - what arguments need to be passed
   - what type of result do we get

- `prodFun::(Int->Bool)->(Int->Char)->Int->(Bool,Char)`
  `prodFun f g = \x -> (f x,g x)`
  The most general type of `prodFun` is
  `(t->a)->(t->b)->t->(a,b)`, but we made it more specific, for `t` = `Int`, `a` = `Bool`, `b` = `Char`.
- Type declarations indicate what type we think a function has. If we got it wrong, the type inference system will detect this. For example

```
fun::Int->Bool->Int
fun True 0 = 0
fun True n = n+1
fun _ n      = n
```

  gives rise to this error in ghci:

```
Couldn't match expected type 'Int' against inferred type 'Bool'
...
```

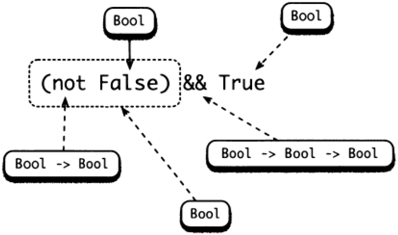
An expression is either

1. a literal or variable of a known type,
2. a function call, or
3. a lambda abstraction.

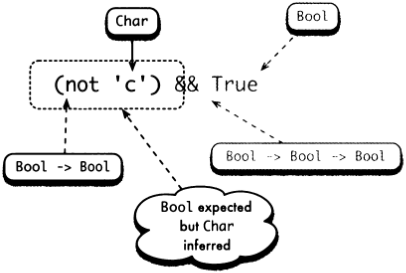
Remark: operators and the `if ... then ... else ...` construct act like functions, but with a different syntax.

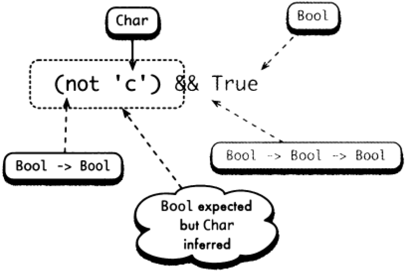
- Type checking a function call `(f e)`
  - `f` must have a type `a->b`
  - `e` must have type `a`, and `(f e)` must have type `b`
- Type checking an abstraction `(\x->e)`
  - `x` must have a type `a`
  - `e` must have type `b`, and `(\x->e)` must have type `a->b`

1. A well typed expression:



2. An ill-typed expression:





The error message of GHCi indicates the cause of the problem:

```
Couldn't match expected type 'Bool' against inferred type 'Char'
In the first argument of 'not', namely 'c'
In the first argument of '(&&)', namely '(not 'c')'
In the expression (not 'c') && True
```

A monomorphic type definition

```
f :: t1->t1->...->tn->t
f p1 p2 ... pk
  | guard1 = e1
  ...
  | guardℓ = eℓ
```

is type-checked as follows:

1. each of the guards *guars*$_i$ must be of type `Bool`
2. each e_i must be of type `t`
3. each pattern $\mathtt{p}_j$ must be consistent with the type $\mathtt{t}_j$ of its argument.
   - Pattern consistency is explained on the next slide.

A pattern `p` is consistent with a type if it will match (some) elements of the type.

Examples

- A variable is consistent with any type
- A literal is consistent with its type
- A pattern `(p:q)` is consistent with a type `[t]` if `p` is consistent with `t` and `q` is consistent with `[t]`.
  For example, `(0:xs)` is consistent with the type `[Int]`, and `(x:xs)` is consistent with any type of lists.

1. Predict the type errors you would obtain by defining the following functions:

```
f n    = 37+n
f True = 34

g 0 = 37
g n = True

h x
  | x>0       = True
  | otherwise = 37
```

Check your answers by typing each definition in a Haskell script, and loading the script into the `ghci`. Remember that you can use `:type` to get the type of an expression.

- In a monomorphic language, an expression is either well typed and has a single type, or it is ill-typed (it has no type).
- In a polymorphic language like Haskell, an object has exactly one polymorphic type, which can be instantiated with many types.

Example (Predefined function `length` has a polymorphic type)

`length::[a]->Int`

This type is a shorthand for saying that `length` has the set of all types `[t]->Int` where `t` is a monotype, that is, a type without type variables.
For example, `length` has types

`[Int]->Int` and `[(Bool,Char)]->Int`

We can apply a polymorphic function to some arguments only if some type constraints are satisfied.

$\Rightarrow$ **type checking** = checking if we can find types which meet the constraints.

We can apply a polymorphic function to some arguments only if some type constraints are satisfied.

$\Rightarrow$ **type checking** = checking if we can find types which meet the constraints.

Polymorphic type checking is based on two notions:

Unification: Finding a type description which meets two or more type constraints

Instantiation: Obtaining a new type from a polymorphic type, by replacing (some of) its type variables with type expressions.

Unification computes the **most general common instance** of two type expressions.

Examples

Q1: Which types meet the two descriptions `(a,[Char])` and `(Int,[b])`?

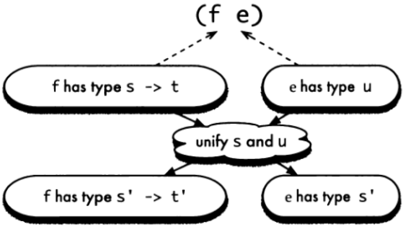
A1: `(Int,[Char])`. We find this type by **unification** = solving the type constraint

`(a,[Char])=(Int,[b])` $\Rightarrow$ `a=Int, [Char]=[b]`
$\Rightarrow$ `a=Int, b=Char`

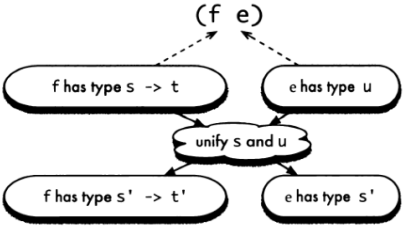
Q2: Which types meet the two descriptions `(a,[a])` and `(b,[c])`?

A2: `(b,[b])`. We find this type by solving the type constraint

`(a,[a])=(b,[c])` $\Rightarrow$ `a=b, [a]=[c]`
$\Rightarrow$ `a=b, c=b`



(f e)
f has type s -> t
e has type u
unify s and u
f has type s' -> t'
e has type s'



Example 1

Type-check `map Circle` where

```
map::(a->b)->[a]->[b]
Circle::Int->Shape
```

unifies `a->b` with `Float->Shape` $\Rightarrow$ `a` $=$ `Float`, `b` $=$ `Shape`. Thus

```
map Circle::[Float]->[Shape]
```

Q: Find the most general general type of the function `foldr` defined by

```
foldr f s []     = s
foldr f s (x:xs) = f x (foldr f s xs)
```

A: `f` takes 3 input arguments $\Rightarrow$ it has type `a1->a2->a3->c`.
Let `a` be type of `s`.

- From the first equation we get the type constraints $a2 = a, a2 = c, a3 = [b]$ because `[]` has a type `[b]` for some type `b` $\Rightarrow a2 = a, a3 = [b], c = a$, thus `foldr::a1->a->[b]->a`
- From the second equation we get that `x::b`, `xs::[b]`, `f::c1->c2->c3`, and the type constraints $c1 = b, c2 = a, c3 = a$, thus `a1=b->a->a` and `foldr::(b->a->a)->a->[b]->a`

Q1: Find the most general general type of `expr` where

```
expr=length ([]++[True]) + length ([]++[2,3,4])
```

Q1: Find the most general general type of `expr` where

```
expr=length ([]++[True]) + length ([]++[2,3,4])
```

A2: **Functions and constants can be used with different types in the same expression.**

- First occurrence of `[]` has type `[Bool]`, and second occurrence of `[]` has type `Integer` ⇒ `expr::Int`.

Q1: Find the most general general type of `expr` where

```
expr=length ([]++[True]) + length ([]++[2,3,4])
```

A2: **Functions and constants can be used with different types in the same expression.**

- First occurrence of `[]` has type `[Bool]`, and second occurrence of `[]` has type `Integer` ⇒ `expr::Int`.

Q2: What should be the type of `funny` defined below?

```
funny xs=length (xs++[True]) + length (xs++[2,3,4])
```

Q1: Find the most general general type of `expr` where

```
expr=length ([]++[True]) + length ([]++[2,3,4])
```

A2: **Functions and constants can be used with different types in the same expression.**

- First occurrence of `[]` has type `[Bool]`, and second occurrence of `[]` has type `Integer` ⇒ `expr::Int`.

Q2: What should be the type of `funny` defined below?

```
funny xs=length (xs++[True]) + length (xs++[2,3,4])
```

A2: **A variable must used with the same type in the same expression.** But

- First occurrence of `xs` should have type `[Bool]`, and second occurrence of `xs` should have type `[Integer]`.
- `[Bool]` and `[Integer]` are not unifiable ⇒ `funny` is ill-typed.

Type checking and type classes

Haskell classes restrict the use of some functions, such as `++`, to types in the class over which they are defined.

- These restrictions are apparent in the contexts which appear in some types.

Example

If we define

```
member []     _ = False
member (x:xs) y = (x==y) || member xs y
```

then the inferred type of `member` will be

```
Eq a => [a] -> a -> Bool
```

because `x`, `y` of type `a` are compared for equality in the definition, thus forcing the type `a` to belong to equality class `Eq`.

Lazy evaluation allows us to define and compute with infinite data structures

$\Rightarrow$ highly efficient and elegant implementations

Most programming languages are strict, without built-in support for call-by-need evaluation.

Q: Can be simulate call-by-need evaluation în a strict programming language, e.g., in Racket?

A: Yes, by checking explicitly when a computation is needed, and writing code which to do the needed computation.

Lazy evaluation in Racket

Main idea: Encapsulate the computation that will be needed in the body of a nullary function, and call the function whenever we need to produce some result.

- A nullary function is a function with 0 arguments. Nullary functions act like factories for delayed work.

```
> (define delayedwork (lambda () body))
```

When we wish to perform the computation of *body*, we call the nullary function

```
> (delayedwork)
```

Remark

With this technique, we have full control of the evaluation process:

- We can delay computations and execute them only when really needed.

Stream: a finite representation of an infinite list, where we know how to generate new elements from previous elements.

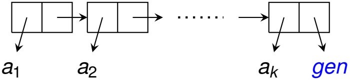
Examples of streams:

- **All ones:** `(1 1 1 ...)`
  Next element is always 1.
- **Natural numbers:** `(0 1 2 3 ...)`
  Next element is successor of previous element.
- **Fibonacci numbers:** `(1 1 2 3 5 8 13 ...)`
  Every element, except first two, is sum of previous two elements.
- **Prime numbers:** `(2 3 5 7 11 13 ...)`
  Every next element is the first natural number different from 1, which is not a multiple of previous elements.

Applications of lazy evaluation

Infinite lists (a.k.a. streams)

Q: How to represent in a finite way a stream $(a_1\ a_2\ a_3\ \ldots)$?
A: As an "incomplete" list



with printed form $'(a_1\ \ldots\ a_k\ .\ gen)$

where *gen* is a nullary function that can generate more elements on demand:

- (gen) computes $(a_{k+1}\ \ldots\ a_{k+\ell}.\ gen')$ with $\ell \geq 1$

- *gen* is called the stream generator.
- A generator is just a function, and function *gen* is recognised with `(procedure?` *gen*`)`

Streams

Examples

```
(define gen-ones
   (lambda () (cons 1 gen-ones)))
; stream of all ones
(define all-ones (gen-ones))

(define (gen-nats n)
      (cons n (lambda () (gen-nats (+ n 1)))))

; stream of all naturals
(define nats (gen-nats 0))

>  all-ones
'(1 . #<procedure:gen-ones>)

> nats
'(0 . #<procedure>)
```

Working with streams

Utility functions: s-take and s-filter

```
; list of first n elements from stream s
(define (s-take n s)
  (cond [(= n 0) '()]
        [(procedure? s) ; s is the stream generator
            (s-take n (s))]
        [#t (cons (car s) (s-take (- n 1) (cdr s)))]))

; stream of all elements of s which satisfy predicate p
(define (s-filter p s)
  (cond [(procedure? s) ; s is the stream generator
        (s-filter p (s))]
        [(p (car s))
            (cons (car s)
                  (lambda () (s-filter p (cdr s))))]
        [#t (s-filter p (cdr s))]))

> (s-take 5 (s-filter even? nats))
'(0 2 4 6 8)
```

`(s-map f s)`

- takes as inputs a stream `s` and a function that computes a value for any element of `s`
- returns the stream obtained by applying function `f` to all elements of `s`

```
(define (s-map f s)
   (if (procedure? s) (s-map f (s))
       (cons (f (car s))
             (lambda () (s-map f (cdr s))))))
```

```
>  (define cubes
     (s-map (lambda (x) (* x x x)) nats))
> (s-take 7 cubes)
'(0 1 8 27 64 125 216)
```

`(s-add s1 s2)`

- takes as inputs two streams of numbers

  `s1 = (`a_1 a_2 `...)`

  `s2 = (`b_1 b_2 `...)`

- returns the stream `(`$a_1 + b_1$ $a_2 + b_2$ `...)`

```
(define (s-add s1 s2)
    (cond [(procedure? s1) (s-add (s1) s2)]
          [(procedure? s2) (s-add s1 (s2))]
          [#t (cons (+ (car s1) (car s2))
                       (lambda ()
                          (s-add (cdr s1) (cdr s2))))]))
```

> ; stream of numbers $n^2 + n$ for all n

```
  (define ns (s-add (s-map (lambda (x) (* x x)) nats)
                    nats))
> (s-take 6 ns)
'(0 2 6 12 20 30)
```

Stream of Fibonacci numbers

Useful observation: the stream `fib` of Fibonacci numbers has the following useful property:

- Adding streams `fib` and `(cdr fib)` yields `(cddr fib)`

$$
\begin{array}{rcllll}
\texttt{fib} & = & f_0 & f_1 & f_2 & \ldots \quad + \\
(\texttt{cdr fib}) & = & f_1 & f_2 & f_3 & \ldots \\
\hline
f_0 \;\; f_1 & & f_2 & f_3 & f_4 & \ldots
\end{array}
$$

- Once we know the first two elements f_0 and f_1, we can start generating the rest of the stream:

```
(define fib
    (cons 1
          (cons 1
                (lambda () (s-add fib (cdr fib))))))

> (s-take 10 fib)
'(1 1 2 3 5 8 13 21 34 55)
```

References

Chapter 13: *Overloading, type classes and type checking*, sections 13.5-13.8 from

Simon Thompson: *Haskell: The Craft of Functional Programming.* Third edition. Pearson Addison Wesley. 2011.