

Types in Programming Languages

Dynamic and Static Typing, Type Inference, Abstract
Data Types, Monads (CTM 2.8.3, 3.7; EPL* 4; GIH**)

Carlos Varela
Rensselaer Polytechnic Institute
September 22, 2026

Partially adapted with permission from:

Seif Haridi
KTH
Peter Van Roy
UCL

(*) Essentials of Programming Languages, 2nd ed., Friedman, Wand, and Haynes, MIT Press 2001

(**) A Gentle Introduction to Haskell, Hudak, Peterson, and Fasel, 1999.

C. Varela

Data types

- A datatype defines a set of values and an associated set of operations
- An abstract datatype is described by a set of operations
- These operations are the only thing that a user of the abstraction can assume
- Examples:
 - Numbers, Tuples, Lists,... (basic data types)
 - Trees, Stacks, Dictionaries,... (user-defined data types)

Types of typing

- Languages can be *weakly typed*
 - Internal representation of types can be manipulated by a program
 - e.g., a string in C is an array of characters ending in ‘\0’.
- *Strongly typed* programming languages can be further subdivided into:
 - *Dynamically typed* languages
 - Variables can be bound to entities of any type, so in general the type is only known at **run-time**, e.g., Oz, SALSA.
 - *Statically typed* languages
 - Variable types are known at **compile-time**, e.g., C++, Java.

Type Checking and Inference

- *Type checking* is the process of ensuring a program is well-typed.
 - One strategy often used is *abstract interpretation*:
 - The principle of getting partial information about the answers from partial information about the inputs
 - Programmer supplies types of variables and type-checker deduces types of other expressions for consistency
- *Type inference* frees programmers from annotating variable types: types are inferred from variable usage, e.g. ML, Haskell.

Example: The identity function

- In a dynamically typed language, e.g., Oz, it is possible to write a generic function, such as the identity combinator:

```
fun {Id X} X end
```

- In a statically typed language, it is necessary to assign types to variables, e.g. in a **statically typed variant of Oz** you would write:

```
fun {Id X:integer}:integer X end
```

These types are checked at compile-time to ensure the function is only passed proper arguments. `{Id 5}` is valid, while `{Id Id}` is not.

Let-bound polymorphism

- Any language using Hindley-Milner type inference, has *let-bound polymorphism*, i.e., identifiers not bound using `let` or `where` (or at top-level of a module) are limited with respect to their polymorphism.
- In particular, a lambda-bound function, i.e., a function value passed as an argument to another function, cannot be instantiated in two different ways.
- For example:

```
let f g = (g 5, g 'c') in f (\x -> x)
```

is illegal: `g` whose principal type is $a \rightarrow a$, is used within `f` in two different ways: once with type $(\text{Num } a) \Rightarrow a \rightarrow a$, and once with type $\text{Char} \rightarrow \text{Char}$.

Example: Improper Operations

- In a dynamically typed language, it is possible to write an improper operation, such as passing a non-list as a parameter, e.g. in Oz:

```
declare fun {ShiftRight L} 0|L end
{Browse {ShiftRight 4}}           % unintended missuse
{Browse {ShiftRight [4]}}        % proper use
```

- In a statically typed language, the same code would produce a type error, e.g. **in a statically typed variant of Oz** you would write:

```
declare fun {ShiftRight L>List}:List 0|L end
{Browse {ShiftRight 4}}           % compiler error!!
{Browse {ShiftRight [4]}}        % proper use
```

Example: Type Inference

- In a statically typed language with type inference (e.g., ML), it is possible to write code without type annotations, e.g. using Oz syntax:

```
declare fun {Increment N} N+1 end
{Browse {Increment [4]}}           % compiler error!!
{Browse {Increment 4}}             % proper use
```

- The type inference system knows the type of ' + ' to be:

<number> X <number> → <number>

Therefore, **Increment** must always receive an argument of type **<number>** and it always returns a value of type **<number>**.

Hindley-Milner Type System

`increment n = n + 1`

- The type inference system knows the type of ' + ' to be:

`<number> → <number> → <number>`

Therefore, **increment** must always receive an argument of type **<number>** and it always returns a value of type **<number>**.

- Every well-typed expression is guaranteed to have a *unique principal type*, i.e., the least general type that contains all instances of the expression.
- The principal type can be inferred automatically.
- In Haskell, `increment`'s principal type is: `(Num a) => a->a`.
 - `a->b`, and even `a` are correct types, but too general, whereas `Int->Int` is too specific.

Static Typing Advantages

- Static typing restricts valid programs (i.e., reduces language's expressiveness) in return for:
 - Improving error-catching ability
 - Efficiency
 - Security
 - Partial program verification

Dynamic Typing Advantages

- Dynamic typing allows all syntactically legal programs to execute, providing for:
 - Faster prototyping (partial, incomplete programs can be tested)
 - Separate compilation---independently written modules can more easily interact--- which enables open software development
 - More expressiveness in language

Combining static and dynamic typing

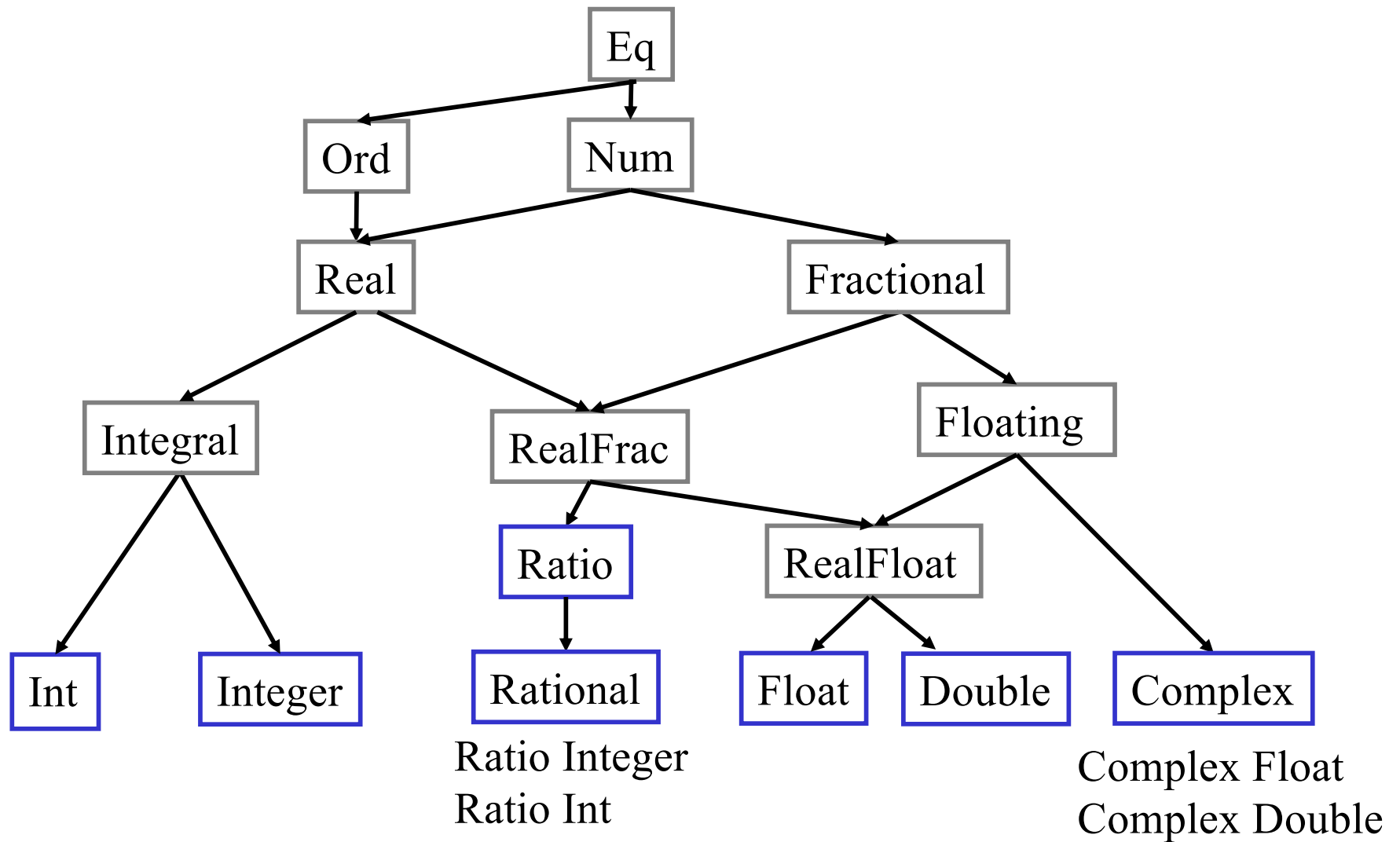
- Programming language designers do not have to make an *all-or-nothing* decision on static vs dynamic typing.
 - e.g, Java has a root **Object** class which enables *polymorphism*
 - A variable declared to be an **Object** can hold an instance of any (non-primitive) class.
 - To enable static type-checking, programmers need to annotate expressions using these variables with *casting* operations, i.e., they instruct the type checker to pretend the type of the variable is different (more specific) than declared.
 - Run-time errors/exceptions can then occur if type conversion (casting) fails.
- Alice (Saarland U.) is a statically-typed variant of Oz.
- SALSA-Lite is a statically-typed variant of SALSA.

Number types in Haskell

Inherited from Common Lisp and Scheme.

- Fixed- and arbitrary-precision integers (Int, Integer).
- Ratios (rational numbers) from each integer type.
- Single- and double-precision real and complex floating-point.

Haskell numeric data types (2)



Stack data type in Haskell

```
data Stack a = Empty | Stack a (Stack a)
```

```
newStack :: Stack a
```

```
newStack = Empty
```

```
push :: Stack a -> a -> Stack a
```

```
push s e = Stack e s
```

```
pop :: Stack a -> (Stack a,a)
```

```
pop (Stack e s) = (s,e)
```

```
isEmpty :: Stack a -> Bool
```

```
isEmpty Empty = True
```

```
isEmpty _ = False
```

Abstract data types

- A datatype is a set of values and an associated set of operations
- A datatype is abstract only if it is completely described by its set of operations regardless of its implementation
- This means that it is possible to change the implementation of the datatype without changing its use
- The datatype is thus described by a set of procedures
- These operations are the only thing that a user of the abstraction can assume

Example: A Stack

- Assume we want to define a new datatype $\langle \text{Stack } T \rangle$ whose elements are of any type T

fun {NewStack}: $\langle \text{Stack } T \rangle$

fun {Push $\langle \text{Stack } T \rangle \langle T \rangle$ }: $\langle \text{Stack } T \rangle$

fun {Pop $\langle \text{Stack } T \rangle$ }: $(\langle \text{Stack } T \rangle, \langle T \rangle)$

fun {IsEmpty $\langle \text{Stack } T \rangle$ }: $\langle \text{Bool} \rangle$

- These operations normally satisfy certain laws:

$\{\text{IsEmpty } \{\text{NewStack}\}\} = \text{true}$

for any E and $S0$, $S1 = \{\text{Push } S0 \ E\}$ and $(S0, E) = \{\text{Pop } S1\}$ hold

$\{\text{Pop } \{\text{NewStack}\}\}$ raises error

Abstract Stack data type

```
data Stack a    -- just the type name
newStack        :: Stack a
push            :: Stack a -> a -> Stack a
pop             :: Stack a -> (Stack a, a)
isEmpty         :: Stack a -> Bool
```

Stack data type module

```
module StackADT (Stack,newStack,push,pop,isEmpty) where
```

```
data Stack a    = Empty | Stack a (Stack a)
```

```
newStack        = Empty
```

```
push s e        = Stack e s
```

```
pop (Stack e s) = (s,e)
```

```
isEmpty Empty   = True
```

```
isEmpty _       = False
```

Stack data type using lists

```
module StackADT2 (Stack,newStack,push,pop,isEmpty) where
```

```
type Stack a = [a] --- type synonym
```

```
newStack      = []
```

```
push s e      = e:s
```

```
pop (e:s)     = (s,e)
```

```
isEmpty []    = True
```

```
isEmpty _     = False
```

Stack abstract data type as a module in Haskell

```
module StackADT (Stack,newStack,push,pop,isEmpty) where
```

```
data Stack a = Empty | Stack a (Stack a)
```

```
newStack = Empty
```

```
...
```

- Modules can then be imported by other modules, e.g.:

```
module Main (main) where
```

```
import StackADT (Stack,newStack,push,pop,isEmpty)
```

```
main = print (push (push newStack 1) 2)
```

Dictionaries

- The datatype dictionary is a finite mapping from a set of <key>s to <value>s
- fun {NewDictionary}
 - returns an empty mapping
- fun {Put D Key Value}
 - returns a dictionary identical to D except Key is mapped to Value
- fun {CondGet D Key Default}
 - returns the value corresponding to Key in D, otherwise returns Default
- fun {Domain D}
 - returns a list of the keys in D

Abstract Dictionary data type

```
data Dictionary k v  -- just the type name
```

```
newDictionary :: Dictionary k v
```

```
put           :: (Ord k) => Dictionary k v -> k -> v -> Dictionary k v
```

```
condGet       :: (Ord k) => Dictionary k v -> k -> v -> v
```

```
domain       :: Dictionary k v -> [k]
```

Implementation

```
type Dictionary k v = [(k,v)]
```

```
newDictionary = []
```

```
put [] key val = [(key,val)]
```

```
put ((k,v):dr) key val | k == key = (key,val):dr  
                        | k > key  = (key,val):(k,v):dr  
                        | k < key  = (k,v):put dr key val
```

```
condGet [] _ defValue = defValue
```

```
condGet ((k,v):dr) key defValue | k == key = v  
                                | k > key  = defValue  
                                | k < key  = condGet dr key defValue
```

```
domain d = map (\(k,_)->k) d
```

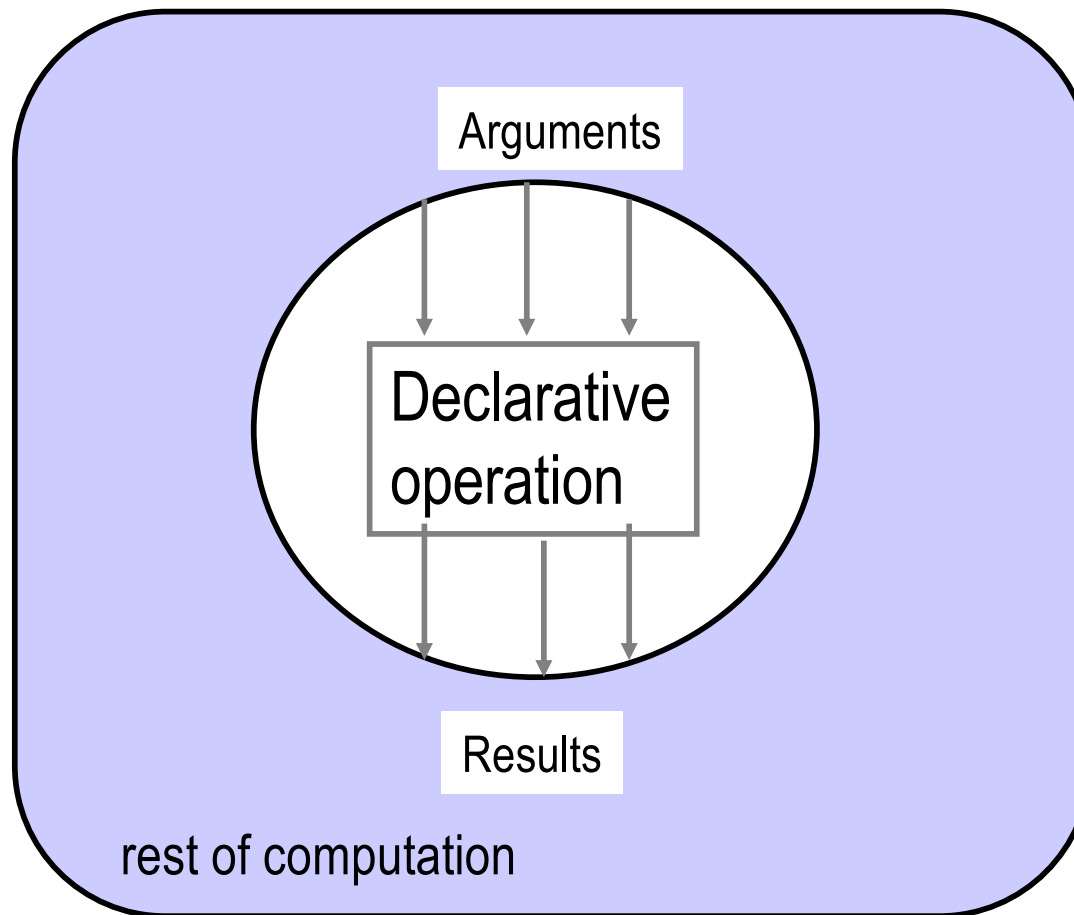
Further implementations

- Because of abstraction, we can replace the dictionary ADT implementation using a list, whose complexity is linear (i.e., $O(n)$), with a binary tree implementation with logarithmic operations (i.e., $O(\log(n))$).
- Data abstraction makes clients of the ADT unaware (other than through perceived efficiency) of the internal implementation of the data type.
- It is important that clients do not use anything about the internal representation of the data type (e.g., using `length` `dictionary` to get the size of the dictionary). Using only the interface (defined ADT operations) ensures that different implementations can be used in the future.

Declarative operations (1)

- An operation is *declarative* if whenever it is called with the same arguments, it returns the same results independent of any other computation state
- A declarative operation is:
 - *Independent* (depends only on its arguments, nothing else)
 - *Stateless* (no internal state is remembered between calls)
 - *Deterministic* (call with same operations always give same results)
- Declarative operations can be composed together to yield other declarative components
 - All basic operations of the Oz declarative model (CTM Chapter 2) are declarative and combining them always gives declarative components

Declarative operations (2)



Monads

- Purely functional programming is **declarative** in nature: whenever a function is called with the same arguments, it returns the same results independent of any other computation state.
- How to model the real world (that may have context dependences, state, nondeterminism) in a purely functional programming language?
 - Context dependences: e.g., does file exist in expected directory?
 - State: e.g., is there money in the bank account?
 - Nondeterminism: e.g., does bank account deposit happen before or after interest accrual?
- Monads to the rescue!

Type Classes in Haskell

- Types in Haskell can be polymorphic, e.g. lists:
 - A list of integers is denoted as being of type [Integer].
 - A list of characters is denoted as being of type [Char].
 - The polymorphic type [a] corresponds to lists of an arbitrary type a.
- Functions can be applicable to polymorphic types, e.g.:
 - Finding an element in a list can take either lists of integers or lists of booleans, or lists of any type a:

`elem x [] = False`

`elem x (y:ys) = (x == y) || elem x ys`

Type Classes in Haskell

```
elem x [] = False
```

```
elem x (y:ys) = (x == y) || (elem x ys)
```

- The type of `elem` is `a->[a]->Bool` for any type `a` that supports equality checking (`==`).
- This is specified in Haskell with a type constraint:

```
elem :: (Eq a) => a->[a]->Bool
```

- All types that support the `==` operation are said to be instances of the type class `Eq`:

```
class Eq a where
```

```
    (==) :: a -> a -> Bool
```

```
    x /= y = not (x == y) -- default method
```

Stack data type is an instance of Eq type class

```
instance Eq a => Eq (Stack a) where
  Empty      == Empty      = True
  (Stack e1 s1) == (Stack e2 s2) = (e1 == e2) && (s1 == s2)
  _          == _          = False
```

Higher order types

- You can think of the polymorphic Stack type as a type constructor that receives a type and produces a new type, e.g.:
 - Stack Integer produces a stack of integers type.

- Consider the Functor higher-order type class:

```
class Functor f where  
    fmap :: (a->b) -> f a -> f b
```

Notice that `f a` applies type (constructor) `f` to type `a`.

- We can declare Stack (not Stack a) to be an instance of the Functor class:

```
instance Functor Stack where  
    fmap f Empty      = Empty  
    fmap f (Stack e s) = Stack (f e) (fmap f s)
```

Functor class laws

- All instances of the Functor class should respect some laws:

`fmap id` `= id`

`fmap (f . g)` `= fmap f . fmap g`

- Polymorphic types can be thought of as containers for values of another type.
- These laws ensure that the container shape (e.g., a list, a stack, or a tree) is unchanged by `fmap` and that the contents are not re-arranged by the mapping operation.
- Functor is a monadic class. Other monadic classes are `Monad`, and `MonadPlus`.

Monad class

- The Monad class defines two basic operations:

```
class Monad m where
```

```
    (>>=)      :: m a -> (a -> m b) -> m b  -- bind
```

```
    return     :: a -> m a
```

```
    fail       :: String -> m a
```

```
    m >> k     = m >>= \_ -> k
```

- The `>>=` infix operation binds two monadic values, while the `return` operation injects a value into the monad (container).
- Example monadic classes are `IO`, lists (`[]`) and `Maybe`.

do syntactic sugar

- In the IO class, $x \gg= y$, performs two actions sequentially (like the Seq combinator in the lambda-calculus) passing the result of the first into the second.

- Chains of monadic operations can use do:

$$\text{do } e1 ; e2 \quad = \quad e1 \gg e2$$
$$\text{do } p \leftarrow e1 ; e2 \quad = \quad e1 \gg= \backslash p \rightarrow e2$$

- Pattern match can fail, so the full translation is:

$$\text{do } p \leftarrow e1 ; e2 \quad = \quad e1 \gg= (\backslash v \rightarrow \text{case of } p \rightarrow e2 \\ \quad \quad \quad _ \rightarrow \text{fail "s"})$$

- Failure in IO monad produces an error, whereas failure in the List monad produces the empty list.

Monad class laws

- All instances of the Monad class should respect the following laws:

<code>return a >>= k</code>	<code>= k a</code>
<code>m >>= return</code>	<code>= m</code>
<code>xs >>= return . f</code>	<code>= fmap f xs</code>
<code>m >>= (\x -> k x >>= h)</code>	<code>= (m >>= k) >>= h</code>

- These laws ensure that we can bind together monadic values with `>>=` and inject values into the monad (container) using `return` in consistent ways.
- The MonadPlus class includes an `mzero` element and an `mplus` operation. For lists, `mzero` is the empty list (`[]`), and the `mplus` operation is list concatenation (`++`).

List comprehensions with monads

```
lc1 = [(x,y) | x <- [1..10], y <- [1..x]]
```

```
lc1' = do x <- [1..10]  
        y <- [1..x]  
        return (x,y)
```

```
lc1'' = [1..10] >>= (\x ->  
                    [1..x] >>= (\y ->  
                                return (x,y)))
```

List comprehensions are implemented using a built-in list monad. Binding (`l >>= f`) applies the function `f` to all the elements of the list `l` and concatenates the results. The `return` function creates a singleton list.

List comprehensions with monads (2)

```
lc3 = [(x,y) | x <- [1..10], y <- [1..x], x+y<= 10]
```

```
lc3' = do x <- [1..10]
```

```
  y <- [1..x]
```

```
  True <- return (x+y<=10)
```

```
  return (x,y)
```

Guards in list
comprehensions assume
that fail in the List monad
returns an empty list.

```
lc3" = [1..10] >>= (\x ->
```

```
  [1..x] >>= (\y ->
```

```
    return (x+y<=10) >>=
```

```
      (\b -> case b of True -> return (x,y); _ -> fail "")))]
```

An instruction counter monad

- We will create an instruction counter using a monad R:

```
data R a = R (Resource -> (a, Resource)) -- the monadic type
```

```
instance Monad R where
```

```
-- (>>=) :: R a -> (a -> R b) -> R b
```

```
R c1 >>= fc2 = R (\r -> let (s,r') = c1 r  
                        R c2 = fc2 s in  
                        c2 r')
```

```
-- return :: a -> R a
```

```
return v    = R (\r -> (v,r))
```

A computation is modeled as a function that takes a resource r and returns a value of type a , and a new resource r' . The resource is implicitly carried state.

An instruction counter monad (2)

- Counting steps:

```
type Resource = Integer -- type synonym
```

```
step      :: a -> R a
```

```
step v    = R (\r -> (v,r+1))
```

```
count     :: R Integer -> (Integer, Resource)
```

```
count (R c) = c 0
```

- Lifting a computation to the monadic space:

```
incR      :: R Integer -> R Integer
```

```
incR n    = do nValue <- n  
             step (nValue+1)
```

```
count (incR (return 5)) -- displays (6,1)
```

An inc computation
(Integer -> Integer) is lifted
to the monadic space:
(R Integer -> R Integer).

An instruction counter monad (3)

- Generic lifting of operations to the R monad:

`lift1 :: (a->b) -> R a -> R b`

`lift1 f n = do nValue <- n
 step (f nValue)`

`lift2 :: (a->b->c) -> R a -> R b -> R c`

`lift2 f n1 n2 = do n1Value <- n1
 n2Value <- n2
 step (f n1Value n2Value)`

`instance Num a => Num (R a) where`

`(+)               = lift2 (+)`

`(-)               = lift2 (-)`

`fromInteger       = return . fromInteger`

With generic lifting
operations, we can define
`incR = lift1 (+1)`

An instruction counter monad (4)

- Lifting conditionals to the R monad:

`ifR :: R Bool -> R a -> R a -> R a`

`ifR b t e = do bVal <- b`

`if bVal then t`

`else e`

`(<=*) :: (Ord a) => R a -> R a -> R Bool`

`(<=*) = lift2 (<=)`

`fib :: R Integer -> R Integer`

`fib n = ifR (n <=* 1) n (fib (n-1) + fib (n-2))`

We can now count the computation steps with:
`count (fib 10) => (55,1889)`

Monads summary

- Monads enable keeping track of imperative features (state) in a way that is modular with purely functional components.
 - For example, fib remains functional, yet the R monad enables us to keep a count of instructions separately.
- Input/output, list comprehensions, and optional values (Maybe class) are built-in monads in Haskell.
- Monads are useful to modularly define semantics of domain-specific languages.

Exercises

31. Compare polymorphic lists in Oz and Haskell. What is the impact of the type system on expressiveness and error-catching ability? Give an example.
32. Why is it important that the representation of an ADT be hidden from its users? Name two mechanisms that can accomplish this representation hiding in Oz and Haskell.
33. Can type inference always deduce the type of an expression? If not, give a counter-example.
34. What is the difference between a type class and a type instance in Haskell. Give an example.
35. Create a monad for stacks that behaves similarly to the List monad in Haskell.