Types in Programming Languages

Dynamic and Static Typing, Type Inference (CTM 2.8.3, EPL* 4)
Abstract Data Types (CTM 3.7)
Monads (GIH** 9)

Carlos Varela

Rensselaer Polytechnic Institute

September 27, 2019

Partially adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

- (*) Essentials of Programming Languages, 2nd ed., by Friedman, Wand, and Haynes, MIT Press
- (**) A Gentle Introduction to Haskell, by Hudak, Peterson, and Fasel, 1999.

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, Records, Lists,… (Oz basic data types)
  - Stacks, Dictionaries,… (user-defined secure 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.

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>`**.

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

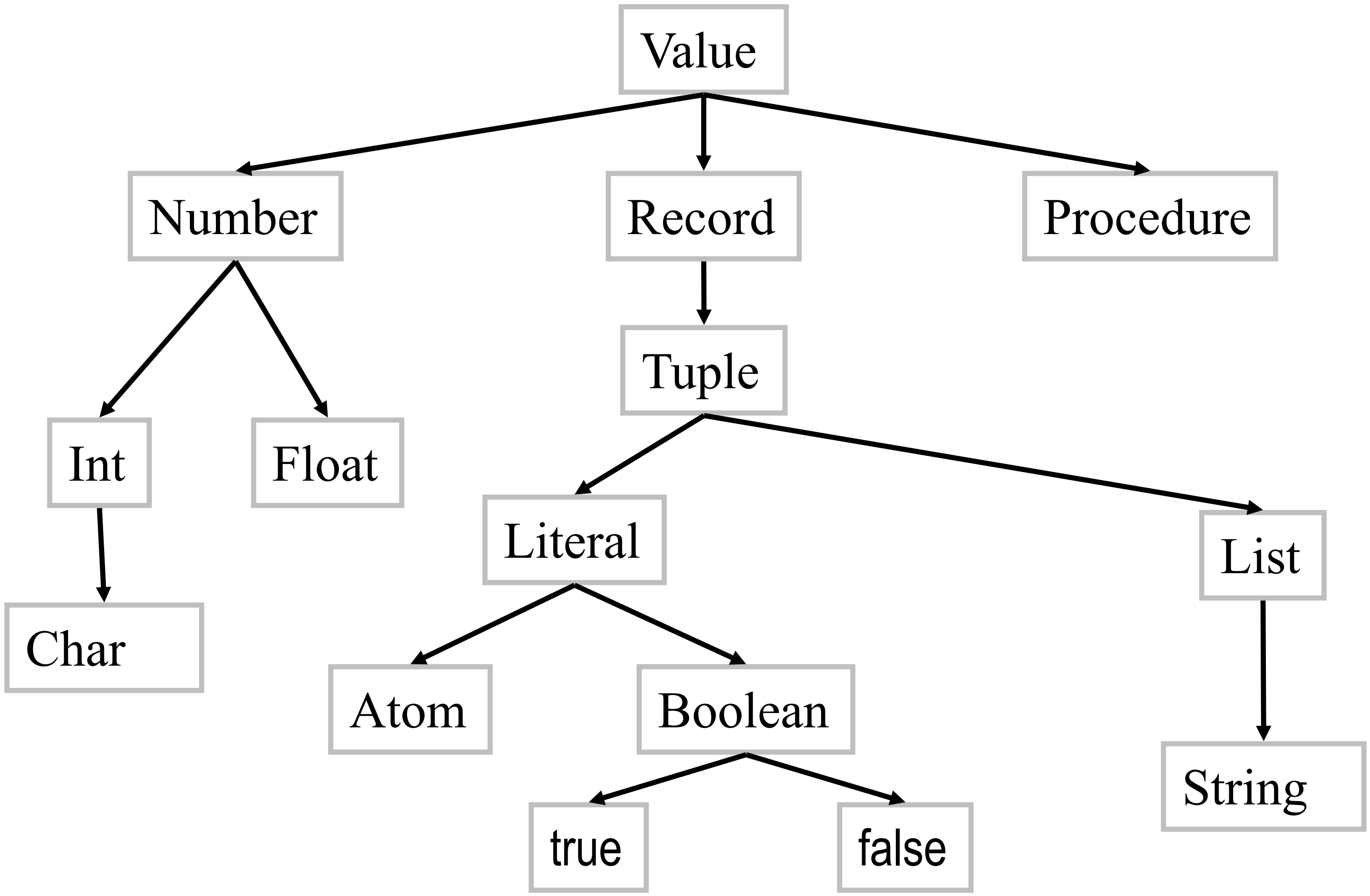
Dynamic Typing Advantages

- 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

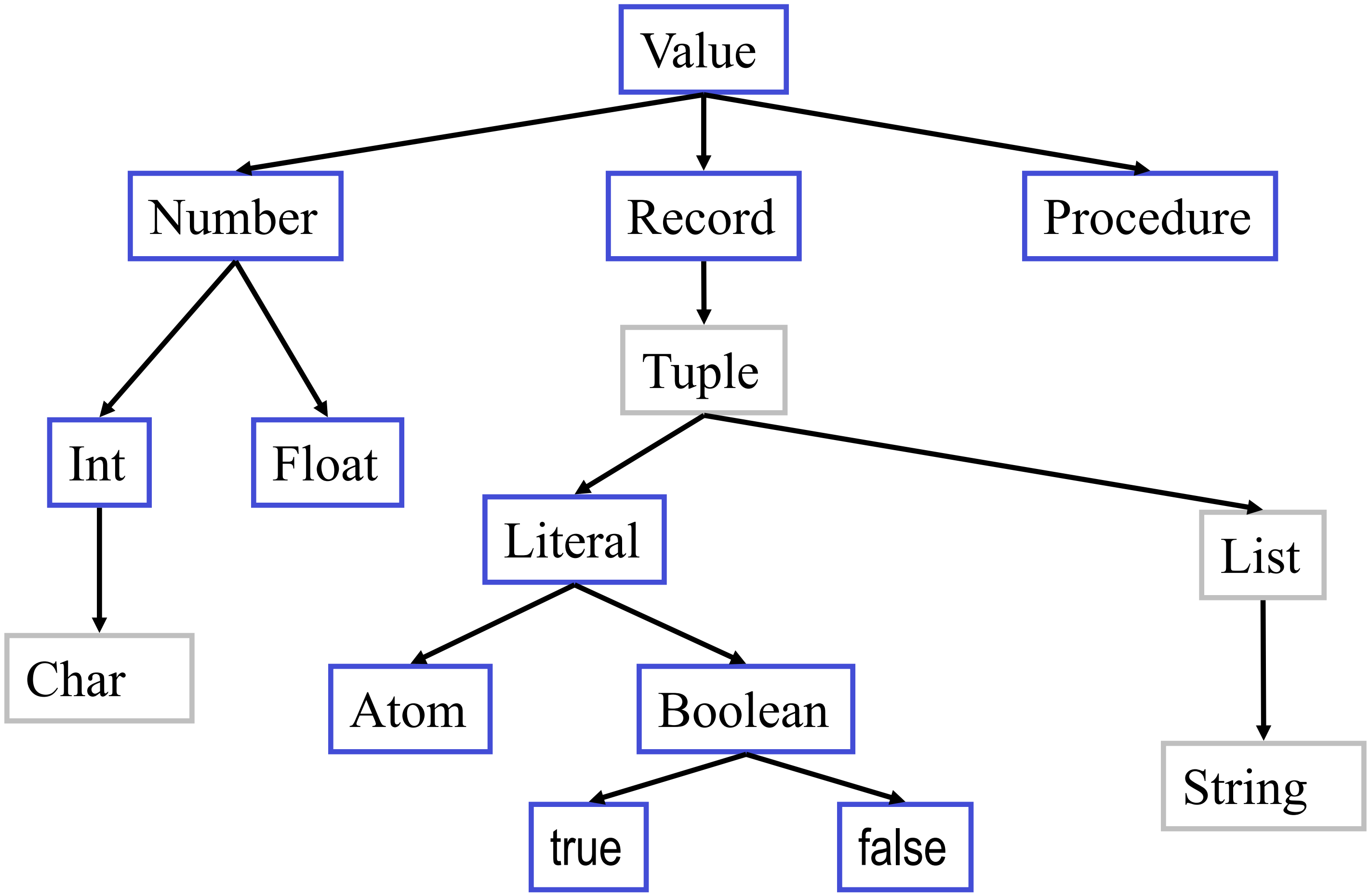
Combining static and dynamic typing

- 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.

Oz data types



Oz data types (2)



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 ⟨stack T⟩ whose elements are of any type T

  fun {NewStack}: ⟨Stack T⟩

  fun {Push ⟨Stack T⟩ ⟨T⟩ }: ⟨Stack T⟩

  fun {Pop ⟨Stack T⟩ ⟨T⟩ }: ⟨Stack T⟩

  fun {IsEmpty ⟨Stack T⟩ }: ⟨Bool⟩

- These operations normally satisfy certain laws:

  {IsEmpty {NewStack}} = true

  for any *E* and *S0*, *S1*={Push *S0 E*} and *S0* ={Pop *S1 E*} hold

  {Pop {NewStack} E} raises error

Stack (implementation)

```
fun {NewStack} nil end
fun {Push S E} E|S end
fun {Pop S E} case S of  X|S1 then E = X  S1 end end
fun {IsEmpty S} S==nil end
```

Stack (another implementation)

```
fun {NewStack} nil end
fun {Push S E} E|S end
fun {Pop S E} case S of  X|S1 then E = X  S1 end end
fun {IsEmpty S} S==nil end
```

```
fun {NewStack} emptyStack end
fun {Push S E} stack(E S) end
fun {Pop S E} case S of stack(X S1) then E = X S1 end end
fun {IsEmpty S} S==emptyStack end
```

Stack data type in Haskell

```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 (Stack _ _) = False
```

Dictionaries

- The datatype dictionary is a finite mapping from a set *T* to ⟨value⟩, where T is either ⟨atom⟩ or ⟨integer⟩
- 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

Implementation

```
fun {Put Ds Key Value}
   case Ds
   of nil then [Key#Value]
   [] (K#V)|Dr andthen Key==K then
      (Key#Value) | Dr
   [] (K#V)|Dr andthen K>Key then
      (Key#Value)|(K#V)|Dr
   [] (K#V)|Dr andthen K<Key then
      (K#V)|{Put Dr Key Value}
   end
end
```

Implementation

```
fun {CondGet Ds Key Default}
  case Ds
  of nil then Default
  [] (K#V)|Dr andthen Key==K then
    V
  [] (K#V)|Dr andthen K>Key then
    Default
  [] (K#V)|Dr andthen K<Key then
    {CondGet Dr Key Default}
  end
end
fun {Domain Ds}
  {Map Ds fun {$ K#_} K end}
end
```

Further implementations

- Because of abstraction, we can replace the dictionary ADT implementation using a list, whose complexity is linear (i.e., O(n)), for 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.

Secure abstract data types: Stack is not secure

```
fun {NewStack} nil end
fun {Push S E} E|S end
fun {Pop S E}
   case S of X|S1 then E=X  S1 end
end
fun {IsEmpty S} S==nil end
```

Secure abstract data types II

- The representation of the stack is visible:

  [a b c d]

- Anyone can use an incorrect representation, i.e., by passing other language entities to the stack operation, causing it to malfunction (like a|b|X or Y=a|b|Y)
- Anyone can write new operations on stacks, thus breaking the abstraction-representation barrier
- How can we guarantee that the representation is invisible?

Secure abstract data types III

- The model can be extended. Here are two ways:
  – By adding a new basic type, an unforgeable constant called a **name**
  – By adding encapsulated state.
- A **name** is like an atom except that it cannot be typed in on a keyboard or printed!
  – The only way to have a name is if one is given it explicitly
- There are just two operations on names:

  N={NewName} : returns a fresh name

  N1==N2 : returns true or false

Secure abstract datatypes IV

- We want to « wrap » and « unwrap » values
- Let us use names to define a wrapper & unwrapper

```
proc {NewWrapper ?Wrap ?Unwrap}
   Key={NewName}
in
   fun {Wrap X}
      fun {$ K} if K==Key then X end end
   end
   fun {Unwrap C}
      {C Key}
   end
end
```

Secure abstract data types: A secure stack

With the wrapper & unwrapper we can build a secure stack

```
local Wrap Unwrap in
    {NewWrapper Wrap Unwrap}
    fun {NewStack} {Wrap nil} end
    fun {Push S E} {Wrap E|{Unwrap S}} end
    fun {Pop S E}
        case {Unwrap S} of X|S1 then E=X  {Wrap S1} end
    end
    fun {IsEmpty S} {Unwrap S}==nil end
end
```

Capabilities and security

- We say a computation is secure if it has well-defined and controllable properties, independent of the existence of other (possibly malicious) entities (either computations or humans) in the system
- What properties must a language have to be secure?
- One way to make a language secure is to base it on capabilities
  - A **capability** is an unforgeable language entity (« ticket ») that gives its owner the right to perform a particular action and only that action
  - In our model, all values are capabilities (records, numbers, procedures, names) since they give the right to perform operations on the values
  - Having a procedure gives the right to call that procedure. Procedures are very general capabilities, since what they do depends on their argument
  - Using names as procedure arguments allows very precise control of rights; for example, it allows us to build secure abstract data types
- Capabilities originated in operating systems research
  - A capability can give a process the right to create a file in some directory

Secure abstract datatypes V

- We add two new concepts to the computation model
- {NewChunk Record}
  - returns a value similar to record but its arity cannot be inspected
  - recall {Arity foo(a:1 b:2)} is [a b]
- {NewName}
  - a function that returns a new symbolic (unforgeable, i.e. cannot be guessed) name
  - foo(a:1 b:2 {NewName}:3) makes impossible to access the third component, if you do not know the arity
- {NewChunk foo(a:1 b:2 {NewName}:3) }
  - Returns what ?

Secure abstract datatypes VI

```
proc {NewWrapper ?Wrap ?Unwrap}
   Key={NewName}
in
   fun {Wrap X}
      {NewChunk foo(Key:X)}
   end
   fun {Unwrap C}
      C.Key
   end
end
```

Secure abstract data types: Another secure stack

With the new wrapper & unwrapper we can build another secure stack (since we only use the interface to wrap and unwrap, the code is identical to the one using higher-order programming)

```
local Wrap Unwrap in
    {NewWrapper Wrap Unwrap}
    fun {NewStack} {Wrap nil} end
    fun {Push S E} {Wrap E|{Unwrap S}} end
    fun {Pop S E}
        case {Unwrap S} of X|S1 then E=X  {Wrap S1} end
    end
    fun {IsEmpty S} {Unwrap S}==nil end
end
```

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.:

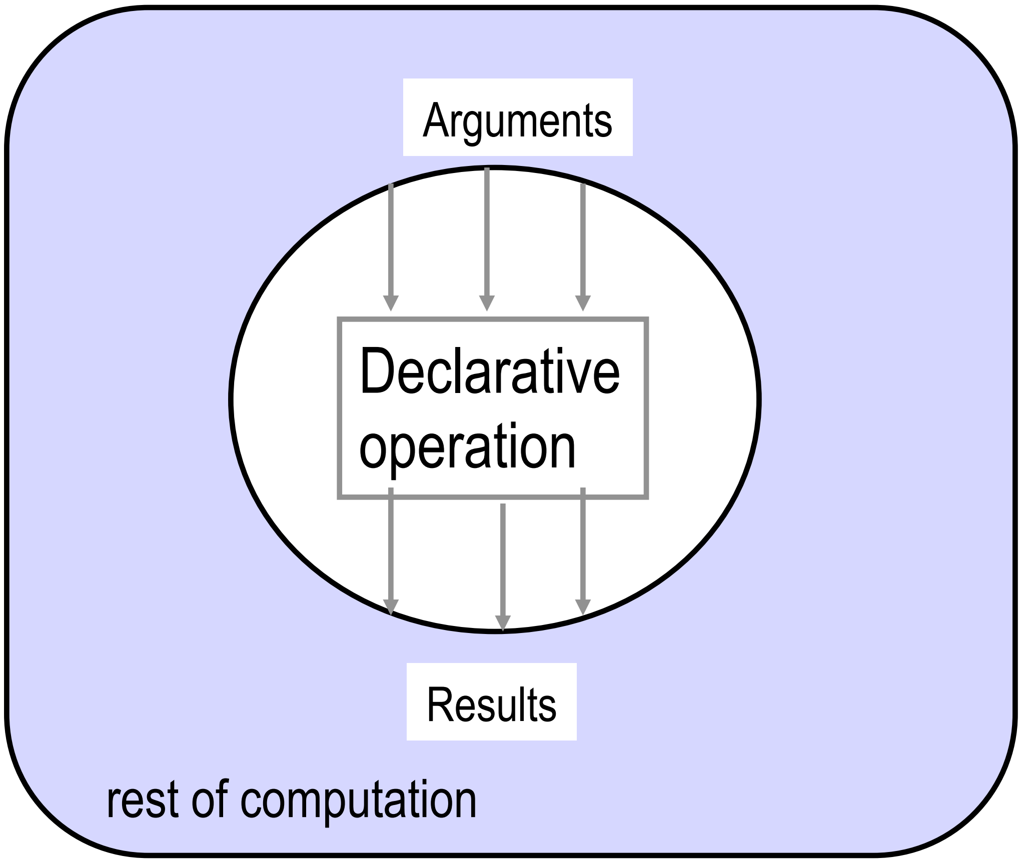
```
module Main (main) where
import StackADT ( Stack, newStack,push,pop,isEmpty )

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

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 declarative model are declarative and combining them always gives declarative components

Declarative operations (2)



Why declarative components (1)

- There are two reasons why they are important:
- *(Programming in the large)* A declarative component can be written, tested, and proved correct independent of other components and of its own past history.
  - The complexity (reasoning complexity) of a program composed of declarative components is the *sum* of the complexity of the components
  - In general the reasoning complexity of programs that are composed of nondeclarative components explodes because of the intimate interaction between components
- *(Programming in the small)* Programs written in the declarative model are much easier to reason about than programs written in more expressive models (e.g., an object-oriented model).
  - Simple algebraic and logical reasoning techniques can be used

Why declarative components (2)

- Since declarative components are mathematical functions, algebraic reasoning is possible i.e. substituting equals for equals
- The declarative model of CTM Chapter 2 guarantees that all programs written are declarative
- Declarative components can be written in models that allow stateful data types, but there is no guarantee

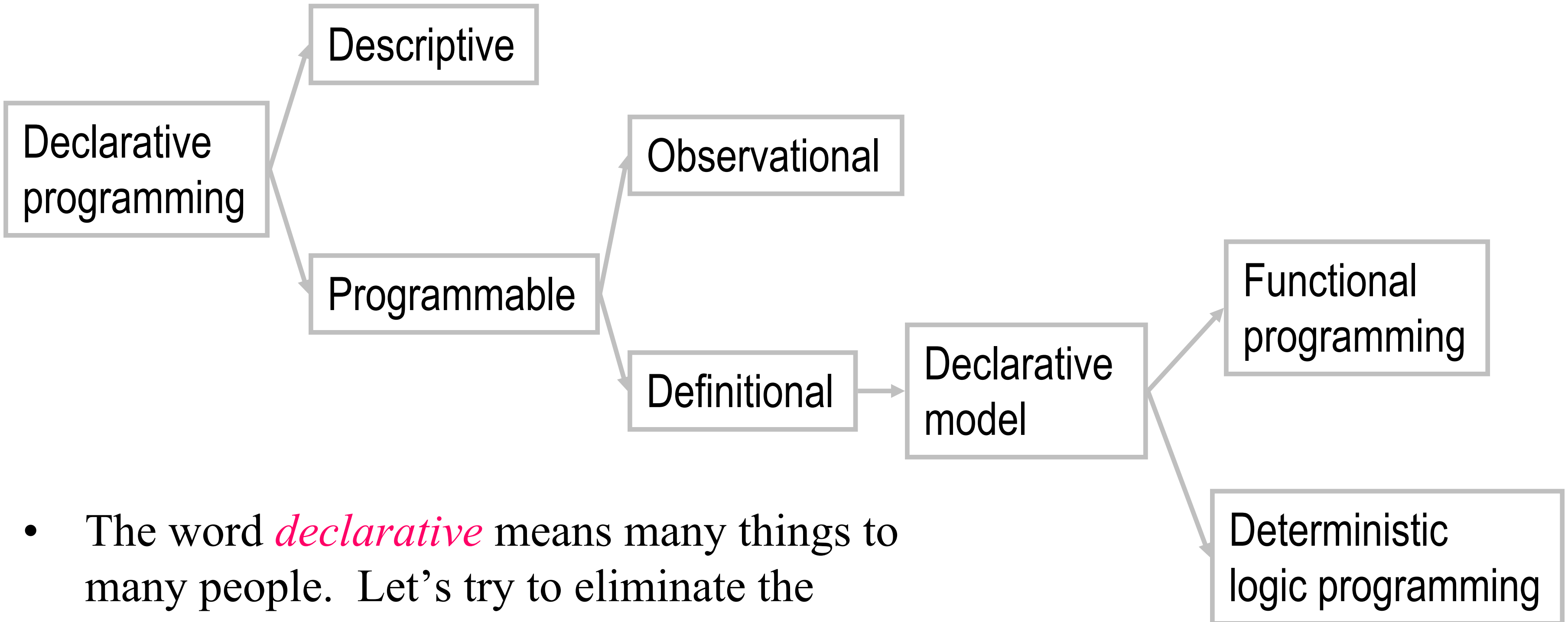
Given

$f(a) = a^2$

We can replace $f(a)$ in any other equation

$b = 7f(a)^2$ becomes $b = 7a^4$

Classification of declarative programming



- The word *declarative* means many things to many people. Let's try to eliminate the confusion.
- The basic intuition is to program by defining the *what* without explaining the *how*

Oz kernel language

The following defines the syntax of a statement, ⟨s⟩ denotes a statement

⟨s⟩	::=	skip	*empty statement*
	|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
	|	⟨x⟩ = ⟨v⟩	*variable-value binding*
	|	$\langle s_1 \rangle$ $\langle s_2 \rangle$	*sequential composition*
	|	local ⟨x⟩ in $\langle s_1 \rangle$ end	*declaration*
	|	proc '{'⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ '}' $\langle s_1 \rangle$ end	*procedure introduction*
	|	if ⟨x⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*conditional*
	|	'{' ⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ '}'	*procedure application*
	|	case ⟨x⟩ of ⟨pattern⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*pattern matching*

Why the Oz KL is declarative

- All basic operations are declarative
- Given the components (sub-statements) are declarative,
  - sequential composition
  - local statement
  - procedure definition
  - procedure call
  - if statement
  - case statement

are all declarative (independent, stateless, deterministic).

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 >>= 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:

```
do e1 ; e2          =       e1 >> e2
do p <- e1; e2      =       e1 >>= \p -> e2
```

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

```
do p <- e1; e2      =       e1 >>= (\v -> case of p -> e2
                                                  _ -> 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:

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

- 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)

lc3'' = [1..10] >>=  (\x ->
            [1..x] >>= (\y ->
                return (x+y<=10) >>=
                  (\b -> case b of True -> return (x,y); _ -> fail “”)))
```

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

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. Write `quicksort` in Oz using list comprehensions.
36. Create a monad for stacks that behaves similarly to the List monad in Haskell.