CS 320: Principles of Programming Languages

Mark P Jones and Andrew Tolmach
Portland State University

Spring 2019

Week 4: Types

language = { syntax / semantics }

syntax: the written/spoken/symbolic/physical form; how things are communicated

language = { syntax / semantics }

semantics: what the syntax means or represents

language = { syntax / semantics }

semantics: what **values** do programs produce?
what **types** can values have?

What is a type?

- A first attempt: types are sets of values:
  - The Boolean type corresponds to the set {False, True}
  - The integer type corresponds to the set {0,1,-1,2,-2,3,-3,…}
  - The character type corresponds to the set {'a', 'b', 'c', 'd', …}
  - The type A ⟶ B corresponds to the set of functions that map values of type A to values of type B …
- Types give us a way to **classify** or organize the values that we work with in our programs
- But does it make sense to think of *any* set of values as a type?
  - Is {128, 192, 256 } a useful type?
  - Is {1, 17, 'a', False} a useful type?

What makes Booleans special?

- What can we do with a Boolean value?
  - AND it with another Boolean value
  - OR it with another Boolean value
  - invert it using NOT
  - use it to make a decision: if b then t else f
- The following also make sense with Boolean values:
  - read or write a value stored in a variable
  - pass a value as an argument to a function
  - return a value as the result of a function

  } But can't we do these things with *any* value?

What makes integers special?

- What can we do with an integer value?
  - add it to or subtract it from another integer
  - multiply or divide it by another integer
  - raise it to the power of another integer
  - compare it with another integer
  - …
- The following also make sense with integer values:
  - read or write a value stored in a variable
  - pass a value as an argument to a function
  - return a value as the result of a function

But can't we do these things with *any* value?

What makes characters special?

- What can we do with an character value?
  - display it on a screen, or write it to a file
  - compare it with another character
  - combine it with other characters in a string
  - calculate a numeric integer code for it
  - …
- The following also make sense with character values:
  - read or write a value stored in a variable
  - pass a value as an argument to a function
  - return a value as the result of a function

But can't we do these things with *any* value?

What makes functions special?

- What can we do with a function value of type A ⟶ B?
  - apply it to an argument
  - compose it with a function of type B ⟶ C
  - compose it with a function of type D ⟶ A
  - …
- The following also make sense with function values:
  - read or write a value stored in a variable
  - pass a value as an argument to a function
  - return a value as the result of a function

But, in many languages, we can't do these things with functions!

Lessons learned

- Types are **sets of values together with specific operations**
  - the operations that the values have in common are what distinguishes a type from an arbitrary set of values
  - this distinction is subjective: it might not always make sense to consider {128, 192, 256 } as a type … but it would make good sense if you're working with AES encryption
- A lot of the time, we work with"first-class values":
  - they can be read from or written to a variable
  - they can be passed as an argument to a function
  - they can be returned as the result of a function
- But, apparently, not all values are first class …

Type system building blocks

Primitive types

- Primitive types cannot be broken down into smaller constituents
- A programming language will typically provide
  - literal syntax for writing/constructing values of primitive types
  - built-in operators and/or statement forms for manipulating values of primitive types
- Primitive types often (but not always) reflect basic machine-level types such as integers, floating point numbers, characters, etc...
  - The numeric types are superficially familiar from mathematics, but may have limited range or precision

Constructed types

Common methods for constructing more interesting types include:

- products (aka tuples, records, structs, arrays)
- sums (aka variants, unions)
- functions
- recursion
- parameterization
- polymorphism

Products

- A **product type** allows us to package multiple values up as a single, composite (or aggregate) value
- Different programming languages support different notions of products, trading off such features as:
  - Access methods: named, numbered, positional, ...
  - Accuracy of typing: heterogeneous vs. homogeneous (e.g., type `list` in Python vs `T[]` in Java)
  - ...

Records/structs/classes

- A **record** holds a collection of values, accessed by name
- Different fields can have different types (heterogeneous)
- Constant time access to each field

```
class Person {
   String name;
   Date   dob;
   long   ssn;
}
```

For example:

Grace	12/09/1906	876543210

```
print("hello" + user.name);
```

(Java)

Tuples

- A **tuple** holds a collection of values, accessed by position
- The type T x S contains pairs (t, s) where t∈T and s∈S
- Lightweight syntax for constructing tuples, pattern matching or numbered selectors for taking them apart
- Different fields can have different types of value
- Constant time access to each field

For example:

```
type Person = (String, Date, SSN)
type Date   = (Month, Day, Year)
grace       = ("Grace", (Dec, 9, 1906), 876543210)

greet (name, dob, ssn) = "hello " ++ name
```

(Haskell)

Arrays

- An **array** stores a collection of values, accessed by position/index
- Typically, all values have the same type
- Constant time access to each value

```
int[] arr;
a[0] = a[m] + a[n];
```

(Java)

For example:

31	28	31	30	31	30	31	31	30	31	30	31

Dictionaries/generalized arrays

- Dicts in Python, for example, are not restricted to integer indices and can store different types of value in each position.

```
grace = {'name':'Grace',
         'dob': (12,09,1906),
         'ssn': 876543210 }
```

- We can access the values defined in `grace` by using array-like notation with run-time computed keys

```
key = 'name'
print(grace[key])
```

- Still roughly constant time access (using hash tables)
- Python's dictionaries are heterogeneous (different keys can be associated to different types of values)

Sums/variants

- **Variants** or **sum types** allow us to capture alternatives
- The sum type (T + S) contains all the values of type T and all the values of type S, with some way to distinguish between them
- Example: in Haskell, the type `Either A B` is a sum type that contains all the values of the form `Left x` (if `x∈A`) and all the values of the form `Right y` (if `y∈B`)
- The type `Either Bool Char` contains:

  `Left True`, `Right 'a'`, `Left False`, `Right 'z'`, ...
- We sometimes refer to `Left` and `Right` here as **tags**

Unions and enumerations

- **Unions** in C provide space to store exactly one of each listed component, without a tag:

```
union intOrString { int num; char* str; }
```

- **Enumerations** are variants in which the only data is a tag:

```
enum tag { NUM, STR };
```

- We can combine these constructs to make our own sum:

```
struct tagged { enum tag t;
                union intOrString data; }
```

- But nothing ensures that the `t` and the `data` components here will always be consistent (to do that, we would have to hide the structure of `tagged` using private state)

Inheritance and variants

- Variants are often coded in an OOP language using inheritance:

```
abstract class IntOrString { ... }
class AnInteger extends IntOrString {
    int n; ...
}
class AString extends IntOrString {
    String str; ...
}
```

- Typically requires elements of dynamic typing, even in a statically typed language

  run time type tests

  type casting/conversion operators (that could fail at run time)

Algebraic datatypes

- A unified approach for defining products and sums (and "sum of products" combinations):

Products:	`data DateOfBirth = DOB Day Month Year`
Sums:	`data Month = Jan|Feb|Mar|Apr|...|Dec`
Combined:	`data Opt = Missing | Entered String`

- Found in many languages (e.g., Haskell, ML, Rust, ...)
- Run time checks are replaced by pattern matching constructs with full static typing

```
greet Missing     = "we haven't been introduced"
greet (Entered n) = "hello " ++ n
```

(Haskell)

Recursion

- Many interesting types can be defined once we allow recursive definitions
- For example, a list of integers is either:

  Empty; or else

  Nonempty, with an integer at its head and a (shorter) list of numbers as its tail.
- In the notations we have seen so far:

```
data IntList = Empty | NonEmpty Int IntList   // Haskell
struct IntList { int head; struct IntList* tail; }; // C
abstract class IntList {}                      // Java
class Empty    extends IntList {}
class NonEmpty extends IntList { int head; IntList tail; }
```


Functions (exponentials)

- Conceptually, a function of type (A⟶B) that takes an argument of type A and returns a result of type B is like an array in which:
  - Components are all indexed by values of type A
  - All components store values of the same type B
  - Values are computed on demand rather than stored
- Functions can be used to build data structures:

```
type IntSet = Int -> Bool

intersect     :: IntSet -> IntSet -> IntSet
intersect s t  = \n -> s n && t n
```

Lambda expressions

Lambda expressions (anonymous functions) provide a way to write functions without giving them a name.

x ⟶ **λx.E** ⟶ E

Haskell	`\x -> x + 1`
LISP	`(lambda (x) (+ x 1))`
Python	`lambda x: x + 1`

Java	`(x) -> x + 1`
C++ 11	`[] (int x) -> int { return x + 1; }`
Scala	`x => x + 1`

Lambda calculus (Alonzo Church, 1930s)

- A simple language for describing calculations with functions:

E	⟶	x	variables
E	⟶	c	constants
E	⟶	E E	function application
E	⟶	λx . E	lambda abstraction
E	⟶	(E)	parentheses

- Examples:
  - The identity function (λx.x)
  - The successor function (λx.x+1)
  - The doubling function (λx.x*2)
  - Function composition (λf.λg.λx.f(g x))
  - ("x+1" is "syntactic sugar" for "plus x 1", where plus is a constant representing the function that adds two numbers)

Substitution in the lambda calculus

- We write [E'/x] E for the expression that is obtained by replacing all occurrences of x in E with E'.
- This operation is called **substitution**:
  "[E'/x]E is obtained by substituting E' for x in E"
- Examples:
  - [3/x] (2 * x) = (2 * 3)
  - [3/x] (x + x) = (3 + 3)
  - [3/x] (y + y) = (y + y)
  - [x+1/x] (2 * x) = (2 * (x+1))
  - [1/x] (λx.x) = (λx.x)
  - [y/x] (λy.x) = [y/x] (λz.x) = (λz.y)

"Alpha conversion": renaming of bound variables (λx. E) = (λy. [y/x]E)

- Surprisingly tricky to define precisely, but very widely used!

Evaluation in the lambda calculus

- The fundamental rule:
  - "beta conversion" ("function application"):
    (λx. E) E' = [E'/x] E
- Examples (of beta reduction):

(λf.λg.λx.f(g x)) (λx.x+1) (λx.2*x) n
= (λf.λg.λx.f(g x)) (λy.y+1) (λz.2*z) n
= (λg.λx. (λy.y+1) (g x)) (λz.2*z) n
= (λg.λx. g x+1) (λz.2*z) n
= (λx. (λz.2*z) x+1) n
= (λx. (2*x)+1) n
= ((2*n)+1)

renaming of bound variables

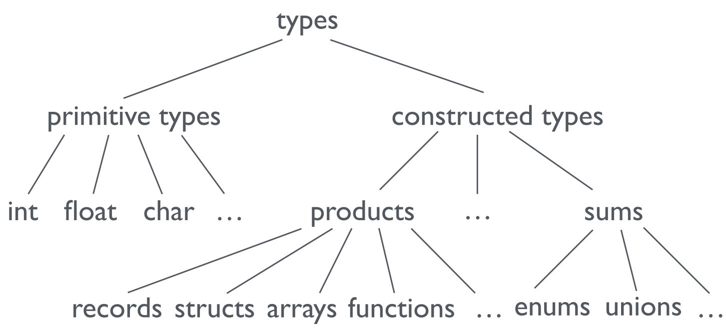
What makes lambda calculus interesting?

- Theory:
  - A formal system in mathematical logic
  - A universal model of computation (can simulate any single taped Turing machine, for example)
  - Many interesting datatypes (natural numbers, tuples, sums, lists, trees, …) can be encoded using lambda expressions
  - Many varieties of lambda calculus are used and studied in advanced programming language research and design
- Practice
  - A major influence on programming language design, from Lisp, to Haskell, to Javascript, and beyond
  - A foundation for functional programming languages

Side-effects and purity

- In many languages, a function may have **side-effects** that are not reflected in its type, such as:
  - Reading or writing memory outside the function's variables
  - Performing I/O operations
  - Throwing exceptions
- Sometimes such functions are executed only for their side-effects, and don't return an interesting type at all
  - Sometimes these are called "**procedures**"
- In **pure** functional languages such as Haskell, function side-effects are forbidden
  - These functions are much closer to the mathematical idea of a (partial) function

A (partial) taxonomy of types



Why "products", "sums", "exponentials"?

If we work with **finite** types A and B, and write |T| for the number of different elements of type T ...

Then:

- $|A \times B| = |A| \times |B|$ (pairs (a,b) where $a \in A$, $b \in B$)
- $|A + B| = |A| + |B|$ (either an $a \in A$ or a $b \in B$)
- $|A \longrightarrow B| = |B|^{|A|}$

  (tuples $(f(a_1), f(a_2), \ldots, f(a_{|A|}))$ with $a_i \in A$, $f(a_i) \in B$)

Polymorphism and parameterized types

Parameterization and polymorphism

- Defining a type of lists of integers is easy, as is defining a function to reverse a list of integers
- But what if we also need to define and work with lists of other types?
- In general, when can we write code that works over more than one type?
- This is where parameterized types and polymorphism come in to the picture …

Some examples

- Some operations work on only **one** type of value:
  - e_1 && e_2 only makes sense if e_1 and e_2 are booleans
- Some operations work on **any** type of value:
  - a[i] makes sense for any array a of type T[] (if i is an int)
- Some operations work only on **some** types of value:
  - $e_1 + e_2$ makes sense if e_1 and e_2 are integers, but not if they are characters
- Some operation names make sense for multiple types of value, but may mean something **different** for each type:
  - $e_1 + e_2$ may also make sense if e_1 and e_2 are strings, but meaning concatenation rather than addition

Monomorphism and Polymorphism

- A monomorphic operator works only on one type of argument (e.g., the && operator)
- A polymorphic operator works on more than one type of argument.
  - Parametric polymorphism: essentially the same implementation/code/algorithm is used for all types of argument (e.g., array indexing)
  - Ad-hoc polymorphism: different implementations are used for different types of value (e.g., numeric operations, comparisons, etc...)
  - Sub-type polymorphism: an operator on a given type also accepts values of a sub-type

Polymorphism

- According to my dictionary:

  polymorphism [poli *mawr* fizm] n occurrence of several types of individual organism within one species

- From the Greek:

  polymorphism = "many shapes"

- In programming languages:

  a single value/function/object can be used at many different types

Enforced monomorphism

- Suppose that `A` and `B` are Haskell types, and consider the following function:

```
swap        :: (A, B) -> (B, A)
swap (x, y) = (y, x)
```

- This definition will work for any choices of `A` and `B`
- The definition uses only polymorphic constructs
- But this function can only be used with pairs of the specific type `(A, B)`

Lifting the restriction

- Let's replace the fixed types `A` and `B` with **type variables** `a` and `b`, indicating that any types can be used:

```
swap        :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

- `x` and `y` are parameters: any values will do ...
- `a` and `b` are parameters: any types will do ...

Using a polymorphic function

- We can apply our polymorphic function to pairs of any type:

```
swap (True, False) ==> (False, True)

swap (1, 2)        ==> (2, 1)

swap ('h', 'i')    ==> ('i', 'h')
```

- The type of the components can be determined automatically by a process of **type inference**

Parameterized types

Polymorphic functions start to become really useful in a language that has parameterized types:

- Pairs of `A`s and `B`s
- Arrays of `A`s
- Lists of `A`s
- Trees of `A`s
- Hashtables of `A`s
- Stacks of `A`s
- ...

Linked lists

- Linked lists are a useful data structure:

```
data List = Nil
          | Cons Value List
```

- What's special about `Value` here?
- Nothing! Any type would do ...

A profusion of linked lists

- But we often need lots of different list types in a program:

```
data ListS = NilS
           | ConsS String ListS

data ListI = NilI
           | ConsI Int ListI

data ListB = NilB
           | ConsB Bool ListB
```

- It is irritating to repeat this "boilerplate" over and over again
- And it makes the program harder to maintain too

A profusion of functions too!

- We often need similar functions for each list type too

```
lengthS                :: ListS -> Int
lengthS NilS            = 0
lengthS (ConsS x xs)    = 1 + lengthS xs

lengthI                :: ListI -> Int
lengthI NilI            = 0
lengthI (ConsI x xs)    = 1 + lengthI xs

lengthB                :: ListB -> Int
lengthB NilB            = 0
lengthB (ConsB x xs)    = 1 + lengthB xs
```

Essentially the same code in each definition!

Parameterized types and polymorphism

- Linked lists are a useful data structure:

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

- This is a parameterized datatype: `a` is a "type variable"
- Polymorphic functions work uniformly over many types:

```
length             :: List a -> Int
length Nil          = 0
length (Cons x xs)  = 1 + length xs
```

- One concept, one definition, many instances!
- Less code to write, less code to understand, less code to maintain

Parameterized types in Haskell

- The names of Haskell types all begin with a capital letter
- Identifiers in Haskell types that begin with a lower case letter are type variables
- Examples:

```
data Maybe a    = Nothing | Just a
data Either a b = Left a | Right b
data Prod a b   = Pair a b
data List a     = Nil | Cons a (List a)
data BST a      = Leaf | Fork (BST a) a (BST a)
data RoseTree a = Node a (List (RoseTree a))
```


Examples

```
getName          :: Maybe String -> String
getName (Just n) = n
getName Nothing  = "not named"

nums  :: List Int
nums   = Cons 1 (Cons 2 (Cons 3 Nil))

bools :: List Bool
bools  = Cons True (Cons False Nil)

find                          :: Int -> BST Int -> Bool
find n Leaf                    = False
find n (Fork l m r) | n==m  = True
                    | n < m = find n l
                    | n > m = find n r
```


Polymorphic operations in Haskell

```
fst              :: Prod a b -> a
fst (Pair x y)    = x

snd              :: Prod a b -> b
snd (Pair x y)    = y

length           :: List a -> Int
length Nil        = 0
length (Cons x xs) = 1 + length xs

find                          :: Ord a => a -> BST a -> Bool
find n Leaf                    = False
find n (Fork l m r) | n==m  = True
                    | n < m = find n l
                    | n > m = find n r
```

works for any a, b

works for any a, b

Ord a => ... is Haskell notation for "any type a for which an ordering has been defined"

A Haskell "**type class**" supports ad hoc polymorphism … but the details are beyond the scope for this class

Aside: Syntactic sugar

- **Syntactic sugar** is used to make the notation of a programming language "sweeter" (i.e., easier to read and write) without adding new expressive power
- Examples:
  - The `List a` and `Prod a b` types are actually written as `[a]`, and `(a, b)`, respectively.
  - The expressions `Nil`, `Cons x xs`, and `Pair p q`, are actually written as `[]`, `(x:xs)` and `(p, q)`, respectively.
- Figuring out where it is appropriate to use syntactic sugar is part of the "art" of good language design

Summary

- There are many standard building blocks in programming language type systems
- Support for first-class functions, originating in functional languages, is now becoming popular in other paradigms too
- Polymorphic type systems, in combination with parameterized types, can increase flexibility and reuse without compromising on type safety.