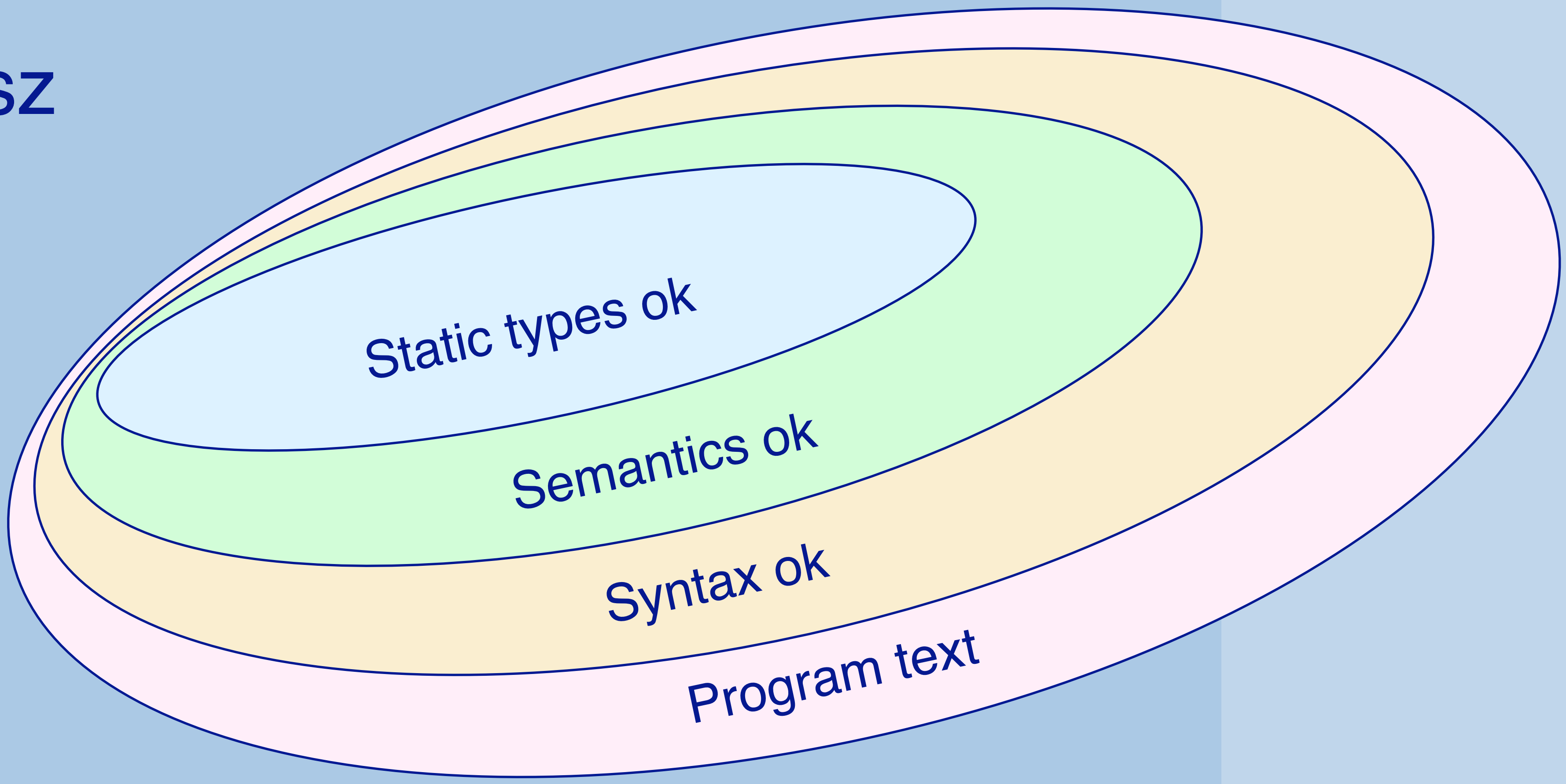
4. Types and Polymorphism

Oscar Nierstrasz



UNIVERSITÄT BERN

Roadmap

> Static and Dynamic Types
> Type Completeness
> Types in Haskell
> Monomorphic and Polymorphic types
> Hindley-Milner Type Inference
> Overloading

References

> Paul Hudak, *"Conception, Evolution, and Application of Functional Programming Languages,"* ACM Computing Surveys 21/3, Sept. 1989, pp 359-411.

> L. Cardelli and P. Wegner, *"On Understanding Types, Data Abstraction, and Polymorphism,"* ACM Computing Surveys, 17/4, Dec. 1985, pp. 471-522.

> D. Watt, *Programming Language Concepts and Paradigms*, Prentice Hall, 1990

Roadmap

> **Static and Dynamic Types**
> Type Completeness
> Types in Haskell
> Monomorphic and Polymorphic types
> Hindley-Milner Type Inference
> Overloading

What is a Type?

Type errors:

```
? 5 + [ ]
ERROR: Type error in application
*** expression : 5 + [ ]
*** term : 5
*** type : Int
*** does not match : [a]
```

A type is a set of values?

```
> int = { ... -2, -1, 0, 1, 2, 3, ... }
> bool = { True, False }
> Point = { [x=0,y=0], [x=1,y=0], [x=0,y=1] ... }
```

The notion of a type as a set of values is very natural and intuitive: Integers are a set of values; the Java type `JButton` corresponds to all possible instance of the `JButton` class (or any of its possible subclasses).

What is a Type?

A type is a partial specification of behaviour?

> `n,m:int` $\Rightarrow$ `n+m` is valid, but `not(n)` is an error

> `n:int` $\Rightarrow$ `n := 1` is valid, but `n := “hello world”` is an error

What kinds of specifications are interesting? Useful?

A Java *interface* is a simple example of a partial specification of behaviour. Any object that conforms to a given interface can be used where that interface is expected. This is arguably more useful than the notion of a type as a set of values, because *we may not care about the specific internal representation* of an object but just *what it can do for us*.

Static and Dynamic Types

Values have static types defined by the programming language. A variable may have a declared, static type.

Variables and expressions have dynamic types determined by the values they assume at run time.

declared, static type is `Applet`

static type of value is `GameApplet`

```
Applet myApplet = new GameApplet();
```

actual dynamic type is `GameApplet`

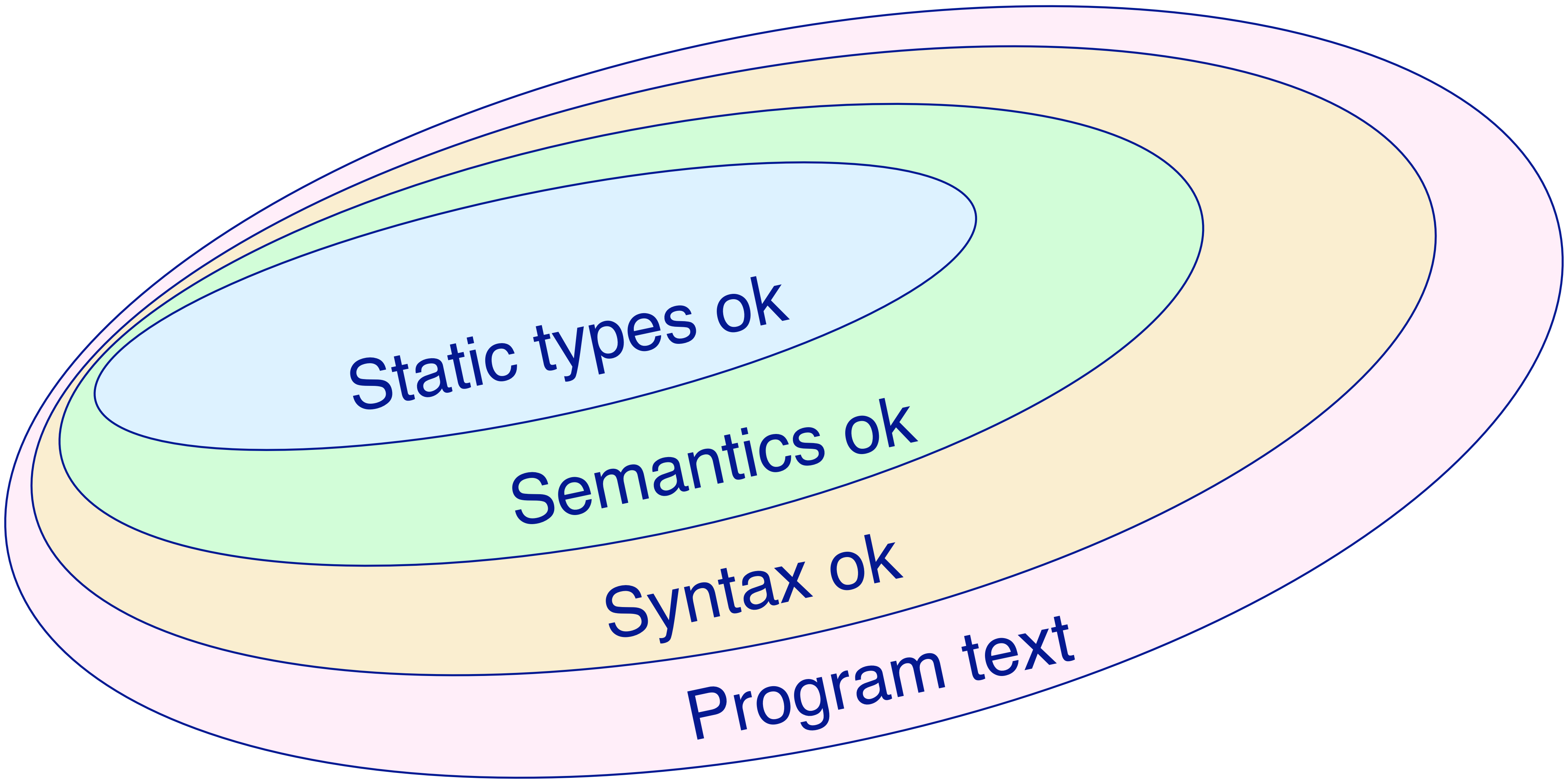
The terms “static” and “dynamic” can be very confusing, especially given their use as keywords in languages like C++ and Java.

“Static” simply means: “based on program text alone.” *Static typing* (for example) means types are checked based on the source code, not by executing the program. *Static analysis* more generally means “analysis based on source code alone.”

“Dynamic” means: “determined at run time.” So, *dynamic analysis* means “analysis based on run-time behaviour.”

Aside: “*run time*” (noun) = execution time; “*run-time*” (adjective) = something happening at run time; “*runtime*” (noun) = run-time language support, e.g., in the virtual machine.

Static types restrict the programs you may write!




```
Object wyatt = new Cowboy();
wyatt.draw();
```

A static type system (typically) forbids you from running programs that the type system cannot validate.

This Java code will not run without the explicit downcast `(int)`, even though the downcast does nothing.

```
List<Object> myList = new ArrayList<Object>();
myList.add(10);
return 2 + (int) myList.get(1);
```

Static and Dynamic Typing

A language is <u>statically typed</u> if it is always possible to *determine the (static) type* of an expression *based on the program text alone.*

A language is <u>dynamically typed</u> if *only values have fixed type*. Variables and parameters may take on different types at run-time, and must be checked immediately before they are used.

A language is "strongly typed" if it is impossible to perform an operation on the wrong kind of object.

Type consistency may be assured by

I. compile-time type-checking,
II. type inference, or
III. dynamic type-checking.

The term “strongly typed” is not very meaningful, just as the term “untyped” is misleading. Actually *all* programming languages have a notion of type; they just handle types in very different ways.

The more useful distinction whether a language is statically-typed, like Java or C++, or dynamically-typed, like Smalltalk or Ruby.

Haskell is interesting in that it is statically-typed, but does not require explicit type annotations.

Strong, weak, static, dynamic

	Static	***Dynamic***
"Strong"	Java, Pascal	Smalltalk, Ruby
"Weak"	C	Assembler

Kinds of Types

All programming languages provide some set of built-in types.

> ***Primitive types:*** booleans, integers, floats, chars ...
> ***Composite types:*** functions, lists, tuples ...

Most statically-typed modern languages provide for additional user-defined types.

> ***User-defined types:*** enumerations, recursive types, generic types, objects ...

Roadmap

> Static and Dynamic Types
> **Type Completeness**
> Types in Haskell
> Monomorphic and Polymorphic types
> Hindley-Milner Type Inference
> Overloading

Type Completeness

The Type Completeness Principle:

> *No operation should be arbitrarily restricted in the types of values involved.*
>
> *— Watt*

First-class values can be *evaluated, passed as arguments and used as components of composite values.*

Functional languages attempt to make no class distinctions, whereas imperative languages typically treat functions (at best) as second-class values.

Pretty much all programming languages limit the kinds of entities that may be pass as values (and therefore have a meaningful type). In C or C++, functions are not values, though pointers to functions are. Classes are not values.

In Java, methods and classes are not values, though you can obtain a reified object representing a class as a value, and in Java 8, you can pass method references as values. Packages are not values, however.

In Haskell, functions are first-class values, so can be passed as arguments and returned as values. Since Haskell is statically-typed, the type system is capable of expressing function types.

Roadmap

> Static and Dynamic Types
> Type Completeness
> **Types in Haskell**
> Monomorphic and Polymorphic types
> Hindley-Milner Type Inference
> Overloading

Function Types

Function types allow one to deduce the types of expressions without the need to evaluate them:

```
fact :: Int -> Int
42 :: Int                 ⇒      fact 42 :: Int
```

Curried types:

```
Int -> Int -> Int         ≡      Int -> (Int -> Int)
```

and

```
plus 5 6                  ≡      ((plus 5) 6)
```

so:

```
plus::Int->Int->Int       ⇒      plus 5::Int->Int
```

All expressions are typed. If you know the types of the individual sub-expressions, you can compute the type of the expression as a whole.

In Haskell, you can query the type of a value with the operator `:t`

```
Prelude> :t 'a'
'a' :: Char
Prelude> :t 1
1 :: Num a => a
```

Note that numbers have a “funny” type: “Num a => a” means “if the type a is a kind of number, then the type is a”. Since Int and Float are both kinds of number, we can be more specific:

```
Prelude> :t 1 :: Int
1 :: Int :: Int
Prelude> :t 1 :: Float
1 :: Float :: Float
```

List Types

A list of values of type `a` has the type `[a]`:

```
[ 1 ] :: [ Int ]
```

NB: All of the elements in a list must be of the same type!

```
['a', 2, False]   -- illegal! can't be typed!
```

Tuple Types

If the expressions `x1, x2, ..., xn` have types `t1, t2, ..., tn` respectively, then the tuple `(x1, x2, ..., xn)` has the type `(t1, t2, ..., tn)`:

```
(1, [2], 3) :: (Int, [Int], Int)
('a', False) :: (Char, Bool)
((1,2),(3,4)) :: ((Int, Int), (Int, Int))
```

The unit type is written `()` and has a single element which is also written as `()`.

User Data Types

New data types can be introduced by specifying

I. a datatype name,

II. a set of parameter types, and

III. a set of constructors for elements of the type:

```
data DatatypeName a1 ... an = constr1 | ... | constrm
```

where the constructors may be either:

1. Named constructors:

```
Name type1 ... typek
```

2. Binary constructors (i.e., anything starting with “:”):

```
type1 BINOP type2
```

Data types are somehow comparable to classes in object-oriented languages, yet also very different. A *constructor* is used to create a value of an abstract data type, but each value must be *deconstructed* before it can be used.

Deconstruction is simply pattern matching using the constructor names: if a value matches a constructor, then that gives you access to the arguments used with that constructor to create that value in the first place.

Enumeration types

User data types that do not hold any data can model enumerations:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

Functions over user data types must *deconstruct* the arguments, with one case for each constructor:

```
whatShallIDo Sun  = “relax”
whatShallIDo Sat  = “go shopping”
whatShallIDo _    = “guess I'll have to go to work”
```

Notice how the data type Day has seven constructors, each without any arguments.

To find out which value a day is, we just pattern match it against its constructors, thus revealing what it is.

This is the *opposite* of encapsulation in OO languages, where you *never* deconstruct a value to access its data — instead the object has its own methods (services) that have exclusive access to the hidden data.

Union types

```
data Temp = Centigrade Float | Fahrenheit Float

freezing :: Temp -> Bool
freezing (Centigrade temp)  = temp <= 0.0
freezing (Fahrenheit temp)  = temp <= 32.0
```

In this case `Temp` has two constructors, each of which takes a single number as an argument. By pattern matching, we deconstruct the value and gain access to the hidden data inside.

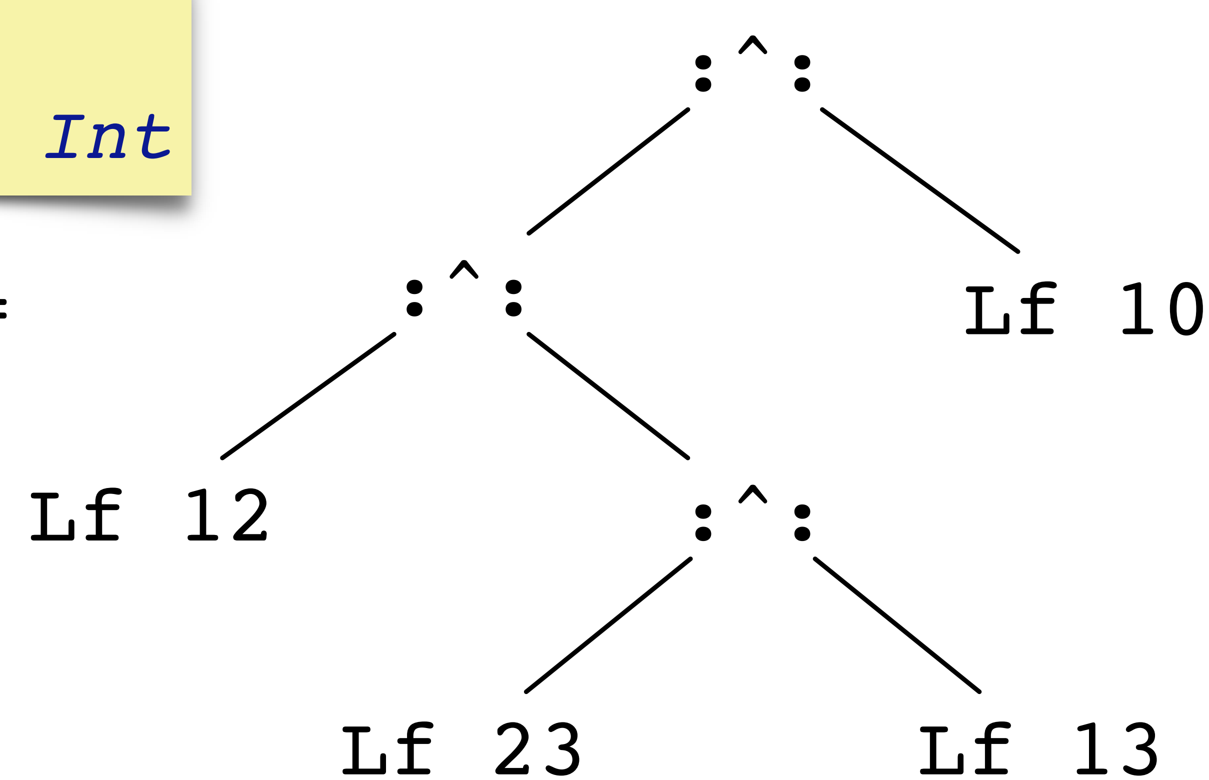
Recursive Data Types

A recursive data type provides constructors over the type itself:

```
data Tree a = Lf a | Tree a :^: Tree a
mytree = (Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10
```

```
? :t mytree
⇨ mytree :: Tree Int
```

my tree =



Recall that binary constructors are operators that start with "`:`". In order to suggest tree-like structures, we pick "`:^:`" as our constructor for building a tree out of two subtrees.

Trees therefore have two constructors: `Lf`, for building a leaf node from a single element of an arbitrary type, and `:^:` for building a composite tree from two subtrees containing elements of consistent types.

Note that Haskell allows us to build a tree of strings or a tree of integers, but not a tree of integers and strings.

Can you define a tree data type in Haskell that could hold either integers or strings? If so, how? If not, why not?

Using recursive data types

```
leaves, leaves' :: Tree a -> [a]
leaves (Lf l)   = [l]
leaves (l :^: r)  = leaves l ++ leaves r

leaves' t = leavesAcc t [ ]
  where leavesAcc (Lf l) = (l:)
        leavesAcc (l :^: r) = leavesAcc l . leavesAcc r
```

NB: `(f . g) x = f (g x)`

- ✎ *What do these functions do?*
- ✎ *Which function should be more efficient? Why?*
- ✎ *What is `(l:)` and what does it do?*

Recall that **:** is Haskell's built-in operator for constructing lists from a head and a tail.

```
Prelude> :t (:)
(:) :: a -> [a] -> [a]
```

Since **:** is a Curried function, we can supply one argument and get a function back. What does this function do?

NB: Dot (.) is the traditional function composition operator from mathematics. It is also a Curried function:

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Roadmap

- Static and Dynamic Types
- Type Completeness
- Types in Haskell
- **Monomorphic and Polymorphic types**
- Hindley-Milner Type Inference
- Overloading

Monomorphism

Languages like Pascal and C have monomorphic type systems: every constant, variable, parameter and function result *has a unique type.*

- good for type-checking
- bad for writing generic code
  - it is *impossible* in Pascal to write a generic sort procedure

“*Monomorphic*” simple means “every value has a single, unique type”.

“*Polymorphic*” means “values may have more than one type.”

Polymorphism

A polymorphic function accepts *arguments of different types*:

```
length          :: [a] -> Int
length [ ]      = 0
length (x:xs)   = 1 + length xs

map             :: (a -> b) -> [a] -> [b]
map f [ ]       = [ ]
map f (x:xs)    = f x : map f xs

(.)             :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x       = f (g x)
```

Kinds of Polymorphism

> *Universal polymorphism:*
  — Parametric: polymorphic map function in Haskell; nil/void pointer type in Pascal/C
  — Inclusion: subtyping — graphic objects

> *Ad Hoc polymorphism:*
  — Overloading: + applies to both integers and reals
  — Coercion: integer values can be used where reals are expected and v.v.

The kind of polymorphism supported by Haskell is also known as *parametric polymorphism* (functions may be generic in their parameter types). Object-oriented languages also support *inclusion* or *subtype polymorphism*, another kind of polymorphism that is not part of Haskell.

More on this in the lecture on “Objects and Types”.

Roadmap

> Static and Dynamic Types
> Type Completeness
> Types in Haskell
> Monomorphic and Polymorphic types
> **Hindley-Milner Type Inference**
> Overloading

Type Inference

We can *infer* the type of many expressions by simply examining their structure. Consider:

```
length [ ]      = 0
length (x:xs)   = 1 + length xs
```

Clearly:

`length :: a -> b`

Furthermore, `b` is obvious `int`, and `a` is a `list`, so:

`length :: [c] -> Int`

We cannot further refine the type, so we are done.

Note how the definition of length deconstructs lists by pattern-matching using the list constructors, i.e., `[ ]` to construct an empty list and `:` to construct a non-empty list.

By examining the definition of length, we can infer its type.

First, it is clearly a function with a single argument and return value, so its shape is clearly `a -> b` (where `a` and `b` are types or type variables).

Second, `a` is clearly a list, so its type can be refined to `[c]`, and `b` is clearly a number (let's say `Int`). Conclusion:

```
length :: [c] -> Int
```

Composing polymorphic types

We can deduce the types of expressions using polymorphic functions by simply *binding type variables to concrete types.*

Consider:

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

Then:

```
map length                         :: [[a]] -> [Int]
[ “Hello”, “World” ]               :: [[Char]]
map length [ “Hello”, “World” ] :: [Int]
```

First the type of `length` (`[a]->Int`) is unified with the argument type of `map` (`a->b`). This means that `a` is bound to `[a]` and `b` to `Int`.

The result type of `map` (`[a]->[b]`) therefore reduces to `[[a]]->[Int]`.

Next, we bind the type of `[“Hello”, “World”]` (`[[Char]]`) to the argument type of `map length` (`[[a]]`) so `a` binds to `Char`. The type of the final result is `[Int]`.

Note that type variables like `a` or `b` can be bound to any Haskell type, no matter how complex.

Polymorphic Type Inference

Hindley-Milner Type Inference automatically determines the types of many polymorphic functions.

```
map   f   [ ]      =   [ ]
map   f   (x:xs)   =   f x : map f xs
```

map :: *X* -> *Y* -> *Z*

map :: *(a -> b)* -> *[c]* -> *[d]*

map :: *(a -> b)* -> *[a]* -> *[b]*

The corresponding type system is used in many modern functional languages, including ML and Haskell.

The Hindley-Milner type inference algorithm was discovered independently first by J. Roger Hindley and later by Robin Milner.

Here we only sketch out the main idea, not the details of the algorithm itself. The algorithm is linear in the size of the source code, so it is very practical for real language implementations. Note that the algorithm only deals with parametric polymorphism, not subtype polymorphism, so it cannot be used for languages like Java.

We will use all available information to infer the type of map.

First we infer from the definition of `map` (a Curried function) that its type must have the form `X->Y->Z`.

Next we see that the first argument, `f`, of type `X`, is a function, so we expand `X` to `a->b`. We also note that the second argument is clearly a list, so we rewrite `Y` as `[c]`. By the same reasoning we rewrite `Z` as `[d]`. From the expression `f x` we deduce that `a` and `c` must be the same type (since `x` of type `c` is an argument to `f` of type `a->b`), so we replace `c` by `a`. By similar reasoning we replace `d` by `b`. Now we have used all the available information and we stop.

Type Specialization

A polymorphic function may be explicitly assigned a *more specific type:*

```
idInt :: Int -> Int
idInt x = x
```

Note that the `:t` command can be used to find the type of a particular expression that is inferred by Haskell:

```
? :t \x -> [x]
⇨ \x -> [x] :: a -> [a]

? :t (\x -> [x]) :: Char -> String
⇨ \x -> [x] :: Char -> String
```

Roadmap

> Static and Dynamic Types
> Type Completeness
> Types in Haskell
> Monomorphic and Polymorphic types
> Hindley-Milner Type Inference
> **Overloading**

Coercion vs overloading

Coercion or overloading — how do you distinguish?

```
3 + 4
3.0 + 4
3 + 4.0
3.0 + 4.0
```

✎ *Are there several overloaded + functions, or just one, with values automatically coerced?*

Are there four different overloaded + functions; two, with coercion to real if one arg is int; or one with coercion to real?

Overloading

Overloaded operators are introduced by means of type classes:

```
class Eq a where
   (==), (/=) :: a -> a -> Bool
   x /= y              = not (x == y)
   -- NB: defined in standard prelude
```

A type class must be *instantiated* to be used:

```
instance Eq Bool where
   True == True        = True
   False == False      = True
   _ == _              = False
```

Note that type classes have nothing to do with classes in OO languages! A type class in Haskell has more affinity with interfaces in Java: A type class defines a set of overloaded operators that must all be implemented by a given data type.

Instantiating overloaded operators

For each overloaded instance a separate definition must be given

```
instance Eq Int where (==)       = primEqInt

instance Eq Char where c == d    = ord c == ord d

instance (Eq a, Eq b) => Eq (a,b) where
   (x,y) == (u,v)                      = x==u && y==v

instance Eq a => Eq [a] where
   [ ] == [ ]                          = True
   [ ] == (y:ys)                       = False
   (x:xs) == [ ]                       = False
   (x:xs) == (y:ys)                    = x==y && xs==ys
```

Equality for Data Types

Why not automatically provide equality for all types of values?

User data types:

```
data Set a = Set [a]
instance Eq a => Eq (Set a) where
   Set xs == Set ys = xs `subset` ys && ys `subset` xs
      where xs `subset` ys = all (`elem` ys) xs
```

✎ *How would you define equality for the Tree data type?*

NB: `all ('elem' ys) xs` tests that every `x` in `xs` is an element of `ys`

Equality for Functions

Functions:

```
? (1==) == (\x->1==x)
ERROR: Cannot derive instance in expression
*** Expression : (==) d148 ((==) {dict} 1) (\x->(==) {dict} 1 x)
*** Required instance : Eq (Int -> Bool)
```

Determining equality of functions is undecidable in general!

What you should know!

- ✎ *How are the types of functions, lists and tuples specified?*
- ✎ *How can the type of an expression be inferred without evaluating it?*
- ✎ *What is a polymorphic function?*
- ✎ *How can the type of a polymorphic function be inferred?*
- ✎ *How does overloading differ from parametric polymorphism?*
- ✎ *How would you define == for tuples of length 3?*
- ✎ *How can you define your own data types?*
- ✎ *Why isn't == pre-defined for all types?*

Can you answer these questions?

- *Can any set of values be considered a type?*
- *Why does Haskell sometimes fail to infer the type of an expression?*
- *What is the type of the predefined function all? How would you implement it?*
- *How would you define equality for the Tree data type?*