

Key Facts on Timber

©Andrey Kruglyak, 2010

Reactivity

- There is a natural resting state for the system when nothing is being executed
- Once an external event or a timer event (passing of a delayed baseline) occurs, the system reacts to it and then returns to the resting state
- The system never blocks waiting for input or in an infinite loop, execution may only block waiting for a resource (an object) that is being actively used

Objects

- All system state (mutable variables) is encapsulated in objects
- The encapsulation is complete, i.e. state variables are only accessible from the object's methods
- Only one method can be active at any given time
- Objects are units of concurrency as methods of any two objects may execute concurrently, with external events and asynchronous messages being the source of concurrency

Methods

- Synchronous methods (requests) block the caller, asynchronous methods (actions) do not
- Actions can be delayed relative to the caller's baseline, and the original baseline is always the time of the external event that triggered the reaction

Functions

- Unlike methods that operate on the object's state, functions operate on the values of the argument and return a value (without producing any side effects)

Bindings

- A name can be bound to a value - but only once
 - at the top level in a module, or locally inside a class, method, branch of an "if" or "case" command, "let" expression, etc.
 - the name is bound everywhere inside the scope (up & down)
- Binding can be
 - a name binding (y = ...)
 - a function binding (f arg1 arg2 ... = ...) - can span multiple lines
 - a pattern binding (Just x = ...)

Recursion

- Bindings can be recursive:

`f x:xs = x + f xs`

`f [] = 0`

- Groups of bindings can also be recursive:

`a = 0:1:b`

`b = a`

Patterns

- A pattern is a special type of expression – a literal (4, "true", True, Nothing), a variable, a data constructor (Just v) or a struct constructor `Point{x = v1, y = v2}`
- Patterns are used in arguments in function bindings, in branches of "case", and in pattern bindings

Type inference and type declarations

- Types of all names, state variables, and terms are inferred and checked for consistency
- Declared types will be accepted but checked for consistency; these may be smaller than the inferred types but not larger

```
f :: [Int] -> Int
```

```
f [] = 0
```

```
f x:xs = x
```