DFINITY

Motoko

The Programming Language of the Internet Computer

Luc Bläser

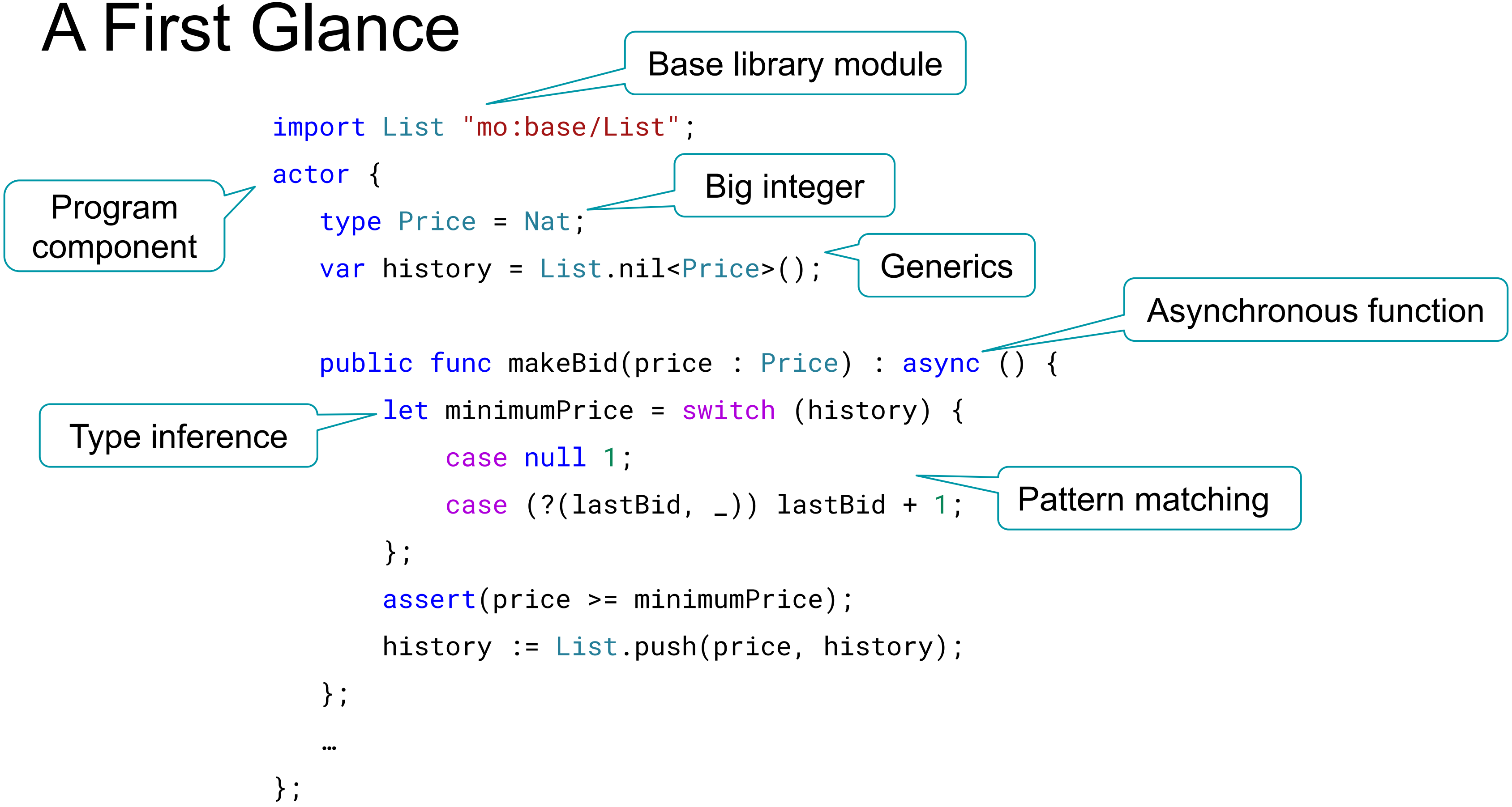
CySeP Summer School, Stockholm, June 13, 2023

What is Motoko?

A programming language specialized for the Internet Computer blockchain

IC	**Motoko**
General-purpose	Feature richness like JavaScript, Rust, and ML
Secure	Rigorous memory, type, and numeric safety
Decentralized	Actor model and asynchrony
Unstoppable	Orthogonal persistence

A First Glance



```
import List "mo:base/List";
actor {
    type Price = Nat;
    var history = List.nil<Price>();

    public func makeBid(price : Price) : async () {
        let minimumPrice = switch (history) {
            case null 1;
            case (?(lastBid, _)) lastBid + 1;
        };
        assert(price >= minimumPrice);
        history := List.push(price, history);
    };
    …
};
```

Base library module

Program component

Big integer

Generics

Asynchronous function

Type inference

Pattern matching

Design Goals

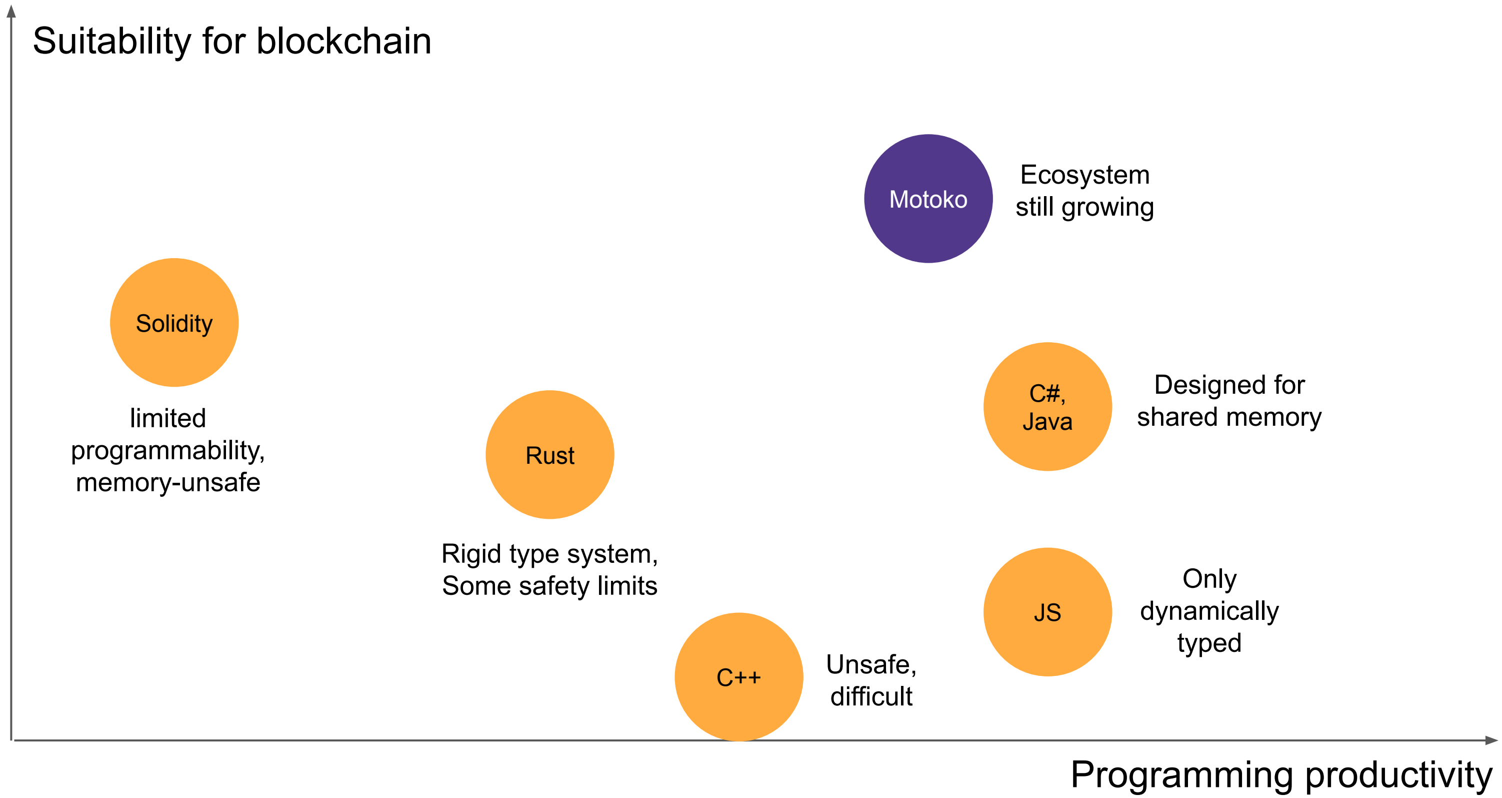
Suitability for blockchain

- Safety
- Expressiveness

Programming productivity

- Expressiveness
- Resemblance to JavaScript, C#, Rust

Language Landscape



Security Aspects

Blockchain-inherent security:

- Byzantine fault-tolerant execution
- Higher DOS resistance by replication and scalability
- Inbuilt authentication mechanism

Language-inherent safety:

- Reducing risks for bugs - and thus security vulnerabilities

Out of scope for this tutorial:

- Threshold ECDSA signing, blockchain data encryption, …

Learning Goals

Tutorial:

- Know the main concepts of the Motoko language
- Get ready for the subsequent Motoko workshop

Workshop:

- Experience how the blockchain can be programmed - and thus its inherent security be seamlessly applied

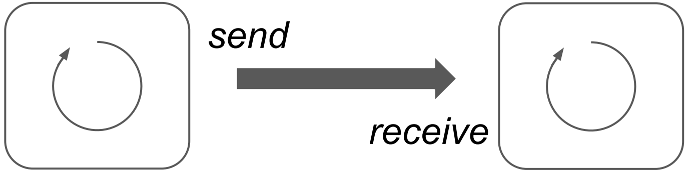
A Top-Down Language Tour

- Actors
- Asynchrony
- Types
- Objects
- Functions
- Persistence

Actors

Program is a set of components = actors that

- carry their encapsulated state
- run concurrently to each other
- communicate by message passing (no shared state)



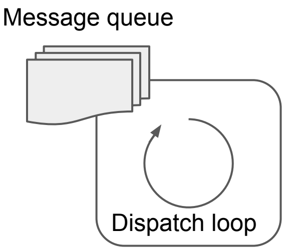
asynchronous

C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI. 1973.

An Implementation Look

Each actor consists of:

- Local memory
- Incoming message queue
- Dispatch loop
    - Processing the queue sequentially
    - Executing code per message



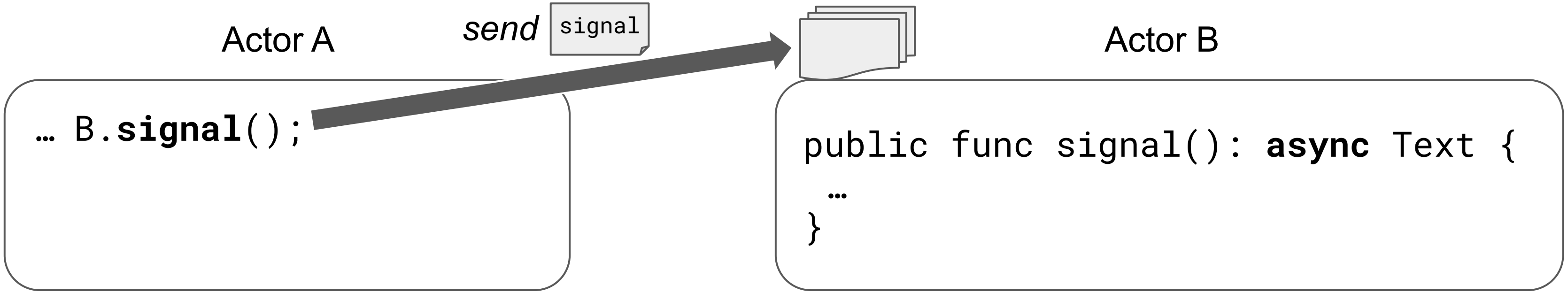
Actors run sequentially on the inside and concurrently on the outside

Asynchrony

In Motoko, actor communication is realized by asynchronous functions

Async function call	Send
Async function execution	Receive
Return from `async` function	Send (reply)
`await` expression	Receive (reply)

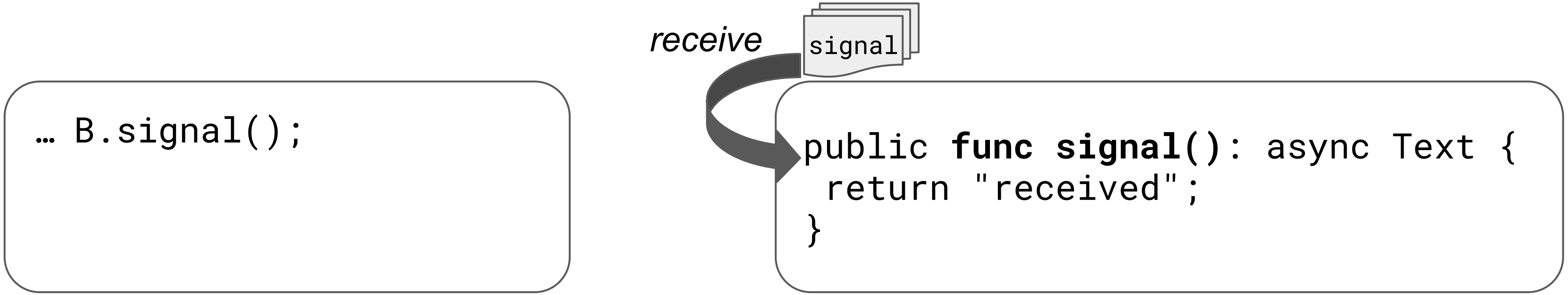
Async Function Call



```
… B.signal();
```

```
public func signal(): async Text {
  …
}
```

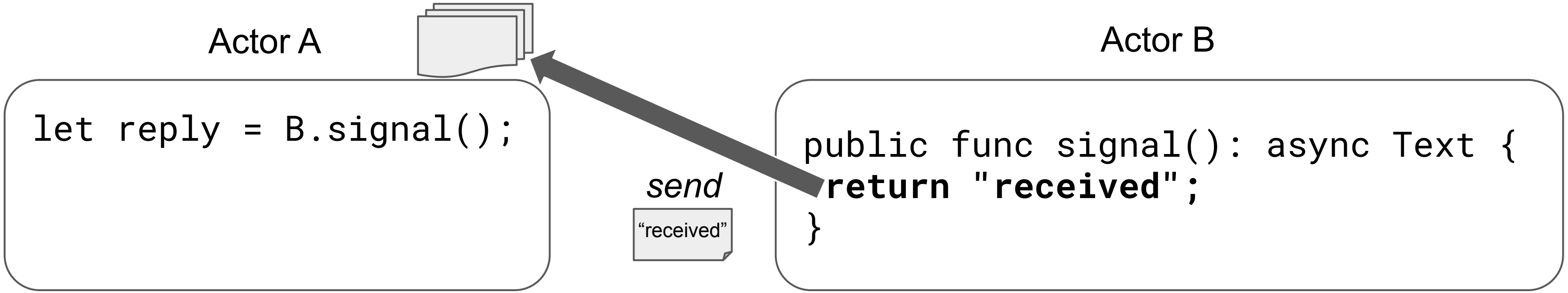
Async Function Execution



```
… B.signal();
```

```
public func signal(): async Text {
 return "received";
}
```

Async Function Return



Actor A

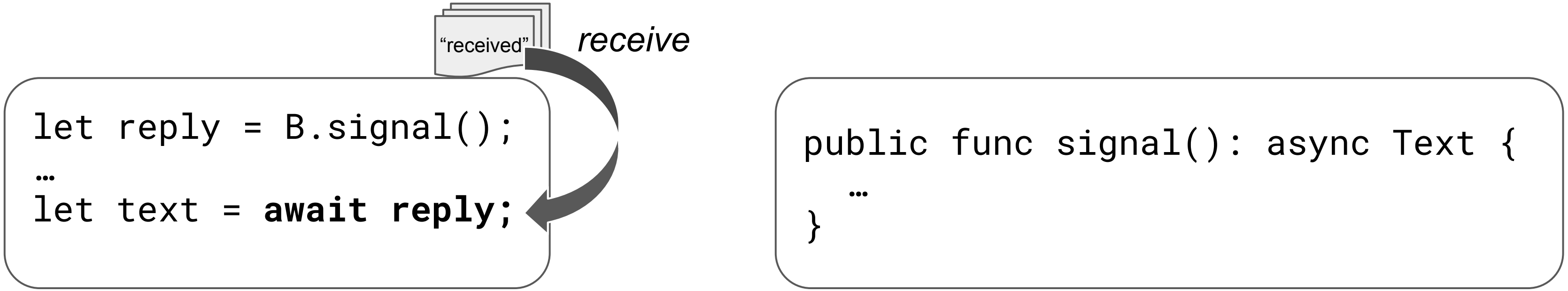
```
let reply = B.signal();
```

send

Actor B

```
public func signal(): async Text {
 return "received";
}
```

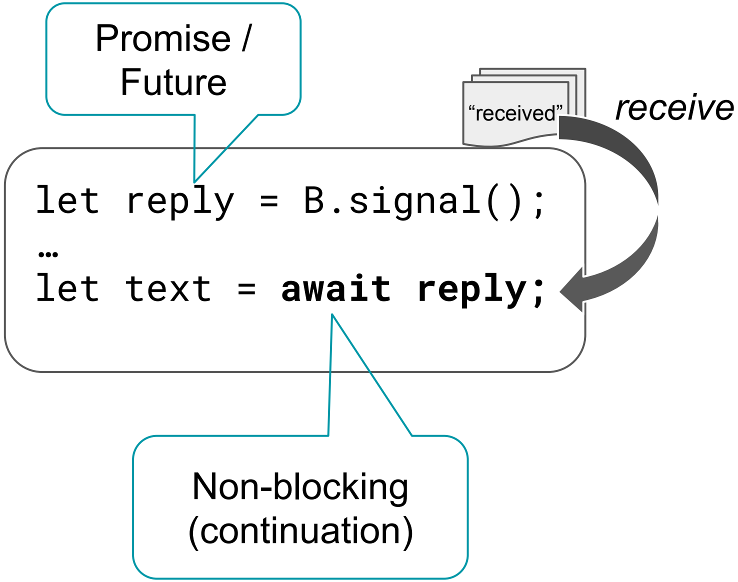
Await Expression



```
let reply = B.signal();
…
let text = await reply;
```

```
public func signal(): async Text {
    …
}
```

Continuation-Style Programming



Promise / Future

“received”

receive

```
let reply = B.signal();
…
let text = await reply;
```

Non-blocking (continuation)

Async/Await Constructs

Similar to JavaScript, C#, or C++ 20

Function with an **`async`** return type

- Caller is not blocked during invocation
- Caller obtains a promise = handle for async function

`await` a promise

- Pause the current execution and let other code run
- Resume later when the function behind the promise has completed
- Obtain the result value of the awaited function

Seamless Integration to the IC

The software components of the IC are canisters:

- A canister is also an actor
- Async/await → actor → canister

Message encoding:

- Standard format on the IC: Candid
- Automatic encoding/decoding by Motoko

Types

Primitive	`Bool, Nat, Int, Float, Text, Blob, …`	
Tuple	`(Nat, Text, Bool)`	`(123, "Motoko", true)`
Record	`{ name: Text; year: Nat }`	`{ name="CySeP"; year=2023 }`
Array	`[Nat]`	`[1, 2, 3]`
Option	`?Bool`	`null, ?true`
Variant	`{ #North; #South; #East; #West }`	`#North`
Function	`Int -> Bool`	`func (x) { x % 2 == 0 }`

Mutable State

Mutable fields/arrays must be explicitly declared as `var`

`{`
`name: Text;`
`var year: Nat;`
`}`
	`{`
	`name = "CySeP";`
	`var year = 2023;`
	`}`

`[var Nat]`	`[var 1, 2, 3]`

Semantics

Value semantics (copying) for primitive types

```
var x = 0;
let y = x;
x += 1;
Debug.print(debug_show(y));
// Output: 0
```

Reference semantics (sharing) for composite types

```
let x = { var value = 0 };
let y = x;
x.value += 1;
Debug.print(debug_show(y));
// Output: {value = 1}
```

Like JavaScript and Java

Shareable Types = Serializable

Types that can be sent across actors:

- Primitive types
- Immutable composed types
- No `var` components
- No function types

For immutability: Reference semantics = Value semantics

Also shareable: Remote calls (“shared functions”), actor references

Structural Typing

Type x is compatible to y if

- They have identical structure
- Record x declares more fields than record y (subtyping)

```
type Work = { author: Text; };
type Picture = { author: Text; image: Blob; };
type Literature = { author: Text; content: Text; };

let book = { author = "Shakespeare"; content = "...to be or not to be..."};
// implicitly compatible to Literature and Work
```

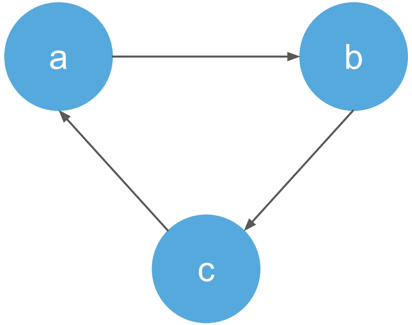
Object-Orientation

```
class Website(url: Text) {
   var links: [Website] = [];

   public func addLink(to: Website) {
       links := Array.append(links, [to]);
   }
};
```

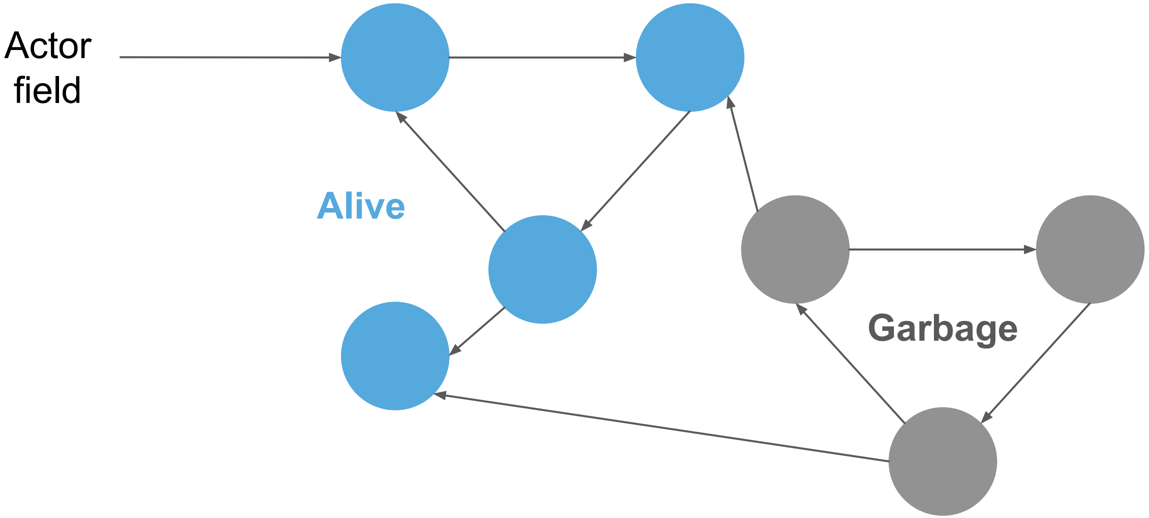
```
type Website = {
  url: Text;
  var links: [Website];
  addLink: Website -> ();
}
```

```
let a = Website("dfinity.org");
let b = Website("internetcomputer.org");
let c = Website("cysep.conf.kth.se");
a.addLink(b);
b.addLink(c);
c.addLink(a);
```



Garbage Collection

Automatic reclamation of unreachable objects inside the actor



Motoko features a powerful incremental GC

A Word about Safety

Type safety

- Static types
- Dynamic types
- No implicit null deref

Memory safety

- Garbage collection

Numeric safety

- Unbound integers
- Overflow always checked

Comparison to Other Languages

Rust

- Memory leaks with reference counters possible
- Overflow not checked in production mode
- “Unsafe” mode

C#, Java, JavaScript

- Unchecked overflows (in production mode)
- BigInt is not the default integer type
- Prone to null deref exceptions

→ Safety is particularly important on blockchain

Functions

```
public func translate(input: Text): async Text { … }
public func store(content: Blob): async () { … }
func max(x: Nat, y: Nat): Nat = x + y;
func printArray(array: [?Int]) { … }
```

Support both imperative and functional programming

- `switch` (with pattern matching), `if-else`
- `if`, `while`, `loop`, `for`, `return`
- function calls, `await`
- Local variables, local functions

Imperative Programming

```
let array: [?Int] = …;
var sum = +0;
var gaps = false;
for (entry in array.vals()) {
    switch entry {
        case (?number) { sum += number };
        case null { gaps := true }
    }
};
Debug.print("Sum " # debug_show(sum) # " gaps: " # debug_show(gaps));
```

Iterator

null test with
pattern matching

Functional Programming

```
let (sum, gaps) = Array.foldLeft<?Int, (Int, Bool)>(
    array,
    (+0, false),
    func((leftSum, leftGaps), entry) {
        switch entry {
            case (?number) (leftSum + number, leftGaps);
            case null (leftSum, true);
        };
    }
);
Debug.print("Sum " # debug_show (sum) # " gaps: " # debug_show (gaps));
```

Anonymous function (lambda)

Orthogonal Persistence

IC canisters and thus actors live conceptually perpetually

- State is automatically persisted
- No need for a database, file system, external storage

Special aspect: Upgrade

- Changing the program implementation
- Requires evolving the existing data

Persistent Program

```
actor {
   …
   type Auction = {
       id : AuctionId;
       item : Item;
       var bidHistory : List.List<Bid>;
       var remainingTime : Nat;
   };

   var auctions = List.nil<Auction>();
   var idCounter = 0;
   …
};
```

However, state is discarded on program change (upgrade)

Prepare for Upgrade

```
actor {
   …
   type Auction = {
       id : AuctionId;
       item : Item;
       var bidHistory : List.List<Bid>;
       var remainingTime : Nat;
   };

   stable var auctions = List.nil<Auction>();
   stable var idCounter = 0;
   …
};
```

Survive upgrade to future program version

Stable Modifier

Everything transitively reachable from `stable` fields is upgraded

- Motoko automatically transitions the stable sub-graph of the heap

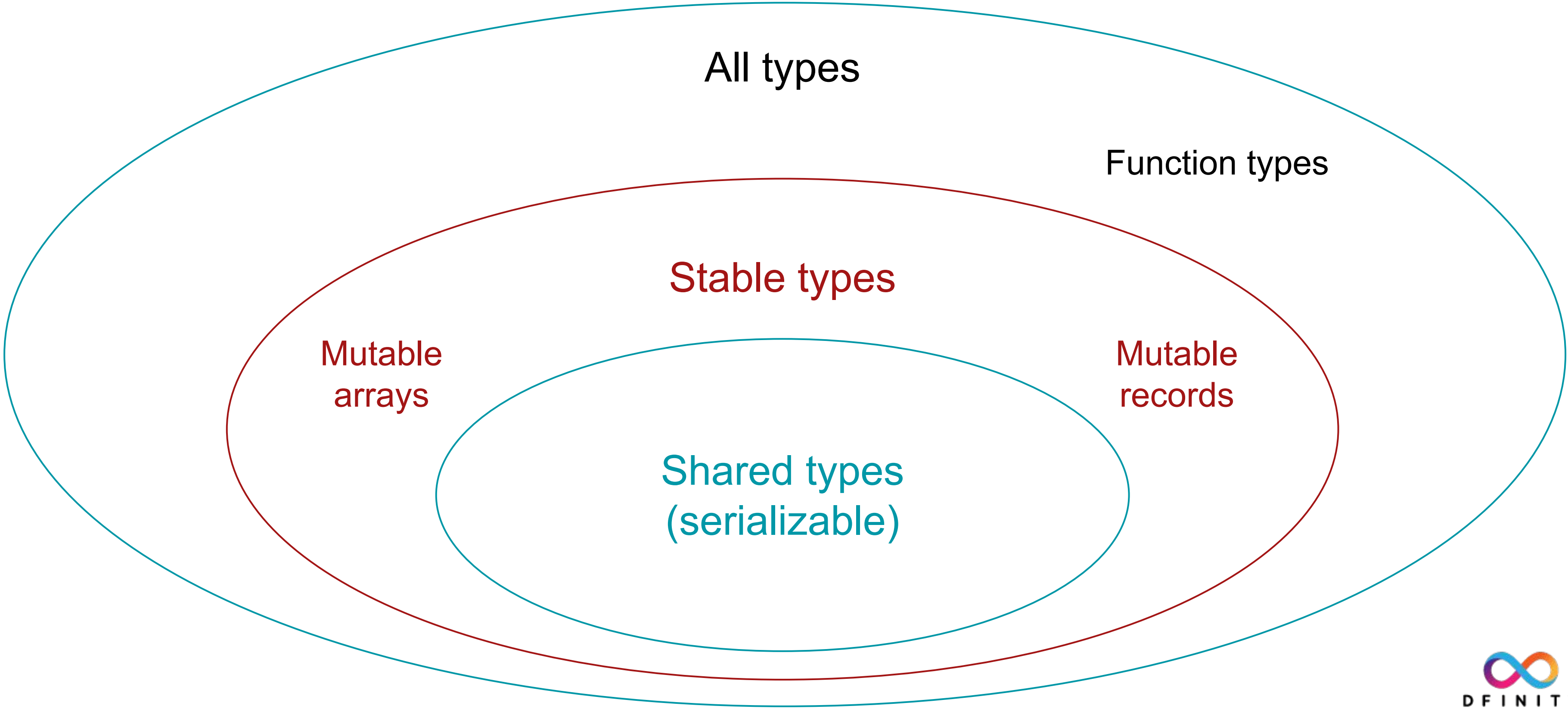
Only certain types can be upgraded

- No function types

Can also upgrade non-stable variables with upgrade hooks

- See documentation

Type Categories



Modules

Set of functionality that can be imported to actors and other modules.

Base library modules:

`"mo:base/Timer"`	One-shot or periodic time events
`"mo:base/Principal"`	Authentication (Internet Identity)
`"mo:base/Debug"`	Debug output, raising errors (traps)
`"mo:base/List"`	List data structure (stable type)
…	

Conclusion

Motoko aims for optimal programming on the IC blockchain

First-class support of IC-concepts

- Actors, orthogonal persistence

Easy to learn

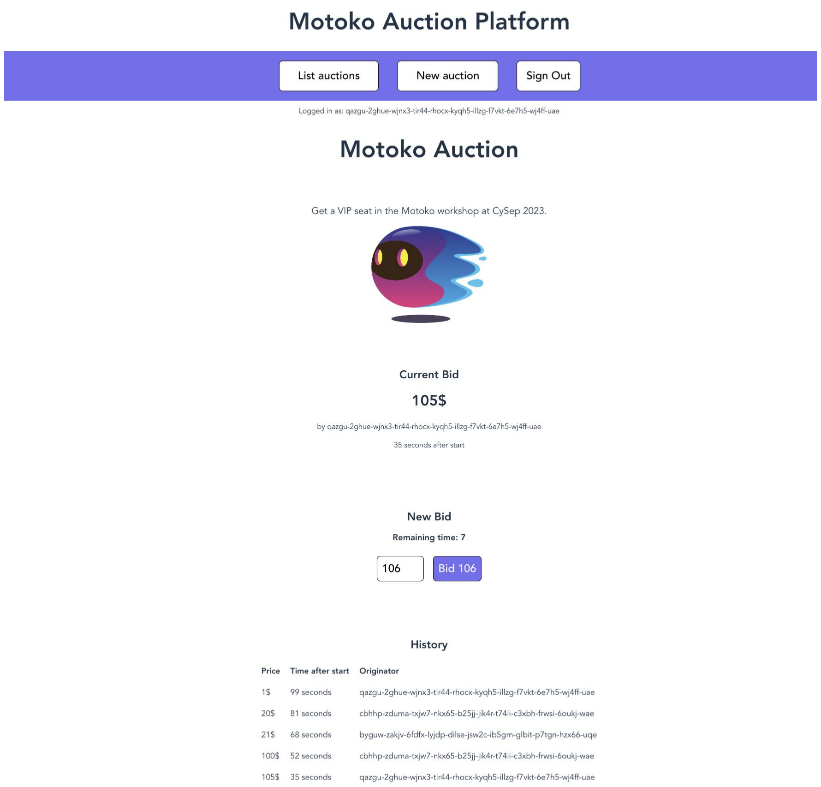
- Resemblance to JavaScript, Rust, ML

Emphasis on safety

- Higher than in other languages

Upcoming: Motoko Workshop

Mini-Hackathon:
Developing an
Auction Platform
with Motoko on the IC



Motoko Auction Platform

Logged in as: qazgu-2ghue-wjnx3-tir44-rhocx-kyqh5-illzg-f7vkt-6e7h5-wj4ff-uae

Motoko Auction

Get a VIP seat in the Motoko workshop at CySep 2023.

Current Bid

105$

by qazgu-2ghue-wjnx3-tir44-rhocx-kyqh5-illzg-f7vkt-6e7h5-wj4ff-uae

35 seconds after start

New Bid

Remaining time: 7

History

Price	Time after start	Originator
1$	99 seconds	qazgu-2ghue-wjnx3-tir44-rhocx-kyqh5-illzg-f7vkt-6e7h5-wj4ff-uae
20$	81 seconds	cbhhp-zduma-txjw7-nkx65-b25jj-jik4r-t74ii-c3xbh-frwsi-6oukj-wae
21$	68 seconds	byguw-zakjv-6fdfx-lyjdp-dilse-jsw2c-ib5gm-glbit-p7tgn-hzx66-uqe
100$	52 seconds	cbhhp-zduma-txjw7-nkx65-b25jj-jik4r-t74ii-c3xbh-frwsi-6oukj-wae
105$	35 seconds	qazgu-2ghue-wjnx3-tir44-rhocx-kyqh5-illzg-f7vkt-6e7h5-wj4ff-uae

Motoko Workshop

https://github.com/luc-blaeser/auction

Learn More

- Motoko Documentation: https://internetcomputer.org/docs/current/motoko/main/motoko
- Motoko Open Source Repository: https://github.com/dfinity/motoko

Common Pitfalls

Using `await` carelessly	Other async code can run in meantime at await. Beware of race conditions!
Missing `stable` modifier (or upgrade hooks)	Data will be lost on program version upgrade!
Using query functions	Requires a certified variable to be secure
Blockchain transaction limit	Message runtime is limited, split into shorter messages or async / await sections
Public actor functions without return type	One-way calls (“fire and forget”), no propagation of errors, specify return type `async()` and await