Powerful Blockchain Programming on the Internet Computer

Luc Bläser

CySeP Summer School, Stockholm, June 12, 2024

The Internet Computer (IC)

A secure distributed virtual machine:

- Replicating computation across distributed nodes
- Byzantine-fault-tolerant consensus on computation

Application cases:

- Decentralized exchanges, smart contracts, DAOs, cloud services, …

Our example: Auction platform

DFINITY

Selection of Languages

Low-level: WebAssembly with specific API

High-level: Any language that compiles to WebAssembly

TypeScript

Rust

Motoko

Designed for IC

...more…

DFINITY

A First Glance with TypeScript

TS

```
import { ic, Canister, Void, update, nat } from 'azle';

let history: nat[] = [];

export default Canister({
 makeBid: update([nat], Void, (price) => {
   if (price < minimumPrice()) {
     ic.trap("Price too low");
   }
   history.unshift(price);
 })
 …
})
```

Typescript IC package

Big natural number on IC

Exported IC async function `makeBid(price: nat)`

Same in Motoko

```
import List "mo:base/List";

actor {
   stable var history = List.nil<Nat>();

   public func makeBid(price : Nat) : async () {
       assert(price >= minimumPrice());
       history := List.push(price, history);
   };
   …
};
```

Motoko base library

Software component

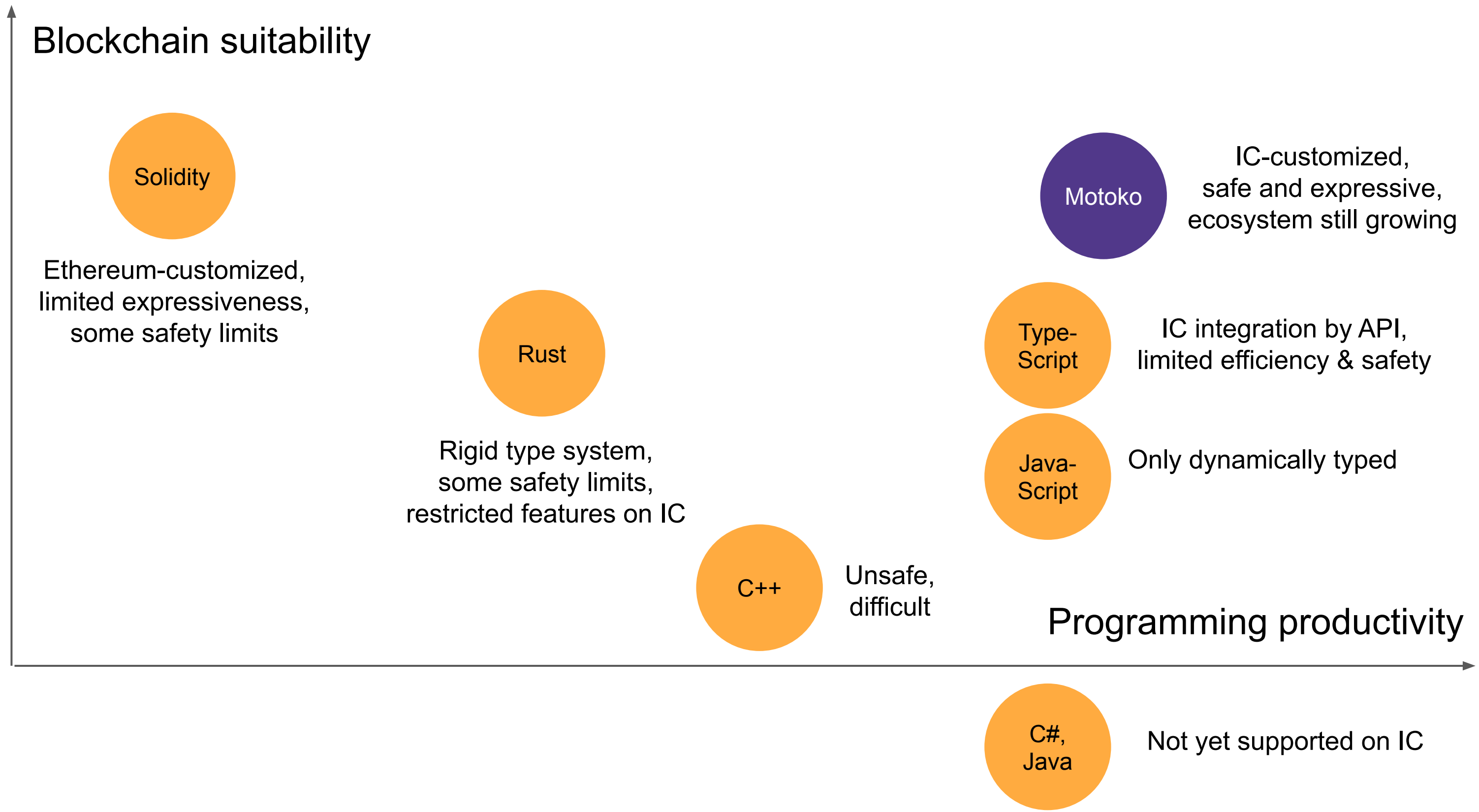
Exported IC function

Motivation of Motoko

Optimized for blockchain programming:

- Direct IC integration
  - Inbuilt language concepts for IC aspects
- Safety & security
  - Type safety covering IC aspects, garbage collection, supply chain security, …
- Easy to learn
  - Resemblance to Typescript, C#, and Ocaml
- Efficiency
  - Runtime system optimized for blockchain

Motoko’s Position



Learning Goals

Tutorial:

- Get an overview of blockchain programming on the IC
- See how this is supported in different programming languages

Workshop:

- Experience how the blockchain can be programmed -
  Choose a language of your preference (Motoko, Typescript, Rust)

DFINITY

Tutorial Overview

IC programming:

- Canisters/Actors
- Asynchrony
- State
- Transactions
- Persistence
- Safety
- Security
- Performance

Examples in

TS

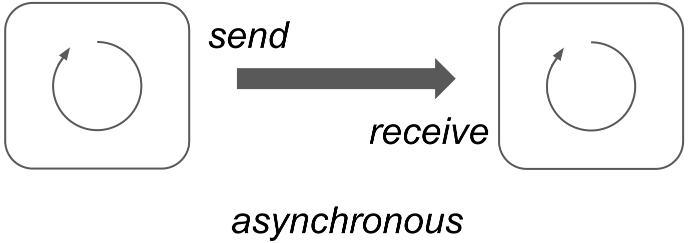
DFINITY

Software Components

A program on the IC is a set of components, called **canisters.**

Canisters are **actors** that

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



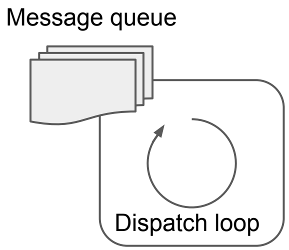
C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI. 1973.

DFINITY

An Implementation Look

Each actor consists of:

- Local memory
  - Stored on blockchain
- Incoming message queue
  - Also on blockchain
- Dispatch loop
  - Processing the queue sequentially
  - Executing code per message



Actors run sequentially on the inside and concurrently on the outside

Asynchrony

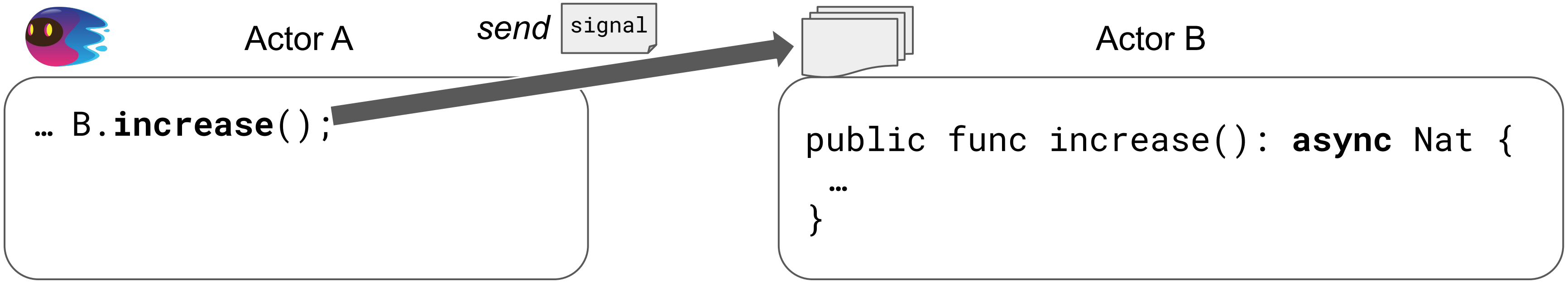
Asynchronous programming can be mapped to actor communication

Async/Await Model	**Actor Model**
Async function call	Send
Async function execution	Receive
Return from `async` function	Send
`await` expression	Receive

Used by Motoko, Rust, TypeScript for the IC

DFINITY

Async Function Call



Async Function Execution

receive

```
signal
```

```
… B.increase();
```

```
public func increase(): async Nat {
  counter += 1;
  return counter;
}
```

Async Function Return

Actor A

```
let future = B.increase();
```

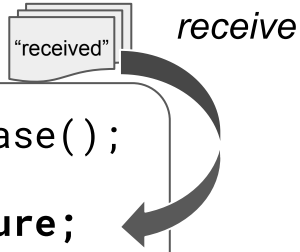
send

"received"

Actor B

```
public func increase(): async Nat {
  counter += 1;
  return counter;
}
```

Await Expression




```
let future = B.increase();
…
let text = await future;
```

```
public func increase(): async Nat {
  …
}
```

Actor in Motoko

```
actor {
   stable var counter = 0;

   public func increase() : async Nat {
     counter += 1;
     return counter;
   };
};
```

Internal state

Callable from outside

Type system statically checks:

- Calls match function declaration
- Arguments & result are serializable

Canister in TypeScript

```
let counter: nat = 0;

export default Canister({

 increase: update([], nat, () => {
   counter++;
   return counter;
 })
 …
})
```

Internal state

Default call mode

Argument types

Return type

⚠️ Function signature is checked at runtime

⚠️ Arguments/result must be IC types

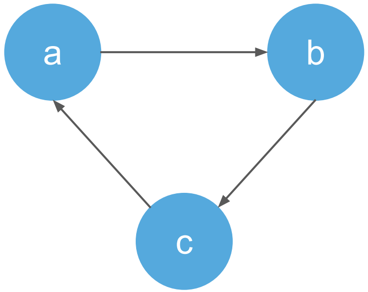
Canister State

State of actor/canister is stored on the blockchain

- Can have any object-oriented structure

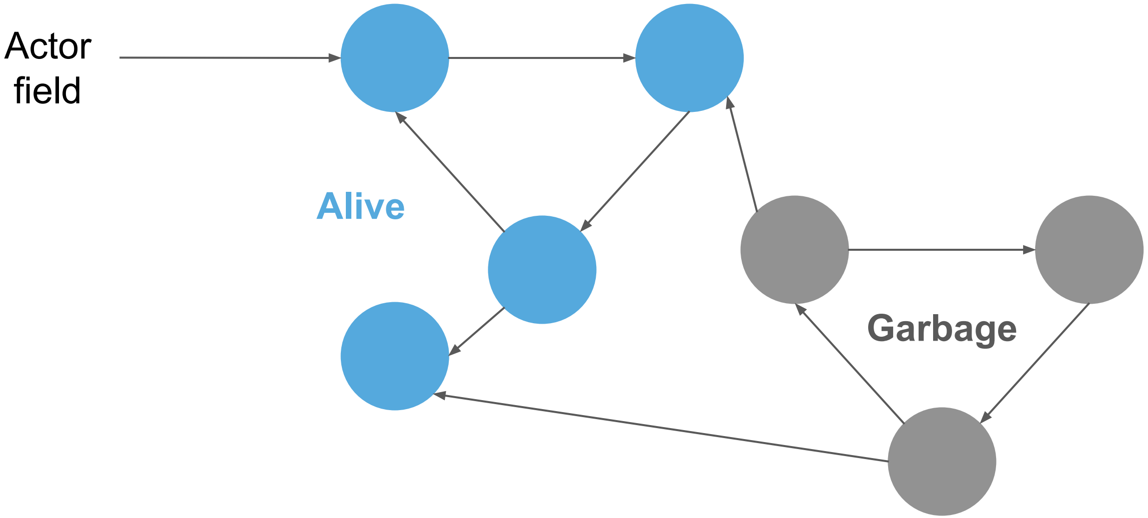
```
class Website(url: Text) {
   var links: [Website] = [];

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

```
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 blockchain-optimized GC

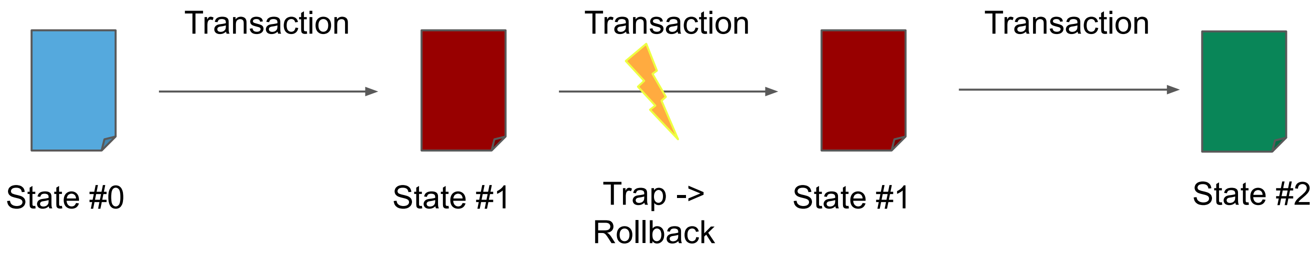
L. Bläser, C. Russo, U. Degenbaev, Ö. S. Agaçan, G. Grief, and J. Ibrahim. Collecting Garbage on the Blockchain, VMIL, 2023.

Transactions

Function calls run as transactions.

Call end and awaits denote commit points:

- Success: Apply all changes to blockchain
- Trap: Rollback all recent changes/effects



Precondition Checking

TS

```
if (price < minimumPrice()) {
  ic.trap("Price too low");
}
history.unshift(price);
```

Abort & Rollback

Commit change on call return

```
assert(price >= minimumPrice());
history := List.push(price, history);
```

Trap if violated

Caller Identification

```
public shared (message) func check() : async () {
   let originator = message.caller;
   if (Principal.isAnonymous(originator)) {
     Debug.trap("Anonymous caller");
   };
   …
};
```

Principal is a public key identifier of the caller, e.g. `un4fu-tqaaa-aaaab-qadjq-cai`

```
check: update([], Void, () => {
   let originator = ic.caller();
   if (originator.isAnonymous()) {
     ic.trap("Anonymous caller");
   }
   …
}
```

Persistence and Upgrades

IC canisters and thus actors live conceptually perpetually

- State is automatically persisted across transactions

Special aspect: Upgrade

- Changing the program implementation
- Requires evolving the existing data

Without special attention, state is discarded on program change (upgrade).

Motoko: Orthogonal Persistence

```
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 should become default in future

Stable Modifier

Everything transitively reachable from `stable` fields is upgraded:

- Motoko automatically transitions the stable sub-graph of the heap.
- Safety check: Ensures that data evolution is compatible.

Only certain types can be upgraded

- No function types

Other Languages: TypeScript, Rust, etc.

No support for orthogonal persistence across upgrades.

Need to store data explicitly in separate stable memory:

- Stable data structures
- See documentation

```
let map = StableBTreeMap<Key, Auction>(0);
```

Restricted to serializable types

Safety for Blockchain Programming

Motoko:

- Memory safety (GC), static type safety, numeric safety
- Static checks include IC aspects (actor calls, persistence etc.)
- Capability system to mitigate supply chain attacks

Other languages:

- IC aspects are not statically checked (e.g. calls)
- Data can be corrupted with stable memory/data structures
- Rust: unsafe code, unchecked overflows in release mode, memory leaks with cyclic reference counting
- Vulnerable to supply chain attacks (unrestricted IC API access)

DFINITY

Performance

IC usage is charged in terms of instructions and memory

- #Instructions per transaction is also limited (40 billion)

Auction with 1000 entries, each 100 bids, `makeBid()`

	TypeScript	Rust	Motoko
Binary size	2.2 MB	690 KB	177 KB
Instructions	19_000_000	25_000	19_000
Memory	26 MB	12 MB	12 MB

Runtime optimized for IC

DFINITY

Benefits of A Bespoke Language

Motoko offers advanced runtime supported tailored to the IC:

- Blockchain-optimized garbage collector
- Static checks of IC features
- Orthogonal persistence for upgrades
- Efficient (de)serialization driven by static types

→ This is not available in mainstream language implementations

Upcoming:

- Constant-time upgrade with 64-bit persistent main memory
  https://github.com/dfinity/motoko/pull/4488

DFINITY

Conclusion

The IC is a powerful runtime platform for secure distributed applications

Supports various programming languages:

- TypeScript, Motoko, Rust, and more

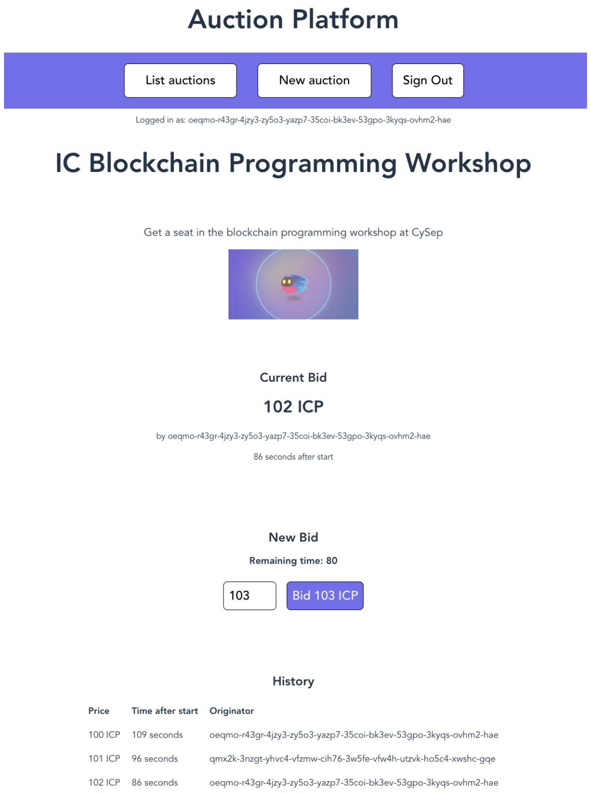
Motoko has been specifically designed for the IC:

- First-class support of IC-concepts
- Focus on safety, yet simple and expressive
- Efficient and advanced runtime mechanisms

DFINITY

Upcoming: IC Programming Workshop

Mini-Hackathon:
Developing an Auction Platform on the IC



Choose a language:
- **Motoko**
- **TypeScript**
- **Rust**

DFINITY

IC Blockchain Programming 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

- TypeScript Development Kit for IC (Azle):
  https://internetcomputer.org/docs/current/developer-docs/backend/typescript
- Rust Development Kit for IC:
  https://internetcomputer.org/docs/current/developer-docs/backend/rust/

DFINITY

Common Pitfalls

Using `await` carelessly	Other async code can run in meantime at await. Beware of race conditions!
Using normal variables for canister state	Data will be lost on program version upgrade! Motoko: Use stable modifier Otherwise: Use stable data structures
Using query functions	Requires a certified variable to be secure. Otherwise: Use regular functions (“update” in TypeScript)
Transaction instruction limit	Transaction runtime is limited, split into shorter running functions or async / await sections
Public actor functions without return type	One-way calls (“fire and forget”), no propagation of errors, Motoko: specify return type `async()` and await

Appendix: Motoko Overview

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 }`

DFINITY

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]`

DFINITY

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 composite types
- No `var` components
- No function types

Automatic serialization/deserialization to IC standard format (Candid)

For immutability: Reference semantics = Value semantics

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

Structural Typing

Types are equal if

- They have the identical structure
- Fields can be reordered

```
type Photo = { pixels: Blob; metadata: Text; };
type Picture = { metadata: Text; pixels: Blob; };
// Photo and Picture are equal
```

Subtyping

Type T is compatible to U if

- They have identical structure, or
- Record T declares more fields than record U

```
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
```

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

Asynchronous Programming

```
func test(): async Text {
  let future = B.increase();
  …
  let text = await future;

  return text;
}
```

Promise

Async call

Non-blocking
(continuation)

```
func increase(): async Nat { … }
```

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

DFINITY

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

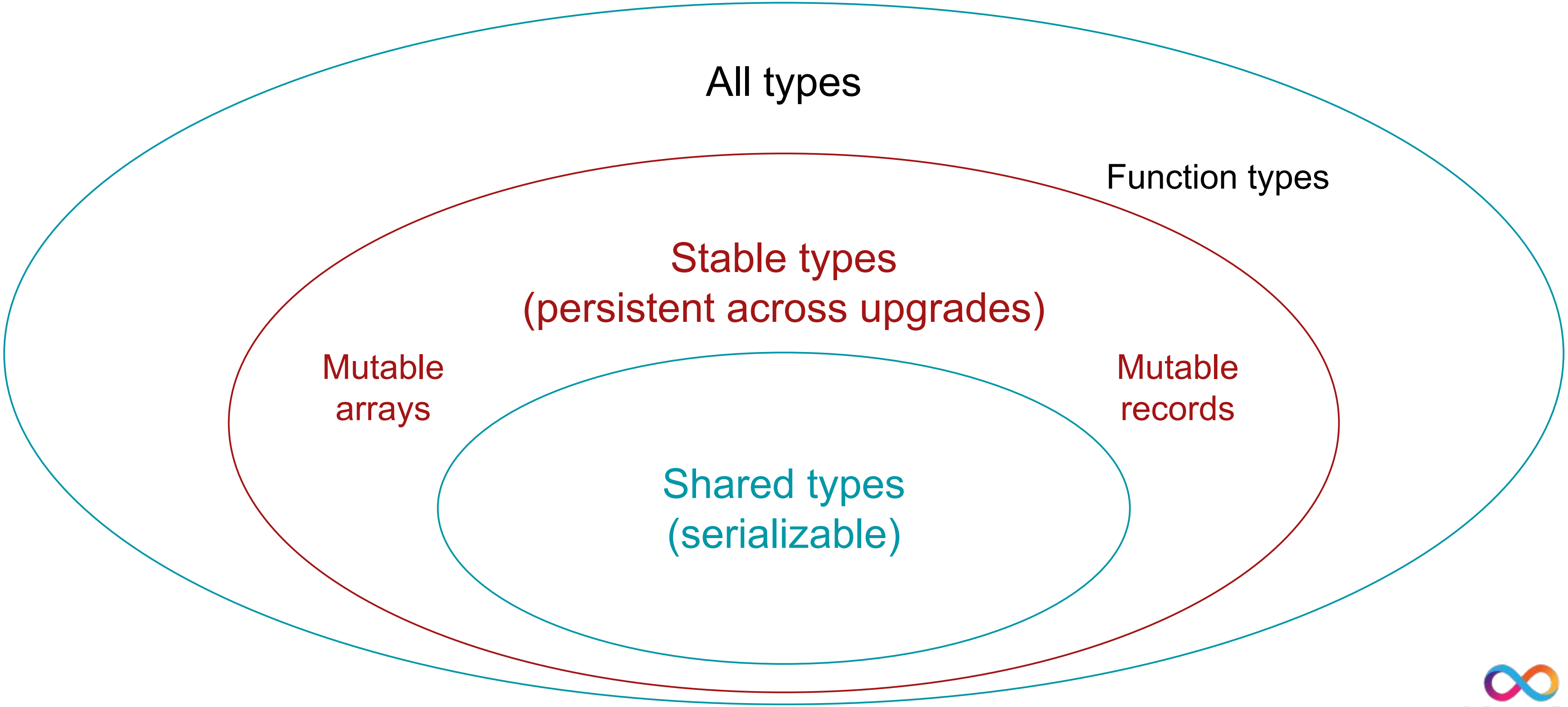
Functional Programming

```
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)

DFINITY

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)
...	

DFINITY