DFINITY

# Motoko

## A Programming Language Designed for Secure Smart Contract Programming

Luc Bläser

CySeP Summer School, Stockholm, June 13, 2025

# Rethinking the Software Stack for Security

| | |
|---|---|
| Software Application | Smart Contract / dApp |
| Programming Language | Motoko |
| Operating System | Web Assembly |
| Computer Machine | Internet Computer |

# The Motoko Programming Language

Designed for **secure** and **productive** development on the Internet Computer

Canisters by language

| | Unique | Total |
|---|---|---|
| Motoko (docs): | 10'300 | 60'580 |
| Rust (docs): | 1'077 | 233'216 |
| JS/TS (Azle): | 186 | 192 |
| C++ (icpp-pro): | 19 | 45 |
| Python (Kybra): | 12 | 12 |
| Unknown: | 6'982 | 349'542 |

Source: icp.zone

Released in 2019

Team of 6 engineers

# A First Glance

```
persistent actor {
  type Price = Nat;
  var history = List.empty<Price>();

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

Program component

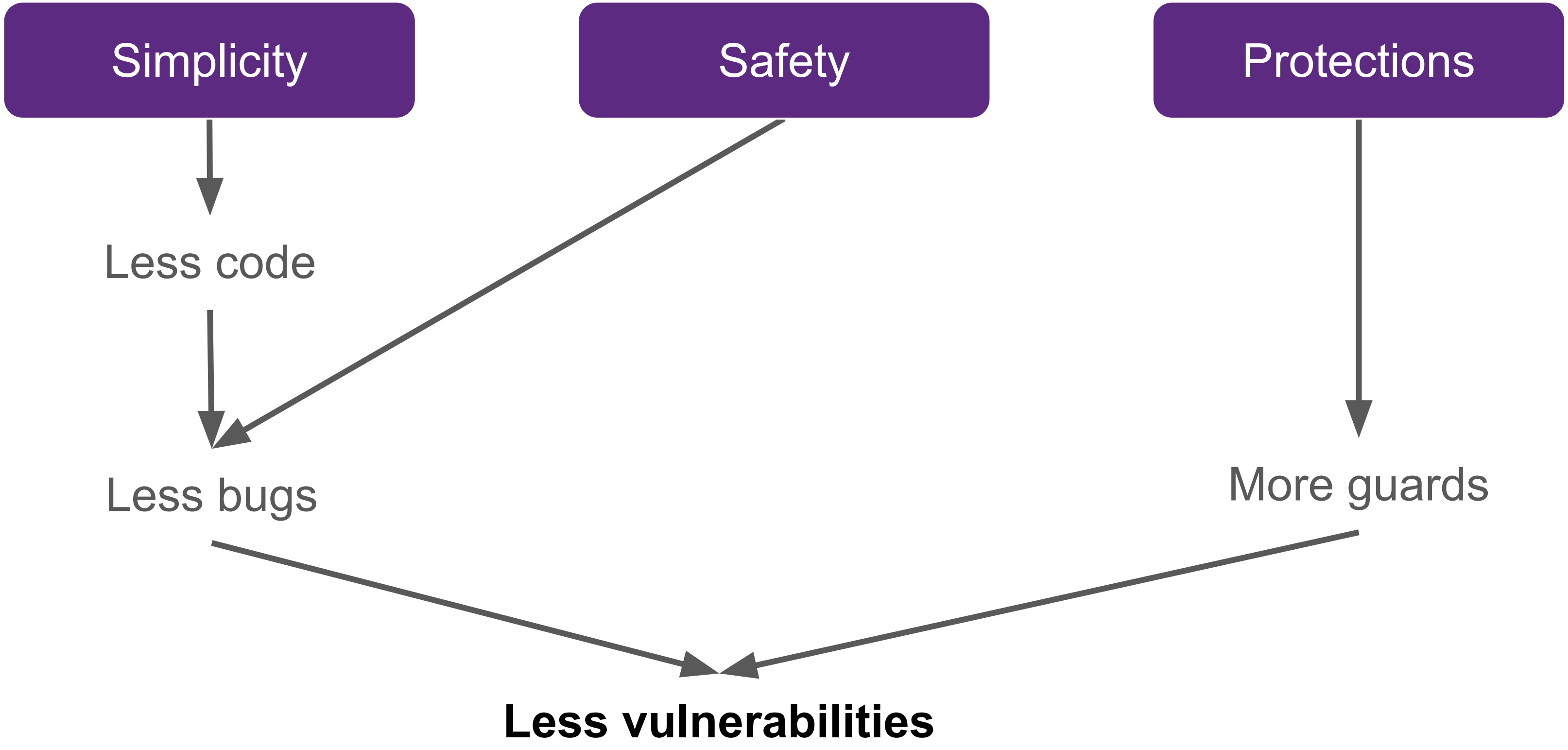
Automatic persistence

API for frontend

Functional flavor

Imperative flavor

# How the Programming Language Impacts Security



# Motoko's Design Philosophy

**Simplicity**

Few but powerful concepts

**Safety**

Static checks as much as possible

**Protections**

Security-centered features

# Learning Goals

Talk:

- Understand how language design can impact security
- Get an overview of Motoko and its bespoke security-centered concepts

Workshop:

- Experience programming in Motoko on the Internet Computer
- Harden the security of a simple decentralized app

# Looking At

**Simplicity** | Safety | Protections

1. Inherent distributability
2. Automatic persistence
3. Garbage collection

# 1. Inherent Distributability

Motoko is built of 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.

# Motoko Actor

```
persistent actor {
    …
    var history = List.empty<Price>();

    public func makeBid(price : Price) : async () {
        …
    };

    public func lastBid() : async Price {
        …
    };
};
```

Encapsulated state

Triggered on message receive

Result is sent back as message

Compiler checks public actor functions:

- Must be async
- Parameters are serializable
- Result is serializable

# Seamless Integration to the IC

The software components of the IC are canisters:

- A canister is also an actor
- Motoko actor can also instantiate new actors

Message encoding:

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

→ IC model is language-inbuilt and compile-checked

# 2. Automatic Persistence

```
persistent actor {
  type Item = {
    description : Text;
    image : Blob;
  };
  type Auction = {
    item : Item;
    bidHistory : List.List<Bid>;
  };
  …
  let auctions = Map.empty<AuctionId, Auction>();
};
```

State is automatically retained

→ No database

→ No files

→ No storage API

Called **orthogonal persistence**

L. Bläser, C. Russo et al. Smarter Contract Upgrades with Orthogonal Persistence. VMIL 2024.

# Program Evolution

Data migration when changing program

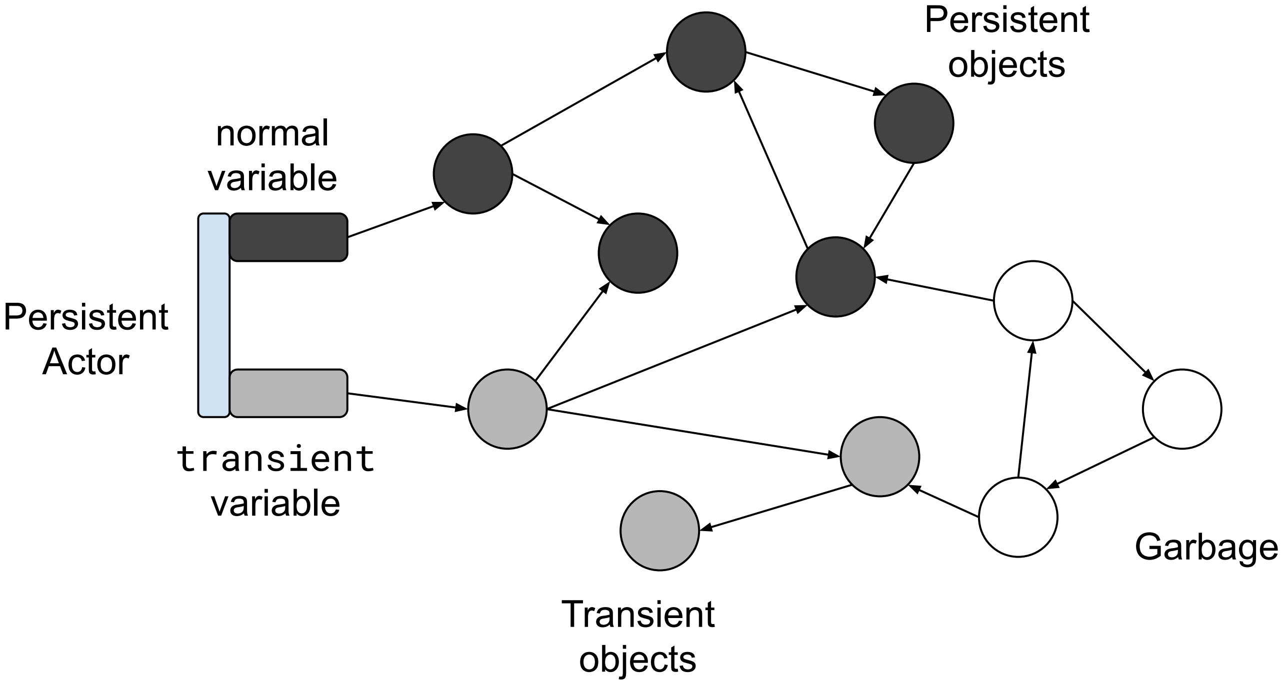
- Automatic migration for defined changes
    - Add actor variables, add options, Nat -> Int, …
- Custom migration logic for complex changes

```
(with migration = convert)
persistent actor {
  …
};
```

```
with convert(old: OldActor) : NewActor { … }
```

→ Static check of migration compatibility

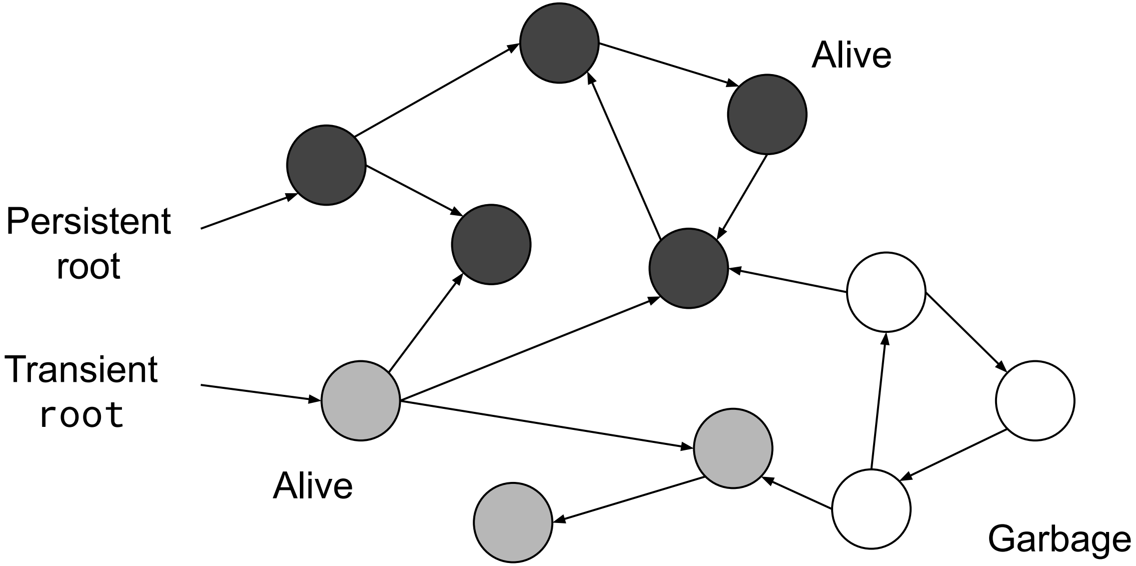
# Transitive Persistence




```
persistent actor Graph {
 type Node = {
   var edges: [Node];
 };
 var start: Node = …;
 transient var temporary : Node = …;
};
```

# Garbage Collection

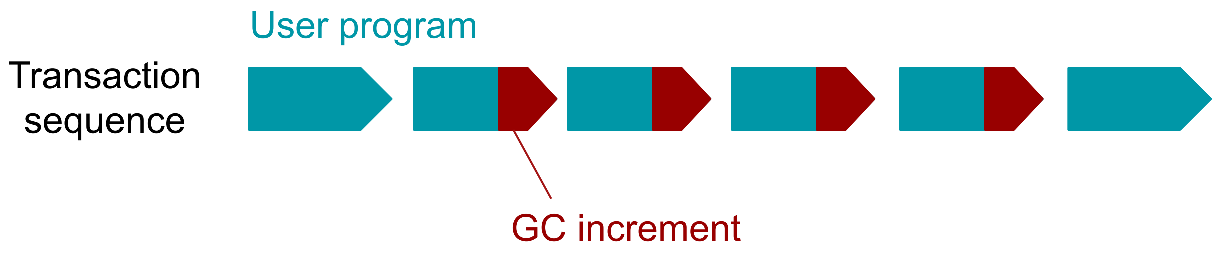
Automatic reclamation of unreachable objects (=garbage) inside the actor



# Motoko's Incremental Garbage Collection

Short bounded interruptions to fit in blockchain transaction

Compacting memory for preventing memory fragmentation




L. Bläser, C. Russo et al. Collecting Garbage on the Blockchain. VMIL 2023.

# Looking At

Simplicity

**Safety**

Protections

1. Type safety
2. Memory safety
3. Arithmetic safety

# 1. Type Safety

Compile-time checks:

- Types inside and across actors
- No dynamic subtype casts
- No null pointer exceptions
- All IC-specific aspects

→ No escape hatches

→ No runtime type errors

~~ClassCastException~~

~~NullPointerException~~

~~Canister ... trapped explicitly:~~
~~Fail to decode argument …~~

# Null Deref Prevention

Explicit use of optional type

Option type

```
func getLastBid() : ?Bid { … };
```

Requires explicit matching and handling of null

```
let minimumPrice = switch (getLastBid()) {
 case null 1;
 case (?lastBid) lastBid.price + 1;
};
```

Exhaustive pattern matching (static check)

# 2. Memory Safety

Managed runtime

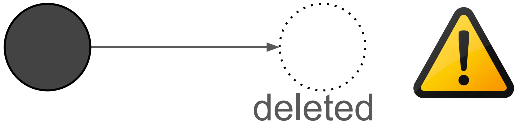
- Garbage collection
- No unsafe raw accesses

No raw secondary storage

- Orthogonal persistence
- Checked migration compatibility

# Risks without Garbage Collection

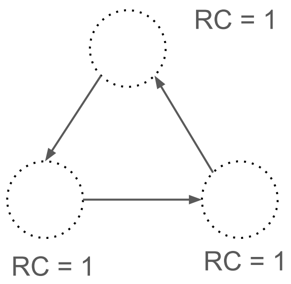
## Dangling Pointer




C++, unsafe code, raw memory

## Memory Leak

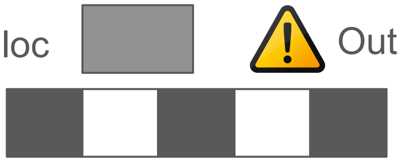
Cyclic reference counting




C++, Rust

## Heap Fragmentation

Alloc



Out of mem

Rust, C++, QuickJS

# 3. Arithmetic Safety

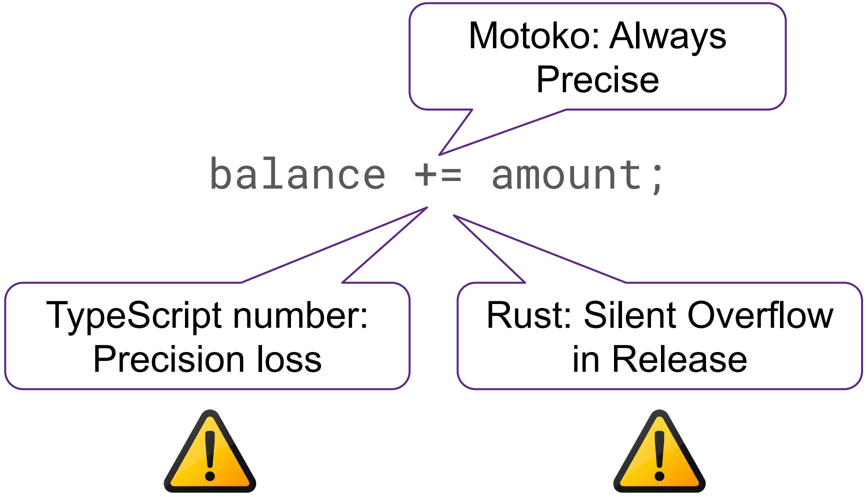
Unbounded integers by default

- Nat, Int

Overflow checks always on

- Nat subtraction
- Bounded integers

No implicit conversions



# Looking At

Simplicity

Safety

Protections

1. Capabilities
2. Authentication
3. Authorization

# 1. Capabilities

- Critical functions require higher privilege
- Privilege must be propagated along call chain

Caller must have this capability

```
module {
 public func standingOrder<system>() {
   ignore Timer.recurringTimer<system>(#days 1, sendMoney);
 };
};
```

Requires system capability

# 1. Capabilities

Prevent supply chain attacks

- Risky library are clearly marked
- Caller must explicitly allow and have capability

Other languages

- Library can issue any IC call
- Rust: Unsafe code can be hidden in safe code ⚠️

# 2. Authentication

Dedicated type for user or actor id

```
var users = Set.empty<Principal>();

public shared (message) func register() : async () {
   let originator = message.caller;
   if (Principal.isAnonymous(originator)) {
     Runtime.trap("Anonymous caller");
   }
   Set.add(users, Principal.compare, originator);
};
```

Public key identifier of caller, e.g. `un4fu-tqaaa-aaaab-qadjq-cai`

# 3. Authorization

Trap if violated

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

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

equivalent

Traps rolls back all changes/effects up to start of public function
(or up to last await point)

# Conclusion

Security needs to cover the **entire software vertical**

- The programming language plays a crucial role

Bespoke language design can severely boost security

- **Simplicity**: Abstractions covering application needs
- **Safety**: Static type checks, rigorous memory safety
- **Protections**: Language-inbuilt security concepts

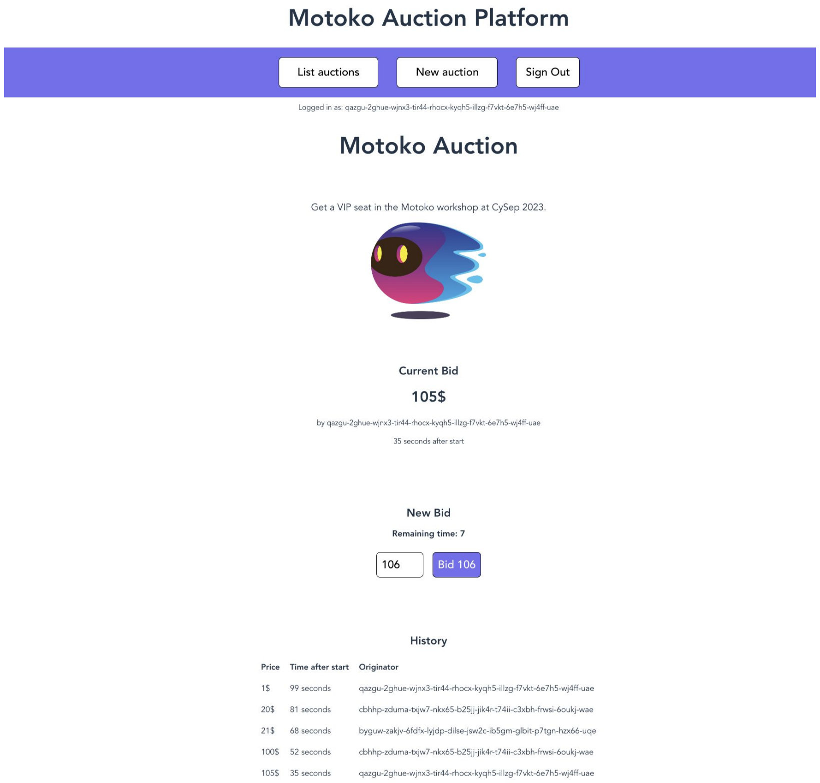
# Upcoming Workshop: Smart Contract Programming

Motoko backend for auction platform:

- Auction bidding
- User authorization
- Auction invariants

Bonus:

- Unpredictable auction ids
- Compare to other languages (Rust and/or TypeScript)



# Motoko Workshop

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

# Learn More

- Motoko Programming Language: https://internetcomputer.org/docs/current/motoko/main/motoko
- Motoko New Base Library: https://dfinity.github.io/new-motoko-base
- Motoko Open Source Repository: https://github.com/dfinity/motoko

# Research Papers


[1] L. Bläser, C. Russo et al. 2024. Smarter Contract Upgrades with Orthogonal Persistence. VMIL 2024. https://doi.org/10.1145/3689490.3690401

[2] L. Bläser, C. Russo, U. Degenbaev et al. Collecting Garbage on the Blockchain. VMIL 2023. https://doi.org/10.1145/3623507.3627672

# 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=2025 }` |
| 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`

<table>
<tr><td><pre>{
 name: Text;
 var year: Nat;
}</pre></td><td><pre>{
 name = "CySeP";
 var year = 2025;
}</pre></td></tr>
<tr><td><code>[var Nat]</code></td><td><code>[var 1, 2, 3]</code></td></tr>
</table>

# 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

# 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

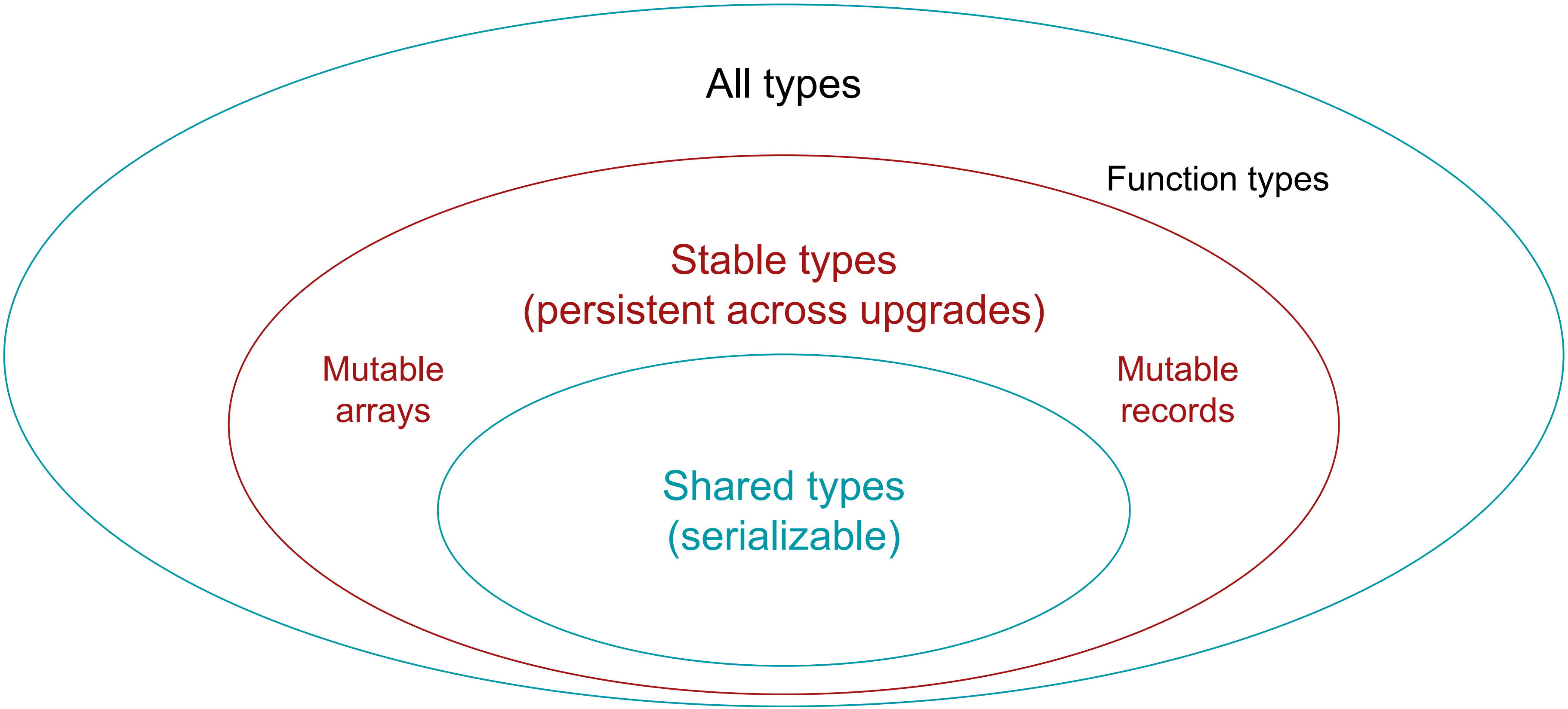
null test with pattern matching

# 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)

# Type Categories



# Modules

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

Base library modules (new version):

| | |
|---|---|
| `"mo:new-base/Principal"` | Authentication (Internet Identity) |
| `"mo:new-base/Runtime"` | Raising errors (traps) |
| `"mo:new-base/List"` | List data structure |
| `"mo:new-base/Map"` | Key-value map data structure |
| `"mo:new-base/Set"` | Set data structure |
| ... | |

# Known Pitfalls

| | |
|---|---|
| Using `await` carelessly | Other async code can run in meantime at await. Beware of race conditions! |
| Forgetting `persistent` modifier | Variable state will be lost on program version upgrade (unless declared `stable`)! |
| Using query functions | Requires a certified variable to be secure. Or needs to be called as replicated query. |
| Public actor functions without return type | One-way calls (“fire and forget”), no propagation of errors. Specify return type `async()` and await. |

Working on improving this