Dagstuhl 25241

Motoko's Enhanced Orthogonal Persistence

Claudio Russo, DFINITY, 13/06/2024

(Slides pilfered from Luc Blaeser)

Smarter Contract Upgrades with Orthogonal Persistence

Luc Bläser, Claudio Russo, Gabor Greif, Ryan Vandersmith, Jason Ibrahim

DFINITY Foundation

VMIL 2024, 20 Oct 2024, Pasadena, CA, USA

Smarter Contract Upgrades with Orthogonal Persistence

Luc Bläser
luc.blaeser@dfinity.org
DFINITY Foundation
Switzerland

Claudio Russo
claudio.russo@dfinity.org
DFINITY Foundation
Switzerland

Gabor Greif
gabor.greif@dfinity.org
DFINITY Foundation
Switzerland

Ryan Vandersmith
ryan.vandersmith@dfinity.org
DFINITY Foundation
Switzerland

Jason Ibrahim
jason.ibrahim@dfinity.org
DFINITY Foundation
Switzerland

Abstract

Altering the smart contract deployed on a blockchain is typically a cumbersome task, necessitating a proxy design, secondary data storage, or the use of special APIs. This can be substantially simplified if the programming language features orthogonal persistence, automatically retaining the native program state across program version upgrades. For this purpose, a customized compiler and runtime system needs to arrange the data in a self-descriptive portable format, such that new program versions can pick up the previous program state, check their compatibility, and support implicit or explicit data evolutions. We have implemented such advanced persistence support for the Motoko programming language on the Internet Computer blockchain. This not only enables simple and safe persistence, but also significantly reduces the cost of upgrades and data accesses.

CCS Concepts: • **Software and its engineering → Runtime environments**; • **Information systems → Main memory engines.**

Keywords: Orthogonal Persistence; Smart Contract Upgrades; Blockchain; WebAssembly

ACM Reference Format:
Luc Bläser, Claudio Russo, Gabor Greif, Ryan Vandersmith, and Jason Ibrahim. 2024. Smarter Contract Upgrades with Orthogonal Persistence. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '24), October 20, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3689490.3690401

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
VMIL '24, October 20, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1213-5/24/10
https://doi.org/10.1145/3689490.3690401

1 Introduction

Modern blockchains, like the Internet Computer [2, 11], establish a secure and distributed virtual machine for running complex programs, such as smart contracts, decentralized applications, or other software solutions. Such blockchain programs can be implemented in a Turing-complete high-level programming language, such as for example the blockchain-tailored languages Solidity [10] and Motoko [18], or mainstream languages like Rust, JavaScript, Python, and others.

Even when deployed on a blockchain, there typically comes a time when a program needs to be changed, be it for feature extensions, improvements, or bug fixes. This requires a mechanism to upgrade a program's code, while retaining its state, replacing an existing version by a new version that implements the desired change. Unfortunately, such support is typically lacking or poor, requiring programmers to apply "creative" alternative solutions or implementing cumbersome storage management. On blockchains, like Ethereum [10], programmers usually prepare proxies to enable upgrades by changing the redirection target [17]. More advanced blockchains support a dedicated upgrade mechanism [13]. However, the integration in the programming language is still influenced by the traditional computer architecture, where the program has a main memory that is lost on an upgrade. As a consequence, the blockchain often exposes extra secon
can be stor
complicates
mance disa
detail in Se

In this w
grades on a
formance, a
program sta
of the langu
persisted or
gram versio
persistence
definitely, it

The Internet Computer

- decentralized, general-purpose compute platform
- (sharded) blockchain
- uses Wasm as VM
- multi-language (mostly Rust and Motoko, some C++, JS/Python)
- relies on deterministic Wasm execution for consensus
- persists Wasm state between transactions
- mostly Wasm 1.0 features (can’t easily use Wasm GC)
- “Gas” model
- can serve full web applications, all stored “on chain”
- supports user code upgrades

Internet Computer “Canisters”

- actors by another name
- shared nothing
- communication is asynchronous
- strongly typed interfaces (clean Candid IDL (Andreas/Joachim/Yan Chen)
- messages/upgrades processed transactionally, rollback on failure
- gas limits per message/upgrade
- main memory (think RAM, up to 4GB)
    - persisted between messages
    - discarded on upgrade (think RAM)
- stable memory:
    - 64-bit Wasm memory (up to 600GB)
    - retained on upgrades (think DISK)

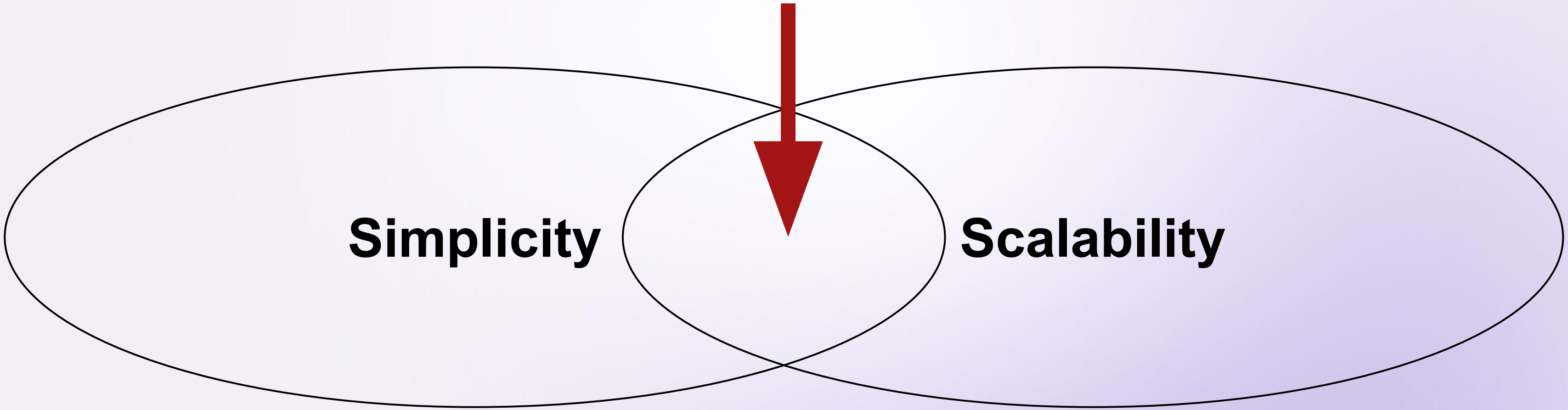
(Feature set before EOP work)

Motoko in a nutshell

- Actor oriented, impure, functional, strict
- General purpose (not your typical blockchain language)
- Strongly typed (generics, structural subtyping, pattern matching, safe arithmetic, no escape hatches, no dynamic casts, no pervasive null)
- Impure, functional and strict
- Async/await with futures (selective CPS)
- Designed for the Internet Computer (also actor based)
- Orthogonal persistence (state preserved between messages)
- Live upgrades with state preservation and extension
- Seamless, efficient Candid interop (hidden from users)
- Incremental GC - now on our 4th GC… (of linear memory)
- Compiled

Designed to showcase the Internet Computer - gateway drug for mere mortals with JS/TS background. Safety First!

Goal: Better Persistence for Motoko



Liberate developers from dealing with stable memory

Support data volumes as large as in stable memory

Smart Contract Upgrades

Program version 1

```
…
var messages : List<Text> = …;
…
```

Program version 2

```
…
var logs : List<(Text, Time)> = …
…
```

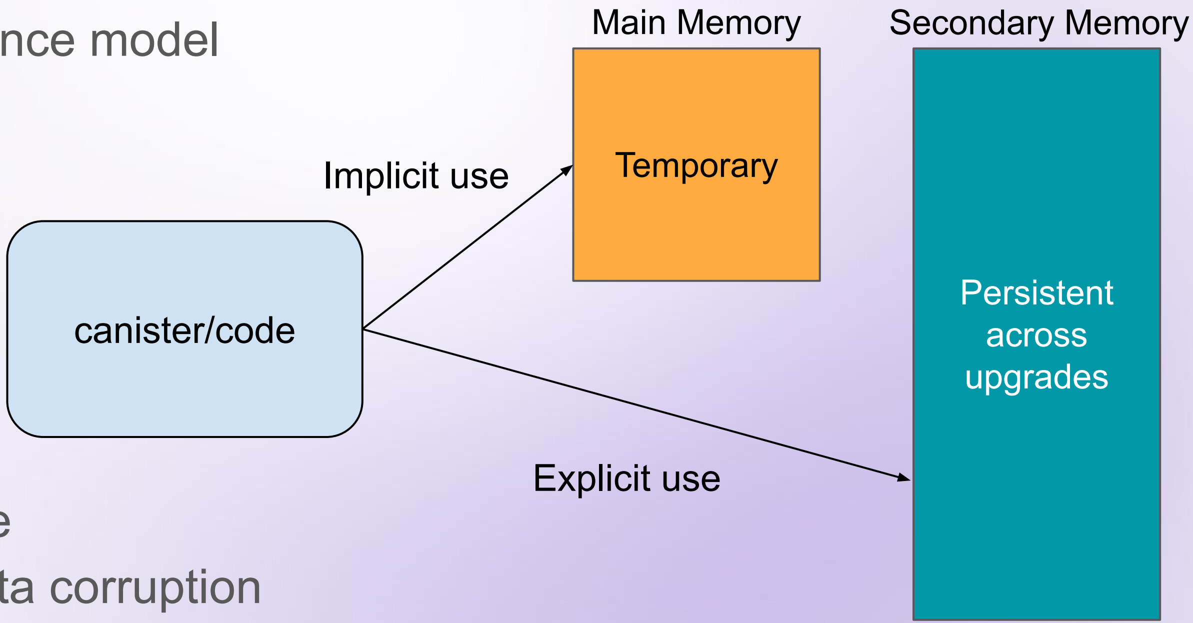
Change program logic

Migrate program data

Not that easy

Upgrades with Secondary Storage

Dedicated upgrade mechanism
but using classical persistence model



- ❌ Extra code
- ❌ Risk of data corruption
- ❌ Expensive copying

Motoko's Enhanced Orthogonal Persistence

Automatically persist canister state **across upgrades:**

- Using the standard inbuilt language concepts
- Supporting arbitrary program data structures
- No database, secondary memory, special APIs

➕

- Supporting automatic and custom data migration
- Ensuring data consistency on upgrades
- Fast upgrades and data accesses

Motoko Example

persistent

```
persistent actor {
    var messages = List.nil<Text>();

    public func log(t : Text) {
        messages := List.push(t, messages);
    };

    public func readLast(count : Nat) : async [Text] {
        List.toArray(List.take(messages, count));
    };
};
```

Upgrade

Retained on upgrade

Initializer does not run

```
persistent actor {
    var messages = List.nil<Text>();
    var times = List.map<Text, Time>(messages, func e { 0 });

    public func log(t : Text) {
        messages := List.push(t, messages);
        times := List.push(now(), times);
    };

    public query func readLast(count : Nat) : async [Text] { … };
    public query func readUntil(t0 : Time) : async [Text] { … };
};
```

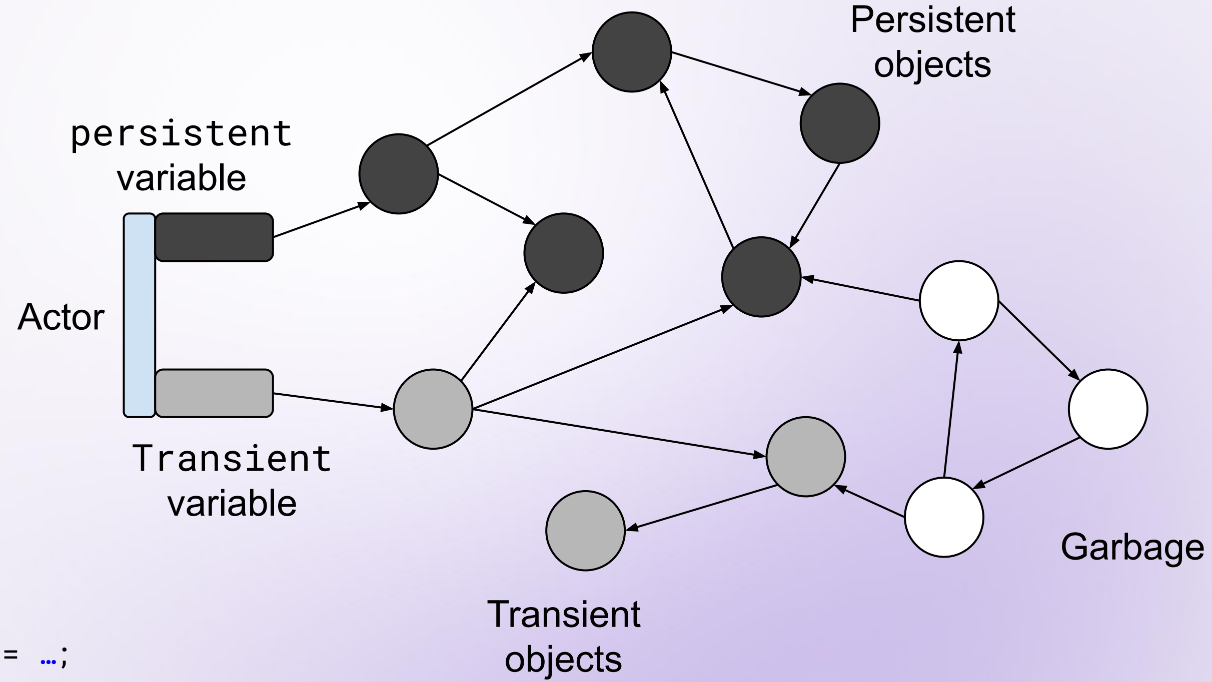
New

persistent keyword

Applied to actors

Transitive persistence

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



Stable Types

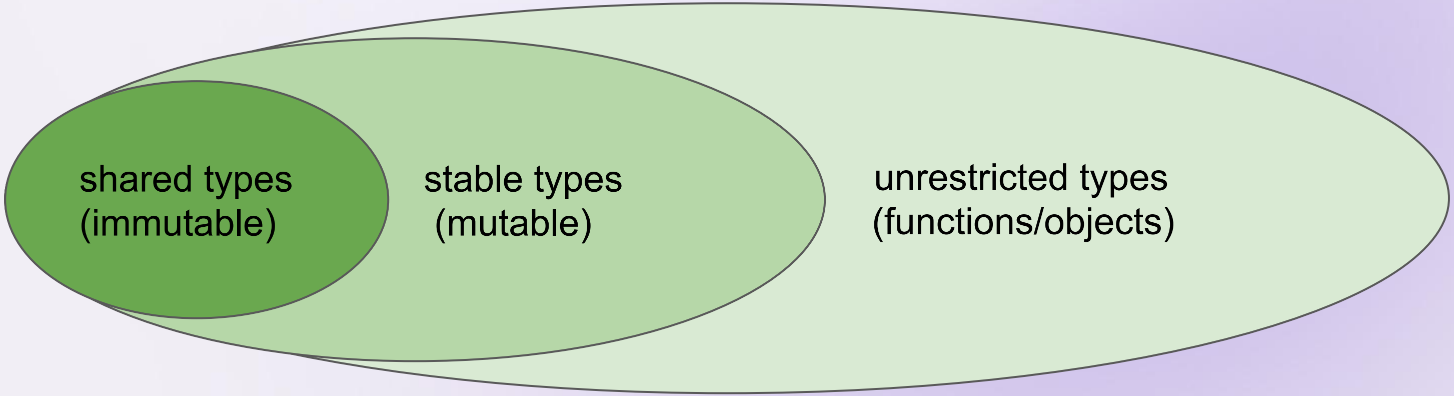
Not all types are suited for persistence across upgrades

🚫 Local function references, lambdas

🚫 Async futures

Restrict persistent variables to **stable** types (statically checked)

(`transient` variables have unrestricted types)



Data Evolution

Implicit migration: Automatically handled

✅ Add actor fields
✅ Nat -> Int
✅ Add cases to variant types
✅ Promote type of fields/variants
✅ (fancy) finite unfolding-> recursive type

(Basically anything compatible with loss-less subtyping.)

Explicit migration: Any more complex case (dropping data, refactoring data)

Explicit Migration

```
persistent actor {
 var messages : List<Text> = ...;
 var times : List<Time> = ...;
}
```

Upgrade

```
persistent actor {
 var messages : List<Text> = ...;
 var times : List<Time> = ...;

 var logs = List.zip(messages, times);
}
```

Upgrade

```
persistent actor {
 var logs : List<(Text, Time)> = List.nil();
}
```

Explicit Migration with Migration Function

```
persistent actor {
 var messages : List<Text> = ...;
 var times : List<Time> = ...;
}
```

Upgrade

```
(with migration =
   func ({messages : List<Text>;
          times : List<Time> }) :
        {logs : List<(Text, Time)>} =
     { logs = List.zip(messages,times} })
persistent actor {
 var logs : List<(Text,Time)> = List.nil();
}
```

Explicit Migration with Imported Migration Function

```
persistent actor {
 var messages : List<Text> = ...;
 var times : List<Time> = ...;
}
```

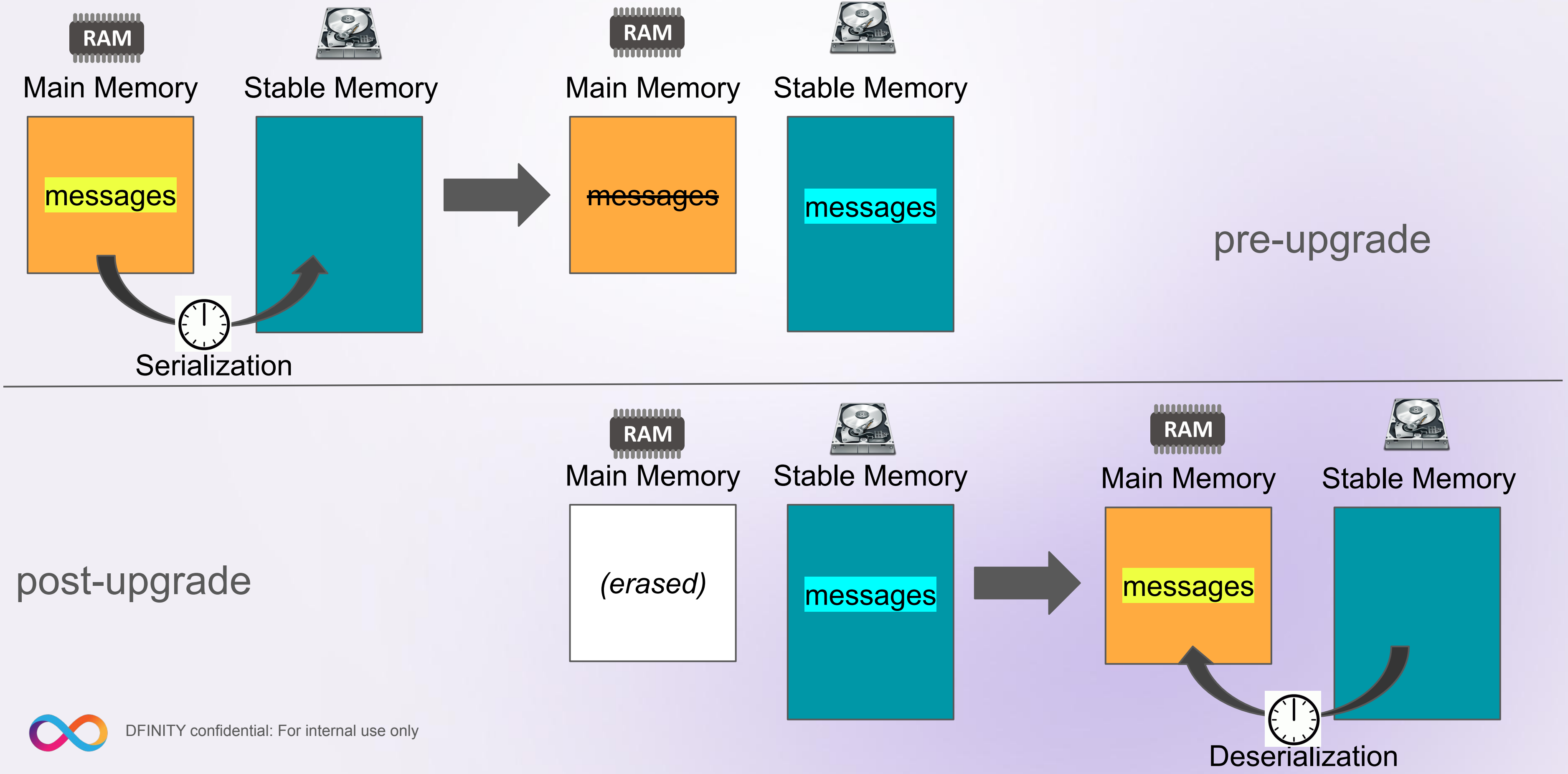
Upgrade

```
import  { migration } “Migration.mo”;

(with migration)
persistent actor {
 var logs : List<(Text,Time)> = List.nil();
}
```

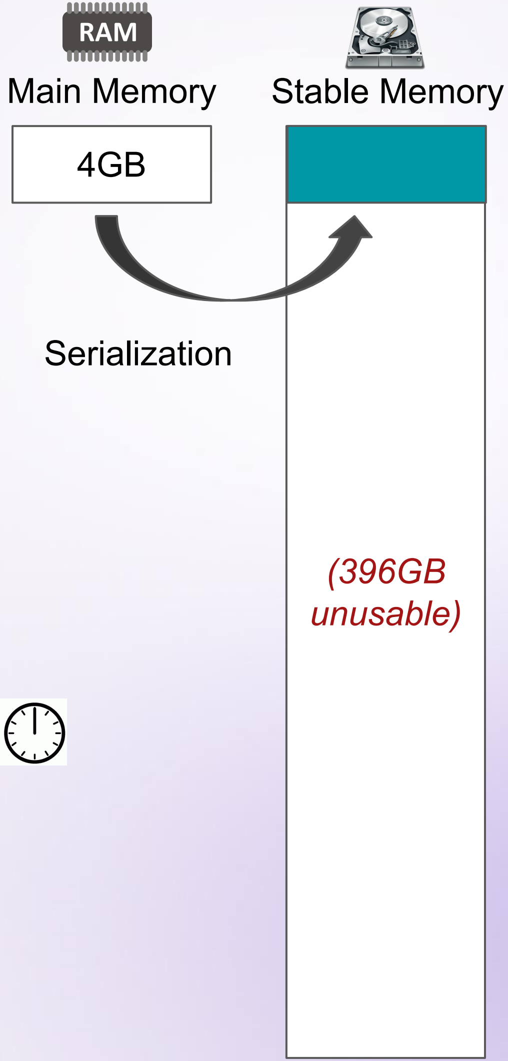
Classic Solution

Old Motoko Upgrade Mechanism



DFINITY confidential: For internal use only

Scalability Limits



Instruction Limit

Serialization is too expensive

Out of Memory

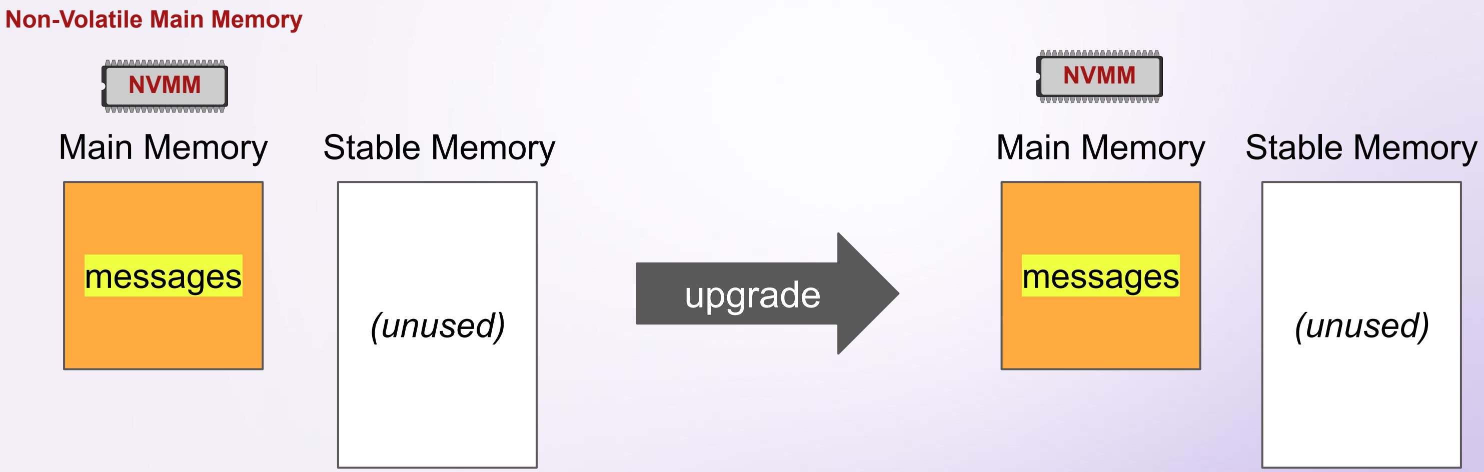
Cannot pass beyond 4GB

DFINITY confidential: For internal use only

EOP Solution

DFINITY confidential: For internal use only

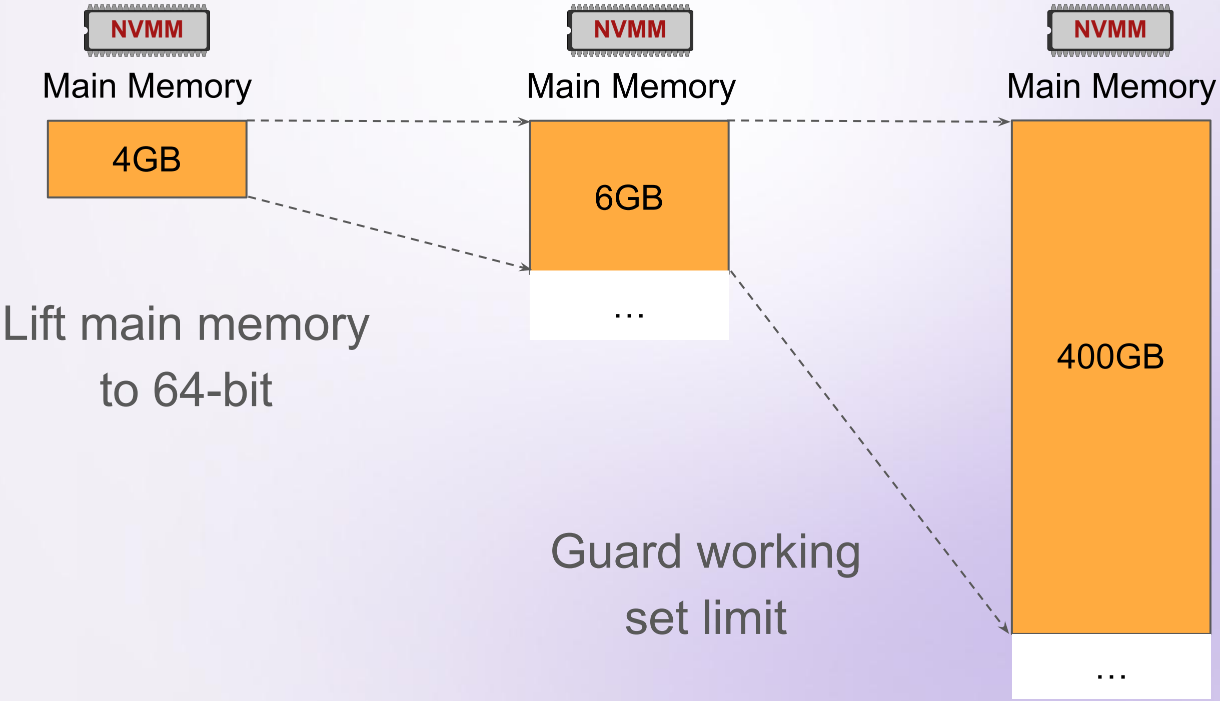
Upgrade without Serialization & Stable Memory



Keep main memory state on upgrade:
→ New programs resume from there
No serialization:
→ Constant-time upgrades

DFINITY confidential: For internal use only

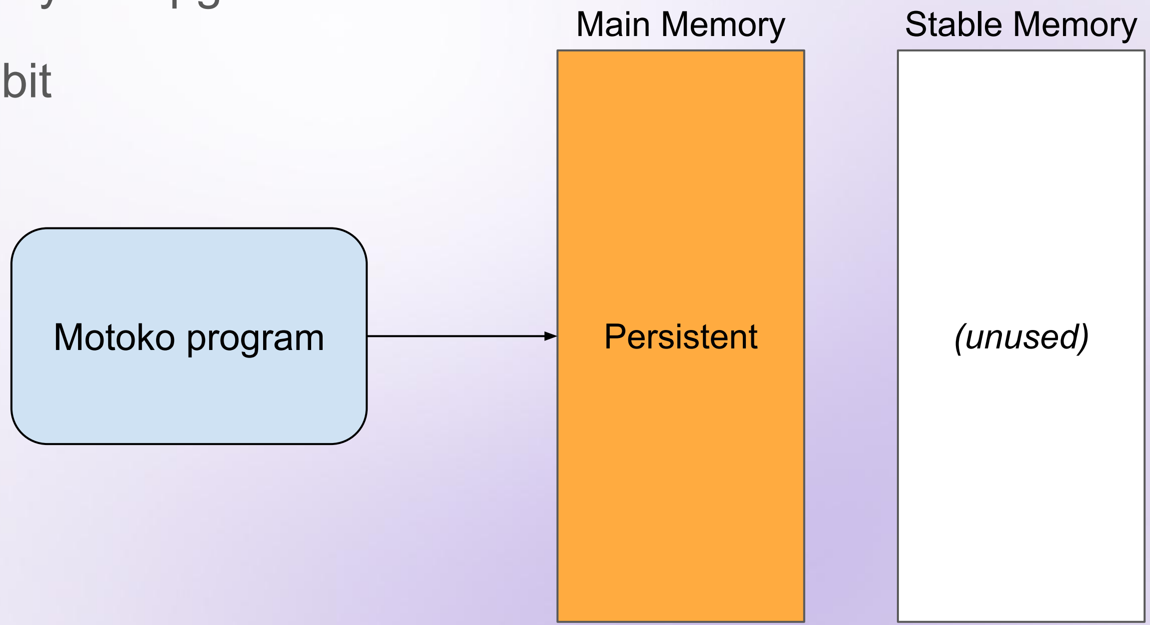
Pass the 4GB Limit



ICP

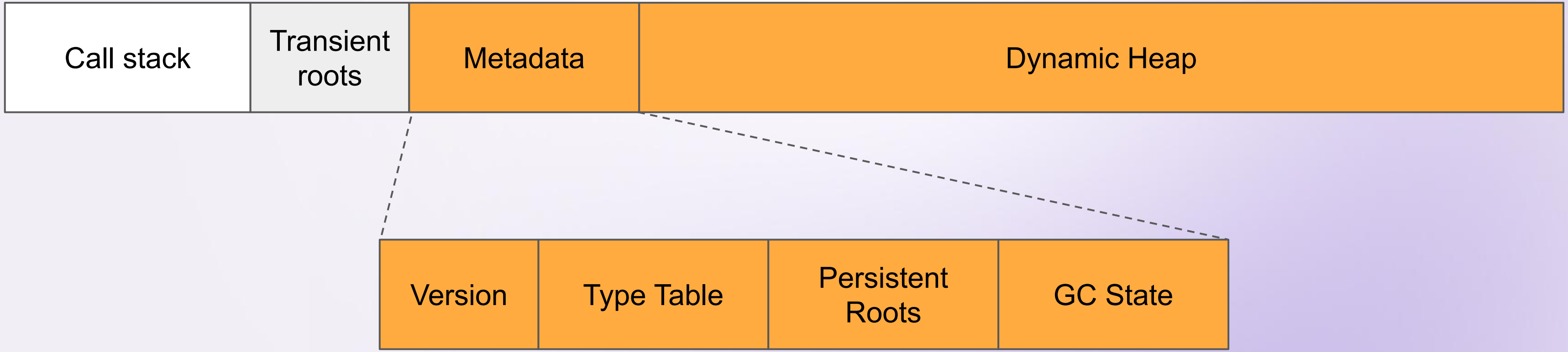
Optionally retain main memory on upgrade

Extend main memory to 64-bit



Compilation-Invariant Memory Layout

New program version picks up existing memory



Reused on upgrade

No Static Allocations

Use Wasm passive data segments

- Can be loaded at runtime
- To a dynamic address on the heap

Pool compile-time known objects

- In the heap
- Access via transient root

Incremental Garbage Collector

Persist the entire GC state on upgrade

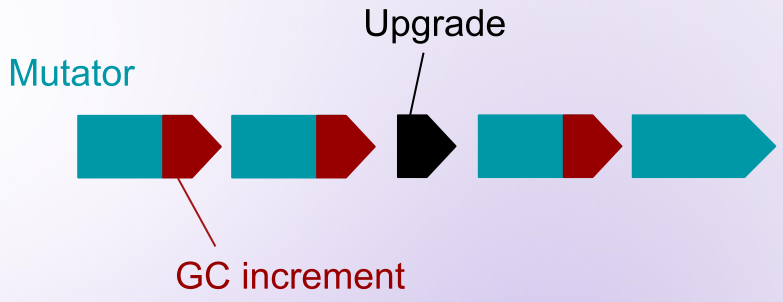
- New version resumes active GC run

Scales to large heaps

- No need to complete GC before upgrade

Roots:

- Persistent: actor fields
- Transient: `transient` actor fields, continuations

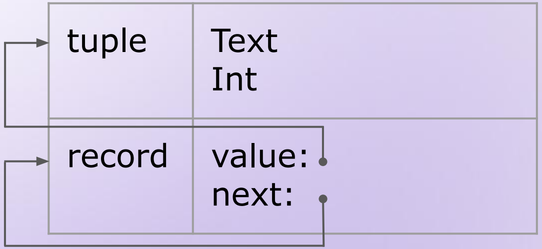


Memory Compatibility Check

Check compatibility when loading new program version

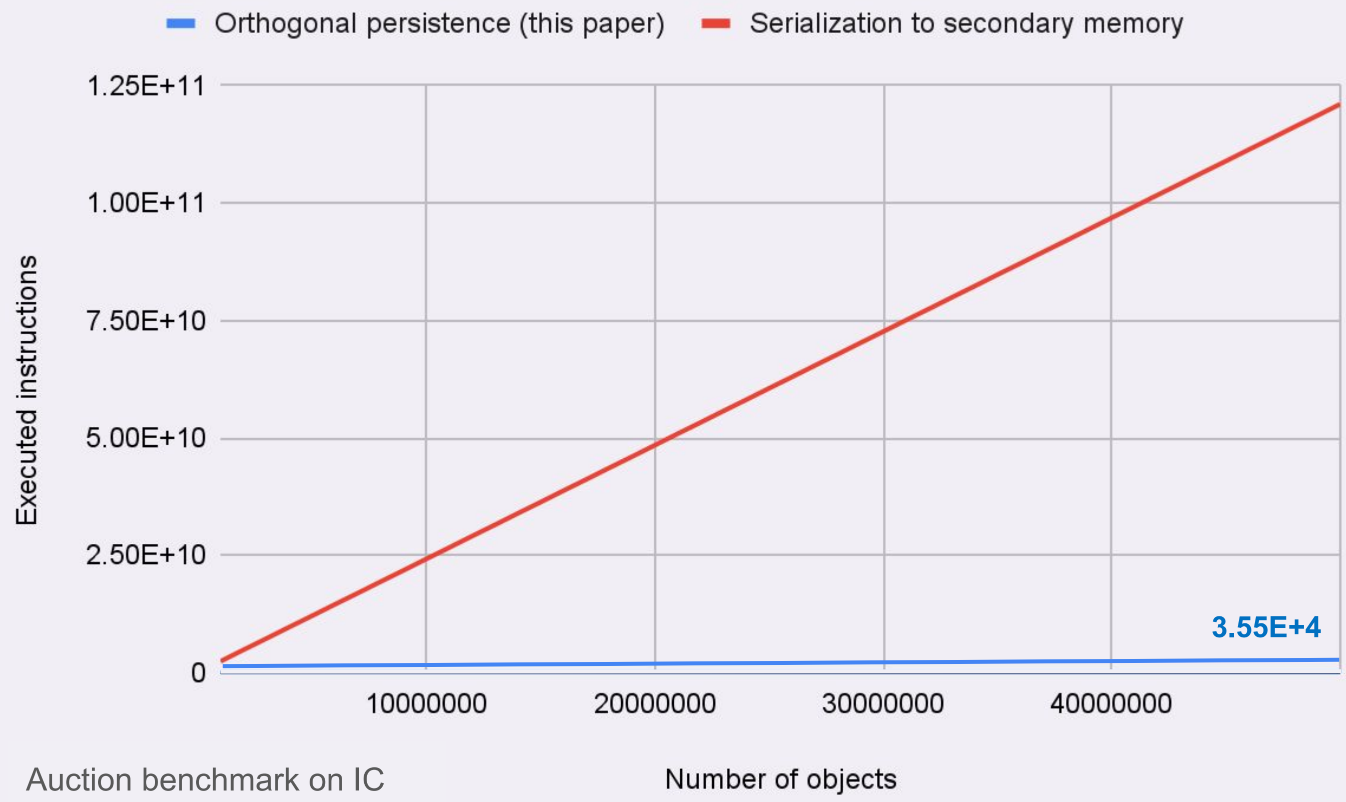
- Store types of old program version in heap
- Compare against types of new Wasm binary
- Reject upgrade if incompatible

Type table



🚀 Check depends on (small) #types, not (large) #objects

Motoko Upgrade Costs



Auction benchmark on IC

Code Complexity

Rust: Stable Data Structures

```
...
type Memory = VirtualMemory<DefaultMemoryImpl>;
#[derive(CandidType, Deserialize, Clone)]
struct Auction {
   item: Item,
   bid_history: Vec<Bid>,
   remaining_time: u64,
}
impl Storable for Auction {
   fn to_bytes(&self) -> std::borrow::Cow<[u8]> {
       Cow::Owned(Encode!(self).unwrap())
   }
   fn from_bytes(bytes: std::borrow::Cow<[u8]>) -> Self {
       Decode!(bytes.as_ref(), Self).unwrap()
   }
   const BOUND: Bound = Bound::Unbounded;
}
thread_local! {
   static MEMORY_MANAGER: RefCell<MemoryManager<DefaultMemoryImpl>> =
   RefCell::new(MemoryManager::init(DefaultMemoryImpl::default()));
   static STABLE_AUCTIONS: RefCell<StableBTreeMap<AuctionId, Auction,
      Memory>> =
       RefCell::new(
           StableBTreeMap::init(
               MEMORY_MANAGER.with(|m| m.borrow().get(MemoryId::new(0))),
           )
       );
} ...
```

Motoko: Orthogonal Persistence

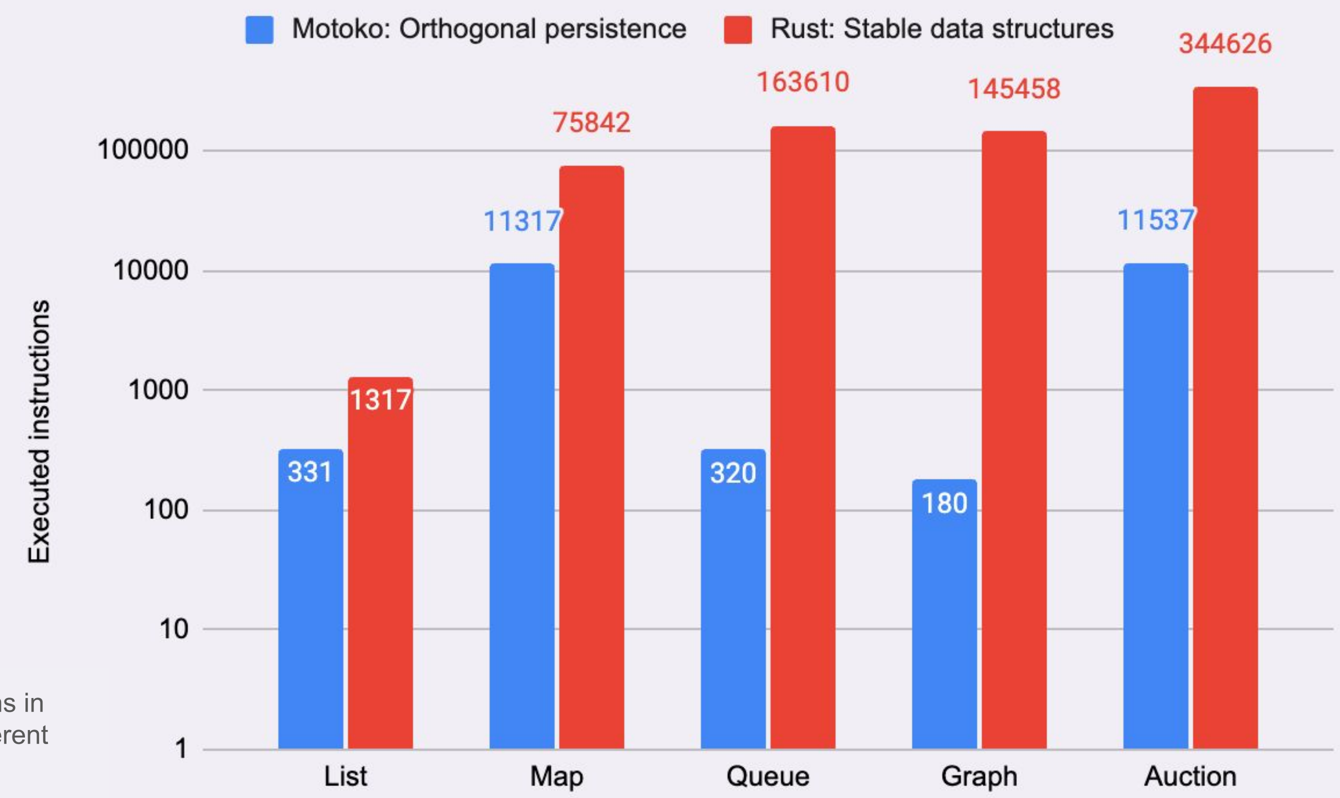
```
persistent actor { ...
   type Auction = {
       item : Item;
       bidHistory : LinkedList.LinkedList<Bid>;
       remainingTime : Nat;
   };
   let auctions = Tree.new<AuctionId,Auction>();
   ...
};
```

Persistence handling

Element Lookup Costs

Huge difference:

- ✅ Free choice of data structure
- ✅ No expensive deserialization
- ✅ Directly navigate over references, no lookup via id
- ✅ No API call to persistent memory



Analogous implementations in Motoko and Rust with different persistence mechanisms

Conclusion

Canister upgrades can be simple, safe, and cheap!

- with enhanced orthogonal persistence

Demonstrated by Motoko:

- persistent data of almost any shape
- implicit data migration when possible
- explicit data migration when needed
- compatibility check on upgrade
- fast, constant-time upgrades

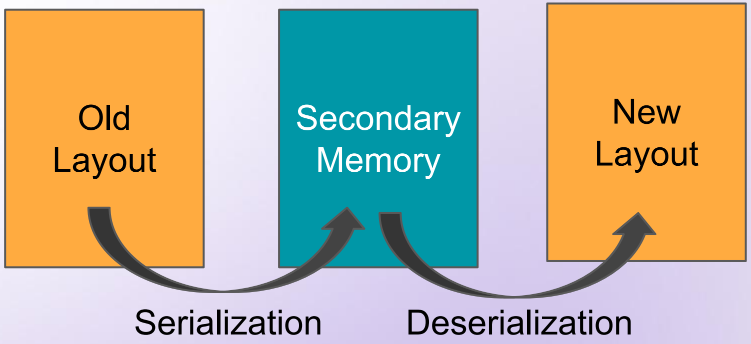
Last Resort Migration Path

When changing the memory layout in future (rare occasion)

Alternative, more expensive mechanism:

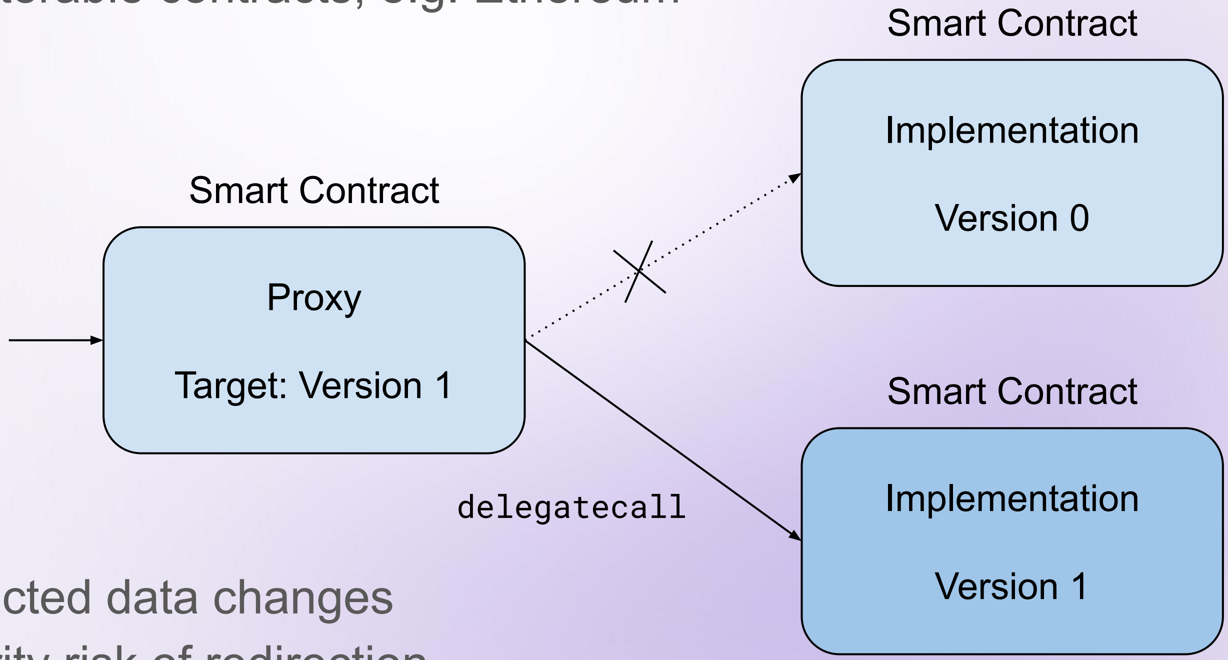
- Serialize objects to secondary memory
- Using a long-term portable format
- Deserialize back to main memory

Split work in multiple transactions, if needed



Upgrades by Call Redirection

Workaround for non-alterable contracts, e.g. Ethereum



❌ Restricted data changes

❌ Security risk of redirection

Upgrades with Special Data Structures

DB-like persistent data structures

```
static LOGS: RefCell<StableVec<Text, Memory>> =
  RefCell::new(StableVec::init(...));

impl Storable for Text {
    ...
}
```

❌ Fixed set of data structures
❌ Object-database mapping complexity
❌ Serialization overheads

Motoko Example (classic syntax)

persistent

```
actor {
  stable var messages = List.nil<Text>();

  public func log(t : Text) {
    messages := List.push(t, messages);
  };

  public func readLast(count : Nat) : async [Text] {
    List.toArray(List.take(messages, count));
  };
};
```

Why new syntax?

In Motoko, actor fields can be transient (flexible) or persistent (stable).

In the classic syntax:

- actor fields **opt-in** to persistence: transience is the default.

In the new syntax:

- actor fields **opt-out** of persistence: persistence is the default.

(For possibly valuable data, persistence is the safest option.)

Upgrade

Retained on upgrade

Initializer does not run

```
actor {
    stable var messages = List.nil<Text>();
    stable var times = List.map<Text, Time>(messages, func e { 0 });

    public func log(t : Text) {
        messages := List.push(t, messages);
        times := List.push(now(), times);
    };

    public query func readLast(count : Nat) : async [Text] { … };
    public query func readUntil(t0 : Time) : async [Text] { … };
};
```

New