Verifying Distributed Systems with Stainless

Master Thesis Defense

Author:
Stevan Ognjanovic

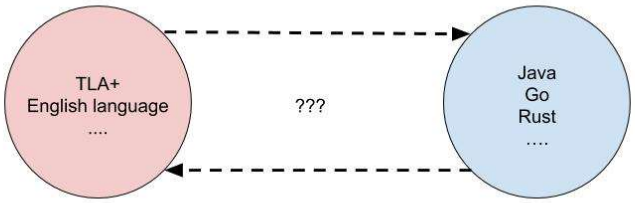
Supervised by:
Professor Viktor Kuncak
Dr. Zarko Milosevic

In this work

- Scala implementation of Light Client
- Stainless verified
  - Blockchain model
  - Core Verification model
- Non verified parts:
  - Network IO
  - Hashing and Cryptography
  - Light Client with Fork Detection

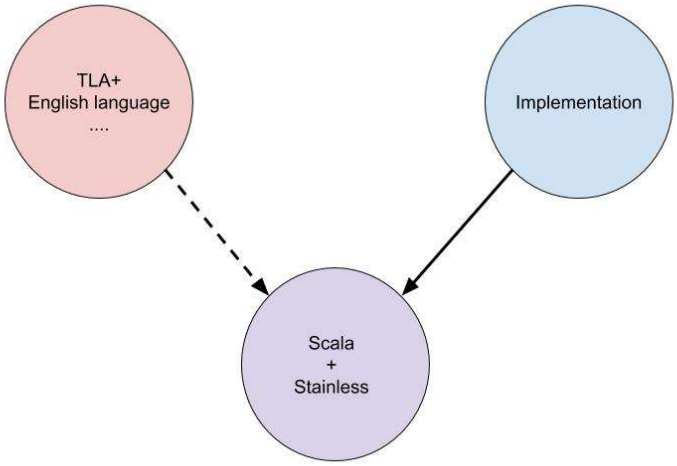
Based on the documentation available at the time as this is part of an ongoing effort
We did not verify everything, scope and feasibility. Hashing, crypto, network, etc.
Using external code we managed to implement an executable implementation

Synchronizing Implementation and Specification



Writing the specification is hard
Other than “inspiring” the implementation there is no hard link between the two.
Implementation introduces a new set of challenges.

Formal Verification of Software



Implementation is moved to a domain in which it is possible to statically verify the implementation.

Why Scala?

- Scala used by: Twitter, Netflix, Goldman Sachs, SAP, Spotify, Guardian,…
- Backbone of:
  - Apache Spark
  - Play Framework
  - BlinkDB

JVM base, implementations can be used with any other java-based language
Extensively used in industry
There is also GraalVM, ScalaNative, ScalaJS

Stainless

- Tool for formally verifying programs written in a subset of Scala
- Uses modern SMT solvers, Z3 and CVC4
- Errors caught at the level of implementation
- Harder to diverge from the formal specification
- Software design is influenced by formal verification

Light Client Core Verification

- Key component of the Light Client
- Uses failure model of the blockchain to validate the data received
- Implemented in Scala and formally verified with Stainless
- Rust implementation developed in parallel by Informal Systems

TLA+ specification

- Specification written by:
  - Dr. Igor Konnov – INRIA and Informal Systems
  - Dr. Josef Widder – TU Wien and Informal Systems
- Split in two parts
  - Blockchain module
  - Core Verification module
- Ongoing effort to specify rest of the Light Client

Fork Detection and Evidence Submission are not modeled in TLA+
Ongoing effort

Formal Verification of Core Verification

- TLA+:
  - Not runnable
  - Simplified
  - Access to global state, with no obvious penalties related to maintainability
  - Bounded model checking
- Scala implementation:
  - Runnable
  - Verified with Stainless
  - Conformance tested
  - Proofs are for any configuration of the blockchain

Just a model,

Implementation Decomposition

- Proving properties of the implementation achieved through proof decomposition
- Design by Contract or Liskov substitution principle
- Correct by construction types

Design by Contract

- In literature also known as Liskov substitution principle (L in SOLID)
- "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."
- Stainless can prove the above statement
- Main design approach in this project
- Also used for hiding external code

https://en.wikipedia.org/wiki/SOLID

Design by Contract

```
1  abstract class NextHeightCalculator {
2    @pure
3    def nextHeight(bottom: Height, top: Height): Height = {
4      require(bottom + 1 < top)
5      ??? : Height
6    }.ensuring(res => bottom < res && res < top)
7  }
```

Given two types, return something in between.
We are using it to calculate the next data to be retrieved

Correct by Construction Types

- When dealing with concrete implementations types are restricted
- Types are restricted using invariants over the class fields
- Stainless proves that instances are always constructed with valid values

```
1   sealed case class Height(value: BigInt) {
2     require(value > BigInt(0))
3     ...
4   }
```

Height is the simplest example of a contract

Combining the two

```
case class InMemoryFetchedStack(
    override val targetLimit: Height,
    pending: List[LightBlock])
      extends FetchedStack {
  require(
    pendingInvariant(pending) &&
    pending.forall(_.header.height <= targetLimit))
  ...
}
```

Contract is inherited from FetchedStack
Invariant internal to the type, necessary for proving the contract.
pendingInvariant enforces increasing height for the top of the stack to the base of the stack
All heights need to be lower or equal than the targetLimit

Pushing down properties

- Protocol properties in TLA+ specification are specified at the top level over traces constructed during model checking
- Specified over bare types like sequences and sets
- When using Stainless invariants are pushed to the type level
- Correct behavior achieved by composition of verified types instead of behavior inference during model checking

Correctness of types is enforced through validity of constructor parameters and transitions on methods.
Proved termination under assumptions related to external code
Successful outcome can only happen when a target is reached
We are hiding external code behind abstractions because we want to provide flexibility

Termination with Stainless

- Numerical values assigned to recursive calls
- Calculated based on input parameters of a function
- Stainless can infer measures in most situations
  - We rely on this most of the time
- Measure needs to have two properties:
  - Non-negative
  - Between two recursive calls the measure must decrease

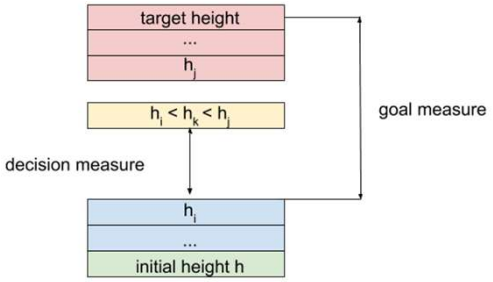
In general, termination is based on termination of recursive functions.
Composition of terminating recursive functions also terminate

Core Verification Termination

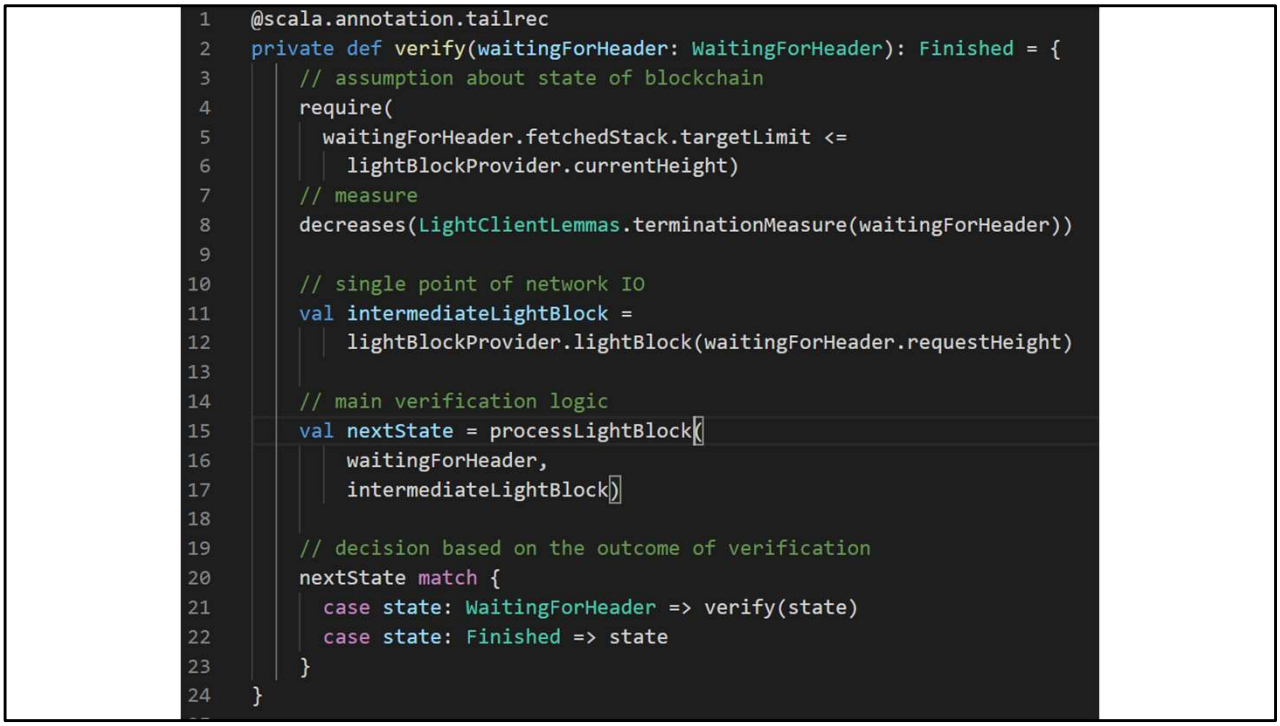
- Assumptions:
  - Network requests do not block indefinitely
  - Request for data at a certain height will result in a response for that height
- With these two assumptions we can prove Termination of Core Verification
- Termination in the TLA+ was defined as a deadlock on the end state which should be manually inspected

Core Verification Termination Measure

- Two-part measure
- Goal measure: distance to reaching target height
- Decision measure: distance before a decision must be made
- Main property of the implementation
- Influences most of the code in way or the other



Goal measure is intuitively clear, we are measuring how close we are to confirming the target
Decision measure, algorithm might need to download intermediate information and during that time the first part is fixed.
In this situation we are measuring how close we are to reaching adjacent data when we must decide



```scala
@scala.annotation.tailrec
private def verify(waitingForHeader: WaitingForHeader): Finished = {
  // assumption about state of blockchain
  require(
    waitingForHeader.fetchedStack.targetLimit <=
      lightBlockProvider.currentHeight)
  // measure
  decreases(LightClientLemmas.terminationMeasure(waitingForHeader))

  // single point of network IO
  val intermediateLightBlock =
      lightBlockProvider.lightBlock(waitingForHeader.requestHeight)

  // main verification logic
  val nextState = processLightBlock(
      waitingForHeader,
      intermediateLightBlock)

  // decision based on the outcome of verification
  nextState match {
    case state: WaitingForHeader => verify(state)
    case state: Finished => state
  }
}
```

Require, blockchain is sufficiently height
Custom measure
There is only one point of network IO which allows easier reasoning
processHeader, does the main logic related to verification, this where the measure decreases
perocessHeader is proved to do some useful work and that it terminates
Recursive call or exit

Verification Driven Development

- Proposed by Informal Systems:
  - Specification first
  - Implementation later
  - Cross-pollination between the two is optional
- Challenging implementation details are not relevant

Verification Driven Development with Stainless

- Verification moves from non executable to executable domain
- Architectural decisions are directly influenced by verification
  - Implementations are significantly different between Go, Rust and Scala
- Cross-pollination between specification and implementation is now forced
- Insights from this work are now an active part of the current effort

Specification and implementation are interconnected explicitly
Some differences rust from different programing paradigms

Implementation Interoperability

- Can be used in combination with any Scala library
- Code designed for extension
- Non verified abstractions should be implemented correctly
- Non verified implementations
  - IO layer uses - sttp and circe
  - Caching of received data - Caffeine
  - Cryptography - Google Tink
  - Hashing - Java Security
  - ProtoBuff – needed for hash computation

Correctness

- Correct behavior achieved through composition of abstractions
- Errors in semantics are impossible to catch:
  - Checks which are based on the blockchain failure model can not be verified
  - External erroneous code can break the implementation
- Testing is still necessary

Testing

- ScalaTest
- tendermint-rs provides conformance tests
- 0.33.5 docker container used for end-to-end usability test
  - Can be used to communicate with a tendermint node
- Testcontainers for Scala used to simplify the process

N-version programming

- Parallel efforts to implement the same functionality in Scala and Rust
- “Stevan test” for Rust Light Client:
  - Jokingly named by Ethan Buchman, CTO Informal Systems
  - Some bugs were found in the implementation

Comparing behavior for the same conformance tests
Trying to understand what was implemented in Rust

Implementation statistics

Module	Lines of code
light-client-core	1957
light-client	1000
tendermint-general	593

Light-client-core, core verification and blockchain model
Light-client, with fork detection and the rest of the implementation
tendermint-general – communication with tendermint, hashing, crypto, etc.

Verification Statistics

Number of VCs	GitHub	P51	P51 with cached VCs
578	153.843	116.03	21.375

- Stainless works with sources, increases compilation times
- Stainless can cache results of previous verifications
- Can be run in watch mode, which reduces the overhead

Users are used to working with jars, for faster workflow and usually compilation is incremental
Caching can help as it reduces the time of verification
Checks which are not automatically inserted

Limitations of the implementation

- Implementation is not benchmarked
- Large parts of code are external?
- Blockchain and Core Verification model are not connected

Is really fact that there is external code a problem, you could use Stainless to strengthen you code

Conclusion

- Proving properties is still hard
- 100% verified software is hard to achieve as working with existing libraries is almost unavoidable
- Currently:
  - Recompilation of sources can be very frustrating
  - Limited support for verified libraries
- Hard to share libraries:
  - TASTy, prof carrying intermediate representation?
- Overall it is possible to utilize Stainless for verifying software

External code is necessary given the usual dependency graph
Implementation needs to be adapted to work with Stainless, but it is attainable

Thank you for your attention

Appendix

- More code examples for question time

Protocol Specifications

- Problems:
  - Tedious and very involved (Paxos)
  - Error prone (original Pastry)
  - Implementations can diverge and have errors
- Verification of specifications:
  - Reduces errors in specification
  - Gives new insights

https://www.sciencedirect.com/science/article/pii/S0167642317301612

- Protocols require a lot of effort from different people to get to a usable protocol. For example TLS is going through several versions
- original pastry protocol had bugs
- Implementing those protocols is a totally different problem
- To help with at least the second one and reduce the burden of first verification of specification

TLA+ (1)

- Language for specifying protocols
- Used for model checking specifications
- Greatly reduces the chance of errors occurring in specifications
- Is it a solution?

Temporal Logic of Actions
By Leslie Lamport
Standard for protocol specification
Successfully used for a variety of protocols in sequential, concurrent and distributed setting

TLA+ (2)

- Implementations can still be erroneous (Redis-Raft, development)
  - Jepsen tests found 21 issues
  - Infinite Loops
  - Total Data Loss on Failover
- Solution?

Does not prevent errors, even though it is development versions, assumption is that some tests did exist which failed to catch error
How to solve this now?

Light Client - intro

- Simple Payment Verification, known since the Bitcoin whitepaper
- Reduces the bandwidth and computation needed to monitor the blockchain
- Can be used for:
  - Wallets
  - Interblockchain Communication Protocol

https://bitcoin.org/bitcoin.pdf

Well known in the world of blockchain, uses blockchain specific properties to allow for easier data synchronization.
Usually used for mobile devices but can also be used for any kind of communication which can benefit from reduces overhead.
Any time it is relevant to have cheaper synchronization.

Tendermint Light Client

- Decomposed into Core Verification, Fork Detection and Evidence Submission
- Roles:
  - Core Verification – validity of data with respect to Tendermint failure model
  - Fork Detection – detection of forks and malicious full nodes
  - Evidence Submission – informs nodes of misbehavior
- Canonical implementation in Go
- Core Verification is specified in TLA+
- Fork Detection and Evidence Submission are parts of an ongoing effort

Recap what Zarko and others say

Formal Verification

- Increases development time
- Oriented towards specifications
- Rarely results in software which is also verified and executable
- Tools for software verification are usually for exotic languages (Coq, Haskell, OCaml, Lisp, etc.)
- Do programmers want to use any of those?
  - StackOverflow 2020 Developer survey, only Haskell is present

It is still oriented towards specification in most cases
Forkflow is slower, requires more rigorous approach to writing code
Does not always result in usable software or is used for toy examples
Notable exceptions exist like CompCert
Not really, Haskell is the only one registered on StackOverflow developer survey https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages

Cosmos

- Network of Blockchains
- Has its own currency used for stakeing
  - Cosmos Hub, first of many
  - 1.7B market capitalization
- Implemented in Go
- Provides Cosmos SDK for easier building of blockchains

https://cosmos.network/

Currently the main application of Tendermint and probably the most evolved ecosystem

Tendermint Consensus

- Byzantine Fault Tolerant State Machine Replication
  - byzantine – authenticated processes can exhibit arbitrary faults
  - fault tolerant – assuming partial synchrony of the network, > 2/3 correct
  - replicates arbitrary deterministic state machines
- Blockchain based
  - Blocks – ordering of transactions to be executed
  - Chain - total order on execution of transactions

Designed for arbitrary application replication.
We assume that processes can be hacked, malicious, etc. but they can not break cryptography
Keeps the history of the app in the blockchain, order on the transactions enforced.
Contains cryptographic and time information
Allows for validation of the blockchain, prevents retroactive changes to the blockchain.

Tendermint - accountability

- Proof of Stake system
  - Monetary incentivization
  - Unbonding period, during which we believe a malicious node can be detected
- Introduces stronger assumptions about timing in the network
- Trusting period, during which we assume a node is correct

Unlike Proof-of-Work which requires non-significant but tractable computations
Proof of Stake
We believe a certain set of processes can be punished during a certain period.

LightBlock

- Main data structure LightBlock, collection of data needed for Core Verification
- LightBlock is a data structure local to the Light Client
- Verifies that a LightBlock at a target height originated from the same chain as a given predecessor LightBlock
- Does not need to retrieve all intermediate LightBlocks
  - Called skipping verification, usually midpoint between two heights