# Building Trustworthy Smart Contracts using Interactive Theorem Proving

A thesis

submitted in fulfilment

of the requirements for the degree

of

**Doctor of Philosophy in Computer Science**

at

**The University of Waikato**

by

**Daniel Britten**

**2024**

THE UNIVERSITY OF
WAIKATO
*Te Whare Wānanga o Waikato*

# Abstract

There are varying approaches to the verification of smart contracts using formal methods. This thesis advocates for the use of high-level specifications coupled with a verified compiler to low-level bytecode, such as for the Ethereum Virtual Machine. Taking this approach allows for specifications to more closely match natural language, while ensuring that the specifications apply to the real bytecode executed on-chain.

Interactive theorem proving can provide the foundation for developing provably correct smart contracts. Due to the immutable nature of smart contracts and their potential to manage highly valuable assets and tokens representing power, techniques to ensure their correctness are of paramount importance.

This thesis extends the DeepSEA (Deep Simulation of Executable Abstractions) smart contract language targeting the Ethereum Virtual Machine by mitigating the issues associated with reentrancy and introducing a model of relevant aspects of a blockchain. This enables the specification and verification of two case studies which exemplify the approach of developing provably correct smart contracts.

The specifications for the case studies are written in the language of the Coq Proof Assistant, making arbitrary mathematical statements expressible. The blockchain model enables stating and proving temporal properties relating to the execution of smart contract over time.

While smart contracts are an ideal application area for formal methods in general and interactive theorem proving in particular, the techniques exemplified in this thesis could be applied throughout software engineering. The relatively young age of smart contract languages and typically small size of smart contract programs makes the application of interactive theorem proving more tractable. Future work could involve demonstrating the applicability to many interrelated smart contracts and to larger software projects in different domains.

The first three chapters of this thesis cover introductory and background material. This is followed by the contributions to the DeepSEA system. The two case studies and the proof themes arising from them are then presented. Finally, concluding remarks and future work are discussed.

*"The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness."* - Dijkstra [1]

# Acknowledgements

I am incredibly grateful for the advice, support, encouragement and expertise of my chief supervisor, Professor Steve Reeves. Steve, thank you for sharing so much of your time and energy with me, whether it was during your sabbatical or re-reading drafts. I really do feel blessed to have been your student and to have grasped much of what it means to be a researcher from your example and advice. Thank you, Steve, it has been an incredible journey.

Dr. Vilhelm Sjöberg at CertiK has kindly offered many hours of his time and helped me plumb the depths of the DeepSEA codebase. I feel honoured to have been given your time and attention which has helped me progress with the technical aspects of this thesis at so many points. Thank you, Vilhelm, I very much appreciate your involvement.

I would also like to thank my supervisor Dr. Vimal Kumar, who has also been a great support over the years with valuable insights along the way. Thank you, Vimal.

The University of Auckland and Associate Professor Jing Sun have kindly hosted me during the majority of my PhD research. I am so grateful for their support which has enabled me to be connected with the vibrant Computer Science PhD community there who have been a great support also.

Thank you to the friends I have made as part of the Computer Science community at the University of Waikato. Catching up at lunch during my visits has been a highlight for me, and a great source of laughter too.

I am very grateful for the financial support provided by the University of Waikato Doctoral Scholarship and the University of Waikato Research and Enterprise Study Award which have enabled me to carry out this research.

Finally, I am immensely grateful to my family, wider family, and friends who have been a constant source of support and encouragement at all points in this journey. In particular, I would like to thank my sisters, Joy and Bethany, for being there for me in so many ways and for all the board games, gigs, and time together that keep my spirits high. Lastly, thank you to my parents, Randall and Yvette, for everything from teaching me to program as a child to supporting me through the tough times during this PhD and in all the years beforehand.

I am extremely grateful to have had such great support, thank you all. You are much appreciated.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Software engineers are focused on producing reliable and correct programs. From the early days of computing to now we have seen the adoption of safer languages [2], stricter type systems [3], and ever-improving testing practices [4]. A natural next step towards even stronger guarantees of correctness is the adoption of the formal method of interactive theorem proving. Smart contracts, in particular, necessitate stronger guarantees of correctness due to their existence on a blockchain and therefore immutable nature, as well as their potential to handle digital assets of great value. This thesis explores the use of interactive theorem proving to provide robust guarantees of smart contract correctness.

## 1.1 Interactive Theorem Proving

The key idea behind interactive theorem proving for programs is for the engineer to capture their own intent and the reasoning for why their implementation matches that intent in a form that a proof-checking program can validate. The engineer's intent and reasoning is written out by the engineer into a formal specification and formal proof, respectively. Separately, the meaning of the constructs of the programming language used must be captured in a formal semantics of the language. This is ideally done by the developers of the programming language and incorporated into the theorem prover.

## 1.2 DeepSEA

This thesis demonstrates the above ideas in action, with the concrete example of DeepSEA[1] (Deep Simulation of Executable Abstractions) smart contracts. The DeepSEA system [7] integrates with the Coq Proof Assistant [8] and has been primarily developed by CertiK [9]. The DeepSEA compiler is partly based upon the CompCert verified compiler [10] developed as a part of the INRIA CompCert research project [11]. DeepSEA smart contracts can be compiled to Ethereum Virtual Machine (EVM) bytecode via a verified compiler, enabling DeepSEA smart contracts to be deployed on the Ethereum blockchain. This demonstrates that the

[1]In this thesis, DeepSEA refers to the version targeting the Ethereum Virtual Machine, as opposed to the version for the C language [5], [6].

ideas explored in this thesis could be incorporated into real-world smart contract development today.

DeepSEA serves well as a concrete example of enabling an engineer's intent and reasoning, as well as the meaning of a programming language, to all be captured in a form that enables the validation of a formal proof that the smart contract's specifications hold. The specifications are validated with respect to the formal semantics of the DeepSEA language. DeepSEA is also rare in that the smart contracts can be written in a relatively high-level language and the formal specification of a smart contract can similarly be expressed at a high level. Importantly, the proofs of the formal specifications carry their validity to real-world deployed bytecode of smart contracts on the blockchain as a result of a refinement proof. This thesis defers an explanation of DeepSEA's verified compilation to the paper "*Foundational Verification of Smart Contracts through Verified Compilation*" [7].

The work of this thesis builds upon the existing DeepSEA system and extends it, bringing it closer to being a realistic system for developing provably correct smart contracts and demonstrating this via two case studies. The DeepSEA system is improved in two key aspects. Firstly, it is improved by handling reentrancy appropriately via enforcing the Checks-Effects-Interactions pattern [12]. This guarantees that proofs about smart contracts which include cryptocurrency transfers and calls to other contracts are not at risk of being invalid due to the possibility of reentrancy, but rather any reentrancy is benign. Secondly, a model of a blockchain [13] is produced as a wrapper around the individual smart contract functions generated by DeepSEA. This increases the expressivity of DeepSEA specifications, allowing them to be phrased in terms of concepts such as which states of a blockchain are reachable through a sequence of steps. The two case studies, a crowdfunding smart contract and an ERC-20 token [14] smart contract, use both these extensions and demonstrate the application of theorem proving to smart contract development.

## 1.3 Case Studies

The case studies exemplify the need for theorem proving for smart contracts. Both have the potential to handle large amounts of cryptocurrency (called "*ether*" in the Ethereum platform) as well as being immutable, thus the protocol disallows any flaws being fixed after deployment. The crowdfunding example and its specification are based upon the crowdfunding smart contract from the Scilla project [15]–[18]. The specification for the token smart contract is extended from the specification for fungible tokens outlined in Ethereum Request for Comment 20 (ERC-20) [14]. Fungible tokens represent a value of varying quantity as opposed to a non-fungible token (NFT) which represents a unique, non-countable item. The introductory example in Section 3.2.9 relating to a smart contract for managing access to a rental property gives a hint of how smart contracts can also manage tokens that represent power. In all these cases, it will be clear that the correctness of each smart contract is of great importance.

## 1.4 Future Potential

Looking forward to the future, towards when it may become commonplace for software to be verified using a proof assistant, the immutability of a blockchain provides a unique advantage of smart contracts over software in general. On a blockchain, due to immutability, every end-user of the same 'on-chain' smart contract can locally verify its correctness and still have confidence that the on-chain version of the smart contract is identical to the locally verified version and that properties proved about it locally hold on-chain. In addition, immutability guarantees that the smart contract's properties will continue to hold as there can be no updates which break correctness properties, introduce bugs or even add intentional exploits. This allows end-users to collectively interact with online software, whose correctness they are able to verify. This is an exciting opportunity for formally verified software to be accessible to a wider range of end-users with minimal reliance on centralised authority and for users to have the ability to collectively interact with the same piece of online software.

It would be naive to look to the future without being aware of the past and the legacy of skilled computer scientists who also hoped for formal verification to become more commonplace. For example, Dijkstra made the following comment around 1964, suggesting his hope for more principled methods to determine that a machine acts according to its specifications:

> *"But what is the value of an acceptance test? It is certainly no guarantee that the machine is correct, that the machine acts according to its specifications. It only says that in these specific test programs the machine worked correctly."* - Dijkstra [19]

More recently in 1996, Barendregt described the potential of what he calls "*computer mathematics*" as a new technology that may help with the design of reliable software.

> *"Presently a new technology is emerging:* computer mathematics. *It consists of the interactive building of definitions, statements and proofs, such that it can be checked automatically whether the definitions are well-formed and the proofs are correct. ... After the technology has matured, it may become a tool for the development of mathematics comparable to systems of computer algebra, but with a scope and strength that is essentially beyond. Moreover, it will probably also be useful for the design of reliable software."* - Barendregt [20, Chapter: The Quest for Correctness]

We can hope that as we as a society come to increasingly rely upon software then we will eventually see the value of practices which can ensure the development of correct software. Due to this, it still seems likely that techniques such as interactive theorem proving will eventually become commonplace, at least in the most critical software.

## 1.5 Thesis Structure

The next two chapters cover the background and describe the work which this thesis builds on. Background literature and an introduction to the DeepSEA language (Chapter 2) are followed by Chapter 3, a technical introduction to the Coq Proof Assistant [8] with a focus on concepts important in DeepSEA. In Chapter 4, the main contributions to the DeepSEA language of handling reentrancy and modelling a blockchain are discussed. Making use of these contributions, the two case studies (Chapter 5 and Chapter 6) are then introduced. Finally, combining lessons from the case studies, proof themes common to the case studies are analysed (Chapter 7).

## 1.6 Summary of Contributions

The main contributions of this thesis are:

1. Handling reentrancy in DeepSEA by enforcing the *Checks-Effects-Interactions Pattern*.
2. Enabling richer high-level specifications in DeepSEA, involving:
   - Modelling smart contracts as a whole.
   - Modelling relevant aspects of a blockchain.
   - Enabling specifications about trace correctness.
3. Significantly progressing towards fulfilling the vision of a system for smart contract development which:
   - Supports arbitrary high-level properties (e.g. including trace correctness).
   - Has strong guarantees that the high-level properties actually apply to the deployed bytecode.

## 1.7 Remarks

The concrete example of applying these techniques to smart contract development gives insight into how interactive theorem proving could be incorporated into next-generation software development practices, leading to increasingly reliable and secure provably correct programs.

# Chapter 2

# Background

## 2.1 Formal Methods

Formal methods, as described by the NASA Langley Formal Methods Research Program, refer to:

> *"mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems."* [21]

For example, theorem proving (in the context of formal methods) involves the use of mathematical proofs about programs, where the proofs are machine-checked in order to provide strong guarantees about the correctness of the programs. Other formal methods techniques include model checking [22], model-based testing [23], Petri nets [24], and the use of formal specification languages such as Z [25].

Formal verification is the use of formal methods techniques to prove that a system adheres to a formal specification. Using theorem proving for formal verification is possible for mainstream languages such as C# [26] and Java [27] as well using proof assistants such as Coq [8] and Isabelle/HOL [28]. Formal verification projects include the seL4 microkernel verification project [29] and a range of projects connected to the DeepSpec expedition in computing [30].

For imperative languages such as C# and Java, verification often involves annotations of pre-conditions and post-conditions surrounding function calls. These can then be composed and made use of in new functions and the accompanying proofs. Loop invariants are also commonly used to assist the proof checker's analysis by providing the relevant statement that is true before and after every iteration of the loop.

Alternatively, in Coq theorems are written about functions defined in its functional language, Gallina. These theorems are then available when writing theorems about related functions. For example, one could prove that a sorting function's output is a sorted permutation of its input with respect to a given comparison operator. Making use of this proof, one could write a function to return the minimum element of a list which returns the first element of the output of the sorting function. The proof of correctness of the function returning the minimum can make use of the proof of the sorting function's correctness.

Formal methods provide the groundwork for proving the correctness of smart contracts. Existing tools and techniques can be applied to the verification of smart contracts. Still, there are new exciting challenges to consider relevant to the blockchain domain.

## 2.2 Blockchain

A blockchain, as defined by the International Organisation for Standardization (ISO), is a:

> *"distributed ledger with confirmed blocks organized in an append-only, sequential chain using cryptographic links; designed to be tamper resistant and to create final, definitive and immutable ledger records."* [31]

For the purposes of this thesis, we are mainly concerned with smart contracts on a blockchain and aspects of normal operation of a blockchain that could affect the validity of correctness proofs about smart contracts. Overall, we assume the correct functioning of the blockchain and that the ledger records are indeed final, definitive and immutable. We recognise that this assumption is not always appropriate, but the efforts to ensure that these assumptions hold are beyond the scope of this thesis.

## 2.3 Smart Contracts

A smart contract, as defined by the ISO standard, is a:

> *"computer program stored in a distributed ledger system wherein the outcome of any execution of the program is recorded on the distributed ledger."* [31]

They have also been described as "*self-executing digital contracts*" [32], [33]. The term was first coined by Nick Szabo in his 1997 paper, *Formalizing and securing relationships on public networks* [34], where he described smart contracts in this way: "*Smart contracts combine protocols with user interfaces to formalize and secure relationships over computer networks*".

The key difference from traditional software is that they operate in a so-called "*trustless environment*". Klems et al. [35] define trustless in the following way:

> *"We define trustlessness as a system property which guarantees rules of interaction that are known to and agreed upon by all participants of the system, and which cannot be unilaterally changed. These guarantees are enforced through, what we call trustless intermediation, a set of mechanisms for decentralizing the enforcement of rules in a system, thereby removing the need for and existence of trusted intermediaries."* [35]

There is no reliance on trust that a central authority will run the contract as it has been programmed. Instead, every computer that validates a transaction involving a smart contract will execute the smart contract code [36], [37]. This way the execution of every smart contract is checked by many parties which ensures it has been executed as it has been programmed. There are various kinds of consensus protocols which ensure the correct execution of smart contracts, often based on the assumption of a majority of honest participant nodes in the blockchain network.

In this thesis, the assumption is made that the consensus protocol runs as intended and that smart contracts can be considered to always be executed according to their semantics. In practice, it is possible for 51% attacks, or other exploits to break this assumption, however this is beyond the scope of this thesis.

### 2.3.1 The DAO and the Parity Wallet

Historically, the techniques used for producing trustworthy smart contracts have failed on multiple occasions, sometimes spectacularly. Two classic examples of this are in relation to The DAO [38] and the Parity wallet [39]. The DAO was a popular fund where people could send ether (the cryptocurrency of the Ethereum [40] blockchain) that would then be invested in different areas. Due to a subtle bug relating to recursively calling a function to drain funds before The DAO contract updated the caller's balance and checked that the caller had enough funds, it was possible for an attacker to steal over 50 million USD worth of ether. This demonstrates the real-world risk of not using formal methods, for example the approach outlined in Section 4.1 would protect DeepSEA smart contracts from the issue that led to the attack on The DAO. Interestingly, the Ethereum community decided to carry out a so-called hard fork to essentially undo the attack, rewriting some data stored on the blockchain that would be considered immutable under normal circumstances. Some disagreed with this approach, leading to the existence of the separate Ethereum Classic chain (ETC), as discussed by Woitschig et al. [41], for example. This all goes to show that prior discussions and commitments amongst a blockchain community about governance is valuable.

A second example is the Parity wallet which relied on a particular library contract that turned out to be vulnerable to being able to be 'killed' by anyone (anyone could become the owner of the contract by calling its initialise function and then, as an owner, could kill the contract). This reliance on the buggy contract meant that once the contract was killed, funds using the Parity wallet were locked and inaccessible to anyone. This kind of vulnerability would likely have been detected by MAIAN [42] and a careful application of theorem-proving using DeepSEA would also prevent this flaw.

### 2.3.2 Eliciting Specifications for Smart Contracts

Anybody may intend to create a smart contract, and ideally their smart contract would be proved correct. This would necessarily involve the use of a rigorous formal specification that faithfully reflects exactly the intention of the person who wanted to make the smart contract. However, this translation from intention to formal specification may well *not* faithfully reflect the intention. This difference could stem from misunderstandings, lack of expressivity of the formal language, time constraints and errors.

This issue is the motivation for being able to write formal specifications at a high level. It makes the gap between the formal specifications and people's intent much smaller than when writing formal specifications directly at the bytecode level.

## 2.4 Discussion of Background Literature

### 2.4.1 Formal Methods

#### 2.4.1.1 The Coq Proof Assistant

Coq [8] is a widely used proof assistant, a piece of software that facilitates writing theorems in a formal language and mechanically checking the correctness of proofs. Coq uses dependent type theory, enabling theorems from mathematics to be expressed and proved as well as theorems representing properties about programs. Dependent type theory allows you to use values as a part of defining a type. For example, you can have a type that is lists of length 3, or a type for 5 by 6 matrices. It turns out that you can use this to phrase theorems as types, which are proved to be true when it is demonstrated that a value (that is a "*proof object*") inhabits the type. The expressivity of this type system makes it undecidable, so, in particular for theorems, it is necessary to provide hints to Coq to guide the type-checking/proving process. In general, hints can also be useful in making type-checking or proof search more tractable in both decidable and undecidable systems.

#### 2.4.1.2 Isabelle/HOL Proof Assistant

Isabelle/HOL [28] is a proof assistant that is also widely used. Isabelle "*provides a higher-order logic theorem proving environment that is ready to use for big applications*". There is also the Archive of Formal Proofs [43] which is a refereed collection of formal proofs in Isabelle. The formal semantics of the Ethereum virtual machine that is used by DeepSEA (discussed in Section 2.4.2.1) also generates an Isabelle version, however there do not appear to have been any high-level languages developed using the formal semantics of the EVM in Isabelle.

#### 2.4.1.3 SMT Solvers

Satisfiability Modulo Theorems (SMT) solvers, such as Z3 [44], are incorporated into various formal verifcation tools, including Dafny [45], Why3 [46], Boogie [26], [47] and Whiley [48]. Such tools translate their proof goals into a form that can be solved by an SMT solver and then invoke the SMT solver. The SMT solver can, in some cases, provide counterexamples when a goal cannot be discharged which can guide the programmer towards the error in their implementation (or specification). There are some clear benefits to automated theorem proving, such as that it requires less expertise from the programmer. However, sometimes at least as much expertise as for interactive theorem proving is required when the SMT solver needs to be guided by helper lemmas. Other SMT solvers include Alt-Ergo [49] and CVC [50].

#### 2.4.1.4 SPARK Pro Proof System

SPARK Pro [51] is a commercial tool building upon an imperative, object-oriented language. It supports the verification of programs written in its language using annotations such as pre-conditions and post-conditions to specify the behaviour of functions. As a backend, SPARK Pro uses SMT solvers such as Z3 or Alt-Ergo. It can also use Coq as a backend to enable interactive theorem proving.

#### 2.4.1.5 Process Analysis Toolkit (PAT) Model Checking Tool

PAT [52] is a model checking tool, designed to be "*a self-contained framework to support composing, simulating and reasoning of concurrent, real-time systems and other possible domains*" [53]. PAT has been used for the model checking of moderately complex systems such as Android applications [54] where certain security properties were verified. PAT has also reached beyond its initial purpose through the use of model checking techniques in the AI planning domain [55].

#### 2.4.1.6 TLA$^+$ Model Checking Tool

TLA$^+$ [56] is also a model checking tool, described by its creator Lamport as "*a high-level language for modeling programs and systems–especially concurrent and distributed ones*" [56]. Due to its focus on concurrent and distributed programs and systems, it is well-placed to be used for modelling smart contracts.

TLA$^+$ has been used by engineers at Amazon Web Services to "*to help solve difficult design problems in critical systems*" [57] as well as by Intel [58], Microsoft, and others.

Technically, TLA$^+$ is a *specification language*, usually coupled with the TLC model checker. The TLA$^+$ Proof System (TLAPS) is another related component, making it possible to carry out TLA$^+$ proofs in a range of ways such as using the Isabelle proof assistant [28] mentioned above.

#### 2.4.1.7 Quick-Check Property-Based Testing

When model checking is used for finding counter-examples it is somewhat related to property-based testing such as the Quick-check tool [59] originally developed for Haskell, but also available for Coq [8] (as Quick-Chick [60]). Quick-check examines the type signature of a given function and auto-generates a range of random tests for user-specified properties of that function, sometimes requiring the user to define how to carry out the random sampling for more complex data types. Like model checking, if all the tests are passed this does not imply that the code is correct, but it is an indication that there are no problems related to any of the randomly generated test cases. Property-based testing provides actionable information when a test fails, but all tests passing does not demonstrate the absence of bugs. Still, property-based testing could be a useful step as part of a tool chain for developing verified software and requires less effort from programmers when compared to the other techniques discussed above.

#### 2.4.1.8 Remarks

Model checking involves representing the system that needs to be formally verified as one where states can transition to other states and then testing the correctness of the system with respect to certain specifications by exploring a wide range of possible states. In a system as complex as a typical smart contract, model checking does *not* prove that the smart contract adheres to a particular specification, rather it is capable of finding counter-examples. The absence of counter-examples does not prove correctness because model checking can't cover all possible cases of the smart contract's execution - there are limits to what parts of the state space is explored and as such there could be hidden counter-examples in the areas of the state space that were not explored. For simpler systems however, a model checker can explore the entire state space and thus prove the correctness of those systems - even if the systems are reasonably complex.

This means that model checking is not suitable as a standalone tool for proving the correctness of smart contracts. It would leave open the possibility that a malicious user could carefully analyse the smart contract (and the model checking system) and find exploits that lay beyond the state space that the model checker would have explored when verifying the contract.

Nevertheless, it would be useful to have model checking form part of a tool chain for verifying the correctness of smart contracts. One limitation of some theorem proving software is that if you are trying to prove something that is indeed false, it can at times be difficult to realise that you are attempting an impossible proof (in some proof systems this is less of an issue). Because of this, tools like model checking could prove useful in order for a developer to find counter-examples that help them check whether their specification is *likely* to be correct. Then once they have iteratively improved their specification such that the model checker finds no counter-examples with it, then they could move on to proving the correctness of their program.

Pneuli [61] makes the important point that deductive technologies (theorem proving) have "*significantly better scalability*" as exploring all possible states can be computationally expensive, sometimes prohibitively so. We can see this in a simple form by comparing the process of proving the correctness of a smart contract that involves some property of an integer value that can be reasoned about using induction, versus the necessity to explore the entire state space of the values the integer may hold in order to achieve the same level of certainty via model checking.

Interactive theorem proving provides the strongest guarantees of correctness and is theoretically applicable to all problems that can be formulated mathematically. This does come at the cost of requiring human expertise, however works such as this thesis aim to take us closer to proof systems which make it possible for proof engineers to focus their energy exactly where needed (for one example, see Section 4.2.2). There is also the benefit that proofs in a proof assistant can generally be audited by other humans to ensure that they do indeed prove the proof goal. This has great value because, undoubtedly, there will very occasionally be bugs in the proof assistants themselves and so having a human be able to check whether the proof script is 'reasonable' goes a long way to ensuring that the proof does not explicitly or inadvertently exploit such a bug.

### 2.4.2 Semantics of Low-level Languages for Smart Contracts

#### 2.4.2.1 EVM Semantics in Lem

Hirai [62], [63] wrote a semantics of the EVM in the Lem language. Lem can be translated into proof assistants such as Coq [8] and Isabelle [28]. Lem is also executable, which has the benefit of allowing the EVM test suite to be run against the semantics to increase confidence that the written semantics matches how Ethereum clients actually behave. The translation into other proof assistants such as Coq is not verified and so although the test suite has been checked against the Lem model, we are required to trust the translation if we are to hold that the semantics in Coq matches how Ethereum clients behave. Hirai's Lem model was tested on the `VMTests` portion of the EVM test suite[1]. At the time, Hirai's model passed all 40,619 tests,

[1]`https://github.com/ethereum/tests/tree/79b14fbd74c0148f1b8cca442252e9299464fe8c/VMTests`.

excluding 24 which were out of scope as they involved interactions between multiple contracts [62]. The DeepSEA system makes use of the Lem semantics in Coq, which form part of its trusted code base.

#### 2.4.2.2 EVM Semantics in K

Hildenbrandt et al. [64] formalised the semantics of the EVM in the K framework [65]. Their specification is also executable and a reference interpreter and debugger for EVM bytecode is generated from the formal specification. This reference interpreter is then used to successfully run the official test suites for EVM implementations, increasing confidence that the formalised semantics matches the real-world implementation of the EVM. However, to be precise, there is still the question of whether the generated reference interpreter faithfully represents the formal semantics, but this is fairly reasonable to assume. This work is built upon by others, such as discussed in Sections 2.4.3.13 and 2.4.3.14. This work is crucial for any approach to verification of Ethereum smart contracts at the bytecode level that wishes to utilise the K framework.

#### 2.4.2.3 EVM Semantics in Dafny

At Consensys, Cassez et al. [66] have written a complete formal semantics of the EVM in the verification-friendly Dafny language. This enables the verification of Ethereum smart contracts at the bytecode level and opens the door to developing a verified compiler written in Dafny from a higher-level language. Dafny leverages the SMT solver Z3 which has the benefit of automating much of the verification effort when proving the correctness of a smart contract. To reason about smart contracts low-level formal semantics will always be necessary and have to be part of the trusted computing base, so this is a valuable contribution that is key to enabling the verification of Ethereum smart contracts using Dafny. Another crucial benefit of their approach is that the semantics is executable which helps increase trust in the semantics because the standard EVM tests can be run on an extracted version of the semantics. This does not guarantee that the non-extracted Dafny code would pass these tests but goes a long way towards establishing that they would.

#### 2.4.2.4 Subset of EVM Semantics in Vale

Cumming et al. [67] have formalised a subset of the EVM semantics in Microsoft's Vale framework. This, again, enables the verification of smart contracts written at the bytecode level. They also support an intermediate language with conditional logic which enables writing contracts in a more readable intermediate language while still facilitating verification. Similarly to the Dafny EVM formalisation, theorem proving in Vale is powered by the Z3 SMT solver. The specifications are written as pre- and post-conditions, as with the EVM semantics in Dafny. A benefit of their framework is that it is like a verified compiler in that the properties proved at the intermediate language level can be known to hold at the bytecode level, when trusting the key components of the Vale framework. Expanding the supported bytecode and increasing the expressiveness of the intermediate language could further demonstrate the utility of their approach, and the existing work demonstrates that verifying smart contracts using the Vale framework is feasible.

#### 2.4.2.5 Michelson Semantics in Coq

Michelson [68] is the smart contract programming language for the Tezos [69] blockchain. Michelson serves the same role for Tezos as EVM bytecode does for Ethereum. The Michelson semantics is developed by

Nomadic Labs [70]. The Michelson language has features which make it particularly amenable to formal verification, for example, integers are of unbounded size (constrained only by gas). If this was the case with Ethereum, this feature alone would greatly simplify aspects of proofs in the case studies (discussed in Sections 5.9 and 6.9). This highlights that designing a low-level smart contract language with formal verification in mind from the outset can be greatly beneficial.

**Philip Wadler's view of Michelson** Philip Wadler (IOHK) has discussed Michelson [71] asking the question "*Why?*". One of his concerns is that it is an "*inexplicably low-level language*". Wadler describes it a "*mix of some of the best and worst of computing*". In fairness to Michelson, Wadler admits that he only later found out that Liquidity (a high-level language) exists and compiles down to Michelson and also Wadler does not touch on the robust work done to enable proofs of the correctness of Michelson smart contracts in Coq. It seems that Wadler may have missed the potential for Michelson to serve as a target language for verified compilers and some of the design decisions which make it an appealing target language of higher-level languages supporting formal verification.

### 2.4.3 Existing Approaches Related to Smart Contract Verification

There are a wide range of approaches to smart contract verification with some techniques focussing on the verification of a fixed set of specific properties. The techniques and approaches in this section are not intended to be an exhaustive list, rather they give insight into the range of approaches available.

#### 2.4.3.1 Position & Survey Papers

A recent survey of pre-deployment analysis of smart contracts by Munir and Taha (2023) [72] describes the range of approaches used to reduce the risks inherent in smart contract development. They describe a range of liveness and safety properties they identified, including user-specified properties and safety from reentrancy. Their survey gives an overview of four main approaches for pre-deployment analysis: static analysis, formal verification (including model checking and theorem proving), pre-deployment dynamic analysis, and machine learning methods. Compared to earlier surveys, the prevalence of machine learning methods used to help demonstrate smart contracts for vulnerabilities appears to have increased. Their survey highlights that there are a wide range of possible exploits to smart contracts and that there does not appear to be a 'silver bullet' method that handles the analysis of all possible vulnerabilities.

Magazzeni et al. (2017) [73] put forward a research agenda for the validation and verification of smart contracts. They suggest that we need "*a formal (machine-readable) representation of a smart contract program and its effects on the world in which it exists*". DeepSEA coupled with the blockchain model developed in Section 4.2 is a potential fulfilment of this goal. Magezzeni et al. also highlight that the behaviour of multiple smart contracts interacting together is important to consider.

Miller et al. (2018) [74] discuss a range of approaches to verifying the correctness of smart contracts. The paper covers efforts to develop languages which reduce the risk of common errors such as the languages FSolidM [75], Rholang [76], Vyper [77], Type-coin [78], Simplicity [79], and Plutus [80] as well as languages with the intention of being able to support formal verification such as Michelson [68], its counterpart Liquidity [81], and Scilla [15]. Generally, the earlier languages are able to restrict the smart contract writer from creating a contract that is vulnerable in certain specific ways. In contrast, the latter languages aim to provide support for verifying the full functional correctness of specific properties relevant to the smart contract, while also helping prevent the more generic possible vulnerabilities. This is enabled by the ability to formally prove specific properties of smart contracts which provide solid guarantees of correctness with respect to a formal specification.

Singh et al. (2020) [82] conducted a systematic literature review focusing on "*the state-of-the-art research and achievements concerning the formalization approaches in smart contracts.*". The work they cover naturally overlaps with the languages looked at by Miller et al. but with their broader focus they also cover some research on model checking for smart contracts as well as other approaches. A sample of work discussed is: FSolidM [75], Bai et al. [83] (using the SPIN model checker), Abdellatif and Broushmiche [84] (using Statistical Model Checking), Bhargavan et al. [85] (using F*), KEVM [64] (executable formal specification of the Ethereum virtual machine), Oyente [86], Raziel [87], ContractLarva [88], Securify [89], Simplicity [79], Amani et al. [90] (extending an Ethereum virtual machine formalisation in Isabelle/HOL), ZEUS model checking [91], Hirai's work on an Ethereum virtual machine model in Lem [62], [63], MAIAN [42], and Scilla [16]. These works involve techniques including theorem proving, model checking, symbolic execution, and formal modelling. The most common technique identified was theorem proving, which is indicative of the key role it plays in this field. They identify three main open issues for smart contract research: formal testing, automated verification, and domain specific languages for Ethereum.

Atzei et al. (2016) [92] collated a taxonomy of vulnerabilities in Ethereum smart contracts covering potential issues at the Solidity, EVM and blockchain level. One of these causes of vulnerabilities, reentrancy, is tackled by the improvements to the DeepSEA language discussed in Section 4.1. Some of the other vulnerabilities discussed in their taxonomy are important, but outside the scope of what this thesis aims to address. These include keeping certain variables secret, preventing ether being lost in transfer (to recipients that do not exist), and considerations around generating randomness for use by smart contracts. The primary item in their taxonomy that this thesis aims to address is "immutable bugs" with the aim of eliminating bugs prior to deployment through the application of interactive theorem proving.

#### 2.4.3.2 Towards Verifying EVM Bytecode in Isabelle/HOL

Amani et al. [90] describe work on partially decompiling EVM bytecode as well as creating a program logic to reason about the partially decompiled code. Their work builds upon the Lem formalisation of the EVM [62], [63] discussed in Section 2.4.2.1. Their program logic makes it possible to specify pre- and post-conditions for segments of code and prove that these hold using Isabelle/HOL [28]. The use of decompilation makes their approach applicable to all smart contracts targeting the EVM. However, it is challenging to recover high-level concepts, such as loops, through decompilation. Perhaps because of these limitations, they conclude that "another promising avenue for research are verified compilers" [90]. DeepSEA, as a verified compiler, is an example of such research.

#### 2.4.3.3 Modelling Pact Smart Contracts with Z3

Pact is a lisp-like language designed to be a safe and user-friendly smart contract language. Some key aspects of its design that help this goal are that it is "*Turing incomplete, human readable, supports upgradable* *contracts, and formal verification*" [93], [94]. In Pact, Turing-incompleteness means that there are no looping constructs except the ability to use `map`, `fold` and `filter` on finite lists. Formal verification in Pact involves the Z3 theorem prover, however there are limitations on what can be expressed as specifications in Pact (at least at this stage).

#### 2.4.3.4 Modelling Scilla Smart Contracts in Coq

Scilla [16] is an intermediate level language designed to facilitate formal verification. According to the original paper about Scilla [15], "*Scilla provides a clean separation between the communication aspect of smart contracts on a blockchain, allowing for the rich interaction patterns, and a programming component, which enjoys principled semantics and is amenable to formal verification*". Scilla takes an automata-based approach, where calls to other contracts must occur only at the end of a Scilla function (called a transition). This approach makes the formal verification of transitions which involve such calls easier. This is exemplified by the types of Scilla contracts which have been verified which include a crowdfunding contract and a puzzle solving game contract. Research on Scilla integrated with theorem proving appears to have halted [18], however the language itself continues to be used for the Zilliqa blockchain [95]. The crowdfunding case study explored in Chapter 5 is heavily inspired by the same example used originally in Scilla and also used by ConCert [96] and serves as a benchmark of sorts.

#### 2.4.3.5 Modelling Solidity-like Smart Contracts in Dafny

Cassez et al. [97], [98] modelled Solidity-like smart contracts in the Dafny language. A benefit of their formalisation was their modelling of reentrancy via the smart contract functions being co-recursive if they included a call to `transfer` at the point that call was made. This enabled handling of the problem of reentrancy because any properties proved about the smart contract would need to still be shown to hold in the context of their co-recursive definitions which modelled reentrancy. They showed that their modelling correctly identified some examples that did and did not follow the *Checks-Effects-Interactions Pattern* discussed in Section 4.1. Dafny uses the SMT solver Z3 [44] to discharge proof goals.

#### 2.4.3.6 Modelling DeepSEA Smart Contract Functions using Lem EVM Semantics

Prior to the start of this research, an earlier version of the DeepSEA system [99] existed. This enabled the modelling of DeepSEA smart contract functions in Coq (individually) with a refinement proof linking that representation to bytecode at the level of the Lem EVM Semantics [62], [63]. There were certain limitations to this model, including a lack of the ability to reason about a smart contract as a whole and over its lifetime, as well as not handling the risk of reentrancy. Both of these limitations have now been resolved via work that is discussed in Section 4.1 and Section 4.2 which have enabled the verification of properties of the crowdfunding smart contract (Chapter 5). For an overview of DeepSEA, please see the blog posts "*An Introduction to DeepSEA*" [100] and "*How DeepSEA Works*" [101], [102], as well as the DeepSEA Language Reference [103]. The recent paper "*Foundational Verification of Smart Contracts through Verified Compilation*" [7] also gives insight into the DeepSEA system as much of it remains as it was before this research started.

#### 2.4.3.7 Model Checking using SPIN

Bai et al. [83] use the SPIN model checker to model a very simplistic smart contract in order to verify properties such as deadlock-freeness and fairness. The techniques demonstrated are limited in the sense that they do not model anything close to what is expressible by a real-world smart contract language. Nevertheless, they do identify and begin to demonstrate that model checking could be a useful tool when applied to smart contracts - especially when the smart contract language is amenable to being represented as a state-machine.

#### 2.4.3.8 Model Checking with ZEUS

ZEUS [91] is the result of more comprehensive work on applying model checking to smart contracts at scale. Kalra et al. [91] consider a range of vulnerabilities specific to Ethereum smart contracts written in Solidity, such as reentrancy, integer overflow/underflow, and transaction order dependence (since a malicious miner has the potential to re-order transactions). ZEUS supports the verification of "*state reachability expressible via quantifier-free logic with integer linear arithmetic*" which is more limited than the expressivity available in interactive proof assistants. Nevertheless, the limitations of ZEUS help make it possible for it to be automatically applied to a vast range of smart contracts (1524 unique smart contracts scraped directly from Ethereum block explorers).

#### 2.4.3.9 VerX Smart Contract Verification

VerX [104] is an automated tool designed for verifying Ethereum smart contracts with respect to custom properties that represent the specification of the contract. It is described as "*The first automated verifier able to prove functional properties of Ethereum smart contracts*". VerX has been applied to real-world smart contracts and has been found to be capable of representing the specifications of them and verifying their correctness. The verification occurs by analysing EVM bytecode, which loses some information compared to the higher-level language it has been compiled from (in this case Solidity). This makes verification more challenging, an issue which is solved by using a verified compiler such as DeepSEA.

#### 2.4.3.10 VERISOL Smart Contract Verification

Microsoft research has developed a tool, VERISOL [105], which is a highly-automated formal verifier for Solidity that makes use of the Boogie [26], [47] intermediate verification language. They describe verifying the correctness of all the smart contracts included with the Azure Blockchain Workbench.

Their approach uses both automated theorem proving and model checking. First, the Solidity smart contract with correctness assertions is translated into Boogie, where it is automatically checked using an SMT solver. If it cannot be automatically proved to be true, then a bounded model checker is applied, either generating a counter-example or the result that the smart contract is verified for up to a certain number of transactions. They also discuss using semi-automated verification for some of the more complex properties of their governance contract for Proof of Authority, indicating that their framework can support human proof effort to a certain extent, along with automated verification.

#### 2.4.3.11 Solidity's SMTChecker

The official Solidity compiler incorporates the tool SMTChecker, described by Marescotti et al. [106]. This tool, makes use of an SMT solver to check that the conditions of `assert` statements hold, as well as other checks such as the absence of division by zero. SMTChecker can generate counterexamples and is well-integrated into the Solidity compiler, making it a natural choice for easily incorporating formal verification into a Solidity project. In situations where the SMT solver fails to prove a goal, SMTChecker gives a warning which *might* be resolved by increasing the time limit for the SMT solver. In contrast, interactive theorem proving makes it possible for the proof engineer to more easily provide insight to the proof system for complex proof goals that an SMT solver fails to discharge. While there are techniques for providing hints to SMT solvers, these are generally less direct than is possible with interactive theorem proving. In addition, while SMTChecker works well at the source-level, there are no formal guarantees that the properties proven actually hold of the compiled EVM bytecode, and so it includes the Solidity compiler in its trusted code base. Despite these drawbacks, SMTChecker is an impressive tool with great practical benefits and highlights how formal verification can be usefully incorporated into smart contract development today.

#### 2.4.3.12 Eth2Vec Machine Learning Analysis of Smart Contracts

Machine learning has also been applied to the challenge of identifying vulnerabilities is smart contracts. Ashizawa et al. [107] make use of machine learning to do this and report an accuracy of 86.6% in detecting reentrancy-related vulnerabilities (relying on the pre-existing tools Oyente [86] and SmartCheck [108] to determine the ground truth in their experiment involving 5,000 smart contracts obtained from a block explorer). Their tool analyses EVM bytecode. Machine learning approaches such as this can be greatly useful for finding some vulnerabilities but do not appear to have the potential to prove the correctness of smart contracts.

#### 2.4.3.13 Runtime Verification's K Framework Approach to a Smart Contract Audit

Park et al. at Runtime Verification used the K framework [65] to carry out end-to-end formal verification of the so-called Ethereum 2.0 deposit smart contract [109]. The deposit smart contract was a critical component of Ethereum's switch to a less energy intensive proof-of-stake consensus protocol. In order to be save gas costs, the smart contract implemented a complex data structure to record deposits. The approach involved a two-part verification effort involving first verifying that the data structure was implemented correctly at a higher level, then verifying that the low-level bytecode was correctly generated from the higher level representation. This approach, like DeepSEA, removes the need to trust the compiler. However, unlike DeepSEA, the proof that the bytecode correctly implemented the higher level representation had to be manually constructed, which took about 70% of the proof-effort.

#### 2.4.3.14 A Generalised Formal Semantic Framework for Smart Contracts

Jiao et al. [110] make use of the K framework [65] with the ambitious goal of creating a formal semantic framework that can be used for any smart contract language. If a semantics for a language is written in K making use of their framework, then security properties can be shown to hold with respect to their blockchain model. If coupled with, for instance, the low-level semantics for the EVM discussed earlier in Section 2.4.2.2, then the compiler no longer needs to be trusted as properties can be shown to hold at both levels. They evaluated their framework by applying their approach to the Solidity language, showing that their semantics for Solidity is consistent with the official compiler on benchmark tests. This approach is commendable, however a limitation is that coupling it with low-level semantics would still seemingly require additional proof effort for each smart contract (similarly to as done in the work described in Section 2.4.3.13). In contrast, in DeepSEA the proof linking the high-level representation to the low-level bytecode is automated.

#### 2.4.3.15 The Move Prover

The Move smart contract language was originally developed for Facebook's Libra/Diem blockchain but now aims to be a ubiquitous language for writing safe code involving assets [111]. The Move Prover enables automated formal verification of Move smart contracts [112] leveraging Boogie [26], [47] and the Z3 SMT solver [44]. The Move language supports specifications phrased as pre- and post-conditions, in addition to invariants over contract-specific data structures or the state of the blockchain. A key advantage of the Move Prover's architecture is that the verification process is automated to the extent that it can be run alongside unit and integration tests. The trusted computing base for the Move Prover involves Boogie and Z3, as well as the necessary translation steps. Nevertheless, this broader trusted computing base helps facilitate the automation that is a strength of their approach.

#### 2.4.3.16 Celestial

Celestial [113] is a smart contract verification framework that translates smart contracts with specifications to F* [114], which facilitates proving the specifications hold. F* enables a mixture of interactive and automated theorem proving (powered by SMT solvers) to be used. Celestial then translates successfully verified smart contracts to Solidity, so that they can be compiled to EVM bytecode. While being able to leverage the proof capabilities of F* is useful, Celestial does not provide a guarantee that the EVM bytecode generated from the Solidity code (generated by Celestial) actually matches the behaviour of the F* version of the smart contract which is verified. Nevertheless, the ability to seamlessly mix interactive and automated theorem proving is a strength of Celestial.

#### 2.4.3.17 Harvey

Harvey [115] is a tool from Consensys [116] that performs greybox fuzz testing, generating inputs to smart contracts randomly in a way which ensures greater code coverage from the tests. Harvey also involves a technique for generating inputs that helps reveal vulnerabilities specific to smart contracts. A strength of Harvey is that it can be run fully-automatically and at scale (Harvey has analysed millions of smart contracts). This sort of approach certainly is useful, however a smart contract that passes thorough fuzz testing from a tool such as Harvey still might have vulnerabilities that were not uncovered by the tests, even if many automatically generated tests are run.

#### 2.4.3.18 Handling Reentrancy using Code Segments

Bräm et al. [117] as well as Albert et al. [118] take an approach to handling reentrancy by modelling reentrant calls that can occur partway through a function, considering code in segments. This is a more precise modelling of reentrancy than the approach taken in this thesis, described in Section 4.1. With that precision comes the ability to reason accurately about locks, for instance. However, the cost of this precision is that it generates additional proof obligations, such as those relating to invariants holding for the segments. Making use of the *Checks-Effects-Interactions Pattern*, as is done in this thesis, leaves the engineer with fewer proof obligations at the expense of some expressivity.

#### 2.4.3.19 SolidiKeY

Ahrendt et al. [119] introduce the tool SolidiKeY, which is an adaptation of the KeY prover [27] (originally a Java verification tool) into a verifier specifically for the Solidity language. The approach of adapting a mature verification tool to handle Solidity verification is admirable, requiring greater effort that translating Solidity into Java for use with the original KeY tool (for example). However, even if the semantics of Solidity programs in SolidiKeY was completely faithful to real-world Solidity, the SolidiKeY tool does not guarantee that the compiled EVM bytecode will behave the same as the verified version of the smart contract in SolidiKeY, necessitating trust in the Solidity compiler.

#### 2.4.3.20 Verification of Hyperledger Fabric Chaincode

Beckert et al. [120] describe verification of smart contracts on the Hyperledger Fabric system [121] which is for “permissioned” blockchains, rather than public blockchains such as Ethereum. Their approach centres around the KeY prover [27] for Hyperledger smart contracts written in Java. Smart contracts are annotated with pre- and post-conditions which can then be proven semi-automatically using the KeY prover. Their approach requires trusting the Java compiler as there are no proofs directly related to the Java bytecode of Hyperledger smart contracts. Nevertheless, it is encouraging to see verification being prioritised across a range of blockchains.

#### 2.4.3.21 MadMax

Grech et al. [122] developed the tool MadMax with a focus specifically on gas-related vulnerabilities. MadMax analysed all smart contracts on the Ethereum blockchain at the time of its publication in 2018, which is made possible by operating directly on bytecode and focusing on a fixed set of specific gas-related properties. While tools such as MadMax are useful for screening for specific vulnerabilities, they do not make it possible to express or prove notions that capture high-level specifications corresponding to showing the correctness of a smart contract. However, the ability to apply tools such as MadMax at scale is a great strength of the approach.

#### 2.4.3.22 EthRacer

Similarly to MadMax, discussed in Section 2.4.3.21, Kolluri et al. [123] developed EthRacer, which also operates on EVM bytecode and targets identifying a specific type of vulnerability. EthRacer focusses on bugs related to the order in which functions of the smart contract are called, and uses techniques to focus the search space and avoid the potential combinatorial explosion as much as possible. EthRacer could be considered a fuzzer, similarly to Harvey (discussed in Section 2.4.3.17). As with MadMax, EthRacer focusses on only specific types of bugs and has its usefulness but falls short of being able to provide strong guarantees about the complete correctness of a smart contract.

### 2.4.4 Other Relevant Works

#### 2.4.4.1 Precise Static Modelling of Ethereum Memory

Lagouvardos et al. [124] describe developing a system that automatically analyses EVM bytecode with a focus on reconstructing higher-level concepts related to Ethereum's smart contract memory. This approach greatly helps with the analysis of programs for which only the bytecode is known, making further analysis of various vulnerabilities as well as gas-cost estimation much more precise. This work's focus on reconstructing higher-level concepts from bytecode is useful when only the bytecode is accessible, however when both the source code and bytecode is known (especially when coupled with a verified compiler) there is no need for decompilation.

#### 2.4.4.2 Study of Inline Assembly in Solidity Smart Contracts

Chaliasos et al. [125] examine the extent of the usage of inline assembly in Solidity smart contracts and note that it was used in 23% of analysed smart contracts. Inline assembly allows Solidity programmers to include functionality that is supported by the EVM but not included as part of the Solidity language. Assuming that all uses of inline assembly were necessary, this would rule out at least 23% of smart contracts for verification techniques requiring the use of a higher-level language without inline assembly. DeepSEA is an example of this. On the other hand, that 77% of smart contracts might be compatible with such verification technqiues is perhaps an encouraging result of this study of the use of inline assembly.

#### 2.4.4.3 Refinement-based Smart Contract Upgradeability

Antonino et al. [126] describe one of the natural benefits that could arise from having formal specifications for smart contracts, along with associated correctness proofs. They describe a shift from "code is law" to "specification is law", by which they mean that smart contracts would be able to be updated, as long as new versions continue to provably meet a specification. They also highlight that specifications for smart contracts can also be updated, as long as new specifications provably refine the existing ones. This opens up great opportunities for enabling smart contract to be updated without the risks that usually entails, because users of the smart contract will always be able to rely on the behaviour of the smart contracts they use as specified. Their work makes use of the `solc-verify` prover [127], targeting Solidity smart contracts. With future work, DeepSEA smart contract specifications could also be used in this manner.

### 2.4.5 Remarks

The selected works discussed above gives a sample of the range of research being undertaken to verify the correctness of smart contracts. Some approaches involve modelling smart contracts at a high level and some involve formalising the semantics of a low-level smart contract language. DeepSEA links both these approaches, enabling reasoning about smart contracts at a high level while grounding the meaning of the smart contract in the low-level semantics of the EVM. This allows us to not lose any information in the compilation process (such as when analysing EVM bytecode directly), aiding with verification.

Listing 2.1: Example DeepSEA Smart Contract Source Code

```
object signature trivial = {
    const boolToInt : bool -> int;
    storeBoolToInt : bool -> int;
    getStored : unit -> int
}

object Trivial : trivial {
    let result : int := 0
    let boolToInt b = if b then 1 else 0
    let storeBoolToInt b =
      if b then result := 1 else result := 0;
      result
    let getStored () = result
}

layer CONTRACT  = {
    o = Trivial
}
```

## 2.5 Brief Overview of the DeepSEA Language

This section gives a brief overview of the DeepSEA language, which the work of this thesis built upon. For a more complete overview see the DeepSEA Language Reference [103] which this section summarises.

DeepSEA [7] includes a range of primitive datatypes including `int`, `bool`, `array`, and `mapping`. A `mapping` is a hash mapping indexed by a numerical type and can be nested to allow for multidimensional mappings. Each smart contract defines functions which have a type signature that must be provided in the `object signature` section. The type signature is checked by the DeepSEA compiler, raising an error if there is a mismatch between the actual type of the function and the declared signature. Smart contracts can be organised into layers and only higher layers can call the functions of lower layers. In this thesis we will only be dealing with smart contracts defined as a single layer.

A trivial example contract is shown in Listing 2.1 (throughout this thesis, DeepSEA source code has a blue background). This contract has one storage variable named `result` of type `int` and defines three functions:

- One called `boolToInt` which converts a Boolean value into the corresponding integer of `0` or `1`. The `const` declaration in its type signature (in the `object signature` section) implies that this function does not write to storage variables.
- The second called `storeBoolToInt` additionally stores the result of converting from `bool` to `int` in the storage variable `result`.
- Finally, the `getStored` function returns the value of the stored `int`, which initially has the value `0`.

The last value in a function, or in a branch of an ‘if’ statement, is the value returned by the function. There is no “return” keyword. The body of the `layer CONTRACT` section defines the layout of the layers, in this case setting up a single layer holding the `Trivial` object.

Most of the other features of the DeepSEA language used in this thesis have intuitively clear meanings, but further explanation will be provided as necessary. Even this brief overview should make it clear that the programs are written at a reasonably high level of abstraction. This is one of the key benefits of using the DeepSEA language, because the approach permits both writing and reasoning about programs at a high level of abstraction, while providing strong guarantees that the properties shown to hold at the high level of abstraction hold also for the low level machine code.

## 2.6 Research Questions

The main research question behind this thesis is:

- *How can we know that a given smart contract is correct?*

This question cannot simply be answered by saying "*using formal methods*". To properly answer the question, it is necessary to delve into specific smart contracts and the 'mathematics of the application domain' that the smart contracts relate to. Then providing definitions, theorems and proofs to go along with the smart contract that are each central to answering the above research questions. The hope is that the lessons learnt from this process are then generalisable to proofs about other smart contracts.

Research questions for future work include:

- *How can we know that interrelated smart contracts behave correctly?*
- *How can we most effectively answer these questions for an increasing range of smart contracts?*

For these questions it would be necessary to consider the mathematics of the application domains of each individual contract *plus any additional domains related to their interrelated action* - with the associated definitions, theorems and proofs.

To best answer the final question, providing definitions, theorems and proofs that *are more broadly applicable* become central along with approaches to formalisation and proof that have wider applicability than a specific application domain. Also, exploring other approaches such as contract templates that may be very effective in answering the question of correctness for a certain set of smart contracts would form a part of answering this question.

# Chapter 3

# Theorem Proving Concepts in the Coq Proof Assistant

Coq is described as a "*formal proof management system*" [8]. A system like Coq typically involves a language expressive enough to capture both specifications and implementations as well as giving us the ability to write proofs that can be machine-checked. Facilitating machine-checked proofs is the main purpose of the system. There are a number of systems similar to Coq such as Isabelle/HOL [28], PVS [128], Idris [129] and Agda [130]. The choice to use the Coq Proof Assistant was primarily driven by the existing codebase of the DeepSEA project [99] using Coq.

In this chapter I will introduce the Coq Proof Assistant as well as the main style of presenting proofs that will be used throughout the thesis. I will then discuss some key theoretical concepts which Coq facilitates reasoning about.

## 3.1 Coq Fundamentals

This section provides an overview of the fundamentals of the Coq Proof Assistant [8] which is the primary proof tool used in this thesis.

### 3.1.1 Example: Natural Numbers in Coq

The specification language of Coq is Gallina. In Gallina, we can define the notion of a natural number as shown in the following example.

```
Inductive nat :=
 | O : nat
 | S : nat → nat.
```

This defines a data type *nat* as well as automatically defining the following induction principle *nat_ind*. The induction principle *nat_ind* takes a property $P$, which is a logical proposition taking one natural number as input, and gives a property of the type shown in the second line of the example. The second line can be thought of as a lemma stating the principle of induction for the *nat* type specialised to the property *P*. `Prop` is the Coq type for logical propositions. This induction principle can then be applied using the tactic `induction`. In Coq, tactics are commands which manipulate the proof state in order to prove a goal, as will be discussed later.

*nat_ind*: $\forall\ P : nat \rightarrow$ `Prop`,
$\quad P\ O \rightarrow (\forall\ n : nat,\ P\ n \rightarrow P\ (S\ n)) \rightarrow \forall\ n : nat,\ P\ n$

We can also see the same definition as it is rendered in the Coq Proof Assistant. The `Check` command prints the *type* of an identifier. Here, the *nat* induction principle is printed.

```
Check nat_ind.
```

```
nat_ind
     : forall P : nat -> Prop,
       P 0 ->
       (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

Defining *nat* alone does not define any of the operations on the natural numbers, such as addition. Here we define *plus* as a recursive function. We make use of a `match` expression to progressively move the uses of the successor function, *S*, from the first argument onto the result until the first argument is *O*. This also involves pattern-matching which allows us to match a pattern such as a number of the form *S n'* and then have access to the inner number *n'*. We can also define an appropriate infix notation for plus.

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

```
Infix "+" := plus.
```

In many programming languages, the definitions above would be built into the language, however in Coq many constructs that would typically be built in are written in Gallina itself. A key purpose of this is to make all these constructs able to be reasoned about in proofs. Coq's rich dependently typed language allows for logical statements such as $\forall\ x{:}nat,\ x = x + O$ to be captured as types, with their proof being shown by providing an object of that type (to explore this further, the reader is encouraged to look into the Curry-Howard correspondence such as discussed in [131, Chapter: The Curry-Howard Correspondence]). We can use Gallina to provide these "proof objects" however there is a more convenient language, `Ltac`, which could be considered to be a scripting language designed for producing "proof objects". There is only a stylistic difference between the keywords `Theorem`, `Lemma`, and `Definition`.

In the proofs below, we show that adding *O* to any *nat* is equal to the original *nat*. The first proof is trivial, by considering the definition of *plus* we can see why. The first branch of the match expression in *plus*, $\mid O \Rightarrow m$, is triggered which causes $x$ to be returned immediately. This means that the `Ltac` command "reflexivity" (which is called a *tactic*) is able to solve the goal by simplifying the equation and then solving the goal of the form $x = x$.

```
Lemma plus_O_n : ∀ x:nat, x = O + x.
Proof.
  reflexivity.
Qed.
```

The second proof is shown similarly to how it would appear in the Coq Proof Assistant. This proof is not quite as trivial and requires the use of the `induction` tactic, which makes use of the *nat_ind* induction principle shown earlier.

NOTE: The content in the grey boxes outlined by a grey border (such as those in the proof below) are automatically generated by the tool Alectryon [132]. When using a Coq IDE this information would appear in a window to the side and change as the user steps through the proof just as the content in the grey boxes changes as the proof progresses. The variables and hypothesis in the context appear above the black line. The current proof goal appears below the black line.
In some circumstances there will be multiple goals that appear at once, for example after a use of `induction`. In these cases, there will be multiple sets of hypotheses and goals, with each pair surrounded by a grey border.
Occasionally a `Check` or `Print` command will be used. These also show Coq output but do not include a black line and everything in the grey box is simply the output of the command (not necessarily a hypothesis or goal).

```
Lemma plus_x_O : forall x:nat, x = x + O.
Proof.

intros.
```

```
x: nat
________________________________________
x = x + O
```

```
induction x.
```

```
________________________________________
O = O + O
```

```
x: nat
IHx: x = x + O
________________________________________
S x = S x + O
```

First we show $P\ O$ where $P$ is $\forall\ x{:}nat,\ x = O + x$. Remembering that the type of $P$ is $nat \rightarrow$ `Prop` we see the use of dependent types here. The $P\ O$ case is trivial, solved by `reflexivity` after simplification using the definition of *plus*.

```
- (* Case: P O *)
simpl.
```

```
________________________________________
O = O
```

```
reflexivity.
```

When we move on to the $P\ (S\ x')$ we now need to rewrite the goal from $S\ x = S\ (x + O)$ to $S\ x = S\ x$ using the inductive hypothesis $P\ x$ which is $n = x + O$. The goal can then be solved by `reflexivity`. This demonstrates the use of an induction principle to prove a straightforward goal.

```
- (* Case: P x -> P (S x) *)
simpl.
```

```
x: nat
IHx: x = x + O
______________________________
S x = S (x + O)
```

```
rewrite <- IHx.
```

```
x: nat
IHx: x = x + O
______________________________
S x = S x
```

```
reflexivity.

Qed.
```

### 3.1.2 Example: Even Numbers

In Coq, we can express both the specification and implementation of functions. As an example of the difference between a specification and an implementation we now consider a function which decides if a number is even or not. First, we define the mathematical notion of evenness. This describes *what* evenness is but does not describe a deterministic method for deciding if a number is even or not. It provides us with a precise *specification* for evenness by fully describing the notion of evenness, even though it is also possible to describe the same notion of evenness in other ways.

Note that here we are using *nat* as defined by the Coq standard library, which includes a definition for multiplication and allows use of standard numeral notation.

```
Definition Even (n : nat) : Prop :=
  ∃ m, n = 2 × m.
```

Next, we define an actual implementation that can decide if a number is even or not. Intuitively it *looks* correct, but our goal is to *prove* that it satisfies the specification above.

```
Fixpoint evenb (n : nat) : bool :=
  match n with
    | 0 ⇒ true
    | 1 ⇒ false
    | S (S n) ⇒ evenb n
  end.
```

Another way of defining *evenb* is to use the `fix` primitive instead of `Fixpoint` as shown here.

```
Definition evenb' : nat → bool :=
  fix evenb' (n:nat) : bool :=
    match n with
    | 0 ⇒ true
    | 1 ⇒ false
    | S (S n) ⇒ evenb' n
  end.
```

In order to help with the proof of the correctness of *evenb*, we can first define a form of induction for natural numbers which has two base cases (for each of 0 and 1) and where the inductive step goes forward two increments at a time rather than the usual one. Here is the definition, using the `fix` primitive as before. This highlights the similarity between processing data and processing proofs.

NOTE: Here, *ev* stands for evidence. For example, the variable *ev_0* is the evidence that the property *P* is true for the base case *P* 0.

```
Definition nat_ind_SS
  (P : nat → Type)
  (ev_0 : P 0)
  (ev_1 : P 1)
  (ev_SS : ∀ n, P n → P (S (S n)))
    : ∀ n, P n :=
    fix nat_ind_SS n :=
      match n with
      | 0 ⇒ ev_0
      | 1 ⇒ ev_1
      | S (S m) ⇒ ev_SS m (nat_ind_SS m)
    end.
```

Here is the proof of the correctness of *even_b*, making use of the *nat_ind_SS* induction principle.

```
Theorem evenb_correct :
  forall n,
    evenb n = true <-> Even n.
Proof.
```

```
────────────────────────────────────────
forall n : nat, evenb n = true <-> Even n
```

The proof has the structure of "if and only if" so we have two directions to prove. To split the goal into the two subgoals we use the `split` tactic.

```
split.
```

```
n: nat
────────────────────────────────────────
evenb n = true -> Even n
```

```
n: nat
────────────────────────────────────────
Even n -> evenb n = true
```

We can see above that the `split` tactic has generated two subgoals for us, each with an arbitrary $n$ of type *nat* in the context. First we aim to prove that when our *evenb* function evaluates to true, this implies the mathematical definition of evenness holds. We use the "-" bullet to focus on the first goal.

```
- (* Case: "->" *)
```

```
n: nat
____________________________________
evenb n = true -> Even n
```

Here, we use our custom induction principle which generates three subgoals for us, with the appropriate variables and hypotheses (if any) above the black line.

```
induction n using nat_ind_SS.
```

```
____________________________________
evenb 0 = true -> Even 0
```

```
____________________________________
evenb 1 = true -> Even 1
```

```
n: nat
IHn: evenb n = true -> Even n
____________________________________
evenb (S (S n)) = true -> Even (S (S n))
```

For the base case, we must show that 0 is *Even*. So, based on the definition of evenness, we use the `exists` tactic to provide the value (namely 0) which when multiplied by 2 is 0. We focus on the first base case using the "+" bullet.

```
+ (* Base case: 0 *)
exists 0.
```

```
H: evenb 0 = true
____________________________________
0 = 2 * 0
```

This simplifies to $0 = 0$ based on the definition of multiplication from the standard library.

```
simpl.
```

```
H: evenb 0 = true
____________________________________
0 = 0
```

Now the goal is solvable by the tactic `reflexivity`.

```
reflexivity.
```

For the second base case, we are asked to show that 1 is *Even* (which is impossible). So instead we use the hypothesis that $\textit{evenb}\ 1 = \textit{true}$ to demonstrate that this scenario is impossible, via a proof by contradiction. We can do this by noting that simplifying *evenb* 1 gives us $\textit{false} = \textit{true}$ as a hypothesis. Proofs by contradiction like this are common in DeepSEA proofs, and useful especially where the branch of the execution of the smart contract under consideration is impossible to reach. To focus on the next goal we use the "+" bullet again.

```
+ (* Base case: 1 *) intros.
```

```
H: evenb 1 = true
________________________________________
Even 1
```

```
simpl in H.
```

```
H: false = true
________________________________________
Even 1
```

The tactic `discriminate` allows us to make use of information about the constructors of an inductive type to determine if certain hypotheses (such as *false* = *true*) are impossible.

For example a *bool* as defined below must, naturally, either be *true* or *false* (never both) and it clearly isn't the case that *false* = *true*.

`Inductive` *bool* :=
  | *true*
  | *false*.

Due to this, `discriminate` is able to discharge any goal from the hypothesis *false* = *true*, in particular the goal *Even* 1 can be discharged. To make it clear that we are indeed invoking ex falso quodlibet and not somehow directly proving *Even* 1, we can first use the *exfalso* tactic which invokes ex falso quodlibet and replaces the current goal with *False* (logical impossibility). `discriminate` still discharges the goal.

```
exfalso.
```

```
H: false = true
________________________________________
False
```

```
discriminate.
```

Now we move on the final case of the "→" direction. Notice in particular the hypothesis *IHn* which is the inductive hypothesis added to the context by the `induction` tactic based on our *nat_int_SS* induction principle.

```
+ (* Inductive case: +2 *) intros.
```

```
n: nat
IHn: evenb n = true -> Even n
H: evenb (S (S n)) = true
________________________________________
Even (S (S n))
```

We can show *evenb n* = *true* by simplifying the hypothesis *H* via the tactic `simpl in` * (which due to the "*" simplifies everything).

```
simpl in *.
```

```
n: nat
IHn: evenb n = true -> Even n
H: evenb n = true
______________________________________
Even (S (S n))
```

We can then use the *IHn* and *H* hypotheses to show *Even n*.

```
apply IHn in H.
```

```
n: nat
IHn: evenb n = true -> Even n
H: Even n
______________________________________
Even (S (S n))
```

Now that we know $n$ is *Even*, we recall the definition of evenness, which we can see here if we `unfold` it in *H*. And it will be helpful to unfold *Even* in the goal as well.

```
unfold Even in H. unfold Even.
```

```
n: nat
IHn: evenb n = true -> Even n
H: exists m : nat, n = 2 * m
______________________________________
exists m : nat, S (S n) = 2 * m
```

Since we know that there exists an $m$ as described in the hypothesis $H$ we can use the `destruct` tactic to 'destructure' the hypothesis and obtain a particular natural number $x$ as shown below. This is existential elimination.

```
destruct H.
```

```
n: nat
IHn: evenb n = true -> Even n
x: nat
H: n = 2 * x
______________________________________
exists m : nat, S (S n) = 2 * m
```

Now we notice that $x + 1$ is a suitable candidate for the variable $m$ in the goal.

```
exists (x + 1).
```

```
n: nat
IHn: evenb n = true -> Even n
x: nat
H: n = 2 * x
______________________________________
S (S n) = 2 * (x + 1)
```

Now we can use the hypothesis, *H*, to make use of information originating from our inductive hypothesis to all but solve the goal.

```
rewrite -> H.
```

```
n: nat
IHn: evenb n = true -> Even n
x: nat
H: n = 2 * x
______________________________________
S (S (2 * x)) = 2 * (x + 1)
```

```
simpl.
```

```
n: nat
IHn: evenb n = true -> Even n
x: nat
H: n = 2 * x
______________________________________
S (S (x + (x + 0))) = x + 1 + (x + 1 + 0)
```

After simplification, it is clear to see that both sides of the equation in the goal are equal, noting that $S$ is the successor function and is equivalent to "+ 1". To save some tedious rewriting using the lemmas for associativity and commutativity of plus we can use the *lia* tactic, which stands for linear integer arithmetic. *lia* solves that class of arithmetic problems and is able to solve our goal.

```
lia.
```

Next, we use the custom induction principle for the other direction of the implication. We assume that $n$ is Even and aim to prove that *evenb* $n =$ *true*.

```
- (* Case: "<-" *)
```

```
n: nat
______________________________________
Even n -> evenb n = true
```

```
induction n using nat_ind_SS.
+ (* Base case: 0 *) intros.
```

```
H: Even 0
______________________________________
evenb 0 = true
```

```
simpl.
```

```
H: Even 0
______________________________________
true = true
```

```
reflexivity.
+ (* Base case: 1 *) intros.
```

```
H: Even 1
______________________________________
evenb 1 = true
```

*evenb* 1 actually equals *false* of course so it is impossible to prove our goal. We can use the tactic *exfalso* to make this clear by replacing our goal with logical impossibility as we did in the base case for 1 for the "$\rightarrow$" direction. But we don't need to.

```
destruct H.
```

```
x: nat
H: 1 = 2 * x
________________________________________
evenb 1 = true
```

Here, we can see that the hypothesis $1 = 2 \times x$ is impossible. The tactic *lia* is powerful enough to make use of this impossible hypothesis and discharge our impossible goal.

```
lia.
+ (* Inductive case: +2 *) intros.
```

```
n: nat
IHn: Even n -> evenb n = true
H: Even (S (S n))
________________________________________
evenb (S (S n)) = true
```

Here we assert and prove a lemma in-line, in this case that $n$ is *Even*. We can also use braces to focus on a subproof, for a similar effect to bullets.

```
assert(Even n). {
```

```
n: nat
IHn: Even n -> evenb n = true
H: Even (S (S n))
________________________________________
Even n
```

```
destruct H.
```

```
n: nat
IHn: Even n -> evenb n = true
x: nat
H: S (S n) = 2 * x
________________________________________
Even n
```

Here we need to destructure $x$ into the cases of whether it is 0 or of the form *S x'* so that the definition of multiplication can be used to simplify the equations. We can keep the knowledge of which case it is as a hypothesis by adding the *eqn:Case*. The "`as` [|*x'*]" syntax means that in the first case the argument to the constructor *O* will not be given a name (*O* has no arguments) and for the *S* case the argument to *S* will be given the name *x'*.

```
destruct x as [|x'] eqn:Case.
*
```

```
n: nat
IHn: Even n -> evenb n = true
x: nat
Case: x = 0
H: S (S n) = 2 * 0
________________________________________
Even n
```

```
exfalso. clear -H. simpl in H.
```

```
n: nat
H: S (S n) = 0
________________________________________
False
```

In this case the goal is impossible, because $n + 2$ cannot equal 0, this is evident from the constructors of the type *nat* so we can use the `discriminate` tactic to discharge the goal.

```
discriminate.
* unfold Even.
```

```
n: nat
IHn: Even n -> evenb n = true
x, x': nat
Case: x = S x'
H: S (S n) = 2 * S x'
________________________________________
exists m : nat, n = 2 * m
```

Here we can see that from *H*, *x'* is the appropriate value for *m*.

```
exists x'.
```

```
n: nat
IHn: Even n -> evenb n = true
x, x': nat
Case: x = S x'
H: S (S n) = 2 * S x'
________________________________________
n = 2 * x'
```

Now the *lia* tactic makes use of its regular simplification and rewriting, plus the injectivity of the successor function $S$ to conclude the goal from *H*. Note that *H* is equivalent to $S\ (S\ n) = S\ (S\ (2 \times x'))$.

```
lia. }
```

Making use of what we have just proved as *H0* we can use the inductive hypothesis to conclude that *evenb n = true*. By the definition of *evenb*, the goal simplifies to exactly this, so our proof is done.

```
apply IHn in H0.
```

```
n: nat
IHn: Even n -> evenb n = true
H: Even (S (S n))
H0: evenb n = true
________________________________________
evenb (S (S n)) = true
```

```
simpl.
```

```
n: nat
IHn: Even n -> evenb n = true
H: Even (S (S n))
H0: evenb n = true
________________________________________
evenb n = true
```

```
assumption.

Qed.
```

To better appreciate the structure of a proof it can be easier to see it as a whole. Here is the *evenb_correct* proof in its entirety. We can clearly see the two bullets for the "→" and "←" directions as well as the three sub-bullets for each of the cases related to the use of our induction principle. At the same time, in this form it is hard to follow the actual details of the proof.

```
Theorem evenb_correct' :
  forall n, evenb n = true <-> Even n.
Proof.

split.
  - (* Case: "->" *)
    induction n using nat_ind_SS.
    + (* Base case: 0 *)
      exists 0. simpl. reflexivity.
    + (* Base case: 1 *)
      intros.  simpl in H. exfalso. discriminate.
    + (* Inductive case: +2 *) intros.
      simpl in *. apply IHn in H.
      unfold Even in H. unfold Even.
      destruct H.
      exists (x + 1).
      rewrite -> H. simpl.
      lia.
  - (* Case: "<-" *)
    induction n using nat_ind_SS.
    + (* Base case: 0 *) intros. simpl.  reflexivity.
    + (* Base case: 1 *) intros. destruct H. lia.
    + (* Inductive case: +2 *)
      intros.
      assert(Even n). {
        destruct H.
        destruct x as [|x'] eqn:Case.
        * exfalso. clear -H. simpl in H. discriminate.
        * unfold Even. exists x'. lia.
      }
      apply IHn in H0. simpl.
      assumption.

Qed.
```

### 3.1.3 Remarks

The examples and concepts explored above are intended to give a taste of what is involved in a proof in the Coq Proof Assistant and to demonstrate the difference between a specification and an implementation which, in this case, *decides* if a number is even. The proof context visualisations give an insight into what stepping through a proof in the Coq Proof Assistant is like. This thesis contains the relevant proof visualisations within the PDF, however if the reader is interested in stepping through the proofs for themselves they are welcome to. To interactively view the proofs in this thesis and step forwards and backwards through them at will please see Appendix A.

For further reading and understanding of the Coq Proof Assistant, the Software Foundations series, beginning with Logical Foundations [131], is highly recommended.

## 3.2 Monads in Coq

In this section I will introduce monads in Coq, giving a taste of how monads can be used as a tool for developing a proof system for smart contracts. Particular monads can be considered formal models of programs, which can then be reasoned about in Coq. When used to model programs, monads formalise the idea of the "context for a computation". For example, monads can be used to model the behaviour of mutable variables in a program, including reading and writing to variables. More generally, monads can formalise the notion of threads and other context relevant to computations. Here, we will focus primarily on the modelling of state, building up to the state transformer monad.

Smart contracts are normally imperative programs and DeepSEA smart contracts are written in the imperative style. However, Coq's Gallina is a functional language and so there needs to be a way of representing DeepSEA smart contracts as pure functions. This is done by the use of monads, which are a common concept in functional languages. In a DeepSEA `.ds` file, each function of the imperative style program is automatically rewritten so that it has a monadic type, in particular a state transformer monad where the state is a record holding the variables of the contract and the monad for the state transformer is the maybe monad so that errors in the execution of a contract can propagate nicely. Here we will introduce, via our own work, a presentation of monads in Coq that bears some similarities to how they are defined in DeepSEA itself, though it is worth noting that the development of the use of monads in DeepSEA was pre-existing work.

### 3.2.1 Monad

First we introduce a *typeclass* which describes the concept of a collection of types that have certain properties, in this case that every *Monad* has *ret* and *bind* functions. Since we are using a proof assistant we can also require that proofs of each of the three monad laws are provided along with any instance of the monad typeclass.

```
Class Monad (M : Type → Type) :=
{
  ret : ∀ {T : Type}, T → M T;
  bind : ∀ {T U : Type}, M T → (T → M U) → M U;
  left_identity_prf : ∀ {T U : Type} a f, @bind T U (ret a) f = f a;
  right_identity_prf : ∀ {T : Type} (m : M T), @bind T T m ret = m;
  associativity_prf : ∀ {T U V : Type} (m : M T) (g : T → M U) (h : U → M V),
    @bind U V (@bind T U m g) h = @bind T V m (fun x ⇒ @bind U V (g x) h)
}.
```

We define the standard notation for *bind* and the Kleisli composition operator here.

```
Notation "a >>= b" := (bind a b) (at level 58, left associativity).
Notation "f >=> g" := (fun x => (f x >>= g)) (at level 58, left associativity).
```

The Kleisli composition operator takes a function that wraps a value of some type into a value of a monadic type instead of directly requiring a monadic value like *bind*. The difference between the type signature of *bind* and the Kleisli operator is that instead of $M\ T$ in *bind*, we have $V \to M\ T$ for the Kleisli operator. In the Kleisli operator, a function is used to apply the function of type $V \to M\ T$ which makes it possible to now call bind because this produces a value of type $M\ T$. This happens to allow us to see why the third law is called associativity more clearly. Next, we prove a similar lemma to the associativity law that shows this. The proof is so similar, in fact, that the key part of the proof is to simply apply the Monad associativity law which here is called *Monad.associativity_prf*.

There are two new ideas in this proof. Firstly the '{*Monad* $m$} means that $m$ is an instance of the *Monad* typeclass and the curly brackets signify that this will be implicitly inferred from the context. Secondly, the import of functional extensionality allows us to assume that two functions are equal when they return the same result for every element in their domain. In other words, it equates extensional equality with intensional equality. This is not a standard part of Coq's logic because doing so makes intensional equality more difficult to work with in Coq. But it is useful here to make the statement of the lemma more elegant and functional extensionality is used in various places throughout the DeepSEA codebase, so it is worth introducing it here. We could instead add an extra argument of type $(x : A)$ and rewrite the statement of the lemma to `((f >=> g) >=> h) x = (f >=> (g >=> h)) x` but there is more benefit than harm here in adding the functional extensionality axiom. This makes the associativity plain to see in `(f >=> g) >=> h = f >=> (g >=> h)`.

```
Require Import Coq.Logic.FunctionalExtensionality.
Lemma kleisli_assoc : forall (A B C D : Type) (m : Type -> Type) `{Monad m}
  (f : A -> m B) (g : B -> m C) (h : C -> m D),
  (f >=> g) >=> h = f >=> (g >=> h).
Proof.

  intros. (* Hypotheses hidden for brevity. *)
```

```
____________________________________________
f >=> g >=> h = f >=> (g >=> h)
```

```
  apply functional_extensionality.
```

```
____________________________________________
forall x : A, f x >>= g >>= h = f x >>= (g >=> h)
```

The notation system moves the $x$ to $f\ x$, but it is the same as `((f >=> g) >=> h) x = (f >=> (g >=> h)) x`.

```
  intros.
  apply Monad.associativity_prf.

Qed.
```

Now that a notation is defined for *bind* we can see the statements of the monad laws more clearly as shown below. The `Search` command searches for defined items which match the given type and can include wildcards. Here we see the monad laws in a clearer notation using `>>=` and `>=>`. It is somewhat similar to Haskell's Hoogle but in Coq it is most often used for searching for lemmas and theorems.

```
Search (forall a f, ret a >>= f = f a).
```

```
left_identity_prf:
  forall {M : Type -> Type} {Monad : Monad M}
    {T U : Type} (a : T) (f : T -> M U),
  ret a >>= f = f a
```

```
Search (forall m, m >>= ret = m).
```

```
right_identity_prf:
  forall {M : Type -> Type} {Monad : Monad M}
    {T : Type} (m : M T), m >>= ret = m
```

```
Search (forall m g h, m >>= g >>= h = m >>= (g >=> h)).
```

```
associativity_prf:
  forall {M : Type -> Type} {Monad : Monad M}
    {T U V : Type} (m : M T) (g : T -> M U)
    (h : U -> M V), m >>= g >>= h = m >>= (g >=> h)
```

### 3.2.2 Maybe Monad

We can now define the *Maybe* type and prove that it is indeed a Monad.

```
Inductive Maybe (T : Type) : Type :=
  | None : Maybe T
  | Some : T → Maybe T.
```

When we declare *Maybe* as a monad, we are now also required to provide a proof of the monad laws following the instance declaration. We define *ret* and *bind* in plain Gallina but construct the proofs using Coq's tactic language below it. The use of the `refine` attribute here allows us to 'fill in' the underscores using Coq's proof mode immediately after the main definition.

```
#[refine] Instance : Monad (Maybe) :=
{
  ret T a := Some a;
  bind T U m f :=
    match m with
        None => None
      | Some a => f a
    end;
  left_identity_prf := _;
  right_identity_prf := _;
  associativity_prf := _
}.
Proof.
```

The bullets below each correspond to a monad law.

```
  - (* Left identity proof *)
```

```
________________________________________
forall (T U : Type) (a : T) (f : T -> Maybe U),
f a = f a
```

This is trivial to prove because the goal has been simplified already to $f\ a = f\ a$. This simplification happened because for the Maybe monad *ret* is *Some* and for the case of *Some*, *bind* returns $f\ a$. As a result `ret a >>= f` is $f\ a$.

```
    intros.
    reflexivity.
  - (* Right identity proof *)
```

For the right identity proof we need to consider the case of *None* and *Some* separately, so we use the tactic `destruct`. Once this is done the goals simplify based on the definition of *bind* to trivially true equalities.

NOTE: When it aids clarity for reading, some hypotheses or goals may be hidden from the output generated by Coq during proofs. Sometimes this will be noted, but not always. This will only be done when it should not introduce confusion.

```
    intros.
    destruct m.
```

```
________________________________________
None = None
________________________________________
Some t = Some t
```

```
    + reflexivity.
    + reflexivity.
  - (* Associativity proof. *)
```

Similarly to the right identity proof, the associativity proof follows from splitting the *Some* and *None* cases, once this is done the goals are trivially true.

```
    intros.
    destruct m.
    +
```

```
________________________________________
None = None
```

```
reflexivity.
    +
```

```
________________________________________
match g t with
| None => None
| Some a => h a
end =
match g t with
| None => None
| Some a => h a
end
```

```
reflexivity.

Defined.
```

### 3.2.3 State/Action Monad

Now we move on to the state monad[1]. Here we will call it the action monad instead, because by examining the definition for the type *Action* it becomes clear that it represents an action much more directly than being associated with state. Note that the use of the term "action monad" here is unrelated to where terms such as

[1]Parts of this subsection were inspired by the "Tasteful stateful computations" section in Learn You a Haskell for Great Good! - A Beginner's Guide [133, Chapter: For a Few Monads More].

monad action are used elsewhere. An *Action* transforms a system (represented by the state of type *St*) from one state $st_i$ to the next state $st_{i+1}$ and as this is done the action makes available a value of type *Result*. The action returns this result and state wrapped as a tuple. The *Action* type is like a chunk of code in a program representing an action that transforms state into a result and state tuple. Here is the definition of *Action*, we then show *Action St* is a *Monad* for any type *St*.

`Definition` *Action* (*St Result* : `Type`) : `Type` := *St* → (*Result* × *St*).

To be precise, *Action* alone is not a *Monad*. Instead, for all types *St*, *Action St* is a *Monad*. A monad must be of type `Type` → `Type` and *Action* is of type `Type` → `Type` → `Type`. However, we will still use the terminology "Action monad" to mean *Action St* where *St* is clear from the context or assumed to exist.

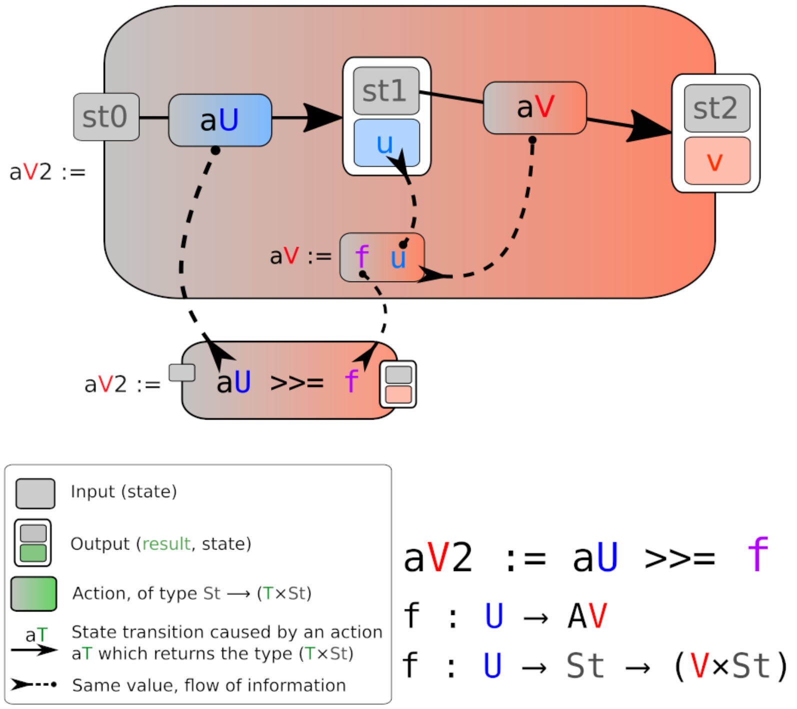
In Coq, there is no need for a *runState* wrapper since we can pass types as arguments when we are defining new types, however a more standard implementation of this monad (such as in DeepSEA) would use a *runState* wrapper.

Similarly to the Maybe monad, we define *ret* and *bind* as done standardly and then prove the Monad laws for the Action monad.

*ret* keeps the state the same and puts the value passed in as the result of the action.

*bind* is more complex. Figure 3.1 gives a pictorial outline of the definition of bind.

Figure 3.1: Diagram representing the definition of bind for the action monad[2]


aV2 :=
st0
aU
st1
u
aV
st2
v
aV := f u
aV2 := aU >>= f
Input (state)
Output (result, state)
Action, of type St ⟶ (T×St)
aT
State transition caused by an action aT which returns the type (T×St)
Same value, flow of information
aV2 := aU >>= f
f : U → AV
f : U → St → (V×St)

²This diagram [134] is based on a diagram by Randall Britten [135]. Used and remixed under CC BY-SA 3.0 Deed.

The boxes with a gradient of colour are each instances of the type *Action St T* for some result type *T*, and *Action St* is an instance of the monad typeclass which will be proved shortly. The dashed arrows represent a term being 'moved around'. The solid arrows represent the application of a function.

First, apply $aU$ to *st0*. $aU$ is of type *Action St U* so it is a function of type $St \to (U \times St)$. Secondly, apply $f$ to the $u$ obtained from the first step. This gives a function of type $St \to (V \times St)$, we name this function $aV$. Now, apply $aV$ to the state obtained from the first step. This gives a tuple of type $(V \times St)$. And this is what bind returns. So overall $aU$ >>= $f$ is of type *Action St V* so it is a function of type $St \to (V \times St)$.

In the lines of a program analogy, we have taken a line of code $aU$ and a wrapper $(f)$ around a second line of code $aV$ (which has $u$ available to it) and produced a new line of code *aV2* which returns the same result type as $aV$.

Coq can be used to define regular functions using the tactic mode normally reserved for proofs. In this approach, we define the type signature of the function we want to define and then omit the body of the definition. We then use tactics to manipulate the 'proof state' just like in a proof. The reason for doing this here is that it allows us to carefully step through the innards of the *bind* definition and explain what is going on more clearly than can be done with a traditional definition.

```
Definition bind' {St U V : Type}
  (aU : Action St U) (f : U -> Action St V) : Action St V.
Proof.
```

First, we note that the type *Action St U* type expands out to $St \to U \times St$. Similarly, for *Action St V*.

```
unfold Action in *.
```

```
St, U, V: Type
aU: St -> U * St
f: U -> St -> V * St
____________________________________
St -> V * St
```

Now we notice that we have an additional argument available to us, because after unfolding our goal *Action St U* the goal becomes $St \to V \times St$. Now, we can introduce this value of type *St* on the left of the arrow, giving it the name *st0*.

```
intro st0.
```

```
St, U, V: Type
aU: St -> U * St
f: U -> St -> V * St
st0: St
____________________________________
(V * St)%type
```

Now that we have something of type *St* available, we plug this in to $aU$ to obtain a tuple which we call $(u,\ st1)$.

```
pose proof (aU st0) as (u, st1).
```

```
St, U, V: Type
aU: St -> U * St
f: U -> St -> V * St
st0: St
u: U
st1: St
________________________________________
(V * St)%type
```

Now we can obtain an action similar to $aU$ but which will return a tuple of type $(V \times$ by applying $f$ to $u$.

```
pose proof (f u) as aV.
```

```
St, U, V: Type
aU: St -> U * St
f: U -> St -> V * St
st0: St
u: U
st1: St
aV: St -> V * St
________________________________________
(V * St)%type
```

Now we can apply $aV$ to *st1* in order to obtain the next result and state pair.

```
pose proof (aV st1) as (v, st2).
```

```
St, U, V: Type
aU: St -> U * St
f: U -> St -> V * St
st0: St
u: U
st1: St
aV: St -> V * St
v: V
st2: St
________________________________________
(V * St)%type
```

Finally we have $v$ and *st2* which are of exactly the type we require.

```
exact (v, st2).

Defined.
```

Now we define *ret* and *bind* defined normally, along with a proof of the monad laws for the Action monad. The goals of the proofs follow from destructuring terms that are tuples and using functional extensionality in a similar way to earlier proofs by changing the goal from the form $f = g$ to $\forall\ x,\ f\ x = g\ x$.

```
#[refine] Instance ActionMonad (St : Type) : Monad (Action St) := {
  ret := fun (U : Type) (u : U) => fun s => (u, s);
  bind (U V : Type) (aU : Action St U) (f : U -> Action St V) :=
      (fun (st0 : St) => let (u, st1) := aU st0 in
                          let aV := f u in
                            aV st1) (* which gives (v, st2) *)
}.
Proof.
- (* Left Identity *)
```

Here we can see the left-hand side of the goal is the expansion of *ret a >>= f* from the *left_identity_prf* defined earlier in the definition of the *Monad* typeclass. After simplification, destructuring of the tuple, and the use of functional extensionality, the goal is trivially solvable.

```
  intros.
```

```
St, T, U: Type
a: T
f: T -> Action St U
________________________________________
(fun st0 : St => let aV := f a in aV st0) = f a
```

```
  simpl. intros. apply functional_extensionality. intros.
  destruct (f a x). reflexivity.
- (* Right Identity *)
```

Similarly, here we can see that the left-hand side of the goal is the expansion of *m >>= ret* from definition of *right_identity_prf*. Again, the same kind of tactics solve the goal.

```
  intros.
```

```
St, T: Type
m: Action St T
________________________________________
(fun st0 : St =>
 let (u, st1) := m st0 in
 let aV := fun s : St => (u, s) in aV st1) = m
```

```
  simpl. unfold Action in m.
  apply functional_extensionality.
  intros.
  destruct (m x).
  reflexivity.
- (* Associativity *)
```

The goal for associativity begins to become difficult to read, but the same kind of tactics apply to solving this goal, which in nicer notation reads `aT >>= g >>= h = aT >>= (g >=> h)`. There comes a point with more complex goals where we need to trust the proof assistant software to do its part correctly. This frees the proof engineer to experiment a bit when proving more complex goals, while still holding the big picture of what the proof *should* be like in mind.

```
  do 3 intro. intros aT g h.
```

```
St, T, U, V: Type
aT: Action St T
g: T -> Action St U
h: U -> Action St V
________________________________________
(fun st0 : St =>
 let (u, st1) :=
   let (u, st1) := aT st0 in let aV := g u in aV st1 in
 let aV := h u in aV st1) =
(fun st0 : St =>
 let (u, st1) := aT st0 in
 let aV :=
   fun st2 : St =>
   let (u0, st3) := g u st2 in
   let aV := h u0 in aV st3 in
 aV st1)
```

```
  simpl. apply functional_extensionality. intro s.
  destruct (aT s) as [t st0].
  destruct (g t st0) as [u st1]. (* Note: aU := g t *)
  destruct (h u st1) as [v st2]. (* Note: aV := h u *)
  reflexivity.

Defined.
```

As a sanity check, we can see if our 'proof mode' definition of bind matches the one above.

```
Notation "a >>=' b" := (bind' a b) (at level 58, left associativity).
```

```
Lemma bindSanityCheck : forall (St U V : Type)
  (aU : Action St U) (f : U -> Action St V),
    bind' aU f = bind aU f.
Proof.
```

To show that the two definitions are indeed equivalent, we employ similar tactics. Essentially we are unfolding the definitions and destructuring the tuples as needed to the point where simplification makes it clear they are equal.

```
  intros.
```

```
St, U, V: Type
aU: Action St U
f: U -> Action St V
________________________________________
aU >>=' f = aU >>= f
```

```
  unfold ">>='". simpl. apply functional_extensionality.
  intros. destruct (aU x). destruct (f u s). reflexivity.
```

```
Qed.
```

### 3.2.4 Do Notation

We can also define notation similar to Haskell's "do notation". The notations used here are based upon the Coq ExtLib library[3].

Do notation makes it convenient when composing a number of functions which return monadic values. It is not precisely function composition as also the result of each function is made available to *all* the subsequent functions. The monadic values must share the same monad but the wrapped type can differ. It makes the 'unwrapped' version of each monadic value available to be used in the definition of all subsequent functions. This also serves the purpose of converting subsequent monadic values into functions of type $T \to M\ U$ to allow sequencing via *bind*.

In the context of the action monad the notation takes the result from each action (like a line of code) and makes the result available to be used in the definitions of all subsequent actions (lines of code).

There are examples of using the do notation below which demonstrate its usage and utility. Note that in Coq the standard notation (used here) does not actually use the keyword "do" but it is very similar to do notation.

```
Notation "a1 ;; a2" := (a1 >>= (fun _ => a2))
  (at level 61, right associativity).
Notation "x <- a1 ;; a2" := (a1 >>= (fun x => a2))
  (at level 61, a1 at next level, right associativity).
Notation "' pat <- a1 ;; a2" :=
  (a1 >>= (fun x => match x with pat => a2 end))
  (at level 61, pat pattern, a1 at next level, right associativity).
```

Returning to the *Maybe* monad briefly, we can see how the do notation works for *Maybe*. First, we define two functions of the form $T \to Maybe\ U$ which matches the second argument to *bind* of type $T \to M\ U$ where $M$ is a monad.

```
Definition minusOne (x : nat) : Maybe nat :=
  match x with
  | O ⇒ None
  | S x' ⇒ Some x'
end.

Definition add3 (t : nat × nat × nat) : Maybe nat :=
  match t with (a, b, c) ⇒
    Some (a + b + c)
  end.
```

To demonstrate that the type which *Maybe* wraps can change within do notation we also define a function *triple* which wraps three numbers into a tuple.

[3]Notation based upon https://github.com/coq-community/coq-ext-lib/blob/master/theories/Structures/Monad.v.

```
Definition triple (x y z: nat) : Maybe (nat × nat × nat) :=
  Some (x, y, z).
```

Now it is possible to calculate the sum of $n$ + ($n$ - 1) + ($n$ - 2) where $n$ is 4 as an example, as follows.

```
Eval compute in
  x <- Some 4 ;;
  y <- minusOne x;; (* y = 3 *)
  z <- minusOne y;; (* z = 2 *)
  t <- triple x y z ;; (* t = (4, 3, 2)*)
  add3 t.
```

```
= Some 9
: Maybe nat
```

The do notation above is syntactic sugar for:

```
Eval compute in
  Some 4 >>= (fun x =>
    minusOne x >>= (fun y =>
      minusOne y >>= (fun z =>
        triple x y z >>= (fun t =>
          add3 t)))).
```

```
= Some 9
: Maybe nat
```

Notice how all the named variables remain in scope for the rest of the definitions of the monadic values. For example, *triple* uses all the previous values.

Further examples of the usage of do notation will continue to appear, and when proving properties about smart contracts in DeepSEA do notation is a common sight.

### 3.2.5 State/Action Monad with Stack

Now that we have defined the State/Action monad and the convenient do notation, we demonstrate its usage when writing code which manipulates a Stack.

```
Definition Stack := list nat.
```

*pop* gives the top element of the stack as its result and removes the top item from the stack (the stack is the state in this case).

```
Definition pop : Stack → (nat × Stack) :=
  fun (s : Stack) ⇒
    match s with
        [] ⇒ (0, [])
      | (x :: xs) ⇒ (x, xs)
    end.
```

Here *pop* does not return a ((*Maybe nat*) × *Stack*) because this would make terms like *pop* >>= *push* ill-typed or force us to make the stack of type *list* (*Maybe nat*). Instead, it returns the value zero when popping an empty stack. We will see a more elegant handling of the situation later on with the *ActionT* monad.

To demonstrate that pop is indeed of type *Action Stack nat* we can see that the following definition passes type checking also.

```
Definition popTypeExample : Action Stack nat := pop.
```

We can also define *push* which returns the value *tt* of type *unit* which essentially means it returns nothing. However, it does add to the stack the element provided to it of type *nat*. Thus, its type is *nat* → *Action Stack unit*, which makes it a prime example of a function *f* discussed earlier.

```
Definition push (n : nat) : Action Stack unit :=
  fun (s : Stack) ⇒ (tt, n :: s).
Definition pushTypeExample : nat → Stack → (unit × Stack) := push.
```

Now we are ready to use *bind* to combine our '*aNat*' and '*f*' (which when provided with the *nat* output from *aNat* will produce '*aNat2*', the result of the bind).

```
Definition stackManipulation1 : Action Stack unit :=
  pop >>= push.
```

This is equivalent to the following in do notation:

```
Definition stackManipulation1' : Action Stack unit :=
  x ← pop ;; push x.
```

We can combine *push* and *pop* commands together using do notation (as well as combine them with previous stack manipulation functions).

```
Definition stackManipulation2 : Action Stack nat :=
  push 4;;
  stackManipulation1;;
  push 5;;
  pop.
```

```
Eval compute in stackManipulation2 [1;2;3].
```

```
= (5, [4; 1; 2; 3])
: nat * Stack
```

### 3.2.6 Monad-State Typeclass

To help have a unified syntax for dealing with operations about state when working with the action monad we can define a typeclass *MonadState* that requires instances of it to have a *get* and *put* method. There are no laws for these methods, but their intended usage in this context is that *put* will replace the state with its argument and *get* will leave the state unchanged and return the current state. Later we define *get* and *push* for the *Action* monad, the *ActionT* monad, and the *ActionT* monad specialised for the *Maybe* type.

```
Class MonadState (T : Type) (m : Type → Type) : Type := {
  get : m T;
  put : T → m unit
}.
```

Here we define *get* and *put* for the *Action* monad.

```
Instance MonadState_Action (St : Type) : MonadState St (Action St) := {
  get := (fun s ⇒ (s, s));
  put (newState : St) := (fun _ ⇒ (tt, newState))
}.
```

### 3.2.7 State/Action Transformer Monad

The action transformer monad is similar to the action monad, except that the result tuple is wrapped in a monad. In general, monad transformers transform monads into other monads. The purpose of defining the action transformer monad here is to transform the maybe monad into a monad similar to the action monad but with features of the maybe monad. One benefit of this is that this approach makes the handling of failure cases like calling *pop* on an empty list more elegant.

First we use Coq's `Variable` mechanism in order to assume there is a monad available to transform. *M* is the monad, *M_prf* is the evidence that *M* is indeed a monad. These will automatically be added as arguments to the functions which use them.

```
Variable M : Type → Type.
Variable M_prf : Monad M.
```

Now we define the action transformer monad, *ActionT*. In the wrapped version common in more standard presentations of the action/state transformer the function defined here is wrapped and called *runStateT* (or using our terminology, *runActionT*).

```
Definition ActionT `{Monad M} (St : Type) (A : Type) := St → M (A × St).
```

We can now show that for all monads *m* (which are implicitly added to the instance declaration by Coq) and for all types *St*, *ActionT St* is a monad. It is worth noting that the *ret* in the body of the definition of *ret* is not the same *ret*. The *ret* in the body is associated with the monad *m*, implicitly added to the context by Coq. Similarly for the *bind* hidden in the do notation in the body of the definition of *bind*.

```
#[refine] Instance Monad_ActionT (St : Type) : Monad (ActionT St) := {
  ret (U : Type) (a : U) (s : St) := ret (a, s);
  bind (U V : Type) (aU : ActionT St U) (f : U -> ActionT St V) :=
    (fun s =>
      ' (u, s') <- aU s ;;
      f u s'
    )
}.
Proof.

- (* Left identity *)
```

```
______________________________________
forall (T U : Type) (a : T) (f : T -> ActionT St U),
(fun s : St =>
 x <- ret (a, s);; (let (u, s') := x in f u s')) =
f a
```

The proof of left identity starts similarly to previous proofs but then relies upon the proof of left identity for the provided monad $m$. We could not prove this lemma without the knowledge that $m$ is a monad and has its own left identity proof.

```
  intros.
  apply functional_extensionality.
  intros.
```

```
______________________________________
x <- ret (a, x);; (let (u, s') := x in f u s') = f a x
```

Here we need to use a tool called setoid rewriting, which in this case allows us to rewrite expressions within lambda expressions. Lambda expressions are present here but hidden by do notation.

```
  setoid_rewrite left_identity_prf.
```

```
______________________________________
f a x = f a x
```

```
  reflexivity.
- (* Right identity *)
```

Right identity is proved similarly, with some rewriting required before applying the right identity proof of the provided monad $m$.

```
  intros T aT.
  apply functional_extensionality.
  intros.
```

```
______________________________________
x <- aT x;; (let (u, s') := x in ret (u, s')) = aT x
```

```
  assert (forall x0 : T * St, (let (a, s') := x0 in ret (a, s')) = ret x0).
  {
    intros.
    destruct x0.
    reflexivity.
  }
  setoid_rewrite H.
```

```
______________________________________
x <- aT x;; ret x = aT x
```

```
  apply right_identity_prf.
- (* Associativity *)
```

Associativity is also proved similarly by using the associativity proof of the supplied monad *m*, with rewriting needed to finish the proof.

```
  intros.
  apply functional_extensionality.
  intros.
```

```
________________________________________
x <- (x <- m x;; (let (u, s') := x in g u s'));;
(let (u, s') := x in h u s') =
x <- m x;;
(let (u, s') := x in
 x0 <- g u s';; (let (u0, s'0) := x0 in h u0 s'0))
```

```
  rewrite associativity_prf.
```

```
________________________________________
x <- m x;;
x0 <- (let (u, s') := x in g u s');;
(let (u, s') := x0 in h u s') =
x <- m x;;
(let (u, s') := x in
 x0 <- g u s';; (let (u0, s'0) := x0 in h u0 s'0))
```

```
  assert(forall x0 : T * St,
     x1 <- (let (a, s') := x0 in g a s');;
     (let (a, s') := (x1 : U * St) in h a s')
     =
     (let (a, s') := x0 in x1 <- g a s';;
     (let (a0, s'0) := (x1 : U * St) in h a0 s'0))).
    {
      intros.
      destruct x0.
      reflexivity.
    }
  setoid_rewrite H.
  reflexivity.

Defined.
```

Finally, we also define the helper functions *get* and *put* which are part of the *MonadState* typeclass which provide useful methods for getting the current state and replacing it by putting new state in.

```
Instance MonadState_ActionT (St : Type) : MonadState St (ActionT St) := {
  get := (fun s ⇒ ret (s, s));
  put (newState : St) := (fun _ ⇒ ret (tt, newState))
}.
```

### 3.2.8 Action Transformer Monad with Maybe and Stack

Having defined the action transformer, which takes any monad and type for state, we define *maybeActionT*. For all types (for state), *maybeActionT* is a monad, which we prove next.

```
Definition maybeActionT (St : Type) := ActionT Maybe St.
```

The proof that *maybeActionT St* is a monad is trivial because we have already proved it for all monads when proving that *ActionT m St* is a monad, for all *m* and *St*.

```
Instance Monad_MaybeActionT (St : Type) : Monad (maybeActionT St) :=
  Monad_ActionT Maybe _ St.
```

Similarly for the *get* and `set` methods, these can also be taken from the instance for *ActionT*.

```
Instance MonadState_MaybeActionT (St : Type) : MonadState St (maybeActionT St) :=
  MonadState_ActionT Maybe _ St.
```

We now define *pop'* and *push'*. The new function *pop'* can now return *None* while still enabling us to pop an item off the stack and put it back on, with the type of the stack simply being *list nat* rather than *list* (*Maybe nat*).

```
Definition pop' : Stack → Maybe (nat × Stack) :=
    fun (s : Stack) ⇒
      match s with
          [] ⇒ None
        | (x :: xs) ⇒ Some (x, xs)
      end.
```

As before we can see the type of *pop'* is also *ActionT Maybe Stack nat*. Similarly for *push'*.

```
Definition pop'TypeExample : ActionT Maybe Stack nat := pop'.

Definition push' (n : nat) : ActionT Maybe Stack unit :=
    fun (s : Stack) ⇒ Some (tt, n :: s).
Definition push'TypeExample : nat → Stack → Maybe (unit × Stack) := push'.
```

Now we can elegantly define plus for a stack. *plusStack* adds the top two elements and returns that value, resulting in a *None* instead if there are not at least two elements in the stack.

```
Definition plusStack : maybeActionT Stack nat :=
  a ← pop' ;;
  b ← pop' ;;
  ret (a + b).
```

Here we can see the result of this computation with the example list [1;5;3].

```
Eval compute in plusStack [1;5;3].
```

```
= Some (6, [3])
: Maybe (nat * Stack)
```

Consider computing the sum of the top three elements on the stack. We could do this by popping the top three elements and then adding them.

```
Definition plus3Stack : maybeActionT Stack nat :=
  a ← pop' ;;
  b ← pop' ;;
  c ← pop' ;;
  ret (a + b + c).
```

Alternatively, we could use the previously defined function *plusStack* which pops and adds the top two elements on the stack. First we add the top two elements, then push the result onto the stack and then by adding the top two elements we have now added the top three elements.

```
Definition plus3Stack' : maybeActionT Stack nat :=
  r ← plusStack ;;
  push' r ;;
  r2 ← plusStack ;;
  ret r2.
```

Since we are using a proof assistant such as Coq, we can prove that these two definitions are the same (under the assumption of functional extensionality). These two functions are not equal without the functional extensionality axiom because they are intensionally different.

```
Lemma plus3Equivalence : plus3Stack = plus3Stack'.
Proof.
```

```
  apply functional_extensionality.
```

```
________________________________________
forall x : Stack, plus3Stack x = plus3Stack' x
```

```
  intros.
```

First we unfold the definitions.

```
  unfold plus3Stack, plus3Stack', plusStack.
```

```
x: Stack
________________________________________
(a <- pop';; b <- pop';; c <- pop';; ret (a + b + c))
  x =
(r <- (a <- pop';; b <- pop';; ret (a + b));;
 _ <- push' r;;
 r2 <- (a <- pop';; b <- pop';; ret (a + b));; ret r2)
  x
```

Next we destructure the list three times to expose the top three elements of the stack. The stack may not have three elements so for the cases where there are 0, 1 or 2 elements in the stack we trivially solve a goal of *None* = *None*.

```
  destruct x.
  - simpl.
```

```
________________________________________
None = None
```

```
reflexivity.
  - destruct x.
  + simpl.
```

```
n: nat
────────────────────────
None = None
```

```
reflexivity.
  + destruct x.
  * simpl.
```

```
n, n0: nat
────────────────────────
None = None
```

```
reflexivity.
```

Finally, when there are indeed three elements (at least) in the stack we see that after simplification we have the expected trivially true goal shown below.

```
  * simpl.
```

```
n, n0, n1: nat
x: list nat
────────────────────────
Some (n + n0 + n1, x) = Some (n + n0 + n1, x)
```

```
reflexivity.

Qed.
```

### 3.2.9 Action Transformer Monad with Maybe and Contract Storage

Now we are set up to represent smart contracts in a similar way to the functional representation used in DeepSEA. We will still use the action transformer monad with the *Maybe* monad, but instead of using *Stack* we will use a contract storage type specialised for the contract at hand.

Here we will use the example of a simple tenancy smart contract, which has a single landlord that is authorised to set a new tenant or change the landlord. Theoretically a smart contract like this could be integrated with smart locks to eliminate the need for physical keys and key exchanges. For simplicity, here we will use natural numbers to model Ethereum addresses.

```
Definition ethereum_address := nat.
```

First, we define a record holding the storage variables of the smart contract.

```
Record tenancy_contract_storage_type := {
  Contract_landlord : ethereum_address;
  Contract_tenant : ethereum_address
}.
```

It is useful to define functions that update a specific field of the contract storage record and leave any remaining fields empty.

```
Definition update_Contract_tenant new_tenant r := {|
  Contract_landlord := Contract_landlord r;
  Contract_tenant := new_tenant
|}.

Definition update_Contract_landlord landlord r := {|
  Contract_landlord := landlord;
  Contract_tenant := Contract_tenant r
|}.
```

Next, we define what is called in DeepSEA the "machine environment". When a smart contract is called there is some context passed to it, along with the arguments of the function. Here we model the *caller* which corresponds to the Ethereum account which called the function. For the sake of the example, we also model the *callvalue* which is the amount of the blockchain currency, ether, which is given by the caller to the contract. We model the amount of ether as a natural number representing the amount of wei which is the smallest denomination of ether: 1 ether is $10^{18}$ wei.

```
Definition wei := nat.
Record machine_environment := {
  caller : ethereum_address;
  callvalue : wei
}.
```

Next we define the initial state of the contract storage to hold the initial tenant's ethereum address and have a specified landlord.

```
Definition init_tenancy_contract_storage : tenancy_contract_storage_type := {|
  Contract_landlord := 4632783269828394728397482347237428347892;
  Contract_tenant := 847221837284732847238972384723847238947 3
|}.
```

For easier reading, we define the type *DS* to abbreviate the action transformer monad using the maybe monad with the above contract storage type as the type of state.

```
Definition DS := ActionT Maybe tenancy_contract_storage_type.
```

It is also useful to define the *gets* function which applies a function to the entire state and returns the result.

```
Definition gets {A : Type} f : DS A :=
  a ← get ;;
  ret (f a).
```

Conversely, *puts* applies a function to the state and stores that value as the new state, returning the value *tt* of type unit.

```
Definition puts f : DS unit :=
  a ← get ;;
  put (f a).
```

`fail` is an additional useful function which, regardless of the state returns *None*. This is useful when writing code which includes a branch which should fail and result in the blockchain state reverting. We do not model the blockchain state reverting in this example, but the idea is that if the function returns *None* then the state of the contract would revert to its previous state.

```
Definition fail {A : Type} : DS A :=
  fun (s : tenancy_contract_storage_type) ⇒ None.
```

Now we are ready to define actual representations of specific smart contract functions. Here we define three functions:

- *setTenant*
- *getTenant*
- *changeLandlord*

These functions would normally be generated by the DeepSEA compiler automatically from a less verbose language designed for more easily writing smart contracts, but here they are written by hand in a similar style to what the DeepSEA compiler might generate.

First *setTenant* extracts the landlord field using *gets* and the automatically generated *Contract_landlord* function which extracts the landlord field from the record. Then it checks if the *caller* is indeed the *landlord* and if so uses *puts* to replace the storage variable *tenant* with *newTenant*.

```
Definition Contract_setTenant (newTenant : ethereum_address) (me : machine_environment) : DS unit :=
  landlord ← gets Contract_landlord;;
  if (landlord =? (caller me))
    then puts (update_Contract_tenant newTenant)
    else fail.
```

*getTenant* simply extracts the tenant from the contract storage and returns it.

```
Definition Contract_getTenant (me : machine_environment) : DS ethereum_address :=
  tenant ← gets Contract_tenant;;
  ret tenant.
```

Notice that the type here is *DS ethereum_address* indicating that this is an ‘action’ which when provided with a state (plus the additional argument of a machine environment) might return a tuple of state and *ethereum_address*. We can see this more clearly in the type example definition below.

```
Definition Contract_getTenantTypeExample :
  machine_environment → tenancy_contract_storage_type
  → Maybe (ethereum_address × tenancy_contract_storage_type)
  := Contract_getTenant.
```

Finally, we have the *changeLandlord* function, which allows the current landlord (and only the current landlord) to change the landlord. Its return type is unit and it returns *None* if the caller wasn't the landlord.

```
Definition Contract_changeLandlord (new_landlord : ethereum_address) (me : machine_environment)
 : DS unit :=
    landlord ← gets Contract_landlord;;
    if (landlord =? (caller me))
      then
        puts (update_Contract_landlord new_landlord)
      else fail.
```

### 3.2.10 Proofs About the Tenancy Contract

Most of what has been discussed so far could be done in a range of standard programming languages, but next we reap the benefits of using a proof assistant.

First, we prove that if the *changeLandlord* function returned successfully (by returning *Some*) then it must have been that the caller was the landlord.

It is helpful to keep in mind the structure of the contract function when doing the proof because the structure of a proof and the contract function itself naturally mirror each other to some degree.

```
Lemma Contract_changeLandlord_onlyLandlord :
  forall new_landlord me st,
     (exists r st', Contract_changeLandlord new_landlord me st = Some (r, st'))
  -> (caller me) = Contract_landlord st.
Proof.
```

We are required to prove that the caller from the machine environment is the landlord, given that the call succeeded with some result.

```
  intros.
```

First, we use `destruct` to expose the values called *r* and st' under the `exists`. This is a form of existential elimination. We also unfold *changeLandlord* to expose its definition.

```
  destruct H as [r]. destruct H as [st']. unfold Contract_changeLandlord in H.
```

```
H: (landlord <- gets Contract_landlord;;
   (if landlord =? caller me
   then
   puts (update_Contract_landlord new_landlord)
   else fail)) st = Some (r, st')
________________________________________
caller me = Contract_landlord st
```

Next we simplify, which in a sense 'runs' the first line of our *changeLandlord* smart contract function at a symbolic level.

```
  simpl in H.
```

```
H: (if Contract_landlord st =? caller me
   then puts (update_Contract_landlord new_landlord)
   else fail) st = Some (r, st')
_____________________________________________
caller me = Contract_landlord st
```

Now we do case analysis based upon the guard of the ‘if’ statement. Coq allows us to keep track of which case we are in via the use of *eqn*:*Case*.

```
  destruct (Contract_landlord st =? caller me) eqn:Case.
    +
```

```
Case: (Contract_landlord st =? caller me) = true
H: puts (update_Contract_landlord new_landlord) st
   =
   Some (r, st')
_____________________________________________
caller me = Contract_landlord st
```

Here we need to use a lemma from Coq’s *EqNat* library which demonstrates a correspondence between Boolean equality using =? and regular equality using =. This allows us to show, based up on the guard of the ‘if’ statement being true, that the goal is true.

```
    apply EqNat.beq_nat_true in Case.
```

```
Case: Contract_landlord st = caller me
H: puts (update_Contract_landlord new_landlord) st
   =
   Some (r, st')
_____________________________________________
caller me = Contract_landlord st
```

```
    auto.
    +
```

```
Case: (Contract_landlord st =? caller me) = false
H: fail st = Some (r, st')
_____________________________________________
caller me = Contract_landlord st
```

Here we are in the case of the guard being false, which after simplification and unfolding implies that the contract failed and returned *None*. However, we assumed that the contract succeeded and returned *Some*.

```
    unfold fail in H.
```

```
H: None = Some (r, st')
_____________________________________________
caller me = Contract_landlord st
```

The `discriminate` tactic used here allows us to prove anything based upon hypothesis *H* and that the constructors *Some* and *None* for the *Maybe* type are disjoint. This proves the final goal in this lemma.

```
    discriminate.

Qed.
```

We can also prove the converse, that if the caller is the landlord then the function call of *changeLandlord* will succeed.

```
Lemma Contract_changeLandlord_onlyLandlord_converse :
  forall new_landlord me st,
  (caller me) = Contract_landlord st
  -> (exists r st', Contract_changeLandlord new_landlord me st = Some (r, st'))
  .
Proof.
  intros.
  unfold Contract_changeLandlord.
  simpl. rewrite H.
```

```
________________________________________
exists
  (r : unit) (st' : tenancy_contract_storage_type),
  (if Contract_landlord st =? Contract_landlord st
   then puts (update_Contract_landlord new_landlord)
   else fail) st = Some (r, st')
```

Here we use the reflexivity of the Boolean equality operator to demonstrate that the guard of the 'if' statement is true.

```
  rewrite PeanoNat.Nat.eqb_refl.
```

```
________________________________________
exists
  (r : unit) (st' : tenancy_contract_storage_type),
  puts (update_Contract_landlord new_landlord) st =
  Some (r, st')
```

Then we provide the values which are now apparent for the resulting state and return value.

```
  exists tt. exists (update_Contract_landlord new_landlord st).
```

```
________________________________________
puts (update_Contract_landlord new_landlord) st =
Some (tt, update_Contract_landlord new_landlord st)
```

After some unfolding and simplification, the goal clearly follows.

```
  unfold puts. simpl.
```

```
————————————————————————————————————————
Some (tt, update_Contract_landlord new_landlord st) =
Some (tt, update_Contract_landlord new_landlord st)
```

```
  reflexivity.

Qed.
```

The identical proof applies for a similar lemma instead about *setTenant*.

```
Lemma Contract_setTenant_onlyLandlord_converse :
  forall new_tenant me st,
  (caller me) = Contract_landlord st
  -> (exists r st', Contract_setTenant new_tenant me st = Some (r, st'))
  .
Proof.
  intros.
  unfold Contract_setTenant.
  simpl. rewrite H.
```

```
————————————————————————————————————————
exists
  (r : unit) (st' : tenancy_contract_storage_type),
  (if Contract_landlord st =? Contract_landlord st
   then puts (update_Contract_tenant new_tenant)
   else fail) st = Some (r, st')
```

Similarly to the previous lemma, we know that the guard of the ‘if’ statement is true. We show this using the *Nat.eqb_refl* lemma of the *PeanoNat* library.

```
  rewrite PeanoNat.Nat.eqb_refl.
```

```
————————————————————————————————————————
exists
  (r : unit) (st' : tenancy_contract_storage_type),
  puts (update_Contract_tenant new_tenant) st =
  Some (r, st')
```

We now know the value returned by the function must be unit (*tt*) and that the resulting contract storage must be the original state except with the tenant replaced with *new_tenant*. So we supply these values as witnesses.

```
  exists tt. exists (update_Contract_tenant new_tenant st).
```

```
————————————————————————————————————————
puts (update_Contract_tenant new_tenant) st =
Some (tt, update_Contract_tenant new_tenant st)
```

Now we perform some unfolding and simplification. The goal is then clearly true and we can apply the `reflexivity` tactic to solve the goal.

```
  unfold puts. simpl.
```

```
________________________________________________________________
Some (tt, update_Contract_tenant new_tenant st) =
Some (tt, update_Contract_tenant new_tenant st)
```

```
  reflexivity.
Qed.
```

Next we prove a different lemma stating that if the caller is the landlord, then a call to *changeTenant* will succeed and the resulting contract storage will hold the new tenant.

```
Lemma Contract_changeTenant_landlordSet :
  forall new_tenant me st,
    (caller me) = Contract_landlord st ->
    match Contract_setTenant new_tenant me st with
    | Some (r, st') => Contract_tenant st' = new_tenant
    | None => False
    end.
Proof.
  intros.
```

First, we bring into the context the previously proved lemma which allows us to know that the *Some* branch of the match expression must hold.

```
  pose proof (Contract_setTenant_onlyLandlord_converse new_tenant me st H).
  destruct H0 as [r]. destruct H0 as [st'].
  rewrite H0.
```

```
H0: Contract_setTenant new_tenant me st
    =
    Some (r, st')
________________________________________________________________
Contract_tenant st' = new_tenant
```

Having established that the *Some* branch must hold, we are now required to prove that the contract storage afterwards holds the new tenant. This requires similar reasoning to before when unfolding the smart contract definition and exploring each branch of its possible execution.

```
  unfold Contract_setTenant in H0.
  simpl in H0.
```

As previously, we consider each of the cases of the 'if' statement guard being true or false separately.

```
  destruct (Contract_landlord st =? caller me) eqn:Case.
    - unfold puts in H0. simpl in H0.
```

The use of the inversion tactic allows us to reason from the injectivity of the *Some* constructor to establish, in particular, that *st'* equals the original state with the *new_tenant* having overwritten the tenant field.

```
    inversion H0.
```

```
H2: tt = r
H3: update_Contract_tenant new_tenant st = st'
-----------------------------------------------
Contract_tenant (update_Contract_tenant new_tenant st) =
new_tenant
```

```
    simpl.
```

```
-----------------------------------------------
new_tenant = new_tenant
```

```
reflexivity.
    -
```

```
H0: fail st = Some (r, st')
-----------------------------------------------
Contract_tenant st' = new_tenant
```

Finally, as done previously we can dismiss the case that the guard was false because that results in the contract failing, when we have already shown it succeeds.

```
    unfold fail in H0.
```

```
H0: None = Some (r, st')
-----------------------------------------------
Contract_tenant st' = new_tenant
```

```
discriminate.

Qed.
```

### 3.2.11 Remarks

This section introduced monads as a technique for modelling the execution of an example tenancy ‘smart contract’ that bears some similarities to how real smart contracts are modelled in DeepSEA. The use of the proof assistant Coq allows us to prove some properties about the tenancy smart contract, again in a similar style to proofs about real DeepSEA smart contracts. The monadic representation of smart contracts is the highest level of abstraction of the smart contract generated by DeepSEA from what is written by the programmer. The technique of refinement is used to link this level of abstraction down to executable Ethereum Virtual Machine bytecode, which we discuss next.

## 3.3 Refinement and Verified Compilation

This section explores the use of stepwise refinement to build a verified compiler for simple arithmetic expressions. The technique demonstrated here gives an insight into how stepwise refinement is used in

DeepSEA to create a verified compiler down to EVM bytecode, with the ability to prove high-level properties that can be known to hold for the bytecode. This example is intended to be closer to the approach taken in DeepSEA, rather than rendering in Coq standard refinement examples such as those found in textbooks on the subject [136]–[138].

First, we will informally define the key concepts relevant to this section.

**Definition 3.3.1** (Simple Arithmetic Expression)**.** Here, a simple arithmetic expression will be an expression such as "`let` $y = 2x$ `in` $y + 4$". The numerals represent natural numbers and may include let expressions, multiplication, addition and subtraction (with subtraction having the result bounded below by zero). The value for the variable $x$ will be considered an input and assumed to be provided when the expression is evaluated. Evaluating the expression is expected to give a natural number.

These simple arithmetic expressions have the additional restriction that any "`let` $x =$" command masks the named variable $x$ from let expressions which surround it. So, for example, "`let` $y = 2x$ `in` $y + 4$" is expressible but "`let` $y = 2x$ `in` $y + x$" is not, because in the expression $y + x$ the value for $x$ has been masked by the let expression "(`let` $y = 2x$ `in` ...)". At an implementation level, there will be one stack holding the values for all the let bound variables, which is the rationale for this limitation.

**Definition 3.3.2** (Layer)**.** Each layer is a level of abstraction representing and implementing the operations of a simple arithmetic expression, which for our purposes move from being easier to reason about in a proof assistant down to being easier to implement on hardware.

This stepwise refinement consists of 4 layers:

- Layer 1: A functional Coq definition.
- Layer 2: An expression tree using a unary representation of natural numbers.
- Layer 3: An expression tree using a binary representation of natural numbers.
- Layer 4: A kind of stack machine using a binary representation of natural numbers.

The idea is that a stack machine is closer to machine-level than an expression tree, and similarly for a binary representation of natural numbers. The goal is to be able to produce 'code' for a stack machine that provably behaves identically to the high-level functional Coq definition which we are able to easily reason about in Coq.

At each layer we have the following two key definitions:

**Definition 3.3.3** (Operation)**.** $L*\_op$ describes the operations of the layer *, which are:

- Layer 1: The purpose of layer 1 is to facilitate reasoning in Coq. As a result, all the operations are combined to give a functional Coq definition of type *nat* → *nat*. However, one can see the individual operations intended to be the layer 1 operations by partially unfolding the Coq definition.

- Layer 2: An expression tree handling natural numbers represented as *nat* (unary). Here the individual operations are indicated by the constructors of type *L2_op*.
- Layer 3: An expression tree handling natural numbers represented as *N* (binary). Again, the operations are indicated by the constructors, here of type *L3_op*.
- Layer 4: *L4_op* is a stack of commands, with the stack represented as a linked list. Similarly to layer 3, these operations handle natural numbers represented as *N* (binary).

The second key definition for each layer is the semantics of the layer. The semantics describe the meaning of the operations of each layer by providing a method to generate a natural number from the layer operations and a given value for $x$.

**Definition 3.3.4** (Semantics)**.** *eval_L*_op* gives the semantics of the layer *, these are:

- Layer 1: The operations are already a function from *nat* to *nat* and so the semantics for this layer is simply applying the function to a given value, $x$.
- Layer 2: The expression tree is evaluated, using the definitions for +,×,- defined over Coq's *nat* type (unary).
- Layer 3: Again, the expression tree is evaluated, this time using the definitions for +,×,- defined over Coq's *N* type (binary).
- Layer 4: The operations stack is evaluated, using a stack of natural numbers represented as *N* (binary) and a stack for values of the variables in let expressions. The same definitions as Layer 3 are used for +,×,- with Coq's *N* type (binary).

There are also relations defined between the representations of natural numbers used in adjacent layers. The relations define when the value of $x$ provided as input in the adjacent layers represent the same natural number, and similarly for the value returned by evaluating an expression.

We will use the following arithmetic expression as the example throughout this section:

`let` $y = 2x$ `in` $y + 4$

This arithmetic expression is written in a way which informally corresponds to the layer 1 representation when the Coq function is partially unfolded. Consider the language that arithmetic expressions like the one above can be written in. This language serves a similar function to the DeepSEA programming language, which smart contracts are written in before being transpiled into Coq. When transpiled into Coq, they are transpiled into a form analogous to our layer 2 operations, which are then translated into a form analogous to our layer 1 operations and this process forms part of the refinement proof. As a result of this, the informal form above and the representation of a smart contract in the DeepSEA surface-level programming language (in '`.ds`' files) are not themselves a layer in the stepwise refinement proof. Rather, through an unverified method, they are transformed into a 'layer 2' representation which is part of a layer in the stepwise refinement proof.

In the DeepSEA system, there is an automatic translation from '`.ds`' files into an intermediate representation which corresponds to layer 2 of this simple arithmetic expression example. Here, we manually translate expressions such as "`let` $y = 2x$ `in` $y + 4$" into their representation in layer 2.

The goal is to prove properties about the layer 1 representation, which we generate from the layer 2 representation. For both DeepSEA and for simple arithmetic expressions, the layer 1 representations resemble the informal representation enough for a programmer to see the informal correspondence. Seeing this correspondence is not crucial as the refinement proof guarantees that the properties proved about the layer 1 representation apply to the bottom layer, which is the layer intended to be used for actually executing the program.

First, we will introduce the definition of the expression tree of layer 2, *L2_op*. Normally the operations for a lower layer are synthesised from a higher layer as the representation becomes more concrete. However, for layer 1 we make an exception. Layer 1 represents a functional Coq definition of type $nat \to nat$ which we could not use to synthesise layer 2 operations. Still, the same sort of refinement proof will be completed between layer 1 and layer 2 as with the other layers.

### 3.3.1 Layer 2: Expression Trees with Unary Natural Numbers

First we present the representation of a simple arithmetic expression in layer 2, which are the operations of layer 2.

```
Inductive L2_op :=
  | L2Const (n : nat)
  | L2ReadX
  | L2WriteX (ops1 ops2 : L2_op)
  | L2Add (ops1 ops2 : L2_op)
  | L2Sub (ops1 ops2 : L2_op)
  | L2Mul (ops1 ops2 : L2_op).
```

Expression trees of type *L2_op* can either be:

- a constant natural number;
- a read of a single storage variable $x$;
- a let expression meaning `let` $x :=$ *ops1* `in` *ops2* (where $x$ masks previous values for $x$);
- or an expression with one of the binary operators: $+$, -, $\times$.

Our example of `let` $y = 2x$ `in` $y + 4$ translated manually into the expression tree of type *L2_op* is:

```
Definition example_L2 := (L2WriteX
                           (L2Mul
                             (L2Const 2)
                             L2ReadX )
                           (L2Add
                             L2ReadX
                             (L2Const 4))).
```

Now we define the semantics of expression trees such as the one above. The semantics function *eval_L2_op* takes an expression tree of type *L2_op* and an initial value for the variable *x*. The notation scope for +, -, × is such that their definition for the natural numbers (*nat*) are used here, which means that subtraction is bounded below by zero.

```
Open Scope nat.

Fixpoint eval_L2_op (ops : L2_op) (x : nat) : nat :=
  match ops with
    | L2Const n ⇒ n
    | L2ReadX ⇒ x
    | L2WriteX ops1 ops2 ⇒ (eval_L2_op ops2 (eval_L2_op ops1 x))
    | L2Add ops1 ops2 ⇒ (eval_L2_op ops1 x) + (eval_L2_op ops2 x)
    | L2Sub ops1 ops2 ⇒ (eval_L2_op ops1 x) - (eval_L2_op ops2 x)
    | L2Mul ops1 ops2 ⇒ (eval_L2_op ops1 x) × (eval_L2_op ops2 x)
  end.
Close Scope nat.
```

We can now evaluate our *example_L2* (`let` $y = 2x$ `in` $y + 4$) with respect to the semantics *eval_L2_op* for specific values of *x*, as shown here.

```
Eval compute in (eval_L2_op example_L2 1).
```

```
= 6
: nat
```

```
Eval compute in (eval_L2_op example_L2 3).
```

```
= 10
: nat
```

## 3.3.2 Layer 1: Coq Functions with Unary Natural Numbers

Now we present the operations of layer 1 and their semantics. Layer 1 operations are simply a function from natural numbers to natural numbers. The semantics, *eval_L1_op*, simply applies the function to *x*. This is somewhat unnecessary, however this is done for the sake of consistency between layers so that there is an explicit definition for the semantics for each layer.

```
Definition L1_op := nat → nat.
Definition eval_L1_op (f : L1_op) (x : nat) := f x.
```

Next, we define a function to synthesise layer 1 operations from layer 2 operations. "`fun` *x* ⇒" is a lambda. Similarly to layer 2, the notation scope for *nat* is used here. The branch for *L2WriteX ops1 ops2* corresponds informally to `let` $x$ := *ops1* `in` *ops2*. We can see that in this branch only the value of *xNew* will be available in the evaluation of *ops2*.

```
Open Scope nat.

Fixpoint synth_L1_op (ops : L2_op) : L1_op :=
  match ops with
    | L2Const n ⇒ fun x ⇒ n
    | L2ReadX ⇒ fun x ⇒ x
    | L2WriteX ops1 ops2 ⇒ fun x ⇒ let xNew := (eval_L1_op (synth_L1_op ops1) x) in (eval_L1_op
(synth_L1_op ops2) xNew)
    | L2Add ops1 ops2 ⇒ fun x ⇒ (eval_L1_op (synth_L1_op ops1) x) + (eval_L1_op (synth_L1_op ops2)
x)
    | L2Sub ops1 ops2 ⇒ fun x ⇒ (eval_L1_op (synth_L1_op ops1) x) - (eval_L1_op (synth_L1_op ops2) x)
    | L2Mul ops1 ops2 ⇒ fun x ⇒ (eval_L1_op (synth_L1_op ops1) x) × (eval_L1_op (synth_L1_op ops2) x)
  end.
Close Scope nat.
```

By converting *example_L2* to a layer 1 operation using *synth_L1_op* and partially unfolding the layer 1 operation we can see the representation of our original simple arithmetic expression most clearly.

```
synth_L1_op example_L2 =
(fun x : nat => let xNew := 2 * x in xNew + 4)
```

### 3.3.3 Layer 1-2 Refinement: Coq Functions to Expression Trees

Now that layer 1 and layer 2 have been defined, we can state and prove that layer 1 is refined by layer 2. First, for the sake of consistency with the layer 2-3 and 3-4 refinement proofs, we explicitly specify the notion of whether the input states ($x$) for these layers are equal, and similarly for the return values. For layer 1 and 2, both $x$ and the return value are represented as a natural number, so the definitions are trivial.

```
Definition input_relation_L1L2 (L1x L2x : nat) : Prop := L1x = L2x.
Definition result_relation_L1L2 (L1x L2x : nat) : Prop := L1x = L2x.
```

The refinement proof between layer 1 and layer 2 states: for all layer 2 operations, if the initial values for $x$ are the same, then evaluating the layer 2 operations via the layer 2 semantics will match the evaluation of the layer 1 semantics on the layer 1 operations corresponding to the layer 2 operations (according to the function *synth_L1_op*).

```
Lemma refinement_proof_L1L2 :
  forall (ops : L2_op) L1x L2x (ValidInit : input_relation_L1L2 L1x L2x),
    result_relation_L1L2 (eval_L1_op (synth_L1_op ops) L1x) (eval_L2_op ops L2x).
Proof.
```

The proof begins by induction over the expression tree type. The cases for *L2Const* and *L2ReadX* are trivially true after unfolding the definitions and are omitted below, with the *L2ReadX* case making use of the *ValidInit* assumption.

```
  induction ops.
```

For the *L2WriteX* case, after simplification (partially omitted), we have the following proof state. The inductive hypotheses are used to prove the goal.

```
  - abbreviated.
```

```
ops1, ops2: L2_op
IHops1: forall L1x L2x : nat,
        L1x = L2x ->
        result_relation_L1L2
        (eval_L1_op (synth_L1_op ops1) L1x)
        (eval_L2_op ops1 L2x)
IHops2: forall L1x L2x : nat,
        L1x = L2x ->
        result_relation_L1L2
        (eval_L1_op (synth_L1_op ops2) L1x)
        (eval_L2_op ops2 L2x)
L1x, L2x: nat
ValidInit: L1x = L2x
________________________________________
synth_L1_op ops2 (synth_L1_op ops1 L1x) =
eval_L2_op ops2 (eval_L2_op ops1 L2x)
```

```
    apply IHops2; apply IHops1; assumption.
```

For the *L2Add* case, the goal follows from rewriting using the inductive hypotheses.

```
  -     abbreviated.
```

```
ops1, ops2: L2_op
IHops1: forall L1x L2x : nat,
        L1x = L2x ->
        synth_L1_op ops1 L1x = eval_L2_op ops1 L2x
IHops2: forall L1x L2x : nat,
        L1x = L2x ->
        synth_L1_op ops2 L1x = eval_L2_op ops2 L2x
L2x: nat
________________________________________
(synth_L1_op ops1 L2x + synth_L1_op ops2 L2x)%nat =
(eval_L2_op ops1 L2x + eval_L2_op ops2 L2x)%nat
```

Here, the '+' symbol is syntactic sugar for *Nat.add* as emphasised by the notation scope indicated by the expression being surrounded by brackets followed by *%nat*.

```
    rewrite <- (IHops1 L2x) by reflexivity;
    rewrite <- (IHops2 L2x) by reflexivity; simpl.
```

```
________________________________________
(synth_L1_op ops1 L2x + synth_L1_op ops2 L2x)%nat =
(synth_L1_op ops1 L2x + synth_L1_op ops2 L2x)%nat
```

```
    reflexivity.
```

The *L2Sub* and *L2Mul* cases follow similarly and are omitted here.

`Qed.`

### 3.3.4 Layer 3: Expression Trees with Binary Natural Numbers

This refinement step replaces the unary definition of natural numbers with the binary representation. Here we can see the difference between the unary *nat* type and the binary *N* type which uses the *positive* type to represent positive numbers in binary form.

```
Print nat.
```

```
Inductive nat : Set :=  O : nat | S : nat -> nat.

Arguments S _%nat_scope
```

```
Print N.
```

```
Inductive N : Set :=  N0 : N | Npos : positive -> N.

Arguments Npos _%positive_scope
```

```
Print positive.
```

```
Inductive positive : Set :=
    xI : positive -> positive
  | xO : positive -> positive
  | xH : positive.

Arguments xI _%positive_scope
Arguments xO _%positive_scope
```

The definition for layer 3 operations is identical to layer 2 operations, except that *L3Const* holds a natural number of type *N* instead of *nat*.

```
Inductive L3_op :=
  | L3Const (n : N)
  | L3ReadX
  | L3WriteX (ops1 ops2 : L3_op)
  | L3Add (ops1 ops2 : L3_op)
  | L3Sub (ops1 ops2 : L3_op)
  | L3Mul (ops1 ops2 : L3_op).
```

Synthesising a layer 3 operation simply replaces *L2* with *L3* for each constructor, except for additionally converting the constant *n* to its binary representation.

```
Fixpoint synth_L3_op (ops : L2_op) : L3_op :=
  match ops with
  | L2Const n ⇒ L3Const (N.of_nat n)
  | L2ReadX ⇒ L3ReadX
  | L2WriteX ops1 ops2 ⇒ L3WriteX (synth_L3_op ops1) (synth_L3_op ops2)
  | L2Add ops1 ops2 ⇒ L3Add (synth_L3_op ops1) (synth_L3_op ops2)
  | L2Sub ops1 ops2 ⇒ L3Sub (synth_L3_op ops1) (synth_L3_op ops2)
  | L2Mul ops1 ops2 ⇒ L3Mul (synth_L3_op ops1) (synth_L3_op ops2)
  end.
```

We can generate the layer 3 representation of our example by applying *synth_L3_op* to *example_L2*, which gives the following layer 3 expression tree.

```
Eval compute in (synth_L3_op example_L2).
```

```
= L3WriteX (L3Mul (L3Const 2) L3ReadX)
    (L3Add L3ReadX (L3Const 4))
: L3_op
```

Again we define the semantics of expression trees such as the example above that now is synthesised down to layer 3. The semantics defined by *eval_L3_op* is identical to *eval_L2_op* except that the notation scope is $N$ so the operations +, × and - are those for the binary representation of natural numbers.

```
Open Scope N.

Fixpoint eval_L3_op (ops : L3_op) (x : N) : N :=
  match ops with
    | L3Const n ⇒ n
    | L3ReadX ⇒ x
    | L3WriteX ops1 ops2 ⇒ (eval_L3_op ops2 (eval_L3_op ops1 x))
    | L3Add ops1 ops2 ⇒ (eval_L3_op ops1 x) + (eval_L3_op ops2 x)
    | L3Sub ops1 ops2 ⇒ (eval_L3_op ops1 x) - (eval_L3_op ops2 x)
    | L3Mul ops1 ops2 ⇒ (eval_L3_op ops1 x) × (eval_L3_op ops2 x)
  end.
Close Scope N.
```

### 3.3.5 Layer 2-3 Refinement: Unary Natural Numbers to their Binary Representation

Similarly to the layer 1-2 refinement proof, we first define equality relations between the inputs (the initial value for $x$) and also for the results. Here, it is not the trivial equality, but equality after the layer 2 value is converted to its binary form that we use. We could have instead converted the layer 3 value to a *nat*, either conversion could work.

```
Definition input_relation_L2L3 L2x L3x : Prop := N.of_nat L2x = L3x.
Definition result_relation_L2L3 L2x L3x : Prop := N.of_nat L2x = L3x.
```

Now we show that layer 3 refines layer 2. Similarly to the previous refinement proof, we assume that the inputs are related with respect to the input relation. We aim to prove that the results of evaluating *ops* and (*synth_L3_op ops*), with respect to their semantics, give results which are related according to the result relation.

```
Lemma refinement_proof_L2L3 :
  forall (ops : L2_op) L2x L3x (ValidInit : input_relation_L2L3 L2x L3x),
    result_relation_L2L3 (eval_L2_op ops L2x) (eval_L3_op (synth_L3_op ops) L3x).
  Proof.
```

The proof, again, proceeds by induction on the layer 2 operations. The case for *L2Const* (omitted) and *L2ReadX* follow trivially, with the use of the *ValidInit* assumption to establish that the result relation holds.

```
    induction ops.
    - abbreviated.
```

```
L2x: nat
L3x: N
ValidInit: N.of_nat L2x = L3x
________________________________________
N.of_nat L2x = L3x
```

```
    assumption.
```

For the *L2WriteX* case, the goal again follows from the inductive hypotheses.

```
    - abbreviated.
```

```
ops1, ops2: L2_op
IHops1: forall (L2x : nat) (L3x : N),
       input_relation_L2L3 L2x L3x ->
       result_relation_L2L3 (eval_L2_op ops1
       L2x)
       (eval_L3_op (synth_L3_op ops1) L3x)
IHops2: forall (L2x : nat) (L3x : N),
       input_relation_L2L3 L2x L3x ->
       result_relation_L2L3 (eval_L2_op ops2
       L2x)
       (eval_L3_op (synth_L3_op ops2) L3x)
L2x: nat
L3x: N
ValidInit: input_relation_L2L3 L2x L3x
________________________________________
result_relation_L2L3
  (eval_L2_op ops2 (eval_L2_op ops1 L2x))
  (eval_L3_op (synth_L3_op ops2)
     (eval_L3_op (synth_L3_op ops1) L3x))
```

```
      pose proof (IHops1 _ _ ValidInit);
      pose proof (IHops2 _ _ H);
      assumption.
```

For the +, - and $\times$ operators, we need the additional knowledge that converting $n + m$ to its binary form equals first converting each of $n$ and $m$ and then computing *N.add*. This is the *Nat2N.inj_add* lemma, which is already proved in the Coq standard library.

```
      - abbreviated.
```

```
ops1, ops2: L2_op
IHops1: forall (L2x : nat) (L3x : N),
       input_relation_L2L3 L2x L3x ->
       result_relation_L2L3 (eval_L2_op ops1
       L2x)
       (eval_L3_op (synth_L3_op ops1) L3x)
IHops2: forall (L2x : nat) (L3x : N),
       input_relation_L2L3 L2x L3x ->
       result_relation_L2L3 (eval_L2_op ops2
       L2x)
       (eval_L3_op (synth_L3_op ops2) L3x)
L2x: nat
L3x: N
ValidInit: input_relation_L2L3 L2x L3x
________________________________________________________________
N.of_nat (eval_L2_op ops1 L2x + eval_L2_op ops2 L2x) =
(eval_L3_op (synth_L3_op ops1) L3x +
 eval_L3_op (synth_L3_op ops2) L3x)%N
```

```
      rewrite Nat2N.inj_add.
```

```
ops1, ops2: L2_op
IHops1: forall (L2x : nat) (L3x : N),
       input_relation_L2L3 L2x L3x ->
       result_relation_L2L3 (eval_L2_op ops1
       L2x)
       (eval_L3_op (synth_L3_op ops1) L3x)
IHops2: forall (L2x : nat) (L3x : N),
       input_relation_L2L3 L2x L3x ->
       result_relation_L2L3 (eval_L2_op ops2
       L2x)
       (eval_L3_op (synth_L3_op ops2) L3x)
L2x: nat
L3x: N
ValidInit: input_relation_L2L3 L2x L3x
________________________________________________________________
(N.of_nat (eval_L2_op ops1 L2x) +
 N.of_nat (eval_L2_op ops2 L2x))%N =
(eval_L3_op (synth_L3_op ops1) L3x +
 eval_L3_op (synth_L3_op ops2) L3x)%N
```

```
      rewrite <- (IHops1 L2x) by assumption;
      rewrite <- (IHops2 L2x) by assumption;
```

```
        reflexivity.
```

Similarly, the lemmas *Nat2N.inj_sub* and *Nat2N.inj_mul* are required for the remaining two branches which are omitted for brevity.

```
Qed.
```

### 3.3.6 Layer 4: Stack Machine with Binary Natural Numbers

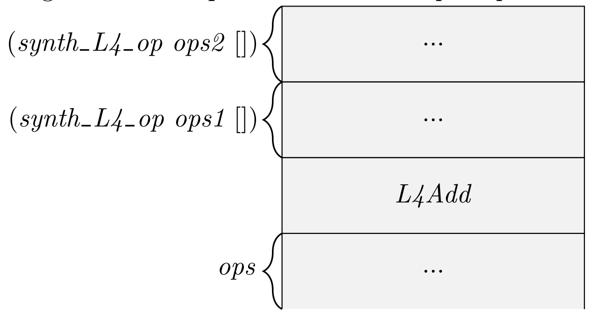
Layer 4 operations are a list of layer 4 commands. The stack machine here will take a stack (list) of commands as well as a stack for holding the variable $x$ (and the masked values of $x$) and will work on a stack holding binary natural numbers, ideally returning a stack with one element.

```
Inductive L4_cmd :=
  | L4Const (l : N)
  | L4ReadX
  | L4WriteXStart
  | L4WriteXEnd
  | L4Add
  | L4Sub
  | L4Mul.

Definition L4_op := list L4_cmd.
Definition stack := list N.
```

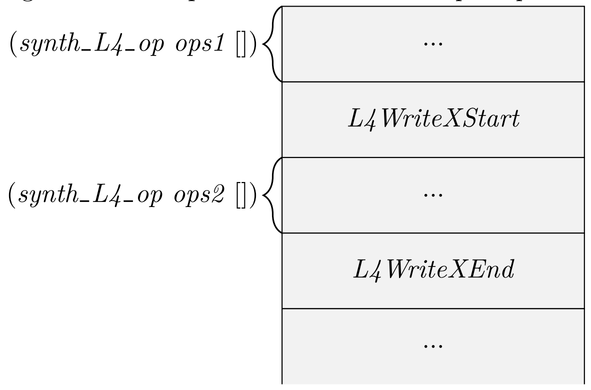
To synthesise the operations stack from the expression tree we take a fairly standard approach. As shown in Figure 3.2, for *L3Add* we first synthesise *ops2*, then synthesise *ops1*, then *L4Add* is placed on the stack beneath the synthesised commands. *L4Sub* and *L4Mul* are synthesised to layer 4 similarly.

Figure 3.2: Compilation of *L3Add ops1 ops2*



For keeping track of the current value of $x$ we have markers for *L4WriteXStart* (which represents the start of *ops2*) and *L4WriteXEnd* (which represents the end of *ops2*), in an expression of the form "... `let` $x$ := *ops1* `in` *ops2* ...". Figure 3.3 shows this compilation step.

Figure 3.3: Compilation of *L3WriteX ops1 ops2*



```
Fixpoint synth_L4_op (n : L3_op) (ops : L4_op) : L4_op :=
  match n with
  | L3Const l ⇒ (L4Const l) :: ops
  | L3ReadX ⇒ L4ReadX :: ops
  | L3WriteX ops1 ops2 ⇒ (synth_L4_op ops1 []) ++ [L4WriteXStart] ++ (synth_L4_op ops2 []) ++
[L4WriteXEnd] ++ ops
  | L3Add ops1 ops2 ⇒ (synth_L4_op ops2 []) ++ (synth_L4_op ops1 []) ++ [L4Add] ++ ops
  | L3Sub ops1 ops2 ⇒ (synth_L4_op ops2 []) ++ (synth_L4_op ops1 []) ++ [L4Sub] ++ ops
  | L3Mul ops1 ops2 ⇒ (synth_L4_op ops2 []) ++ (synth_L4_op ops1 []) ++ [L4Mul] ++ ops
  end.
```

The evaluation of the layer 4 operations is standard for a stack machine, with special handling of the *L4WriteXStart* and *L4WriteXEnd* commands. For *L4WriteXStart* we take the top value from the main stack and evaluate *ops2* with that top value pushed onto the *xVals* stack. The *xVals* stack keeps track of the current $x$ value and holds all the masked $x$ values. *WriteXEnd* causes the top value from the *xVals* stack to be dropped, with the evaluation continuing as before after that change. The scope is for binary natural numbers $N$ and so the +, - and × operators handle binary natural numbers.

```
Open Scope N.

Fixpoint eval_L4_op (ops : L4_op) (s : stack) (xVals : stack) : option stack :=
  match ops with
    | h :: tl ⇒
    match h with
      | L4Const l ⇒ eval_L4_op tl ((l :: s)) xVals
      | L4ReadX ⇒
        match xVals with
          | x :: xs ⇒ eval_L4_op tl ((x :: s)) xVals
          | _ ⇒ None
        end
      | L4WriteXStart ⇒
        match s with
        | x' :: tl2 ⇒ eval_L4_op tl tl2 (x' :: xVals)
        | [] ⇒ None
        end
      | L4WriteXEnd ⇒
        match xVals with
        | x' :: tl2 ⇒ eval_L4_op tl s tl2
        | [] ⇒ None
        end
      | L4Add ⇒
        match s with
        | h1 :: h2 :: tl2 ⇒ eval_L4_op tl (((h1 + h2) :: tl2)) xVals
        | _ ⇒ None
        end
      | L4Sub ⇒
        match s with
        | h1 :: h2 :: tl2 ⇒ eval_L4_op tl (((h1 - h2) :: tl2)) xVals
        | _ ⇒ None
        end
      | L4Mul ⇒
        match s with
        | h1 :: h2 :: tl2 ⇒ eval_L4_op tl (((h1 × h2) :: tl2)) xVals
        | _ ⇒ None
        end
    end
    | [] ⇒ Some s
  end.
Close Scope N.
```

### 3.3.7 Layer 3-4 Refinement: Expression Trees to Stack Machine

As with earlier layers, we define relations for inputs and for results. Since *eval_L4_op* can fail, this is represented by it returning an optional type, so the result relation reflects this. We also consider a layer 4 value equal to a layer 3 value if the layer 4 value is a stack holding exactly one element.

```
Definition input_relation_L3L4 (L3x : N) (L4x : stack) : Prop := [L3x] = L4x.
Definition result_relation_L3L4 (L3x : N) (L4x : option stack) : Prop := Some [L3x] = L4x.
```

To prove this refinement step, we first begin with the helper lemma shown below. The goal is to show that for any *L3_op*, evaluating additional operations with the result of evaluation the layer 3 operations already on the stack, gives the same result as synthesising the layer 3 operations into layer 4 operations and executing those before the additional operations. This approach is inspired by the stack machine section in Chlipala's Certified Programming with Dependent Types [139].

```
Lemma refinement_proof_helper : forall (ops : L3_op) additionalOps s x xs,
    eval_L4_op additionalOps ((eval_L3_op ops x) :: s) (x::xs)
  = eval_L4_op (synth_L4_op ops [] ++ additionalOps) s (x::xs).
Proof.
```

The proof begins by induction on the *L3_op*. The first two cases for *L3Const* and *L3ReadX* are trivial (omitted).

```
  induction ops.
```

The case for *L3WriteX* follows from the use of the inductive hypotheses, after some simplification and rewriting.

```
  - intros.
 abbreviated.
```

```
ops1, ops2: L3_op
IHops1: forall (additionalOps : L4_op)
       (s : list N) (x : N) (xs : list N),
       eval_L4_op additionalOps
       (eval_L3_op ops1 x :: s)
       (x :: xs) =
       eval_L4_op
       (synth_L4_op ops1 [] ++ additionalOps) s
       (x :: xs)
IHops2: forall (additionalOps : L4_op)
       (s : list N) (x : N) (xs : list N),
       eval_L4_op additionalOps
       (eval_L3_op ops2 x :: s)
       (x :: xs) =
       eval_L4_op
       (synth_L4_op ops2 [] ++ additionalOps) s
       (x :: xs)
additionalOps: L4_op
s: list N
x: N
xs: list N
________________________________________________
eval_L4_op additionalOps
  (eval_L3_op (L3WriteX ops1 ops2) x :: s)
  (x :: xs) =
eval_L4_op
  (synth_L4_op ops1 [] ++
   [L4WriteXStart] ++
   synth_L4_op ops2 [] ++
   [L4WriteXEnd] ++ [] ++ additionalOps) s
  (x :: xs)
```

```
 rewrite <- IHops1. simpl.
```

```
________________________________________________
eval_L4_op additionalOps
  (eval_L3_op ops2 (eval_L3_op ops1 x) :: s)
  (x :: xs) =
eval_L4_op
  (synth_L4_op ops2 [] ++ L4WriteXEnd :: additionalOps)
  s (eval_L3_op ops1 x :: x :: xs)
```

```
 rewrite <- IHops2. simpl.
```

```
----------------------------------------------------------------
eval_L4_op additionalOps
  (eval_L3_op ops2 (eval_L3_op ops1 x) :: s)
  (x :: xs) =
eval_L4_op additionalOps
  (eval_L3_op ops2 (eval_L3_op ops1 x) :: s)
  (x :: xs)
```

```
  reflexivity.
```

For the arithmetic operators, each case follows from the same reasoning except that the inductive hypotheses are applied in reverse order. This is due to the order in which those operations are placed on the stack being different, as can be seen by comparing Figure 3.2 and Figure 3.3.

```
- intros.
```

abbreviated.

```
ops1, ops2: L3_op
IHops1: forall (additionalOps : L4_op)
       (s : list N) (x : N) (xs : list N),
       eval_L4_op additionalOps
       (eval_L3_op ops1 x :: s)
       (x :: xs) =
       eval_L4_op
       (synth_L4_op ops1 [] ++ additionalOps) s
       (x :: xs)
IHops2: forall (additionalOps : L4_op)
       (s : list N) (x : N) (xs : list N),
       eval_L4_op additionalOps
       (eval_L3_op ops2 x :: s)
       (x :: xs) =
       eval_L4_op
       (synth_L4_op ops2 [] ++ additionalOps) s
       (x :: xs)
additionalOps: L4_op
s: list N
x: N
xs: list N
----------------------------------------------------------------
eval_L4_op additionalOps
  (eval_L3_op (L3Add ops1 ops2) x :: s)
  (x :: xs) =
eval_L4_op
  (synth_L4_op ops2 [] ++
   synth_L4_op ops1 [] ++
   [L4Add] ++ [] ++ additionalOps) s
  (x :: xs)
```

```
rewrite <- IHops2. simpl.
```

```
______________________________________
eval_L4_op additionalOps
  ((eval_L3_op ops1 x + eval_L3_op ops2 x)%N :: s)
  (x :: xs) =
eval_L4_op
  (synth_L4_op ops1 [] ++ L4Add :: additionalOps)
  (eval_L3_op ops2 x :: s)
  (x :: xs)
```

```
rewrite <- IHops1. simpl.
```

```
______________________________________
eval_L4_op additionalOps
  ((eval_L3_op ops1 x + eval_L3_op ops2 x)%N :: s)
  (x :: xs) =
eval_L4_op additionalOps
  ((eval_L3_op ops1 x + eval_L3_op ops2 x)%N :: s)
  (x :: xs)
```

```
reflexivity.
```

The near-identical cases for *L3Sub* and *L3Mul* are omitted for brevity.

```
Qed.
```

Now that we have proved the helper lemma, the proof that layer 4 refines layer 3 follows easily.

```
Lemma refinement_proof_L3L4 :
  forall (ops : L3_op) Ax Cx (ValidInit : input_relation_L3L4 Ax Cx),
    result_relation_L3L4
      (eval_L3_op (ops) Ax)
      (eval_L4_op (synth_L4_op (ops) []) [] Cx).
Proof.
```

After simplification, first we specialise the helper lemma to our current case.

```
  abbreviated.
  pose proof (refinement_proof_helper ops [] [] Ax []) as helper.
```

```
ops: L3_op
Ax: N
helper: eval_L4_op [] [eval_L3_op ops Ax] [Ax] =
        eval_L4_op (synth_L4_op ops [] ++ []) [] [Ax]
______________________________________
Some [eval_L3_op ops Ax] =
eval_L4_op (synth_L4_op ops []) [] [Ax]
```

```
  simpl in *.
```

```
ops: L3_op
Ax: N
helper: Some [eval_L3_op ops Ax] =
       eval_L4_op (synth_L4_op ops [] ++ []) [] [Ax]
------------------------------------------------------------
Some [eval_L3_op ops Ax] =
eval_L4_op (synth_L4_op ops []) [] [Ax]
```

Now we can see that the only difference between our helper lemma and the goal is that an empty list is appended in the helper lemma. The Coq standard library lemma *app_nil_r* allows us to rewrite this, allowing our goal to be solved.

```
  rewrite app_nil_r in helper.
```

```
ops: L3_op
Ax: N
helper: Some [eval_L3_op ops Ax] =
       eval_L4_op (synth_L4_op ops []) [] [Ax]
------------------------------------------------------------
Some [eval_L3_op ops Ax] =
eval_L4_op (synth_L4_op ops []) [] [Ax]
```

```
  assumption.
Qed.
```

### 3.3.8 Layer 1-4 Stepwise Refinement: Coq Functions (Unary) to Stack Machine (Binary)

To help state that layer 4 refines layer 1 we compose the result relations to enable us to phrase the final stepwise refinement proof.

Definition *result_relation_L1L4 l1 l4* := $\exists$ *l2 l3*, (*result_relation_L1L2 l1 l2*) $\wedge$ (*result_relation_L2L3 l2 l3*) $\wedge$ (*result_relation_L3L4 l3 l4*).

We can now state the central theorem of this section, that layer 1 is refined by layer 4.

```
Theorem stepwise_refinement_proof :
  forall (ops : L2_op) L1x L2x L3x L4x
    (ValidInit12 : input_relation_L1L2 L1x L2x)
    (ValidInit23 : input_relation_L2L3 L2x L3x)
    (ValidInit34 : input_relation_L3L4 L3x L4x),
      result_relation_L1L4
        (eval_L1_op (synth_L1_op ops) L1x)
        (eval_L4_op (synth_L4_op (synth_L3_op ops) []) [] L4x).
Proof.
  intros.
  unfold input_relation_L1L2, input_relation_L2L3,
         input_relation_L3L4, result_relation_L1L4 in *.
```

```
ops: L2_op
L1x, L2x: nat
L3x: N
L4x: stack
ValidInit12: L1x = L2x
ValidInit23: N.of_nat L2x = L3x
ValidInit34: [L3x] = L4x
____________________________________________
exists (l2 : nat) (l3 : N),
  result_relation_L1L2
    (eval_L1_op (synth_L1_op ops) L1x) l2 /\
  result_relation_L2L3 l2 l3 /\
  result_relation_L3L4 l3
    (eval_L4_op (synth_L4_op (synth_L3_op ops) []) []
       L4x)
```

After unfolding the definitions, we provide the witnesses which represent the result values of the intermediary steps in the refinement. The witnesses are the results of evaluating the appropriately compiled operations at layer 2 and layer 3.

```
  exists (eval_L2_op ops L2x).
  exists (eval_L3_op (synth_L3_op ops) L3x).
```

We now have three goals, which each correspond to one of the previously proved refinement proofs.

```
  split; [|split].
```

```
____________________________________________
result_relation_L1L2
  (eval_L1_op (synth_L1_op ops) L1x)
  (eval_L2_op ops L2x)
```

```
____________________________________________
result_relation_L2L3
  (eval_L2_op ops L2x)
  (eval_L3_op (synth_L3_op ops) L3x)
```

```
____________________________________________
result_relation_L3L4
  (eval_L3_op (synth_L3_op ops) L3x)
  (eval_L4_op (synth_L4_op (synth_L3_op ops) []) []
     L4x)
```

```
  - apply refinement_proof_L1L2; assumption.
  - apply refinement_proof_L2L3; assumption.
  - apply refinement_proof_L3L4; assumption.
Qed.
```

This proves that the Coq functions dealing with unary natural numbers are refined by the stack machine dealing with binary natural numbers, giving us confidence that specifications that hold for the Coq functions also hold for executions of the stack machine.

### 3.3.9 Proving a Property of a Simple Arithmetic Expression

Having shown that layer 4 operations refine layer 1 operations, we are now able to prove properties about a particular simple arithmetic expression such as our example: "let $y = 2x$ in $y + 4$". The specifications are phrased about the synthesised layer 1 operations, which are synthesised from the layer 2 operations. Recall that our *example_L2* once synthesised to a layer 1 operation and partially unfolded is as shown below.

```
________________________________________
synth_L1_op example_L2 =
(fun x : nat => let xNew := 2 * x in xNew + 4)
```

Clearly, one property of this function is that the result is always even. We can prove this as follows.

First we define *Even*, similarly to the mathematical definition of evenness.

Open Scope *nat*.
Definition *Even* $(n : nat)$ : Prop :=
  $\exists\ m,\ n = 2 \times m$.

Now we show that synthesised layer 1 operations always produce an even number, for all inputs $x$.

```
Lemma Even_example_L2_proof : forall x, Even (eval_L1_op (synth_L1_op example_L2) x).
Proof.

  intros; unfold synth_L1_op, eval_L1_op, example_L2, Even.
```

```
x: nat
________________________________________
exists m : nat, 2 * x + 4 = 2 * m
```

After unfolding the definitions and some automatic simplification we can see that we need to provide a witness $m$ which is half of $2 \times x + 4$. Clearly, $x + 2$ is a valid witness.

```
  exists (x + 2).
```

```
x: nat
________________________________________
2 * x + 4 = 2 * (x + 2)
```

The tactic for linear integer arithmetic can now solve the goal.

```
  lia.

Qed.
```

We could make use of the stepwise refinement proof to show that this property holds for the version of *example_L2* compiled down into a stack machine. Though since we know that the results always match this is somewhat unnecessary. For the sake of completeness, here is the proof.

First we define what a valid input stack must look like for layer 4. It must be a stack with exactly one element.

```
Definition ValidL4x (L4x : stack) :=
  match L4x with
    | [_] ⇒ True
    | _ ⇒ False
  end.
```

Next we define a helper lemma involving the result relation between layer 1 and layer 4.

```
Lemma result_relation_L1_L4_helper : forall L1 L1' L4,
      result_relation_L1L4 L1 L4
  ->  result_relation_L1L4 L1' L4
  -> L1 = L1'.
Proof.
```

The proof is omitted for brevity. It relies upon the injectivity of the *N.of_nat* function.

```
  abbreviated.
  apply Nat2N.inj in H0; auto.

Qed.
```

Now we can state that evaluating the compiled version of *example_L2* will always give an even result, with respect to the *result_relation_L1L4*.

```
Theorem Even_example_L2 :
forall L4x,
  ValidL4x L4x ->
forall result,
  result_relation_L1L4
    result
    (eval_L4_op (synth_L4_op (synth_L3_op example_L2) []) [] L4x)
-> Even result.
Proof.
```

Here we make use of the helper lemma proved earlier to show that *result* equals the evaluation of *example_L2* on the initial value for *x* converted to unary form.

```
  abbreviated.
```

```
n: N
H: ValidL4x [n]
result: nat
H0: result_relation_L1L4 result
   (eval_L4_op
   (synth_L4_op (synth_L3_op example_L2) []) []
   [n])
H1: result_relation_L1L4
   (eval_L1_op (synth_L1_op example_L2)
   (N.to_nat n))
   (eval_L4_op
   (synth_L4_op (synth_L3_op example_L2) []) []
   [n])
________________________________________
Even result
```

```
 pose proof (result_relation_L1_L4_helper _ _ _ H0 H1).
 rewrite H2.
```

```
________________________________________
Even
  (eval_L1_op (synth_L1_op example_L2) (N.to_nat n))
```

Now we are in a position to make use of the *Even_example_L2_proof*, having established that *result* is of a form which the lemma applies to.

```
 apply Even_example_L2_proof.
Qed.
```

This demonstrates how the properties proved at the level of Coq functions apply at the deepest level in the refinement proof also. Here, we have discussed simple arithmetic expressions, but the technique applies generally to full smart contract programs. A concrete smart contract program is analogous to a specific example such as *example_L2* presented here.

### 3.3.10 Verified Compilation of Simple Arithmetic Expressions

Importantly, this process of stepwise refinement also provides us with a verified compiler from layer 2 down to the deepest layer, along with the assurance that the behaviour of layer 1 matches the other layers. The compiler is simply the composition of each of functions synthesising to deeper layers. For our arithmetic expressions example we could define our compiler as follows.

`Definition` *compile* (*ops* : *L2_op*) := *synth_L4_op* (*synth_L3_op* *ops*) [].

We can then compile our example based on the informal `let` $y = 2x$ `in` $y + 4$ to stack machine code.

```
Eval compute in compile example_L2.
```

```
= [L4ReadX; L4Const 2; L4Mul; L4WriteXStart;
  L4Const 4; L4ReadX; L4Add; L4WriteXEnd]
: L4_op
```

### 3.3.11 Trusted Computing Base

The trusted computing base for this verified compiler includes only the semantics function for the bottom layer, *eval_L4_op*, and the input and output relations. The functions to synthesise different layers' operations which are used to define the compiler are *not* part of the trusted computing base. The higher layers' semantics functions of the form *eval_L*_op* are also not part of the trusted computing base, because their behaviour is constrained by the input and output relations and the trusted bottom layer. Of course, Coq itself is also part of the trusted computing base. This is a fairly minimalistic trusted computing base and demonstrates the usefulness of writing a formal semantics in Coq for a low-level language (like *L4_op*) which is a starting point for this technique to be applied to a 'real-world' language. For DeepSEA, the low-level semantics is Hirai's Lem semantics [62]. The Coq semantics for Michelson [140] also lend themselves to having this technique applied in the development of a higher level language for Michelson with a verified compiler.

# Chapter 4

# Safe Smart Contract Language Design

This chapter introduces DeepSEA as a programming language for safe and trustworthy smart contracts, with a focus on the parts that I have been most closely involved with. Due to the design of smart contracts on Ethereum, reentrancy is an issue that all smart contract languages targeting the Ethereum platform need to handle - or risk opening the door to vulnerabilities such as the one affecting The DAO smart contract [38] leading to the loss of the equivalent of over 50 million USD at the time. The technique described in Section 4.1 is an existing approach that ensures any reentrancy is benign, and our work implemented that approach in DeepSEA. The second improvement to DeepSEA, explored in Section 4.2, was the development of a high-level blockchain model, with a focus on the aspects of the blockchain which are relevant to smart contract correctness. This model is designed to be a thin wrapper around the DeepSEA smart contract functions enabling the specification of properties that deal with the contract as a whole, or properties of the contract over its lifetime.

## 4.1 Coping with Reentrancy

This section is based upon our paper: *Using Coq to Enforce the Checks-Effects-Interactions Pattern in DeepSEA Smart Contracts* [12]. It has been updated to properly handle when a smart contract reads a balance, which should not occur after an "effect" as well as with other commentary related to the approach taken in the latest version of the blockchain model. This work and the paper were written in collaboration with Vilhelm Sjöberg and Steve Reeves.

Using the DeepSEA system for smart contract proofs, we investigated how to use the Coq theorem prover to enforce that smart contracts follow the *Checks-Effects-Interactions Pattern*. This pattern is widely understood to mitigate the risks associated with reentrancy. For example, a common IDE for Ethereum smart contract development, Remix, already has a tool to check that the *Checks-Effects-Interactions Pattern* has been followed as part of the Solidity Static Analysis module which is available as a plugin [141].

Reentrancy involves a smart contract $C$ that triggers the execution of code of another smart contract $D$ which then calls a function in the original smart contract $C$ before the original execution of $C$ has completed.

However, when not handled properly, reentrancy can cause a smart contract to behave incorrectly and be exploited. This happens with malicious reentrancy, which maliciously exploits the situation that the original execution of $C$ has not completed.

On the Ethereum blockchain, interacting with a malicious smart contract is even possible when transferring ether. This is because if the recipient is a smart contract then it has the opportunity to run some code on receiving funds.

This issue can be mitigated by following the *Checks-Effects-Interactions Pattern* which suggests that a smart contract should first do the relevant *Checks*, then make the relevant internal changes to its state (*Effects*), and only then interact with other smart contracts which may well be malicious. We make the stricter constraint that only one effect is permitted. When following the *Checks-Effects-Interactions Pattern* a reentrant call is essentially no different to a call that is initiated after the first call is finished. As a result of this, no additional risk from malicious reentrant calls is possible.

There is an exception to this. A malicious contract could also stage a Denial of Service attack by throwing an error or consuming too much gas when called by a smart contract function. This can then cause the function of the original smart contract to fail. It is possible for errors to be caught and simply cause a return value of `false` but as DeepSEA is implemented currently it would revert. Changing this in DeepSEA would be valuable future work. But regardless, out of gas exceptions would still provide an avenue for this kind of Denial of Service attack. As discussed by Samreen and Alalfi [142] this is most problematic when a contract ‘pushes’ out funds by iterating through a loop of transfers. A loop of transfers is not possible when following the strict version of the *Checks-Effects-Interactions Pattern* as there can only be at most one transfer per function call, but even one transfer failing can block critical updates to storage variables, for example. To mitigate this issue, the best practice is to allow recipients to ‘pull’ their funds from a contract function which has the pulling of an individual’s funds as its sole purpose, rather than ‘push’ them to the recipient or intertwine pushing funds with other contract functionality. In that scenario, the recipient can at worst cause a Denial of Service for themselves only. Ensuring that this best practice is adhered to currently needs to be checked manually. An example of this issue occurring on a real blockchain is situation with the King of the Ether Throne smart contract [143] where the contract attempted to transfer funds to wallet contracts but an out of gas exception occurred, resulting in the funds not being sent to the correct recipients and manual intervention was required to address the situation.

Without following the *Checks-Effects-Interactions Pattern*, the modelling of smart contract execution when there is the possibility of reentrancy would be more difficult in general and the related correctness proofs would be complex as well. As mentioned, even modelling the humble ether transfer needs to take the possibility of reentrancy into account.

Using the DeepSEA [99] system for proofs about smart contract correctness, a method of enforcing the *Checks-Effects-Interactions Pattern* has been developed. Enforcing the *Checks-Effects-Interactions Pattern* greatly simplifies the modelling of any action that might involve external calls (including ether transfers).

Tangibly, enforcing the *Checks-Effects-Interactions Pattern* means that the DeepSEA code for a smart contract function shown on the left (Listing 4.1) should not be permitted and the code shown on the right (Listing 4.2) should be allowed.

Listing 4.1: DeepSEA code not following the pattern

```
let notFollowingPatternExample() =
  transferEth(msg_sender, 0u42);
  transferSuccessful := true
```

Listing 4.2: DeepSEA code following the pattern

```
let followingPatternExample() =
  transferSuccessful := true;
  transferEth(msg_sender, 0u42)
```

The end result of this work is a system which automatically proves that the *Checks-Effects-Interactions Pattern* has been followed for most cases when it indeed has been, although there are some cases where the *Checks-Effects-Interactions Pattern* has been followed, but this system cannot prove it, as a compromise for automation.

The key aspects of this work are the following:

- A Coq [8] proposition formalising the notion of a smart contract function following the *Checks-Effects-Interactions Pattern*. This is discussed in Section 4.1.1.3.
- Automated proofs related to the previous contribution. See Section 4.1.2.

### 4.1.1 Representing the Absence of Reentrancy Situations as a Proof Goal

#### 4.1.1.1 The *Checks-Effects-Interactions Pattern*

The *Checks-Effects-Interactions Pattern* suggests that a smart contract should follow a pattern in which calls to external contracts are always the last step [37].

When following the *Checks-Effects-Interactions Pattern*, calls that are a result of reentrancy are equivalent to calls invoked one after another as the reentrant calls cannot influence the outcome of the original call (excluding considering gas and thrown exceptions). This enables a simpler model than completely modelling reentrancy with co-recursive functions. The simpler model is considered to be equivalent to a complete model in terms of modelling what states are reachable, and we rely on an informal knowledge for this. Ideally, we would like to model reentrancy foundationally making use of the EVM semantics and then prove that the simple model is equivalent to the more complex model in the case where the *Checks-Effects-Interactions Pattern* is followed. This could be future work.

The stricter version of the *Checks-Effects-Interactions Pattern* is considered here where only one *Interaction* is permitted. This eliminates modelling complications in the situations where two external calls are done, but the first one turns out to throw an error. It is virtually impossible to know, when modelling, whether an arbitrary external call will throw an error, particularly due to the possibility of gas being exhausted. As a result, in the model there would be uncertainty as to whether the effects of the second external call should be treated as having occurred or not. Having a limit of one external call only makes the model clearer.

This strict version of the *Checks-Effects-Interactions Pattern* will now simply be referred to as the *CEIP*.

#### 4.1.1.2 Relevant Aspects of the DeepSEA System

Listing 4.3 shows the same DeepSEA smart contract function in different representations. The intermediate level and high level representation are both generated automatically from the DeepSEA source code. First, the intermediate level abstract syntax tree in Coq is generated from the source code. The denotational

Listing 4.3: ‘Safe’ function in different representations with similarities highlighted

**DeepSEA smart contract source code (not Coq):**
```
let safeExample() =
  transferSuccessful := true ;
  transferEth(msg_sender, 0u42)
```
**DeepSEA intermediate level language in Coq:**
```
(CC sequence
(CCstore
  (LCvar Contract_transferSuccessful := true_var)
  (ECconst_int256 tint_bool true Int256.one))
(CC transfer
  (@ECbuiltin0 _ _ _ builtin0_caller_impl)
  (ECconst_int256 tint_U (Int256.repr 42_var))
  (Int256.repr 42))))
```
**DeepSEA high level language in Coq:**
```
(get;;
MonadState.modify (update_Contract_transferSuccessful true)) ;;
d <- get;;
(let (success, d') :=
me_transfer me (me_caller me) (Int256.repr 42) d in
if Int256.eq success Int256.one then put d' else mzero)
```

semantics of the AST gives the high level representation (by the `synth_stmt_spec_opt` Coq function as a part of the DeepSEA system). The AST for each function contains the relevant information required to formulate the notion of whether the function adheres to the *CEIP*. The inductive proposition described in the next section makes use of the intermediate level AST representation. The highlighted code shows the correspondence between certain constructs at different levels of abstraction.

### 4.1.1.3 Coq Inductive Proposition: `cmd_constr_CEI_pattern_prf`

The typing rule (Figure 4.1) corresponds to the definition of `cmd_constr_CEI_pattern_prf` which is an inductive proposition in Coq capturing the notion of a function following the *CEIP*. The typing rule is based upon the syntax of the smart contract as represented in the DeepSEA intermediate level language. It uses the assumption that reentrancy is only possible when certain syntax, such as `CCtransfer` is encountered. `CCtransfer` is the intermediate level language construct corresponding to a `transferEth` call. The 🟢 icon indicates that the contract cannot in any way have triggered reentrancy yet and the 🟠 icon indicates that reentrancy may have been triggered by that point (and so no unsafe commands such as writing to storage should be allowed after that point). The 🔴 icon would indicate a contract that is vulnerable to malicious reentrancy but does not occur in the typing rule as the rule defines what is safe. In addition, the $ icon indicates that the expression does not contain a call to read a balance, and $ indicates that the expression does. Reading a balance is similar to reading from storage and is only safe to do before an external call, when following the *CEIP*.

The transfer related rule in Coq is shown in Listing 4.4. The notion that at most one external call is allowed is captured by the fact that the proof requires the state `Safe_no_reentrancy` (🟢) beforehand. Due to the transfer the contract is then in a state where reentrancy may have occurred and this is captured by the state `Safe_with_potential_reentrancy` (🟠).

Figure 4.1: Typing rule for a command that adheres to the *CEIP*, corresponding to the Coq inductive proposition `cmd_constr_CEI_pattern_prf`. (Some rarely used rules have been omitted). $\rho_x \in \{\text{🟢}, \text{🟠}\}$.

$$\frac{}{\{\rho\}\ \texttt{skip}\ \{\rho\}} \qquad \frac{\{\rho_1\}\ c_1\{\rho_2\} \qquad \{\rho_2\}\ c_2\ \{\rho_3\}}{\{\rho_1\}\ \texttt{let}\ x = c_1\ \texttt{in}\ c_2\ \{\rho_3\}} \qquad \frac{}{\{\text{🟢}\}\ \texttt{load}\ \{\text{🟢}\}} \qquad \frac{}{\{\text{🟢}\}\ e_1 := e_2\ \{\text{🟢}\}}$$

$$\frac{\{\rho_1\}\ c_1\{\rho_2\} \qquad \{\rho_2\}\ c_2\{\rho_3\}}{\{\rho_1\}\ c_1\ ;\ c_2\ \{\rho_3\}} \qquad \frac{\{\text{🟢}\}\ c_{true}\{\text{🟢}\} \qquad \{\text{🟢}\}\ c_{false}\{\text{🟢}\}}{\{\text{🟢}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟢}\}}$$

$$\frac{\{\text{🟢}\}\ c_{true}\{\text{🟠}\} \qquad \{\text{🟢}\}\ c_{false}\{\text{🟢}\}}{\{\text{🟢}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟠}\}} \qquad \frac{\{\text{🟢}\}\ c_{true}\{\text{🟢}\} \qquad \{\text{🟢}\}\ c_{false}\{\text{🟠}\}}{\{\text{🟢}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟠}\}}$$

$$\frac{\{\text{🟢}\}\ c_{true}\{\text{🟠}\} \qquad \{\text{🟢}\}\ c_{false}\{\text{🟠}\}}{\{\text{🟢}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟠}\}} \qquad \frac{e_1\text{\$}\cancel{\text{👁}} \qquad \{\text{🟠}\}\ c_{true}\{\text{🟠}\} \qquad \{\text{🟠}\}\ c_{false}\{\text{🟠}\}}{\{\text{🟠}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟠}\}}$$

$$\frac{e_1\text{\$👁} \qquad \{\rho\}\ c\ \{\rho\}}{\{\text{🟢}\}\ \texttt{for}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c\ \{\text{🟢}\}} \qquad \frac{e_2\text{\$👁} \qquad \{\rho\}\ c\ \{\rho\}}{\{\text{🟢}\}\ \texttt{for}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c\ \{\text{🟢}\}} \qquad \frac{e_1\text{\$}\cancel{\text{👁}} \qquad e_2\text{\$}\cancel{\text{👁}} \qquad \{\rho\}\ c\ \{\rho\}}{\{\rho\}\ \texttt{for}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c\ \{\rho\}}$$

$$\frac{\{\rho_1\}\ \mathit{function}\ \{\rho_2\}}{\{\rho_1\}\ \mathit{function\ call}\ \{\rho_2\}} \qquad \frac{}{\{\text{🟢}\}\ \texttt{transferEth}(e_1, e_2)\ \{\text{🟠}\}} \qquad \frac{\{\rho\}\ c\ \{\rho\}}{\{\rho\}\ \texttt{assert}\ c\ \{\rho\}}$$

$$\frac{\{\rho\}\ c\ \{\rho\}}{\{\rho\}\ \texttt{deny}\ c\ \{\rho\}}$$

In terms of calling functions in external contracts, there are three main situations to be aware of. Firstly, calling a function whose definition is unknown (implicitly, this is potentially the case when doing a transfer to an address about which nothing has been assumed). This gives no guarantees about reentrancy safety and should only be considered safe when all actions afterwards are safe even after reentrancy (i.e. "$\{\text{🟠}\}\ c\ \{\text{🟠}\}$"). Secondly, calling an external contract which is unknown except for its 'colours'. Here we might want to incorporate our assumption about the 'colours' of the unknown function we are calling into our proof so that we can still be sure the *CEIP* is being followed. This is not yet supported fully by this work, though it could be added in future work. However, most of the time if you know an external function's 'colours' you also know its definition. If the external contract is not written in DeepSEA, it would be best to write a faithful copy of the original contract in the DeepSEA language and have DeepSEA generate the functional specification as well as what is needed by the *CEIP* checker automatically (however there are other approaches).

If the contract is already in DeepSEA, the process is trivial. Either way, once in DeepSEA and given the assumption that the external contract behaves as described by the DeepSEA version, then it turns out that for external contracts whose behaviour is exactly known, those external calls can be treated as if they were internal calls for the sake of checking if the *CEIP* is followed. We can see this by considering if there is any difference in control flow between one DeepSEA contract implementing the functionality of two by having an additional lower layer, and two DeepSEA contracts which together implement the same functionality as the original one contract by using external calls to the second contract (acting as a lower layer) with appropriate adjustments relating to storage variables.

Listing 4.4: Defining *CEIP* adherence for `CCTransfer`

```
| CCCEIPtransfer :
 forall e1 e2,
  cmd_constr_CEI_pattern_prf
  _ (* Infer the return type *)
  Safe_no_reentrancy (* 🟢 *)
  (CCtransfer e1 e2) (* Typically related to a 'transferEth' call. *)
  Safe_with_potential_reentrancy (* 🟠 *)
    (* After, the possibility of reentrancy is noted. *)
```

Listing 4.5: Defining *CEIP* adherence for `CCFor`

```
| CCCEIPfor :
 forall {ρ} id_it id_end e1 e2 c,
  cmd_constr_CEI_pattern_prf _ ρ c ρ
    (* Given a command that stays at state ρ *)
  -> cmd_constr_CEI_pattern_prf _ ρ (CCfor id_it id_end e1 e2 c) ρ
    (* Then the for loop as a whole stays at state ρ *)
```

It is worth noting that in DeepSEA layers have a hierarchy with higher layers only being able to call lower layers, so we can rule out cyclic contract calls in the case where the contracts are defined together in DeepSEA. Although it is possible in principle to set an external contract address via a function call in a deployed contract and have cyclic calls between two DeepSEA contracts, for now that setup would fall into the category of the external contract being unknown. An assumption of the form "At some point, the contract will have a storage variable set to the address of a specific contract with a known definition, and this assumption should be incorporated into whether the *CEIP* is considered to be followed" is beyond the scope of this work. It is not in principle impossible to implement, but it would be non-trivial for what is probably an unlikely scenario.

In Listing 4.5 we define that if the body of a for loop stays at state $\rho$ (either "{🟢} $c$ {🟢}" or "{🟠} $c$ {🟠}") then the for loop as a whole is also defined to stay at state $\rho$. The first rule shown in Figure 4.2 is equivalent to the combination of the two rules following it. The rightmost typing rule in Figure 4.2 is correct because a command that is safe after reentrancy is also safe before reentrancy. However, if the rule instead included "{🟢} $c$ {🟠}" that would be incorrect. It is important that $\rho$ stays in the same state in this rule, and other rules which have $\rho$ the same before and after a command. In rules where $\rho$ can differ this is shown by a subscript, e.g. $\rho_1$. In those rules wherever $\rho$ occurs with a specific subscript the 'colour' must match wherever else it occurs with the same subscript.

Figure 4.2: Selected typing rules related to `for` loops

$$\frac{e_1\text{\$👁} \qquad \{\rho\}\ c\ \{\rho\}}{\{🟢\}\ \texttt{for}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c\ \{🟢\}} \qquad \frac{e_1\text{\$👁} \qquad \{🟢\}\ c\ \{🟢\}}{\{🟢\}\ \texttt{for}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c\ \{🟢\}} \qquad \frac{e_1\text{\$👁} \qquad \{🟠\}\ c\ \{🟠\}}{\{🟢\}\ \texttt{for}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c\ \{🟢\}}$$

The remaining definitions are available in the GitHub repository as described in Appendix A. This defines what it means for a DeepSEA smart contract function to follow the *CEIP*. If `cmd_constr_CEI_pattern_prf` can be proved for a given function then that function follows the *CEIP*.

Listing 4.6: Coq tactic to prove adherence to the *CEIP*

```
Ltac CEI_auto :=
  repeat (
    reflexivity
  + typeclasses eauto
  + eapply CCCEIPskip + eapply CCCEIPlet + eapply CCCEIPload
  + eapply CCCEIPfor + eapply CCCEIPtransfer + ... ).
```

A drawback of this formulation is that interrelated 'if' statements are not able to be reasoned about. If the logical content of interrelated 'if' statements made it possible to know the *CEIP* was indeed followed, this formulation would not allow those functions to be proved to be safe. This does however simplify proof automation. An alternative approach which made use of the high level representation of the smart contract was also explored. This "instrumented semantics" approach added reentrancy state tracking to the semantics of DeepSEA, and as a result *is* able to reason about interrelated 'if' statements. This alternative approach still assumes specific syntactic elements correspond to the possibility of causing reentrancy.

Another drawback (with both approaches) is that other techniques to manage reentrancy issues such as locks are not considered to be safe by these methods, even when they may have been used in a way which is safe. On the other hand, this does simplify modelling by only needing to consider cases equivalent to when no reentrancy occurs.

### 4.1.2 Automatically proving the absence of reentrancy situations

Now that we have defined the notion of a smart contract following the *CEIP* the goal is to automatically prove this for most functions that follow the *CEIP*. To automatically prove this for all possible functions is impossible in general, so to make the automatic proof tractable we focus on most functions which follow the *CEIP* in a straightforward manner. In particular, one requirement is that both branches of all 'if' statements follow the *CEIP* even if certain combination of branches are impossible to reach, as discussed above. The automation will be carried out by Coq tactics.

The tactic, partially shown in Listing 4.6, will resolve typeclass goals automatically and repeatedly apply the constructors from the `cmd_constr_CEI_pattern_prf` definition. The + used to combine the tactics is critical to ensure the tactic backtracks as necessary because sometimes it is not the first matching constructor that is relevant.

See the GitHub repository as described in Appendix A for the full definitions of all the tactics involved. The proofs are done automatically and provide the user with an error if they fail (which would likely indicate the *CEIP* was not followed).

### 4.1.3 Simplifying the Modelling of Ether Transfer

The fact that we are following the *CEIP* simplifies the modelling of ether transfer due to the fact that reentrant calls can be considered to be called one after another as no reentrant calls can influence the outcome of the original call (excluding gas considerations), as discussed in Section 4.1.1.1. This means that when considering what states are reachable it is sound to treat the transfer as only affecting ether balances and ignore any other potential state changes. Also, since we are following the strict version of the *CEIP* we know that there is at most one call to `transferEth` which further simplifies the modelling.

When modelling ether transfer in DeepSEA, currently it is implicitly assumed that the *CEIP* is followed. We record at most one transfer as an optional type and if somehow two transfers occurred the first would be overwritten in the model rather than causing a second transfer. The validity of this approach is based on the reasoning that there can only be at most one transfer if the *CEIP* is followed and that the *CEIP* must be proved to have been followed for every DeepSEA function. A previous version of this work[1] modelled a list of transfers being produced and included an automatically generated proof that the list of transfers generated was at most length one. The approach of modelling a list of transfers turned out to be challenging to handle at later points in developing the model of a blockchain and as a result the approach of implicitly assuming that there was at most one transfer was preferred.

This technique simplifies the modelling of ether transfer without leaving the door open for malicious reentrancy. The proofs are automated, only requiring the DeepSEA smart contract programmer to follow the strict version of the *CEIP*.

### 4.1.4 Remarks

A number of other tools aim to tackle the problem of reentrancy, such as [16], [86], [144], [145]. Our work describes an approach for representing and automatically proving that DeepSEA smart contracts follow the *CEIP*. This is demonstrated by defining an inductive proposition in Coq that states that a particular smart contract function follows the *CEIP*. An automated proof of the inductive proposition for a wide range of smart contract functions which follow the *CEIP* is given. The impact of this approach on the modelling of ether transfer is then discussed.

## 4.2 Modelling a Blockchain

This section includes content based on our paper "*Modelling a Blockchain for Smart Contract Verification using DeepSEA*" [13]. The paper was written in collaboration with Steve Reeves.

A critical component of taking a rigorous approach to smart contract development is having an adequate model of a blockchain. The model should be sufficiently expressive, but also abstract, to make the phrasing of lemmas and theorems related to the behaviour of smart contracts straightforward.

Ideally, any high-level model of a blockchain should be linked to a low-level model of that blockchain, such as Hirai's Lem formalisation of the EVM [62]. The low-level model should be shown to *refine* the high-level model. Although refinement down to executable EVM bytecode is possible in principle with the blockchain model discussed here, a proof of this is left for future work. However, DeepSEA does already generate an all but completed refinement proofs for each individual function of each smart contract written in DeepSEA, which form a key part of this blockchain model.

[1] `https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021`

### 4.2.1 The Snapshot Approach

#### 4.2.1.1 Motivation

The focus of this model of the blockchain is on the aspects relevant to the correctness of the smart contract written in the DeepSEA language. The model's initial state corresponds with the moment when the smart contract of interest is deployed on the blockchain. The approach taken is a *snapshot* one. The model begins from an arbitrary state of the blockchain and models possible actions from then on. This helps guarantee the validity of the lemmas and theorems proved, regardless of the actual state of the blockchain.

#### 4.2.1.2 Overview

We implement the snapshot approach by universally quantifying over all possible states the blockchain could be in at the point that the smart contract in question is deployed. This is done using a mechanism in Coq [8] called *section variables*, so that the arbitrary variables can be referred to throughout multiple definitions within the section.

This approach is used to define the reachability predicate, `ReachableFromBy`, which captures the notion of which states are reachable from an original state by a list of states and actions leading from the original state to the state in question. `ReachableFromBy` is used in the crowdfunding `donation_preserved` theorem shown in Listing 4.2.1. The `donation_preserved` theorem is discussed further in Section 5.6. Starting from an arbitrary initial state in the proof helps us be sure that the result applies to the real-world Ethereum blockchain.

Listing 4.2.1: Statement of the `donation_preserved` theorem

```
Definition since_as_long
  (P : BlockchainState → Prop) (Q : BlockchainState → Prop) (R : Step → Prop) :=
    ∀ sc st step',
      ReachableFromBy st step' sc →
      P st →
      (∀ sa, List.In sa sc → R sa) →
      Q (Step_state step').
Notation "Q 'since' P 'as-long-as' R" :=
  (since_as_long P Q R) (at level 1).
Definition donation_recorded (a : addr) (amount : Z) (s : BlockchainState) :=
  Int256Tree.get_default 0 a (Crowdfunding_backers (contract_state s)) = amount
  ∧ amount > 0.
Definition no_claims_from (a : addr) (s : Step) :=
  match Step_action s with
    | (call_Crowdfunding_claim _ a _ _ _) ⇒ False
    | _ ⇒ True
  end.
Theorem donation_preserved :
  ∀ (a : addr) (d : Z),
                  (donation_recorded a d)
    'since' (donation_recorded a d)
    'as-long-as' (no_claims_from a).
```

#### 4.2.1.3 Implementation

To implement the snapshot approach Coq's section variables are used, as mentioned earlier. As shown in Listing 4.2.2, the timestamp, block number, block hashes and balances are taken to be section variables and in the context of the proofs their actual values are arbitrary. The source code relating to this section is available as described in Appendix A.

Listing 4.2.2: Section variables for the model

```
Context
  (snapshot_timestamp : int256)
  (snapshot_number : int256)
  (snapshot_blockhash : int256 → int256)
  (snapshot_balances : addr → wei).
```

It is possible to add additional assumptions to be used in the model such as those in Listing 4.2.3. Here we choose to assume that funds sent by the smart contract are always accepted by the recipient. This assumption is then used in the definition of `Action` (shown later in Listing 4.2.5) by being passed to `make_machine_env` and then used in the function responsible for checking that a transfer is successful.

Listing 4.2.3: Example of an assumption

```
Context (address_accepts_funds : option ContractState → addr → addr → wei → bool).

Definition
  address_accepts_funds_assumed_for_from_contract
    d sender recipient amount :=
      if sender =? contract_address then true
      else address_accepts_funds d sender recipient amount.

Definition address_accepts_funds_assumption :=
  address_accepts_funds_assumed_for_from_contract.
```

Based on the snapshot information we define the `initial_state` of the blockchain as in our model (which is not to be confused with the actual initial state of the blockchain back at the genesis block). This is defined as shown in Listing 4.2.4 and contains all the information assumed about the blockchain up to the point where the snapshot is taken.

Listing 4.2.4: Initial state of the model

```
Definition initial_state :=
  mkBlockchainState
    snapshot_timestamp snapshot_number
    snapshot_balances snapshot_blockhash
    init_global_abstract_data.
```

The blockchain state recorded is focused on what is relevant to the smart contract that is being verified. From this perspective, there is no need to keep track of all of the blockchain's data, only what is in Listing 4.2.4. The `init_global_abstract_data` refers to the initial values of the storage variables of the smart contract.

## 4.2.2 The Successful-Calls Approach

### 4.2.2.1 Motivation

When calling a smart contract function it may ‘revert’ (i.e. cancel) the current execution, which results in it returning its state to the state it was in before the call. Here are some examples of what may cause this:

- A runtime error in a smart contract call, such as an array out of bounds exception;
- Execution reaching a point in the smart contract code explicitly causing a revert. In DeepSEA this is done via `fail` or `assert(false)`;
- The gas cost of the transaction exceeding the gas that is provided by the sender.

When a revert occurs, a transaction fee (gas fee) is still charged to the caller as a charge for the partial computation that was carried out. The current model implemented in DeepSEA ignores the transaction fee (though fully accounting for gas is feasible with DeepSEA in principle). Ignoring the transaction fee means that a revert results in no change at all to the modelled blockchain state, as the calling accounts do not have their balance reduced due to the transaction fee in the model.

### 4.2.2.2 Overview

Typically, in a proof, the cases of interest are those where the smart contract function call succeeds. However, it is still important to model the scenario where the functions revert. From a snapshot, we can model all possible further actions. That is, we can consider the effect of all possible calls to the smart contract and then model the outcome of those calls. This is a satisfactory approach, but we can do better. Bearing in mind that the successful calls are typically the most relevant to consider, we can take a more elegant approach with our model as we are using a proof assistant. We require a proof that each call will not result in a revert and then handle the revert case separately just once. This novel approach allows us to focus the proof effort on the successful calls, while still considering all scenarios.

This approach can also be thought of as using refinement types, where the values passed to the contract have certain properties about them which are assumed to hold, with the revert case covering the scenarios when they do not hold.

### 4.2.2.3 Implementation

The purpose of implementing the successful-calls approach is to make the proofs shorter and more elegant by removing the need to repeatedly prove the statements of lemmas and theorems for the case where the state reverts over and over again.

Listing 4.2.5 defines the transitions (in a state-machine analogy) that can happen. There is one transition for each smart contract function, a transition for balance transfers, a transition for time passing, and a transition for revert.

The subtlety is that the transitions for each smart contract function require not only the arguments to the smart contract function but also a proof that, with those arguments and the current blockchain and contract state, the call to that smart contract succeeds. This is shown in the step function in Listing 4.2.6 with the inclusion of `case_donate_prf` being a required argument, and it is similar for the other smart contract functions (though hidden by the ellipsis). `case_donate_prf` guarantees that the modelled function call will not revert.

This highlights one strength of using this approach in Coq rather than relying on *tests* of the smart contract: when testing, we would need to set up the system so that we are sure, *without even running the test*, that a particular test will not cause a revert, but that is impossible in general.

Listing 4.2.5: Action dependent on the current blockchain state

```
Inductive Action (before : BlockchainState) :=
  | call_Crowdfunding_donate (context : CallContext)
      (callvalue_prf : noOverflowOrUnderflowInTransfer (caller context) contract_address
                             (callvalue context) (balance before) = true)
      r
      contract_state_after
      (case_donate_prf :
          runStateT (Crowdfunding_donate_opt (make_machine_env contract_address
            before context address_accepts_funds_assumption)) (contract_state before)
          = Some (r, contract_state_after))
  | call_Crowdfunding_getFunds ...
  | call_Crowdfunding_claim ...
  | externalBalanceTransfer ...
  | timePassing ...
  | revert.
```

The type of `Action` is `BlockchainState -> Type`, so it is a type which depends upon one argument of type `BlockchainState`. In particular, it depends upon the blockchain state prior to the action being executed, i.e. the argument `before`. This argument is used by `callvalue_prf` and `case_donate_prf` (or the equivalent for each function of the smart contract). The proof relating to `callvalue` ensures that the caller of the smart contract has sufficient funds to send the `callvalue` and that the smart contract's balance does not overflow on receiving those funds. The statement of this proof can only be phrased with the knowledge of the balances before the smart contract is called, hence the need for the `before` variable. The `case_donate_prf` statement relies much more heavily upon the `before` variable because `before` contains the current contract state including the value of the contract's storage variables. When 'calling' `donate` in the model, the 'caller' is required to provide a proof that `Some (r, contract_state_after)` is returned, i.e. that the contract does not revert by returning `None`. The benefit of this approach is that the various situations where the contract reverts are handled once, leaving only the cases which succeed to be proved – these typically are the cases of interest in the proof.

Consider a contract where one of its functions always reverts. For this function we would have a contradiction between the `case_function_prf` and that the function always reverts. By the principle of explosion this fulfils the proof obligation for that function's branch regardless of the specification in question. This is appropriate because the proof obligations related to when any function of the contract reverts are handled once and for all by the branch associated with the final case `revert` shown in Listings 4.2.5 and 4.2.6.

The state transition `step` function for the blockchain state is defined as shown in Listing 4.2.6.

For a smart contract function, the required arguments for an action include the state of the smart contract

Listing 4.2.6: Step function

```
Fixpoint step
  (before : BlockchainState) (action : Action before) : BlockchainState :=
match action with
| call_Crowdfunding_donate context
    callvalue_prf r d_after case_donate_prf ⇒
      next_blockchain_state before d_after
| call_Crowdfunding_claim ...
| call_Crowdfunding_getFunds ...
| timePassing ...
| externalBalanceTransfer ...
| revert ⇒ before
end.
```

after the call, its return value, and a proof that these are in fact the result of executing the function - with the guarantee that it does not revert. Given this, it is trivial to define the step function for smart contract function calls. The step function simply takes the resulting state of the smart contract and wraps it, after processing the possible ether transfer to an external address which the smart contract may initiate.

### 4.2.3 Remarks

A limitation of this model is that it does not (yet) handle gas constraints. It implicitly assumes that the gas cost paid by the sender is always sufficient for the transaction to complete. DeepSEA does actually track gas usage, but it does not yet do so at the high-level of abstraction used in this model.

Although the purpose of the successful-calls approach is to simplify the proof effort, there are some ways in which it does make proofs slightly more complex. The complexities arise from the use of the `Action` type depending on a value of type `BlockchainState`. For example, an intuitive approach as a part of modelling a blockchain is to have a separate type for a list of states and a list of actions. However, to do this with the `Action` type dependent on a `BlockchainState` it is necessary to instead have a list of dependent records which include both a state and action in each element of the list.

An additional challenge is that of ensuring the verification system stays current. For example, there have been changes[2] to the test set which Hirai's Lem model [62] was tested on[3] but Hirai's Lem model has not been updated since 2018. At the time, Hirai's model passed all 40,619 tests, excluding 24 which were out of scope as they involved interactions between multiple contracts [62]. Updating Hirai's Lem model and any required changes in DeepSEA would be important before the full benefits of the DeepSEA system could be realised.

This approach involves modelling the Ethereum blockchain in Coq by making use of the ideas of a snapshot and a successful-calls technique leveraging Hirai's Lem model of the EVM and the DeepSEA system. These ideas help simplify proofs without compromising on correctness. The modelling is expressive enough that it can be used to express properties about smart contracts at a high level.

Future work would involve demonstrating that the aspects of the model which 'glue' the individual functions

[2]Latest version of relevant tests for Hirai's Lem model: https://github.com/ethereum/tests/tree/develop/GeneralStateTests/VMTests.

[3]Likely version of the tests Hirai's Lem model was tested on: https://github.com/ethereum/tests/tree/79b14fbd74c0148f1b8cca442252e9299464fe8c/VMTests.

together into a model of the contract (and blockchain) as a whole are also refined by a low-level blockchain model, such as Hirai's Lem model [62]. In the meantime, careful inspection of the aspects which 'glue' the individual functions together as a part of the high-level model is required, with them being part of the trusted computing base for now. Such careful inspection could involve examining the model's Coq source code to see if it corresponds with common knowledge about how the Ethereum blockchain functions, or perhaps comparison with the Ethereum Yellow Paper [40]. The first of these has been done so far. The high-level nature of the model lends itself to this kind of analysis. Nevertheless, future work involving a refinement proof for all aspects of the model would give greater assurance the model is correct.

## 4.3 Remarks on Safe Smart Contract Language Design

The two improvements to the DeepSEA language discussed in this chapter make it possible to write specifications and proofs for smart contracts that make use of ether transfer, with the confidence that the proofs are still valid in light of the possibility of reentrancy. This expands the types of smart contracts which can be handled by the DeepSEA system greatly. Since ether transfer is so ubiquitous and central to the operation of many smart contracts, it is a critical improvement that was necessary.

There is an interplay between the *Checks-Effects-Interactions Pattern* and the modelling of the blockchain. The model of the blockchain assumes that the *CEIP* is followed, which makes the model simpler and the resulting proofs simpler also. In particular, the model assumes that there can be at most one ether transfer resulting from a smart contract. This demonstrates that having a more restricted language can greatly simplify the modelling and result in simpler proofs. A restricted language can still be expressive enough for most uses, though demonstrating this is left for future work.

Having improved the DeepSEA system as described in this chapter, DeepSEA is now ready to handle the two case studies presented in the following two chapters. Both these case studies make use of ether transfer and as a result rely upon the work described in this chapter.

# Chapter 5

# Case Study: Crowdfunding

This chapter explores the specification, implementation and proof of a crowdfunding smart contract based upon related work by Sergey et al. [15]–[18]. This proof targets an implementation transpiled into Coq automatically via the DeepSEA compiler (whereas the contract was translated by hand from Scilla). Additionally, the individual functions of DeepSEA smart contracts have refinement proofs down to Ethereum Virtual Machine (EVM) bytecode building on the semantics of the EVM in Lem [62].

A crowdfunding campaign, like Kickstarter [146], has the aim of allowing the owner of the campaign to try and raise enough funds (a goal) by a specified time (the deadline). If the goal is met by the deadline, then all the funds are made available to the owner and the campaign is considered to have succeeded. If the goal is not met by the deadline, then all those who donated (the backers) are able to have their funds returned to them and the campaign is considered to have failed. The smart contract explored in this chapter implements this idea.

This case study looks at correctness from the point of view of a donor to a crowdfunding campaign managed by this smart contract; correctness from the point of view of the owner is left for future work. It is beyond the scope of this case study to consider factors like whether a successfully funded campaign actually results in a marketable product or one that meets the claims made by the campaign owner. However, it is well within the capabilities of smart contracts to facilitate most, if not all, of the mission-critical features of a fully-fledged crowdfunding platform like Kickstarter. This would require a more complex smart contract and the use of oracle smart contracts to make certain real-world data available on-chain, such as whether a backer has received the 'backer reward' they may have been promised as part of the campaign. In that case, the proofs would build upon the proofs described in this chapter.

This crowdfunding contract has three functions: *donate*, *claim* and *getFunds*. In addition, the initial deployment of each instance of the crowdfunding contract serves the purpose of establishing the goal, deadline and owner address. Each instance manages exactly one crowdfunding campaign. The *donate* and *claim* functions are intended for use by backers and allow a backer to donate to the campaign and, if the campaign fails, to retrieve their ether. The *getFunds* function allows the campaign owner to retrieve all the funds if it succeeds with the goal met by the deadline.

The blockchain also allows for a number of actions to be possible regardless of the smart contract's code. The focus with the model of the blockchain used for verification is on the actions which may affect the correctness of the smart contract. In particular between any call to a crowdfunding contract's function there may be the passing of time (`block_number` and `timestamp` may advance) and there may be ether transfers between any addresses (except originating from the contract).

## 5.1 Contract Source Code

This section includes the full source code for a crowdfunding smart contract written in DeepSEA. The smart contract is based upon a similar crowdfunding smart contract written in the Scilla language presented by Sergey et al. [15]. While the Scilla version was manually translated into Coq, the individual functions of the DeepSEA version are automatically translated into Coq and a refinement proof down to EVM bytecode is also automatically generated. In addition, the design of the Scilla language with the Zilliqa blockchain excludes the possibility of reentrancy, while DeepSEA makes use of the work discussed in Section 4.1 to cope with the possibility of reentrancy on the Ethereum blockchain.

### 5.1.1 Preliminary Definitions

Listing 5.1 defines a type alias for messages, and defines constants as well as an algebraic type for events. DeepSEA does not yet have the capability to return strings and so the messages that may be emitted are given an integer corresponding to each message. A simple algebraic type is defined to hold event messages. DeepSEA does not facilitate proofs about emitted messages, but these are included in the smart contract to closely parallel the original Scilla contract.

Listing 5.1: Crowdfunding Contract Source Code - Part 1

```
type message := int

const _deadlinePassed_msg = 0
const _successfullyDonated_msg = 1
const _alreadyDonated_msg = 2
const _funded_msg = 3
const _failed_msg = 4
const _too_early_to_claim_funds_msg = 5
const _too_early_to_reclaim_msg = 6
const _cannot_refund_msg = 7
const _here_is_your_money_msg = 8
const _only_owner_can_get_funds_msg = 9

event
    | Message (msg : message)
```

### 5.1.2 Type Signatures

The next portion of code shown in Listing 5.2 declares the type signatures of the three functions of the smart contract: `donate`, `getFunds`, and `claim`. All three functions take no parameters and do not return a value (returning `unit`). However, they do take as input the call context, which includes information such as how much ether a person is transferring to the contract. This information is what is used by `donate` as input, for example.

Listing 5.2: Crowdfunding Contract Source Code - Part 2

```
object signature CrowdfundingSig = {
    donate : unit -> unit;
    getFunds : unit -> unit;
    claim: unit -> unit
}
```

### 5.1.3 State Variables

Next, the state variables are defined as shown in Listing 5.3. Here, the owner, goal and time limit for the crowdfunding contract are hard-coded in, though these could have been set via a constructor instead. Importantly, the two variables which will be altered by the smart contract during its execution are also defined: the `backers` mapping which records the donations, and the `funded` flag which indicates whether the crowdfunding campaign has been successful and that the owner has withdrawn the funds.

Listing 5.3: Crowdfunding Contract Source Code - Part 3

```
object Crowdfunding () : CrowdfundingSig {
    let owner : address := address(0x9Be6210aD2EB7D510C7dBC7eA0C91e4591a9f813)
    let max_block : uint := 0u140 (* The last block number when donations are accepted. *)
    let goal : int := 50

    let backers : mapping[address] int := mapping_init
    let funded : bool := false
```

### 5.1.4 Donate Function

Now we see in Listing 5.4 the first function of the crowdfunding smart contract. The `donate` function allows people to donate to the campaign. The function takes no arguments because the implicit call context contains the amount of ether they send to the contract, which is taken to mean the amount they are donating to the campaign, as well as the new backer's Ethereum address.

The assert statements insert runtime checks which are then added as assumptions in the context when proving properties about `donate`. Integers in DeepSEA are unsigned, however, due to a quirk in the way integers are modelled in DeepSEA (they are represented by the Coq type `Z` rather than `N`), the second assertion `assert(msg_value >= 0)` is required.

The remainder of the `donate` function enforces the business logic of donating. A donation must occur only until `_max_block` (inclusive). In this smart contract, each Ethereum address can only back the campaign once, otherwise the `_alreadyDonated_msg` error code is emitted. If a donation is successful, the `backers` mapping is updated to hold the backer's Ethereum address and the amount of ether backed as a new key-value pair.

Listing 5.4: Crowdfunding Contract Source Code - Part 4

```
  let donate () =
    assert(msg_sender <> this_address);
    assert(msg_value >= 0);
    let bs = backers in
    let blk = block_number in
    let _max_block = max_block in
    if (blk > _max_block) then
      begin
        emit Message(_deadlinePassed_msg);
        assert(false) (* Revert: do not accept funds *)
      end
    else
      begin
        let backed_amount = backers[msg_sender] in
        if (backed_amount = 0) then
            backers[msg_sender] := msg_value
        else
          begin
            emit Message(_alreadyDonated_msg);
            assert(false) (* Revert: do not accept funds *)
          end
      end
```

### 5.1.5 Get-Funds Function

If the goal has been met and `_max_block` has passed, then the owner of the crowdfunding campaign can retrieve the funds. The `getFunds` function facilitates this, emitting appropriate error messages if the constraints are not met. Here we see `transferEth` in action, transferring the funds to the campaign owner. If the line just above it setting the `funded` flag to `true` were moved *below* `transferEth` the contract would no longer compile properly. This is because of the *Checks-Effects-Interactions Pattern*, as discussed in Section 4.1.

Listing 5.5: Crowdfunding Contract Source Code - Part 5

```
 let getFunds () =
   assert(msg_sender <> this_address);
   assert(msg_value = 0);
   let _owner = owner in
   if (msg_sender = _owner) then
     begin
       let blk = block_number in
       let bal = balance(this_address) in
       let _max_block = max_block in
       if (blk > _max_block) then
         let _goal = goal in
         if (_goal <= bal) then
           begin
             funded := true;
             (* Send all funds to owner *)
             emit Message(_funded_msg); (* Continued over page *)
```

```
              transferEth(_owner, bal)
            end
          else
            (* Funding campaign failed, do not send funds to owner *)
            emit Message(_failed_msg)
        else
          (* Too early to claim funds, do not send funds to owner. *)
          emit Message(_too_early_to_claim_funds_msg)
      end
    else
      emit Message(_only_owner_can_get_funds_msg)
```

### 5.1.6 Claim Function

If the crowdfunding campaign failed to reach the `_goal` by the block after `_max_block`, then the backers are entitled to reclaim their funds. Again, we see that `transferEth` is the last call made in that branch of the function, in order to follow the *Checks-Effects-Interactions Pattern*.

In the second 'if' statement, the condition `_goal <= bal` is necessary to prevent donors from reclaiming their funds in the period between the campaign meeting the conditions for success and the owner actually calling the `getFunds` function and setting the `_funded` flag to true.

Listing 5.6: Crowdfunding Contract Source Code - Part 6

```
    let claim() =
        assert(msg_sender <> this_address);
        assert(msg_value = 0);
        let blk = block_number in
        let _max_block = max_block in
        if (blk <= _max_block) then
          (* Too early to reclaim, do not send funds back. *)
          emit Message(_too_early_to_reclaim_msg)
        else
          let bal = balance(this_address) in
          let backed_amount = backers[msg_sender] in
          let _funded = funded in
          let _goal = goal in
          if (backed_amount = 0 \/ _funded \/ _goal <= bal) then
            (* Didn't back or campaign was successful so can't refund. *)
            emit Message(_cannot_refund_msg)
          else
            begin
              (* Can refund, send funds back *)
              backers[msg_sender] := 0;
              emit Message(_here_is_your_money_msg);
              transferEth(msg_sender, backed_amount)
            end
}
```

### 5.1.7 Layer Definition

Finally, we must provide a layer definition that represents the entire smart contract. Since we are just dealing with a single layer smart contract, this is simply the syntax required to declare that single layer in DeepSEA.

Listing 5.7: Crowdfunding Contract Source Code - Part 7

```
layer CONTRACT : [ { } ]  {crowdfunding : CrowdfundingSig}  = {
    crowdfunding = Crowdfunding
}
```

## 5.2 Initial State

The initial state of the smart contract is provided by the DeepSEA system. To generate the *init_state* we run the constructor provided by the DeepSEA system, based upon processing the '`.ds`' file. In our case, the goal and deadline are hard-coded into the smart contract and the constructor does not change anything so there is no difference between the *init_global_abstract_data* and the *init_state*.

The contract's initial state is incorporated into the snapshot of the blockchain as a part of the blockchain model, as discussed in Section 4.2.1.

In the presentation of the state/action transformer monad in Section 3.2.7 we avoided the need for *runStateT* by leveraging Coq's type system, however more standard approaches wrap the function contained by the state transformer monad in a record, giving the name *runStateT* to that field of the record. As a result, to 'run' our smart contract constructor, we need to use *runStateT* to access the function.

```
Definition init_state :=
  match runStateT (Crowdfunding_constructor_opt constructor_machine_env) init_global_abstract_data
  with
  | Some (_, d) ⇒ Some d
  | None ⇒ None
  end.
```

We can prove that the result of running the constructor gives the same result as *init_global_abstract_data* by the following simple proof.

```
Lemma same_init : init_state = Some init_global_abstract_data.
Proof.
  unfold init_state.
  vm_compute.
  unfold init_global_abstract_data.
  reflexivity.
Qed.
```

As a result, we will use *init_global_abstract_data* for the initial contents of the crowdfunding smart contract. We can see the contents of the storage variables by printing *init_global_abstract_data*.

```
Print init_global_abstract_data.
```

```
init_global_abstract_data =
{|
  Outgoing_transfer_recipient_and_amount := None;
  Crowdfunding_owner :=
    Int256.repr
      890025619261977197475349953386626580431692757011;
  Crowdfunding_max_block := Int256.repr 140;
  Crowdfunding_goal := 50;
  Crowdfunding_backers := empty Z32;
  Crowdfunding_funded := false
|}
     : global_abstract_data_type
```

Having defined the initial state of the smart contract we can now define the record of the blockchain state (that is relevant to the smart contract's correctness). This is a record holding:

- the timestamp;
- block number;
- the balances of all accounts;
- the blockhash of all blocks;
- and the contract's storage variables.

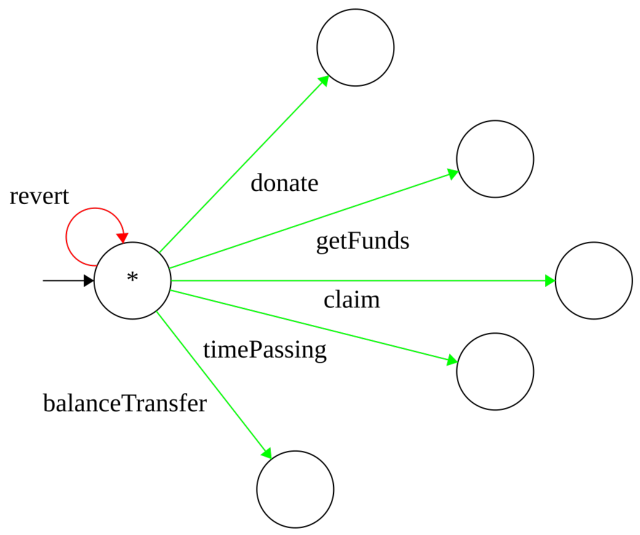
Here is the definition of the initial blockchain state. The snapshot variables are introduced as arbitrary values using the `Context` keyword in Coq with the aim of being used to show that theorems proved using those variables apply to all blockchain states, similarly to the principle of universal introduction.

```
Definition initial_state :=
  mkBlockchainState
    snapshot_timestamp
    snapshot_number
    snapshot_balances
    snapshot_blockhash
    init_global_abstract_data.
```

## 5.3 Step Function

The step function is defined using a successful-calls approach as described in Section 4.2.2, where Listings 4.2.5 and 4.2.6 outline how the step function is defined for the crowdfunding smart contract. Figure 5.1 outlines the possible transitions of the step function. There are the three transitions for each of the smart contract functions. These only consider transitions where the call succeeds, with the revert transition capturing

Figure 5.1: Outline of the step function for the crowdfunding smart contract



the case when the calls fail. Similarly, the generic `timePassing` and `balanceTransfer` transitions are also restricted to those which succeed, with `revert` covering the failure cases. These last two transitions describe generic actions which can occur on the blockchain, rather than specific smart contract functions. The transition `timePassing` is self-explanatory, and `balanceTransfer` describes transfers between addresses except transfers originating from the smart contract's address.

Here is the type of the step function, defined as described in Section 4.2.2. This states that `step` takes two arguments, the first named `before` of type `BlockchainState` and the second of type `Action before` which is an action on the state `before` that results in a successful call (or is the action `revert`). The `step` function returns a single value, the resulting `BlockchainState`.

```
Check step.
```

```
step
     : forall before : BlockchainState,
       Action before -> BlockchainState
```

## 5.4 The Reachability Predicate

Of central importance to all the proofs is the definition of what states are reachable. Here we define a predicate describing which states are reachable from a starting state as well as the list of intermediary states linking the initial and final states. This definition needs to handle the successful-calls only approach correctly, in particular with the type of actions being a dependent type that depend on a particular blockchain state.

First we define a dependent record, *Step*, which encapsulates a particular blockchain state with a particular action that is valid for that blockchain state. The type of the field *Step_action* depends on the value of the first field, *Step_state*. This will allow us to define the notion of a list of steps. The advantage of taking this approach is that this list of steps will only include successful calls and explicit calls to revert. The dependency on blockchain state arises from the fact that, for example, in order to know if a smart contract function executes successfully, the prior smart contract state is required (which is part of the blockchain state). This approach has the benefit of resulting in simpler proofs because failure cases are all handled by the single revert case rather than needing to be dealt with at various points throughout the proofs as discussed in Section 4.2.2.

```
Record Step := mkStep
  {
    Step_state : BlockchainState;
    Step_action : Action Step_state
  }.
```

As helper functions, we define *stepOnce* and *stepOnceAndWrap* which are thin wrappers around the *step* function defined earlier. The wrappers are made for use with the *Step* dependent record. The function *stepOnce* unpacks the *prev* dependent record of type *Step* into the blockchain state and the action, then returns the resulting blockchain state from executing this action on the blockchain state from *prev*. The function *stepOnceAndWrap* executes *stepOnce* on the argument *prev* and additionally wraps the resulting blockchain state into a dependent record with the provided *next_action*.

```
Definition stepOnce prev := (step (Step_state prev) (Step_action prev)).
Definition stepOnceAndWrap prev next_action := (mkStep (stepOnce prev) next_action).
```

Now we can define the reachability predicate as an inductive predicate. The reachability predicate describes what is reachable between any two states, not only what is reachable from the initial state.

The *initial_case* declares that the first state, *from*, is reachable from itself. This does require a valid next action to be able to be provided, but the `revert` action would always suffice here. The additional requirement is that there are no unprocessed outgoing transfers provided with the first blockchain state, these should be processed and incorporated into the `balances` variable before use with *ReachableFromBy*.

The *step_case* requires that *ReachableFromBy* holds for existing arguments and that there is a valid *next_action* based on the state *after* stepping once (there will always at least be `revert`). If those conditions hold, then the blockchain state resulting from stepping once from the action encapsulated in the step *prev*, is reachable from the original *from* state by the previous list of actions prepended with the next blockchain state (and a valid *next_action*). In other words, we extend the chain of reachable states by one by applying the action contained in *prev* and allow any valid next action to be set up for use in calculating the following blockchain state.

```
Inductive ReachableFromBy from : BlockchainState → Step → list Step → Prop :=
| initial_case
    (Hno_leftover_outgoings : Outgoing_transfer_recipient_and_amount (contract_state from) = None)
    (next_action : Action from)
    : ReachableFromBy from from (mkStep from next_action) [mkStep from next_action]
| step_case
    (prevSt : BlockchainState) (prev : Step) (prevList : list Step)
    (Hprev : ReachableFromBy from prevSt prev prevList)
    (next_action : Action (stepOnce prev))
    : ReachableFromBy from (stepOnce prev)
      (stepOnceAndWrap prev next_action)
      (stepOnceAndWrap prev next_action :: prevList).
```

## 5.5 Remark on the Correctness of Specifications and the Blockchain Model

As mentioned in Sections 4.2 and 4.2.3, the high-level blockchain model is currently part of the trusted computing base. This includes the step function and reachability predicate. We can increase our trust by carefully examining these definitions which are used in defining the specifications of smart contracts. Future work could also involve extending the refinement proof to relate to smart contracts as a whole and not just individual functions. This would remove the need to trust the high-level definitions of the step function and reachability predicate.

Related to this is the question of the correctness of the specifications in the following two sections and in the second case study (Chapter 6). These specifications are the direct translation of the informal specifications and any technique to verify their correctness would bring up the question of the correctness of that technique. It is important to read and examine the properties carefully to ensure they fully correspond to the informal specifications for each property. One could also prove some properties about the specifications to strengthen trust, but these would necessarily only relate to part of the meaning of a specification and therefore be somewhat limited. Reading and examining the properties carefully is likely to be the best way to ensure that the formal specifications fully match the informal specification that the reader has in their mind. This is also why it is crucial that these sorts of formal specifications are high-level specifications in an expressive specification language, so that the correspondence between the formal and informal specifications is most clear and there is only a small leap of understanding.

## 5.6 Preservation of Donation Record Theorem

The first theorem we will state and prove establishes that the donation record is preserved over time. This temporal property makes use of the list of steps declared as part of the chain of reachable states.

First, we define the predicate *since_as_long* which captures the idea that some property $Q$ holds since the property $P$ held, as long as the property $R$ held for all intermediary steps. This is straightforward to define, making use of the predicate *ReachableFromBy*. The use of *ReachableFromBy* which uses the *step* function specifically for the crowdfunding smart contract links this predicate to the crowdfunding smart contract.

```
Definition since_as_long (P : BlockchainState → Prop) (Q : BlockchainState → Prop) (R : Step → Prop) :
Prop :=
  ∀ (steps : list Step) (from_state to_state : BlockchainState) (to_step : Step),
    ReachableFromBy from_state to_state to_step steps →
    P from_state →
    (∀ sa, List.In sa steps → R sa) →
    Q to_state.
```

This predicate bears some similarity to, though is different from, necessity specification predicates described by Mackay et al. [147] of the form `from` $A_1$ `to/next` $A_2$ `onlyIf/onlyThrough` $A$. A key difference is that the state mentioned in the conclusion of the `since_as_long` predicate is the final state in the sequence, whereas in necessity specifications the conclusion is an assertion about an intermediate state. The predicates serve different purposes, but in future work predicates corresponding to those for necessity specifications could also be added to DeepSEA as needed.

Next we define some notation which enables us to phrase statements using *since_as_long* more naturally.

```
Notation "Q `since` P `as-long-as` R" := (since_as_long P Q R) (at level 1).
```

Now we define the property which will be used as property $P$ as well as property $Q$. This is the notion that a donation has been recorded. The intention is to capture the idea that a donation remains recorded ever since the donation was first recorded. As a result the property *donation_recorded* is used for both $P$ and $Q$.

```
Definition donation_recorded (a : addr) (amount : Z) (s : BlockchainState) : Prop :=
  Int256Tree.get_default 0 a (Crowdfunding_backers (contract_state s)) = amount ∧ amount > 0.
```

We do not want to require that the donation must be recorded after the donor has made a call to reclaim their funds. As a result, we define property $R$, which must hold for all intermediary steps, to be that the action of each intermediary step cannot be a call from the donor's Ethereum address to the *claim* function of the crowdfunding smart contract.

```
Definition no_claims_from (a : addr) (s : Step) : Prop :=
  match Step_action s with
  | (call_Crowdfunding_claim context _ _ _ _ _ _) ⇒ a ≠ caller context
  | _ ⇒ True
  end.
```

Now we can state the theorem as a whole, ensuring that the Ethereum address and amount is consistent throughout the use of *since_as_long* so that we are defining the preservation of the donation for the same donor with a consistent amount. We can now begin to prove this theorem. It is important to notice the syntax of the bullets: `-`, `+`, `*` and `--` which indicate that a new branch of the proof is being proved.

```
Theorem donation_preserved :
  forall (a : addr) (d : Z),
                 (donation_recorded a d)
    `since`      (donation_recorded a d)
    `as-long-as` (no_claims_from a).
Proof.
```

First, we unfold the definition of the *since_as_long* predicate, giving the following proof goal with the shown context. Parts of the context will be hidden later in the proof, for brevity.

```
unfold since_as_long.
```

```
snapshot_timestamp, snapshot_number: int256
snapshot_blockhash: int256 -> int256
snapshot_balances: addr -> wei
snapshot_balances_valid_prf: forall a : addr,
                             0 <=
                             snapshot_balances a
                             <
                             Int256.modulus
contract_address: addr
address_accepts_funds: option ContractState ->
                       addr -> addr -> wei -> bool
HmemOps, memModelOps: MemoryModelOps mem
______________________________________________________
forall (a : addr)
  (d : Z) (steps : list Step)
  (from_state to_state : BlockchainState)
  (to_step : Step),
ReachableFromBy from_state to_state to_step steps ->
donation_recorded a d from_state ->
(forall sa : Step, In sa steps -> no_claims_from a sa) ->
donation_recorded a d to_state
```

```
intros.
```

```
H: ReachableFromBy from_state to_state to_step steps
H0: donation_recorded a d from_state
H1: forall sa : Step,
   In sa steps -> no_claims_from a sa
______________________________________________________
donation_recorded a d to_state
```

After introducing the universally quantified variables we can then proceed by induction, making use of the hypothesis $H$ which is an inductive predicate.

```
induction H.
```

-

```
H0: donation_recorded a d from_state
______________________________________________________
donation_recorded a d from_state
```

In the *initial_case*, the variable *to_state* has been identified with *from_state* since this branch of the definition of *ReachableFromBy* is of the form (*ReachableFromBy from from ...*) as described in Section 5.4. As a result, the hypothesis *H0* is exactly our goal. It is from the assumption in *since_as_long* associated with the predicate $P$, that the donation was recorded in the *from_state*. This allows us to dismiss the current goal with the tactic `assumption`.

```
assumption.
```

-

```
H: ReachableFromBy from_state prevSt prev prevList
H0: donation_recorded a d from_state
H1: forall sa : Step,
    In sa
    (stepOnceAndWrap prev next_action :: prevList) ->
    no_claims_from a sa
IHReachableFromBy: (forall sa : Step,
                   In sa prevList ->
                   no_claims_from a sa) ->
                   donation_recorded a d prevSt
-----------------------------------------------------------------
donation_recorded a d (stepOnce prev)
```

Now, in the inductive case we have the additional knowledge described by *IHReachableFromBy*. Using this, plus *H1*, we can establish that the donation is recorded in *prevSt*. We establish this fact as *H2*.

```
assert(donation_recorded a d prevSt) by (apply IHReachableFromBy;
intros; apply H1; apply in_cons; assumption).
```

```
H2: donation_recorded a d prevSt
-----------------------------------------------------------------
donation_recorded a d (stepOnce prev)
```

Next, we need to engage with the meaning of *donation_recorded* and so we unfold its definition and split the conjunction in the hypothesis, which becomes *H0* and *H2*.

```
unfold donation_recorded in *; destruct_and.
```

```
H0: get_default 0 a
    (Crowdfunding_backers (contract_state prevSt)) =
    d
H2: d > 0
-----------------------------------------------------------------
get_default 0 a
  (Crowdfunding_backers
     (contract_state (stepOnce prev))) = d /\
d > 0
```

We are trying to prove a conjunction here. *H2*, the knowledge that the donation *d* is greater than zero, is already known. As a result we can split our goal into two and immediately solve the right hand branch via the tactics: `split`; [|`assumption`]. The location of `assumption` being to the right of the vertical bar indicates that we will dismiss the right hand branch of our split goal with the tactic `assumption`.

```
split; [|assumption].
```

```
______________________________________________________________
get_default 0 a
  (Crowdfunding_backers
     (contract_state (stepOnce prev))) = d
```

Now our goal is the left-hand branch only. The proof will soon involve considering each of the functions of the crowdfunding smart contract. To exclude the `claim` function from consideration, we need to establish that the action of the *prev* step cannot be `claim`. This is done by making use of *H1* and showing that *prev* is an element of *prevList*. This is facilitated by a helper lemma called *Hlinks* which establishes the relevant correspondences between variables in *ReachableFromBy*.

```
Hlinks.
```

```
HS: prevSt = Step_state prev
HL: exists tl : list Step, prev :: tl = prevList
______________________________________________________________
get_default 0 a
  (Crowdfunding_backers
     (contract_state (stepOnce prev))) = d
```

Given this knowledge, we can establish that there were *no_claims_from* the donor's address in the *prev* step's action.

```
assert (no_claims_from a prev) by
(apply H1; destruct HL; subst; right; left; auto).
```

```
H3: no_claims_from a prev
______________________________________________________________
get_default 0 a
  (Crowdfunding_backers
     (contract_state (stepOnce prev))) = d
```

The next tactics produce long text output in their intermediary steps, so their output has been omitted for brevity. The most important step is `destruct` *Step_action0* which is performing case analysis on the term of type *Action prevSt* which splits the goal into six subgoals corresponding to the transitions shown in Figure 5.1, which are mirrored by the six + bullets in this proof.

```
destruct prev; autounfold in *; simpl in *.
unfold no_claims_from in H3.
unfold donation_recorded in *. destruct Step_action0; simpl in *;
rewrite <- HS in *.
```

We now move on to the branches for each transition.

```
+
```

Now we begin to engage with the code of actual smart contract functions that have been translated into Coq. First, we declare the functions as `Transparent` so that they can be unfolded. By default they are `Opaque` so that Coq does not unfold them prematurely.

```
    Transparent Crowdfunding_donate_opt.
```

```
case_donate_prf: runStateT
                 (Crowdfunding_donate_opt
                 (make_machine_env
                 contract_address Step_state0
                 context
                 address_accepts_funds_assumption
                 callvalue_bounded_prf
                 balances_bounded_prf
                 callvalue_prf))
                 (contract_state Step_state0) =
                 Some (r, contract_state_after)
-------------------------------------------------------------
get_default 0 a
  (Crowdfunding_backers contract_state_after) = d
```

The `donate` smart contract function has been translated into *Crowdfunding_donate_opt* which, due to the successful calls approach, returns an optional type holding (*Some* (*r*, *contract_state_after*)). The *None* case is taken care of by the *revert* case, which is the final + bullet in this proof.

*Crowdfunding_donate_opt* takes as a parameter the machine environment and the prior contract state. If the smart contract function had any arguments, these would also be provided to *Crowdfunding_donate_opt* at this point.

To make the body of *Crowdfunding_donate_opt* easier to read, we will rename the *make_machine_env* command which is creating a record from the arguments provided to it.

```
    remember (make_machine_env contract_address Step_state0 context
                         address_accepts_funds_assumption
                         callvalue_bounded_prf balances_bounded_prf
                         callvalue_prf) as machine_environment.
```

```
case_donate_prf: runStateT
                 (Crowdfunding_donate_opt
                 machine_environment)
                 (contract_state Step_state0) =
                 Some (r, contract_state_after)
-------------------------------------------------------------
get_default 0 a
  (Crowdfunding_backers contract_state_after) = d
```

Now we can unfold *Crowdfunding_donate_opt* and see the body of the crowdfunding donate function, translated into Coq.

```
    unfold Crowdfunding_donate_opt in *.
```

```
case_donate_prf
     : runStateT
         ((v <-
           ret
             (negb
                (me_caller machine_environment =?
                 me_address machine_environment));;
           guard v);;
          (v <-
           ret
             (me_callvalue machine_environment >=? 0);;
           guard v);;
          gets Crowdfunding_backers;;
          spec2 <-
          ret (me_number machine_environment);;
          spec3 <-
          gets Crowdfunding_max_block;;
          (if Int256.ltu spec3 spec2
           then ret tt;; v <- ret false;; guard v
           else
            spec4 <-
            gets
              (fun s : GetHighData =>
               get_default 0
                 (me_caller machine_environment)
                 (Crowdfunding_backers s));;
            (if (spec4 =? 0)%Z
             then
              MonadState.modify
                (fun s : GetHighData =>
                 update_Crowdfunding_backers
                 (set
                 (me_caller machine_environment)
                 (me_callvalue machine_environment)
                 (Crowdfunding_backers s)) s)
             else ret tt;; v <- ret false;; guard v)))
         (contract_state Step_state0) =
       Some (r, contract_state_after)
```

```
get_default 0 a
  (Crowdfunding_backers contract_state_after) = d
```

With some understanding of monads as discussed in Section 3.2, we can see a reasonably close correspondence between the content of the unfolded *Crowdfunding_donate_opt* and the donate function introduced in Listing 5.4. The deployed smart contract is proved by DeepSEA to behave as the Coq version behaves. The Coq representation is the version the proofs use, so we can be confident that the theorems we prove about the Coq version hold for the deployed smart contract even if we cannot see the correspondence between the Coq version and the smart contract as in Listing 5.4. Nevertheless, it is reassuring to see at least a superficial similarity.

For example, the code `assert(false)` in two of the branches has been translated into:

*ret tt*;; *v* ← *ret false*;; *guard v*.

Next, we make use of the `inversion` tactic, incorporated into the *ds_inv* tactic. This allows us to reason from the fact that *Crowdfunding_donate_opt* (now seen as the application of *runStateT* to its body) succeeds, returning (*Some* (*r*, *contract_state_after*)).

The *ds_inv* tactic makes use of DeepSEA's *inv_runStateT_branching* tactic, which automatically calls the `destruct` tactic to perform case analysis on the guards of 'if' statements in the function body. This results in multiple goals, one for each of the possible branches of the program. Since there are two nested 'if' statements, this gives three subgoals.

```
ds_inv.
*
```

```
H1: contract_state Step_state0 = m0
H4: a1 = true
H5: m0 = m
H7: m = m2
H18: false = a8
H20: a8 = true
____________________________________________
get_default 0 a
  (Crowdfunding_backers contract_state_after) = d
```

*ds_inv* generates various equalities based upon the use of the `inversion` tactic. Only some are shown here, for brevity. Its use is usually coupled with the `subst` tactic which performs substitutions based on equalities.

```
subst.
```

```
H20: false = true
____________________________________________
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0)) =
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0))
```

This branch results in a revert due to the `assert(false)` statement. As a result it does not alter the `backers` mapping, which makes the goal provable by `reflexivity`. In addition, the `assert(false)` statement contradicts that the function succeeds (in line with the successful-calls approach), which makes this branch impossible as is clear from *H20*.

This situation means that there are two ways we can solve the goal from here, either by `reflexivity` or by `inversion` *H20* making use of the impossibility of the situation. Here we will use `inversion` *H20*.

```
inversion H20.
```

We now move on to the next branch.

```
* subst. simpl in *.
```

This branch corresponds to the case when a new backer is added to the `backers` mapping and is the heart of this proof. At the beginning of this theorem we quantified over all addresses. To make progress in the proof we need to split between the cases of whether the address we quantified over, *a*, is the current caller or not.

```
destruct (a =? (caller context)) eqn:Case.
--
```

```
H2: get_default 0 a
   (Crowdfunding_backers
   (contract_state Step_state0)) > 0
Case: (a =? caller context) = true
-------------------------------------------
get_default 0 a
  (set (caller context)
     (callvalue context)
     (Crowdfunding_backers
        (contract_state Step_state0))) =
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0))
```

First, the case where the caller is the address *a*. Recall that we have shown based on the inductive hypothesis that the address *a* already has its donation recorded. In particular, this means that *H2* holds which states that the value for *a* in the `backers` mapping is positive. This will lead to a contradiction, because this branch succeeding (as assumed by the successful-calls approach) requires the backer to not yet have donated which implies that the value for *a* in the `backers` mapping is zero.

To make things clearer, we will invoke ex falso quodlibet and replace our goal with *False*, which is logical contradiction in Coq and is impossible to prove except from contradictory hypotheses.

```
exfalso.
```

```
H2: get_default 0 a
   (Crowdfunding_backers
   (contract_state Step_state0)) > 0
Heqb0: (get_default 0 (caller context)
      (Crowdfunding_backers
      (contract_state Step_state0)) =? 0)%Z =
      true
-------------------------------------------
False
```

To show this contradiction we first need to show *Case* is true as a proposition rather than simply as a Boolean equality, which we can use the lemma *Int256eq_true* for.

```
apply Int256eq_true in Case.
```

```
H2: get_default 0 a
    (Crowdfunding_backers
    (contract_state Step_state0)) > 0
Heqb0: (get_default 0 (caller context)
       (Crowdfunding_backers
       (contract_state Step_state0)) =? 0)%Z =
       true
Case: a = caller context
________________________________________________
False
```

Now we can rewrite the occurrences of *caller* `context` to *a* throughout the proof state.

```
rewrite <- Case in *.
```

```
H2: get_default 0 a
    (Crowdfunding_backers
    (contract_state Step_state0)) > 0
Heqb0: (get_default 0 a
       (Crowdfunding_backers
       (contract_state Step_state0)) =? 0)%Z =
       true
________________________________________________
False
```

We can see we almost have a contradiction between *H2* and *Heqb0*.

We need a similar lemma to *Int256eq_true* to relate the Boolean equality with the *Z* type to propositional equality. This is provided by the *Z.eqb_eq* lemma whose statement is shown here.

```
Check Z.eqb_eq.
```

```
Z.eqb_eq
     : forall n m : Z, (n =? m)%Z = true <-> n = m
```

```
apply Z.eqb_eq in Heqb0.
```

```
Heqb0: get_default 0 a
       (Crowdfunding_backers
       (contract_state Step_state0)) = 0
________________________________________________
False
```

Now we can rewrite *H2* to arrive at a clearly contradictory hypothesis.

```
rewrite Heqb0 in H2.
```

```
H2: 0 > 0
________________________________________________
False
```

To dismiss our impossible goal based on the impossible hypothesis, we make use of the linear integer arithmetic tactic *lia* to discharge the goal.

```
        lia.
```

Having proved this, we now move onto the second subgoal of our case analysis.

```
      --
```

```
Case: (a =? caller context) = false
-----------------------------------
get_default 0 a
  (set (caller context)
     (callvalue context)
     (Crowdfunding_backers
        (contract_state Step_state0))) =
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0))
```

We also need to consider the case when the caller is not the address *a*. The goal here is to show that even though the `backers` mapping is altered, the change does not affect the address *a* and so retrieving the amount donated for the address *a* has not changed since the previous step. Note that the syntax `<>` means inequality in Coq.

```
        apply Int256eq_false in Case.
```

```
Case: a <> caller context
-----------------------------------
get_default 0 a
  (set (caller context)
     (callvalue context)
     (Crowdfunding_backers
        (contract_state Step_state0))) =
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0))
```

We need to make use of a lemma about the mapping function *get_default* here. Here is the statement of that lemma. The *so* stands for "set-other" indicating that the key being retrieved is different to the one that has just been set.

```
        Check get_default_so.
```

```
get_default_so
     : forall (A : Type)
         (def : A)
         (k k0 : elt)
         (v : A) (m : t A),
       k <> k0 ->
       get_default def k (set k0 v m) =
       get_default def k m
```

```
        apply get_default_so.
```

```
Case: a <> caller context
-------------------------
a <> caller context
```

Now the goal is trivially provable by showing that $a \neq caller$ `context` as required to fully apply *get_default_so*.

```
        apply Case.
```

Having now proved that branch, we move on to the next branch related to the `donate` function.

```
    * subst.
```

```
H22: false = true
-------------------------
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0)) =
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0))
```

Similarly to the first `*` bullet, there are two ways we can solve this goal as we are in a similar situation. This time we will use the `reflexivity` tactic to solve the goal, even though it is perhaps more precise to use `inversion` *H22* which corresponds with the reality that this branch never succeeds.

```
      reflexivity.
```

Now that we have proved the subgoals relating to the `donate` function, we move on to the `getFunds` function.

```
  + Transparent Crowdfunding_getFunds_opt.
```

The branch for the `getFunds` smart contract function is trivial to prove because no branch of its code alters the `backers` mapping. As a result, after applying *ds_inv* to perform case analysis on all the branches, every branch can be solved by the `reflexivity` tactic. Of interest, however, is the translation into Coq of the `getFunds` function, which is shown here.

```
    unfold Crowdfunding_getFunds_opt in *.
```

```
case_getFunds_prf
      : runStateT
           ((v <-
             ret
               (negb
                   (me_caller machine_environment =?
                    me_address machine_environment));;
             guard v);;
            (v <-
             ret
               (me_callvalue machine_environment =? 0)%Z;;
             guard v);;
            spec1 <-
            gets Crowdfunding_owner;;
            (if me_caller machine_environment =? spec1
             then
              spec2 <-
              ret (me_number machine_environment);;
              spec3 <-
              ret
                (me_balance machine_environment
                    (me_address machine_environment));;
              spec4 <-
              gets Crowdfunding_max_block;;
              (if Int256.ltu spec4 spec2
               then
                spec5 <-
                gets Crowdfunding_goal;;
                (if spec5 <=? spec3
                 then
                  MonadState.modify
                    (update_Crowdfunding_funded true);;
                  ret tt;;
                  d <- MonadState.get;;
                  (let
                    (success, d') :=
                    me_transfer machine_environment spec1
                    spec3 d in
                    if success =? Int256.one
                    then put d'
                    else mzero)
                 else ret tt)
               else ret tt)
             else ret tt))
           (contract_state Step_state0) =
        Some (r, contract_state_after)
```

```
______________________________________________________________________________
get_default 0 a
  (Crowdfunding_backers contract_state_after) = d
```

Here we solve all the subgoals by `reflexivity`. The semicolon syntax applies all the tactics following each semicolon to all of the generated subgoals.

```
    ds_inv; subst; reflexivity.
```

Having dealt with the `getFunds` function, we now move on to the `claim` function.

```
  +
case_claim_prf: runStateT
                (Crowdfunding_claim_opt
                (make_machine_env
                contract_address Step_state0
                context
                address_accepts_funds_assumption
                callvalue_bounded_prf
                balances_bounded_prf
                callvalue_prf))
                (contract_state Step_state0) =
                Some (r, contract_state_after)
H3: a <> caller context
________________________________________
get_default 0 a
  (Crowdfunding_backers contract_state_after) = d
```

This branch corresponds to the `claim` function. As part of using `since_as_long`, we assumed that there were *no_claims_from* the address, *a*, and so this branch of the proof proceeds by unfolding the `claim` function and observing that any change to the `backers` mapping does not affect the address *a*.

```
    Transparent Crowdfunding_claim_opt.
    unfold Crowdfunding_claim_opt in case_claim_prf.
```

As before, we use the tactic *ds_inv* to explore all branches of the `claim` function. Here, two of the branches are dismissed by the tactic `reflexivity` because they do not alter the `backers` mapping, leaving us with two goals.

```
    ds_inv; subst; simpl in *; try reflexivity.
    *
H3: a <> caller context
________________________________________
get_default 0 a
  (set (caller context) 0
     (Crowdfunding_backers
        (contract_state Step_state0))) =
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0))
```

We have relevant information in the hypothesis *H3* which originates from the use of *no_claims_from* excluding the address, *a*, from being the caller. As before, we can use the lemma *get_default_so* to effectively remove the `set` from the goal, because it affects an address other than *a*. Here is the lemma *get_default_so* again.

```
Check get_default_so.
```

```
get_default_so
     : forall (A : Type)
         (def : A)
         (k k0 : elt)
         (v : A) (m : t A),
       k <> k0 ->
       get_default def k (set k0 v m) =
       get_default def k m
```

We rewrite the goal and dismiss the subgoal which is $a \neq$ *caller* `context` by the assumption tactic which applies *H3*.

```
rewrite get_default_so by assumption.
```

```
________________________________________________
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0)) =
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0))
```

Now the goal is clearly true and can be solved by the `reflexivity` tactic.

```
reflexivity.
```

We now move on to the next subgoal.

```
*
```

```
Heqb1: (Int256.zero =? Int256.one) = true
________________________________________________
get_default 0 a
  (set (caller context) 0
     (Crowdfunding_backers
        (contract_state Step_state0))) =
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0))
```

This branch corresponds to the case when the transfer fails, returning *Int256.zero*. However, with the successful-calls approach, we have assumed that the call succeeds which requires *transferEth* to return *Int256.one*. As a result we have the hypothesis *Heqb1*, from which we can easily derive a contradiction after rewriting the Boolean equation into its propositional equivalent.

```
apply Int256eq_true in Heqb1.
```

```
Heqb1: Int256.zero = Int256.one
______________________________________
get_default 0 a
  (set (caller context) 0
     (Crowdfunding_backers
        (contract_state Step_state0))) =
get_default 0 a
  (Crowdfunding_backers (contract_state Step_state0))
```

Here, the tactic `discriminate` makes use of that the fact that different constructors of the same inductive type cannot be equal to solve the goal from the hypothesis *Heqb1*.

```
      discriminate.
```

The final three cases, which are all dismissed by the tactic `assumption`, correspond to the transitions *externalBalanceTransfer*, *timePassing*, and *revert*, respectively. These functions do not alter the donation record and as a result the fact established based on the inductive hypothesis that the donation record of *prevSt* holds the correct value is sufficient to prove these cases.

```
  +
H0: get_default 0 a
    (Crowdfunding_backers (contract_state prevSt)) =
    d
______________________________________
get_default 0 a
  (Crowdfunding_backers (contract_state prevSt)) = d

    assumption.
  +
H0: get_default 0 a
    (Crowdfunding_backers (contract_state prevSt)) =
    d
______________________________________
get_default 0 a
  (Crowdfunding_backers (contract_state prevSt)) = d

    assumption.
  +
H0: get_default 0 a
    (Crowdfunding_backers (contract_state prevSt)) =
    d
______________________________________
get_default 0 a
  (Crowdfunding_backers (contract_state prevSt)) = d

    assumption.

Qed.
```

The detailed discussion of the proof above demonstrates how an interactive proof assistant can facilitate communicating a proof as well as ensuring a high level of rigour. Any details glossed over in the natural language description of the proof are directly available to be unpacked in the codebase and all details of the formal proof are checked by Coq. This gives a high degree of confidence that the proof is correct. Of course, what has been proved correct is the phrasing in Coq of the theorem and the statement of the theorem needs to be carefully evaluated to see if it adequately matches the intended specification.

## 5.7 Safety Property of Sufficient Balance Theorem

The next property is a safety property, that at all reachable states the smart contract has a balance that is sufficient to repay all the donors in the donation record (the `backers` mapping).

Here is the generic safety property predicate:

`Definition` $\mathit{Safe}\ P :=$
$\quad\forall\ \mathit{state}\ s\ l,\ \mathit{ReachableFromBy}\ \mathit{initial\_state}\ \mathit{state}\ s\ l \rightarrow P\ \mathit{state}.$

The next property, which we call the balance being backed, is the specific safety property we are aiming to show holds for all reachable states, it is an invariant. It states that the smart contract has a sufficient balance to repay all the donors at a given state. It requires the *funded* flag to be *false*, to indicate that the crowdfunding campaign has not succeeded with the owner withdrawing all the funds.

`Definition` $\mathit{balance\_backed}\ \mathit{state} :=$
$\quad(\mathit{Crowdfunding\_funded}\ (\mathit{contract\_state}\ \mathit{state})) = \mathit{false}$
$\quad\rightarrow$
$\quad\mathit{sum}\ (\mathit{Crowdfunding\_backers}\ (\mathit{contract\_state}\ \mathit{state}))$
$\qquad\leq (\mathit{balance}\ \mathit{state}\ (\mathit{contract\_address})) \wedge$
$\qquad(\forall\ \mathit{key}\ \mathit{value},\ \mathit{get}\ \mathit{key}\ (\mathit{Crowdfunding\_backers}\ (\mathit{contract\_state}\ \mathit{state})) = \mathit{Some}\ \mathit{value} \rightarrow ((\mathit{value} \geq 0) \wedge$
$(\mathit{value} < \mathit{Int256.modulus}))).$

First, we will prove a lemma called *balance_backed_in_processed_state*. The function *next_blockchain_state* takes a newly computed *contract_state_after*, the call context and blockchain state prior to the contract call and it computes the blockchain state after the call. It enacts any transfer of ether to or from the smart contract and puts in place the new *contract_state* once the information about any transfer has been removed. Here is its definition:

```
Print next_blockchain_state.
```

```
next_blockchain_state =
fun (before : BlockchainState) (context : CallContext)
  (contract_state_after : ContractState) =>
{|
  timestamp := timestamp before;
  block_number := block_number before;
  balance :=
    new_balance_after_contract_call before context
      contract_state_after;
  blockhash := blockhash before;
  contract_state :=
    resetTransfers contract_state_after
|}
     : BlockchainState ->
       CallContext -> ContractState -> BlockchainState
```

The lemma *balance_backed_in_processed_state* states the following: if the balance is backed in the blockchain state after the smart contract function has run and the `callvalue` was zero and there were no transfers outgoing from the smart contract, then the balance is still backed after the (empty) transfers are processed.

```
Lemma balance_backed_in_processed_state :
  forall after context,
    balance_backed (Step_state after)
    -> Outgoing_transfer_recipient_and_amount (contract_state (Step_state after))
         = None
    -> (callvalue context =? 0) = true
    -> balance_backed
     (next_blockchain_state (Step_state after) context
     (contract_state (Step_state after))).
Proof.
```

This proof makes use of similar tactics to those previously discussed. A portion of the proof which makes use of a generic lemma is unpacked below.

```
intros.
unfold step. unfold balance_backed. simpl in *.
unfold balance_backed in H.
apply Z.eqb_eq in H1.
unfold new_balance_after_contract_call. unfold current_balances. simpl.
rewrite H1.
intros.
split.
- rewrite H0.
```

```
sum
  (Crowdfunding_backers
     (contract_state (Step_state after))) <=
update_balances (caller context) contract_address 0
  (balance (Step_state after)) contract_address
```

Here, there is a call to *update_balances* with zero as the amount being updated. Clearly, an update to the balances of zero leaves the balances unchanged. We make use of the generic lemma *addZeroBalance* discussed in Section 7.3. The proof of *addZeroBalance* is available in Appendix D.14.

```
   rewrite addZeroBalance.
```

```
H: Crowdfunding_funded
   (contract_state (Step_state after)) = false ->
   sum
   (Crowdfunding_backers
   (contract_state (Step_state after))) <=
   balance (Step_state after) contract_address /\
   (forall (key : elt) (value : Z32),
   get key
   (Crowdfunding_backers
   (contract_state (Step_state after))) =
   Some value ->
   value >= 0 /\ value < Int256.modulus)
H2: Crowdfunding_funded
    (contract_state (Step_state after)) = false
--------------------------------------------------------
sum
  (Crowdfunding_backers
     (contract_state (Step_state after))) <=
balance (Step_state after) contract_address
```

After rewriting, we can see our goal is simplified as expected. The goal is then proved by the application of *H* and *H2*. The hypothesis *H* has one antecedent and a consequent comprised of conjunctions, one of which matches the goal.

```
   apply H.
```

```
H2: Crowdfunding_funded
    (contract_state (Step_state after)) = false
--------------------------------------------------------
Crowdfunding_funded
   (contract_state (Step_state after)) = false
```

```
   apply H2.
 - apply H. apply H2.
```

```
Qed.
```

Now we are ready to prove the second of three properties describing a correct crowdfunding contract from a donor's perspective. For the sake of brevity only an outline of the proof will be discussed. The full proof is available in Appendix B.2 as well as being able to be interactively stepped through in the codebase, as described in Appendix A.

```
Theorem sufficient_funds_safe : Safe balance_backed.
Proof.
```

This proof proceeds by induction on *H*.

```
  unfold Safe.
  intros.
```

```
H: ReachableFromBy initial_state state s l
________________________________________
balance_backed state
```

```
  induction H.
  -
```

```
________________________________________
balance_backed initial_state
```

First, we must prove that the balance is backed in the *initial_state*. This branch of the proof is omitted for brevity.

```
  - Hlinks. repeat rewrite HS in *.
```

```
prev: Step
IHReachableFromBy: balance_backed (Step_state prev)
HS: prevSt = Step_state prev
________________________________________
balance_backed (stepOnce prev)
```

Now, we must prove that the balance is backed in the next state (*stepOnce prev*) from the inductive hypothesis that the balance is backed in the previous state *Step_state prev*. Recall that *Step_state* is the function used to access the blockchain-state field from a value of type *Step*, as opposed to *stepOnce* which progresses *prev* to the next state making use of the action in the step *prev*.

To do this, we will consider each of the six transitions as outlined in Figure 5.1, by performing case analysis on (*Step_action prev*).

```
    destruct (Step_action prev) eqn:Case.
    +
case_donate_prf
     : runStateT
         (Crowdfunding_donate_opt
            (make_machine_env contract_address
               (Step_state prev) context
               address_accepts_funds_assumption
               callvalue_bounded_prf
               balances_bounded_prf callvalue_prf))
         (contract_state (Step_state prev)) =
       Some (r, contract_state_after)
```

```
________________________________________________________________________________
balance_backed (stepOnce prev)
```

First we will consider the `donate` function. The proofs are available in Appendix B.2.

```
      Transparent Crowdfunding_donate_opt.
      ds_inv; subst; try discriminate.
```

After using the *ds_inv* tactic to consider all the branches and dismissing two of the branches with the `discriminate` tactic, we are left with one branch to prove for the `donate` function. The proof is omitted.

```
      *
```

```
________________________________________________________________________________
balance_backed (stepOnce prev)
```

Next we must prove the inductive case for the `getFunds` function. The proof is omitted.

```
    +
```

```
case_getFunds_prf
     : runStateT
         (Crowdfunding_getFunds_opt
            (make_machine_env contract_address
               (Step_state prev) context
               address_accepts_funds_assumption
               callvalue_bounded_prf
               balances_bounded_prf callvalue_prf))
         (contract_state (Step_state prev)) =
       Some (r, contract_state_after)
```

```
________________________________________________________________________________
balance_backed (stepOnce prev)
```

Finally, we prove the inductive case for the `claim` function. The proof is also omitted.

```
    +
case_claim_prf
     : runStateT
         (Crowdfunding_claim_opt
            (make_machine_env contract_address
               (Step_state prev) context
               address_accepts_funds_assumption
               callvalue_bounded_prf
               balances_bounded_prf callvalue_prf))
         (contract_state (Step_state prev)) =
       Some (r, contract_state_after)
──────────────────────────────────────────
balance_backed (stepOnce prev)
```

Next, we must prove that the balance is backed regardless of external balance transfers. Note that an external balance transfer in this model cannot originate from the contract's address, so external balance transfers can only increase a contract's balance. The proof is omitted.

```
             +
Case: Step_action prev =
     externalBalanceTransfer
     (Step_state prev) sender recipient amount prf
──────────────────────────────────────────
balance_backed (stepOnce prev)
```

The next case is for time passing, which does not affect the contract's balance or the backers mapping. As a result, the proof follows quickly from the inductive hypothesis after unfolding the relevant definitions.

```
           +
IHReachableFromBy: balance_backed (Step_state prev)
Case: Step_action prev =
     timePassing (Step_state prev)
     block_count
     time_passing prf
──────────────────────────────────────────
balance_backed (stepOnce prev)
```

```
            unfold stepOnce, step.
            rewrite Case.
            unfold balance_backed; simpl.
            unfold balance_backed in IHReachableFromBy.
            simpl in *.
```

```
IHReachableFromBy
     : Crowdfunding_funded
         (contract_state (Step_state prev)) = false ->
       sum
         (Crowdfunding_backers
            (contract_state (Step_state prev))) <=
       balance (Step_state prev) contract_address /\
       (forall (key : elt) (value : Z32),
        get key
          (Crowdfunding_backers
             (contract_state (Step_state prev))) =
        Some value ->
        value >= 0 /\ value < Int256.modulus)
```

```
______________________________________________________________
Crowdfunding_funded (contract_state (Step_state prev)) =
false ->
sum
  (Crowdfunding_backers
     (contract_state (Step_state prev))) <=
balance (Step_state prev) contract_address /\
(forall (key : elt) (value : Z32),
 get key
   (Crowdfunding_backers
      (contract_state (Step_state prev))) =
 Some value -> value >= 0 /\ value < Int256.modulus)
```

```
apply IHReachableFromBy.
```

Finally, we have the catch-all case for failures, the revert case. Since this results in the same case as the previous step, the goal clearly follows from the inductive hypothesis.

```
+
```

```
IHReachableFromBy: balance_backed (Step_state prev)
Case: Step_action prev = revert (Step_state prev)
______________________________________________________________
balance_backed (stepOnce prev)
```

```
unfold stepOnce, step.
rewrite Case.
```

```
IHReachableFromBy: balance_backed (Step_state prev)
______________________________________________________________
balance_backed (Step_state prev)
```

```
apply IHReachableFromBy.
```

That completes the proof of the safety property, showing that the donation record is backed by the smart contract's balance in all reachable states.

```
Qed.
```

## 5.8 Backers Can Retrieve Their Donation Theorem

The final crowdfunding theorem is a kind of liveness property. It states that, given certain conditions, if a backer is recorded as having donated and the campaign failed, then there is a way for the backer to make a call to one of the smart contract's functions that results in a transfer from the smart contract to themselves of the amount they backed. In other words, backers can successfully retrieve their donation if the campaign fails.

First we will discuss each of the conditions assumed to hold to prove this theorem.

- $ReachableFromBy\ initial\_state\ state\ s\ l$. It is assumed that the blockchain state $state$ (or $Step\ s$) is reachable from the $initial\_state$ via the intermediary steps $l$.
- $backed\_amount = Int256Tree.get\_default\ 0\ backer\_addr\ (Crowdfunding\_backers\ (contract\_state\ state))$. We define $backed\_amount$ to be the value of $backer\_addr$ in the $backers$ mapping.
- $contract\_address \neq backer\_addr$. We assume the backer is not the contract itself.
- $backed\_amount > 0$. The meaning of this assumption is clear.
- $(balance\ state\ backer\_addr + backed\_amount <?\ Int256.modulus) = true$. Here we assume that the backer is not so rich that receiving their refund would cause their balance to overflow.
- $Crowdfunding\_funded\ (contract\_state\ state) = false$. We also assume that the $funded$ flag indicating whether the campaign is successful is $false$.
- $balance\ state\ contract\_address < (Crowdfunding\_goal\ (contract\_state\ state))$. We assume the contract's balance is small and has not reached the goal.
- $(\forall\ a : addr, 0 \leq balance\ state\ a < Int256.modulus)$. We also assume the balances start in a legitimate range.
- $Int256.ltu\ (Crowdfunding\_max\_block\ (contract\_state\ state))\ (block\_number\ state) = true$. We assume the time limit has passed.

Based on these assumptions, we aim to prove that there exists an action which could reasonably be called by the backer to retrieve their funds. Here we see the Coq `exists` keyword which represents an existential quantifier. The match expression represents the idea that the action must not merely be some action by anyone, but it must be one doable by the backer. In addition, the backer can get their funds back without giving away more funds, hence the callvalue must be zero.

This proof has a different structure to the previous two proofs. Clearly, the function that the backer needs to call is the `claim` function. As a result, this is the only function that needs to be unfolded rather than all three functions as in the previous proofs. The proof is analogous to a symbolic execution of the `claim` function, making use of the hypotheses when needed. The structure of the proof is shown below, with the full Coq proof available in Appendix B.3.

```
Theorem can_claim_back :
forall state s l backer_addr backed_amount,
  ReachableFromBy initial_state state s l ->
  backed_amount = Int256Tree.get_default 0 backer_addr
                   (Crowdfunding_backers (contract_state state)) ->
  contract_address <> backer_addr ->
  backed_amount > 0 ->
  (balance state backer_addr + backed_amount <? Int256.modulus) = true ->
  Crowdfunding_funded (contract_state state) = false ->
  balance state contract_address < (Crowdfunding_goal (contract_state state)) ->
  (forall a : addr, 0 <= balance state a < Int256.modulus) ->
  Int256.ltu
    (Crowdfunding_max_block (contract_state state)) (block_number state) = true ->
  exists (action : Action state),
          Outgoing_transfer_recipient_and_amount
            (contract_state (step_keep_transfer state action))
          = Some (backer_addr, backed_amount)
          /\
          match action with
          | call_Crowdfunding_donate context _ _ _ _ _ _ =>
              callvalue context = 0 /\ caller context = backer_addr
          | call_Crowdfunding_getFunds context _ _ _ _ _ _ =>
              callvalue context = 0 /\ caller context = backer_addr
          | call_Crowdfunding_claim context _ _ _ _ _ _ =>
              callvalue context = 0 /\ caller context = backer_addr
          | _ => False
          end.
Proof.
```

The proof begins by constructing an appropriate call to the `claim` function. We must also provide the resulting contract state and a proof that the call to the `claim` function succeeds.

```
intros.
```

After introducing the variables and the hypotheses into the context we are ready to provide the body of the appropriate call to `claim`. We will use the variant `eexists`, which allows us to leave underscores for portions which we will fill in later. By carefully comparing the type signature of *call_Crowdfunding_claim* (shown next) and the order of arguments provided to *call_Crowdfunding_claim* in the call to the `eexists` tactic we can see that the underscores correspond to the proof terms required for a successful call. The bulk of the proof involves showing that these hold.

```
Check call_Crowdfunding_claim.
```

```
call_Crowdfunding_claim
     : forall (before : BlockchainState)
         (context : CallContext)
         (callvalue_bounded_prf :
          0 <= callvalue context < Int256.modulus)
         (balances_bounded_prf :
          forall a : addr,
          0 <= balance before a < Int256.modulus)
         (callvalue_prf :
          noOverflowOrUnderflowInTransfer
            (caller context) contract_address
            (callvalue context)
            (balance before) = true)
         (r : unit)
         (contract_state_after : GetHighData),
       runStateT
         (Crowdfunding_claim_opt
            (make_machine_env contract_address before
               context
               address_accepts_funds_assumption
               callvalue_bounded_prf
               balances_bounded_prf callvalue_prf))
         (contract_state before) =
       Some (r, contract_state_after) ->
       Action before
```

```
eexists
  (call_Crowdfunding_claim
    state
    {| origin := backer_addr;
       caller := backer_addr;
       callvalue := 0;
       coinbase := Int256.zero;
       chainid := Int256.zero
    |}
    _ _ _ tt
    {| Outgoing_transfer_recipient_and_amount := (Some (backer_addr, backed_amount));
       Crowdfunding_owner := Crowdfunding_owner (contract_state state);
       Crowdfunding_max_block := Crowdfunding_max_block (contract_state state);
       Crowdfunding_goal := Crowdfunding_goal (contract_state state);
       Crowdfunding_backers :=
         set backer_addr 0 (Crowdfunding_backers (contract_state state));
```

```
        Crowdfunding_funded := Crowdfunding_funded (contract_state state);
    |} _).
```

The main proof goal of showing that an outgoing transfer of the form *Some* (*backer_addr*, *backed_amount*) sent from the *backer_addr* and with a callvalue of zero is now trivially true. The rest of the proof effort is involved in showing that this outcome is actually the true outcome of executing the `claim` function with these arguments.

```
    simpl.
```

```
______________________________________
Some (backer_addr, backed_amount) =
Some (backer_addr, backed_amount) /\
0 = 0 /\ backer_addr = backer_addr
```

```
split; [reflexivity|split; reflexivity].
```

The remaining goals from the underscores in `eexists` are then unshelved and solved. The proof is omitted here but is available in Appendix B.3.

```
Unshelve.
```

A point of interest in the proof is that the *sufficient_funds_safe* theorem is required to help prove a subgoal in this theorem. The goal at this point is to prove the particular branch is impossible, so the goal is *False* (logical impossibility in Coq). We have information in the context that, when combined with the knowledge that the balance is backed in the current state, gives a contradiction. So, we would like to establish that the balance is backed in the current state. We introduce the sufficient funds safe theorem into the context with the `pose` *proof* tactic.

```
pose proof sufficient_funds_safe as sufficient_funds_safe.
```

```
sufficient_funds_safe: Safe balance_backed
______________________________________
False
```

Next we unfold the definition of *Safe*.

```
unfold Safe in sufficient_funds_safe.
```

```
H: ReachableFromBy initial_state state s l
sufficient_funds_safe: forall
                      (state : BlockchainState)
                      (s : Step) (l : list Step),
                      ReachableFromBy initial_state
                      state s l ->
                      balance_backed state
______________________________________
False
```

Now, we have the antecedent of *sufficient_funds_safe* as the hypothesis *H* and so we can establish the consequent of *sufficient_funds_safe* by the `apply` tactic.

```
apply sufficient_funds_safe in H.
```

```
H: balance_backed state
________________________________
False
```

Now we have established that the balance is backed at the current state, the proof proceeds to derive a contradiction, showing that this branch is impossible given the assumptions made. The full proof is shown in Appendix B.3.

```
Qed.
```

## 5.9 Unsigned Integer Side-Conditions

In the proofs above, we have used the type $Z$ to represent the amount of ether that is donated and stored in the `backers` mapping. However, in reality, this should be a 256-bit unsigned integer. The DeepSEA system handles this situation by allowing us to reason about the unbounded integer type $Z$ in our main proofs, while generating side-conditions that should be proved to hold. These side-conditions can be proved by the *code_proofs_auto* tactic below, when no arithmetic operations on a variable of type $Z$ occur. Since we do not perform any arithmetic operations on unsigned integers in the crowdfunding smart contract, the proofs all succeed with the application of this tactic. The tactic unfolds the main obligation and repeatedly splits the conjunctions, solving each with the tactic `auto`. The next case study, in Chapter 6, does perform arithmetic operations on variables of type $Z$ and the tactic *code_proofs_auto* will be insufficient for some of its proof goals.

```
Ltac code_proofs_auto :=
    intros;
    unfold synth_func_obligation;
    repeat (split; auto).
```

Here we show the main lemmas for the constructor, the `donate`, `getFunds` and `claim` functions.

```
Lemma Crowdfunding_constructor_vc me d :
    high_level_invariant d →
    synth_func_cond Crowdfunding_constructor Crowdfunding_constructor_wf
                    me d.
Proof.
code_proofs_auto.
Qed.

Lemma Crowdfunding_donate_vc me d :
    high_level_invariant d →
    synth_func_cond Crowdfunding_donate Crowdfunding_donate_wf
                    me d.
Proof.
code_proofs_auto.
Qed.
```

```
Lemma Crowdfunding_getFunds_vc me d :
    high_level_invariant d →
    synth_func_cond Crowdfunding_getFunds Crowdfunding_getFunds_wf
                    me d.
Proof.
code_proofs_auto.
Qed.

Lemma Crowdfunding_claim_vc me d :
    high_level_invariant d →
    synth_func_cond Crowdfunding_claim Crowdfunding_claim_wf
                    me d.
Proof.
code_proofs_auto.
Qed.
```

## 5.10 Remarks

These three theorems together, along with the proofs of the unsigned integer side-conditions, give a strong guarantee that the crowdfunding smart contract behaves correctly from a donor's perspective. Ideally, the statements of these theorems would be examined by any prospective donor to ensure they were consistent with their mental model of what a crowdfunding smart contract should behave like. If the donor trusted the proof system and the model described in Section 4.2, there would be no need to examine the body of these specific proofs or to examine the source code of the smart contract. The hope is that, in general, the statements of the theorems would be more easily understandable than the implementation of the smart contract and allow end-users to have greater trust in the smart contracts they interact with without the need to carefully examine their implementations.

# Chapter 6

# Case Study: ERC-20 Wrapped Ether Token

Having explored correctness proofs for properties of a crowdfunding smart contract, we now explore an ERC-20 [14] wrapped ether smart contract that implements a fungible token inherently pegged to the value of ether at a 1:1 ratio. Fungible tokens correspond to values of varying quantity, unlike non-fungible tokens (NFTs) which correspond to unique, non-countable items. Anyone can send ether to this fungible token smart contract ('minting' the token) and the amount of ether they sent will be set aside for the user to withdraw at a later stage ('burning' the token). The user can also transfer their wrapped ether to someone else as well as permit a third party to transfer some amount of their wrapped tokens on their behalf, in line with the ERC-20 specification [14]. This token is written to be compliant with the ERC-20 specification, and some properties described or implied by the standard have been proved to hold. The implementation of this smart contract and its proofs could be adapted to create custom ERC-20 tokens without needing to redo most of the proof effort.

We will explore five proofs, the first similar to the preservation of the donation record from the crowdfunding smart contract. Here, we will prove that once ether has been wrapped, if there are no transfers or burning of the token, then the user retains at least that much wrapped ether. Secondly, we will prove a property about the transfer function alone, to demonstrate how one can phrase a property directly about a single smart contract function *without* making use of the blockchain model introduced in Section 4.2. The remaining three proofs are safety properties and bear similarities to the proof of the sufficient balance theorem of the crowdfunding smart contract.

## 6.1 Contract Source Code

First, we will introduce the source code for the ERC-20 wrapped ether smart contract as written in the DeepSEA language. The automatic translations into Coq by the DeepSEA system of these smart contract functions are available in Appendix C.1.

### 6.1.1 Preliminary Definitions

The ERC-20 token standard [14] describes events which must be emitted when a transfer or approval occurs. The following code sets up an algebraic type which can construct the relevant events to be emitted. DeepSEA does not currently model the emitting of events, so there is no way to prove that the events are emitted as the specification outlines. Future work could involve extending DeepSEA to model the emitting of events. However, the focus is instead on modelling the business logic of smart contracts.

Listing 6.1: ERC-20 Wrapped Ether Contract Source Code - Part 1

```
event
    | Transfer (_from : address indexed) (_to : address indexed) (_value : int)
    | Approval (_owner : address indexed) (_spender : address indexed) (_value : int)
```

### 6.1.2 Type Signatures

We define nine functions, six which are required by the ERC-20 specification, one which is recommended as an improvement to the ERC-20 specification, and two which allow the user to mint and burn ether to create tokens and withdraw ether from existing value stored as tokens. The functions annotated by `const` do not alter the smart contract storage in any way.

Listing 6.2: ERC-20 Wrapped Ether Contract Source Code - Part 2

```
object signature ERC20WrappedEthSig = {
    const totalSupply : unit -> int;
    const balanceOf : address -> int;
    transfer : address * int -> bool;
    transferFrom : address * address * int -> bool;
    const allowance : address * address -> int;
    approve : address * int -> bool;
    approveSafely : address * int * int -> bool;
    mint : unit -> bool;
    burn : int -> bool
}
```

### 6.1.3 State Variables

There are three state variables: the `wrapped` and `allowances` mappings and `_totalSupply`. The variable `_totalSupply` tracks the amount of the token currently in circulation, which should be equal to the sum of the values of the `wrapped` mapping. The allowances mapping is multidimensional, with the first address having permission to spend the tokens of the second address up to the amount held in that mapping. The `approve` function enables a user to give this permission to another user (or smart contract).

Listing 6.3: ERC-20 Wrapped Ether Contract Source Code - Part 3

```
object ERC20WrappedEth () : ERC20WrappedEthSig {
    let wrapped : mapping[address] int := mapping_init
    let allowances : mapping[address] mapping[address] int := mapping_init
    let _totalSupply : int := 0
```

### 6.1.4 Total Supply Function

This function returns the total amount of wrapped ether currently stored by the contract. The value of `_totalSupply` is initially zero and is changed only by the `mint` and `burn` functions.

Listing 6.4: ERC-20 Wrapped Ether Contract Source Code - Part 4

```
    let totalSupply () =
      _totalSupply
```

### 6.1.5 Balance Function

The balance function returns the value of ether held by the address supplied as an argument, `_owner`, as stored in the `wrapped` mapping.

Listing 6.5: ERC-20 Wrapped Ether Contract Source Code - Part 5

```
    let balanceOf(_owner) =
      wrapped[_owner]
```

### 6.1.6 Transfer Function

The `transfer` function enables a user to transfer a specified amount (`_value`) of their wrapped ether to a different Ethereum address. It adjusts the `wrapped` mapping so that the recipient's ERC-20 token balance increases by `_value` and the caller's ERC-20 token balance reduces by `_value`. We disallow transfers to oneself but do allow transfers of value zero, as required by the ERC-20 specification. The function returns `true`, to indicate that the transfer occurred successfully, if none of the assertions fail.

Listing 6.6: ERC-20 Wrapped Ether Contract Source Code - Part 6

```
    let transfer(_to, _value) =
      assert(_value >= 0);
      assert(msg_sender <> this_address);
      assert(msg_sender <> _to);
      assert(msg_value = 0);
      let wrapped_amount_from = wrapped[msg_sender] in
      let wrapped_amount_to = wrapped[_to] in
      assert(wrapped_amount_from >= _value);
      wrapped[_to] := wrapped_amount_to + _value; (* Continued over page *)
      wrapped[msg_sender] := wrapped_amount_from - _value;
      emit Transfer(msg_sender, _to, _value);
      true
```

### 6.1.7 Transfer-From Function

The `transferFrom` function is similar to the `transfer` function, except that it allows a user who has been given permission via the `approve` or `approveSafely` functions (discussed next) to spend another user's wrapped ether on their behalf. The function checks that the user is authorised to spend the amount of the other user's wrapped ether that they are attempting to spend and then enacts the transfer. The function returns true if all the assertions pass to indicate a successful transfer took place.

Listing 6.7: ERC-20 Wrapped Ether Contract Source Code - Part 7

```
    let transferFrom(_from, _to, _value) =
      assert(_value >= 0);
      assert(_from <> this_address);
      assert(_from <> _to);
      assert(msg_value = 0);
      let approved_amount = allowances[_from][msg_sender] in
      assert(approved_amount >= _value);
      allowances[_from][msg_sender] := approved_amount - _value;
      let wrapped_amount_from = wrapped[_from] in
      let wrapped_amount_to = wrapped[_to] in
      assert(wrapped_amount_from >= _value);
      wrapped[_to] := wrapped_amount_to + _value;
      wrapped[_from] := wrapped_amount_from - _value;
      emit Transfer(_from, _to, _value);
      true
```

### 6.1.8 Allowance Function

The `allowance` function returns the amount of the wrapped ether tokens of an `_owner` that a `_spender` has been permitted to spend. The values in this mapping are set using the `approve` and `approveSafely` functions, described next. Initially, all values are set to zero.

Listing 6.8: ERC-20 Wrapped Ether Contract Source Code - Part 8

```
    let allowance(_owner, _spender) =
      allowances[_owner][_spender]
```

### 6.1.9 Approve Function

If a user wishes to allow another Ethereum address to transfer tokens on their behalf, the `approve` function enables the user to give this permission. This permission, which is stored in the `allowances` mapping, is checked by the `transferFrom` function before the that function transfers tokens.

When calling the `approve` function to update an approved amount, there is the potential for the approved user to notice the upcoming change to the approved amount in the pool of transactions that are yet to be finalised and to quickly spend both the original approved amount (a form of front-running attack) and then also spend the new approved amount. This vulnerability is described in [148], which suggests a solution which we have implemented as the `approveSafely` function, described next. For backwards-compatibility and compliance with the official ERC-20 specification, the `approve` function is also included as described in the original specification. The blockchain model in this thesis is not expressive enough to model front-running attacks. However, in principle it would be possible to extend the model to do so. This would be valuable future work.

Listing 6.9: ERC-20 Wrapped Ether Contract Source Code - Part 9

```
    let approve (_spender, _value) =
      assert(_value >= 0);
      allowances[msg_sender][_spender] := _value;
      emit Approval(msg_sender, _spender, _value);
      true
```

### 6.1.10 Approve-Safely Function

The `approveSafely` function is the improved version of the `approve` function which requires the caller to provide a `_currentValue` of the value they expect the approved user to have as their current allowance. The intended usage is as follows:

1. The caller queries `allowance(their_own_address, spender)` and notes this result, we will call it `expected_allowance`.
2. The caller immediately calls `approveSafely(spender, expected_allowance, new_allowance)`

When the second call is made, if there is a difference between the `expected_allowance` and the current allowance of the caller's tokens permitted to be transferred by the spender this could mean that a front-running attack as described earlier has occurred. Alternatively, it could mean that the spender has innocently happened to transfer funds on behalf of the caller at almost exactly the same time as their change in allowance. Either way, the caller may now wish to set a different allowance given this new information about how much the spender has spent on their behalf. As a result, `approveSafely` does not alter the allowances mapping in this scenario, instead making no changes and returning `false`.

Listing 6.10: ERC-20 Wrapped Ether Contract Source Code - Part 10

```
let approveSafely (_spender, _currentValue, _value) =
  assert(_value >= 0);
  let actualCurrentValue = allowances[msg_sender][_spender] in
  if (_currentValue = actualCurrentValue) then
    begin
      allowances[msg_sender][_spender] := _value;
      emit Approval(msg_sender, _spender, _value);
      true
    end
  else
    false
```

### 6.1.11 Mint Function

The `mint` function enables a user to transfer ether to the smart contract in exchange for being allocated the equivalent amount of the wrapped ether token, recorded in the `wrapped` mapping. It also updates the `_totalSupply` variable to keep track of the total amount of wrapped ether held by the smart contract as per the ERC-20 specification. The entire `msg_value` (callvalue) is assumed to be intended to become wrapped ether, so there are no explicit arguments to the `mint` function.

Listing 6.11: ERC-20 Wrapped Ether Contract Source Code - Part 11

```
let mint () =
  assert(msg_sender <> this_address);
  assert(msg_value > 0);

  let wrapped_amount = wrapped[msg_sender] in
  wrapped[msg_sender] := wrapped_amount + msg_value;
  let prev_totalSupply = _totalSupply in
  _totalSupply := prev_totalSupply + msg_value;
  emit Transfer(address(0x0), msg_sender, msg_value);
  true
```

### 6.1.12 Burn Function

Finally, the burn function enables users to withdraw the specified `_value` of their wrapped ether tokens and receive ether in exchange. The `_totalSupply` variable is also updated. Note that the call to `transferEth` to send the funds to the user occurs after the variables are updated. This is to comply with the *Checks-Effects-Interactions Pattern* as discussed in Section 4.1.

Listing 6.12: ERC-20 Wrapped Ether Contract Source Code - Part 12

```
let burn (_value) =
  assert(_value >= 0);
  assert(msg_sender <> this_address);
  assert(msg_value = 0); (* Continued over page *)
```

```
      let wrapped_amount = wrapped[msg_sender] in
      assert(wrapped_amount >= _value);
      wrapped[msg_sender] := wrapped_amount - _value;
      let prev_totalSupply = _totalSupply in
      _totalSupply := prev_totalSupply - _value;
      transferEth(msg_sender, _value);
      emit Transfer(msg_sender, address(0x0), _value);
      true
}
```

### 6.1.13 Layer Definition

Since this smart contract is also only of one layer, the layer definition is as follows.

Listing 6.13: ERC-20 Wrapped Ether Contract Source Code - Part 13

```
  layer CONTRACT : [ { } ]  {erc20wrappedeth : ERC20WrappedEthSig}  = {
    erc20wrappedeth = ERC20WrappedEth
}
```

## 6.2 Initial State

Similarly to the crowdfunding smart contract, the initial state is automatically provided by the DeepSEA system. The initial smart contract data given by *init_global_abstract_data* includes the three variables of the contract, as well as the initially empty *Outgoing_transfer_recipient_and_amount* entry. Here is the contents of the initial smart contract state:

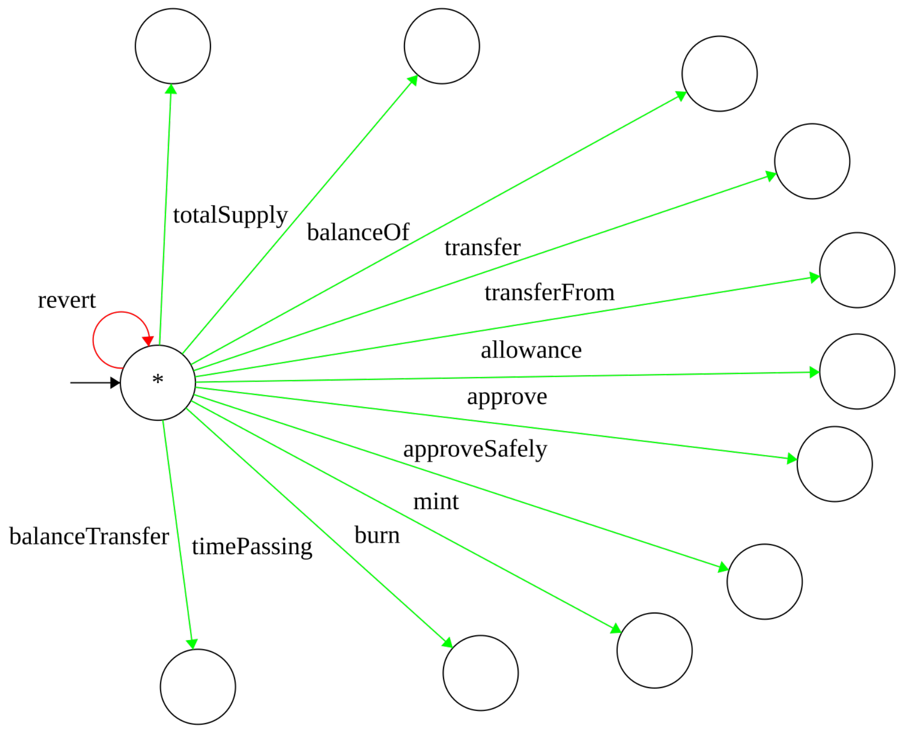
```
Print init_global_abstract_data.
```

```
init_global_abstract_data =
{|
  Outgoing_transfer_recipient_and_amount := None;
  ERC20WrappedEth_wrapped := empty Z32;
  ERC20WrappedEth_allowances := empty (t Z32);
  ERC20WrappedEth__totalSupply := 0
|}
      : global_abstract_data_type
```

## 6.3 Step Function

The step function is similar to the step function for the crowdfunding smart contract, with the transitions for the smart contract functions replaced by the functions for the ERC-20 wrapped ether smart contract. Figure 6.1 summarises the possible actions described by the *Action* dependent type and the *step* function for the ERC-20 wrapped ether smart contract. *Action* and *step* are included in Appendices C.2 and C.3.

Figure 6.1: Outline of the step function for the ERC-20 wrapped ether smart contract



## 6.4 Reachability Predicate

The reachability predicate is identical to the crowdfunding smart contract reachability predicate described in Section 5.4, except that the definition for the step function now is the version for the ERC-20 wrapped ether smart contract.

## 6.5 Preservation of Wrapped Ether Record Theorem

Having defined the ERC-20 wrapped ether smart contract, we now explore the proofs of four theorems about it, looking at two in fine detail. First, the *wrapped_preserved* theorem. This is similar to the *donation_preserved* theorem and also makes use of the same *since_as_long* predicate. We define the properties *wrappedAtLeast* and *no_transfer_or_burn_from* which are similar to the properties *donation_recorded* and *no_claims_from*, respectively.

The reason we use the property *wrappedAtLeast*, that at least a specified amount is wrapped rather than that exactly a specific amount being wrapped, is because the user may be the recipient of transfers which increase their balance beyond the amount they initially have wrapped. We are aiming to prove that once at least a specified amount is wrapped then at least that amount remains wrapped, unless there is a call to any of the `transfer`, `transferFrom` or `burn` functions.

```
Definition wrappedAtLeast (a : addr) (amount : Z) (s : BlockchainState) :=
    Int256Tree.get_default 0 a (ERC20WrappedEth_wrapped (contract_state s)) ≥ amount ∧ amount > 0.

Definition no_transfer_or_burn_from (a : addr) (s : Step) :=
  match Step_action s with
  | (call_ERC20WrappedEth_burn _ context _ _ _ _ _ _) ⇒ caller context ≠ a
  | (call_ERC20WrappedEth_transferFrom _from _ _ context _ _ _ _ _ _) ⇒ _from ≠ a
  | (call_ERC20WrappedEth_transfer _ _ context _ _ _ _ _ _) ⇒ caller context ≠ a
  | _ ⇒ True
  end.
```

The proof has aspects which are similar to the *donation_preserved* theorem. However, one difference is that the definitions of the ERC-20 wrapped ether smart contract functions (defined in Section 6.1) make use of assertions instead of a heavy use of 'if' statements as in the crowdfunding smart contract. Since the DeepSEA system places assertions directly as hypotheses this results in less explicit case analysis in this proof, as is clear by the slightly shallower nesting of bullets.

Now we unpack the proof of the `wrapped_preserved` theorem.

```
Theorem wrapped_preserved (a : addr) (amount : Z) :
                (wrappedAtLeast a amount)
  `since`       (wrappedAtLeast a amount)
  `as-long-as` (no_transfer_or_burn_from a).
Proof.
```

```
unfold since_as_long. intros.
```

```
H: ReachableFromBy from_state to_state to_step steps
____________________________________________________
wrappedAtLeast a amount to_state
```

We proceed by induction on the reachability predicate *H*.

```
induction H.
-
```

```
H0: wrappedAtLeast a amount from_state
____________________________________________________
wrappedAtLeast a amount from_state
```

We prove the base case using the assumption from property *P* in the *since_as_long* predicate.

```
  assumption.
-
```

Proving the inductive case requires case analysis on the action, breaking our goal into twelve subgoals corresponding to the twelve transitions shown in Figure 6.1.

The first nine subgoals correspond to the functions of the smart contract and the final three correspond to the generic actions of time passing on the blockchain, ether transfers not originating from the contract, and the catch-all revert action for failure cases. First there is some housekeeping.

```
  assert(wrappedAtLeast a amount prevSt) as IHReachableFromByCorollary by
```

```
    (apply IHReachableFromBy; intros; apply H1; apply in_cons; assumption).
  unfold wrappedAtLeast in *;
    destruct IHReachableFromByCorollary
      as [IHReachableFromByCorollary1 IHReachableFromByCorollary2].
  split; [|assumption].
  Hlinks.
  assert(no_transfer_or_burn_from a prev) by
    (apply H1; destruct HL; subst; right; left; auto).
  destruct prev; autounfold in *; simpl in *.
  unfold no_transfer_or_burn_from in H2.
  destruct Step_action0; simpl in *.
```

The first subgoal is the case for the `totalSupply` function, which does not alter the `wrapped` mapping. As a result, the goal follows from a corollary of the inductive hypothesis after establishing that the `wrapped` mapping is not altered. The proof engages with the definition of the `totalSupply` function after unfolding it and makes use of the *ds_inv* tactic.

```
  + Transparent ERC20WrappedEth_totalSupply_opt.
    unfold ERC20WrappedEth_totalSupply_opt in case_totalSupply_prf.
    ds_inv; subst; simpl in *.
```

```
case_totalSupply_prf: Some
                      (ERC20WrappedEth__totalSupply
                      (contract_state
                      Step_state0),
                      contract_state Step_state0) =
                      Some (r, contract_state_after)
-----------------------------------------------------------
get_default 0 a
  (ERC20WrappedEth_wrapped contract_state_after) >=
amount
```

We use inversion to establish that *contract_state_after* = *contract_state Step_state0*

```
    inversion case_totalSupply_prf.
```

```
IHReachableFromByCorollary1: get_default 0 a
                             (ERC20WrappedEth_wrapped
                             (contract_state
                             Step_state0)) >=
                             amount
-----------------------------------------------------------
get_default 0 a
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0)) >= amount
```

Now the goal matches a corollary of the inductive hypothesis exactly.

```
    exact IHReachableFromByCorollary1.
```

Next is the case for the `balanceOf` function, which also does not alter the `wrapped` mapping. The proof is essentially identical to that for the `totalSupply` case.

```
  + Transparent ERC20WrappedEth_balanceOf_opt.
    unfold ERC20WrappedEth_balanceOf_opt in case_balanceOf_prf.
    ds_inv; subst; simpl in *.
    inversion case_balanceOf_prf.
    exact IHReachableFromByCorollary1.
```

We then have the `transfer` function. Here, we need to consider whether or not the recipient, *_to*, is the address we are considering, *a*.

```
  + Transparent ERC20WrappedEth_transfer_opt.
    unfold ERC20WrappedEth_transfer_opt in case_transfer_prf.
    clear H HL.
    ds_inv; subst; simpl in *.
    destruct (a =? _to)%int256 eqn:Case.
      * apply Int256eq_true in Case.
```

```
Case: a = _to
─────────────────────────────────────────
get_default 0 a
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)) - _value)
     (set _to
        (get_default 0 _to
           (ERC20WrappedEth_wrapped
              (contract_state Step_state0)) + _value)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)))) >= amount
```

In the case where it is, the `wrapped` funds for *a* increase, or stay the same if the transfer is of value 0, so the property *wrappedAtLeast* is preserved.

```
        subst.
        apply (f_equal negb) in H12. rewrite negb_involutive in H12.
        apply Int256eq_false in H12.
```

```
H12: caller context <> _to
________________________________________
get_default 0 _to
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)) - _value)
     (set _to
        (get_default 0 _to
           (ERC20WrappedEth_wrapped
              (contract_state Step_state0)) + _value)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)))) >= amount
```

```
        rewrite get_default_so by auto.
```

```
________________________________________
get_default 0 _to
  (set _to
     (get_default 0 _to
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)) + _value)
     (ERC20WrappedEth_wrapped
        (contract_state Step_state0))) >= amount
```

```
        apply geb_ge in H4.
        rewrite get_default_ss.
```

```
IHReachableFromByCorollary1: get_default 0 _to
                             (ERC20WrappedEth_wrapped
                             (contract_state
                             Step_state0)) >=
                             amount
H4: _value >= 0
________________________________________
get_default 0 _to
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0)) + _value >= amount
```

Now the goal follows from the inductive hypothesis, along with the knowledge that the argument `_value` is non-negative. This should always be the case but needs to be guaranteed by an assertion due to the current handling of unsigned integers in DeepSEA. The goal is solvable by the linear integer arithmetic tactic.

```
        lia.
```

In the case where the recipient *_to* is not the address $a$ that we are concerned with then the alterations this function makes to the `wrapped` mapping do not affect the address *a* so our goal is exactly a corollary of the inductive hypothesis after rewriting the goal.

```
      * apply Int256eq_false in Case.
```

```
H2: caller context <> a
--------------------------------------------------------------------------------
get_default 0 a
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)) - _value)
     (set _to
        (get_default 0 _to
           (ERC20WrappedEth_wrapped
              (contract_state Step_state0)) + _value)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)))) >= amount
```

Here, we use the "get default set other" lemma, along with the hypotheses shown to simplify the goal because the key we are retrieving the value for is not the one that is having its value changed. The statement of *get_default_so* is shown below.

```
        Check get_default_so.
```

```
get_default_so
     : forall (A : Type)
         (def : A)
         (k k0 : elt)
         (v : A) (m : t A),
       k <> k0 ->
       get_default def k (set k0 v m) =
       get_default def k m
```

```
        rewrite get_default_so by auto.
```

```
Case: a <> _to
--------------------------------------------------------------------------------
get_default 0 a
  (set _to
     (get_default 0 _to
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)) + _value)
     (ERC20WrappedEth_wrapped
        (contract_state Step_state0))) >= amount
```

```
        rewrite get_default_so by auto.
```

```
IHReachableFromByCorollary1: get_default 0 a
                             (ERC20WrappedEth_wrapped
                             (contract_state
                             Step_state0)) >=
                             amount
------------------------------------------------------------
get_default 0 a
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0)) >= amount
```

```
        exact IHReachableFromByCorollary1.
```

Next, we have the `transferFrom` function. The proof is identical to the case for `transfer` once the definition of `transferFrom` has been unfolded.

```
  + Transparent ERC20WrappedEth_transferFrom_opt.
    unfold ERC20WrappedEth_transferFrom_opt in case_transferFrom_prf.
    clear H HL.
    ds_inv; subst; simpl in *.
    destruct (a =? _to)%int256 eqn:Case.
    * apply Int256eq_true in Case.
      subst.
      apply (f_equal negb) in H12. rewrite negb_involutive in H12.
      apply Int256eq_false in H12.
      rewrite get_default_so by auto.
      apply geb_ge in H4.
      rewrite get_default_ss.
      clear -IHReachableFromByCorollary1 H4.
      lia.
    * apply Int256eq_false in Case.
      rewrite get_default_so by auto.
      rewrite get_default_so by auto.
      exact IHReachableFromByCorollary1.
```

Next is the `allowance` function. It does not alter the `wrapped` mapping. The proof is essentially identical to the proofs for the other cases which do not alter the `wrapped` mapping.

```
  + Transparent ERC20WrappedEth_allowance_opt.
    unfold ERC20WrappedEth_allowance_opt in case_allowance_prf.
    ds_inv; subst; simpl in *.
    inversion case_allowance_prf.
    exact IHReachableFromByCorollary1.
```

The `approve` function also does not alter the `wrapped` mapping and the proof is similar to such previous cases.

```
+ Transparent ERC20WrappedEth_approve_opt.
  unfold ERC20WrappedEth_approve_opt in case_approve_prf.
  ds_inv; subst; simpl in *.
  inversion case_approve_prf.
  clear H HL case_approve_prf.
  destruct (_value >=? 0); simpl in *; inversion H4.
  simpl in *.
  exact IHReachableFromByCorollary1.
```

The `approveSafely` function needs to have an explicit case analysis in the proof due to its 'if' statement. But it also does not alter the `wrapped` mapping and the proof bears strong similarities to previous cases.

```
+ Transparent ERC20WrappedEth_approveSafely_opt.
  unfold ERC20WrappedEth_approveSafely_opt in case_approveSafely_prf.
  ds_inv; subst; simpl in *.
  inversion case_approveSafely_prf.
```

```
________________________________________
get_default 0 a
  (ERC20WrappedEth_wrapped contract_state_after) >=
amount
```

```
  clear H HL case_approveSafely_prf.
  destruct (_value >=? 0); simpl in *; inversion H4.
```

Here we need to perform case analysis on the guard of the 'if' statement to demonstrate that neither case affects the `wrapped` mapping.

```
  destruct (_currentValue =?
              get_default 0 _spender
                (get_default (empty Z) (caller context)
                  (ERC20WrappedEth_allowances (contract_state Step_state0))));
    inversion H3; simpl in *; exact IHReachableFromByCorollary1.
```

We now move on to the two functions which are added beyond the ERC-20 specification and make this a *wrapped ether* ERC-20 token. Firstly, the `mint` function.

```
+ Transparent ERC20WrappedEth_mint_opt.
  unfold ERC20WrappedEth_mint_opt in case_mint_prf.
  clear H HL. simpl in *.
  ds_inv; subst; simpl in *.
  destruct(caller context =? contract_address)%int256; simpl in *;
  ds_inv; subst; simpl in *; try discriminate.
  destruct(callvalue context >? 0)%int256; simpl in *; simpl in *;
  ds_inv; subst; simpl in *; try discriminate.
  inversion case_mint_prf; simpl in *.
```

The heart of this branch of the proof is where we consider whether or not the address we are concerned with, $a$, is the caller. We perform case analysis.

```
destruct (a =? (caller context))%int256 eqn:Case.
* apply Int256eq_true in Case.
```

```
Case: a = caller context
----------------------------------------------------------------
get_default 0 a
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)) +
      callvalue context)
     (ERC20WrappedEth_wrapped
        (contract_state Step_state0))) >= amount
```

First we consider when the address $a$ is the caller.

```
rewrite <- Case in *.
rewrite get_default_ss.
```

```
callvalue_bounded_prf: 0 <= callvalue context
                        <
                        Int256.modulus
IHReachableFromByCorollary1: get_default 0 a
                             (ERC20WrappedEth_wrapped
                             (contract_state
                             Step_state0)) >=
                             amount
----------------------------------------------------------------
get_default 0 a
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0)) +
callvalue context >= amount
```

Having simplified the goal, it now follows from the inductive hypothesis plus the knowledge that the callvalue is non-negative. The goal is solved by the linear integer arithmetic tactic.

```
lia.
```

In the case where the address $a$ is not the caller, the goal simplifies to a corollary of the inductive hypothesis. This is because the setting of values in the `wrapped` mapping does not affect the value read for the address $a$.

```
* apply Int256eq_false in Case.
```

```
Case: a <> caller context
------------------------------------------------
get_default 0 a
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)) +
      callvalue context)
     (ERC20WrappedEth_wrapped
        (contract_state Step_state0))) >= amount
```

```
      rewrite get_default_so by apply Case.
```

```
IHReachableFromByCorollary1: get_default 0 a
                             (ERC20WrappedEth_wrapped
                             (contract_state
                             Step_state0)) >=
                             amount
------------------------------------------------
get_default 0 a
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0)) >= amount
```

```
      exact IHReachableFromByCorollary1.
```

The next transition is for the *burn* smart contract function. As with all these proofs, the body of the function is engaged with, along with use of the *ds_inv* to make inferences from the knowledge that the body of the function succeeds.

We have assumed from the *no_transfer_or_burn_from* property in *since_as_long* that the caller is not the address, *a*, which we are interested in. As a result, the proof of this case follows from simplifying the goal using our assumption. This case should not be provable without the *no_transfer_or_burn_from* assumption.

```
  + Transparent ERC20WrappedEth_burn_opt.
    unfold ERC20WrappedEth_burn_opt in case_burn_prf.
    clear H HL.
    ds_inv; subst.
    * simpl in *.
```

```
H2: caller context <> a
____________________________________________
get_default 0 a
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)) - _value)
     (ERC20WrappedEth_wrapped
        (contract_state Step_state0))) >= amount
```

At this point in the proof we have assumed from the *no_transfer_or_burn_from* property in *since_as_long* that the caller is not the address, *a*, which we are interested in. As a result we can use the *get_default_so* lemma to effectively ignore the `set` which is affecting an address other than *a*.

We dismiss the antecedent of *get_default_so* by the tactic `auto` which swaps the order of *a* and *caller* `context` in *H2* and applies it.

```
      rewrite get_default_so by auto.
```

```
____________________________________________
get_default 0 a
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0)) >= amount
```

```
      exact IHReachableFromByCorollary1.
```

This case corresponds to the transfer of ether to the user failing. Since we have assumed via the success-calls approach that the contract succeeds, this case leads to a contradictory assumption *Heqb* and can be dismissed via the principle of ex falso quodlibet.

```
    * exfalso. simpl in *. apply Int256eq_true in Heqb.
```

```
Heqb: Int256.zero = Int256.one
____________________________________________
False
```

```
      inversion Heqb.
```

The final three cases correspond to the external balance transfer, time passing and revert cases. Since none of these actions alter the `wrapped` mapping, they are trivially true by a corollary of the inductive hypothesis.

```
  + rewrite <- HS. apply IHReachableFromByCorollary1.
  + rewrite <- HS. apply IHReachableFromByCorollary1.
  + rewrite <- HS. apply IHReachableFromByCorollary1.

Qed.
```

## 6.6 Transfer Correctness Theorem

This theorem demonstrates how one can phrase a theorem without referring to the blockchain model introduced in Section 4.2. Here we universally quantify over a number of variables, and then assume that these variables result in a successful call to the `transfer` function. It is also possible to demonstrate that a function succeeds, rather than assuming it does, which gives a theorem more similar to the *can_claim_back* theorem from Section 5.8.

Once we have assumed the `transfer` function succeeds, we assume that the variables *_from_balance_before*, *_to_balance_before*, *_from_balance_after*, and *_to_balance_after* have their natural meanings with respect to the `wrapped` mapping in the *contract_state before* and *contract_state_after*. Then we aim to prove that the amounts stored are altered as expected for a transfer of value in the resulting *contract_state_after*. In other words, we aim to prove that if the smart contract function succeeds, then the `wrapped` mapping successfully reflects this transfer. By assuming that the smart contract succeeds, we have implicitly assumed that the sender, *_from*, has a sufficient balance to make the transfer. This is due to assuming that the `transfer` function succeeds, which implies the sender had a sufficient balance.

Here is the proof of the theorem.

```
Theorem transfer_correct :
  forall _to _value _from_balance_before _from_balance_after
                    _to_balance_before   _to_balance_after
   before context r contract_state_after
    callvalue_bounded_prf balances_bounded_prf callvalue_prf,
  runStateT (ERC20WrappedEth_transfer_opt _to _value
    (make_machine_env contract_address before context
      address_accepts_funds_assumption callvalue_bounded_prf
      balances_bounded_prf callvalue_prf))
    (contract_state before)
= Some (r, contract_state_after)
  -> _from_balance_before =
      Int256Tree.get_default 0 (caller context)
        (ERC20WrappedEth_wrapped (contract_state before))
  -> _to_balance_before =
      Int256Tree.get_default 0 _to
        (ERC20WrappedEth_wrapped (contract_state before))
  -> _from_balance_after =
      Int256Tree.get_default 0 (caller context)
        (ERC20WrappedEth_wrapped contract_state_after)
  -> _to_balance_after =
      Int256Tree.get_default 0 _to
        (ERC20WrappedEth_wrapped contract_state_after)
  ->     _to_balance_after = _to_balance_before + _value
     /\  _from_balance_after = _from_balance_before - _value.
Proof.
```

Here, we proceed by introducing the variables and hypotheses, unfolding the definition of the `transfer` function, and using the tactic *ds_inv* to make use of the knowledge that the `transfer` function succeeds.

```
intros.
Transparent ERC20WrappedEth_transfer_opt. unfold ERC20WrappedEth_transfer_opt in H.
ds_inv. subst. simpl in *.
```

```
______________________________________________________________________
get_default 0 _to
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state before)) - _value)
     (set _to
        (get_default 0 _to
           (ERC20WrappedEth_wrapped
              (contract_state before)) + _value)
        (ERC20WrappedEth_wrapped
           (contract_state before)))) =
get_default 0 _to
  (ERC20WrappedEth_wrapped (contract_state before)) +
_value /\
get_default 0 (caller context)
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state before)) - _value)
     (set _to
        (get_default 0 _to
           (ERC20WrappedEth_wrapped
              (contract_state before)) + _value)
        (ERC20WrappedEth_wrapped
           (contract_state before)))) =
get_default 0 (caller context)
  (ERC20WrappedEth_wrapped (contract_state before)) -
_value
```

This leaves us with two subgoals joined by a conjunction, which have had *WrappedEth_wrapped contract_state_after* replaced with the value determined by *ds_inv* as what must be equal with that. The changes correspond to the effect on the `wrapped` mapping from the two lines preceding the `emit` statement of the `transfer` function shown in Listing 6.7.

```
split.
```

The first goal corresponds to $\_to\_balance\_after = \_to\_balance\_before + \_value$.

```
- apply (f_equal negb) in H9. rewrite negb_involutive in H9.
  apply Int256eq_false in H9.
```

```
H9: caller context <> _to
--------------------------------------------------------------------------------
get_default 0 _to
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state before)) - _value)
     (set _to
        (get_default 0 _to
           (ERC20WrappedEth_wrapped
              (contract_state before)) + _value)
        (ERC20WrappedEth_wrapped
           (contract_state before)))) =
get_default 0 _to
  (ERC20WrappedEth_wrapped (contract_state before)) +
_value
```

The first conjunct is solved by application of the tactics *get_default_so* and *get_default_ss*. First, *get_default_so* eliminates the setting of the *_from* balance in the chain of updates shown in the goal. This is a correct simplification because transfers to oneself are not permitted by this smart contract.

```
rewrite get_default_so by auto.
```

```
--------------------------------------------------------------------------------
get_default 0 _to
  (set _to
     (get_default 0 _to
        (ERC20WrappedEth_wrapped
           (contract_state before)) + _value)
     (ERC20WrappedEth_wrapped (contract_state before))) =
get_default 0 _to
  (ERC20WrappedEth_wrapped (contract_state before)) +
_value
```

The goal now can be simplified further because we are reading the value for the same key which was just set, for the recipient *_to*.

```
rewrite get_default_ss.
```

```
________________________________________________________________
get_default 0 _to
  (ERC20WrappedEth_wrapped (contract_state before)) +
_value =
get_default 0 _to
  (ERC20WrappedEth_wrapped (contract_state before)) +
_value
```

Now we have shown that the first conjunct simplifies to the value we expected and the goal can be solved via the `reflexivity` tactic.

```
    reflexivity.
```

The second goal corresponds to *_from_balance_after* = *_from_balance_before* - *_value*.

```
  - apply negb_true_iff in H9. apply Int256eq_false in H9.
```

```
H9: caller context <> _to
________________________________________________________________
get_default 0 (caller context)
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state before)) - _value)
     (set _to
        (get_default 0 _to
           (ERC20WrappedEth_wrapped
              (contract_state before)) + _value)
        (ERC20WrappedEth_wrapped
           (contract_state before)))) =
get_default 0 (caller context)
  (ERC20WrappedEth_wrapped (contract_state before)) -
_value
```

Here, the intermediate step of setting the value for the recipient in the mapping does not affect the value of getting the value for the caller from the mapping which is set afterwards. As a result, we can rewrite using the *get_default_ss* lemma.

```
    rewrite get_default_ss.
```

```
________________________________________________________________
get_default 0 (caller context)
  (ERC20WrappedEth_wrapped (contract_state before)) -
_value =
get_default 0 (caller context)
  (ERC20WrappedEth_wrapped (contract_state before)) -
_value
```

Now the goal has been simplified to the expected form and is trivially solvable by the `reflexivity` tactic.

```
    reflexivity.
```

```
Qed.
```

## 6.7 Correctness of the Total Supply Variable Theorem

The `totalSupply` variable is intended to track the total amount of wrapped ether recorded in the `wrapped` mapping. The goal here is to prove that at all reachable states, this is indeed the case. We use the same *Safe* predicate as for the crowdfunding smart contract.

`Definition` *Safe P* :=
  ∀ *state s l*, *ReachableFromBy initial_state state s l* → *P state*.

The property, *total_supply_tracks_correctly*, declares that the `totalSupply` variable equals the sum of the values in the `wrapped` mapping. We also require the property that all values in the wrapped mapping are non-negative as a sanity check required due to the way unsigned integers are handled in DeepSEA.

`Definition` *total_supply_tracks_correctly state* :=
  *sum* (*ERC20WrappedEth_wrapped* (*contract_state state*))
    = (*ERC20WrappedEth__totalSupply* (*contract_state state*))
    ∧ (∀ *key value*, *get_default* 0 *key*
    (*ERC20WrappedEth_wrapped* (*contract_state state*))
      = *value* → (*value* ≥ 0)).

Now we are ready to prove the theorem. Below is an outline of the proof. The full proof is available in Appendix C.6.

```
Theorem total_supply_correct : Safe total_supply_tracks_correctly.
Proof.

unfold Safe. intros.
```

```
H: ReachableFromBy initial_state state s l
________________________________________
total_supply_tracks_correctly state
```

We proceed by induction on the reachability predicate *H*.

```
induction H.
-
```

We prove the base case by unfolding the initial state and simplifying.

```
  unfold total_supply_tracks_correctly.
  unfold initial_state. simpl.
  split.
  * unfold sum. unfold empty. unfold Int256Tree.fold1. simpl.
```

```
    reflexivity.
  * intros. unfold get_default in H. rewrite gempty in H.
    lia.
-
```

We then perform case analysis on the twelve possible transitions shown in Figure 6.1.

```
Hlinks.
destruct prev; autounfold in *; simpl in *.
destruct Step_action0; simpl in *.
```

This is the case for `totalSupply`, which does not alter the `wrapped` mapping or `totalSupply` variable.

```
+ Transparent ERC20WrappedEth_totalSupply_opt.
  unfold ERC20WrappedEth_totalSupply_opt in case_totalSupply_prf.
  ds_inv; subst; simpl in *.
  inversion case_totalSupply_prf.
  unfold total_supply_tracks_correctly.
  simpl.
  unfold total_supply_tracks_correctly in IHReachableFromBy.
  split.
  * apply IHReachableFromBy.
  * destruct IHReachableFromBy as
        [IHReachableFromByCorollary1 IHReachableFromByCorollary2].
    apply IHReachableFromByCorollary2.
```

This is the case for `balanceOf`, which does not alter the wrapped mapping. The proof is identical to the previous case and is omitted for brevity.

```
+ Transparent ERC20WrappedEth_balanceOf_opt.
  abbreviated.
```

We then have the `transfer` function. Here, we will show the heart of the proof. The remainder of the proof is available in Appendix C.6.

```
+ Transparent ERC20WrappedEth_transfer_opt.
    --
```

```
______________________________________
sum
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)) - _value)
     (set _to
        (get_default 0 _to
           (ERC20WrappedEth_wrapped
              (contract_state Step_state0)) + _value)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)))) =
sum
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0))
```

The heart of this proof is the application of the *constant_sum'* lemma, which shows the following:

```
Check Int256Tree_Properties.constant_sum'.
```

```
constant_sum'
     : forall (m : t Z)
         (k : elt)
         (v : Z) (k' : elt)
         (v' i : Z),
       get_default 0 k m = v ->
       get_default 0 k' m = v' ->
       k <> k' ->
       sum (set k (v - i) (set k' (v' + i) m)) = sum m
```

A careful examination of the goal will show that this lemma is appropriate for this situation.

```
apply Int256Tree_Properties.constant_sum'; try reflexivity.
```

```
H9: caller context <> _to
______________________________________
caller context <> _to
```

After dismissing most of the requirements of *constant_sum'* with the `reflexivity` tactic, we dismiss the final assumption based on *H9* which comes from the requirement that you cannot transfer to yourself.

```
assumption.
```

The `transferFrom` case is identical to `transfer` case except that instead of (*caller* `context`) we have *_from*. The proof is omitted for brevity and is available in Appendix C.6.

```
  + Transparent ERC20WrappedEth_transferFrom_opt.
    abbreviated.
```

As with the `balanceOf` and `totalSupply` functions, the case for the `allowance` function does not alter any fields relevant to this lemma and its proof (omitted here) is identical to the proofs for those functions.

```
  + Transparent ERC20WrappedEth_allowance_opt.
    abbreviated.
```

The proofs for `approve` and `approveSafely` are very similar to previous cases because they also do not alter variables relevant to this theorem.

```
  + Transparent ERC20WrappedEth_approve_opt.
    abbreviated.
  + Transparent ERC20WrappedEth_approveSafely_opt.
    abbreviated.
```

Now for the `mint` and `burn` functions, which do alter the `totalSupply` variable. These functions are central to whether this theorem holds or not. The heart of this case is the update to the `totalSupply` variable. This part of the proof is shown below, the remainder is available in Appendix C.6.

```
    + Transparent ERC20WrappedEth_mint_opt.
      -- unfold sum.
```

```
________________________________________________
fold1 Z.add
  (set (caller context)
     (get_default 0
        (caller context)
        (ERC20WrappedEth_wrapped
           (contract_state Step_state0)) +
      callvalue context)
     (ERC20WrappedEth_wrapped
        (contract_state Step_state0))) 0 =
ERC20WrappedEth__totalSupply
  (contract_state Step_state0) +
callvalue context
```

Here we can see that *sum* is defined as a kind of `fold` using the function *Z.add* over the mapping with an initial value of zero.

```
        Print sum.
```

```
sum = fun m : t Z => fold1 Z.add m 0
     : t Z -> Z
```

Now consider the lemma *sum_set_x_minus_from_arbitrary_init*, discussed as a part of the proof themes in Chapter 7.

```
        Check sum_set_x_minus_from_arbitrary_init.
```

```
sum_set_x_minus_from_arbitrary_init
     : forall (k : elt) (m : t Z) (v x init : Z),
       get_default 0 k m = v ->
       fold1 Z.add (set k x m) init =
       fold1 Z.add m init + (x - v)
```

When summing a list, if we replace one element, $v$, with another, the sum of the list is now equal to the original sum plus the new element and minus the replaced element, $v$. This is essentially the heart of this proof, except that here the new element equals the old element, $v$, plus the value of the callvalue. So the resulting sum is the original sum plus $v$ plus the callvalue minus $v$. This simplifies to just the original sum plus the callvalue.

Also, we are not dealing with a list but rather the values of the Coq representation of the DeepSEA `mapping` type. Nevertheless, the idea of the proof remains the same at a high level.

```
        rewrite sum_set_x_minus_from_arbitrary_init with
          (v:=(get_default 0 (caller context)
            (ERC20WrappedEth_wrapped
               (contract_state Step_state0)))) by reflexivity.
```

```
____________________________________________________________
fold1 Z.add
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0)) 0 +
(get_default 0 (caller context)
   (ERC20WrappedEth_wrapped
      (contract_state Step_state0)) +
 callvalue context -
 get_default 0 (caller context)
   (ERC20WrappedEth_wrapped
      (contract_state Step_state0))) =
ERC20WrappedEth__totalSupply
  (contract_state Step_state0) +
callvalue context
```

This is clearer if we save the value of the caller's entry in the wrapped mapping from beforehand as $v$ and fold the occurrences of *sum*.

```
        remember((get_default 0 (caller context)
          (ERC20WrappedEth_wrapped
            (contract_state Step_state0)))) as v.
        fold (sum (ERC20WrappedEth_wrapped (contract_state Step_state0))).
```

```
________________________________________
sum
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0)) +
(v + callvalue context - v) =
ERC20WrappedEth__totalSupply
  (contract_state Step_state0) +
callvalue context
```

Now we can see that the left-hand side corresponds to a more useful form equal to the sum of the wrapped mapping after the callvalue is added to the entry for the caller in the mapping.

```
        rewrite <- IHReachableFromByCorollary1.
```

```
________________________________________
sum
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0)) +
(v + callvalue context - v) =
sum
  (ERC20WrappedEth_wrapped
     (contract_state Step_state0)) +
callvalue context
```

After rewriting the `totalSupply` variable's value based on the inductive hypothesis, it is clear that the left and right hand sides evaluate to the same value. We can show this using the linear integer arithmetic tactic.

```
        lia.
```

The other branch of the proof, showing that all values in the `wrapped` mapping are nonnegative, follows from proof techniques that have been demonstrated already in this thesis and is omitted for brevity. It is available in Appendix C.6.

```
      -- abbreviated.
    + Transparent ERC20WrappedEth_burn_opt.
        lia.
```

The final three proof-goals correspond to the actions for time passing, a balance transfer not originating from the contract, and the revert case. Since none of these actions affect the `wrapped` mapping, they all follow from the inductive hypothesis.

```
  + rewrite <- HS. unfold total_supply_tracks_correctly.
      simpl. apply IHReachableFromBy.
  + rewrite <- HS. unfold total_supply_tracks_correctly.
      simpl. apply IHReachableFromBy.
  + rewrite <- HS. unfold total_supply_tracks_correctly.
```

```
      simpl. apply IHReachableFromBy.

Qed.
```

As a corollary to the above theorem, we can now easily show that, as a sanity check, in all reachable states the `wrapped` mapping holds only nonnegative values. First we define the property we are trying to show is a safety property.

`Definition` $balances\_positive\ state :=$
$(\forall\ key\ value,\ get\_default\ 0\ key$
$(ERC20WrappedEth\_wrapped\ (contract\_state\ state))$
$= value \rightarrow (value \geq 0)).$

```
Theorem balances_always_positive : Safe balances_positive.
Proof.

unfold Safe.

intros.

pose proof (total_supply_correct state s l H).
```

```
H0: total_supply_tracks_correctly state
________________________________________
balances_positive state
```

```
unfold balances_positive, total_supply_tracks_correctly in *.

destruct H0 as [H0 H1].
```

```
H0: sum
    (ERC20WrappedEth_wrapped (contract_state state)) =
    ERC20WrappedEth__totalSupply
    (contract_state state)
H1: forall (key : elt) (value : Z),
    get_default 0 key
    (ERC20WrappedEth_wrapped (contract_state state)) =
    value -> value >= 0
________________________________________
forall (key : elt) (value : Z),
get_default 0 key
  (ERC20WrappedEth_wrapped (contract_state state)) =
value -> value >= 0
```

Here, we have extracted the evidence of the property *balances_positive* from the previous theorem *total_supply_correct*. We can also see the other conjunct from that theorem as *H0*.

```
assumption.

Qed.
```

## 6.8 Sufficient Funds Safe Theorem

Finally, we will discuss a theorem analogous to the *sufficient_funds_safe* theorem for the crowdfunding smart contract. This time, the proof builds upon the *total_supply_correct* theorem, so we will be able to avoid reasoning directly about the Coq representation of the DeepSEA `mapping` type.

Here is the invariant we aim to prove as a safety property. Having proved the *balances_always_positive* theorem, we omit that condition for our proof goal here.

`Definition` *balance_backed state* :=
  *sum* (*ERC20WrappedEth_wrapped* (*contract_state state*))
    ≤ (*balance state contract_address*).

Now we aim to prove that *balance_backed* is a safety property for the ERC-20 wrapped ether smart contract. A proof of this theorem gives us confidence that the smart contract always has a sufficient balance to refund every user. It eliminates the possibility that someone has written a back-door to siphon away funds. However, this theorem alone does not prevent funds from being forever locked in the contract. To prove such a property requires a theorem similar to the *can_claim_back* theorem shown for the crowdfunding smart contract in Section 5.8.

Here, we will focus only on the heart of the proof. The full proof is available in Appendix C.7.

```
Theorem sufficient_funds_safe : Safe balance_backed.
Proof.

induction H.
- simpl.
```

```
snapshot_balances_valid_prf: forall a : addr,
                             0 <=
                             snapshot_balances a
                             <
                             Int256.modulus
-----------------------------------------------------------------------
0 <= snapshot_balances contract_address
```

In the base case, we rely upon the assumption made that the snapshot balances are in a valid range as a part of a snapshot approach described in Section 4.2.1.

```
  apply snapshot_balances_valid_prf.
- abbreviated.
  + Transparent ERC20WrappedEth_totalSupply_opt.
    abbreviated.
  + Transparent ERC20WrappedEth_balanceOf_opt.
    abbreviated.
  + Transparent ERC20WrappedEth_transfer_opt.
    abbreviated.
  + Transparent ERC20WrappedEth_transferFrom_opt.
```

```
    abbreviated.
  + Transparent ERC20WrappedEth_allowance_opt.
    abbreviated.
  + Transparent ERC20WrappedEth_approve_opt.
    abbreviated.
  + Transparent ERC20WrappedEth_approveSafely_opt.
    abbreviated.
  + Transparent ERC20WrappedEth_mint_opt.
    abbreviated.
    rewrite H2.
```

```
IHReachableFromBy: ERC20WrappedEth__totalSupply
                   (contract_state Step_state0) <=
                   balance Step_state0
                   contract_address
H6: 0 < callvalue context
------------------------------------------------------------
ERC20WrappedEth__totalSupply
  (contract_state Step_state0) +
callvalue context <=
balance Step_state0 contract_address +
callvalue context
```

Here we see that in the successful case of minting a wrapped ether token, the inequality increases on both sides by the callvalue. As a result the *balance_backed* property is maintained.

```
    lia.
  + Transparent ERC20WrappedEth_burn_opt.
    abbreviated.
      rewrite H6.
```

```
IHReachableFromBy: ERC20WrappedEth__totalSupply
                   (contract_state Step_state0) <=
                   balance Step_state0
                   contract_address
H2: 0 <= _value
------------------------------------------------------------
ERC20WrappedEth__totalSupply
  (contract_state Step_state0) - _value <=
balance Step_state0 contract_address - _value
```

For the `burn` case, this time the inequality is reduced on the left and right by the argument `_value` and so the property is maintained.

```
      lia.
    * exfalso.
```

```
Heqb: (Int256.zero =? Int256.one)%int256 = true
------------------------------------------------
False
```

We also have the scenario where the transfer fails, which we assume not to be the case as part of the successful-calls approach discussed in Section 4.2.2. This case is dismissed by the **inversion** tactic and the contradictory knowledge that the **transferEth** function returned a zero when it was assumed to return a one in order for the function to succeed. The aim of this theorem is to show that in all reachable states, the *balance_backed* property is maintained. It would be a separate proof goal to show that a user can successfully withdraw funds and the proof would be similar to the *can_claim_back* theorem discussed for the crowdfunding smart contract in Section 5.8.

```
    inversion Heqb.
```

Next is the case for external balance transfers, which can only increase the balance, so the property continues to hold.

```
  + unfold current_balances, update_balances.
    abbreviated.
      rewrite <- Case.
```

```
H: amount >= 0
IHReachableFromBy: ERC20WrappedEth__totalSupply
                   (contract_state Step_state0) <=
                   balance Step_state0
                   contract_address
------------------------------------------------
ERC20WrappedEth__totalSupply
  (contract_state Step_state0) <=
balance Step_state0 contract_address + amount
```

Here we see the case where some *amount* of ether has been transferred to the smart contract, increasing its balance, but the property is maintained.

```
      lia.
```

Finally, we have the two cases for time passing and for reverting. Neither affect balances or the **wrapped** mapping so they follow from the inductive hypothesis.

```
  + rewrite HS in *. apply IHReachableFromBy.
  + rewrite HS in *. apply IHReachableFromBy.
```

```
Qed.
```

## 6.9 Unsigned Integer Side-Conditions

As with the crowdfunding case study, there are side-conditions generated automatically by the DeepSEA system which should be proved to hold. Unlike the crowdfunding smart contract, the ERC-20 wrapped ether smart contract does perform arithmetic on values represented by the Coq type $Z$. As a result, when this occurs the proof goals will not be dismissed automatically by the *code_proofs_auto* tactic shown again below.

```
Ltac code_proofs_auto :=
    intros;
    unfold synth_func_obligation;
    repeat (split; auto).
```

We will explore the main side-condition subgoals for the non-trivial subgoals in the nine smart contract functions. Each subgoal that needs to be proved in a non-trivial manner corresponds to a line in the smart contract implementation (discussed in Section 6.1). These subgoals have extra hypotheses added compared to what DeepSEA generated, corresponding to the assumption that that no-one will be able to mint close to $10^{38}$ wei worth of the token (equivalent to $10^{20}$ ether), nor will all users be able to collectively. This is reasonable to assume because $10^{20}$ ether far exceeds the current amount of ether in circulation of around $1.2 \times 10^{8}$ [149] and since this token wraps ether at a $1 : 1$ ratio. We also add an assumption to the subgoal for the `burn` function to make available the knowledge proven in the *total_supply_correct* theorem from Section 6.7. Future work would involve fully integrating the blockchain model (described in Section 4.2) with the generation of these side-conditions so that explicitly adding that assumption would not be necessary. We also show the proof state at the point in which the correspondence between the line of code causing the side-condition proof and the proof state is most clear.

The constructor and the functions `totalSupply`, `balanceOf`, `allowance`, `approve` and `approveSafely` do not perform any arithmetic on values of type $Z$ and so the tactic *code_proofs_auto* completes their proofs. All the proofs are available in Appendix C.8, however here we are only concerned with the functions which do perform arithmetic on values of type $Z$: `transfer`, `transferFrom`, `mint` and `burn`.

First, we define a helper function, *toZ*, which is necessary due to the machinery used for handling types in DeepSEA. It is useful for demonstrating that *unpair_ft* (*tint_Z_bounded Int256.modulus*) and $Z$ are the same type. This allows us to phrase the added hypotheses more naturally.

```
Definition toZ : unpair_ft (tint_Z_bounded Int256.modulus) → Z := id.
```

We also define a large value as an *unattainableWeiAmount*, equal to $10^{20}$ ether.

```
Definition etherToWei (ether : Z) : Z := ether × (10 ^ 18).
Definition unattainableWeiAmount := etherToWei (10 ^ 20).
```

Now we consider the `transfer` function. Here again is the definition of the `transfer` function in DeepSEA.

Listing 6.14: ERC-20 Wrapped Ether Contract Source Code - Part 6 (recap)

```
let transfer(_to, _value) =
  assert(_value >= 0);
  assert(msg_sender <> this_address);
  assert(msg_sender <> _to);
  assert(msg_value = 0);
  let wrapped_amount_from = wrapped[msg_sender] in
  let wrapped_amount_to = wrapped[_to] in
  assert(wrapped_amount_from >= _value); (* Continued over page *)
```

```
9    wrapped[_to] := wrapped_amount_to + _value;
10   wrapped[msg_sender] := wrapped_amount_from - _value;
11   emit Transfer(msg_sender, _to, _value);
12   true
```

On lines 9 and 10 we perform addition and subtraction, respectively, on values represented in Coq by the *Z* type.

```
Lemma ERC20WrappedEth_transfer_vc a0 a1 me d :
    (forall a,
      Int256Tree.get_default 0 a (ERC20WrappedEth_wrapped d)
        < unattainableWeiAmount) ->
    (toZ a1 < unattainableWeiAmount) ->
    ht_ft_cond a0 -> ht_valid_ft_cond a0 ->
    ht_ft_cond a1 -> ht_valid_ft_cond a1 ->
    high_level_invariant d ->
    synth_func_cond ERC20WrappedEth_transfer ERC20WrappedEth_transfer_wf
                    a0 a1 me d.
Proof.
```

We begin these proofs by introducing the assumptions as hypotheses. These correspond to what is assumed to hold for the arguments to the transfer function. *a0* corresponds to the *_to* address and *a1* to the *_value* provided. The assumptions *Hextra1* and *Hextra2* have been added, related to the specified amounts being less than the *unattainableWeiAmount*.

```
intros Hextra1 Hextra2.
intros.
```

```
memModelOps: MemoryModelOps mem
HCompatDataOps: CompatDataOps
                global_abstract_data_type
a0: unpair_ft _tp_type_pair
a1: unpair_ft (tint_Z_bounded Int256.modulus)
me: MachineModel.machine_env GetHighData
d: global_abstract_data_type
Hextra1: forall a : Int256Tree.elt,
        Int256Tree.get_default 0 a
        (ERC20WrappedEth_wrapped d) <
        unattainableWeiAmount
Hextra2: toZ a1 < unattainableWeiAmount
H: ht_ft_cond a0
H0: ht_valid_ft_cond a0
H1: ht_ft_cond a1
H2: ht_valid_ft_cond a1
H3: high_level_invariant d
____________________________________________
synth_func_cond ERC20WrappedEth_transfer
  ERC20WrappedEth_transfer_wf a0 a1 me d
```

```
unfold _tp_type_pair in *.
unfold unpair_ft in *.
unfold _tp_type_pair in *.
simpl in *.
unfold Z32 in a1.
remember a0 as _to. remember a1 as _value.
unfold Z32 in _value.
simpl in *.
```

```
_value: Z
H1:-1 < _value /\ Int256.modulus > _value
_to: int256
____________________________________________
synth_func_cond ERC20WrappedEth_transfer
  ERC20WrappedEth_transfer_wf _to _value me d
```

We then unfold the side-condition goal.

```
unfold synth_func_cond.
simpl.
```

After some simplification and splitting of conjunctions we are left with two subgoals.

```
repeat split.
```

```
─────────────────────────────────────────
v0 +
SpecTree.get 12
  (SpecTree.set
     (Pos.succ
        (FC_param_ident_start ERC20WrappedEth_transfer))
     int_Z32_pair _value
     (SpecTree.set
        (FC_param_ident_start ERC20WrappedEth_transfer)
        int_U_pair _to SpecTree.empty)) <
Int256.modulus
```

```
─────────────────────────────────────────
v >=
SpecTree.get 12
  (SpecTree.set
     (Pos.succ
        (FC_param_ident_start ERC20WrappedEth_transfer))
     int_Z32_pair _value
     (SpecTree.set
        (FC_param_ident_start ERC20WrappedEth_transfer)
        int_U_pair _to SpecTree.empty))
```

The next subgoal corresponds to line 9 of Listing 6.14.

```
- subst.
  simpl in *.
```

After some simplification, we are required to prove the following goal. This corresponds to showing that the new entry for the recipient in the `wrapped` mapping does not overflow due to receiving a transfer.

```
  inversion H9.
```

```
─────────────────────────────────────────
Int256Tree.get_default 0 _to
  (ERC20WrappedEth_wrapped m5) + _value <
Int256.modulus
```

```
  inversion H8.
  destruct(MachineModel.me_callvalue me =? 0) eqn:Case; [|simpl in H7; inversion H7].
  simpl in H7. inversion H7.
  destruct(Int256.eq (MachineModel.me_caller me) _to) eqn:SCase;
    [simpl in H6; inversion H6|].
  simpl in H6. inversion H6.
  destruct(Int256.eq (MachineModel.me_caller me)
  (MachineModel.me_address me)) eqn:SSCase; [simpl in H5; inversion H5|].
```

```
simpl in H5. inversion H5.
destruct(_value >=? 0) eqn:SSSCase; [|simpl in H4; inversion H4].
simpl in H4. inversion H4.
subst.
```

Here we make use of our assumption that no-one has more than $10^{20}$ of wrapped ether.

```
pose proof (Hextra1 _to). subst.
unfold Int256.modulus, two_power_nat. simpl.
unfold toZ, id in Hextra2.
clear -Hextra2 H11.
unfold Z_bounded in *.
remember (@Int256Tree.get_default Z Z0 _to (ERC20WrappedEth_wrapped m5)) as y.
unfold unattainableWeiAmount, etherToWei in *.
lia.
```

The next subgoal corresponds to line 10 of Listing 6.14. Here we are trying to show that there is no underflow when the sender has the transferred value deducted from their balance as recorded in the `wrapped` mapping. First we rename the right-hand side of the inequality to *STORED_VALUE* for ease of reading.

```
-
```

```
______________________________________
v >=
SpecTree.get 12
  (SpecTree.set
     (Pos.succ
        (FC_param_ident_start ERC20WrappedEth_transfer))
     int_Z32_pair _value
     (SpecTree.set
        (FC_param_ident_start ERC20WrappedEth_transfer)
        int_U_pair _to SpecTree.empty))
```

```
abbreviated.
```

```
H10: match
    StateMonad.runStateT
    (MonadZero.guard (v >=? STORED_VALUE)) m5
    with
    | Some v => Some (snd v)
    | None => None
    end = Some m6
______________________________________
v >= STORED_VALUE
```

Now we can see that *H10* implies our goal. *H10* is from line 8 of Listing 6.14. We dismiss the branch where *Some _ = None* with the tactic `inversion` *H10*.

```
  destruct (v >=? STORED_VALUE) eqn:Case; [|inversion H10]. simpl in *.
```

```
Case: (v >=? STORED_VALUE) = true
H10: Some m5 = Some m6
______________________________________
v >= STORED_VALUE
```

```
  apply Z.geb_le in Case.
```

```
Case: STORED_VALUE <= v
______________________________________
v >= STORED_VALUE
```

Now our goal is solvable by the linear integer arithmetic tactic.

```
  lia.
Qed.
```

The proof for the `transferFrom` function is similar, except that we have the additional side-condition relating to the deduction from the allowance of the sender. The definition of `transferFrom` is repeated here, as follows.

Listing 6.15: ERC-20 Wrapped Ether Contract Source Code - Part 7 (recap)

```
let transferFrom(_from, _to, _value) =
  assert(_value >= 0);
  assert(_from <> this_address);
  assert(_from <> _to);
  assert(msg_value = 0);
  let approved_amount = allowances[_from][msg_sender] in
  assert(approved_amount >= _value);
  allowances[_from][msg_sender] := approved_amount - _value;
  let wrapped_amount_from = wrapped[_from] in
  let wrapped_amount_to = wrapped[_to] in
  assert(wrapped_amount_from >= _value);
  wrapped[_to] := wrapped_amount_to + _value;
  wrapped[_from] := wrapped_amount_from - _value;
  emit Transfer(_from, _to, _value);
  true
```

The remaining proofs have a similar structure to the proof for the `transfer` function, so we will focus on the point in each proof where the cases correspond to the line of code performing arithmetic in each function.

```
Lemma ERC20WrappedEth_transferFrom_vc a0 a1 a2 me d :
    (forall a,
      Int256Tree.get_default 0 a (ERC20WrappedEth_wrapped d)
        < unattainableWeiAmount) ->
    (toZ a2 < unattainableWeiAmount) ->
    ht_ft_cond a0 -> ht_valid_ft_cond a0 ->
    ht_ft_cond a1 -> ht_valid_ft_cond a1 ->
    ht_ft_cond a2 -> ht_valid_ft_cond a2 ->
    high_level_invariant d ->
    synth_func_cond ERC20WrappedEth_transferFrom ERC20WrappedEth_transferFrom_wf
                    a0 a1 a2 me d.
Proof.
```

This subgoal corresponds to the deduction in the approved amount of the spender in line 8 of Listing 6.15. Similarly to `transfer` we have two additional assumptions, *Hextra1* and *Hextra2*.

```
intros Hextra1 Hextra2.
intros.
unfold synth_func_cond.
simpl.
repeat split.
-
  abbreviated.
```

```
H11: match
    StateMonad.runStateT
    (MonadZero.guard (v >=? STORED_VALUE)) m4
    with
    | Some v => Some (snd v)
    | None => None
    end = Some m5
--------------------------------------------
v >= STORED_VALUE
```

```
  destruct(v >=? STORED_VALUE) eqn:Case; [|inversion H11].
  apply Z.geb_le in Case.
```

```
Case: STORED_VALUE <= v
--------------------------------------------
v >= STORED_VALUE
```

Here we prove the subgoal using knowledge from *H11*, which is based on line 7 of Listing 6.15.

```
lia.
```

The next subgoal corresponds to line 12 of Listing 6.15 relating to potential overflow.

```
-
simpl in H15.
subst. simpl.
unfold ht_ft_cond in H3.
simpl in H3.
inversion H14.
simpl in *. unfold Z32 in a2.
```

```
--------------------------------------------
Int256Tree.get_default 0 a1
  (ERC20WrappedEth_wrapped m8) + a2 < Int256.modulus
```

```
inversion H13.
inversion H12.
destruct(v >=? a2) eqn:SSSCase; [|simpl in H11; inversion H11].
inversion H11.
inversion H10.
destruct(MachineModel.me_callvalue me =? 0) eqn:Case; [|simpl in H9; inversion H9].
inversion H9.
destruct(a2 >=? 0) eqn:SCase; [|simpl in H6; inversion H6].
inversion H6.
destruct(Int256.eq a0 a1) eqn:SSCase; [simpl in H8; inversion H8|].
inversion H8.
destruct(Int256.eq a0 (MachineModel.me_address me)) eqn:SSSSCase;
  [simpl in H7; inversion H7|].
inversion H7.
subst.
destruct m5. simpl in *.
clear -Hextra2 Hextra1.
unfold toZ, id in Hextra2.
```

Here we make use of the assumption that no-one has a wrapped ether balance greater than $10^{20}$ ether.

```
pose proof (Hextra1 a1) as H.
```

```
Hextra1: forall a : Int256Tree.elt,
         Int256Tree.get_default 0 a
         ERC20WrappedEth_wrapped <
         unattainableWeiAmount
Hextra2: a2 < unattainableWeiAmount
H: Int256Tree.get_default 0 a1
  ERC20WrappedEth_wrapped < unattainableWeiAmount
________________________________________________________
Int256Tree.get_default 0 a1 ERC20WrappedEth_wrapped +
a2 < Int256.modulus
```

```
unfold Int256.modulus, two_power_nat. simpl.
unfold Z_bounded in *.
unfold unattainableWeiAmount, etherToWei in *.
lia.
```

The final subgoal for the `transferFrom` function corresponds to line 13 of Listing 6.15. It relates to the potential underflow when making a transfer which is guaranteed to never happen due to the assertion in line 11.

```
-
abbreviated.
destruct(v0 >=? STORED_VALUE) eqn:Case; [|inversion H15].
```

```
inversion H13.
rewrite Z.geb_le in Case.
rewrite <- H18 in Case.
rewrite <- H19.
clear -Case.
```

```
Case: STORED_VALUE <=
     Int256Tree.get_default 0 a0
     (ERC20WrappedEth_wrapped m6)
--------------------------------------------------
Int256Tree.get_default 0 a0
  (ERC20WrappedEth_wrapped m6) >= STORED_VALUE
```

```
lia.
Qed.
```

The `mint` function naturally has subgoals relating to overflow. Both these subgoals rely on the implicit assumption that no-one has an amount of ether to mint and amount of tokens close to the maximum 256-bit value. Here we also assume that not only does an individual not have that much ether but not even the value of everyone's combined tokens as recorded in the `totalSupply` variable will reach a sum of $10^{20}$ ether.

Listing 6.16: ERC-20 Wrapped Ether Contract Source Code - Part 11 (recap)

```
 let mint () =
   assert(msg_sender <> this_address);
   assert(msg_value > 0);

   let wrapped_amount = wrapped[msg_sender] in
   wrapped[msg_sender] := wrapped_amount + msg_value;
   let prev_totalSupply = _totalSupply in
   _totalSupply := prev_totalSupply + msg_value;
   emit Transfer(address(0x0), msg_sender, msg_value);
   true
```

```
Lemma ERC20WrappedEth_mint_vc me d :
    (forall a,
      Int256Tree.get_default 0 a (ERC20WrappedEth_wrapped d)
        < unattainableWeiAmount) ->
    (MachineModel.me_callvalue me < unattainableWeiAmount) ->
    (ERC20WrappedEth__totalSupply d < unattainableWeiAmount) ->
    high_level_invariant d ->
    synth_func_cond ERC20WrappedEth_mint ERC20WrappedEth_mint_wf
                    me d.
Proof.
intros Hextra1 Hextra2 Hextra3.
intros.
unfold synth_func_cond.
```

```
simpl.
repeat split.

-
inversion H2.
```

```
────────────────────────────────────────
Int256Tree.get_default 0
  (MachineModel.me_caller me)
  (ERC20WrappedEth_wrapped m2) +
MachineModel.me_callvalue me < Int256.modulus
```

This subgoal corresponds to the overflow condition on minting adding value to the caller's token balance. This is line 6 of Listing 6.16. Here we again make use of the assumption that no-one has a wrapped ether balance greater than $10^{20}$ ether.

```
destruct (Int256.eq
(@MachineModel.me_caller
   global_abstract_data_type
   me)
(@MachineModel.me_address
   global_abstract_data_type
   me)) eqn:Case; [simpl in H0; inversion H0|].
inversion H0.
destruct(Z.gtb
(@MachineModel.me_callvalue
   global_abstract_data_type
   me) Z0); [|simpl in H1; inversion H1].
inversion H1.
subst.
pose proof (Hextra1 (MachineModel.me_caller me)) as H3.
```

```
Hextra1: forall a : Int256Tree.elt,
         Int256Tree.get_default 0 a
         (ERC20WrappedEth_wrapped m2) <
         unattainableWeiAmount
Hextra2: MachineModel.me_callvalue me
         <
         unattainableWeiAmount
H3: Int256Tree.get_default 0
    (MachineModel.me_caller me)
    (ERC20WrappedEth_wrapped m2) <
    unattainableWeiAmount
────────────────────────────────────────
Int256Tree.get_default 0
  (MachineModel.me_caller me)
  (ERC20WrappedEth_wrapped m2) +
MachineModel.me_callvalue me < Int256.modulus
```

```
unfold Int256.modulus, two_power_nat. simpl.
clear -Hextra1 Hextra2.
pose proof (Hextra1 (MachineModel.me_caller me)).
unfold Z_bounded in *. simpl in *.
unfold unattainableWeiAmount, etherToWei in *.
lia.
```
-

To solve this subgoal we make use of the assumptions that the `totalSupply` variable and the *callvalue* are not greater than $10^{20}$ ether.

```
inversion H4.
inversion H3.
inversion H2.
destruct(Z.gtb
(@MachineModel.me_callvalue
   global_abstract_data_type
   me) Z0); [|simpl in H1; inversion H1].
inversion H1.
destruct (Int256.eq
(@MachineModel.me_caller
   global_abstract_data_type
   me)
(@MachineModel.me_address
   global_abstract_data_type
   me)) eqn:Case; [simpl in H0; inversion H0|].
inversion H0.
subst.
destruct m2. simpl in *.
clear -Hextra2 Hextra3.
```

```
Hextra2: MachineModel.me_callvalue me
         <
         unattainableWeiAmount
Hextra3: ERC20WrappedEth__totalSupply
         <
         unattainableWeiAmount
______________________________________________
ERC20WrappedEth__totalSupply +
MachineModel.me_callvalue me < Int256.modulus
```

```
unfold Int256.modulus, two_power_nat. simpl.
unfold unattainableWeiAmount, etherToWei in *.
lia.
```

Qed.

Finally, the `burn` function. This function has two subtractions, the first of which is provable using similar techniques to those used in the `transfer` and `transferFrom` functions. The final case corresponds to line 10 in Listing 6.17. We make use of the assumption that the *wrapped_amount* is less than or equal to the *totalSupply*. This knowledge is a consequence of the *total_supply_correct* theorem proved in Section 6.7, but the *total_supply_correct* theorem is not able to be applied here due to limitations in the DeepSEA system. As stated earlier, ideally, the model of the blockchain discussed in Section 4.2 would be integrated into the generation of these side-conditions. This would be valuable future work.

Listing 6.17: ERC-20 Wrapped Ether Contract Source Code - Part 12 (recap)

```
1      let burn (_value) =
2        assert(_value >= 0);
3        assert(msg_sender <> this_address);
4        assert(msg_value = 0);
5
6        let wrapped_amount = wrapped[msg_sender] in
7        assert(wrapped_amount >= _value);
8        wrapped[msg_sender] := wrapped_amount - _value;
9        let prev_totalSupply = _totalSupply in
10       _totalSupply := prev_totalSupply - _value;
11       transferEth(msg_sender, _value);
12       emit Transfer(msg_sender, address(0x0), _value);
13       true
14   }
```

```
Lemma ERC20WrappedEth_burn_vc a0 me d :
   (forall (a:Int256Tree.elt),
      ERC20WrappedEth__totalSupply d
      >=
      Int256Tree.get_default 0 a (ERC20WrappedEth_wrapped d))
   ->
    ht_ft_cond a0 -> ht_valid_ft_cond a0 ->
    high_level_invariant d ->
    synth_func_cond ERC20WrappedEth_burn ERC20WrappedEth_burn_wf
                    a0 me d.
Proof.

intros Hextra1.

intros.

unfold synth_func_cond.

simpl in *.
repeat split.
```

This branch corresponds to line 8 of Listing 6.17, which is provable due to the assertion in line 7.

```
- abbreviated.

destruct (v >=? STORED_VALUE) eqn:Case; [|inversion H6].

rewrite Z.geb_le in Case.
```

```
Case: STORED_VALUE <= v
________________________________________
v >= STORED_VALUE
```

```
clear -Case.
lia.
```

The final case corresponds to line 10 in Listing 6.17. The proof of this case relies on the assumption *Hextra1*.

```
- simpl in *. inversion H8.
destruct(v >=? a0) eqn:Case; simpl in *; [|discriminate].
rewrite Z.geb_le in Case.
inversion H5. subst.
abbreviated.
inversion H7.
inversion H6.
destruct(Z.eqb
(@MachineModel.me_callvalue
   global_abstract_data_type
   me) Z0); [|simpl in H4; inversion H4].
inversion H4.
destruct (Int256.eq
(@MachineModel.me_caller
   global_abstract_data_type
   me)
(@MachineModel.me_address
   global_abstract_data_type
   me)) eqn:SCase; [simpl in H3; inversion H3|].
inversion H3.
destruct (a0 >=? 0); [|simpl in H2; inversion H2].
inversion H2.
subst.
clear -Hextra1 Case.
destruct m4. simpl in *.
pose proof (Hextra1 (MachineModel.me_caller me)) as H.
unfold Z_bounded in *. simpl in *.
```

```
Hextra1: forall a : Int256Tree.elt,
        ERC20WrappedEth__totalSupply
        >=
        Int256Tree.get_default 0 a
        ERC20WrappedEth_wrapped
Case: a0 <=
     Int256Tree.get_default 0
     (MachineModel.me_caller me)
     ERC20WrappedEth_wrapped
H: ERC20WrappedEth__totalSupply
  >=
  Int256Tree.get_default 0
  (MachineModel.me_caller me)
  ERC20WrappedEth_wrapped
______________________________________
ERC20WrappedEth__totalSupply >= a0
lia.
Qed.
```

## 6.10 Formal Verification Effort

The development effort in person-hours for this ERC-20 wrapped ether smart contract with its proofs is shown in Table 6.1 and Table 6.2, in total taking about 24.5 hours. The effort is split into the tasks of implementing, modelling, specifying, proving, and other. Tasks falling into the "other" category involved getting familiar with the ERC-20 specification as well as exploratory proof efforts that ended up not being included. For each of the four main theorems as well as for the overflow and underflow safety lemmas, the effort for each is broken down into the time it took to write the specification for the theorem and how long the proof took. The time taken to write the commentary included in this thesis is not included.

Table 6.1: Person-hours for specifying and proving the ERC-20 wrapped ether theorems

| **Theorem name** | **Specifying** | **Proving** | **Total** |
|---|---|---|---|
| **`wrapped_preserved`** | 0.8h | 3.5h | 4.3h |
| **`transfer_correct`** | 0.1h | 0.4h | 0.5h |
| **`total_supply_correct`** | 0.6h | 9.3h | 10h |
| **`sufficient_funds_safe`** | 0.1h | | |
| **Overflow/underflow safety** | 1.8h | 4.6h | 6.4h |
| **Total** | 3.4h | 17.8h | **21.2h** |

Table 6.2: Person-hours for all development of the ERC-20 wrapped ether smart contract

| **Implementing** | **Modelling** | **Specifying** | **Proving** | **Other** | **Total** |
|---|---|---|---|---|---|
| 1.8h | 0.2h | 3.4h | 17.8h | 1.3h | **24.5h** |

While useful as a rough indictator, this information has very limited generality as it is based on the times recorded by myself only. Future work in understanding proof effort and the benefits DeepSEA may provide in this regard, could involve a group of experts being introduced to DeepSEA and then being timed carrying out proofs and writing specifications. A major benefit of the DeepSEA system, in terms of proof effort, is that no effort is spent on bytecode verfication, since DeepSEA's verified compilation removes the need for this. In comparison, 71% of the proof effort for the verification of the Ethereum 2.0 deposit contract [109] was spent on bytecode verification.

The main Coq file (`ERC20WrappedEth.v`) for the ERC-20 wrapped ether smart contract proofs contains 536 lines which relate to specifications and 745 lines which relate to proofs, as counted by the `coqwc` utility. In contrast, for the main file for the crowdfunding case study (`Crowdfunding.v`) from Chapter 5, there are 399 lines which relate to specification and 733 lines which relate to proofs. These files and those they depend on are available as described in Appendix A. We also note that the size of the generated bytecode for the crowdfunding smart contract and ERC-20 wrapped ether smart contract are 2511 bytes and 5135 bytes, respectively.

## 6.11 Remarks

The ERC-20 wrapped ether case study demonstrates that the approach used to verify properties of the crowdfunding smart contract can easily be adapted to other smart contracts, in particular being applied to a realistic ERC-20 token. It also highlights that where there are similarities between smart contracts, such as having a `backers`/`wrapped` mapping, much can be gained from learning how to prove properties about the original smart contract as it is likely that the lemmas required and the structure of the new proofs will resemble the original proofs. This emphasises the point that the more formal proofs are carried out, the easier new proofs will become due to similarities with existing proofs.

# Chapter 7

# Proof Themes

This chapter highlights the common threads across the proofs discussed in Chapters 5 and 6, across the two smart contracts, across the different proofs and across the branches of proofs.

## 7.1 Representation of Integers

One limitation of DeepSEA is that some integers, such as those representing Ethereum addresses, are represented by the *int256* data type. This means that the proof engineer is required to be familiar both with the lemmas relevant for *int256* as well as for the Coq native *Z* type. In addition, the authors of the *int256* data type would ideally need to prove most, if not all, of the relevant helper lemmas related to the usage of *int256*. The clearest example of this are the similar roles of *Z.eqb_eq* and *Int256eq_true* which are used at multiple points during the proofs. The statements of these lemmas are shown here:

```
Check Z.eqb_eq.
```

```
Z.eqb_eq
     : forall n m : Z, (n =? m) = true <-> n = m
```

```
Check Int256eq_true.
```

```
Int256eq_true
     : forall x y : int256,
       (x =? y)%int256 = true -> x = y
```

There is not a straightforward solution to this concern, because it would not make sense to treat addresses as the *Z* type since arithmetic operations are not supposed to be used on addresses. Nevertheless, greater effort on the standardisation of the names of lemmas would be beneficial for proof engineers because it would make it easier to find similar lemmas for different types.

## 7.2 Converting between Boolean and Propositional Equality

The lemmas shown in Section 7.1 for converting between Boolean equality and propositional equality were commonly used throughout the proofs. This is due to the specification language preferring the propositional phrasing, whereas when dealing with actual program code the Boolean phrasing is necessary. For example, after performing `destruct` on the guard `(a =? (caller context))` in the proof in the *donation_preserved* theorem to simplify the program code, it is necessary to convert that Boolean knowledge to propositional form via the application of the lemma *Int256eq_true* so that it can be used to rewrite other terms in the context.

## 7.3 Generic *Int256Tree* Lemmas

Both case studies involve a mapping with the type for the keys being *int256* and values being of type *Z*, which gives the mapping the type of *Int256Tree.t Z*. This is more difficult to reason about than a plain function. Nevertheless, there is a range of lemmas provided with the *Int256Tree.t* type. Still, to prove the theorems of the case studies the following lemmas were also needed to be proved in addition. These lemmas primarily assist with reasoning about calculations involving the sum of values stored in a `mapping`. The statements of the lemmas are shown here, the full proofs are available in Appendix D.

The lemma *fold_snd_map* assists with reasoning with *fold_left* and `mapping` types.

```
Lemma fold_snd_map :
  ∀ A B (m : list (A × B)) x f,
  (fold_left (fun (a : B) (p : A × B) ⇒ f a (snd p))
    m x) =
  (fold_left f
  (List.map snd m) x).
```

```
Lemma sum_starting_from_init_equals_sum_plus_init_arbitrary_start :
∀ (x init : Z) (m : Int256Tree.t Z),
Int256Tree.fold1 Z.add m (init + x) = Z.add (Int256Tree.fold1 Z.add m x) init.
```

The following lemma is a specialised version of the above lemma.

```
Lemma sum_starting_from_init_equals_sum_plus_init :
∀ (init : Z) (m : Int256Tree.t Z),
Int256Tree.fold1 Z.add m init = Z.add (Int256Tree.fold1 Z.add m 0) init.
```

```
Lemma Int256Tree_sum_set_value_initially_zero :
  ∀ (m: Int256Tree.t Z32) k v, Int256Tree.get_default 0 k m = 0
                       → Int256Tree_Properties.sum (Int256Tree.set k v m) =
                         Int256Tree_Properties.sum m + v.
```

```
Lemma sum_set_y_remove_from_starting_x :
  ∀ k m x y,
  Int256Tree.fold1 Z.add (Int256Tree.set k y m) x =
  Int256Tree.fold1 Z.add (Int256Tree.remove k m) x + y.
```

The next lemma is a specialised version of *sum_set_y_remove_from_starting_x*.

```
Lemma sum_set_zero_remove_from_starting_x :
  ∀ k m x,
  Int256Tree.fold1 Z.add (Int256Tree.set k 0 m) x =
  Int256Tree.fold1 Z.add (Int256Tree.remove k m) x.

Lemma sum_set_zero_remove :
  ∀ k m,
  Int256Tree.fold1 Z.add (Int256Tree.set k 0 m) 0 =
  Int256Tree.fold1 Z.add (Int256Tree.remove k m) 0.

Lemma sum_set_x_minus_from_arbitrary_init :
  ∀ (k : elt) (m : t Z) (v x init : Z),
  get_default 0 k m = v →
  fold1 Z.add (set k x m) init = fold1 Z.add m init + (x - v).
```

This next lemma is a specialised version of *sum_set_x_minus_from_arbitrary_init*.

```
Lemma sum_set_zero_minus_from_arbitrary_init :
  ∀ (k : elt) (m : t Z) (v init : Z),
  get_default 0 k m = v →
  fold1 Z.add (set k 0 m) init = fold1 Z.add m init - v.

Lemma sum_set_zero_minus : ∀ k m v, Int256Tree.get_default 0 k m = v →
    Int256Tree_Properties.sum (Int256Tree.set k 0 m)
  = Int256Tree_Properties.sum m - v.

Lemma Int256Tree_sum_minus_equality :
  ∀ m k x,
    Int256Tree_Properties.sum m ≥ x →
      Int256Tree_Properties.sum (Int256Tree.set k 0 m)
    = (Int256Tree_Properties.sum m) - (Int256Tree.get_default 0 k m).

Lemma Int256Tree_sum_minus_from_starting_x :
  ∀ (m : t Z) (k : elt) (x : Z),
      fold1 Z.add (set k 0 m) x =
      fold1 Z.add m x - get_default 0 k m.

Lemma Int256Tree_sum_minus :
  ∀ m k x,
    Int256Tree_Properties.sum m ≤ x
    →
       Int256Tree_Properties.sum (Int256Tree.set k 0 m)
    ≤ x - (Int256Tree.get_default 0 k m).
```

These lemmas build upon each other to prove the lemmas that are used in the *sufficient_funds_safe* theorem of the crowdfunding smart contract and the *total_supply_correct* theorem of the ERC-20 wrapped ether smart contract. The lemmas explicitly used are: *Int256Tree_sum_minus*, *Int256Tree_sum_set_value_initially_zero* and *sum_set_x_minus_from_arbitrary_init*.

## 7.4 Other Generic Lemmas

Another generic lemma, unrelated to the *int256* mapping data type, is the *addZeroBalance* lemma which is used at multiple points in the *sufficient_funds_safe* theorem and is also used in the *can_claim_back* theorem. It proves that adding zero to a balance leaves the balance function unchanged. The proof is in Appendix D.14.

```
Lemma addZeroBalance : ∀ sender recipient balances a,
  update_balances sender recipient 0 balances a = balances a.
```

When reasoning about balances, it can be useful to know that the model always processes the outgoing transfers left over from a previous smart contract call. In other words, that there are no remaining outgoing transfers in any reachable state. This lemma is used in the *sufficient_funds_safe* theorems of both case studies. The proof of *NoLeftoverOutgoings* is in Appendix D.15.

```
Lemma NoLeftoverOutgoings : ∀ {st st' s l},
  ReachableFromBy st st' s l
  → Outgoing_transfer_recipient_and_amount (contract_state st') = None.
```

## 7.5 Induction on the Reachability Predicate

A common pattern amongst the proofs that involve the *ReachableFromBy* predicate is to perform induction on the reachability predicate. This pattern makes sense because the statements of those lemmas involve attempting to prove that some property holds for all states reachable from either a state with a specified property, or the initial state. This highlights why it was so critical to ensure that the reachability predicate was relatively easy to work with, a key motivation for the successful-calls approach discussed in Section 4.2.2. Prior to implementing the successful-calls approach, the proof for the case where the state reverts had to be repeatedly proved, which further complicated the already somewhat difficult to navigate proofs.

## 7.6 Tactics

In addition to lemmas, the *ds_inv* tactic was widely used throughout the proofs. It builds upon the *inv_runStateT_branching* tactic but adds improved handling related to the way transfers are modelled as a result of the *Checks-Effects-Interactions Pattern*. The *ds_inv* tactic also makes use of the *me_transfer_cases* tactic to discharge some goals associated with transfers automatically. The *Hlinks* tactic was also developed to bring into context information that is relevant when using the *ReachableFromBy* predicate. Here are those three tactics:

```
Ltac me_transfer_cases :=
  try match goal with
    H : (Int256.one =? Int256.one)%int256 = false ⊢ _ ⇒
      rewrite Int256.eq_true in H; discriminate
      end;
  try match goal with
    H : runStateT mzero _ = ret _ ⊢ _ ⇒
    simpl in H; discriminate
  end.
```

```
Ltac ds_inv :=
  repeat ( try inv_runStateT_branching;
           let Case := fresh "NoOverflowOrUnderflowInTransferCase" in
           try match goal with | H : context[me_transfer _ _ _] ⊢ _ ⇒
             unfold me_transfer, make_machine_env in H;
             destruct (noOverflowOrUnderflowInTransfer _ _ _ _ && (_ _ _ _ _)) eqn:Case
           end );
  me_transfer_cases.

Ltac Hlinks :=
  match goal with
  | H : ReachableFromBy _ _ _ _ ⊢ _ ⇒
    let StateToStepName := fresh "HS" in
    let StepToListName := fresh "HL" in
    epose proof (ReachableFromByLinkStateToStep _ _ _ _ H) as StateToStepName;
    epose proof (ReachableFromByLinkStepToList _ _ _ _ H) as StepToListName
  end.
```

## 7.7 Remarks

The proof themes described in this section show that there is much to be gained from carrying out smart contract proofs that can live beyond a proof relating to a specific smart contract. There are tactics, lemmas, and proof techniques that are likely to be more generally applicable to other smart contracts, especially ones which bear some similarity to existing the smart contracts.

# Chapter 8

# Conclusion

This thesis demonstrates that the formal method of interactive theorem proving can be successfully applied to the problem of needing to demonstrate that smart contracts are correct, with high-level specifications known to hold at the bytecode level. It tackles the challenge of reentrancy which is an inherent issue in the popular Ethereum blockchain and demonstrates how to model the aspects of a blockchain which are relevant to smart contract correctness. The case studies give an insight into how these techniques can be applied to the full range of smart contracts.

This work and the work it is built upon give the key ingredients which can answer the main research question of "*How can we know that a given smart contract is correct?*". This question has to be answered one smart contract at a time, but, as demonstrated, proof themes do emerge which make the process of proving the correctness of smart contracts progressively easier.

The diversity of potential smart contracts to be written and ideally proved correct, and the diversity of programs in general, are similarly wide, giving rise to essentially unending future work: specifying, implementing and proving correct all smart contracts. As a result, the future of this field is ripe. However, the field still needs more robust proof tools and languages tailored for theorem proving with formal semantics.

In a setting where the smart contract author is not malicious, the requirement to show that their implementation provably meets the smart contract's specifications would go a long way to helping prevent unintentional errors being a part of a deployed smart contract. When specifications are written well, the use of a robust proof system could guarantee the absence of errors. Also, in the case where the smart contract author may be malicious or is not trusted, a very robust proof system and trusted specifications could protect end-users from a malicious developer. In this scenario, if the specifications fully describe the behaviour that the end-user expects and the smart contract is proved to meet that specification, then the end-user could safely use the smart contract regardless of whether they trust the developer or not. It would not be possible for a malicious smart contract author to 'trick the system' and introduce behaviour not allowed by the specification. This strong guarantee can be provided by theorem proving and goes beyond any guarantee that can be made by testing, for smart contracts in general. Using proofs not only as a tool for increasing software quality but also as a defence against malicious developers would be an interesting area of future research.

Another potential application area of formal specification and proof relates to semantic versioning [150]. Lam et al. [151] discuss formal contracts (in the sense of a software contract, rather than a smart contract) and their applicability to semantic versioning. Their ideas are relevant when applied to smart contracts with formal specifications. If used to their fullest effect by developers, patch versions would be non-existent due to formal proofs of correctness. Minor versions would involve changing the behaviour of a smart contract and its specification in a way such that the specification prior to the change would still hold. Major versions would involve changing the specification for a smart contract and its behaviour such that it no longer implied that the specification prior to the change held. In a blockchain of the future which supported on-chain proofs, perhaps smart contracts could no longer be immutable and be updated if the new version continued to meet the specifications of the old version (a minor version upgrade), similarly to as discussed by Antonino et al. [126]. Investigating the feasibility of this for smart contracts would be useful future work.

If it were possible to have on-chain proof certificates that a smart contract met a certain specification, then smart contracts could also interact with other smart contracts that met a predefined specification, such as meeting the ERC-20 token specification. The proofs backing these certificates would guarantee that the certified smart contracts would behave according to the specification without requiring manual vetting and enable new opportunities for smart contract interaction. This would also be a powerful way to compose proofs, enabling new smart contracts and their proofs to build upon other proved-correct smart contracts. Even if the verification was done off-chain instead, this has the potential to be worthwhile future work.

These future-looking advancements need to build upon the solid foundation of a trusted proof system with smart contract language semantics integrated into it. We demonstrate that this is feasible building upon the DeepSEA system. Anecdotally, the experience of writing and proving correct smart contracts in DeepSEA is already enjoyable. The most crucial future work involves making that experience more accessible to the average software engineer, with the hope of increasing the adoption of formal methods in general and interactive theorem proving in particular.

In conclusion, the work of this thesis has shown an approach to building trustworthy smart contracts and opens the door to a range of exciting future work.

# Appendix A

# Source Code

The source code relating to Chapters 3 to 7 is available on GitHub at `https://github.com/Coda-Coda/PhD-Thesis-Code-Artefact`. Where possible, the files are organised by folders corresponding to chapters, however links to the relevant files for Sections 4.1 and 4.2 are provided in the *README* file. The files have been prepared so that only Coq v8.14.1 is required.

Here are three ways to interactively step through the files:

- GitHub Codespaces (web).
  1. Go to `https://github.com/Coda-Coda/PhD-Thesis-Code-Artefact`.
  2. Create a new Codespace and follow the instructions in the *README* file.
- Linux/macOS (local): Using the Nix Package Manager.
  1. Install Nix from `https://nixos.org/download`.
  2. After cloning the repository above, run `nix-shell` followed by `make` from inside the cloned folder.
  3. Run `coqide` and then open any of the `.v` files in CoqIDE to step through them interactively.
- Windows/macOS/Linux (local).
  1. Obtain Coq v8.14.1 and optionally CoqIDE from a package manager or through `opam` at `https://opam.ocaml.org/`.
  2. Clone the repository above.
  3. Run `make` from inside the cloned repository.
  4. Run `coqide` and then open any of the `.v` files in CoqIDE to step through them interactively. Alternatively, install the "VSCoq Legacy" Visual Studio Code extension and step through the proofs interactively in Visual Studio Code, available at `https://code.visualstudio.com/`.

# Appendix B

# Crowdfunding Smart Contract Proofs

## B.1 Preservation of Donation Record Theorem

```
Theorem donation_preserved :
  forall (a : addr) (d : Z),
                  (donation_recorded a d)
    `since`       (donation_recorded a d)
    `as-long-as` (no_claims_from a).
Proof.
unfold since_as_long.
intros.
induction H.
- assumption.
- assert(donation_recorded a d prevSt) by (apply IHReachableFromBy;
  intros; apply H1; apply in_cons; assumption).
  clear H0 IHReachableFromBy.
  unfold donation_recorded in *; destruct_and.
  split; [|assumption].
  Hlinks.
  assert (no_claims_from a prev) by
  (apply H1; destruct HL; subst; right; left; auto).
  destruct prev; autounfold in *; simpl in *.
  clear H1 H HL.
  unfold no_claims_from in H3.
  unfold donation_recorded in *. destruct Step_action0; simpl in *;
  rewrite <- HS in *.
  + Transparent Crowdfunding_donate_opt.
    remember (make_machine_env contract_address Step_state0 context
                          address_accepts_funds_assumption
                          callvalue_bounded_prf balances_bounded_prf
                          callvalue_prf) as machine_environment.
    unfold Crowdfunding_donate_opt in *.
    ds_inv.
    * subst.
      inversion H20.
    * subst. simpl in *.
      destruct (a =? (caller context)) eqn:Case.
      -- exfalso.
         apply Int256eq_true in Case.
         rewrite <- Case in *.
         clear -H2 Heqb0.
         Check Z.eqb_eq.
         apply Z.eqb_eq in Heqb0.
         rewrite Heqb0 in H2.
         lia.
      -- apply Int256eq_false in Case.
         Check get_default_so.
         apply get_default_so.
         apply Case.
    * subst.
      reflexivity.
  + Transparent Crowdfunding_getFunds_opt.
    remember (make_machine_env contract_address Step_state0 context
                          address_accepts_funds_assumption
                          callvalue_bounded_prf balances_bounded_prf
                          callvalue_prf) as machine_environment.
    unfold Crowdfunding_getFunds_opt in *.
    rewrite Heqmachine_environment in *.
    ds_inv; subst; reflexivity.
  + Transparent Crowdfunding_claim_opt.
    unfold Crowdfunding_claim_opt in case_claim_prf.
    ds_inv; subst; simpl in *; try reflexivity.
    * Check get_default_so.
      rewrite get_default_so by assumption.
      reflexivity.
    * apply Int256eq_true in Heqb1.
      discriminate.
  + assumption.
  + assumption.
  + assumption.

Qed.
```

## B.2 Safety Property of Sufficient Balance Theorem

```
Theorem sufficient_funds_safe : Safe balance_backed.
Proof.
  unfold Safe.
  intros.
  induction H.
  - unfold initial_state, balance_backed.
    simpl.
    intros.
    unfold Int256Tree_Properties.sum.
    unfold Int256Tree.empty.
    unfold Int256Tree.fold1.
    simpl.
    split.
    + apply snapshot_balances_valid_prf.
    + unfold Int256Tree.get_default, Int256Tree.get.
      simpl.
      unfold PTree.empty, Int256Indexed.index.
      intros.
      destruct key.
      unfold "!".
      destruct intval; intros; discriminate.
  - Hlinks. repeat rewrite HS in *.
    clear HS HL.
    destruct (Step_action prev) eqn:Case.
    + Transparent Crowdfunding_donate_opt.
      simpl in next_action.
      unfold next_blockchain_state in next_action.
      simpl in next_action.
      unfold Crowdfunding_donate_opt in case_donate_prf.
      pose proof case_donate_prf as case_donate_prf'.
      ds_inv; subst; try discriminate.
      * unfold stepOnce.
        unfold step.
```

```
rewrite -> Case.
simpl in *.
unfold next_blockchain_state.
unfold new_balance_after_contract_call.
unfold current_balances.
unfold update_balances.
unfold update_Crowdfunding_backers.
simpl.
rewrite (NoLeftoverOutgoings H).
unfold resetTransfers.
simpl.
unfold balance_backed.
simpl.
intros.
clear Case.
clear case_donate_prf.
unfold balance_backed in IHReachableFromBy.
apply IHReachableFromBy in H0.
split.
apply Z.eqb_eq in Heqb0.
rewrite negb_true_iff in H2.
rewrite H2.
rewrite Int256.eq_true.
rewrite Int256.eq_sym.
rewrite H2.
rewrite Int256Tree_sum_set_value_initially_zero;
  [|assumption].
unfold noOverflowOrUnderflowInTransfer in callvalue_prf.
destruct H0.
simpl.
lia.
destruct H0.
intros.
destruct (Int256.eq key (caller context)) eqn:SSCase.
apply Int256eq_true in SSCase.
rewrite SSCase in H3.
rewrite Int256Tree.gss in H3.
inversion H3.
clear -callvalue_bounded_prf.
lia.
```

```
      apply Int256eq_false in SSCase.
      rewrite Int256Tree.gso in H3 by assumption.
      apply H1 in H3.
      assumption.
  + Transparent Crowdfunding_getFunds_opt.
    pose proof case_getFunds_prf as case_getFunds_prf'.
    unfold Crowdfunding_getFunds_opt in case_getFunds_prf'.
    unfold stepOnce.
    unfold step.
    rewrite Case.
    simpl in case_getFunds_prf.
    clear Case.
    unfold  address_accepts_funds_assumption in *.
    ds_inv; subst; try discriminate.
    unfold me_transfer in case_getFunds_prf'2.
    unfold make_machine_env in case_getFunds_prf'2.
    simpl in *.
    destruct (noOverflowOrUnderflowInTransfer
    contract_address
    (Crowdfunding_owner
       (contract_state (Step_state prev)))
    (ContractModel.update_balances
       (caller context) contract_address
       (callvalue context)
       (balance (Step_state prev))
       contract_address)
    (ContractModel.update_balances
       (caller context) contract_address
       (callvalue context)
       (balance (Step_state prev)))).
          simpl in *.
          inversion case_getFunds_prf'2.
          unfold next_blockchain_state; simpl.
          unfold balance_backed; simpl.
          intros; discriminate.
          unfold balance_backed; simpl.
          intros; discriminate.
          clear case_getFunds_prf.
          unfold balance_backed in IHReachableFromBy.
          unfold next_blockchain_state.
```

```
            unfold balance_backed.
            simpl.
            intros.
            apply IHReachableFromBy in H0.
            unfold new_balance_after_contract_call.
            pose proof (NoLeftoverOutgoings H).
            rewrite H1.
            simpl.
            unfold step; unfold balance_backed; simpl in *.
            apply Z.eqb_eq in H6.
            rewrite H6.
            rewrite addZeroBalance.
            assumption.
            clear case_getFunds_prf.
            unfold balance_backed in IHReachableFromBy.
            unfold next_blockchain_state.
            unfold balance_backed.
            simpl.
            intros.
            apply IHReachableFromBy in H0.
            unfold new_balance_after_contract_call.
            pose proof (NoLeftoverOutgoings H).
            rewrite H1.
            simpl.
            unfold step; unfold balance_backed; simpl in *.
            apply Z.eqb_eq in H6.
            rewrite H6.
            rewrite addZeroBalance.
            assumption.
            clear case_getFunds_prf.
            unfold balance_backed in IHReachableFromBy.
            unfold next_blockchain_state.
            unfold balance_backed.
            simpl.
            intros.
            apply IHReachableFromBy in H0.
            unfold new_balance_after_contract_call.
            pose proof (NoLeftoverOutgoings H).
            rewrite H1.
            simpl.
```

```
              unfold step; unfold balance_backed; simpl in *.
              apply Z.eqb_eq in H6.
              rewrite H6.
              rewrite addZeroBalance.
              assumption.
      + Transparent Crowdfunding_claim_opt.
        pose proof case_claim_prf as case_claim_prf'.
        unfold Crowdfunding_claim_opt in case_claim_prf'.
        unfold stepOnce.
        unfold step.
        rewrite Case.
        simpl in case_claim_prf.
        clear Case.
        unfold  address_accepts_funds_assumption in *.
        ds_inv; subst; try discriminate.
        * simpl in H6.
          pose proof (NoLeftoverOutgoings H).
          apply balance_backed_in_processed_state; try assumption.
        * simpl in H6.
          pose proof (NoLeftoverOutgoings H).
          apply balance_backed_in_processed_state; try assumption.
        * simpl in *.
          destruct (noOverflowOrUnderflowInTransfer
          contract_address (caller context)
          (get_default 0 (caller context)
            (Crowdfunding_backers
                (contract_state (Step_state prev))))
          (ContractModel.update_balances
            (caller context) contract_address
            (callvalue context)
            (balance (Step_state prev)))) eqn:SCase.
          --
          simpl in *.
            inversion case_claim_prf'1.
            unfold balance_backed; simpl.
            intros.
            apply IHReachableFromBy in H0.
            unfold new_balance_after_contract_call.
            unfold update_Outgoing_transfer_recipient_and_amount.
            simpl.
            unfold update_balances.
            simpl.
            rewrite negb_true_iff in H2.
            rewrite H2.
            rewrite Int256.eq_true.
            rewrite Int256.eq_sym in H2.
            rewrite H2.
            split.
            ++
              match goal with
                H : (callvalue context =? 0) = true |- _ =>
                  apply Z.eqb_eq in H end.
              match goal with H : (callvalue context = 0) |- _ =>
                rewrite H end.
              repeat rewrite Z.add_0_r.
              match goal with H : _ /\ _ |- _ => destruct H end.
              apply Int256Tree_sum_minus.
              assumption.
            ++
            match goal with H : _ /\ _ |- _ => destruct H end.
              intros.
                destruct (Int256.eq key (caller context)) eqn:SSCase.
              **
                apply Int256eq_true in SSCase.
                  match goal with H : context[get key] |- _ =>
                    rewrite SSCase in H end.
                  match goal with H :  get _ (set _ _ _) = _ |- _ =>
                    rewrite Int256Tree.gss in H; inversion H end.
                  lia.
              **
                apply Int256eq_false in SSCase.
                  match goal with H : context[get key] |- _ =>
                    rewrite Int256Tree.gso in H by assumption end.
                  match goal with
                    H1 : (forall k v, get k _ = Some v -> v >= 0 /\
                            v < Int256.modulus),
                    H2 : (get key _ = _) |- _ =>
                      apply H1 in H2; assumption
                  end.
              -- simpl in *.
```

```
                 discriminate.
               + unfold stepOnce, step.
                 rewrite Case.
                 unfold balance_backed.
                 unfold update_balance.
                 simpl.
                 destruct prf.
                 destruct a.
                 intros.
                 apply IHReachableFromBy in H0.
                 unfold update_balances.
                 destruct (Int256.eq sender recipient); try assumption.
                 destruct (Int256.eq contract_address sender) eqn:SCase.
                   * apply Int256eq_true in SCase.
                     symmetry in SCase.
                     contradiction.
                   * destruct(Int256.eq contract_address recipient) eqn:SSCase.
                     -- apply Int256eq_true in SSCase.
                        rewrite <- SSCase.
                        split; [|apply H0].
                        lia.
                     -- split; apply H0.
               + unfold stepOnce, step.
                 rewrite Case.
                 unfold balance_backed; simpl.
                 unfold balance_backed in IHReachableFromBy.
                 simpl in *.
                 apply IHReachableFromBy.
               + unfold stepOnce, step.
                 rewrite Case.
                 apply IHReachableFromBy.

Qed.
```

## B.3 Backers Can Retrieve Their Donation Theorem

```
Theorem can_claim_back :
forall state s l backer_addr backed_amount,
  ReachableFromBy initial_state state s l ->
  backed_amount = Int256Tree.get_default 0 backer_addr
                   (Crowdfunding_backers (contract_state state)) ->
  contract_address <> backer_addr ->
  backed_amount > 0 ->
  (balance state backer_addr + backed_amount <? Int256.modulus) = true ->
  Crowdfunding_funded (contract_state state) = false ->
  balance state contract_address < (Crowdfunding_goal (contract_state state)) ->
  (forall a : addr, 0 <= balance state a < Int256.modulus) ->
  Int256.ltu
    (Crowdfunding_max_block (contract_state state)) (block_number state) = true ->
  exists (action : Action state),
          Outgoing_transfer_recipient_and_amount
            (contract_state (step_keep_transfer state action))
          = Some (backer_addr, backed_amount)
          /\
          match action with
          | call_Crowdfunding_donate context _ _ _ _ _ _ =>
              callvalue context = 0 /\ caller context = backer_addr
          | call_Crowdfunding_getFunds context _ _ _ _ _ _ =>
              callvalue context = 0 /\ caller context = backer_addr
          | call_Crowdfunding_claim context _ _ _ _ _ _ =>
              callvalue context = 0 /\ caller context = backer_addr
          | _ => False
          end.
Proof.
intros.
eexists
  (call_Crowdfunding_claim
    state
    {| origin := backer_addr;
       caller := backer_addr;
       callvalue := 0;
       coinbase := Int256.zero;
       chainid := Int256.zero
    |}
    _ _ _ tt
    {| Outgoing_transfer_recipient_and_amount :=
         (Some (backer_addr, backed_amount));
       Crowdfunding_owner := Crowdfunding_owner (contract_state state);
       Crowdfunding_max_block := Crowdfunding_max_block (contract_state state);
       Crowdfunding_goal := Crowdfunding_goal (contract_state state);
       Crowdfunding_backers :=
         set backer_addr 0 (Crowdfunding_backers (contract_state state));
       Crowdfunding_funded := Crowdfunding_funded (contract_state state);
    |} _).
simpl.
```

```
split; [reflexivity|split; reflexivity].
Unshelve.
split.
simpl.
unfold Int256.modulus, two_power_nat. lia.
unfold Int256.modulus, two_power_nat. simpl.
lia.
assumption.
simpl.
unfold noOverflowOrUnderflowInTransfer.
rewrite Z.sub_0_r.
rewrite Z.add_0_r.
apply andb_true_intro.
split.
apply andb_true_intro.
split.
apply andb_true_intro.
split.
lia.
unfold Int256.modulus, two_power_nat.  simpl.  lia.
rewrite Z.geb_le.  pose proof (H6 backer_addr).  lia.
rewrite Z.ltb_lt.  pose proof (H6 contract_address).  lia.

Transparent Crowdfunding_claim_opt.
unfold Crowdfunding_claim_opt.  simpl.
destruct((backer_addr =? contract_address)%int256) eqn:Case.
apply Int256eq_true in Case. symmetry in Case. contradiction.
simpl.
destruct ((Int256.ltu (Crowdfunding_max_block (contract_state state))
                      (block_number state))) eqn:SCase; [|discriminate].
simpl.
destruct((get_default 0 backer_addr (Crowdfunding_backers
                                      (contract_state state)) =? 0)) eqn:SSCase.
apply Z.eqb_eq in SSCase. lia.
destruct ((Crowdfunding_funded (contract_state state))) eqn:SSSCase;
  [discriminate|].
destruct ((Crowdfunding_goal (contract_state state) <=?
  ContractModel.update_balances backer_addr contract_address 0
                                (balance state) contract_address)) eqn:SSSSCase.
rewrite Z.leb_le in SSSSCase.
```

```
  rewrite addZeroBalance in SSSSCase. lia.
  simpl.
  unfold address_accepts_funds_assumption.
  destruct (noOverflowOrUnderflowInTransfer contract_address backer_addr
            (get_default 0 backer_addr (Crowdfunding_backers (contract_state state)))
             (ContractModel.update_balances backer_addr contract_address 0
              (balance state))) eqn:SSSSSCase.
  simpl.
  unfold update_Outgoing_transfer_recipient_and_amount.
  simpl.
  rewrite <- H0.
  rewrite SSSCase.
  reflexivity.
  simpl.
  unfold noOverflowOrUnderflowInTransfer in SSSSSCase.
  rewrite andb_false_iff in SSSSSCase.
  destruct SSSSSCase.
  exfalso.
 {
  assert((forall (k : Int256Tree.elt) (v : Z),
           Int256Tree.get k (Crowdfunding_backers (contract_state state))= Some v
            -> v >= 0 /\ v < Int256.modulus)
              -> sum (Crowdfunding_backers (contract_state state)) <=
                   balance state contract_address
                   -> (balance state contract_address -
                        Int256Tree.get_default 0 backer_addr
                          (Crowdfunding_backers (contract_state state)) >=? 0)
                            = true).
        {
          clear.
          intros.
          rewrite Z.geb_le.
          assert(Int256Tree.get_default 0 backer_addr
            (Crowdfunding_backers  (contract_state state))
              <= sum (Crowdfunding_backers (contract_state state))).
          apply sum_bound1; try assumption; try lia.
          intros.  apply H in H1.  destruct H1.
          lia.
          lia.
        }
        unfold ContractModel.update_balances in H8.
        rewrite Int256.eq_true in H8.
        rewrite Int256.eq_false in H8 by auto.
        rewrite Int256.eq_false in H8 by auto.
        rewrite Z.add_0_r in H8.
        pose proof sufficient_funds_safe as sufficient_funds_safe.
        unfold Safe in sufficient_funds_safe.
        apply sufficient_funds_safe in H.
        unfold balance_backed in H.
        destruct H.  assumption.
        apply H9 in H; [|assumption].
        assert((0 <=? get_default 0 backer_addr
                  (Crowdfunding_backers (contract_state state))) = true).
        unfold get_default.
        unfold Z32 in H10.
        pose proof (H10 backer_addr).
        destruct(@get Z backer_addr (Crowdfunding_backers (contract_state  state))).
        pose proof (H11 z).
        assert (Some z = Some z) by reflexivity.
        apply H12 in H13.
        destruct H13.
        rewrite Z.leb_le.  lia.
        rewrite Z.leb_le.  lia.
        rewrite H11 in H8.
        assert((get_default 0 backer_addr
          (Crowdfunding_backers (contract_state state)) <? Int256.modulus) = true).
        unfold get_default.
        unfold Z32 in H10.
        pose proof (H10 backer_addr).
        destruct(@get Z backer_addr
          (Crowdfunding_backers (contract_state  state))).
        pose proof (H12 z).
        assert (Some z = Some z) by reflexivity.
        apply H13 in H14.
        destruct H14.
        rewrite Z.ltb_lt.  lia.
        rewrite Z.ltb_lt.  unfold Int256.modulus, two_power_nat.  lia.
        rewrite H12 in H8.
        rewrite H in H8.
        simpl in H8.
```

```
          discriminate.
      }
      exfalso. rewrite H0 in H3.
      unfold ContractModel.update_balances in H8.
      rewrite Int256.eq_true in H8.
      rewrite Int256.eq_false in H8 by auto.
      rewrite Z.sub_0_r in H8.
      clear -H3 H8 H0.
      lia.

   Qed.
```

## B.4 Unsigned Integer Side-Conditions

```
Ltac code_proofs_auto :=
    intros;
    unfold synth_func_obligation;
    repeat (split; auto).
```

### B.4.1 Constructor

```
Lemma Crowdfunding_constructor_vc me d :
    high_level_invariant d →
    synth_func_cond Crowdfunding_constructor Crowdfunding_constructor_wf
                    me d.
Proof.
code_proofs_auto.
Qed.

Lemma Crowdfunding_constructor_oblg me d :
    high_level_invariant d →
    synth_func_obligation Crowdfunding_constructor Crowdfunding_constructor_wf
                          me d.
Proof.
code_proofs_auto.
Qed.
```

### B.4.2 Donate Function

```
Lemma Crowdfunding_donate_vc me d :
    high_level_invariant d →
    synth_func_cond Crowdfunding_donate Crowdfunding_donate_wf
                    me d.
Proof.
code_proofs_auto.
Qed.
```

```
Lemma Crowdfunding_donate_oblg me d :
    high_level_invariant d →
    synth_func_obligation Crowdfunding_donate Crowdfunding_donate_wf
                          me d.
Proof.
code_proofs_auto.
Qed.
```

### B.4.3 Get-Funds Function

```
Lemma Crowdfunding_getFunds_vc me d :
    high_level_invariant d →
    synth_func_cond Crowdfunding_getFunds Crowdfunding_getFunds_wf
                    me d.
Proof.
code_proofs_auto.
Qed.

Lemma Crowdfunding_getFunds_oblg me d :
    high_level_invariant d →
    synth_func_obligation Crowdfunding_getFunds Crowdfunding_getFunds_wf
                          me d.
Proof.
code_proofs_auto.
Qed.
```

### B.4.4 Claim Function

```
Lemma Crowdfunding_claim_vc me d :
    high_level_invariant d →
    synth_func_cond Crowdfunding_claim Crowdfunding_claim_wf
                    me d.
Proof.
code_proofs_auto.
Qed.

Lemma Crowdfunding_claim_oblg me d :
    high_level_invariant d →
    synth_func_obligation Crowdfunding_claim Crowdfunding_claim_wf
                          me d.
Proof.
code_proofs_auto.
Qed.
```

# Appendix C

# ERC-20 Wrapped Ether Smart Contract Proofs

## C.1 Coq Translations of Smart Contract Functions

Note: *GetHighData* is a type which corresponds to the type of the record holding the smart contract storage variables. The machine environment record, *me*, includes the caller's Ethereum address and the amount of ether that is sent with the call, as well as other details about the call.

### C.1.1 Total Supply Function

```
ERC20WrappedEth_totalSupply_opt me =
gets ERC20WrappedEth__totalSupply
```

### C.1.2 Balance Function

```
ERC20WrappedEth_balanceOf_opt _owner me =
gets
  (fun s : GetHighData =>
   get_default 0 _owner (ERC20WrappedEth_wrapped s))
```

### C.1.3 Transfer Function

```
ERC20WrappedEth_transfer_opt _to _value me =
((v <- ret (_value >=? 0);; guard v);;
 (v <-
  ret (negb (me_caller me =? me_address me)%int256);;
  guard v);;
 (v <- ret (negb (me_caller me =? _to)%int256);;
  guard v);;
 (v <- ret (me_callvalue me =? 0);; guard v);;
 spec1 <-
 gets
   (fun s : GetHighData =>
    get_default 0
      (me_caller me)
      (ERC20WrappedEth_wrapped s));;
 spec2 <-
 gets
   (fun s : GetHighData =>
    get_default 0 _to (ERC20WrappedEth_wrapped s));;
 (v <- ret (spec1 >=? _value);; guard v);;
 modify
   (fun s : GetHighData =>
    update_ERC20WrappedEth_wrapped
      (set _to (spec2 + _value)
          (ERC20WrappedEth_wrapped s)) s);;
 modify
   (fun s : GetHighData =>
    update_ERC20WrappedEth_wrapped
      (set (me_caller me)
          (spec1 - _value)
          (ERC20WrappedEth_wrapped s)) s);;
 ret tt;; ret true)
```

### C.1.4 Transfer-From Function

```
ERC20WrappedEth_transferFrom_opt _from _to _value me =
((v <- ret (_value >=? 0);; guard v);;
 (v <- ret (negb (_from =? me_address me)%int256);;
  guard v);;
 (v <- ret (negb (_from =? _to)%int256);; guard v);;
 (v <- ret (me_callvalue me =? 0);; guard v);;
 spec1 <-
 gets
   (fun s : GetHighData =>
    get_default 0
       (me_caller me)
       (get_default
           (empty Z) _from
           (ERC20WrappedEth_allowances s)));;
 (v <- ret (spec1 >=? _value);; guard v);;
 modify
   (fun s : GetHighData =>
    update_ERC20WrappedEth_allowances
       (set _from
           (set (me_caller me)
               (spec1 - _value)
               (get_default
                   (empty Z) _from
                   (ERC20WrappedEth_allowances s)))
           (ERC20WrappedEth_allowances s)) s);;
 spec2 <-
 gets
   (fun s : GetHighData =>
    get_default 0 _from (ERC20WrappedEth_wrapped s));;
 spec3 <-
 gets
   (fun s : GetHighData =>
    get_default 0 _to (ERC20WrappedEth_wrapped s));;
 (v <- ret (spec2 >=? _value);; guard v);;
 modify
   (fun s : GetHighData =>
    update_ERC20WrappedEth_wrapped
       (set _to (spec3 + _value)
           (ERC20WrappedEth_wrapped s)) s);;
 modify
   (fun s : GetHighData =>
    update_ERC20WrappedEth_wrapped
       (set _from (spec2 - _value)
           (ERC20WrappedEth_wrapped s)) s);;
 ret tt;; ret true)
```

### C.1.5 Allowance Function

```
ERC20WrappedEth_allowance_opt _owner _spender me =
gets
  (fun s : GetHighData =>
   get_default 0 _spender
     (get_default
        (empty Z) _owner
        (ERC20WrappedEth_allowances s)))
```

### C.1.6 Approve Function

```
ERC20WrappedEth_approve_opt _spender _value me =
((v <- ret (_value >=? 0);; guard v);;
 modify
   (fun s : GetHighData =>
    update_ERC20WrappedEth_allowances
      (set (me_caller me)
         (set _spender _value
            (get_default
               (empty Z)
               (me_caller me)
               (ERC20WrappedEth_allowances s)))
         (ERC20WrappedEth_allowances s)) s);;
 ret tt;; ret true)
```

### C.1.7 Approve-Safely Function

```
ERC20WrappedEth_approveSafely_opt _spender
  _currentValue _value me =
((v <- ret (_value >=? 0);; guard v);;
 spec1 <-
 gets
   (fun s : GetHighData =>
    get_default 0 _spender
      (get_default
         (empty Z)
         (me_caller me)
         (ERC20WrappedEth_allowances s)));;
 (if _currentValue =? spec1
  then
   modify
     (fun s : GetHighData =>
      update_ERC20WrappedEth_allowances
        (set (me_caller me)
           (set _spender _value
              (get_default
                 (empty Z)
                 (me_caller me)
                 (ERC20WrappedEth_allowances s)))
           (ERC20WrappedEth_allowances s)) s);;
   ret tt;; ret true
  else ret false))
```

### C.1.8 Mint Function

```
ERC20WrappedEth_mint_opt me =
((v <-
  ret (negb (me_caller me =? me_address me)%int256);;
  guard v);;
 (v <- ret (me_callvalue me >? 0);; guard v);;
 spec1 <-
 gets
   (fun s : GetHighData =>
    get_default 0
      (me_caller me)
      (ERC20WrappedEth_wrapped s));;
 modify
   (fun s : GetHighData =>
    update_ERC20WrappedEth_wrapped
      (set (me_caller me)
         (spec1 + me_callvalue me)
         (ERC20WrappedEth_wrapped s)) s);;
 spec2 <- gets ERC20WrappedEth__totalSupply;;
 modify
   (update_ERC20WrappedEth__totalSupply
      (spec2 + me_callvalue me));;
 ret tt;; ret true)
```

### C.1.9 Burn Function

```
ERC20WrappedEth_burn_opt _value me =
((v <- ret (_value >=? 0);; guard v);;
 (v <-
  ret (negb (me_caller me =? me_address me)%int256);;
  guard v);;
 (v <- ret (me_callvalue me =? 0);; guard v);;
 spec1 <-
 gets
   (fun s : GetHighData =>
    get_default 0
      (me_caller me)
      (ERC20WrappedEth_wrapped s));;
 (v <- ret (spec1 >=? _value);; guard v);;
 modify
   (fun s : GetHighData =>
    update_ERC20WrappedEth_wrapped
      (set (me_caller me)
          (spec1 - _value)
          (ERC20WrappedEth_wrapped s)) s);;
 spec2 <- gets ERC20WrappedEth__totalSupply;;
 modify
   (update_ERC20WrappedEth__totalSupply
      (spec2 - _value));;
 (d <- get;;
  (let (success, d') :=
     me_transfer me (me_caller me) _value d in
   if (success =? Int256.one)%int256
   then put d'
   else mzero));;
 ret tt;; ret true)
```

## C.2 Action Type for Blockchain Model

Inductive *Action* (before : *BlockchainState*) :=
| *call_ERC20WrappedEth_totalSupply* (context : *CallContext*)
  (*callvalue_bounded_prf* : $0 \leq$ *callvalue* context $<$ *Int256.modulus*)
  (*balances_bounded_prf* : $\forall$ *a*, $0 \leq$ (*balance* before) *a* $<$ *Int256.modulus*)
  (*callvalue_prf* : *noOverflowOrUnderflowInTransfer* (*caller* context)
   *contract_address* (*callvalue* context) (*balance* before) = *true*)
  *r*
  *contract_state_after*
  (*case_totalSupply_prf* :
   *runStateT* (*ERC20WrappedEth_totalSupply_opt*
     (*make_machine_env contract_address* before context
      *address_accepts_funds_assumption callvalue_bounded_prf*
       *balances_bounded_prf callvalue_prf*))
    (*contract_state* before)
   = *Some* (*r*, *contract_state_after*))
| *call_ERC20WrappedEth_balanceOf* (*_owner* : *addr*) (context : *CallContext*)
  (*callvalue_bounded_prf* : $0 \leq$ *callvalue* context $<$ *Int256.modulus*)

```
    (balances_bounded_prf : ∀ a, 0 ≤ (balance before) a < Int256.modulus)
    (callvalue_prf : noOverflowOrUnderflowInTransfer (caller context)
      contract_address (callvalue context) (balance before) = true)
    r
    contract_state_after
    (case_balanceOf_prf :
        runStateT (ERC20WrappedEth_balanceOf_opt _owner
                    (make_machine_env contract_address before context
                      address_accepts_funds_assumption callvalue_bounded_prf
                        balances_bounded_prf callvalue_prf))
                  (contract_state before)
        = Some (r, contract_state_after))
| call_ERC20WrappedEth_transfer (_to : addr) (_value : wei) (context : CallContext)
    (callvalue_bounded_prf : 0 ≤ callvalue context < Int256.modulus)
    (balances_bounded_prf : ∀ a, 0 ≤ (balance before) a < Int256.modulus)
    (callvalue_prf : noOverflowOrUnderflowInTransfer (caller context)
      contract_address (callvalue context) (balance before) = true)
    r
    contract_state_after
    (case_transfer_prf :
        runStateT (ERC20WrappedEth_transfer_opt _to _value
                    (make_machine_env contract_address before context
                      address_accepts_funds_assumption callvalue_bounded_prf
                        balances_bounded_prf callvalue_prf))
                  (contract_state before)
        = Some (r, contract_state_after))
| call_ERC20WrappedEth_transferFrom (_from _to : addr) (_value : wei) (context : CallContext)
    (callvalue_bounded_prf : 0 ≤ callvalue context < Int256.modulus)
    (balances_bounded_prf : ∀ a, 0 ≤ (balance before) a < Int256.modulus)
    (callvalue_prf : noOverflowOrUnderflowInTransfer (caller context)
      contract_address (callvalue context) (balance before) = true)
    r
    contract_state_after
    (case_transferFrom_prf :
        runStateT (ERC20WrappedEth_transferFrom_opt _from _to _value
                    (make_machine_env contract_address before context
                      address_accepts_funds_assumption callvalue_bounded_prf
                        balances_bounded_prf callvalue_prf))
                  (contract_state before)
        = Some (r, contract_state_after))
| call_ERC20WrappedEth_allowance (_owner _spender : addr) (context : CallContext)
    (callvalue_bounded_prf : 0 ≤ callvalue context < Int256.modulus)
    (balances_bounded_prf : ∀ a, 0 ≤ (balance before) a < Int256.modulus)
    (callvalue_prf : noOverflowOrUnderflowInTransfer (caller context)
      contract_address (callvalue context) (balance before) = true)
    r
    contract_state_after
    (case_allowance_prf :
        runStateT (ERC20WrappedEth_allowance_opt _owner _spender
                    (make_machine_env contract_address before context
                      address_accepts_funds_assumption callvalue_bounded_prf
                        balances_bounded_prf callvalue_prf))
                    (contract_state before)
          = Some (r, contract_state_after))
  | call_ERC20WrappedEth_approve (_spender : addr) (_value : wei) (context : CallContext)
      (callvalue_bounded_prf : 0 ≤ callvalue context < Int256.modulus)
      (balances_bounded_prf : ∀ a, 0 ≤ (balance before) a < Int256.modulus)
      (callvalue_prf : noOverflowOrUnderflowInTransfer (caller context)
        contract_address (callvalue context) (balance before) = true)
      r
      contract_state_after
      (case_approve_prf :
          runStateT (ERC20WrappedEth_approve_opt _spender _value
                      (make_machine_env contract_address before context
                        address_accepts_funds_assumption callvalue_bounded_prf
                          balances_bounded_prf callvalue_prf))
                    (contract_state before)
          = Some (r, contract_state_after))
  | call_ERC20WrappedEth_approveSafely (_spender : addr) (_currentValue _value : wei)
      (context : CallContext)
      (callvalue_bounded_prf : 0 ≤ callvalue context < Int256.modulus)
      (balances_bounded_prf : ∀ a, 0 ≤ (balance before) a < Int256.modulus)
      (callvalue_prf : noOverflowOrUnderflowInTransfer (caller context)
        contract_address (callvalue context) (balance before) = true)
      r
      contract_state_after
      (case_approveSafely_prf :
          runStateT (ERC20WrappedEth_approveSafely_opt _spender _currentValue _value
                      (make_machine_env contract_address before context
                        address_accepts_funds_assumption callvalue_bounded_prf
                          balances_bounded_prf callvalue_prf))
                    (contract_state before)
          = Some (r, contract_state_after))
| call_ERC20WrappedEth_mint (context : CallContext)
      (callvalue_bounded_prf : 0 ≤ callvalue context < Int256.modulus)
      (balances_bounded_prf : ∀ a, 0 ≤ (balance before) a < Int256.modulus)
      (callvalue_prf : noOverflowOrUnderflowInTransfer (caller context)
        contract_address (callvalue context) (balance before) = true)
      r
      contract_state_after
      (case_mint_prf :
          runStateT (ERC20WrappedEth_mint_opt
                      (make_machine_env contract_address before context
                        address_accepts_funds_assumption callvalue_bounded_prf
                          balances_bounded_prf callvalue_prf))
                    (contract_state before)
          = Some (r, contract_state_after))
  | call_ERC20WrappedEth_burn (_value : wei) (context : CallContext)
      (callvalue_bounded_prf : 0 ≤ callvalue context < Int256.modulus)
      (balances_bounded_prf : ∀ a, 0 ≤ (balance before) a < Int256.modulus)
      (callvalue_prf : noOverflowOrUnderflowInTransfer (caller context)
        contract_address (callvalue context) (balance before) = true)
      r
      contract_state_after
```

```
    (case_burn_prf :
        runStateT (ERC20WrappedEth_burn_opt _value
                      (make_machine_env contract_address before context
                        address_accepts_funds_assumption callvalue_bounded_prf
                          balances_bounded_prf callvalue_prf))
                    (contract_state before)
        = Some (r, contract_state_after))
  | externalBalanceTransfer (sender recipient : addr) (amount : wei)
      (prf : sender ≠ contract_address ∧ amount ≥ 0 ∧
        ((noOverflowOrUnderflowInTransfer sender recipient amount (balance before))
        && (address_accepts_funds_assumption None sender recipient amount) = true))
  | timePassing (block_count time_passing : int256)
      (prf : validTimeChange block_count time_passing (block_number before)
              (timestamp before) = true)
  | revert.
```

## C.3 Step Function for Blockchain Model

```
Definition step (before : BlockchainState) (action : Action before) : BlockchainState :=
match action with
| call_ERC20WrappedEth_totalSupply context
    callvalue_bounded_prf balances_bounded_prf callvalue_prf
    r d_after case_donate_prf ⇒
      next_blockchain_state before context d_after
| call_ERC20WrappedEth_balanceOf _owner context
    callvalue_bounded_prf balances_bounded_prf callvalue_prf
    r d_after case_balanceOf_prf ⇒
      next_blockchain_state before context d_after
| call_ERC20WrappedEth_transfer _to _value context
    callvalue_bounded_prf balances_bounded_prf callvalue_prf
    r d_after case_transfer_prf ⇒
      next_blockchain_state before context d_after
| call_ERC20WrappedEth_transferFrom _from _to _value context
    callvalue_bounded_prf balances_bounded_prf callvalue_prf
    r d_after case_transferFrom_prf ⇒
      next_blockchain_state before context d_after
| call_ERC20WrappedEth_allowance _owner _spender context
    callvalue_bounded_prf balances_bounded_prf callvalue_prf
    r d_after case_donate_prf ⇒
      next_blockchain_state before context d_after
| call_ERC20WrappedEth_approve _spender _value context
    callvalue_bounded_prf balances_bounded_prf callvalue_prf
    r d_after case_approve_prf ⇒
      next_blockchain_state before context d_after
| call_ERC20WrappedEth_approveSafely _spender _currentValue _value context
    callvalue_bounded_prf balances_bounded_prf callvalue_prf
    r d_after case_approveSafely_prf ⇒
      next_blockchain_state before context d_after
| call_ERC20WrappedEth_mint context
    callvalue_bounded_prf balances_bounded_prf callvalue_prf
    r d_after case_donate_prf ⇒
```

```
      next_blockchain_state before context d_after
| call_ERC20WrappedEth_burn _value context
    callvalue_bounded_prf balances_bounded_prf callvalue_prf
    r d_after case_donate_prf ⇒
      next_blockchain_state before context d_after
| timePassing block_count time_passing prf ⇒
    updateTimeAndBlock before block_count time_passing
| externalBalanceTransfer sender recipient amount prf ⇒
    update_balance before (update_balances sender recipient amount (balance before))
| revert ⇒ before
end.
```

## C.4 Preservation of Wrapped Ether Record Theorem

```
Definition wrappedAtLeast (a : addr) (amount : Z) (s : BlockchainState) :=
    Int256Tree.get_default 0 a (ERC20WrappedEth_wrapped (contract_state s)) ≥ amount ∧ amount > 0.
```

```
Definition no_transfer_or_burn_from (a : addr) (s : Step) :=
  match Step_action s with
  | (call_ERC20WrappedEth_burn _ context _ _ _ _ _ _) ⇒ caller context ≠ a
  | (call_ERC20WrappedEth_transferFrom _from _ _ context _ _ _ _ _ _) ⇒ _from ≠ a
  | (call_ERC20WrappedEth_transfer _ _ context _ _ _ _ _ _) ⇒ caller context ≠ a
  | _ ⇒ True
  end.
```

```
Theorem wrapped_preserved (a : addr) (amount : Z) :
                (wrappedAtLeast a amount)
  `since`       (wrappedAtLeast a amount)
  `as-long-as` (no_transfer_or_burn_from a).
Proof.

unfold since_as_long. intros.

induction H.

- assumption.

- assert(wrappedAtLeast a amount prevSt) as IHReachableFromByCorollary by

    (apply IHReachableFromBy; intros; apply H1; apply in_cons; assumption).

  unfold wrappedAtLeast in *;

    destruct IHReachableFromByCorollary

      as [IHReachableFromByCorollary1 IHReachableFromByCorollary2].

  split; [|assumption].

  Hlinks.

  assert(no_transfer_or_burn_from a prev) by

    (apply H1; destruct HL; subst; right; left; auto).

  destruct prev; autounfold in *; simpl in *.

  unfold no_transfer_or_burn_from in H2.

  destruct Step_action0; simpl in *.

  + Transparent ERC20WrappedEth_totalSupply_opt.
```

```
    unfold ERC20WrappedEth_totalSupply_opt in case_totalSupply_prf.
    ds_inv; subst; simpl in *.
    inversion case_totalSupply_prf.
    exact IHReachableFromByCorollary1.
  + Transparent ERC20WrappedEth_balanceOf_opt.
    unfold ERC20WrappedEth_balanceOf_opt in case_balanceOf_prf.
    ds_inv; subst; simpl in *.
    inversion case_balanceOf_prf.
    exact IHReachableFromByCorollary1.
  + Transparent ERC20WrappedEth_transfer_opt.
    unfold ERC20WrappedEth_transfer_opt in case_transfer_prf.
    clear H HL.
    ds_inv; subst; simpl in *.
    destruct (a =? _to)%int256 eqn:Case.
      * apply Int256eq_true in Case.
        subst.
        apply (f_equal negb) in H12. rewrite negb_involutive in H12.
        apply Int256eq_false in H12.
        rewrite get_default_so by auto.
        apply geb_ge in H4.
        rewrite get_default_ss.
        clear -IHReachableFromByCorollary1 H4.
        lia.
      * apply Int256eq_false in Case.
        Check get_default_so.
        rewrite get_default_so by auto.
        rewrite get_default_so by auto.
        exact IHReachableFromByCorollary1.
  + Transparent ERC20WrappedEth_transferFrom_opt.
    unfold ERC20WrappedEth_transferFrom_opt in case_transferFrom_prf.
    clear H HL.
    ds_inv; subst; simpl in *.
    destruct (a =? _to)%int256 eqn:Case.
    * apply Int256eq_true in Case.
      subst.
      apply (f_equal negb) in H12. rewrite negb_involutive in H12.
      apply Int256eq_false in H12.
      rewrite get_default_so by auto.
      apply geb_ge in H4.
      rewrite get_default_ss.
      clear -IHReachableFromByCorollary1 H4.
      lia.
    * apply Int256eq_false in Case.
      rewrite get_default_so by auto.
      rewrite get_default_so by auto.
      exact IHReachableFromByCorollary1.
  + Transparent ERC20WrappedEth_allowance_opt.
    unfold ERC20WrappedEth_allowance_opt in case_allowance_prf.
    ds_inv; subst; simpl in *.
    inversion case_allowance_prf.
    exact IHReachableFromByCorollary1.
  + Transparent ERC20WrappedEth_approve_opt.
    unfold ERC20WrappedEth_approve_opt in case_approve_prf.
    ds_inv; subst; simpl in *.
    inversion case_approve_prf.
    clear H HL case_approve_prf.
    destruct (_value >=? 0); simpl in *; inversion H4.
    simpl in *.
    exact IHReachableFromByCorollary1.
  + Transparent ERC20WrappedEth_approveSafely_opt.
    unfold ERC20WrappedEth_approveSafely_opt in case_approveSafely_prf.
    ds_inv; subst; simpl in *.
    inversion case_approveSafely_prf.
    clear H HL case_approveSafely_prf.
    destruct (_value >=? 0); simpl in *; inversion H4.
    destruct (_currentValue =?
                get_default 0 _spender
                  (get_default (empty Z) (caller context)
                    (ERC20WrappedEth_allowances (contract_state Step_state0))));
      inversion H3; simpl in *; exact IHReachableFromByCorollary1.
  + Transparent ERC20WrappedEth_mint_opt.
    unfold ERC20WrappedEth_mint_opt in case_mint_prf.
    clear H HL. simpl in *.
    ds_inv; subst; simpl in *.
    destruct(caller context =? contract_address)%int256; simpl in *;
    ds_inv; subst; simpl in *; try discriminate.
    destruct(callvalue context >? 0)%int256; simpl in *; simpl in *;
    ds_inv; subst; simpl in *; try discriminate.
    inversion case_mint_prf; simpl in *.
    destruct (a =? (caller context))%int256 eqn:Case.
    * apply Int256eq_true in Case.
      rewrite <- Case in *.
      rewrite get_default_ss.
      clear -IHReachableFromByCorollary1 callvalue_bounded_prf.
      lia.
    * apply Int256eq_false in Case.
      rewrite get_default_so by apply Case.
      exact IHReachableFromByCorollary1.
  + Transparent ERC20WrappedEth_burn_opt.
    unfold ERC20WrappedEth_burn_opt in case_burn_prf.
    clear H HL.
    ds_inv; subst.
    * simpl in *.
      rewrite get_default_so by auto.
      exact IHReachableFromByCorollary1.
    * exfalso. simpl in *. apply Int256eq_true in Heqb.
      inversion Heqb.
  + rewrite <- HS. apply IHReachableFromByCorollary1.
  + rewrite <- HS. apply IHReachableFromByCorollary1.
  + rewrite <- HS. apply IHReachableFromByCorollary1.
Qed.
```

## C.5 Transfer Correctness Theorem

```
Theorem transfer_correct :
  forall _to _value _from_balance_before _from_balance_after
                    _to_balance_before   _to_balance_after
   before context r contract_state_after
    callvalue_bounded_prf balances_bounded_prf callvalue_prf,
  runStateT (ERC20WrappedEth_transfer_opt _to _value
    (make_machine_env contract_address before context
      address_accepts_funds_assumption callvalue_bounded_prf
      balances_bounded_prf callvalue_prf))
    (contract_state before)
= Some (r, contract_state_after)
  -> _from_balance_before =
      Int256Tree.get_default 0 (caller context)
        (ERC20WrappedEth_wrapped (contract_state before))
  -> _to_balance_before =
      Int256Tree.get_default 0 _to
        (ERC20WrappedEth_wrapped (contract_state before))
  -> _from_balance_after =
      Int256Tree.get_default 0 (caller context)
```

```
          (ERC20WrappedEth_wrapped contract_state_after)
  -> _to_balance_after =
       Int256Tree.get_default 0 _to
          (ERC20WrappedEth_wrapped contract_state_after)
  ->     _to_balance_after = _to_balance_before + _value
     /\  _from_balance_after = _from_balance_before - _value.
Proof.

intros.
Transparent ERC20WrappedEth_transfer_opt. unfold ERC20WrappedEth_transfer_opt in H.
ds_inv. subst. simpl in *.
split.
  - apply (f_equal negb) in H9. rewrite negb_involutive in H9.
    apply Int256eq_false in H9.
    rewrite get_default_so by auto.
    rewrite get_default_ss.
    reflexivity.
  - apply negb_true_iff in H9. apply Int256eq_false in H9.
    rewrite get_default_ss.
    reflexivity.
Qed.
```

## C.6 Correctness of the Total Supply Variable Theorem

Definition *Safe P* :=
  ∀ *state s l*, *ReachableFromBy initial_state state s l* → *P state*.

Definition *total_supply_tracks_correctly state* :=
  *sum* (*ERC20WrappedEth_wrapped* (*contract_state state*))
    = (*ERC20WrappedEth__totalSupply* (*contract_state state*))
    ∧ (∀ *key value*, *get_default* 0 *key*
    (*ERC20WrappedEth_wrapped* (*contract_state state*))
       = *value* → (*value* ≥ 0)).

```
Theorem total_supply_correct : Safe total_supply_tracks_correctly.
Proof.

unfold Safe. intros.
induction H.
- unfold total_supply_tracks_correctly.
  unfold initial_state. simpl.
  split.
  * unfold sum. unfold empty. unfold Int256Tree.fold1. simpl.
    reflexivity.
  * intros. unfold get_default in H. rewrite gempty in H.
    lia.
- Hlinks.
  destruct prev; autounfold in *; simpl in *.
  destruct Step_action0; simpl in *.
  + Transparent ERC20WrappedEth_totalSupply_opt.
    unfold ERC20WrappedEth_totalSupply_opt in case_totalSupply_prf.
    ds_inv; subst; simpl in *.
    inversion case_totalSupply_prf.
    unfold total_supply_tracks_correctly.
    simpl.
    unfold total_supply_tracks_correctly in IHReachableFromBy.
    split.
    * apply IHReachableFromBy.
    * destruct IHReachableFromBy as
        [IHReachableFromByCorollary1 IHReachableFromByCorollary2].
      apply IHReachableFromByCorollary2.
  + Transparent ERC20WrappedEth_balanceOf_opt.
    unfold ERC20WrappedEth_balanceOf_opt in case_balanceOf_prf.
    ds_inv; subst; simpl in *.
    inversion case_balanceOf_prf.
    unfold total_supply_tracks_correctly.
    simpl.
    unfold total_supply_tracks_correctly in IHReachableFromBy.
    split.
    * apply IHReachableFromBy.
    * destruct IHReachableFromBy
        as [IHReachableFromByCorollary1 IHReachableFromByCorollary2].
      apply IHReachableFromByCorollary2.
  + Transparent ERC20WrappedEth_transfer_opt.
    unfold ERC20WrappedEth_transfer_opt in case_transfer_prf.
    destruct IHReachableFromBy
      as [IHReachableFromByCorollary1 IHReachableFromByCorollary2].
    clear H HL.
    ds_inv; subst; simpl in *.
    * unfold total_supply_tracks_correctly.
      simpl.
      unfold total_supply_tracks_correctly in IHReachableFromByCorollary1.
      rewrite <- IHReachableFromByCorollary1.
      apply (f_equal negb) in H9.
      rewrite negb_involutive in H9. simpl in H9.
      apply Int256eq_false in H9.
      split.
      -- apply Int256Tree_Properties.constant_sum'; try reflexivity.
         assumption.
      -- intros.
         apply (f_equal negb) in H5. rewrite negb_involutive in H5.
         apply Int256eq_false in H5.
         destruct((caller context) =? key)%int256 eqn:SCase.
           ++ apply Int256eq_true in SCase. subst.
              rewrite get_default_ss.
              lia.
           ++ apply Int256eq_false in SCase. subst.
              rewrite get_default_so by auto.
              pose proof (IHReachableFromByCorollary2 (key)
                           (get_default 0 (key)
                             (ERC20WrappedEth_wrapped
                               (contract_state Step_state0)))).
              destruct(_to =? key)%int256 eqn:SSCase.
                ** apply Int256eq_true in SSCase. subst.
                   rewrite get_default_ss.
                   lia.
                ** apply Int256eq_false in SSCase. subst.
                   rewrite get_default_so by auto.
                   lia.
  + Transparent ERC20WrappedEth_transferFrom_opt.
    unfold ERC20WrappedEth_transferFrom_opt in case_transferFrom_prf.
    destruct IHReachableFromBy
      as [IHReachableFromByCorollary1 IHReachableFromByCorollary2].
    clear H HL.
    ds_inv; subst; simpl in *.
    * unfold total_supply_tracks_correctly.
      simpl.
      unfold total_supply_tracks_correctly in IHReachableFromByCorollary1.
      rewrite <- IHReachableFromByCorollary1.
      apply (f_equal negb) in H9.
      rewrite negb_involutive in H9. simpl in H9.
      apply Int256eq_false in H9.
      split.
      -- apply Int256Tree_Properties.constant_sum'; try reflexivity.
```

```
            assumption.
         -- intros.
            apply (f_equal negb) in H5. rewrite negb_involutive in H5.
            apply Int256eq_false in H5.
            destruct(_from =? key)%int256 eqn:SCase.
              ++ apply Int256eq_true in SCase. subst.
                 rewrite get_default_ss.
                 lia.
              ++ apply Int256eq_false in SCase. subst.
                 rewrite get_default_so by auto.
                 pose proof (IHReachableFromByCorollary2 (key)
                              (get_default 0 (key)
                                (ERC20WrappedEth_wrapped
                                  (contract_state Step_state0)))).
                 destruct(_to =? key)%int256 eqn:SSCase.
                   ** apply Int256eq_true in SSCase. subst.
                      rewrite get_default_ss.
                      lia.
                   ** apply Int256eq_false in SSCase. subst.
                      rewrite get_default_so by auto.
                      lia.
     + Transparent ERC20WrappedEth_allowance_opt.
       unfold ERC20WrappedEth_allowance_opt in case_allowance_prf.
       ds_inv; subst; simpl in *.
       inversion case_allowance_prf.
       unfold total_supply_tracks_correctly.
       simpl.
       unfold total_supply_tracks_correctly in IHReachableFromBy.
       split.
       * apply IHReachableFromBy.
       * destruct IHReachableFromBy
           as [IHReachableFromByCorollary1 IHReachableFromByCorollary2].
         apply IHReachableFromByCorollary2.
     + Transparent ERC20WrappedEth_approve_opt.
       unfold ERC20WrappedEth_approve_opt in case_approve_prf.
       ds_inv; subst; simpl in *.
       inversion case_approve_prf.
       clear H HL case_approve_prf.
       destruct (_value >=? 0); simpl in *; inversion H1.
       unfold total_supply_tracks_correctly.
```

```
      simpl.
      apply IHReachableFromBy.
    + Transparent ERC20WrappedEth_approveSafely_opt.
      unfold ERC20WrappedEth_approveSafely_opt in case_approveSafely_prf.
      ds_inv; subst; simpl in *.
      inversion case_approveSafely_prf.
      clear H HL case_approveSafely_prf.
      destruct (_value >=? 0); simpl in *; [|inversion H1].
      destruct (_currentValue =?
                  get_default 0 _spender
                    (get_default (empty Z) (caller context)
                      (ERC20WrappedEth_allowances
                        (contract_state Step_state0)))); simpl in *.
      inversion H1.
      unfold total_supply_tracks_correctly. simpl. assumption.
      inversion H1.
      assumption.
      + Transparent ERC20WrappedEth_mint_opt.
      unfold ERC20WrappedEth_mint_opt in case_mint_prf.
      clear H HL.
      ds_inv; subst; simpl in *.
      * unfold total_supply_tracks_correctly.
        simpl.
        destruct IHReachableFromBy as
          [IHReachableFromByCorollary1 IHReachableFromByCorollary2].
        unfold total_supply_tracks_correctly in IHReachableFromByCorollary1.
        split.
        -- unfold sum.
           rewrite sum_set_x_minus_from_arbitrary_init with
            (v:=(get_default 0 (caller context)
              (ERC20WrappedEth_wrapped
                 (contract_state Step_state0)))) by reflexivity.
           remember((get_default 0 (caller context)
             (ERC20WrappedEth_wrapped
               (contract_state Step_state0)))) as v.
           fold (sum (ERC20WrappedEth_wrapped (contract_state Step_state0))).
           rewrite <- IHReachableFromByCorollary1.
           lia.
        -- intros.
           subst.
```

```
         pose proof (IHReachableFromByCorollary2 (caller context)
                       (get_default 0 (caller context)
                         (ERC20WrappedEth_wrapped
                           (contract_state Step_state0)))).
         destruct((caller context) =? key)%int256 eqn:SCase.
           ++ apply Int256eq_true in SCase. subst.
             rewrite get_default_ss.
             lia.
           ++ apply Int256eq_false in SCase. subst.
             rewrite get_default_so by auto.
             pose proof (IHReachableFromByCorollary2 key
                           (get_default 0 key
                             (ERC20WrappedEth_wrapped
                               (contract_state Step_state0)))).
             lia.
  + Transparent ERC20WrappedEth_burn_opt.
  unfold ERC20WrappedEth_burn_opt in case_burn_prf.
  clear H HL.
  ds_inv; subst; simpl in *.
  * unfold total_supply_tracks_correctly.
    simpl.
    destruct IHReachableFromBy
      as [IHReachableFromByCorollary1 IHReachableFromByCorollary2].
    unfold total_supply_tracks_correctly in IHReachableFromByCorollary1.
    split.
    -- unfold sum.
      rewrite sum_set_x_minus_from_arbitrary_init with
        (v:=(get_default 0 (caller context)
          (ERC20WrappedEth_wrapped
             (contract_state Step_state0)))) by reflexivity.
      remember((get_default 0 (caller context)
        (ERC20WrappedEth_wrapped
          (contract_state Step_state0)))) as v.
      fold (sum (ERC20WrappedEth_wrapped (contract_state Step_state0))).
      rewrite <- IHReachableFromByCorollary1.
      lia.
      -- intros.
      subst.
      pose proof (IHReachableFromByCorollary2 (caller context)
                    (get_default 0 (caller context)
                         (ERC20WrappedEth_wrapped (contract_state Step_state0)))).
          destruct((caller context) =? key)%int256 eqn:SCase.
            ++ apply Int256eq_true in SCase. subst.
              rewrite get_default_ss.
              lia.
            ++ apply Int256eq_false in SCase. subst.
              rewrite get_default_so by auto.
              pose proof (IHReachableFromByCorollary2 key (get_default 0 key
              (ERC20WrappedEth_wrapped (contract_state Step_state0)))).
              lia.
      * inversion Heqb.
    + rewrite <- HS. unfold total_supply_tracks_correctly.
      simpl. apply IHReachableFromBy.
    + rewrite <- HS. unfold total_supply_tracks_correctly.
      simpl. apply IHReachableFromBy.
    + rewrite <- HS. unfold total_supply_tracks_correctly.
      simpl. apply IHReachableFromBy.
Qed.
```

### C.6.1 Balances are Positive Corollary

`Definition` *balances_positive state* :=
  $(\forall$ *key value*, *get_default* 0 *key*
    (*ERC20WrappedEth_wrapped* (*contract_state state*))
      = *value* $\rightarrow$ (*value* $\geq$ 0)).

```
Theorem balances_always_positive : Safe balances_positive.
Proof.
unfold Safe.
intros.
pose proof (total_supply_correct state s l H).
unfold balances_positive, total_supply_tracks_correctly in *.
destruct H0 as [H0 H1].
assumption.
Qed.
```

## C.7 Sufficient Funds Safe Theorem

`Definition` *balance_backed state* :=
  *sum* (*ERC20WrappedEth_wrapped* (*contract_state state*))
    $\leq$ (*balance state contract_address*).

```
Theorem sufficient_funds_safe : Safe balance_backed.
Proof.

unfold Safe. intros.
pose proof (total_supply_correct state s l H).
unfold total_supply_tracks_correctly in H0.
unfold balance_backed.
destruct H0.
rewrite H0.
clear H0 H1.
induction H.
- simpl.
  apply snapshot_balances_valid_prf.
- Hlinks.
  destruct prev; autounfold in *; simpl in *.
  destruct Step_action0; simpl in *.
  + Transparent ERC20WrappedEth_totalSupply_opt.
    unfold ERC20WrappedEth_totalSupply_opt in case_totalSupply_prf.
    ds_inv; subst; simpl in *.
    inversion case_totalSupply_prf.
    unfold new_balance_after_contract_call.
    pose proof (NoLeftoverOutgoings H).
    rewrite H0.
    unfold current_balances, update_balances.
    rewrite Int256.eq_true.
    destruct((caller context) =? contract_address)%int256 eqn:Case.
    * apply IHReachableFromBy.
    * rewrite Int256.eq_sym.
      rewrite Case.
      lia.
  + Transparent ERC20WrappedEth_balanceOf_opt.
    unfold ERC20WrappedEth_balanceOf_opt in case_balanceOf_prf.
    ds_inv; subst; simpl in *.
    inversion case_balanceOf_prf.
    unfold new_balance_after_contract_call.
    pose proof (NoLeftoverOutgoings H).
    rewrite H0.
    unfold current_balances, update_balances.
    rewrite Int256.eq_true.
    destruct((caller context) =? contract_address)%int256 eqn:Case.
    * apply IHReachableFromBy.
    * rewrite Int256.eq_sym.
      rewrite Case.
      lia.
  + Transparent ERC20WrappedEth_transfer_opt.
    unfold ERC20WrappedEth_transfer_opt in case_transfer_prf.
    pose proof (NoLeftoverOutgoings H).
    clear HL H.
    ds_inv; subst; simpl in *.
    unfold new_balance_after_contract_call.
    simpl.
    rewrite H0.
    unfold current_balances, update_balances.
    rewrite Int256.eq_true.
    destruct((caller context) =? contract_address)%int256 eqn:Case.
    * apply IHReachableFromBy.
    * rewrite Int256.eq_sym.
      rewrite Case.
      lia.
  + Transparent ERC20WrappedEth_transferFrom_opt.
    unfold ERC20WrappedEth_transferFrom_opt in case_transferFrom_prf.
    pose proof (NoLeftoverOutgoings H).
    clear HL H.
    ds_inv; subst; simpl in *.
    unfold new_balance_after_contract_call.
    simpl.
    rewrite H0.
    unfold current_balances, update_balances.
    rewrite Int256.eq_true.
    destruct((caller context) =? contract_address)%int256 eqn:Case.
    * apply IHReachableFromBy.
    * rewrite Int256.eq_sym.
      rewrite Case.
      lia.
  + Transparent ERC20WrappedEth_allowance_opt.
    unfold ERC20WrappedEth_allowance_opt in case_allowance_prf.
    pose proof (NoLeftoverOutgoings H).
    clear HL H.
    ds_inv; subst; simpl in *.
    unfold new_balance_after_contract_call.
    simpl.
    rewrite H0.
    unfold current_balances, update_balances.
    rewrite Int256.eq_true.
    destruct((caller context) =? contract_address)%int256 eqn:Case.
    * apply IHReachableFromBy.
    * rewrite Int256.eq_sym.
      rewrite Case.
      lia.
  + Transparent ERC20WrappedEth_approve_opt.
    unfold ERC20WrappedEth_approve_opt in case_approve_prf.
    pose proof (NoLeftoverOutgoings H).
    clear HL H.
    ds_inv; subst; simpl in *.
    unfold new_balance_after_contract_call.
    simpl.
    rewrite H0.
    unfold current_balances, update_balances.
    rewrite Int256.eq_true.
    destruct((caller context) =? contract_address)%int256 eqn:Case.
    * apply IHReachableFromBy.
    * rewrite Int256.eq_sym.
      rewrite Case.
      lia.
  + Transparent ERC20WrappedEth_approveSafely_opt.
    unfold ERC20WrappedEth_approveSafely_opt in case_approveSafely_prf.
    pose proof (NoLeftoverOutgoings H).
    clear HL H.
    ds_inv; subst; simpl in *.
    * unfold new_balance_after_contract_call.
      simpl.
      rewrite H0.
      unfold current_balances, update_balances.
      rewrite Int256.eq_true.
      destruct((caller context) =? contract_address)%int256 eqn:Case.
      -- apply IHReachableFromBy.
      -- rewrite Int256.eq_sym.
         rewrite Case.
         lia.
    * unfold new_balance_after_contract_call. simpl. rewrite H0.
      unfold current_balances, update_balances.
```

```
        rewrite Int256.eq_true.
        destruct((caller context) =? contract_address)%int256 eqn:Case.
        -- apply IHReachableFromBy.
        -- rewrite Int256.eq_sym.
           rewrite Case.
           lia.
    + Transparent ERC20WrappedEth_mint_opt.
      unfold ERC20WrappedEth_mint_opt in case_mint_prf.
      pose proof (NoLeftoverOutgoings H).
      clear HL H.
      ds_inv; subst; simpl in *.
      unfold new_balance_after_contract_call.
      simpl.
      rewrite H0.
      unfold current_balances, update_balances.
      rewrite Int256.eq_true.
      apply (f_equal negb) in H2. rewrite negb_involutive in H2.
      simpl in H2.
      rewrite H2.
      rewrite Int256.eq_sym in H2.
      rewrite Z.gtb_lt in H6.
      rewrite H2.
      clear -IHReachableFromBy.
      lia.
    + Transparent ERC20WrappedEth_burn_opt.
      unfold ERC20WrappedEth_burn_opt in case_burn_prf.
      pose proof (NoLeftoverOutgoings H).
      clear HL H.
      ds_inv; subst; simpl in *.
      * unfold new_balance_after_contract_call.
        simpl.
        unfold current_balances, update_balances.
        rewrite Int256.eq_true.
        apply (f_equal negb) in H6. rewrite negb_involutive in H6.
        simpl in H6.
        rewrite H6.
        rewrite Int256.eq_sym in H6.
        rewrite Z.geb_le in H2.
        rewrite Z.eqb_eq in H10. rewrite H10.
        rewrite Z.add_0_r, Z.sub_0_r.
        rewrite Z.geb_le in H16.
        rewrite H6.
        clear -IHReachableFromBy H2.
        lia.
      * exfalso.
      inversion Heqb.
    + unfold current_balances, update_balances.
      destruct prf.
      clear H HL.
      apply Int256.eq_false in n.
      rewrite Int256.eq_sym in n.
      rewrite n.
      rewrite HS in *.
      destruct(contract_address =? recipient)%int256 eqn:Case.
      * destruct(sender =? recipient)%int256 eqn:SCase; try lia.
        destruct a.
        apply Int256eq_true in Case.
        rewrite <- Case.
        clear -IHReachableFromBy H.
        lia.
      * destruct(sender =? recipient)%int256 eqn:SCase; try lia.
    + rewrite HS in *. apply IHReachableFromBy.
    + rewrite HS in *. apply IHReachableFromBy.
  Qed.
```

## C.8 Unsigned Integer Side-Conditions

Ltac *code_proofs_auto* :=
    intros;
    unfold *synth_func_obligation*;
    repeat (split; auto).

Definition *toZ* : *unpair_ft* (*tint_Z_bounded Int256.modulus*) → *Z* := *id*.
Definition *etherToWei* (*ether* : *Z*) : *Z* := *ether* × (10 ^ 18).
Definition *unattainableWeiAmount* := *etherToWei* (10 ^ 20).

### C.8.1 Constructor

```
Lemma ERC20WrappedEth_constructor_vc me d :
    high_level_invariant d →
    synth_func_cond ERC20WrappedEth_constructor ERC20WrappedEth_constructor_wf
                    me d.
Proof.
code_proofs_auto.
Qed.

Lemma ERC20WrappedEth_constructor_oblg me d :
    high_level_invariant d →
    synth_func_obligation ERC20WrappedEth_constructor ERC20WrappedEth_constructor_wf
                          me d.
Proof.
code_proofs_auto.
Qed.
```

### C.8.2 Total Supply Function

```
Lemma ERC20WrappedEth_totalSupply_vc me d :
    high_level_invariant d →
    synth_func_cond ERC20WrappedEth_totalSupply ERC20WrappedEth_totalSupply_wf
                    me d.
Proof.
code_proofs_auto.
Qed.

Lemma ERC20WrappedEth_totalSupply_oblg me d :
    high_level_invariant d →
    synth_func_obligation ERC20WrappedEth_totalSupply ERC20WrappedEth_totalSupply_wf
                          me d.
Proof.
code_proofs_auto.
Qed.
```

### C.8.3 Balance Function

```
Lemma ERC20WrappedEth_balanceOf_vc a0 me d :
    ht_ft_cond a0 → ht_valid_ft_cond a0 →
    high_level_invariant d →
    synth_func_cond ERC20WrappedEth_balanceOf ERC20WrappedEth_balanceOf_wf
                    a0 me d.
Proof.
code_proofs_auto.
Qed.
```

```
Lemma ERC20WrappedEth_balanceOf_oblg a0 me d :
      ht_ft_cond a0 → ht_valid_ft_cond a0 →
      high_level_invariant d →
      synth_func_obligation ERC20WrappedEth_balanceOf ERC20WrappedEth_balanceOf_wf
                                    a0 me d.
Proof.
code_proofs_auto.
Qed.
```

### C.8.4 Allowance Function

```
Lemma ERC20WrappedEth_allowance_vc a0 a1 me d :
      ht_ft_cond a0 → ht_valid_ft_cond a0 →
      ht_ft_cond a1 → ht_valid_ft_cond a1 →
      high_level_invariant d →
      synth_func_cond ERC20WrappedEth_allowance ERC20WrappedEth_allowance_wf
                             a0 a1 me d.
Proof.
code_proofs_auto.
Qed.

Lemma ERC20WrappedEth_allowance_oblg a0 a1 me d :
      ht_ft_cond a0 → ht_valid_ft_cond a0 →
      ht_ft_cond a1 → ht_valid_ft_cond a1 →
      high_level_invariant d →
      synth_func_obligation ERC20WrappedEth_allowance ERC20WrappedEth_allowance_wf
                                    a0 a1 me d.
Proof.
code_proofs_auto.
Qed.
```

### C.8.5 Approve Function

```
Lemma ERC20WrappedEth_approve_vc a0 a1 me d :
      ht_ft_cond a0 → ht_valid_ft_cond a0 →
      ht_ft_cond a1 → ht_valid_ft_cond a1 →
      high_level_invariant d →
      synth_func_cond ERC20WrappedEth_approve ERC20WrappedEth_approve_wf
                             a0 a1 me d.
Proof.
code_proofs_auto.
Qed.

Lemma ERC20WrappedEth_approve_oblg a0 a1 me d :
      ht_ft_cond a0 → ht_valid_ft_cond a0 →
      ht_ft_cond a1 → ht_valid_ft_cond a1 →
      high_level_invariant d →
      synth_func_obligation ERC20WrappedEth_approve ERC20WrappedEth_approve_wf
                                    a0 a1 me d.
Proof.
code_proofs_auto.
Qed.
```

### C.8.6 Approve-Safely Function

Lemma *ERC20WrappedEth_approveSafely_vc a0 a1 a2 me d* :
*ht_ft_cond a0* → *ht_valid_ft_cond a0* →
*ht_ft_cond a1* → *ht_valid_ft_cond a1* →
*ht_ft_cond a2* → *ht_valid_ft_cond a2* →
*high_level_invariant d* →
*synth_func_cond ERC20WrappedEth_approveSafely ERC20WrappedEth_approveSafely_wf*
*a0 a1 a2 me d.*
Proof.
*code_proofs_auto.*
Qed.

Lemma *ERC20WrappedEth_approveSafely_oblg a0 a1 a2 me d* :
*ht_ft_cond a0* → *ht_valid_ft_cond a0* →
*ht_ft_cond a1* → *ht_valid_ft_cond a1* →
*ht_ft_cond a2* → *ht_valid_ft_cond a2* →
*high_level_invariant d* →
*synth_func_obligation ERC20WrappedEth_approveSafely ERC20WrappedEth_approveSafely_wf*
*a0 a1 a2 me d.*
Proof.
*code_proofs_auto.*
Qed.

### C.8.7 Transfer Function

```
Lemma ERC20WrappedEth_transfer_vc a0 a1 me d :
    (forall a,
      Int256Tree.get_default 0 a (ERC20WrappedEth_wrapped d)
        < unattainableWeiAmount) ->
    (toZ a1 < unattainableWeiAmount) ->
    ht_ft_cond a0 -> ht_valid_ft_cond a0 ->
    ht_ft_cond a1 -> ht_valid_ft_cond a1 ->
    high_level_invariant d ->
    synth_func_cond ERC20WrappedEth_transfer ERC20WrappedEth_transfer_wf
                    a0 a1 me d.
Proof.
intros Hextra1 Hextra2.
intros.
unfold _tp_type_pair in *.
unfold unpair_ft in *.
unfold _tp_type_pair in *.
simpl in *.
unfold Z32 in a1.
remember a0 as _to. remember a1 as _value.
clear a0 a1 Heq_to Heq_value.
unfold Z32 in _value.
simpl in *.
```

```
  unfold synth_func_cond.
  simpl.
  repeat split.
 - subst.
   simpl in *.
   inversion H9.
   inversion H8.
   destruct(MachineModel.me_callvalue me =? 0) eqn:Case; [|simpl in H7; inversion H7].
   simpl in H7. inversion H7.
   destruct(Int256.eq (MachineModel.me_caller me) _to) eqn:SCase;
     [simpl in H6; inversion H6|].
   simpl in H6. inversion H6.
   destruct(Int256.eq (MachineModel.me_caller me)
   (MachineModel.me_address me)) eqn:SSCase; [simpl in H5; inversion H5|].
   simpl in H5. inversion H5.
   destruct(_value >=? 0) eqn:SSSCase; [|simpl in H4; inversion H4].
   simpl in H4. inversion H4.
   subst.
   pose proof (Hextra1 _to). subst.
   unfold Int256.modulus, two_power_nat. simpl.
   unfold toZ, id in Hextra2.
   clear -Hextra2 H11.
   unfold Z_bounded in *.
   remember (@Int256Tree.get_default Z Z0 _to (ERC20WrappedEth_wrapped m5)) as y.
   unfold unattainableWeiAmount, etherToWei in *.
   lia.
 - remember ((@SpecTree.get
   (param_env
      (@cons hyper_type_pair int_U_pair
         (@cons hyper_type_pair int_Z32_pair (@nil hyper_type_pair)))
      (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
         (@ERC20WrappedEth_transfer memModelOps)))
   (xO (xO (xI xH)))
   (@SpecTree.set
      (@AList.set hyper_type_pair
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transfer memModelOps)) int_U_pair
         (@AList.empty TypePairProjection.A))
      (Pos.succ
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transfer memModelOps))) int_Z32_pair
      _value
      (@SpecTree.set (@AList.empty TypePairProjection.A)
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transfer memModelOps)) int_U_pair _to
         SpecTree.empty)))) as STORED_VALUE.
   destruct (v >=? STORED_VALUE) eqn:Case; [|inversion H10]. simpl in *.
   apply Z.geb_le in Case.
   lia.

 Qed.
```

Lemma *ERC20WrappedEth_transfer_oblg a0 a1 me d* :
*ht_ft_cond a0* → *ht_valid_ft_cond a0* →
*ht_ft_cond a1* → *ht_valid_ft_cond a1* →
*high_level_invariant d* →
*synth_func_obligation ERC20WrappedEth_transfer ERC20WrappedEth_transfer_wf*
*a0 a1 me d.*
Proof.
*code_proofs_auto.*
Qed.

## C.8.8 Transfer-From Function

```
Lemma ERC20WrappedEth_transferFrom_vc a0 a1 a2 me d :
    (forall a,
      Int256Tree.get_default 0 a (ERC20WrappedEth_wrapped d)
        < unattainableWeiAmount) ->
    (toZ a2 < unattainableWeiAmount) ->
    ht_ft_cond a0 -> ht_valid_ft_cond a0 ->
    ht_ft_cond a1 -> ht_valid_ft_cond a1 ->
    ht_ft_cond a2 -> ht_valid_ft_cond a2 ->
    high_level_invariant d ->
    synth_func_cond ERC20WrappedEth_transferFrom ERC20WrappedEth_transferFrom_wf
                    a0 a1 a2 me d.
Proof.
intros Hextra1 Hextra2.
intros.
unfold synth_func_cond.
simpl.
repeat split.
-
remember ((@SpecTree.get
(param_env
   (@cons hyper_type_pair int_U_pair
      (@cons hyper_type_pair int_U_pair
         (@cons hyper_type_pair int_Z32_pair
            (@nil hyper_type_pair))))
   (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
      (@ERC20WrappedEth_transferFrom memModelOps)))
(xI (xO (xI xH)))
(@SpecTree.set
   (@AList.set hyper_type_pair
      (Pos.succ
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps)))
      int_U_pair
      (@AList.set hyper_type_pair
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps))
         int_U_pair (@AList.empty TypePairProjection.A)))
   (Pos.succ
      (Pos.succ
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps))))
   int_Z32_pair a2
   (@SpecTree.set
      (@AList.set hyper_type_pair
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps))
         int_U_pair (@AList.empty TypePairProjection.A))
      (Pos.succ
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps)))
      int_U_pair a1
      (@SpecTree.set (@AList.empty TypePairProjection.A)
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps))
         int_U_pair a0 SpecTree.empty))))) as STORED_VALUE.
  destruct(v >=? STORED_VALUE) eqn:Case; [|inversion H11].
  apply Z.geb_le in Case.
lia.
-
simpl in H15.
subst. simpl.
```

```
unfold ht_ft_cond in H3.
simpl in H3.
inversion H14.
simpl in *. unfold Z32 in a2.
inversion H13.
inversion H12.
destruct(v >=? a2) eqn:SSSCase; [|simpl in H11; inversion H11].
inversion H11.
inversion H10.
destruct(MachineModel.me_callvalue me =? 0) eqn:Case; [|simpl in H9; inversion H9].
inversion H9.
destruct(a2 >=? 0) eqn:SCase; [|simpl in H6; inversion H6].
inversion H6.
destruct(Int256.eq a0 a1) eqn:SSCase; [simpl in H8; inversion H8|].
inversion H8.
destruct(Int256.eq a0 (MachineModel.me_address me)) eqn:SSSSCase;
  [simpl in H7; inversion H7|].
inversion H7.
subst.
destruct m5. simpl in *.
clear -Hextra2 Hextra1.
unfold toZ, id in Hextra2.
pose proof (Hextra1 a1) as H.
unfold Int256.modulus, two_power_nat. simpl.
unfold Z_bounded in *.
unfold unattainableWeiAmount, etherToWei in *.
lia.
-
remember ( (@SpecTree.get
(param_env
   (@cons hyper_type_pair int_U_pair
      (@cons hyper_type_pair int_U_pair
         (@cons hyper_type_pair int_Z32_pair
            (@nil hyper_type_pair))))
   (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
      (@ERC20WrappedEth_transferFrom memModelOps)))
(xI (xO (xI xH)))
(@SpecTree.set
   (@AList.set hyper_type_pair
      (Pos.succ
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps)))
      int_U_pair
      (@AList.set hyper_type_pair
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps))
         int_U_pair (@AList.empty TypePairProjection.A)))
   (Pos.succ
      (Pos.succ
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps))))
   int_Z32_pair a2
   (@SpecTree.set
      (@AList.set hyper_type_pair
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps))
         int_U_pair (@AList.empty TypePairProjection.A))
      (Pos.succ
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps)))
      int_U_pair a1
      (@SpecTree.set (@AList.empty TypePairProjection.A)
         (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
            (@ERC20WrappedEth_transferFrom memModelOps))
         int_U_pair a0 SpecTree.empty))))) as STORED_VALUE.
destruct(v0 >=? STORED_VALUE) eqn:Case; [|inversion H15].
inversion H13.
rewrite Z.geb_le in Case.
rewrite <- H18 in Case.
rewrite <- H19.
clear -Case.
lia.

Qed.
```

Lemma *ERC20WrappedEth_transferFrom_oblg a0 a1 a2 me d* :
$\quad$ *ht_ft_cond a0* → *ht_valid_ft_cond a0* →
$\quad$ *ht_ft_cond a1* → *ht_valid_ft_cond a1* →
$\quad$ *ht_ft_cond a2* → *ht_valid_ft_cond a2* →
$\quad$ *high_level_invariant d* →
$\quad$ *synth_func_obligation ERC20WrappedEth_transferFrom ERC20WrappedEth_transferFrom_wf*
$\quad\quad\quad\quad$ *a0 a1 a2 me d.*

Proof.

*code_proofs_auto.*
Qed.

## C.8.9 Mint Function

```
Lemma ERC20WrappedEth_mint_vc me d :
    (forall a,
      Int256Tree.get_default 0 a (ERC20WrappedEth_wrapped d)
        < unattainableWeiAmount) ->
    (MachineModel.me_callvalue me < unattainableWeiAmount) ->
    (ERC20WrappedEth__totalSupply d < unattainableWeiAmount) ->
    high_level_invariant d ->
    synth_func_cond ERC20WrappedEth_mint ERC20WrappedEth_mint_wf
                    me d.
Proof.
intros Hextra1 Hextra2 Hextra3.
intros.
unfold synth_func_cond.
simpl.
repeat split.
-
inversion H2.
destruct (Int256.eq
(@MachineModel.me_caller
   global_abstract_data_type
   me)
(@MachineModel.me_address
   global_abstract_data_type
   me)) eqn:Case; [simpl in H0; inversion H0|].
inversion H0.
destruct(Z.gtb
(@MachineModel.me_callvalue
   global_abstract_data_type
   me) Z0); [|simpl in H1; inversion H1].
inversion H1.
subst.
pose proof (Hextra1 (MachineModel.me_caller me)) as H3.
unfold Int256.modulus, two_power_nat. simpl.
clear -Hextra1 Hextra2.
pose proof (Hextra1 (MachineModel.me_caller me)).
unfold Z_bounded in *. simpl in *.
unfold unattainableWeiAmount, etherToWei in *.
lia.
```

```
-
  inversion H4.
  inversion H3.
  inversion H2.
  destruct(Z.gtb
  (@MachineModel.me_callvalue
     global_abstract_data_type
     me) Z0); [|simpl in H1; inversion H1].
  inversion H1.
  destruct (Int256.eq
  (@MachineModel.me_caller
     global_abstract_data_type
     me)
  (@MachineModel.me_address
     global_abstract_data_type
     me)) eqn:Case; [simpl in H0; inversion H0|].
  inversion H0.
  subst.
  destruct m2. simpl in *.
  clear -Hextra2 Hextra3.
  unfold Int256.modulus, two_power_nat. simpl.
  unfold unattainableWeiAmount, etherToWei in *.
  lia.

  Qed.
```

Lemma *ERC20WrappedEth_mint_oblg me d* :
*high_level_invariant d* →
*synth_func_obligation ERC20WrappedEth_mint ERC20WrappedEth_mint_wf*
*me d*.
Proof.
*code_proofs_auto*.
Qed.

## C.8.10 Burn Function

```
Lemma ERC20WrappedEth_burn_vc a0 me d :
   (forall (a:Int256Tree.elt),
      ERC20WrappedEth__totalSupply d
      >=
      Int256Tree.get_default 0 a (ERC20WrappedEth_wrapped d))
   ->
    ht_ft_cond a0 -> ht_valid_ft_cond a0 ->
    high_level_invariant d ->
    synth_func_cond ERC20WrappedEth_burn ERC20WrappedEth_burn_wf
                     a0 me d.
Proof.
intros Hextra1.
intros.
unfold synth_func_cond.
simpl in *.
repeat split.
- remember ((@SpecTree.get
(param_env
   (@cons hyper_type_pair int_Z32_pair (@nil hyper_type_pair))
   (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
      (@ERC20WrappedEth_burn memModelOps)))
(xI (xI (xO xH)))
(@SpecTree.set (@AList.empty TypePairProjection.A)
   (@FC_param_ident_start (@GlobalLayerSpec memModelOps)
      (@ERC20WrappedEth_burn memModelOps)) int_Z32_pair a0
   SpecTree.empty))) as STORED_VALUE.
destruct (v >=? STORED_VALUE) eqn:Case; [|inversion H6].
rewrite Z.geb_le in Case.
clear -Case.
lia.
- simpl in *. inversion H8.
destruct(v >=? a0) eqn:Case; simpl in *; [|discriminate].
rewrite Z.geb_le in Case.
inversion H5. subst.
remember ((@Int256Tree.get_default
(@Z_bounded Int256.modulus modulus_bound) Z0
(@MachineModel.me_caller global_abstract_data_type me)
(ERC20WrappedEth_wrapped m3))) as wrapped_amount.
inversion H7.
inversion H6.
destruct(Z.eqb
(@MachineModel.me_callvalue
   global_abstract_data_type
   me) Z0); [|simpl in H4; inversion H4].
inversion H4.
destruct (Int256.eq
(@MachineModel.me_caller
   global_abstract_data_type
   me)
(@MachineModel.me_address
   global_abstract_data_type
   me)) eqn:SCase; [simpl in H3; inversion H3|].
inversion H3.
destruct (a0 >=? 0); [|simpl in H2; inversion H2].
inversion H2.
subst.
clear -Hextra1 Case.
destruct m4. simpl in *.
pose proof (Hextra1 (MachineModel.me_caller me)) as H.
unfold Z_bounded in *. simpl in *.
lia.

Qed.
```

Lemma *ERC20WrappedEth_burn_oblg a0 me d* :
    *ht_ft_cond a0* → *ht_valid_ft_cond a0* →
    *high_level_invariant d* →
    *synth_func_obligation ERC20WrappedEth_burn ERC20WrappedEth_burn_wf*
                                    *a0 me d*.
Proof.
*code_proofs_auto*.
Qed.

# Appendix D

# Generic Lemmas

## D.1 *fold_snd_map*

```
Lemma fold_snd_map :
  forall  A B (m : list (A * B)) x f,
  (fold_left (fun (a : B) (p : A * B) => f a (snd p))
   m x) =
  (fold_left f
  (List.map snd m) x).
Proof.

    intro.

    induction m.

    - intros. simpl. reflexivity.
    - intros. simpl. rewrite IHm. reflexivity.

Qed.
```

## D.2 *sum_starting_from_init_equals_sum_plus_init_arbitrary_start*

```
Lemma sum_starting_from_init_equals_sum_plus_init_arbitrary_start :
forall (x init : Z) (m : Int256Tree.t Z),
Int256Tree.fold1 Z.add m (init + x) = Z.add (Int256Tree.fold1 Z.add m x) init.
Proof.

  intros.

  repeat rewrite Int256Tree.fold1_spec.

  assert(

  forall x,

    (fold_left (fun (a : Z) (p : Int256Tree.elt * Z) => Z.add a (snd p))
    (Int256Tree.elements m) x) =
    (fold_left Z.add
    (List.map snd (Int256Tree.elements m)) x)).
    {
      intros.
      apply fold_snd_map.
    }
  repeat rewrite H. clear H.
  rewrite <- fold_left_last.
  repeat rewrite fold_symmetric; try (intros; lia).
  remember (List.map snd (Int256Tree.elements m)) as l.
  clear Heql. clear m. generalize dependent l.
  induction l.
    - simpl. lia.
    - simpl.
    rewrite IHl.
    reflexivity.

Qed.
```

## D.3 *sum_starting_from_init_equals_sum_plus_init*

```
Lemma sum_starting_from_init_equals_sum_plus_init :
forall (init : Z) (m : Int256Tree.t Z),
Int256Tree.fold1 Z.add m init = Z.add (Int256Tree.fold1 Z.add m 0) init.
Proof.

  intros.
  rewrite <- sum_starting_from_init_equals_sum_plus_init_arbitrary_start.
  rewrite Z.add_0_r.
  reflexivity.

Qed.
```

## D.4 *Int256Tree_sum_set_value_initially_zero*

```
Lemma Int256Tree_sum_set_value_initially_zero :
  forall (m: Int256Tree.t Z32)  k v, Int256Tree.get_default 0 k m = 0
                -> Int256Tree_Properties.sum (Int256Tree.set k v m) =
                   Int256Tree_Properties.sum m + v.
Proof.

  unfold Z32.
  intros.
  pose (@Int256Tree_Properties.sum_get_default 0 k v (Int256Tree.set k v m))
    as Lemma1.
  simpl in Lemma1.
  unfold Int256Tree_Properties.sum.
  rewrite Lemma1; [|  unfold Int256Tree.get_default;
                      rewrite Int256Tree.gss;
                      reflexivity].
  rewrite Int256Tree_Properties.fold1_remove_set; [|intros; lia].
  unfold Int256Tree.get_default in H.

  destruct (Int256Tree.get k m) eqn:Case.
  - rewrite H in Case.
     assert(Zswap : forall x y a : Z, a + x + y = a + y + x) by (intros; lia).
     epose (Int256Tree_Properties.fold1_get Z.add Zswap v Case) as H0.
     rewrite Z.add_0_r in H0.
     rewrite <- H0.
     pose Int256Tree_Properties.sum_extensional.
     apply sum_starting_from_init_equals_sum_plus_init.
   -
   assert(Int256Tree.get_default 0 k m = 0).
   unfold Int256Tree.get_default.
   rewrite Case. reflexivity.
   pose (@Int256Tree_Properties.sum_get_default v k 0 m H0).
   rewrite Z.add_0_r in e.
   rewrite <- e.
   apply sum_starting_from_init_equals_sum_plus_init.

Qed.
```

## D.5 *sum_set_y_remove_from_starting_x*

```
Lemma sum_set_y_remove_from_starting_x :
  forall k m x y,
  Int256Tree.fold1 Z.add (Int256Tree.set k y m) x =
  Int256Tree.fold1 Z.add (Int256Tree.remove k m) x + y.
Proof.
  intros.
  pose (Int256Tree.grs k m).
  pose (Int256Tree_Properties.set_permutation 0 e).
  rewrite <- Int256Tree_Properties.elements_set_decompose in p.
  rewrite fold1_set by lia.
  rewrite sum_starting_from_init_equals_sum_plus_init.
  symmetry.
  rewrite sum_starting_from_init_equals_sum_plus_init.
  rewrite Z.add_assoc.
  reflexivity.

Qed.
```

## D.6 *sum_set_zero_remove_from_starting_x*

```
Lemma sum_set_zero_remove_from_starting_x :
  forall k m x,
  Int256Tree.fold1 Z.add (Int256Tree.set k 0 m) x =
  Int256Tree.fold1 Z.add (Int256Tree.remove k m) x.
Proof.
  intros.
  rewrite sum_set_y_remove_from_starting_x.
  rewrite Z.add_0_r.
  reflexivity.

Qed.
```

## D.7 *sum_set_zero_remove*

```
Lemma sum_set_zero_remove :
  forall k m,
  Int256Tree.fold1 Z.add (Int256Tree.set k 0 m) 0 =
  Int256Tree.fold1 Z.add (Int256Tree.remove k m) 0.
Proof.
  intros.
  apply sum_set_zero_remove_from_starting_x.

Qed.
```

## D.8 *sum_set_x_minus_from_arbitrary_init*

```
Lemma sum_set_x_minus_from_arbitrary_init :
  forall (k : elt) (m : t Z) (v x init : Z),
  get_default 0 k m = v ->
  fold1 Z.add (set k x m) init = fold1 Z.add m init + (x - v).
Proof.

unfold sum.
  intros.
  unfold Int256Tree_Properties.sum.
  unfold Int256Tree.get_default in H.
  destruct (Int256Tree.get k m) eqn:Case.
    - subst.
      assert((forall x y a : Z, a + x + y = a + y + x)) by (intros; lia).
      epose (Int256Tree_Properties.fold1_get Z.add H init Case).
      rewrite e.
      simpl.
      assert (init + v = v + init) by apply Z.add_comm.
      rewrite H0. clear H0.
      rewrite (sum_starting_from_init_equals_sum_plus_init_arbitrary_start init).
      rewrite sum_set_y_remove_from_starting_x.
      lia.
    - subst.
      assert((forall x y a : Z, a + x + y = a + y + x)) by (intros; lia).
      simpl.
      rewrite Z.sub_0_r.
      rewrite sum_set_y_remove_from_starting_x.
      assert(get_default 0 k m = 0).
        unfold get_default. rewrite Case. reflexivity.
      pose proof (@sum_get_default init k 0 m H0).
      rewrite Z.add_0_r in H1.
      rewrite <- H1.
      reflexivity.

Qed.
```

## D.9 *sum_set_zero_minus_from_arbitrary_init*

```
Lemma sum_set_zero_minus_from_arbitrary_init :
  forall (k : elt) (m : t Z) (v init : Z),
  get_default 0 k m = v ->
  fold1 Z.add (set k 0 m) init = fold1 Z.add m init - v.
Proof.

intros.
apply sum_set_x_minus_from_arbitrary_init; assumption.

Qed.
```

## D.10 *sum_set_zero_minus*

```
Lemma sum_set_zero_minus : forall k m v, Int256Tree.get_default 0 k m = v ->
    Int256Tree_Properties.sum (Int256Tree.set k 0 m)
  = Int256Tree_Properties.sum m - v.
Proof.

  intros.

  unfold sum.

  apply sum_set_zero_minus_from_arbitrary_init.
  assumption.

Qed.
```

## D.11 *Int256Tree_sum_minus_equality*

```
Lemma Int256Tree_sum_minus_equality :
  forall m k x,
    Int256Tree_Properties.sum m >= x
    ->
    Int256Tree_Properties.sum (Int256Tree.set k 0 m) =
    (Int256Tree_Properties.sum m) - (Int256Tree.get_default 0 k m).
Proof.

intros.

unfold sum.

rewrite sum_set_zero_minus_from_arbitrary_init
  with (v:= Int256Tree.get_default 0 k m) by reflexivity.
reflexivity.

Qed.
```

## D.12 *Int256Tree_sum_minus_from_starting_x*

```
Lemma Int256Tree_sum_minus_from_starting_x :
  forall (m : t Z) (k : elt) (x : Z),
      fold1 Z.add (set k 0 m) x =
      fold1 Z.add m x - get_default 0 k m.
Proof.
  intros.
  rewrite sum_set_zero_minus_from_arbitrary_init
    with (v:= Int256Tree.get_default 0 k m) by reflexivity.
  reflexivity.

Qed.
```

## D.13 *Int256Tree_sum_minus*

```
Lemma Int256Tree_sum_minus :
  forall m k x,
    Int256Tree_Properties.sum m <= x
    ->
    Int256Tree_Properties.sum (Int256Tree.set k 0 m) <=
    x - (Int256Tree.get_default 0 k m).
Proof.
intros.
rewrite sum_set_zero_minus
  with (v:= Int256Tree.get_default 0 k m) by reflexivity.
lia.

Qed.
```

## D.14 *addZeroBalance*

```
Lemma addZeroBalance : forall sender recipient balances a,
  update_balances sender recipient 0 balances a = balances a.
Proof.
  intros.
  unfold update_balances.
  destruct (Int256.eq sender recipient).
  - reflexivity.
  - destruct (Int256.eq a sender) eqn:Case.
    + rewrite Z.sub_0_r.
    apply Int256eq_true in Case. subst.
    reflexivity.
    + destruct (Int256.eq a recipient) eqn:SCase.
      * rewrite Z.add_0_r.
        apply Int256eq_true in SCase. subst.
        reflexivity.
      * reflexivity.

Qed.
```

## D.15 *NoLeftoverOutgoings*

```
Lemma NoLeftoverOutgoings : forall {st st' s l},
  ReachableFromBy st st' s l
  -> Outgoing_transfer_recipient_and_amount (contract_state st') = None.
Proof.
  intros.
  induction H.
  - assumption.
  - unfold stepOnce.
    unfold step.
    destruct (Step_action prev) eqn:Case; simpl; try reflexivity.
    all: reachableFromByLinks; subst; assumption.
Qed.
```

# Bibliography

[1] E. W. Dijkstra, 'The humble programmer,' circulated privately, n.d. [Online]. Available: https://www.cs.utexas.edu/~EWD/ewd03xx/EWD340.PDF (visited on 28/10/2023).

[2] M. Emre, R. Schroeder, K. Dewey and B. Hardekopf, 'Translating C to safer Rust,' *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. DOI: 10.1145/3485498. [Online]. Available: https://doi.org/10.1145/3485498.

[3] L. Di Grazia and M. Pradel, 'The evolution of type annotations in Python: An empirical study,' in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 209–220, ISBN: 9781450394130. DOI: 10.1145/3540250.3549114. [Online]. Available: https://doi-org.ezproxy.waikato.ac.nz/10.1145/3540250.3549114.

[4] L. Lampropoulos, M. Hicks and B. C. Pierce, 'Coverage guided, property based testing,' *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360607. [Online]. Available: https://doi.org/10.1145/3360607.

[5] V. Sjöberg, Y. Sang, S.-c. Weng and Z. Shao, 'DeepSEA: A language for certified system software,' *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[6] S.-C. Weng, *DeepSpec: Modular certified programming with deep specifications*, May 2016. [Online]. Available: https://scw.tw/works/pub/dissertation-1130.pdf (visited on 20/05/2024).

[7] V. Sjöberg, K. Dave, D. Britten *et al.*, *Foundational verification of smart contracts through verified compilation*, 2024. arXiv: 2405.08348 [cs.PL].

[8] The Coq Development Team, 'The Coq Proof Assistant,' Jan. 2022. DOI: 10.5281/zenodo.5846982. [Online]. Available: https://coq.inria.fr (visited on 10/08/2022).

[9] CertiK, *CertiK - securing the Web3 world.* [Online]. Available: https://certik.com/ (visited on 09/03/2020).

[10] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister and C. Ferdinand, 'CompCert – a formally verified optimizing compiler,' in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, SEE, Toulouse, France, Jan. 2016. [Online]. Available: https://hal.inria.fr/hal-01238879.

[11] X. Leroy, *The CompCert verified compiler*, 2007–2023. [Online]. Available: https://compcert.org/ (visited on 18/10/2023).

[12] D. Britten, V. Sjöberg and S. Reeves, 'Using Coq to enforce the checks-effects-interactions pattern in DeepSEA smart contracts (short paper),' in *3rd International Workshop on Formal Methods for Blockchains (FMBC 2021)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021. DOI: 10.4230/OASIcs.FMBC.2021.3.

[13] D. Britten and S. Reeves, 'Modelling a blockchain for smart contract verification using DeepSEA,' in *Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems*, 2022, pp. 88–94.

[14] F. Vogelsteller and V. Buterin, *ERC-20: Token standard*, Nov. 2015. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20 (visited on 30/09/2023).

[15] I. Sergey, A. Kumar and A. Hobor, 'Scilla: A Smart Contract Intermediate-Level LAnguage,' en, *arXiv:1801.00687 [cs]*, Jan. 2018, arXiv: 1801.00687. DOI: 10.48550/arXiv.1801.00687. (visited on 04/04/2019).

[16] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov and K. C. G. Hao, 'Safer smart contract programming with Scilla,' *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[17] I. Sergey, A. Kumar and A. Hobor, 'Temporal properties of smart contracts,' in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2018, pp. 323–338, ISBN: 978-3-030-03427-6.

[18] Zilliqa, *Scilla-Coq repository*. [Online]. Available: https://github.com/Zilliqa/scilla-coq/ (visited on 25/08/2023).

[19] E. W. Dijkstra, 'Some meditations on advanced programming,' circulated privately, n.d. [Online]. Available: https://www.cs.utexas.edu/users/EWD/ewd00xx/EWD32.PDF (visited on 28/10/2023).

[20] Mathematisch Centrum (Amsterdam, Netherlands), A. M. Cohen and Centrum voor Wiskunde en Informatica (Amsterdam), *Images of SMC Research 1996*. Stichting Mathematisch Centrum, 1996, ISBN: 9789061964629.

[21] NASA Langley Formal Methods Research Program, *What is formal methods?* [Online]. Available: https://shemesh.larc.nasa.gov/fm/fm-what.html (visited on 02/04/2020).

[22] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem *et al.*, *Handbook of model checking*. Springer, 2018, vol. 10.

[23] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*, eng, 1st ed. San Francisco: Elsevier Science & Technology, 2007, ISBN: 0123725011.

[24] S. Jaidka, S. Reeves and J. Bowen, 'A Coloured Petri Net approach to model and analyze safety-critical interactive systems,' in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2019, pp. 347–354.

[25] J. M. Spivey and J. Abrial, *The Z notation*. Prentice Hall Hemel Hempstead, 1992, vol. 29.

[26] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs and K. R. M. Leino, 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs,' en, in *Formal Methods for Components and Objects*, D. Hutchison, T. Kanade, J. Kittler *et al.*, Eds., vol. 4111, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387, ISBN: 978-3-540-36749-9 978-3-540-36750-5. (visited on 26/07/2019).

[27] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt and M. Ulbrich, Eds., *Deductive Software Verification – The KeY Book* (Lecture Notes in Computer Science). Cham: Springer International Publishing, 2016, vol. 10001, ISBN: 978-3-319-49811-9 978-3-319-49812-6. (visited on 26/07/2019).

[28] T. Nipkow, M. Wenzel and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[29] A. Lyons, A. Danis, yyshen *et al.*, *seL4/seL4: MCS pre-release*, version 10.1.1-mcs, Dec. 2018. DOI: 10.5281/zenodo.1867796. [Online]. Available: https://doi.org/10.5281/zenodo.1867796.

[30] A. W. Appel, L. Beringer, A. Chlipala *et al.*, *The science of deep specification*. [Online]. Available: https://deepspec.org/ (visited on 03/08/2023).

[31] ISO Central Secretary, 'Blockchain and distributed ledger technologies — Vocabulary,' en, International Organization for Standardization, Geneva, CH, Standard ISO 22739:2020, 2020. [Online]. Available: https://www.iso.org/standard/73771.html.

[32] S. Wang, X. Ni, Y. Yuan, F.-Y. Wang, X. Wang and L. Ouyang, 'A preliminary research of prediction markets based on blockchain powered smart contracts,' *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 1287–1293, 2018.

[33] A. Wright and P. De Filippi, 'Decentralized blockchain technology and the rise of lex cryptographia,' *Available at SSRN 2580664*, 2015.

[34] N. Szabo, 'Formalizing and securing relationships on public networks,' *First Monday*, vol. 2, no. 9, 1997.

[35] M. Klems, J. Eberhardt, S. Tai, S. Härtlein, S. Buchholz and A. Tidjani, 'Trustless intermediation in blockchain-based decentralized service marketplaces,' in *Service-Oriented Computing*, M. Maximilien, A. Vallecillo, J. Wang and M. Oriol, Eds., Cham: Springer International Publishing, 2017, pp. 731–739, ISBN: 978-3-319-69035-3.

[36] V. Buterin *et al.*, *Ethereum white paper: A next-generation smart contract and decentralized application platform*, 2013. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper#blockchain-and-mining.

[37] M. Wohrer and U. Zdun, 'Smart contracts: Security patterns in the Ethereum ecosystem and Solidity,' in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, IEEE, 2018, pp. 2–8.

[38] M. I. Mehar, C. L. Shier, A. Giambattista *et al.*, 'Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack,' en, *Journal of Cases on Information Technology*, vol. 21, no. 1, pp. 19–32, Jan. 2019, ISSN: 1548-7717, 1548-7725. (visited on 31/07/2019).

[39] P. Technologies, *A Postmortem on the Parity Multi-Sig Library Self-Destruct*, en, Nov. 2017. [Online]. Available: https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/ (visited on 31/07/2019).

[40] G. Wood *et al.*, 'Ethereum: A secure decentralised generalised transaction ledger,' *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[41] P. Woitschig, G. S. Uddin, T. Xie and W. K. Härdle, 'The energy consumption of the Ethereum-ecosystem,' *Available at SSRN 4526732*, 2023. DOI: 10.2139/ssrn.4526732.

[42] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena and A. Hobor, 'Finding The Greedy, Prodigal, and Suicidal Contracts at Scale,' en, *arXiv:1802.06038 [cs]*, Feb. 2018, arXiv: 1802.06038. (visited on 02/04/2019).

[43] M. Eberl, G. Klein, T. Nipkow, L. Paulson, R. Thiemann *et al.*, *Archive of formal proofs*. [Online]. Available: https://www.isa-afp.org/ (visited on 16/05/2020).

[44] L. De Moura and N. Bjørner, 'Z3: An efficient SMT solver,' in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[45] K. R. M. Leino, 'Dafny: An automatic program verifier for functional correctness,' in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370, ISBN: 978-3-642-17511-4.

[46] J.-C. Filliâtre and A. Paskevich, 'Why3 — Where Programs Meet Provers,' en, in *Programming Languages and Systems*, D. Hutchison, T. Kanade, J. Kittler *et al.*, Eds., vol. 7792, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128, ISBN: 978-3-642-37035-9 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_8. [Online]. Available: http://link.springer.com/10.1007/978-3-642-37036-6_8 (visited on 05/04/2019).

[47] *Boogie: An Intermediate Verification Language*, en-US. [Online]. Available: https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/ (visited on 27/07/2019).

[48] D. J. Pearce and L. Groves, 'Whiley: A platform for research in software verification,' in *Software Language Engineering*, M. Erwig, R. F. Paige and E. Van Wyk, Eds., Cham: Springer International Publishing, 2013, pp. 238–248, ISBN: 978-3-319-02654-1.

[49] OCamlPRO, *An SMT solver for software verification*, n.d. [Online]. Available: https://alt-ergo.ocamlpro.com/ (visited on 23/08/2023).

[50] A. Stump, C. W. Barrett and D. L. Dill, 'CVC: A cooperating validity checker,' in *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*, Springer, 2002, pp. 500–504.

[51] T. Taft, 'SPARK formal verification for security,' *ACM Sigada Ada Letters*, vol. 39, pp. 83–99, 2020.

[52] J. Sun, Y. Liu, J. S. Dong and J. Pang, 'PAT: Towards flexible verification under fairness,' ser. Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 709–714.

[53] N. U. of Singapore, *PAT: Process analysis toolkit*. [Online]. Available: https://pat.comp.nus.edu.sg/ (visited on 22/05/2020).

[54] G. Bai, Q. Ye, Y. Wu *et al.*, 'Towards model checking Android applications,' *IEEE Transactions on Software Engineering*, vol. 44, no. 6, pp. 595–612, 2017.

[55] Y. Li, J. S. Dong, J. Sun, Y. Liu and J. Sun, 'Model checking approach to automated planning,' *Formal Methods in System Design*, vol. 44, no. 2, pp. 176–202, 2014.

[56] L. Lamport, *The TLA$^+$ home page*. [Online]. Available: http://lamport.azurewebsites.net/tla/tla.html (visited on 29/02/2020).

[57] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker and M. Deardeuff, 'How Amazon web services uses formal methods,' *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, 2015.

[58] B. Batson and L. Lamport, 'High-level specifications: Lessons from industry,' in *International Symposium on Formal Methods for Components and Objects*, Springer, 2002, pp. 242–261.

[59] K. Claessen and J. Hughes, 'Quickcheck: A lightweight tool for random testing of Haskell programs,' in *SIGP*, 2011.

[60] M. Dénès, C. Hritcu, L. Lampropoulos, Z. Paraskevopoulou and B. C. Pierce, 'QuickChick: Property-based testing for Coq,' in *The Coq Workshop*, 2014.

[61] A. Pneuli, *Deduction is forever*, Unpublished, invited talk at Formal Methods '99, Toulouse, Sep. 1999.

[62] Y. Hirai, 'Defining the Ethereum Virtual Machine for interactive theorem provers,' in *International Conference on Financial Cryptography and Data Security*, Springer, 2017, pp. 520–535. DOI: 10.1007/978-3-319-70278-0_33.

[63] Y. Hirai, *A Lem formalization of EVM and some Isabelle/HOL proofs*. [Online]. Available: https://github.com/pirapira/eth-isabelle (visited on 05/04/2019).

[64] E. Hildenbrandt, M. Saxena, N. Rodrigues *et al.*, 'KEVM: A complete formal semantics of the Ethereum virtual machine,' in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, IEEE, 2018, pp. 204–217.

[65] G. Roșu and T. F. Șerbănută, 'An overview of the K semantic framework,' *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010, Membrane computing and programming, ISSN: 1567-8326. DOI: 10.1016/j.jlap.2010.03.012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1567832610000160.

[66] F. Cassez, J. Fuller, M. K. Ghale, D. J. Pearce and H. M. Quiles, 'Formal and executable semantics of the Ethereum virtual machine in Dafny,' in *International Symposium on Formal Methods*, Springer, 2023, pp. 571–583.

[67] D. K. Cumming, M. Utting, N. Dong, F. Cassez, S. Tork and M. Risius, 'Verification of EVM bytecode with Vale,' 2022.

[68] N. Labs, *Michelson: The language of smart contracts in Tezos*. [Online]. Available: https://tezos.gitlab.io/whitedoc/michelson.html (visited on 09/06/2019).

[69] L. Goodman, *Tezos—a self-amending crypto-ledger white paper*, Sep. 2014. [Online]. Available: https://www.tezos.com/static/papers/white_paper.pdf.

[70] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin and J. Tesson, 'Mi-Cho-Coq, a framework for certifying Tezos smart contracts,' in *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part I 3*, Springer, 2020, pp. 368–379.

[71] P. Wadler, *Simplicity and Michelson*, Dec. 2017. [Online]. Available: https://iohk.io/blog/simplicity-and-michelson/ (visited on 05/03/2019).

[72] S. Munir and W. Taha, *Pre-deployment analysis of smart contracts – a survey*, 2023. arXiv: 2301.06079 [cs.CR].

[73] D. Magazzeni, P. McBurney and W. Nash, 'Validation and verification of smart contracts: A research agenda,' *Computer*, vol. 50, no. 9, pp. 50–57, 2017.

[74] A. Miller, Z. Cai and S. Jha, 'Smart Contracts and Opportunities for Formal Methods,' en, in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, T. Margaria and B. Steffen, Eds., vol. 11247, Cham: Springer International Publishing, 2018, pp. 280–299, ISBN: 978-3-030-03426-9 978-3-030-03427-6. DOI: 10.1007/978-3-030-03427-6_22. [Online]. Available: http://link.springer.com/10.1007/978-3-030-03427-6_22 (visited on 21/06/2019).

[75] A. Mavridou and A. Laszka, 'Tool demonstration: FSolidM for designing secure Ethereum smart contracts,' *ArXiv*, vol. abs/1802.09949, 2018.

[76] R. Cooperative, *Rholang*. [Online]. Available: https://github.com/rchain/rchain/tree/master/rholang (visited on 17/05/2020).

[77] Vyper Team, *Vyper documentation*. [Online]. Available: https://vyper.readthedocs.io/en/latest/index.html (visited on 17/05/2020).

[78] K. Crary and M. J. Sullivan, 'Peer-to-peer affine commitment using Bitcoin,' in *PLDI 2015*, 2015.

[79] R. O'Connor, 'Simplicity: A new language for blockchains,' *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, 2017.

[80] M. Chakravarty, R. Kireev, K. MacKenzie *et al.*, *Functional blockchain contracts (draft)*, May 2019. [Online]. Available: https://iohk.io/en/research/library/papers/functional-blockchain-contracts/.

[81] OCamlPRO, *Liquidity*. [Online]. Available: https://www.liquidity-lang.org/ (visited on 10/06/2019).

[82] A. Singh, R. M. Parizi, Q. Zhang, K.-K. R. Choo and A. Dehghantanha, 'Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities,' *Computers & Security*, vol. 88, p. 101 654, 2020, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2019.101654. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404818310927.

[83] X. Bai, Z. Cheng, Z. Duan and K. Hu, 'Formal Modeling and Verification of Smart Contracts,' en, in *Proceedings of the 2018 7th International Conference on Software and Computer Applications - ICSCA 2018*, Kuantan, Malaysia: ACM Press, 2018, pp. 322–326, ISBN: 978-1-4503-5414-1. DOI: 10.1145/3185089.3185138. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3185089.3185138 (visited on 13/06/2019).

[84] T. Abdellatif and K. Brousmiche, 'Formal verification of smart contracts based on users and blockchain behaviors models,' in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018, pp. 1–5.

[85] K. Bhargavan, N. Swamy, S. Zanella-Béguelin *et al.*, 'Formal Verification of Smart Contracts: Short Paper,' en, in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS'16*, Vienna, Austria: ACM Press, 2016, pp. 91–96, ISBN: 978-1-4503-4574-3. (visited on 01/04/2019).

[86] L. Luu, D.-H. Chu, H. Olickel, P. Saxena and A. Hobor, 'Making smart contracts smarter,' *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[87] D. C. Sánchez, 'Raziel: Private and verifiable smart contracts on blockchains,' *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 878, 2017.

[88] S. Azzopardi, J. Ellul and G. J. Pace, 'Monitoring smart contracts: ContractLarva and open challenges beyond,' in *RV*, 2018.

[89] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli and M. Vechev, 'Securify: Practical Security Analysis of Smart Contracts,' en, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18*, Toronto, Canada: ACM Press, 2018, pp. 67–82, ISBN: 978-1-4503-5693-0. (visited on 26/07/2019).

[90] S. Amani, M. Bégel, M. Bortin and M. Staples, 'Towards verifying Ethereum smart contract bytecode in Isabelle/HOL,' en, in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2018*, Los Angeles, CA, USA: ACM Press, 2018, pp. 66–77, ISBN: 978-1-4503-5586-5. (visited on 01/04/2019).

[91] S. Kalra, S. Goel, M. Dhawan and S. Sharma, 'Zeus: Analyzing safety of smart contracts.,' in *NDSS*, 2018, pp. 1–12.

[92] N. Atzei, M. Bartoletti and T. Cimoli, 'A survey of attacks on ethereum smart contracts (sok),' in *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*, Springer, 2017, pp. 164–186.

[93] K. D. Team, *Welcome to Pact*. [Online]. Available: https://pactlang.org/beginner/welcome-to-pact/ (visited on 27/07/2019).

[94] S. Popejoy, *Pact - smart contract language*, en, Jun. 2017. [Online]. Available: https://www.kadena.io/whitepapers.

[95] Zilliqa, *Zilliqa | Scilla is made for DeFi*, n.d. [Online]. Available: https://www.zilliqa.com/language (visited on 23/08/2023).

[96] D. Annenkov, J. B. Nielsen and B. Spitters, 'ConCert: A smart contract certification framework in Coq,' in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020, pp. 215–228.

[97] F. Cassez, J. Fuller and H. M. A. Quiles, 'Deductive verification of smart contracts with Dafny,' in *International Conference on Formal Methods for Industrial Critical Systems*, Springer, 2022, pp. 50–66.

[98] F. Cassez, J. Fuller and H. M. Antón Quiles, 'Deductive verification of smart contracts with Dafny,' *International Journal on Software Tools for Technology Transfer*, pp. 1–15, 2024.

[99] CertiK Foundation, *DeepSEA repository*, n.d. [Online]. Available: https://github.com/shentufoundation/deepsea (visited on 19/10/2023).

[100] CertiK, *An introduction to DeepSEA*, Jan. 2020. [Online]. Available: https://www.certik.com/resources/blog/an-introduction-to-deepsea (visited on 10/05/2024).

[101] CertiK, *How DeepSEA works*, Jan. 2020. [Online]. Available: https://www.certik.com/resources/blog/how-deepsea-works-with-an-example-token-contact (visited on 10/05/2024).

[102] CertiK, *How DeepSEA works - Internet Archive*, Jan. 2020. [Online]. Available: https://web.archive.org/web/20211020184322/https://certik.io/blog/technology/how-deepsea-works-with-an-example-token-contact (visited on 16/05/2024).

[103] V. Sjöberg *et al.*, *DeepSEA language reference*, Mar. 2021. [Online]. Available: https://github.com/shentufoundation/deepsea/blob/master/DeepSEA%20language%20reference.pdf (visited on 08/09/2023).

[104] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen and M. T. Vechev, 'VerX: Safety verification of smart contracts,' en, in *IEEE S&P 2020*, 2020.

[105] Y. Wang, S. K. Lahiri, S. Chen *et al.*, 'Formal Specification and Verification of Smart Contracts for Azure Blockchain,' en, *arXiv preprint arXiv:1812.08829*, Dec. 2018. (visited on 25/07/2019).

[106] M. Marescotti, R. Otoni, L. Alt, P. Eugster, A. E. J. Hyvärinen and N. Sharygina, 'Accurate smart contract verification through direct modelling,' in *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2020, pp. 178–194, ISBN: 978-3-030-61467-6.

[107] N. Ashizawa, N. Yanai, J. P. Cruz and S. Okamura, 'Eth2Vec: Learning contract-wide code representations for vulnerability detection on Ethereum smart contracts,' in *Proceedings of the 3rd ACM international symposium on blockchain and secure critical infrastructure*, 2021, pp. 47–59.

[108] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. Alexandrov, 'SmartCheck: Static analysis of Ethereum smart contracts,' in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 9–16, ISBN: 9781450357265. DOI: 10.1145/3194113.3194115.

[109] D. Park, Y. Zhang and G. Rosu, 'End-to-end formal verification of Ethereum 2.0 deposit smart contract,' in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds., Cham: Springer International Publishing, 2020, pp. 151–164, ISBN: 978-3-030-53288-8.

[110] J. Jiao, S.-W. Lin and J. Sun, 'A generalized formal semantic framework for smart contracts.,' in *FASE*, vol. 20, 2020, pp. 75–96.

[111] Move langauge contributors, *Move language repository.* [Online]. Available: https://github.com/move-language/move (visited on 02/05/2024).

[112] D. Dill, W. Grieskamp, J. Park, S. Qadeer, M. Xu and E. Zhong, 'Fast and reliable formal verification of smart contracts with the Move Prover,' in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds., Cham: Springer International Publishing, 2022, pp. 183–200, ISBN: 978-3-030-99524-9.

[113] S. Dharanikota, S. Mukherjee, C. Bhardwaj, A. Rastogi and A. Lal, 'Celestial: A smart contracts verification framework,' in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 133–142. DOI: 10.34727/2021/isbn.978-3-85448-046-4_22.

[114] N. Swamy, C. Hriţcu, C. Keller *et al.*, 'Dependent types and multi-monadic effects in F*,' in *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, Jan. 2016, pp. 256–270, ISBN: 978-1-4503-3549-2.

[115] V. Wüstholz and M. Christakis, 'Harvey: A greybox fuzzer for smart contracts,' in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.

[116] Consensys, *Consensys home page.* [Online]. Available: https://consensys.io/ (visited on 02/05/2024).

[117] C. Bräm, M. Eilers, P. Müller, R. Sierra and A. J. Summers, 'Rich specifications for Ethereum smart contract verification,' *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. DOI: 10.1145/3485523.

[118] E. Albert, S. Grossman, N. Rinetzky, C. Rodríguez-Núñez, A. Rubio and M. Sagiv, 'Taming callbacks for smart contract modularity,' *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428277.

[119] W. Ahrendt and R. Bubel, 'Functional verification of smart contracts via strong data integrity,' in *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2020, pp. 9–24, ISBN: 978-3-030-61467-6.

[120] B. Beckert, M. Herda, M. Kirsten and J. Schiffl, 'Formal specification and verification of Hyperledger Fabric chaincode,' in *3rd Symposium on Distributed Ledger Technology (SDLT-2018) co-located with ICFEM*, 2018, pp. 44–48.

[121] E. Androulaki, A. Barger, V. Bortnikov *et al.*, 'Hyperledger Fabric: A distributed operating system for permissioned blockchains,' in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18, Porto, Portugal: Association for Computing Machinery, 2018, ISBN: 9781450355841. DOI: 10.1145/3190508.3190538.

[122] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz and Y. Smaragdakis, 'MadMax: Surviving out-of-gas conditions in Ethereum smart contracts,' *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. DOI: 10.1145/3276486.

[123] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor and P. Saxena, 'Exploiting the laws of order in smart contracts,' in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 363–373, ISBN: 9781450362245. DOI: 10.1145/3293882.3330560.

[124] S. Lagouvardos, N. Grech, I. Tsatiris and Y. Smaragdakis, 'Precise static modeling of Ethereum "memory",' *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428258.

[125] S. Chaliasos, A. Gervais and B. Livshits, 'A study of inline assembly in solidity smart contracts,' *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, Oct. 2022. DOI: 10.1145/3563328.

[126] P. Antonino, J. Ferreira, A. Sampaio, A. Roscoe and F. Arruda, 'A refinement-based approach to safe smart contract deployment and evolution,' *Software and Systems Modeling*, pp. 1–37, 2024.

[127] Á. Hajdu and D. Jovanović, 'Solc-verify: A modular verifier for Solidity smart contracts,' in *Verified Software. Theories, Tools, and Experiments*, S. Chakraborty and J. A. Navas, Eds., Cham: Springer International Publishing, 2020, pp. 161–179, ISBN: 978-3-030-41600-3.

[128] S. Owre, J. M. Rushby and N. Shankar, 'PVS: A prototype verification system,' in *International Conference on Automated Deduction*, Springer, 1992, pp. 748–752.

[129] E. Brady, 'Idris, a general-purpose dependently typed programming language: Design and implementation,' *Journal of functional programming*, vol. 23, no. 5, pp. 552–593, 2013.

[130] A. Bove, P. Dybjer and U. Norell, 'A brief overview of Agda – a functional language with dependent types,' in *International Conference on Theorem Proving in Higher Order Logics*, Springer, 2009, pp. 73–78.

[131] B. C. Pierce, A. A. de Amorim, C. Casinghino *et al.*, *Logical Foundations* (Software Foundations), B. C. Pierce, Ed. Electronic textbook, 2022, vol. 1, Version 6.2, https://softwarefoundations.cis.upenn.edu.

[132] C. Pit-Claudel, 'Untangling mechanized proofs,' in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2020, Virtual, USA: Association for Computing Machinery, 2020, pp. 155–174, ISBN: 9781450381765. DOI: 10.1145/3426425.3426940. [Online]. Available: https://pit-claudel.fr/clement/papers/alectryon-SLE20.pdf.

[133] M. Lipovaca, *Learn you a Haskell for great good!: a beginner's guide.* no starch press, 2011. [Online]. Available: http://learnyouahaskell.com.

[134] D. Britten, *State monad diagram*, 2023. [Online]. Available: https://github.com/Coda-Coda/StateMonadDiagram (visited on 19/10/2023).

[135] R. Britten, *State monad bind*, 2013. [Online]. Available: https://en.m.wikibooks.org/wiki/File:State_Monad_Bind.svg (visited on 19/10/2023).

[136] J. Derrick and E. Boiten, *Refinement: Semantics, Languages and Applications*. Springer, 2018, vol. 95.

[137] J. Davies and J. Woodcock, 'Using Z,' *Specification Refinement and Proof. Series in Computer Science*, 1996.

[138] W. P. de Roever, *Data refinement : model-oriented proof methods and their comparison* (Cambridge tracts in theoretical computer science ; 47), eng. Cambridge, UK ; Cambridge University Press, 1998, ISBN: 0521641705.

[139] A. Chlipala, *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2022.

[140] Nomadic Labs, *Mi-Cho-Coq*. [Online]. Available: https://gitlab.com/nomadic-labs/mi-cho-coq (visited on 09/06/2019).

[141] Remix, *Solidity analyzers — Remix - Ethereum IDE 1 documentation*, n.d. [Online]. Available: https://remix-ide.readthedocs.io/en/latest/static_analysis.html#category-security (visited on 31/08/2023).

[142] N. F. Samreen and M. H. Alalfi, 'SmartScan: An approach to detect denial of service vulnerability in ethereum smart contracts,' in *2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2021, pp. 17–26. DOI: 10.1109/WETSEB52558.2021.00010.

[143] K. Elby, *KotET - post-mortem investigation*, Feb. 2016. [Online]. Available: https://www.kingoftheether.com/postmortem.html (visited on 31/08/2023).

[144] J. Feist, G. Grieco and A. Groce, 'Slither: A static analysis framework for smart contracts,' in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, IEEE, 2019, pp. 8–15.

[145] Remix, *Remix - Ethereum IDE*. [Online]. Available: https://remix.ethereum.org/ (visited on 23/05/2021).

[146] Kickstarter, PBC, *Kickstarter*, n.d. [Online]. Available: https://www.kickstarter.com/ (visited on 30/09/2023).

[147] J. Mackay, S. Eisenbach, J. Noble and S. Drossopoulou, 'Necessity specifications for robustness,' *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, Oct. 2022. DOI: 10.1145/3563317.

[148] M. Vladimirov and D. Khovratovich, *ERC20 API: An attack vector on the approve/transferfrom methods*, n.d. [Online]. Available: https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/ (visited on 10/10/2023).

[149] CoinMarketCap, *Ethereum price today, ETH to USD live price, marketcap and chart | CoinMarketCap*, n.d. [Online]. Available: https://coinmarketcap.com/currencies/ethereum/ (visited on 13/05/2024).

[150] T. Preston-Werner, *Semantic versioning 2.0.0*, 2013. [Online]. Available: https://semver.org/ (visited on 28/10/2023).

[151] P. Lam, J. Dietrich and D. J. Pearce, 'Putting the semantics into semantic versioning,' in *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2020, Virtual, USA: Association for Computing Machinery, 2020, pp. 157–179, ISBN: 9781450381789. DOI: 10.1145/3426428.3426922.