Certified compilation from a financial contract DSL to smart contract languages

Malthe Bisgaard Lange, 201404689

Master's Thesis, Computer Science
January 2021
Advisors: Bas Spitters, Danil Annenkov

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

The growing popularity of blockchains for various financial applications includes execution of financial contracts. Previous works have presented financial contract domain specific languages, DSLs, which can be represented and evaluated by smart contracts on blockchains. Smart contracts pose an inherent risk. Once a contract is deployed on a blockchain it cannot be changed. Therefore it is crucial to avoid mistakes in the contract logic, otherwise it could potentially lead to significant financial losses. In this work we present a certified compilation scheme, from a well established financial contract DSL with a formalized semantics, to a intermediate representation which we call CLVM. We present a definitional interpreter for CLVM and a Coq verified proof that compiled contracts follow the formalized semantics of the original contract. Utilizing this compilation scheme we define a smart-contract, which can represent and evaluate compiled contracts in accordance with the formalized semantics, in the ConCert framework.

By implementing the contract evaluator as a smart contract in ConCert, it is possible to formally reason about the evaluator's interaction with other smart contracts and users on the blockchain. We argue that the evaluator could be extracted to the Liquidity language. As the extraction capabilities of ConCert is still in development, it should eventually be possible to extract the contract to multiple functional smart-contract languages.

Resumé

Den stigende interrese for anvendelse af blockchainteknologi til håndetering af finansielle systemer indbefatter eksekvering af finansielle kontrakter. Tidligere værk har presenteret "domain specific languages", DSLs, som kan representerer og evalueres af smart-contracts på blockchains. Smart-contracts har en iboende risiko. Når først de er deployed på en blockchain kan de ikke ændres. Derfor er det afgørende at undgå fejl i kontraktens logik, ellers kan det fører til store finansielle tab.

I denne afhandling præsenterer vi en certifiet kompileringsprocedure fra et veletablert finanskontrakt DSL med en formaliseret semantik, til et intermediate language vi navngiver CLVM. Vi presenterer en definitional interpreter for CLVM og et bevis for at kompilerede kontrakter følger semantikken i den oprindelig kontrakt. Beviset er formaliseret og verificeret med Coq. Ved at anvende denne kompilerings procedure definere vi en smart-contract i frameworket ConCert.

Ved at implementere en CLVM evaluerings smart-contract i ConCert er det muligt at formalisere smart-contract interaktioner. Vi argumentere for at den definerede evaluerings smart-contract kan udtrækkes til det funktionelle smart-contract sprog Liquidity. Eftersom ConCert stadigvæk er under udvikling vil systemet potentielt kunne udtrækkes til flere funktionelle smart-contract sprog i fremtiden.

Malthe Bisgaard Lange,
Aarhus, January 2021.

Contents

Chapter 1

Introduction

The growing popularity of blockchains for various financial applications includes execution of financial contracts (like options, futures, etc.). Blockchain is an umbrella term for peer-to-peer protocols that allows multiple parties to maintain a shared ledger. Blockchain protocols allows the participants of the system to reach a shared consensus on the state of the ledger without assuming mutual trust, a trusted third-party or privileged parties. By evaluating the semantics of a financial contract on a blockchain, the parties can all agree on the state of the contract and ensure that it is followed, without the need for a third-party to ensure correctness. Modern blockchains support smart-contracts, which are state-machines distributed by the blockchain [8]. They are programmed in smart-contract languages, of which many are functional languages.

Existing works such as the article "Automated execution of financial contracts on blockchains" [8] investigate an execution model for a financial contact DSL on the Ethereum platform but does not provide a formal semantics of the language. The execution model presented in the article separates the concern of contract evaluation and contract management. Contract management is handled by a contract manager smart-contract, and the evaluation is handled by contract evaluator smart-contract. In their execution model a financial contract is signed by multiple parties by agreeing on a contract and a contract manager. The contract manager and contract evaluator is then deployed on the blockchain. The contract manager queries the contract evaluator, as required, with the information required for evaluating the contract. The contract evaluator then evaluates the contract and returns a trace of asset transfers, stipulated by the contract, to the contract manager. The contract manager can then enforce these stipulations with regard to some policy, which the parties agreed on at the time of signing.

In this work we examine if it is possible to develop a certified compilation scheme from a domain specific language (DSL) for financial contracts with a formalized semantics, to functional smart contract languages. By utilizing such a compilation scheme, can we define a smart-contract financial contract evaluator, which will evaluate a financial contract DSL in accordance with a formalized semantics?

We chose the financial contract DSL presented in the article "Certified symbolic management of financial multi-party contracts"[7]. It is a powerful language, capable of representing most actual financial contracts. It also has formal semantics specified in the Coq proof assistant, which makes it suitable for certified translation. After considering multiple target languages we ended up not translating directly to a smart contract language.

Instead we chose to implement an evaluator for CL as a smart contract in the ConCert framework, which is further described in section 2.3.2. Implementing the interpreter in the ConCert framework, instead of a specific smart contract language, could eventually allow for extraction to multiple target languages, preserving any guarantees proven in the ConCert framework. The ConCert framework, described in the article [6], contains a formalization of an abstraction of blockchains. Meaning once the CL interpreter has been implemented, it would be possible to examine how it would interact with different blockchains, state properties of this behaviour and prove them. However this is not examined further in this work.

Implementing and interpreter for CL in ConCert poses three main problems. The formal semantics of CL represents the language terms and expressions as recursively defined data types, which are not supported by most common smart contract languages, i.e. Solidity and Liquidity. Related works [8, 11] solve this problem by implementing interpreters for their own DSLs suitable for representation and evaluation on blockchain, utilizing records and pointers to mimic the behaviour of recursively defined data types. The second problem is serializing the CL terms and expressions. In ConCert all messages sent to and from smart contracts and all contract state must be serializable. Proving that the original representation of CL contracts as recursively defined data types is serializable is very difficult, and serializing the CL representation of environments and traces is impossible. Furthermore, most functional smart contract languages only support tail-recursive calls. This is also the case for Liquidity, which is currently the only language for which ConCert provides extraction. Due to these restrictions we decided to translate CL to an intermediate language which we call CLVM. CLVM is a simple stack based interpreted language. In section 3.3.4 we argue that the representation of CLVM can be extracted directly to Liquidity.

We formalized the CLVM language in Coq, along with a CL to CLVM compiler and a definitional interpreter for CLVM, which we call CLVMI. We define a smart contract in the ConCert framework which acts as a financial contract evaluator, by representing a financial contract as a compiled CL contract, and evaluating it with CLVMI. The design of this compilation scheme and the contract evaluator is presented in chapter 3. In chapter 4 we present and evaluate our certification of the compilation scheme and the evaluator. Our certification guarantee that if the evaluator is queried, as long as the environment provided by the contract manager is complete up to a certain date, then the stipulations returned by the evaluator will follow the formalized denotational semantics of CL.

This system satisfies the goal of developing a *certified* compilation scheme from a financial contracts DSL, with formalized semantics to functional smart contract languages. Using this result we were also able to define a financial contract evaluator as a smart-contract in the ConCert framework. In section 3.3.4 we argue why this ConCert formalization of the smart-contract is extractable to Liquidity. This answers our research question. The compilation scheme and financial-contract evaluator can be found on the project Github page [3].

In chapter 2 we describe and reflect on the contemporary literature on financial DSLs and blockchains. We consider the efforts and results of similar works and how they differ from ours. We also describe the technology used in this work i.e. Coq, ConCert, CL, Liquidity.

Chapter 2

State of the art

In this chapter we describe the contemporary literature and the current state of the art technologies related to automation of financial contracts using blockchain technology. We discuss their contributions and how they differ from ours.

2.1 The CL Language

The paper "Certified Symbolic Management of Financial Multi-party Contracts" [7] presents a domain specific language, DSL, for multi party financial contracts which we will call CL for "Contract Language". The paper states that DLSs for financial contracts are commonly used in the financial sector. Since the financial sector is "immensely sensitive to software bugs" [7] they present a language with formalized semantics and verified behavioral properties in the proof assistant Coq. This includes a denotational semantics, a type system and type inference procedure which ensures that the contracts follow the principle of causality and a reduction semantics which follow the denotational semantics. Here the principle of causality informally state that a transfer of assets at any given time cannot depend on information which will only be available at a later time. The language supports multiple parties and is identical to all involved parties, in contrast the DSLs akin to the work Jones et al [10] assume the perspective of the contract owner. This property is important for evaluating the DSL in a distributed system such as a blockchain. The language is expressive and well-suited for practical use, that along with the formalized semantics, temporal type system and verified properties makes CL an ideal candidate for representing and managing financial contracts on blockchains. For the same reason CL is also the basis for the DSL used in the paper "Automated Execution of Financial Contracts on Blockchains" [8] which also present a way of managing financial contracts on blockchain. But while they reimplement an interpreter for a language similar to CL our goal is to preserve the guarantees provided by the Coq formalization of CL.

CL is a declarative language based of the work by S. Peyton Jones [10], the syntax is simple and combinatorial. It consists of Contract terms which form arbitrarily large contracts from smaller sub contracts, and an expression sub-language. The semantics of a contract depend on an environment of observables, which represent all external information. Decisions made by parties, the current exchange rate between currencies and the value of stocks are all represented as observables in the environment. Given a well-typed contract and a satisfying environment the denotational semantics specify a trace of which transactions between parties should occur at a given time. The authors also provide a reduction semantics, allowing for contracts to be specialized to a given environment and advanced in time. This allows for the contract manager to continuously evolve the contract and determine the transactions. The reduction semantics are proven to follow the denotational semantics. In chapter 3 we describe the syntax and semantics for CL in greater detail.

2.2 Financial contracts on blockchain

While there currently is a growing interest in exploring financial applications of blockchains, the challenge of managing financial contracts on blockchains has only recently become a subject of research. The article by Egelund-Müllller [8] mentions that by 2016 several efforts had been made to implement specific types of financial contracts such as interest-rate swaps, but the earliest mention of a financial contract DSL on blockchain we have found in the literature is a work by Morten S mentioned by Egelund-Müllller [8], which should be an evaluator for a DSL akin to the one described by Jones et al. [10] on the Corda ledger. Since the reference provided by Egelund-Müller are no longer available we have not been able to verify this.

As inspiration and starting point for this work we have looked at the few and recent works which examine the challenge of managing financial contracts on blockchain. In the following subsections we discuss these works, which contributions are used in this work and how they differ.

2.2.1 Egelund-Müller

The primary inspiration for this work is the paper by Egelund-Müller et al [8]. Motivated by reducing the costs of management in the financial industry they present a framework for both evaluating and managing financial contracts specified in a CL-like DSL on the Ethereum blockchain. The reason for evaluating the contract semantics on blockchain is that the parties can agree on the state of the contract and the obligations that follows without requiring privileged parties or trusted third-parties.

The framework separates the concern of contract evaluation and contract management. The contract *evaluator* defines the semantics of the contracts and evaluates contracts accordingly. The contract *manager* then executes the obligations specified by the evaluator in accordance with a contract *strategy* chosen by the parties. The manager is responsible for providing the environment for evaluation and maintains the contact with the parties. At the signing of the contract the manager is given permission to transfer certain assets on behalf of the parties. This is possible by representing the assets as common smart contracts known as *tokens*. With the use of managers the execution of financial contracts can be automated to large extent which could lead to great cost-reductions in the financial sector.

Blockchain smart contracts are passive. They are only evaluated when a called by another contract or a peer on the blockchain. There are different ways of solving this problem. The authors address this issue by arguing that any party which stand to benefit from evaluation will make sure to call the contract manager and advance the contract. Since the state of the contract is public knowledge the parties can evaluate the contract off-chain and examine the consequences of advancing the contract, this is helpful for parties that might need to assess the value of their contract portfolio. Since the contracts will not be evaluated continuously, the obligations stated by the contract should be independent of the time intervals between evaluation. They name this property the *consistency principle*. In this work we ensure the property by verifying that the sum of transactions prescribed by any evaluation of a contract follows the denotational semantics of CL.

Our research is motivated by the same challenge as Egelund-Müller et al. We adopt the separation of concern between contract evaluation and contract management. There are many interesting problems to address regarding contract managers, but they are mostly concerned with blockchain specific technology and legislation. The challenge of implementing contract DSL evaluation is more novel and interesting, at least from a computer science perspective, and is therefore the primary concern of our work. Egelund-Müller et al. acknowledge the importance of an evaluator with mathematically certified semantics. They refer to a study by Luu et al [9] from 2016 which found that 8,833 out of the 19,366 Ethereum smart contracts they scanned had a one or more known vulnerabilities. By implementing a certified compiler for CL we provide this guarantee. Furthermore it is possible for parties to utilize the Coq formalization of CL semantics to prove properties of a specific contract before deployment, we can then guarantee that these properties are preserved.

2.2.2 Marlowe

Another work of examining financial contract management on blockchain is the paper "Marlowe: Financial Contracts on Blockchain" [11] from 2018. The authors argue that DSLs are suitable for specifying financial contracts, since their behaviour are more simply defined than general purpose languages, and are therefore easier to use for domain experts. They consider some of the limitations of blockchains. In their design they require all commitments of assets to be timed. So if the parties do not advance the contract, fail to follow it (by withholding payments etc.) or fail to make decisions, all committed funds are returned to their rightful owner.

They design the DSL Marlowe for the financial contracts, so that it is suitable for execution on blockchain and can allow for timeouts on transactions. The language is a simple combinatorial language which closely resembles CL. Marlowe does not provide an expression language for arithmetic computations, like CL. Instead they expect Marlowe to be embedded and the host language to provide these capabilities. They embed the DSL in Haskell so the Haskell functionality can be used when defining contracts.

They develop a tool for simulating interaction with Marlowe contracts, this allows users to examine how a contract will behave before actually deploying it on a blockchain. While certainly less user friendly our ConCert implementation does provide the same option, and even allows for stating properties of interaction with blockchains to be either formally proven or tested. The authors expect blockchain implementations of Marlowe to handle external observables by a trusted oracle third-party. Payments are automatically granted but must be manually redeemed by the appropriate party. The smart-contract can only request commitments, unlike in Egelund-Müller [8] where the contract manager can transfer funds on behalf of the parties, based on a consent given when signing.

The paper discuss why it would be possible to execute Marlowe contracts on blockchain, by compiling them to individual smart contracts. The details of the compilation however is never discussed. Meaning the paper aims to explore the concept of a financial contract DSL on blockchain but do not examine any particulars. The authors have made efforts to test and prove properties of Marlowe, like the authors of CL [7]. But since they do not consider how Marlowe should actually be compiled to a target language, proving that the compilation preserves these properties might be rather difficult. Since the most common smart contract languages do not support recursive data types and non-tail recursion, the translated contracts will be quite different from the source. We solve this problem by translating CL to an intermediate language, and it is not unlikely that the same would be required for Marlowe.

2.2.3 Wickramarachchi et al.

In an article titled "Efficiently Transform Contracts Written in Peyton Jones Contract Descriptive Language to Solidity" published in 2019, the authors Wickramarachchi et al. provide a compilation scheme from the financial contract DSL proposed by Peyton Jones, which they name CDL, to the solidity language. Like Egelund-Müller [8] they argue that requirement of a trusted party in contract evaluation introduces risks and overhead which could be avoided by evaluating on a blockchain.

The authors emphasize the importance of the composable nature of DSLs such as CDL and CL since it eases development for financial domain experts. The authors claim that the re-implementation of CL on blockchain by Egelund-Müller eliminates this composable nature. Therefore, contrary to the Marlowe system and Egelund-Müller, they define a new DSL and implement an interpreter as a solidity contract. They write a source to source compiler between CDL and their own DSL allowing users to specify their contracts directly in CDL thereby preserving the composability.

The authors argue for the correctness of their compiler with a very informal proof by induction. They examine the basic contracts, which are not made up by others, and conclude that the translated versions follows the semantics of the original. They then conclude that the same holds for composed contracts as long as the semantics are preserved for all sub-contracts. The approach is sound, but without utilizing a proof assistant like Coq to validate the proof, it is very easy to make mistakes.

This is the only article we have found in the contemporary literature which compiles from a DSL to a DSL interpreted by a smart contract. In this work we utilize the same strategy to preserve the properties of CL. Contrary to Wickramarachchi et al. we provide a formal proof for the compilers correctness verified in the Coq proof assistant.

2.3 Technologies used in this work

2.3.1 Coq

The Coq proof assistant [1] is a formal proof management system. The proof assistant provides a formal language based on the "Calculus of Inductive Constructions" which combines higher-order logic and a richly typed functional programming language. Using this language it is possible to define executable algorithms and software specifications along with mathematical definitions and theorems. From their tactics language users can write machine-checked proofs of specifications, thereby verifying the behaviour of algorithms with a small verification kernel.

The CL language along with its type-system and denotational semantics is formalized in Coq. In this work we specify the compilation algorithm from CL to CLVM in Coq along with the definitional interpreter. We present a specification of the compilation scheme which guarantees that compilation and interpretation of a CL contract satisfies the denotational semantics of the original CL contract. We consider this to be a strong certification of our compilation scheme. The parties using our system have a formal guarantee of the obligations stated in the contracts they sign, relying only on the relatively small and well-trusted kernel of the Coq proof-verification algorithm and the extraction procedure of ConCert.

2.3.2 ConCert

In the article "ConCert: A Smart Contract Certification Framework in Coq" by Annenkov et al. The authors present the smart contract execution framework ConCert. The authors argue that functional smart contract languages are becoming increasingly popular since they allow for formal reasoning. Once a smart contract has been deployed, they cannot be changed and even small mistakes in the contract might lead to huge financial consequences. This is a big incentive to provide a formal verification of a smart-contract's logic prior to deployment.

The execution framework contains a formalization of an abstraction of functional smart contract languages, and an abstract model of blockchains in the Coq proof assistant. By specifying a smart-contract in this framework it is possible to formally state and prove properties of smart-contracts using the tools provided by Coq. Including how the contract interacts with other contracts and peers on the blockchain. We chose the ConCert abstraction of smart-contracts to be the target of our compilation scheme. This allows for future work on formalizing the interaction between our contract evaluator and contract managers.

The extraction procedure from ConCert to concrete functional smart-contract languages is still in development. At the time of writing, extraction to the Liquidity language has just been supported. The development of Liquidity extraction was ongoing throughout our research. We want our financial-contract evaluator to be extractable to at least one concrete smart-contract language, therefor we made some design choices so that the evaluator would be extractable to Liquidity. We elaborate on this in chapter 3.

2.3.3 Liquidity

The Liquidity smart-contract language [5] is a fully-typed high-level functional language which compiles to the low-level language Michelson, which is provided for the Dunenetwork and Tezos. The language uses the syntax of OCaml. As of the time of writing ConCert only provides extraction to the Liquidity language. The language has several limitations which impacted the design of our compilation scheme, we elaborate on this in chapter 3.

Chapter 3

Designing the compilation scheme

In this chapter we discuss the process of designing and implementing the compilation scheme from CL to CLVM, from the initial goal and approach to the technical details of the final implementation.

3.1 Design overview

The goal of this project is to provide evaluation of a financial contract DSL on a blockchain while preserving the semantics of the original DSL through certified compilation. As discussed in section 2.1 we settled on the CL language as the financial contract DSL. CL has a formalized semantics in Coq which makes it a suitable candidate for certified compilation. In addition CL is expressive enough to describe common practical contracts and supports multiple parties without assuming a particular viewpoint, the later being favorable for evaluation in distributed system as described by Egelund-Müller [8]. CL has a type system and type-inference procedure which ensures causality and well-formedness of typeable contracts. Since we certify that compilation preserves the dynamic behaviour of contracts, we ensure that well-typed CL contracts evaluate to some value and satisfy the causality principle after compilation. Like most DSLs CL has a simple and compositional syntax which makes it accessible to domain-experts without much programming knowledge. By providing evaluation of the contracts through a compilation scheme like in the article Wickramarachchi [12] and not a re-implementation like Egelund-Müller [8] and Marlowe [11], we allow the end-users to write in the favorable original language. The final implementation and certification can be found on the project github page. [1]

Like Egelund-Müller we separate the concerns of contract management and contract evaluation. The contract evaluator is responsible for evaluating and advancing a contract according to the semantics of DSL. The manager queries the contract evaluator and provides it with an environment detailing the observables required for evaluation, the manager then executes the obligations dictated by the output of the evaluator according to some strategy which the parties signing the contract has agreed upon. In this project we only examine the challenge of providing an evaluator with certified semantics. We

[1] The final system can be found at the github page: https://github.com/malthelange/CLVM The repository is a fork of the ConCert framework, the README provides further details.

do however set some requirements for possible contract managers. The final system we present consists of a smart contract which acts as an evaluator by maintaining a contract DSL and last-queried-time, LQT, counter. When queried with a more recent time than the LQT and an environment which is complete to the point of the queried time, the contract is evaluated according to the environment and a trace of the stipulated transactions between LQT and the queried time is returned, the LQT is then advanced to the queried time. The evaluator we present can then be utilized as the core for any desired contract manager.

Instead of choosing a particular smart contract language as the target language we first decided on simply implementing an interpreter for CL as a smart contract in the ConCert framework. ConCert is still in development but it should eventually be possible to extract smart contracts from ConCert to multiple smart contract languages. As previously mentioned the other benefit of providing the evaluator in ConCert is the possibility examining the evaluators interactions on the blockchain, e.g. interactions with managers, and even proving guarantees of this behavior. Unfortunately this ended up being beyond the scope of this project but is left as possible future work.

As of the time of writing, ConCert only provides extraction of Liquidity and it is still in development. Therefore we want the finished smart contract to at least be extractable to Liquidity. The development of Concert's extraction capabilities was still ongoing throughout the course of this project, meaning some requirements for Liquidity extraction became evident late in process, leading to some revision of the design. But from the beginning it was clear that since Liquidity types closures and does not support recursively defined data types and non-tail recursive functions, it would not be possible to directly implement an interpreter for CL. Solidity and other common smart contract languages have the same limitations. Furthermore all state in ConCert smart contracts must be proven serializable which is a challenging task for recursively defined data types. CL represents environments and traces as total Coq functions, we expect the manager to provide the environment to the evaluator through messages. All messages between smart contracts must be a serializable data structure. Proving that the function representation of environments are serializable would be impossible. So instead of CL we require an intermediate language, which can be both represented and evaluated in Liquidity and ConCert.

We present CLVM, a simple stack-oriented interpretive language, along with an implementation of an interpreter for CLVM, CLVMI. We also present a compiler from CL to CLVM. We provide a verified Coq proof that all compiled CL contracts when executed on CLVMI will follow the denotational semantics of the original contract. This is elaborated on in chapter 4.

To summarize, the use case of our system is as follows: The parties define a CL contract and uses the temporal type-system of CL to ensure that the contract is well-formed and causal. The contract is then compiled to CLVM. The parties can then deploy a contract manager which will use the financial-contract evaluator which we defined, using the compiled CL contract as the representation of the financial contract. The manager will then query the evaluator, as required, with an environment and some natural number q. q specifies the time-offset from deployment, up to which the contract manager requires the stipulated transfer. The evaluator then evaluates the CLVM contract using CLVMI, and returns the trace of transfers from the CLVMI output trace, which are stipulated between the last queried time and q.

3.2 The CL language

The CL DSL provides a way to express financial contracts in a declarative programming language. The language consists of compositional contract terms and an expression sub-language. Financial contracts often depend on some external observable values, such as the stock exchange rate and actions taken by parties. Therefore the semantics of CL contracts are defined in relation to some environment providing these observables. The CL language provides a temporal type system and a type inference algorithm. Typeable contracts are ensured to be executeable and to follow the principle of causality, meaning that if the contract stipulates an asset transfer at a given time, then that stipulation will not depend on an observable which will only be available at a later time. Since we intend the end user of our system to specify their contracts in CL and not CLVM, we will rely on this type-system and not provide a type system for CLVM.

$$\begin{array}{ll}
\text{expressions} & e ::= x \mid z \mid b \mid \mathbf{obs}(l,t) \mid op(e_1,...,e_n) \mid \mathbf{acc}(\lambda x.e_1, d, e_2) \\
\text{contracts} & c ::= \emptyset \mid \mathbf{let}\ x = e\ \mathbf{in}\ c \mid d \uparrow c \mid c_1 \& c_2 \mid e \times c \mid \\
 & \quad a(p \to q) \mid \mathbf{if}\ e\ \mathbf{within}\ d\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \\
\text{where} & \quad x \in \mathbf{Var}, z \in \mathbb{Z}, b \in \mathbb{B}, l \in \mathbf{Label}, t \in \mathbb{Z} \\
 & \quad d \in \mathbb{N}, a \in \mathbf{Asset}, p, q \in \mathbf{Party}, op \in \mathbf{Op}
\end{array}$$

Figure 3.1: Syntax of the CL language, as used in our project

The syntax of CL is presented in figure 3.1. Here **Var** is a countably infinite set of variables, **Label** is the union of the two label domains $\mathbf{Label}_{\mathbb{B}}$ and $\mathbf{Label}_{\mathbb{Z}}$ which contain labels, **Op** is a set of common operations in boolean and integer arithmetic, **Asset** is a set of assets and **Party** is a set of parties. As mentioned previously CL has both a denotational and reduction semantics, the original paper [7] presents a formal mathematical overview of the denotational semantics of CL which we show in figure 3.2. In both figure 3.2 and 3.1 note that our adaptation represents quantities as integers instead of reals.

Here $[\![\tau]\!]$ denotes the semantic domain of the expression type τ. **Env** denotes environments, the domain of environments are functions $\rho : \mathbf{Label} \times \mathbb{Z} \to \mathbb{Z} \cup \mathbb{B}$ which map each time-offset z and label $l \in \mathbf{Label}_\tau$ to some value in $[\![\tau]\!]$. I.e. boolean labels are mapped to booleans and integer labels are mapped to integers. They define the promotion of an environment by the time-shift operator $/$, For $\rho \in \mathbf{Env}, z \in \mathbb{Z}$ then ρ/z maps the observables z days into the future, note that z can be negative.

For any typing environment Γ, $[\![\Gamma]\!]$ denotes the set of variable assignments consisting of mappings γ from variable names to $\mathbb{Z} \cup \mathbb{B}$ which satisfy $\gamma(x) \in [\![\tau]\!]$ iff $x : \tau \in \Gamma$, meaning all variable assignments in $[\![\Gamma]\!]$ must be compatible with the typing environment Γ.

Given some typing environment Γ such that $\Gamma \vdash e : \tau$ then the denotational semantics of e, written $\varepsilon[\![e]\!]$ is a mapping from a variable assignment and environment to $[\![\tau]\!]$. For some variable assignment γ and some environment ρ we write $\varepsilon[\![e]\!]_{\gamma,\rho}$ as short for $\varepsilon[\![e]\!](\gamma,\rho)$. For each operator $op \in \mathbf{Op}$ they define a semantic equivalent of the same arity $[\![op]\!]$. As an example $[\![+]\!]$ is the same as the normal $+$ operator for integers. The **acc** expressions shift environments in time, for $\rho \in \mathbf{Env}$ and $t \in \mathbb{Z}$ the *promotion* of ρ by t, written ρ/t, is defined as

$$\rho/t : (l,i) \mapsto (l,i+t) \qquad\qquad (i \in \mathbb{Z}, l \in \mathbf{Label})$$

$\gamma[x \mapsto v]$ denotes the remapping of x to v in γ.

The semantics of contracts specify a trace of asset transfers between parties at given times. The domain of transfers and traces are defined as

$$\mathbf{Trans} = \mathbf{Party} \times \mathbf{Party} \times \mathbf{Asset} \to \mathbb{Z}$$
$$\mathbf{Trace} = \mathbb{N} \to \mathbf{Trans}$$

Given a typing environment Γ the denotational semantics of a contract c which we write $C[\![c]\!]$ is a mapping of type

$$[\![\Gamma]\!] \times \mathbf{Env} \to \mathbf{Trace}$$

Like expressions we write $C[\![c]\!]_{\gamma,\rho}$ for $C[\![c]\!](\gamma,\rho)$. Note that the semantics of unit transfers guarantee that if the contract stipulates an asset transfer of a positive amount from party p to q, then the same asset transfer is stipulated from q to p, but with the negative amount.

3.2.1 Examples

To give a more intuitive understanding of CL the article by Bahr et al. [7] present some examples of common financial contracts represented in CL. In this subsection we present some of these examples. An American option allows some party to exercise an option within a fixed time frame. An example of an American option could be "Party X may buy 100 USD for a fixed rate of 7 DKK within the next 90 days". We present the exchange of assets as the contract:

$$100 \times (\mathrm{USD}(Y \to X) \& 7 \times \mathrm{DKK}(X \to Y))$$

We model the transfer of 1 USD with a unit-transfer base-contract, the corresponding payment of DKK is modeled by scaling a unit-transfer by the exchange rate of 7, using the scale-combinator. Since we want both transfers to happen simultaneously we combine the two transfer contracts with the &-combinator, and then the whole transaction is scaled by 100. This transaction should only happen if X exercises the option within 90 days, to model this we use the **if**...**within**...**then**...**else** combinator.

$$\begin{aligned}&\textbf{if obs}(X \text{ exercises option }, 0)\textbf{within } 90\\&\textbf{then } 100 \times (\mathrm{USD}(Y \to X) \& 7 \times \mathrm{DKK}(X \to Y))\\&\textbf{else } \emptyset\end{aligned}$$

The combinator checks whether the condition is true within the 90 days. In this case the condition is the observable (X exercises option, 0). Here "X exercises option" is a label defied in the label set $\mathbf{Label}_{\mathbb{B}}$, and 0 is the time-offset at which the observable should be observed. When the contract is evaluated d days into the future, the environment is time shifted d days into the future, therefore 0 will continue to be the correct time offset. If the condition is satisfied within the 90 days at day d, the resulting trace is that of the **then** subcontract evaluated with the environment ρ/d delayed by d. Meaning if X chooses to exercise the option 23 days after the signing of the contract, the contract will stipulate that the transaction should occur 23 days after the signing. If the condition is not satisfied the trace becomes that of the **else** sub-contract. Since we do not need to stipulate any transfers in this case we specify the empty base-contract $\emptyset$.

An Asian option allows the holder to buy some amount of an asset at the average price of that asset over a given time. An example of an asian contract could be "In 90 days party X may buy 100 USD from party Y for DKK at a rate specified by the average exchange rate of USD and DKK measured once every day over the last 30 days." In CL this contract could be stated as:

$$
\begin{aligned}
&90 \uparrow \textbf{if obs}(X\text{exercises option }, 0)\,\textbf{within}\ 0\\
&\textbf{then}\ 100 \times (\text{USD}(Y \to X) \& ((\textbf{acc}(\lambda r.r + \textbf{obs}(\text{FX}(USD, DKK), 0), 30, 0)/30) \times \text{DKK}(X \to Y)))\\
&\textbf{else}\ \emptyset
\end{aligned}
$$

To specify the average exchange rate we utilize the **acc** expression. While the denotational semantics of the expression is specified in figure 3.2, we give an informal explanation of how the expressions are evaluated. The e_2 expression, which in this example is 0, is evaluated with the environment, which we call ρ, time shifted by $-d$, which in our case is 30, i.e $\rho/-30$ and the variable r is remapped to the resulting value. Then from $\rho/-d+1$ to $\rho/0$ we evaluate e_1 which this example is $r + \text{FX}(USD, DKK, 0)$ and r is remapped to the result. So using r as the accumulator the **acc**-expression sums the exchange rate over the last 30 days and the results is divided by 30 giving us the average exchange rate. The whole contract is translated 90 days into the future from the signing date.

As a final example we show how the let-combinator can be used to store a value, observed at a given time, as a reference point to be used in time-shifted environments. This is the example provided in [7]. We need a contract which collapses if the exchange rate between two currencies raises too much in the next 30 days, relative to the rate observed at time 0.

$$
\begin{aligned}
&\textbf{let}\ x = \textbf{obs}(\text{FX(DKK, USD)}, 0)\\
&\textbf{in if obs}(\text{FX(DKK,USD)}, 0 \geq 1.1 \cdot x\ \textbf{within}\ 30)\\
&\textbf{then}\ \emptyset\ \textbf{else}\ \text{C'}
\end{aligned}
$$

Here we present the remaining contract as C'. The let binding maps the exchange rate at time 0 to the variable x, and so we are able to access this observable within the time-shifted environments of the if-within expression.

3.2.2 Importing CL to our project

To certify the semantics of CLVM we slightly modify the formal specification of CL [2], and we verify our semantics in accordance with this copied specification. Except the formal specification of CL keeps the types of Parties and Assets abstract only specifying that they require a boolean equivalence relation which must be decideable. For convenience we replace these abstract types and represent Parties and Assets with natural numbers which clearly follow the specification. We also replace the use of reals as transaction amounts with integers. This model is more fitting for blockchains where assets are most commonly represented as integers. Since none of the lemmas presented in the CL paper [7] utilize properties of reals which are not also properties of integers, this modification does not break the guarantees provided by the Coq verification of CL.

3.3 CLVM

As mentioned previously ConCert and Liquidity have several limitations which makes implementing an interpreter for the original CL representation and denotational semantics infeasible. The limitations we first considered were: No non-tail recursive functions, no recursive data types, no datatype for reals and recursive functions can only take a single argument. The last constraint is ignored since we can simply wrap and unwrap the desired arguments in tuples. As a solution we developed the simple stack-oriented interpreted language CLVM along with a compiler from CL to CLVM and the definitional interpreter CLVMI. The only intended use of CLVM is as target language for CL, so we do not need to bother with user-friendly properties such as readability and composability and we do not provide a type-system or a formal semantics. We expect the parties to specify and type check their financial contracts in CL. We prove the soundness of compilation and subsequent interpretation by CLVMI in chapter 4.

3.3.1 Original design

The original design of CLVM relied on the usefulness of reverse polish notation in stack-oriented languages. In reverse polish notation the operators follow the operands and there is no need for parenthesis to clarify possible ambiguities. The program can then be represented as a list of instructions, being either literals or operators. The instructions can be parsed from head to tail, pushing literals to the stack and having operators pop their arguments from the stack and pushing the result.

Like CL, CLVM consists of a contract language and an expression sub-language. During compilation the source CL contract is translated to a list of CLVM instructions, every expression present in the CL contract is in turn compiled to a list of expression-instructions. We define an interpreter for the expressions which is utilized by CLVMI. In the first iteration of CLVMI the stack maintained a list of higher-order functions of the domain :

$$[\![\Gamma]\!] \rightarrow \mathbf{EnvM} \rightarrow \text{Option Trace}$$

Here **EnvM** is the domain of CLVM environments. If the stack contained exactly one higher order function once the last instruction was evaluated, then this function was applied to the provided variable assignment and external environment yielding the final result. The reason why we maintained a stack of higher-order functions was that it allowed us to postpone the final evaluation until all time shifts were known. Consider the following example.

$$90 \uparrow (\mathrm{obs}(\mathrm{label}, 0) \times \mathrm{DKK}(X \to Y))$$

Rewriting this in the first iteration of CLVM we get the following list of instructions:

$$\begin{aligned} &[\mathrm{DKK}(X \to Y), \\ &\mathrm{Scale}[\mathrm{Obs}(l, z)], \\ &\mathrm{Translate}\ 90] \end{aligned}$$

The challenge here is that by the second instruction we do not know that the next instruction will be a translate. By keeping the stack as higher-order functions we can solve the problem by defining the interpretation of scale as described in figure 3.3.

Here `CI` is the interpretation function for CLVM instruction and `EI` is the interpreter for the expression sub-language.

This way we could postpone looking up the observable. Using this approach it was possible to define a simple semantics for CL with a one to one correspondence between CL and CLVM terms. This approach was favorable but during development we discovered that Liquidity types functions with their closures. This renders the functions produced by our combinators incompatible. We present a small example of this problem in listing 3.1.

Listing 3.1: Example of type mismatch in Liquidity due to closures

```
type env = (string, int) map
type stack = (env -> int) list

type inst =
    PushZero
    | PushZ of int

let rec interp (insts_s : inst list * stack) : stack =
    let insts, s = insts_s in
     match insts with
        | [] -> s
        | hd :: inst' ->
          match hd with
            | PushZero -> interp (inst', ((fun e -> 0) :: s))
            | PushZ i -> interp (inst', ((fun e -> i) :: s))
                (* Error :
                "types (string, int) map -> int and
                ((string, int) map -> int)[@closure :int] are not compatible"*)
```

The problem in this example is that we maintain a list of functions with type `(string, int) map -> int`, but since the function `(fun e -> i)` references the variable `i` from the outer scope, it is a closure and therefore incompatible with the list. We see the same problem with our combinators. As an example consider the function produced by `Scale` which references the value `v` from the outer scope. Since we require the interpreter to be extractable to Liquidity, we needed a new approach.

3.3.2 Final design

Since the interpreter must be extractable to Liquidity we can only maintain first-order values on the stack. This means we cannot simply represent all operators in reverse polish notation. In our final design we compile some contract combinators into multiple

CLVM instructions and place the instructions from the compiled subcontracts in between. The instructions prior to that of the subcontract perform computations which are maintained for reference during evaluation of the subcontract and then finally removed by the final instruction. The compilation procedure is best described by the Coq-code itself, which we present in figure 3.4 and 3.5 on page 22.

The syntax of CLVM is presented in figure 3.6. The definitional interpreters are presented in listings 1 and 2.

The listings present the Coq code which defines the interpreter functions, except we have rewritten some function and type names so to match the notation used in this work. We also removed some artefacts which do not impact evaluation, but which remain from previous iterations. In the expression interpreter we only show the case of $op = \text{Add}$. The remaining cases follow the same procedure of ensuring the right number of correctly typed operands are on the stack, and then popping the operands and pushing the evaluation of the operator on the stack. The listings should be considered detailed pseudo-code, for the actual Coq implementation we refer to the project github-page.

In listing 1, `EI` is the interpreter function for the expression sub-language, `instruction` is the type of expressions instructions, `Val` is the type of CL values which is an inductive type, defined as either `BVal b` or `ZVal z`, for $b \in \mathbb{B}, z \in \mathbb{Z}$. `G` is the type of CL variable assignments, since variables are referenced by indices the assignments are represented as a list of values. `ExtMap` is the type of CLVM environments, represented by finite-maps. The function `find_default` is defined as

$$\texttt{find\_default}((l,i),p) = \begin{cases} v & \text{if } \rho(l,i) = \texttt{Some } v \\ \texttt{ZVal 0} & \text{if } \rho(l,i) = \texttt{None} \text{ and } l \in \mathbf{Label}_{\mathbb{Z}} \\ \texttt{BVal false} & \text{if } \rho(l,i) = \texttt{None} \text{ and } l \in \mathbf{Label}_{\mathbb{B}} \end{cases}$$

The function `StackLookupEnv(n,g)` returns the index `n` in variable assignment `g`. `adv_map` is the time-shift operator / for CLVM environments.

In listing 2, `CI` is the interpreter function CLVMI, `CInstruction` is the type of CLVM instructions, `TraceM` is the type of CLVM traces, which are represented by finite maps from integers to transfers. T_0 is the trace which contains no transfers and $T_{p1,p2,c}$ is defined as

$$T_{p1,p2,c}(d)(p1',p2',c') = \begin{cases} 1 & \text{if } d = 0 \wedge c' = c \wedge p1' = p1 \wedge p2' = p2 \\ -1 & \text{if } d = 0 \wedge c' = c \wedge p1' = p2 \wedge p2' = p1 \\ \texttt{None} & \text{otherwise} \end{cases}$$

The function `scalem` scales all transferred amounts stipulated in a trace by some integer. It is defined as

$$scalem(z,T)(d)(p,q,a) = \begin{cases} z \cdot v & \text{if } T(d)(p,q,a) = \texttt{Some } v \\ \texttt{None} & \text{otherwise} \end{cases}$$

The function addm is the union of two traces. If a transfer is present in both cases, their amounts are added. It is defined as

$$addm(T_1,T_2)(d)(p,q,a) = \begin{cases} v & \text{if } T_1(d)(p,q,a) = \texttt{Some } v \land T_2(d)(p,q,a) = \texttt{None} \\ v & \text{if } T_1(d)(p,q,a) = \texttt{None} \land T_2(d)(p,q,a) = \texttt{Some } v \\ v_1 + v_2 & \text{if } T_1(d)(p,q,a) = \texttt{Some } v_1 \land T_2(d)(p,q,a) = \texttt{Some} v_2 \\ \texttt{None} & \text{otherwise} \end{cases}$$

The function `delaym(n,t)` is the CLVM equivalent of the CL *delay* function. It postpones all stipulations of trace t by n days. The $wsem(le,n,\gamma,\rho)$ function is defined by a tail-recursive Coq function which checks if the list of CLVMI expression instructions le evaluates to true with ρ time-shifted from 0 to n. If so, then (true,i) is returned, where $i = n - d$ with d being first time-shift to evaluate to true. If not then (false,n) is returned. The function is defined as

$$wsem(l_e,n,\gamma,\rho) = \begin{cases} \texttt{Some (True,}n\texttt{)} & \text{if } EI(l_e,[],\gamma,\rho) = \texttt{Some (BVal True)} \\ wsem(l_e,(n-1),\gamma,\rho/1) & \text{if } EI(l_e,[],\gamma,\rho) = \texttt{Some (BVal False)} \\ \texttt{None} & \text{otherwise} \end{cases} \tag{3.1}$$

Given these definitions the code should be readable given a basic understanding of Coq. The compilation procedure and the definitional interpreter provide a compact description of the compilation scheme, in the rest of this section we elaborate on the more interesting design choices.

Due to the problems with Liquidity's typed closures and the fact that we need all data structures to be proven serializable in Coq, we cannot represent external environments and traces as functions like in CL. Instead we use the Coq implementation of finite maps from the Coq extended standard library, stdpp, provided by the Iris framework. This implementation has several favorable properties which we utilize in chapter 4. The interpreter is defined as a tail-recursive function. Like in CL the variable assignments are represented as a list, and variable references are given by indices. Since we need to handle nested changes to the external environment we maintain a list of environments. We also maintain a list of natural numbers and a counter to propagate information when handling If-combinators. The stack is represented as a list of traces. The interpreter for the expression language is also a tail-recursive function maintaining the variable assignments, environment, and a stack of values. If the contracts are not well-formed the interpreters will return `None`.

The expression interpreter is mostly a straightforward adaptation of the CL denotational semantics, except for the handling of CL `Acc` expressions. In section 3.2.1 we show through an example how `Acc e1 d e2` can be computed as d computations of either `e1` or `e2` with shifting environments. The compiler unfolds the CL expression to these computation steps and utilize intermediate instructions to shift the environments and move results between stack and variable assignment. The variable introduced by the expression is referenced by the first index of the variable assignment.

The compilation scheme handles the `Translate` combinator of CL by compiling the sub contract and wrapping the result in a `Translate n` and a `TranslateEnd n` instruction. `Translate n` peeks the top element of the environment stack ρ and pushes the time-shifted ρ/n on top, then the subcontract is evaluated given the correct environment and finally `TranslateEnd n` delays the result by n and pops ρ/n restoring the stack. Note that we need a stack of environments since we might encounter nested `Translate` combinators. Similarly the CL `Let` combinator is handled by wrapping the subcontract in `Let es` and `LetEnd` expressions which handles the variable assignments.

The `If .. within .. then .. else ..` combinator is compiled into a `If es n` instruction followed by the first compiled subcontract, then a `Then` instruction followed by second compiled subcontract and finally a `IfEnd` instruction. The first instruction evaluates $wsem(es, n, \gamma, \rho)$ where ρ is the top element of the environment stack and γ is the current variable assignment. Using this result ρ/d is pushed on the environment stack and the bf-counter is set to either 0 or 1 to ensure evaluation of the right branch and d is pushed on the w-stack. If the bf counter is greater than 0 most instructions are ignored, the exceptions being `If`, `Then` and `IfEnd`. `If` increases the counter in order to handle nested if-within contracts. `Then` switches the counter between 0 and 1 and ignores higher values. `If end` checks if the counter is less than 2 if so the top element of the w-stack, d, is popped and the top element of the stack is delayed by d and the environment stack is restored. If the counter is higher than 1 we are in the nested case and so the counter is decreased.

$$\boxed{\mathcal{E}\left[\!\left[e \right]\!\right] : \left[\!\left[\Gamma \right]\!\right] \times \mathsf{Env} \to \left[\!\left[\tau \right]\!\right]}$$

$$\mathcal{E}\left[\!\left[r \right]\!\right]_{\gamma,\rho} = r; \quad \mathcal{E}\left[\!\left[b \right]\!\right]_{\gamma,\rho} = b; \quad \mathcal{E}\left[\!\left[x \right]\!\right]_{\gamma,\rho} = \gamma(x)$$

$$\mathcal{E}\left[\!\left[\mathbf{obs}(l,t) \right]\!\right]_{\gamma,\rho} = \rho(l,t)$$

$$\mathcal{E}\left[\!\left[op(e_1,\dots,e_n) \right]\!\right]_{\gamma,\rho} = \left[\!\left[op \right]\!\right] \left(\mathcal{E}\left[\!\left[e_1 \right]\!\right]_{\gamma,\rho},\dots,\mathcal{E}\left[\!\left[e_n \right]\!\right]_{\gamma,\rho}\right)$$

$$\mathcal{E}\left[\!\left[\mathbf{acc}(\lambda x.\, e_1, d, e_2) \right]\!\right]_{\gamma,\rho} = \begin{cases} \mathcal{E}\left[\!\left[e_2 \right]\!\right]_{\gamma,\rho} & \text{if } d = 0 \\ \mathcal{E}\left[\!\left[e_1 \right]\!\right]_{\gamma[x\mapsto v],\rho} & \text{if } d > 0 \end{cases}$$

$$\text{where} \qquad v = \mathcal{E}\left[\!\left[\mathbf{acc}(e_1, d-1, e_2) \right]\!\right]_{\gamma,\rho/-1}$$

$$\boxed{\mathcal{C}\left[\!\left[c \right]\!\right] : \left[\!\left[\Gamma \right]\!\right] \times \mathsf{Env} \to \mathbb{N} \to \mathsf{Party} \times \mathsf{Party} \times \mathsf{Asset} \to \mathbb{R}}$$

$$\mathcal{C}\left[\!\left[\emptyset \right]\!\right]_{\gamma,\rho} = \lambda n.\lambda t.0$$

$$\mathcal{C}\left[\!\left[e \times c \right]\!\right]_{\gamma,\rho} = \lambda n.\lambda(p,q,a).\mathcal{E}\left[\!\left[e \right]\!\right]_{\gamma,\rho} \cdot \mathcal{C}\left[\!\left[c \right]\!\right]_{\gamma,\rho}(n)(p,q,a)$$

$$\mathcal{C}\left[\!\left[c_1 \,\&\, c_2 \right]\!\right]_{\gamma,\rho} = \lambda n.\lambda t.\mathcal{C}\left[\!\left[c_1 \right]\!\right]_{\gamma,\rho}(n)(t) + \mathcal{C}\left[\!\left[c_2 \right]\!\right]_{\gamma,\rho}(n)(t)$$

$$\mathcal{C}\left[\!\left[d \uparrow c \right]\!\right]_{\gamma,\rho} = delay(d, \mathcal{C}\left[\!\left[c \right]\!\right]_{\gamma,\rho}), \quad \text{where}$$

$$delay(d,f) = \lambda n. \begin{cases} f(n-d) & \text{if } n \geq d \\ \lambda x.0 & \text{otherwise} \end{cases}$$

$$\mathcal{C}\left[\!\left[a(p \to q) \right]\!\right]_{\gamma,\rho} = \begin{cases} \lambda n.\lambda t.0 & \text{if } p = q \\ unit_{a,p,q} & \text{otherwise,} \quad \text{where} \end{cases}$$

$$unit_{a,p,q}(n)(p',q',b) = \begin{cases} 1 & \text{if } b = a, p = p', q = q', n = 0 \\ -1 & \text{if } b = a, p = q', q = p', n = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{C}\left[\!\left[\mathbf{let}\ x = e\ \mathbf{in}\ c \right]\!\right]_{\gamma,\rho} = \mathcal{C}\left[\!\left[c \right]\!\right]_{\gamma[x\mapsto v],\rho}, \ \text{where } v = \mathcal{E}\left[\!\left[e \right]\!\right]_{\gamma,\rho}$$

$$\mathcal{C}\left[\!\left[\mathbf{if}\ e\ \mathbf{within}\ d\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \right]\!\right]_{\gamma,\rho} = iter(d,\rho), \ \text{where}$$

$$iter(i,\rho') =$$

$$\begin{cases} \mathcal{C}\left[\!\left[c_1 \right]\!\right]_{\gamma,\rho'} & \text{if } \mathcal{E}\left[\!\left[e \right]\!\right]_{\gamma,\rho'} = true \\ \mathcal{C}\left[\!\left[c_2 \right]\!\right]_{\gamma,\rho'} & \text{if } \mathcal{E}\left[\!\left[e \right]\!\right]_{\gamma,\rho'} = false \wedge i = 0 \\ delay(1, iter(i-1, \rho'/1)) & \text{if } \mathcal{E}\left[\!\left[e \right]\!\right]_{\gamma,\rho'} = false \wedge i > 0 \end{cases}$$

Figure 3.2: Semantics of the CL language. This figure is taken directly from the original CL paper [7]

```
CI (Scale e)::l f::stack g p = do v <- EI le [] g p;
    CI l (Some (fun g' p' => v * f(g',p')))::stack g p
```

Figure 3.3: Scale-semantics for first iteration of CLVMI. Here the stack consists of higher-order functions.

```
Fixpoint CompileC (c : Contr) : option (list CInstruction) :=
  match c with
  | Zero => Some [CIZero]
  | Transfer p1 p2 a => Some [CITransfer p1 p2 a]
  | Scale e c => do es <- CompileE e;
                do s <- CompileC c;
                Some (s ++  [CIScale (es)])
  | Both c1 c2 => do s1 <- CompileC c1;
                 do s2 <- CompileC c2;
                 Some (s2 ++ s1 ++ [CIBoth] )
  | Translate n c1 => do s <- (CompileC c1) ;
                     (Some ([CITranslate n] ++ s ++ [CITranslateEnd n]))
  | If e n c1 c2 => do es <- CompileE e;
                   do s1 <- CompileC c1;
                   do s2 <- CompileC c2;
                   Some ([CIIf es n] ++ s2 ++ [CIThen] ++ s1 ++ [CIIfEnd])
  | Let e c => do es <- CompileE e;
              do s <- CompileC c;
              Some ([CILet es] ++ s ++ [CILetEnd])
  end.
```

Figure 3.4: The compilation procedure from CL to CLVM

```
Fixpoint CompileE (e : Exp) : option (list instruction) :=
  match e with
  | OpE op args => match op with
                | BLit b => match args with | [] => Some [IPushB b] | _ => None end
                | ZLit z => match args with | [] => Some [IPushZ z] | _ => None end
                | Neg => match args with
                        | [exp1] =>
                          do s1 <- CompileE exp1;
                          Some (s1 ++ [IOp Neg])
                        | _ => None end
                | Not => match args with
                        | [exp1] =>
                          do s1 <- CompileE exp1;
                          Some (s1 ++ [IOp Not])
                        | _ => None end
                | Cond => match args with
                         | [exp1; exp2; exp3] =>
                           do s1 <- (CompileE exp1);
                           do s2 <- (CompileE exp2);
                           do s3 <- (CompileE exp3);
                           Some (s3 ++ s2 ++ s1 ++ [IOp Cond])
                         | _ => None end
                | op => match args with
                       | [exp1; exp2] =>
                         do s1 <- CompileE exp1;
                         do s2 <- CompileE exp2;
                         Some ( s2 ++ s1 ++ [IOp op]) | _ => None end
                end
  | Obs l i => Some [IObs l i]
  | VarE v => Some [IVar (translateVarToNat v)]
  | Acc e1 d e2 => match d with
                | O => do s2 <- (CompileE e2);
                      Some ([IAccStart1 (Z.of_nat d)] ++ s2)
                | _ => do s1 <- (CompileE e1);
                      do s2 <- (CompileE e2);
                      Some ([IAccStart1 (Z.of_nat d)] ++ s2 ++ [IAccStart2] ++
                      (repeat_app (s1 ++ [IAccStep]) (d-1)) ++ s1 ++ [IAccEnd])
                end
  end.
```

Figure 3.5: The compilation procedure from CL expression sub language to the CLVM expression sub language

expressions $e ::=$	PushZ z | PushB b | Obs (l, z) | Op op | AccStart$_1$ z |
	AccStart$_2$ | AccStep | AccEnd | Var n
contracts $c ::=$	$\emptyset$ | $a(p \rightarrow q)$ | Scale es | Both | Translate n | TranslateEnd n |
	Let es | LetEnd | If es n | Then | IfEnd
where	$z \in \mathbb{Z}, l \in \textbf{Label}, op \in \textbf{Op}, n \in \mathbb{N}, a \in \textbf{Asset}, p, q \in \textbf{Party}, es \in \textbf{list e}$

Figure 3.6: Syntax of CLVM

```
Fixpoint EI (l : list instruction) (stack : list (option Val))
            (g: G) (p: ExtMap) : option Val :=
  match l with
  | [] => match stack with [val] => val | _ => None end
  | hd::tl =>
    match hd with
    | IPushZ z => EI tl ((Some (ZVal z))::stack) g p
    | IPushB b => EI tl ((Some (BVal b))::stack) g p
    | IObs l i => EI tl ((Some (find_default (l,i) p))::stack) g p
    | IOp op => match op with
               | Add => match stack with
                       | (Some (ZVal z1))::(Some (ZVal z2))::tl2 =>
                        EI tl (Some (ZVal (z1 + z2))::tl2) g p
                       | _ => None
                       end
                ...
               end
    | IVar n => do v <- (StackLookupEnv n g); EI tl ((Some v)::stack) g p
    | IAccStart1 z => EI tl stack g (adv_map (-z) p)
    | IAccStart2 => match stack with (Some v)::tl2 => EI tl tl2 (v::g) (adv_map 1 p)  | _ => None end
    | IAccStep => match stack with (Some v)::tl2 => let g' := List.tl g in
                                                    EI tl tl2 (v::g') (adv_map 1 p)  | _ => None end
    | IAccEnd => EI tl stack (List.tl g) p
    end
  end.
```

Listing 1: The interpreter for CLVM expressions used by the definitional interpreter CLVMI of CLVM

```
Fixpoint CI (l : list CInstruction) (stack : list (option TraceM))
         (g : G) (P: list ExtMap) (ws : list nat) (bf : nat): option TraceM :=
  match l with
  | [] => match stack with [res] => res | _ => None end
  | hd::tl =>
    match hd with
    | Zero => match bf with
             | 0 => CI tl (T0::stack) g P ws bf
             | _ => CI tl stack g P ws bf
             end
    | Transfer p1 p2 c => match bf with
                         | 0 => CI tl ((Some T_{p1,p2,c} )::stack) g P ws bf
                         | _ => CI tl stack g P ws bf
                         end
    | Scale le => match bf with
                 | 0 =>
                   match stack with
                   | hd2::tl2 =>
                     do hd2' <- hd2;
                     do et <- hd_error P;
                     do v <- (EI le [] g et false);
                     do z <- toZ v;
                     CI tl (Some(scalem z hd2')::tl2) g P ws bf
                   | [] => None
                   end
                 | _ => CI tl stack g P ws bf
                 end
    | Both => match bf with
             | 0 =>
               match stack with
               | t1::t2::tl2 =>
                 let trace := (liftM2 addm t1 t2) in
                 CI tl (trace::tl2) g P ws bf
               | _ => None
               end
             | _ => CI tl stack g P ws bf
             end
    | Translate n => match bf with
                    | 0 =>
                      do et <- hd_error P;
                      let et' := (adv_map (Z.of_nat n) et) in
                      CI tl stack g (et'::P) ws bf
                    | _ => CI tl stack g P ws bf
                    end
    | TranslateEnd n => match bf with
                       | 0 =>
                         match stack with
                         |t::tl2 => match P with
                                   | et::P' =>
                                   match t with
                                   | Some t' =>
                                     let
                                       trace := (delaym n t')
                                     in
                                     CI tl ((Some trace)::tl2) g P' ws bf
```

```
                                | None => CI tl (None::tl2) g P' ws bf
                                end
                              | p_ => None
                              end
                     | _ => None
                     end
                   | _ => CI tl stack g P ws bf
                   end
| Let le => match bf with
           | O =>
             do et <- hd_error P;
             do v <-  (EI le [] g et false);
             CI tl stack (v::g) P ws bf
           | _ => CI tl stack g P ws bf
           end
| LetEnd => match bf with
           | O => CI tl stack (List.tl g) P ws bf
           | _ => CI tl stack g P ws bf
           end
| If le n => match bf with
            | O => match P with
                  | et::P' =>
                    match wsem le n g et with
                    | Some (branch, d_left) =>
                      let d_p := (n - d_left)%nat in
                      let et' := adv_map (Z.of_nat d_p) et in
                      CI tl stack g (et'::P) (d_p::ws) (if branch then 1 else 0)
                    | _ => None
                    end
                  | _ => None
                  end
            | bf' => CI tl stack g P ws (S bf)
            end
| Then => match bf with
         | O => CI tl stack g P ws 1
         | (S O) => CI tl stack g P ws O
         | _ => CI tl stack g P ws bf
         end
| IfEnd => match bf with
          | O => match stack with
                | t1::tl2 => match ws with
                           | w::ws' =>
                             let d_p := w in
                             do t1' <- t1;
                             let trace := delaym d_p t1' in
                             CI tl ((Some trace)::tl2) g (List.tl P) ws' O
                           | _ => None
                           end
                | _ => None
                end
          | (S O) => match stack with
                    | t1::tl2 => match ws with
                               | w::ws' =>
                                 let d_p := w in
                                 do t1' <- t1;
                                 let trace := delaym d_p
                                                     t1' in
```

```
                                    CI tl ((Some trace)::tl2) g (List.tl P) ws' O
                                  | _ => None
                                  end
                        | _ => None
                        end
              | (S bf') => CI tl stack g P ws bf'
              end
    end
  end.
```

Listing 2: The definitional interpreter CLVMI of CLVM

3.3.3 Implementing the contract evaluator

So far we have specified the compilation scheme in Coq, the next step is implementing the contract evaluator as a smart contract in the ConCert framework. Smart contracts in ConCert consists of a `init` and `receive` function along with `Setup`, `State` and `Message` data types. When the contract is deployed the `init` function is evaluated with a `Setup`, `Chain` and `ContractCallContext` argument. `Chain` represents properties of the blockchain, such as the current slot number and account balances and `ContractCallContext` represents properties of the contract call, such as the address of the caller. `init` then computes the initial state of the contract. When the contract is called the caller might supply a `Message`, the `receive` function is then evaluated with a `option Message`, `State`, `Chain` and `ContractCallContext`. Here the `State` argument is the current state of the contract. The `receive` function then computes the new state of the contract along with a list of possible actions. The function might also evaluate to `None`, closing the smart contract. The computed actions might be asset transfers, calls to other contracts with some serialized value or deployment of new contracts. The `Setup`, `State` and `Message` types must be serializable.

As mentioned in section 3.1 we separate the concern of contract management into contract evaluation and contract management. There are many aspects of defining contract managers that are beyond the scope of this project, so we provide the evaluator so that it might be used for different managers.

In the the evaluator smart contract, `Setup` is a record containing the list of instructions representing the contract. `State` is a record maintaining the contract, an integer which keeps track of the last queried time and the address of the manager contract which deployed the evaluator. `Message` is defined as an inductive datatype with only one constructor containing an environment and a natural number, we also define a similar inductive datatype consisting of an `option Trace` to be used as response. The ConCert framework provides powerful Coq tactics that can automatically derive the Serializable predicate for most data types and combinators. The only proof we needed to provide was for the `ObsLabel` datatype for observation labels which is defined as:

```
Inductive ObsLabel : Set := LabZ (l: Z) | LabB (l: Z).
```

We prove that the set of `ObsLabel`'s is countable by proving it has decideable equality and is injective to disjoint sums of integers. This property is sufficient for ConCert to automatically derive proof of serializability.

When deployed the `init` function generates a state consisting of the provided contract, the time 0 and the address provided by the caller context. The last queried time, LQT, is set to 0 since all time in CLVM is measured as offsets from the time of the signing, for practical use this might require further consideration. When the contract is called, the `receive` function will check that an environment and a time offset is provided, that the time offset is larger than the LQT and that the caller is indeed the manager. If so, the contract is evaluated using CLVMI with the provided environment. Since we assume the manager has already handled all previously stipulated transfers, the transfers between LQT and the queried time is extracted from the CLVMI result and sent to the manager as a response, the LQT is then set to the queried time.

This concludes the design of our system, and completes our goal of providing a certified compilation scheme for a financial contract DSL to blockchain smart contracts. The code for the ConCert smart contract is presented in figure 3.7. In the next section we discuss how we might extract the smart contract to liquidity.

3.3.4 Extraction

In this section we discuss how our the compilation scheme might be extracted from Coq and how the financial-contract evaluator might be extracted from ConCert. Given the time-frame of this project we were not able to extract the code, so instead we argue why this is possible.

The Coq proof assistant [1] provides an extraction procedure to Objective Caml, Haskell or Scheme. The compilation from CL to CLVM does not need to happen on the blockchain, and so it should be possible to directly use this extraction procedure for the compiler functions.

As of the time of writing, ConCert has just recently provided support for extraction to Liquidity. This extraction poses some restraints on the Coq code used in ConCert smart contracts. These restrictions are mentioned in the section 3.3.2. In addition to these constraint, we need to provide mappings from standard Coq types and functions to their Liquidity equivalents. This includes pairs, lists, integers, booleans and most importantly finite maps. The functions and types defined uniquely for our system can be extracted automatically. The formalization of finite maps provided by the stdpp-library is an abstraction of any finite map and makes heavy use of Coq's rich type system. In order to map stdpp finite maps to the Liquidity equivalent, we need to help the type system by providing lookup interfaces for each kind of finite map we use. In order to extract our system with the current extraction procedure, we would also need to slightly change our definition of the option Monad, and replace the bind-combinator with basic pattern-matching.

The authors of ConCert provide an example of Liquidity extraction for a small stack-based arithmetic language and its definitional interpreter. This language is essentially a subset of the expression interpreter in our system. While our system is vastly bigger and would require several small rewrites and additional mappings, it does not require any functionality that this small example does not. The example can be found on the ConCert Github page [4]. Given this example we feel confident that we could extract the financial-contract evaluator to Liquidity using the extraction procedure provided by ConCert.

```
Record Setup :=
  build_setup {
      setup_contract : list CInstruction;
    }.
Record State :=
  build_state {
      contract : list CInstruction;
      LQT : nat;
    }.

Inductive Msg :=
| update : ExtMap -> nat -> Msg.

Inductive Response :=
| response : option TraceM -> Response.

Definition init (chain : Chain) (ctx: ContractCallContext) (setup: Setup) : option State :=
  let contract := setup_contract setup in
  Some (build_state contract 0).

Definition extract_element (trace : TraceM) (index : nat) : TraceM :=
  match FMap.find index trace with
  | Some trans => FMap.add index trans empty_traceM
  | None => empty_traceM
  end.

Fixpoint cutTrace (trace : TraceM) (startTime idx: nat) : TraceM :=
  match idx with
  | 0%nat => empty_traceM
  | S n => add_traceM (extract_element trace (startTime + idx)%nat) (cutTrace trace startTime n)
  end.

  Definition receive
           (chain : Chain) (ctx : ContractCallContext)
           (state : State) (msg : option Msg)
  : option (State * list ActionBody) :=
  match msg with
  | Some (update ext t) => if (Nat.ltb (LQT state) t) then
                            let res := do trace <- (vmC (contract state) [] ext);
                                       Some (cutTrace trace (LQT state) t)
                            in
                            Some (state <|LQT := t |>,
                                 [act_call (ctx_from ctx) 0 (serialize (response res))])
                          else
                            None
  | _ => Some (state,[])
  end.

Program Definition evaluation_contract : Contract Setup Msg State :=
  build_contract init _ receive _.
```

Figure 3.7: The final contract evaluator, implemented as a smart contract in ConCert.

Chapter 4

Certifying the compilation scheme

4.1 Overview

In the previous chapter 3, we presented the CLVM language, a CL to CLVM compiler and the interpreter CLVMI. We also presented a financial contract evaluator implemented as a ConCert smart contract which provided continuous evaluation of financial contracts represented in CLVM using CLVMI. The goal of this project is to provide such a compilation scheme along with a proof of soundness, formalized in Coq, which certifies the behaviour of the compiled contracts. In this chapter we present the proof of soundness and discuss the strength of this certification.

For this chapter we introduce some additional notation. $Com(c)$ is the procedure which compiles the CL contract, c, into a list of CLVM instructions, likewise $ECom(e)$ compiles the CL expression e into a list of CLVM expressions. The CLVMI interpretation of a compiled instruction list l with the variable assignment γ, an environment stack P, the stack $stack$, bf counter bf and the list of natural numbers ws, is written as $CI[\![l]\!]_{\gamma,P,stack,ws,0}$. When the constructor `Some` is given by the context, we omit it for readability.

The interpretation of CLVM expressions l by CLVMI with the variable assignment γ the environment ρ and the stack $stack$ is written $EI[\![l]\!]_{\Gamma,\rho,stack}$

In the CL semantics, environments are represented by a total function. In CLVM we need the representation to be serializable, so instead we represent the environments as finite maps. CLVMI assumes all observables not present in this finite map is a default value. To state the relation between the two different environment representations we introduce a function Te specified as

$$Te(\rho)(l,i) = \begin{cases} v & \text{if } \rho(l,i) = \texttt{Some v} \\ \texttt{ZVal 0} & \text{otherwise and } l \in \mathbf{Label}_{\mathbb{Z}} \\ \texttt{BVal false} & \text{otherwise and } l \in \mathbf{Label}_{\mathbb{B}} \end{cases}$$

Here $\rho \in \mathbf{EnvM}, l \in \mathbf{Label}, i \in \mathbb{Z}, v \in \mathbf{Val}$. Where **EnvM** is the domain of CLVM environments. It is the responsibility of the contract manager to provide a finite map of all observables required for the desired evaluation of the contract. In CL traces are also represented as total functions. We use the same approach and define CLVM traces as finite maps of the type $\mathbb{N} \to$ **option TransM** here **TransM** are transfers represented as finite maps of the type $\mathbf{Party} \times \mathbf{Party} \times \mathbf{Asset} \to \mathbb{Z}$, all transfers not present in the finite map is assumed to be the transfer which maps any input to 0. We relate the two representations of traces by the function Tt

$$Tt(T)(d,p_1,p_2,a) = \begin{cases} z & \text{if } T(d)(p_1,p_2,a) = z \\ 0 & \text{otherwise} \end{cases}$$

Here $T \in \mathbf{TraceM}, p_1, p_2 \in \mathbf{Party}, a \in \mathbf{Asset}, z \in \mathbb{Z}$. Where **TraceM** is the domain of CLVM traces.

The certification is provided by Theorem 1. This theorem states that compiled contracts interpreted by CLVMI always produce the same trace as the denotational semantics of the original contract. The Coq formalization and verified proof of Theorem 1 can be found in the project repository [1]. In sections 4.3 and 4.4 we provide an informal presentation of the proof for Theorem 1, presenting the interesting segments of the proof while avoiding the technical details required of a proof assistant.

Theorem 1 *Let c be a CL contract, γ a variable assignment, ρ a CLVM environment and l a list of CLVM instructions.*
If $Com(c) = Some\ l$, then for any CL-trace T :

(i) $C[\![c]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ T \Rightarrow \exists T_M.Tt(T_M) = T \wedge CI[\![l]\!]_{\gamma,\rho.[],[],0} = T_M$

(ii) $C[\![c]\!]_{\gamma,Te(\rho)} = \texttt{None} \Rightarrow CI[\![l]\!]_{\gamma,\rho.[],[],0} = None$

4.2 Relating representations

Before we present the certification of the CLVMI semantics in the next section, we need to establish some relations between the different representations of environments and traces used by CL and CLVM. And some properties of the CLVM representations. First we present a relation between CL and CLVM environments and two properties of the CLVM representations.

Lemma 1 *For any CLVM environment rho and $z \in \mathbb{Z}$. Then*

$$Te(\rho)/z = Te(\rho/z)$$

Lemma 2 *For any CLVM environment rho, the following equality holds*

$$\rho/0 = \rho$$

Lemma 3 *Let ρ be any CLVM environment, and $z_1, z_2 \in \mathbb{Z}$. Then*

$$(\rho/z_1)/z_2 = \rho/(z_2 + z_1)$$

The Iris stdpp-library provides a proof of extensional equality of finite maps, the specific case of extensional equality of CLVM environments is stated in Proposition 1

[1]https://github.com/malthelange/CLVM . The repository is a fork of the ConCert framework, the README provides a guide to finding the CLVM formalization and proofs.

Proposition 1 *For any two CLVM environments ρ_1, ρ_2, if for any $d \in \mathbb{N}$.*

$$\rho_1(d) = \rho_2(d)$$

then

$$\rho_1 = \rho_2$$

With this equality the three lemmas all follows trivially from the following lemma

Lemma 4 *Let ρ be any CLVM environment and $d, i \in \mathbb{Z}, l \in \textbf{Label}$. Then*

$$\rho(l, d+i) = \rho(l, i)/d$$

Before proving Lemma 4 we need to establish some facts about the time-shifting operator / for CLVM and the formalization of finite maps. The stdpp library defines two functions

$$\begin{aligned} \mathit{toList} &: \mathtt{M} \to \mathtt{list}\ (K * A) \\ \mathit{fromList} &: \mathtt{list}\ (K * A) \to \mathtt{M} \end{aligned}$$

Where $\mathtt{M}$ is the domain of finite maps from K to A and $(K * A)$ is the Cartesian product of the two domains. The / operator for CLVM is defined as

$$\begin{aligned} &\rho/d = \mathit{fromList}(\mathit{adv}_l(d, \mathit{toList}(\rho))) \\ &\text{where } \mathit{adv}_l(d, xs) = \mathit{map}(\mathit{adv}_e(d), xs) \\ &\text{where } \mathit{adv}_e(d, (l, z, v)) = (l, z - d, v) \\ &\text{where } d, z \in \mathbb{Z}, l \in \textbf{Label}, v \in \textbf{Val}, xs \in \mathtt{list\ (Label\ } *\mathbb{Z}* \mathtt{\ Val)} \end{aligned}$$

Here *map* is the map operation on lists, a common feature of functional languages.

Next we need to prove the lemma

Lemma 5 *For any list l of $\textbf{Label} * \mathbb{Z} * \textbf{Val}$, $k_1 \in \textbf{Label}, k_2, d \in \mathbb{Z}, v \in \textbf{Val}$*

$$\mathtt{In}((k_1, k_2, v), \mathit{adv}_l(d, l)) \Leftrightarrow \mathtt{In}((k_1, k_2 + d, v), l)$$

$\mathtt{In}(a, l)$ is a standard Coq predicate which is true if the value a is present in the list l, it is defined as

$$\mathtt{In}(a, l) = \begin{cases} \bot & \text{if} l = [] \\ b = a \vee \mathtt{In}(a, l) & \text{if } l = b :: m \end{cases} \tag{4.1}$$

Proof of Lemma 5

We prove Lemma 5 by induction in l. The base case of $l = []$ follows trivially, so we proceed with the induction case where $l = b :: m$. We first prove the relation left to right

$$\mathtt{In}((k_1, k_2, v), \mathit{adv}_l(d, l)) \Rightarrow \mathtt{In}((k_1, k_2 + d, v), l)$$

From the definition of In 4.1 we know that

$$(k_1,k_2,v) = adv_e(d,b) \vee \texttt{In}((k_1,k_2,v), adv_l(m,d))$$

In the case of $adv_e(d,b) = (k_1,k_2,v)$ it follows from the definition of adv_e that

$$adv_e(b) = (k_1,z,v)$$

For some $z \in \mathbb{Z} \mid z - d = k_2$, it then follows that

$$b = (k_1, z+d, v) = (k_1, k_2 - d + d, v) = (k_1,k_2,v)$$

Proving the case. In the case of $\texttt{In}((k_1,k_2,v), adv_l(d,m))$, the proof follows directly from the induction hypothesis.

Next we consider the relation from right to left.

$$\texttt{In}((k_1,k_2+d,v),l) \Rightarrow \texttt{In}((k_1,k_2,v), adv_l(d,l))$$

From the definition of In 4.1 we know that

$$(k_1,k_2+d,v) = b \vee \texttt{In}((k_1,k_2+d,v),m)$$

We consider the case of $(k_1,k_2+d,v) = b$, from this definition of b and the definition of $adv_e(d,b)$ it follows directly that

$$adv_e(d,b) = (k_1,k_2+d-d,v) = (k_1,k_2,v)$$

Proving the case. In the case of $\texttt{In}((k_1,k_2+d,v),m)$, the proof follows directly from the induction hypothesis. Q.E.D

Next we need to prove the following lemma

Lemma 6 *For any $d \in \mathbb{Z}$ and list l of **Label** $* \mathbb{Z} *$ **Val**, such that NoDup(map(fst,l)) Then*

$$\texttt{Permutation}(toList(fromList(adv_l(d,l))), adv_l(d,l))$$

Here $\texttt{NoDup}(l)$ is standard Coq predicate which states that the list l has no duplicates, $\texttt{Permutation}(l_1,l_2)$ is a standard Coq predicate stating that l_1 is a permutation of l_2 and fst is the standard Coq function which returns the first element of a tuple.

In order to prove the lemma we need to utilize the following propositions provided by the stdpp-library and the Coq standard library.

Proposition 2 *Let l be any list of $K * V$, for some domains K and V, then*

$$\texttt{NoDup}(map(fst,l)) \rightarrow \texttt{Permutation}(toList(fromList(l)),l)$$

Proposition 3 *Let l be any list of A and f be any injective function $f : A \rightarrow B$, for some domains A and B. Then*

$$\texttt{NoDup}(l) \rightarrow \texttt{NoDup}(map(f,l))$$

We can now prove Lemma 6.

Proof of Lemma 6

.

The proof follows directly from Proposition 2, if we can prove the premise

$$\texttt{NoDup}(map(fst,(adv_l(d,l))))$$

From the definition of *map* and adv_l it is easy to show that

$$\begin{aligned}
&\texttt{NoDup}(map(fst,(adv_l(d,l)))) \Leftrightarrow \\
&\texttt{NoDup}(map(fst \circ adv_e(d),l)) \Leftrightarrow \\
&\texttt{NoDup}(map((\lambda(l,i).(l,i-d)) \circ fst,l)) \Leftrightarrow \\
&\texttt{NoDup}(map((\lambda(l,i).(l,i-d)),map(fst,l)))
\end{aligned}$$

We omit the details here, referring to the Coq formalization. From Proposition 3 we know that

$$map((\lambda(l,i).(l,i-d)),map(fst,l))$$

satisfies the NoDup predicate if fst and $(\lambda(l,i).(l,i-d))$ are injective. It should be clear that this is true, and so we omit the details referring to the Coq formalization. This concludes the proof

---

The proof of Lemma 4 requires the 4 following propositions provided by the stdpp-library and the Coq standard library

Proposition 4 *Let K,V be any two domains, m a finite map from K to V, and $k \in K, v \in V$. Then*

$$\texttt{In}\ \ ((k,v),toList(m)) \Leftrightarrow m(k) = \texttt{Some}\ \ v$$

Proposition 5 *Let K,V be any two domains, m a finite map from K to V, and $k \in K, v \in V$. Then*

$$\neg\texttt{In}((k,v),toList(m)) \Leftrightarrow m(k) = \texttt{None}$$

Here $\neg$ is the negation of a proposition.

Proposition 6 *For any finite map m.*

$$\texttt{NoDup}(map(fst,toList(m)))$$

Proposition 7 *Let l,l' be any two lists of domain A which satisfies the* `Permutation` *predicate. Then*

$$\texttt{In}(x,l) \Leftrightarrow \texttt{In}(x,l')$$

Proof of Lemma 4

Finally we can prove Lemma 4. We first consider the case of $\rho(l, d+i) = \texttt{Some}\ \ v$ By Proposition 4 we know that

$$\texttt{In}((l, d+i, v), toList(\rho))$$

And we need to show that

$$\texttt{In}((l, i, v), toList(\rho/d))$$

From the definition of / it follows that

$$\texttt{In}((l, i, v), toList(\rho/d)) \Leftrightarrow \texttt{In}((l, i, v), toList(fromList(adv_l(d, toList(\rho))))) \tag{4.2}$$

From Lemma 6 and Proposition 6 it follows that

$$\texttt{Permutation}(toList(fromList(adv_l(d, toList(\rho)))), adv_l(d, toList(\rho)))$$

From Proposition 7 and equation 4.2 it follows that

$$\texttt{In}((l, i, v), toList(fromList(adv_l(d, toList(\rho))))) \rightarrow \texttt{In}((l, i, v), adv_l(d, toList(\rho)))$$

The proof then follows from 5

Next we prove the case of $\rho(l, d+i) = \texttt{None}$ by contradiction. We assume that $(\rho/d)(l, d) = \texttt{Some}\ \ v$ By Proposition 5 we know that

$$\begin{gathered} \rho(l, d+i) = \texttt{None} \Leftrightarrow \\ \forall v'. \neg \texttt{In}((l, d+i, v'), toList(\rho)) \end{gathered} \tag{4.3}$$

From the reasoning as in the previous case we know that

$$\texttt{In}((l, i, v), toList(\rho/d))$$

And from Lemma 5 it follows that

$$\texttt{In}((l, d+i, v), toList(\rho))$$

Which contradicts our assumption by equation 4.3. This concludes the proof of Lemma 4

Relating traces

We also need the following 4 lemmas which relate the CL and CLVM representations of traces

Lemma 7 *Let $p, q \in$ **Party**$, a \in$ **Asset** then*

$$Tt(T_{M,a(p\rightarrow q)}) = T_{a(p\rightarrow q)}$$

Lemma 8 *Let T be any CL trace and T_M any CLVM trace such that $Tt(T_M) = T$ and $n \in \mathbb{N}$. Then:*

$$Tt(delaym(n,T_M)) = delay(n,T)$$

Lemma 9 *Let T_M be any CLVM trace and $z \in \mathbb{Z}$. Then:*

$$Tt(scalem(z,T_M)) = scale(z,Tt(T_M))$$

Here $scale(z,T') = \lambda n.\lambda(p,q,a).z \cdot T'(n)(p,q,a)$, and $scalem(z,T_M)$, is the trace T_M with all transfer amounts multiplied by z

Lemma 10 *Let T_{M1}, T_{M2} be any two CLVM traces. Then:*

$$Tt(addm(T_{M1},T_{M2}) = add(Tt(T_{M1},Tt(T_{M2}))$$

Here $add(T_1,T_2) = \lambda n.\lambda(p,q,a).T_1(n)(p,q,a) + T_2(n)(p,q,a)$ and $Tt(addm(T_{M1},T_{M2})$ is the union of the two finite maps T_{M1}, T_{M2}, if a transfer of asset a between parties p,q is stipulated at time t in the both of the two finite maps, the transfer $T_{M1}(t)(a,p,q) + T_{M2}(t)(a,p,q)$ is stipulated at time t.

The proofs of these lemmas are long, but they are either simple or follow mostly the same structure as the proof of 4. Since we do not want to waste too much space repeating the same ideas we refer to the Coq formalization of the proofs.

4.3 Certifying compilation scheme for CL expressions

In this section we present the proof of Theorem 1. As mentioned the Coq verified proof can be found at the project GitHub page. In this section we give an informal presentation of the novel aspects of the proof, omitting the technical details required for Coq verification.

The first step of the proof is proving that compilation and interpretation of CL expressions follow the denotational semantics of CL, this property is stated in Theorem 2. Instead of proving Theorem 2 directly, we first prove the more general lemmas 11 and 12.

Theorem 2 *Let e be any CL expression, γ any variable assignment, ρ any CLVM environment and l any list of CLVM expression instructions, such that $ECom(e) = l$, . Then*

$$\varepsilon[\![e]\!]_{\gamma,Te(\rho),[]} = EI[\![l]\!]_{\gamma,\rho,[]}.$$

Lemma 11 *Let e be any CL expression, γ any variable assignment, ρ any CLVM environment, stack any list of `option Val` and l_0, l_1 any list of CLVM expression instructions, such that $ECom(e) = l_0$ then*

$$\varepsilon[\![e]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ v \rightarrow$$
$$EI[\![l_0 +\!\!+ l_1]\!]_{\gamma,\rho,stack} = EI[\![l_1]\!]_{\gamma,\rho,v::stack}$$

Lemma 12 *Let e be any CL expression, γ any variable assignment, ρ any CLVM environment, stack any list of* `option Val` *and l_0, l_1 any list of CLVM expression instructions, such that $ECom(e) = l_0$ then*

$$\varepsilon[\![e]\!]_{\gamma,Te(\rho)} = \texttt{None} \rightarrow$$
$$EI[\![l_0 +\!\!+ l_1]\!]_{\gamma,\rho,stack} = \texttt{None}$$

Intuitively both lemmas state, that if the first sub-list l_0 of the instruction list is equal to a compiled expression e. Then if the denotational semantics specify some value, v for e,the execution of the l_0 instruction sublist pushes the value of v on top of the stack. If the denotational semantics does not specify any value then execution fails.

Both lemmas are proven by induction in e, which gives a stronger induction hypothesis than we would have gotten directly from Theorem 2. To verify the behaviour of any expression we need to know that any sub-expressions are first evaluated and pushed on stack, this proposition is provided by the strong induction hypothesis. Our Coq specification of the two lemmas 11 and 12 are presented in figure 4.1. Here the list `expis` and the corresponding property is redundant and not used in the proof, but we did not have time to refactor the proof and remove it. Most of the code should be self explanatory for any reader familiar with basic Coq syntax. `CompileE` is the compilation procedure for CL expressions to CLVM expression instructions. `StackEInterp` is the interpretation procedure for expressions, the boolean flag given as the last argument is also an artefact we did not have the time to remove. `ExtMap` is the type of finite maps used for representing CLVM environments. `ExtMap_to_ExtEnv` is the function Tt which translates CLVM environments to the total functions used to represent environments in CL. `Esem` is the formalization of the denotational semantics for CL expressions, for non-well-formed expressions it returns `None`.

```
Lemma TranslateExpressionStep : forall (e : Exp) (env : Env)
                                (expis l0 l1 : list instruction)
                                (ext : ExtMap)
                                (v : Val),
expis = l0 ++ l1 ->
CompileE e = Some l0 ->
Esem e env (ExtMap_to_ExtEnv ext) = Some v ->
StackEInterp (l0 ++ l1) stack env ext false =
StackEInterp l1 ((Some v)::stack) env ext false.

Lemma TranslateExpressionNone : forall (e : Exp) (env : Env)
                                (l0 l1 : list instruction)
                                (ext : ExtMap)  (stack : list (option Val)),
CompileE e = Some l0 ->
Esem e env (ExtMap_to_ExtEnv ext) = None ->
StackEInterp (l0 ++ l1) stack env ext false = None.
```

Figure 4.1: Coq equivalent for lemmas 11 and 12

First we prove Lemma 11. As mentioned we do so by induction in e. By the standard Coq induction scheme, the induction hypothesis states the proposition for any sub-expressions in e. However we also need to know that the proposition holds for any operands in $op(e_1, ..., , e_n)$ expressions. Luckily the CL formalization provides a strengthened induction scheme, `Exp_ind'`, which provides this hypothesis.

Case: $e = v$ **and** $e = \textbf{Obs}(l, i)$

In the base-step of the induction we consider the expressions which do not contain sub expressions, $\textbf{Obs}(l, i)$ and v, where $v \in \textbf{Var}$. In both cases $ECom(e)$, and therefore l_0 is uniquely defined and the case is proven directly from the evaluation of $EI[\![l_0]\!]_{\gamma,\rho}$.

The induction step is more interesting. Here we consider the expressions, $op(e_0, .., e_n)$ and $\textbf{Acc}(\lambda x.e_1, d, e_2)$ where $op \in \textbf{Op}$. First we consider the case of operators.

Case $e = op(e_1, ..., e_r)$

Here we consider each operator individually. There are 16 cases and the proof for each is almost identical. For each case of op of arity r we know that the sub-expressions e_i where $0 \leq i \leq r$ is compiled to some instruction list l'_i, otherwise it contradicts our assumption that $ECom(e) = l_0$ since the compilation would fail. l_0 is then uniquely defined as $l'_r +\!\!+ ... +\!\!+ l'_0 +\!\!+ [\textbf{Op}\ op]$. For each e_i we can deduce that $\varepsilon[\![e_i]\!]_{\gamma,Te(\rho)} =$ `Some` v_i, since otherwise $\varepsilon[\![e]\!]_{\gamma,Te(\rho)} =$ `None` contradicting our assumption. Since we established these two propositions for each e_i, we can apply the strengthened induction hypothesis for each e_i, giving us the equivalence relation

$$EI[\![l'_r +\!\!+ ... +\!\!+ l'_0 +\!\!+ [\textbf{Op}\ op] +\!\!+ l_1]\!]_{\gamma,\rho,stack} = EI[\![[\textbf{Op}\ op] +\!\!+ l_1]\!]_{\gamma,\rho,v_1::...::v_r::stack}$$

From the defined execution of $[\textbf{Op}\ op]$ by EI it follows for every op that

$$EI[\![[\textbf{Op}\ op] +\!\!+ l_1]\!]_{\gamma,\rho,v_1::...::v_r::stack} = EI[\![l_1]\!]_{\gamma,\rho,op(v_1,...,v_r)::stack}$$

Which proves the case.

Case: $e = \textbf{Acc}(\lambda x.e_1, d, e_2)$

For the case of $\textbf{Acc}(\lambda x.e_1, d, e_2)$ the compiled instructions have a different structure depending on the value d. So we consider the case of $d = 0$, $d = 1$ and $d > 1$ separately. In all cases we deduce that $ECom(e_1) = l'_1$ and $ECom(e_2) = l'_2$ since otherwise it would contradict our assumption that $ECom(e) = l_0$.

Case: $e = \textbf{Acc}(\lambda x.e_1, d, e_2)$ **and** $d = 0$

First we consider the case where $d = 0$. Here l_0 is uniquely defined as

$$[\texttt{AccStart}_\texttt{I} 0] +\!\!+ l'_2$$

The execution of $[\texttt{AccStart}_\texttt{I} n]$ time-shifts the environment by $-n$. In this case it effectively does nothing. Giving us the relation

$$EI[\![[\texttt{AccStart}_\texttt{I} 0] +\!\!+ l'_2 +\!\!+ l_1]\!]_{\gamma,\rho,stack} = EI[\![l'_2 +\!\!+ l_1]\!]_{\gamma,\rho,stack}$$

As in the case of op we know that $\varepsilon[\![e_2]\!]_{\gamma,Te(\rho)} =$ `Some` v_2 since otherwise it would contradict our assumption. It then follows from the induction hypothesis that

$$EI[\![l'_2 +\!\!+ l_1]\!]_{\gamma,\rho,stack} = EI[\![l_1]\!]_{\gamma,\rho,v_2::stack}$$

Proving the case.

Auxiliary lemmas

Now we need to introduce the auxiliary lemma 13, which states a property of `Acc_sem`. The function `Acc_sem` specifies the iterative steps of CL denotational semantics of $\mathbf{Acc}(\lambda x.e_1, d, e_2)$. We will omit the technical details here, for reference the Coq formalization of CL expressions semantics is present in figure 4.2.

Lemma 13 *Let e_1 and e_2 be any CL expression, ρ any CL environment and γ any variable assignment and $d_1, d_2 \in \mathbb{N}, v \in \textbf{Val}$. If*

$$\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,\rho}), (d_1 + d_2), \varepsilon[\![e_2]\!]_{\gamma,\rho}) = \texttt{Some}\ \ v$$

Then there exists a value v_2, such that

$$\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,\rho}), d_1, \varepsilon[\![e_2]\!]_{\gamma,\rho}) = \texttt{Some}\ \ v_2$$

This lemma is easily proven by induction in d_2 and the semantics of `Acc_sem` and `Fsem`, the proof is omitted here for readability, as any other lemma referenced in the section the Coq verification can be found in the repository.

Next we need to prove an additional lemma

Lemma 14 *Let e_1 and e_2 be any CL expressions, ρ any CLVM environment, γ any variable assignment, stack any list of `option`Val, l_1, l_2 any list of CLVM expression instructions, $v_1, v_s \in \textbf{Val}$, $n_{left}, n_{last} \in \mathbb{N}$ If the proposition*

$$\begin{aligned}&\forall \gamma', l', \rho', stack', v'.\\ &\varepsilon[\![e_1]\!]_{\gamma',Te(\rho')} = \texttt{Some}\ \ v' \rightarrow\\ &EI[\![l_2 +\!\!+ l']\!]_{\gamma',\rho',stack'} = EI[\![l']\!]_{\gamma',\rho',v'::stack'}\end{aligned}$$

holds. Then if

$$\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,Te(\rho)}), n_{last}, \varepsilon[\![e_2]\!]_{\gamma,Te(\rho)}) = \texttt{Some}\ \ v_1$$
$$\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,Te(\rho)}), n_{last} + n_{left}, \varepsilon[\![e_2]\!]_{\gamma,Te(\rho)}) = \texttt{Some}\ \ v_s$$

it follows that

$$\begin{aligned}&EI[\![repeat(l_2 +\!\!+ [IAccStep], n_{left}) +\!\!+ l_2 +\!\!+ [IAccEnd] +\!\!+ l_1]\!]_{v_1::\gamma,\rho/(1+n_{last}),stack} =\\ &EI[\![l_2 +\!\!+ [IAccEnd] +\!\!+ l_1]\!]_{v_s::\gamma,\rho/(1+n_{last}+n_{left}),stack}\end{aligned}$$

This lemma might seem confusing and oddly specific. The first assumption is a specialization of Lemma 11, which can be provided by the induction hypothesis in Lemma 11. We need to state this proposition as a separate lemma to get the desired induction hypothesis in Coq. Intuitively this lemma states, that if evaluation of l_2 pushes $\varepsilon[\![e_1]\!]_{\gamma,Te(\rho)}$ on the stack, for any γ and ρ, then the evaluation of repeated $l_2 +\!\!+ [IAccStep]$ instruction lists computes the iteration described by **Acc_sem**.

We prove this lemma by induction in n_{left}

Proof of Lemma 14 Case: $n_{left} = 0$

In the base case of $n_{left} = 0$ we can rewrite the desired equality as

$$EI[\![l_2 +\!\!+ [IAccEnd] +\!\!+ l_1]\!]_{v_1::\gamma,\rho/(1+n_{last}),stack} =$$
$$EI[\![l_2 +\!\!+ [IAccEnd] +\!\!+ l_1]\!]_{v_s::\gamma,\rho/(1+n_{last}),stack}$$

And from the assumptions it follows that $v_1 = v_s$. Proving the equality.

Proof of Lemma 14 Case: $n_{left} > 0$

In the induction case $n_{left} > 0$ we know from Lemma 13 and the assumption

$$\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,Te(\rho)}), n_{last} + n_{left}, \varepsilon[\![e_2]\!]_{\gamma,Te(\rho)}) = \texttt{Some}\ \ v_s$$

That the following equality holds

$$\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,Te(\rho)}), n_{last} + 1, \varepsilon[\![e_2]\!]_{\gamma,Te(\rho)}) = \texttt{Some}\ \ v'$$

For some value v'. By the definition of `Acc_sem` this equality can be rewritten as

$$\varepsilon[\![e]\!]_{v_1::\gamma,Te(\rho)/(n_{last}+1)} = \texttt{Some}\ \ v'$$

From this equality we can rewrite the hypothesis using the first assumption, the definition of the expression interpreter and Lemma 3. We are left with proving the following equality

$$EI[\![repeat(l_2 +\!\!+ [IAccStep], (n_{left} - 1)) +\!\!+ l_2 +\!\!+ [IAccEnd] +\!\!+ l_1]\!]_{v'::\gamma,\rho/(2+n_{last}),stack} =$$
$$EI[\![l_2 +\!\!+ [IAccEnd] +\!\!+ l_1]\!]_{v_s::\gamma,\rho/(1+(n_{last}+1)+(n_{left}-1)),stack}$$

This equality follows directly from the induction hypothesis, proving the lemma.

Proof of Lemma 11 case $e = \mathbf{Acc}(\lambda x.e_1, d, e_2)$ and $d > 0$

Resuming the proof of Lemma 11. The proof for cases $d = 1$ and $d > 1$ have some common steps. In both cases the beginning of the instruction list l_0 is uniquely defined as

$$[\texttt{AccStart}_1\ \ d] +\!\!+ l_2'$$

The first instruction time-shifts the environment by $-d$. In order to apply the induction hypothesis, we need to prove that there exists some v_2 such that

$$\varepsilon[\![e_2]\!]_{\gamma,Te(\rho/(-d))} = \texttt{Some}\ \ v_2$$

From the assumption $\varepsilon[\![\mathbf{Acc}(\lambda x.e_1, d, e_2)]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ v$, Lemma 1 and the semantics of `Acc`, we can deduce the following equality

$$\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,Te(\rho/(-d))}), (0+d), \varepsilon[\![e_2]\!]_{\gamma,Te(\rho/(-d))}) = \texttt{Some}\ \ v$$

From Lemma 13 we get the equality

$$\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,Te(\rho/(-d))}),0,\varepsilon[\![e_2]\!]_{\gamma,Te(\rho/(-d))}) = \texttt{Some}\ \ v_2'$$

For some value v_2'.

From the definition of Acc_sem this equality can be reduced to

$$\varepsilon[\![e_2]\!]_{\gamma,Te(\rho/(-d))} = \texttt{Some}\ \ v_2'$$

Setting $v_2 = v_2'$ we get the proposition required for the induction hypothesis. In both cases we apply the induction hypothesis for e_2 and are left with proving the following relation

$EI[\![\texttt{AccStart}_2 +\!\!+ \mathit{repeat}(l_1' +\!\!+ [\texttt{AccStep}],(d-1)) +\!\!+ l_1' +\!\!+ [\texttt{AccEnd}] +\!\!+ l_1]\!]_{\gamma,\rho/(-d),v_2::stack} =$
$EI[\![l_1]\!]_{\gamma,\rho,v::stack}$

From the definition of CLVMI execution of $\texttt{AccStart}_2$ this can be reduced to

$$\begin{aligned}&EI[\![\mathit{repeat}(l_1' +\!\!+ [\texttt{AccStep}],(d-1)) +\!\!+ l_1' +\!\!+ \\ &\quad [\texttt{AccEnd}] +\!\!+ l_1]\!]_{v_2::\gamma,(\rho/(-d))/1,stack} = \\ &EI[\![l_1]\!]_{\gamma,\rho,v::stack}\end{aligned}$$

Here we split the cases.

Case $e = \textbf{Acc}(\lambda x.e_1,d,e_2)$ and $d = 1$

For $d = 1$, the equality we need to show can be rewritten by Lemma 2 and 3 as

$$\begin{aligned}&EI[\![l_1' +\!\!+ [\texttt{AccEnd}] +\!\!+ l_1]\!]_{v_2::\gamma,\rho,stack} = \\ &EI[\![l_1]\!]_{\gamma,\rho,v::stack}\end{aligned}$$

The assumption

$$\varepsilon[\![\textbf{Acc}(\lambda x.e_1,1,e_2)]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ v$$

Can be rewritten by the denotational semantics as

$$\varepsilon[\![e_1]\!]_{v_2::\gamma,(Te(\rho)/-1)/1} = \texttt{Some}\ \ v$$

By Lemma 1, 2 and 3 this can be rewritten as

$$\varepsilon[\![e_1]\!]_{v_2::\gamma,Te(\rho)} = \texttt{Some}\ \ v$$

With this equality we can apply the induction hypothesis giving the equality.

$$\begin{aligned}&EI[\![l_1' +\!\!+ [\texttt{AccEnd}] +\!\!+ l_1]\!]_{v_2::\gamma,\rho,stack} = \\ &EI[\![[\texttt{AccEnd}] +\!\!+ l_1]\!]_{v_2::\gamma,\rho,v::stack}\end{aligned}$$

Finally from the definition of CLVMI execution of AccEnd we get the equality

$$\begin{aligned}&EI[\![[\texttt{AccEnd}] +\!\!+ l_1]\!]_{v_2::\gamma,\rho,v::stack} \\ &EI[\![l_1]\!]_{\gamma,\rho,v::stack}\end{aligned}$$

Proving the case.

Case $e = \mathbf{Acc}(\lambda x.e_1, d, e_2)$ and $d > 1$

For $d > 1$ the equality we need to show is

$$\begin{aligned} &EI[\![repeat(l_1' +\!\!+ [\texttt{AccStep}], (d-1)) +\!\!+ l_1' +\!\!+ \\ &\quad [\texttt{AccEnd}] +\!\!+ l_1]\!]_{v_2::\gamma,(\rho/(-(d-1)),stack} = \\ &EI[\![l_1]\!]_{\gamma,\rho,v::stack} \end{aligned} \tag{4.4}$$

First we show that

$$\begin{aligned} &\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,Te(\rho/(-d))}), 1, \varepsilon[\![e_2]\!]_{\gamma,Te(\rho/(-d))}) = \\ &\varepsilon[\![e_1]\!]_{v_2::\gamma,Te(\rho/(-(d-1)))} = \\ &\texttt{Some}\ \ v_1 \end{aligned} \tag{4.5}$$

This follows from the definition of `Acc_sem`, lemmas 1 and 3, and the assumption $\varepsilon[\![\mathbf{Acc}(\lambda x.e_1, d, e_2)]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ v$. Since we know the value of $\varepsilon[\![e_1]\!]_{v_2::\gamma,Te(\rho/(-(d-1)))}$ and we know $d > 1$ we get the following equality from the induction hypothesis.

$$\begin{aligned} &EI[\![repeat(l_1' +\!\!+ [\texttt{AccStep}], (d-1)) +\!\!+ l_1' +\!\!+ \\ &\quad [\texttt{AccEnd}] +\!\!+ l_1]\!]_{v_2::\gamma,(\rho/(-(d-1)),stack} = \\ &EI[\![l_1' +\!\!+ [\texttt{AccStep}] +\!\!+ repeat(l_1' +\!\!+ [\texttt{AccStep}], (d-2)) +\!\!+ l_1' +\!\!+ \\ &\quad [\texttt{AccEnd}] +\!\!+ l_1]\!]_{v_2::\gamma,(\rho/(-(d-1)),stack} = \\ &EI[\![repeat(l_1' +\!\!+ [\texttt{AccStep}], (d-2)) +\!\!+ l_1' +\!\!+ \\ &\quad [\texttt{AccEnd}] +\!\!+ l_1]\!]_{v_1::\gamma,(\rho/(-(d-2)),stack} \end{aligned}$$

We then show that

$$\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,Te(\rho/(-d))}), d, \varepsilon[\![e_2]\!]_{\gamma,Te(\rho/(-d))}) = v \tag{4.6}$$

This follows directly from the definition of the denotational semantics. From Lemma 13 we get the equality

$$\texttt{Acc\_sem}(\texttt{Fsem}(\varepsilon[\![e_1]\!]_{\gamma,Te(\rho/(-d))}), (d-1), \varepsilon[\![e_2]\!]_{\gamma,Te(\rho/(-d))}) = v' \tag{4.7}$$

For some value v'. By Lemma 14 given the equalities presented in equation (4.5) and (4.7) and the induction hypothesis we get the equality

$$\begin{aligned} &EI[\![repeat(l_1' +\!\!+ [\texttt{AccStep}], (d-2)) +\!\!+ l_1' +\!\!+ \\ &\quad [\texttt{AccEnd}] +\!\!+ l_1]\!]_{v_1::\gamma,(\rho/(-(d-2)),stack} = \\ &EI[\![l_1' +\!\!+ [\texttt{AccEnd}] +\!\!+ l_1]\!]_{v'::\gamma,\rho,stack} = \end{aligned}$$

From the definition of $\mathbf{Acc}(\lambda x.e_1, d, e_2)$ and the equation (4.6) we get the equality

$$\varepsilon[\![e_1]\!]_{v'::\gamma,Te(\rho)} = v$$

By the induction hypothesis and the definition of the expression interpreter, we get the equality:

$$\begin{aligned} &EI[\![l_1' +\!\!+ [\texttt{AccEnd}] +\!\!+ l_1]\!]_{v'::\gamma,\rho,stack} = \\ &EI[\![l_1]\!]_{\gamma,\rho,v::stack} \end{aligned}$$

Proving the case. Q.E.D.

The proof for Lemma 12 is structured very similarly to the proof of Lemma 11 so we it omit here, referring instead to the Coq formalization in the repository.

Now we present a proof for Theorem 2. We first consider the case $\varepsilon[\![e]\!]_{\gamma,Te(\rho)} =$ `Some` v By Lemma 11 and the definition of $EI[\![[]]\!]_{\gamma,\rho,stack}$ we get the equality

$$\begin{aligned} EI[\![l]\!]_{\gamma,\rho,[]} &= \\ EI[\![l +\!\!+ []]\!]_{\gamma,\rho,[]} &= \\ EI[\![[]]\!]_{\gamma,\rho,[v]} &= v \end{aligned}$$

Which proves the case. For the case $\varepsilon[\![e]\!]_{\gamma,Te(\rho)} =$ `None` We get the following equality from the assumption and Lemma 12:

$$\begin{aligned} EI[\![l]\!]_{\gamma,\rho,[]} &= \\ EI[\![l +\!\!+ []]\!]_{\gamma,\rho,[]} &= \\ &\texttt{None} \end{aligned}$$

Proving the case. Q.E.D.

```
Fixpoint Acc_sem {A} (f : nat -> A -> A) (n : nat) (z : A) : A :=
  match n with
  | O => z
  | S n' => f n (Acc_sem f n' z)
  end.

Definition Fsem {A} (f : Env -> ExtEnv -> option A) (env : Env) (ext : ExtEnv)
  := (fun m x => x >>= fun x' =>  f (x' :: env) (adv_ext (Z.of_nat m) ext)).

(** Semantics of expressions in CL *)
Fixpoint Esem (e : Exp) (env : Env) (ext : ExtEnv) : option Val :=
  match e with
  | OpE op args => sequence (map (fun e => E[|e|] env ext) args) >>= OpSem op
  | Obs l i => Some (ext l i)
  | VarE v => lookupEnv v env
  | Acc f l z => let ext' := adv_ext (- Z.of_nat l) ext
              in Acc_sem (Fsem E[|f|] env ext') l (E[|z|] env ext')
  end
where "'E[|' e '|]'" := (Esem e ).
```

Figure 4.2: Coq formalization of the CL semantics

4.4 Certifying compilation scheme for CL contracts

In this section we present the proof of Theorem 1. Like Theorem 2 we first prove a more general lemma which provides a stronger induction hypothesis.

Lemma 15 *Let c be any CL contract and l_1 and l_2 any lists of CLVM instructions such that $Com(c) = l_1$. Let ρ be any CLVM environment, P any list of CLVM environments, γ any variable assignment, stack any list of* `option TraceM`*, and ws any list of natural numbers. Then for all CL traces T such that $C[\![c]\!]_{\gamma,Te(\rho)} = T$ there exists a CLVM trace T_M such that $Tt(T_M, 0) = T$ and*

$$CI[\![l_1 +\!\!+ l_2]\!]_{\gamma,\rho::P,stack,ws,0} = CI[\![l_2]\!]_{\gamma,\rho::P,T_M::stack,ws,0}$$

And if $C[\![c]\!]_{\gamma,Te(\rho)} =$ `None`, the following equality holds:

$$CI[\![l_1 +\!\!+ l_2]\!]_{\gamma,\rho::P,stack,ws,0} = \texttt{None}$$

The Coq formalization of the lemma can be found in fig 4.3. Here `Contr` is the type for CL contracts, `CInstruction` is the type of CLVM instructions, `Trace` is the type of CL traces and `TraceM` is the type of CLVM Traces. `ExtMap_to_ExtEnv extM` is the function Te. The function `traceMtoTrace` is the function Tt. `CompileC` is the function specifying the compilation procedure from CL to CLVM `StackCInterp` is the CLVMI interpreter for CLVM. `Csem` is the function which specifies the formalization of the CL denotational semantics, it is a partial function which returns `None` for non-well-formed contracts. With these definitions the code should be self-explanatory and readable for readers with a basic understanding of Coq.

```
Lemma TranslateContractStep : forall (c : Contr) (env : Env) (extM : ExtMap)
                                     (extMs : list ExtMap) (l1 l2 : list CInstruction)
                                     (stack : list (option TraceM)) (w_stack : list nat),
        CompileC c = Some l1 ->
        (forall (t: Trace), Csem c env (ExtMap_to_ExtEnv extM) = Some t ->
         exists tm,
           traceMtoTrace tm 0 = t /\
           StackCInterp (l1 ++ l2) stack env (extM::extMs) w_stack 0 =
            StackCInterp l2 ((Some tm)::stack) env (extM::extMs) w_stack 0)
        /\ (Csem c env (ExtMap_to_ExtEnv extM) = None ->
           StackCInterp (l1 ++ l2) stack env (extM::extMs) w_stack 0 = None).
```

Figure 4.3: Coq formalization of Lemma 15

Auxiliary lemmas for `If-within` statements

Before we proceed to the proof of Lemma 15 we need establish the following 5 auxiliary lemmas. 4 of the lemmas concern the function

$$wsem : \textbf{list e\_inst} \times \mathbb{N} \times \textbf{Env} \times \textbf{EnvM} \rightarrow \text{option}(\mathbb{B} * \mathbb{N})$$

Here **e_inst** is the domain CLVM expression instructions. Intuitively, for some $l \in$ **list e_inst**, $n \in \mathbb{N}$, $\gamma \in$ **Env** $\rho \in$ **EnvM**, $wsem(l, n, \gamma, \rho)$ goes through environments $\rho/0$ to ρ/n. if $EI[\![l]\!]_{\gamma,\rho/i,[]} =$ `BVal true` for any $0 \leq i \leq n$, then for the lowest i satisfying the condition, $\texttt{Some}(\texttt{True}, n-i)$ is returned. If no i satisfies the condition, then $\texttt{Some}(\texttt{False}, 0))$ is returned. If the expression is of type $\mathbb{Z}$ then `None` is returned.

Lemma 16 *Let ρ be any CLVM environment, γ any variable assignment, $n,i \in \mathbb{N}$ and l any list of CLVM expression instructions such that*

$$wsem(l,n,\gamma,\rho) = \texttt{Some (True, i)}$$

Then $i \leq n$

Lemma 17 *Let c_1 and c_2 be any CL contracts, and e any CL expression such that $ECom(e) = l_e$ for some list of CLVM expression instructions l_e, let ρ be an CLVM environment, γ any variable assignment and $n,i \in \mathbb{N}$. If*

$$wsem(l_e,n,\gamma,\rho) = \texttt{Some}\ (\texttt{True},i)$$

Then

$$C[\![\texttt{If}\ e\ \texttt{within}\ d\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]_{\gamma,Te(\rho)} = \begin{cases} delay((n-i),T') & \textit{If}\ C[\![c_1]\!]_{\gamma,Te(\rho)/(n-i)} = T' \\ None & \textit{Otherwise} \end{cases}$$

Lemma 18 *Let c_1 and c_2 be any CL contracts, and e any CL expression such that $ECom(e) = l_e$ for some list of CLVM expression instructions l_e, let ρ be an CLVM environment γ any variable assignment and $n,i \in \mathbb{N}$. If*

$$wsem(l_e,n,\gamma,\rho) = \texttt{Some}\ (\texttt{False},i)$$

Then

$$C[\![\texttt{If}\ e\ \texttt{within}\ d\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]_{\gamma,Te(\rho)} = \begin{cases} delay((n-i),T') & \textit{If}\ C[\![c_2]\!]_{\gamma,Te(\rho)/(n-i)} = T' \\ None & \textit{Otherwise} \end{cases}$$

Lemma 19 *Let c_1 and c_2 be any CL contracts, and e any CL expression such that $ECom(e) = l_e$ for some list of CLVM expression instructions l_e, let ρ be an CLVM environment γ any variable assignment and $n,i \in \mathbb{N}$. If*

$$wsem(l_e,n,\gamma,\rho) = \texttt{None}$$

Then

$$C[\![\texttt{If}\ e\ \texttt{within}\ d\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]_{\gamma,Te(\rho)} = \texttt{None}$$

Lemma 20 *Let c be any CL contract such that $Com(c) = l_1$ for some list of CLVMI instructions l_1. Then for all variable assignments γ, list of CLVM environments P, list of CLVM instructions l_2, stack any list of* `option TraceM`*, ws any list of $\mathbb{N}$ and $bf \in \mathbb{N}$, $bf > 0$, the following equality holds:*

$$\begin{aligned} &CI[\![l_1 +\!\!+ l_2]\!]_{\gamma,P,stack,ws,bf} = \\ &CI[\![l_2]\!]_{\gamma,P,stack,ws,bf} \end{aligned}$$

The proof of lemmas 17, 18 and 19 are very similar and the proof of 20 and 16 are simple and more technical than interesting, so we only present the proof for Lemma 17. For the remaining proofs we refer to the project Github page.

Proof of Lemma 17

We prove Lemma 17 by induction in n.

Proof of Lemma 17 Case: $n = 0$

For the base-case of $n = 0$, $wsem(l_e, 0, \gamma, \rho)$ is defined as

$$wsem(l_e, 0, \gamma, \rho) = \begin{cases} \texttt{Some (True}, 0) & \text{if } EI[\![l_e]\!]_{\gamma,\rho,[]} = \texttt{Some (BVal True)} \\ \texttt{Some (False}, 0) & \text{if } EI[\![l_e]\!]_{\gamma,\rho,[]} = \texttt{Some (BVal False)} \\ \texttt{None} & \text{otherwise} \end{cases}$$

From the assumption $wsem(l_e, 0, \gamma, \rho) = (\texttt{True}, i)$ it follows that

$$EI[\![l_e]\!]_{\gamma,\rho,[]} = \texttt{Some (BVal True)}$$
$$i = 0 \tag{4.8}$$

From Theorem 2 we get the equality

$$\varepsilon[\![e]\!]_{\gamma,\rho} = \texttt{Some (BVal True)} \tag{4.9}$$

From the definition of the denotational semantics we get the equality

$$C[\![\texttt{If } e \texttt{ within } 0 \texttt{ then } c_1 \texttt{ else } c_2]\!]_{\gamma,Te(\rho)} = \begin{cases} C[\![c_1]\!]_{\gamma,Te(\rho)} & \text{if } \varepsilon[\![e]\!]_{\gamma,Te(\rho)} = \texttt{Some (BVal True)} \\ C[\![c_2]\!]_{\gamma,Te(\rho)} & \text{if } \varepsilon[\![e]\!]_{\gamma,Te(\rho)} = \texttt{Some (BVal False)} \\ \texttt{None} & \text{otherwise} \end{cases}$$

By equation 4.9 we can get the equality

$$C[\![\texttt{If } e \texttt{ within } 0 \texttt{ then } c_1 \texttt{ else } c_2]\!]_{\gamma,Te(\rho)} = C[\![c_1]\!]_{\gamma,Te(\rho)}$$

Finally we need to show the equality

$$C[\![c_1]\!]_{\gamma,Te(\rho)} = \begin{cases} delay((0-0), T') & \text{If } C[\![c_1]\!]_{\gamma,Te(\rho)/(0-0)} = T' \\ None & \text{Otherwise} \end{cases}$$

Which follows from the definition of *delay*, and the time-shifting operator $/$. Proving the case.

Proof of Lemma 17 Case: $n > 0$

For the inductive-case of $n > 0$, $wsem(l_e, n, \gamma, \rho)$ is defined as

$$wsem(l_e, n, \gamma, \rho) = \begin{cases} \texttt{Some (True}, n) & \text{if } EI[\![l_e]\!]_{\gamma,\rho,[]} = \texttt{Some (BVal True)} \\ wsem(l_e, (n-1), \gamma, \rho/1) & \text{if } EI[\![l_e]\!]_{\gamma,\rho,[]} = \texttt{Some (BVal False)} \\ \texttt{None} & \text{otherwise} \end{cases} \tag{4.10}$$

From our assumption $wsem(l_e, n, \gamma, \rho) = \texttt{Some (True}, i)$ it follows that

$$EI[\![l_e]\!]_{\gamma,Te,\rho,[]} = \texttt{Some (BVal b)}$$

For any $b \in \mathbb{B}$. From Theorem 2 we get the equality

$$\varepsilon[\![e]\!]_{\gamma,Te(\rho)} = \texttt{Some (BVal b)} \tag{4.11}$$

We first consider the case of $EI[\![l_e]\!]_{\gamma,\rho,[]} = \texttt{Some (BVal True)}$, from 4.10 it follows that

$$i = n \tag{4.12}$$

From equation 4.11, the assumption that $b = \texttt{True}$ and the definition of the denotational semantics of CL we get the equality

$$C[\![\texttt{If } e \texttt{ within } n \texttt{ then } c_1 \texttt{ else } c_2]\!]_{\gamma,Te(\rho)} = C[\![c_1]\!]_{\gamma,Te(\rho)}$$

Finally by the equality 4.12 we need to show the equality.

$$C[\![c_1]\!]_{\gamma,Te(\rho)} = \begin{cases} delay(T, n-n) & \text{if } C[\![c_1]\!]_{\gamma,Te(\rho)/(n-n)} = T \\ \texttt{None} & \text{Otherwise} \end{cases}$$

Which follows from the definition of *delay* and the time-shift operator $/$, proving the case

---

Next we consider the case of $EI[\![l_e]\!]_{\gamma,\rho,[]} = \texttt{Some (BVal False)}$. From the assumption $wsem(l_e, n, \gamma, \rho) = \texttt{True}$ and equation 4.10 we get the equality

$$wsem(l_e, (n-1), \gamma, \rho/1) = \texttt{Some (True,}i\texttt{)} \tag{4.13}$$

From equation 4.11, the assumption that $b = \texttt{False}$ and the definition of the denotational semantics of CL we get the equality

$$C[\![\texttt{If } e \texttt{ within } n \texttt{ then } c_1 \texttt{ else } c_2]\!]_{\gamma,Te(\rho)} =$$
$$\begin{cases} delay(T, 1) & \text{If } C[\![\texttt{If } e \texttt{ within } (n-1) \texttt{ then } c_1 \texttt{ else } c_2]\!]_{\gamma,Te(\rho)/1} = \texttt{Some } T \\ \texttt{None} & \text{otherwise} \end{cases}$$

By Lemma 1, equation 4.13, the definition of the CL time-shift operator $/$, and the induction hypothesis we can rewrite the equality as

$$\begin{aligned} &C[\![\texttt{If } e \texttt{ within } n \texttt{ then } c_1 \texttt{ else } c_2]\!]_{\gamma,Te(\rho)} = \\ &\begin{cases} delay(delay(T,1), ((n-1)-i)) & \text{If } C[\![c_1]\!]_{\gamma,Te(\rho)/(((n-1)-i)+1)} = \texttt{Some } T \\ \texttt{None} & \text{otherwise} \end{cases} \end{aligned} \tag{4.14}$$

Here we would like to rewrite $((n-1)-i)+1$ as $(n-i)$ but since the Coq formalization of natural numbers define $n_1 - n_2 = 0$ for operands $n_1, n_2 \in \mathbb{N} \mid n_2 > n_1$ the order of operations is important. We can only rewrite if $i \leq (n-1)$. We get this equality by Lemma 16 and equation 4.13. By this equality and the definition of *delay* we can rewrite the equation 4.14 as

$$C[\![\texttt{If } e \texttt{ within } n \texttt{ then } c_1 \texttt{ else } c_2]\!]_{\gamma,Te(\rho)} = \begin{cases} delay(T,(n-i)) & \text{If } C[\![c_1]\!]_{\gamma,Te(\rho)/(n-i)} = \texttt{Some } T \\ \texttt{None} & \text{otherwise} \end{cases}$$

Which proves the case an concludes the proof

Proof of Lemma 15

We prove Lemma 15 by induction in C. We only present the proof of the case $C[\![c]\!]_{\gamma,Te(\rho)} = \texttt{Some } T$. The case of $C[\![c]\!]_{\gamma,Te(\rho)} = \texttt{None}$ is very similar, and so we omit it for readability. The full proof can be found in the project Github page.

Case: $c = \emptyset$

For the base case of $c = \emptyset$, from the definition of the denotational semantics of CL we get that $C[\![\emptyset]\!]_{\gamma,Te(\rho)} = T_0$ we let T_M be the empty CLVM trace, T_{M0} which is the finite map containing no mappings. We first need to show the equality

$$Tt(T_{M0}) = T_0$$

This equality follows from functional extensionality. l_1 is uniquely defined as $\emptyset$ by the definition of *Com*. Next we need to show the equality

$$CI[\![[\emptyset] +\!\!+ l_2]\!]_{\gamma,\rho::P,stack,ws,0} = CI[\![l_2]\!]_{\gamma,\rho::P,T_{M0}::stack,ws,0}$$

This follows directly from the definition of $\emptyset$ execution by CLVMI, proving the case.

Case: $c = a(p \to q)$

For the case of $c = a(p \to q)$ we let T_M be the finite map, $T_{M,a(p\to q)}$, which contain only the mapping from 0 to the transfer $Tr_{M,a(p\to q)}$ which is defined as

$$Tr_{M,a(p\to q)}(p',q',a') = \begin{cases} 1 & \text{if } a' = a \wedge p' = p \wedge q' = q \\ -1 & \text{if } a' = a \wedge p' = q \wedge q' = p \\ \texttt{None} & \text{otherwise} \end{cases}$$

From the definition of the denotational semantics of CL we have the equality

$$C[\![a(p \to q)]\!]_{\gamma,Te(\rho)} = unit_{a,p,q}$$

The definition of $unit_{a,p,q}$ can be found in figure 3.2 on page 21. We need to show the equality

$$T_{M,a(p\to q)} = unit_{a,p,q}$$

This equality follows from Lemma 7. In this case $l_1 = [a(p \to q)]$ by the definition of *Com*. Finally we show the equality

$$CI[\![[a(p \to q)] +\!\!+ l_2]\!]_{\gamma,\rho::P,stack,ws,0} = CI[\![l_2]\!]_{\gamma,\rho::P,T_{M,0,a(p\to q)}::stack,ws,0}$$

Which follows from the definition of CLVMI, proving the case

Case: $c = d \uparrow c'$

Next we consider the first inductive case of $c = d \uparrow c'$, from the assumption $Com(d \uparrow c') = \texttt{Some}\ l$ it follows from the definition of Com and the assumption $Com(d \uparrow c') = \texttt{Some}\ l_1$ that $Com(c') = \texttt{Some}\ l'$ for some list of CLVM instructions l'. From the definition of Com, l_1 is defined as

$$l_1 = [\text{Translate } d] +\!\!+ l' +\!\!+ [\text{TranslateEnd } d]$$

From the definition of CLVMI we get

$$\begin{aligned}&CI[\![[\text{Translate } d] +\!\!+ l' +\!\!+ [\text{TranslateEnd } d] +\!\!+ l_2]\!]_{\gamma,\rho::P,stack,ws,0} = \\ &CI[\![l' +\!\!+ [\text{TranslateEnd } d] +\!\!+ l_2]\!]_{\gamma,\rho/d::\rho::P,stack,ws,0}\end{aligned}$$

From the definition of the denotational semantics of CL the following equality holds.

$$C[\![d \uparrow c']\!]_{\gamma,Te(\rho)} = \begin{cases} delay(T,d) & \text{if } C[\![c']\!]_{\gamma,Te(\rho)/d} = \texttt{Some}\ T \\ \texttt{None} & \text{otherwise} \end{cases}$$

From our assumption $C[\![d \uparrow c']\!]_{\gamma,Te(\rho)} = \texttt{Some}\ T$ it follows that

$$\begin{aligned}&C[\![c']\!]_{\gamma,Te(\rho)/d} = \texttt{Some}\ T' \\ &C[\![d \uparrow c']\!]_{\gamma,Te(\rho)} = delay(d,T')\end{aligned} \tag{4.15}$$

From equation (4.15), Lemma 1 and the induction hypothesis, we get the equality

$$\begin{aligned}&CI[\![l' +\!\!+ [\text{TranslateEnd } d] +\!\!+ l_2]\!]_{\gamma,\rho/d::\rho::P,stack,ws,0} = \\ &CI[\![[\text{TranslateEnd } d] +\!\!+ l_2]\!]_{\gamma,\rho/d::\rho::P,T'_M::stack,ws,0}\end{aligned}$$

Where T'_M is a CLVM trace that satisfy the equation

$$Tt(T'_M) = T' \tag{4.16}$$

From the definition of CLVMI we get the equality

$$\begin{aligned}&CI[\![[\text{TranslateEnd } d] +\!\!+ l_2]\!]_{\gamma,\rho/d::\rho::P,stack,ws,0} = \\ &CI[\![l_2]\!]_{\gamma,\rho::P,delaym(n,T'_M)::stack,ws,0}\end{aligned}$$

Where the function $delaym(n, T_M)$ delays all CLVM traces by n days. This proves the first part of the hypothesis, finally we need to show the equality

$$Tt(delaym(n,T'_M)) = delay(d,T')$$

This follows from the Lemma 8 and equation (4.16), proving the case.

Case : $c = \texttt{Let}\ e\ \texttt{in}\ c'$

In the case of $c = \texttt{Let}\ e\ \texttt{in}\ c'$ it follows from the assumption $Com(\texttt{Let}\ e\ \texttt{in}\ c') = \texttt{Some}\ l_1$ and the definition of Com that

$$\begin{aligned} ECom(e) &= \texttt{Some}\ l_e \\ Com(c) &= \texttt{Some}\ l' \end{aligned}$$

For some list of CLVM expression instructions l_e and some list of CLVM instructions l', l_1 is then defined as

$$[\texttt{Let}\ l_e] +\!\!+ l' +\!\!+ [\texttt{LetEnd}]$$

From the assumption $C[\![\texttt{Let}\ e\ \texttt{in}\ c']\!]_{\gamma,Te(\rho)} = \texttt{Some}\ T$, and the definition of the CL denotational semantics, we get that

$$\begin{aligned} \varepsilon[\![e]\!]_{\gamma,Te(\rho)} &= \texttt{Some}\ v \\ C[\![c']\!]_{v::\gamma,Te(\rho)} &= \texttt{Some}\ T \end{aligned} \tag{4.17}$$

From the definition of CLVMI, equation 4.17 and Theorem 2 we get the equality

$$\begin{aligned} &CI[\![[\texttt{Let}\ l_e] +\!\!+ l' +\!\!+ [\texttt{LetEnd}] +\!\!+ l_2]\!]_{\gamma,\rho::P,stack,ws,0} = \\ &CI[\![l' +\!\!+ [\texttt{LetEnd}] +\!\!+ l_2]\!]_{v::\gamma,\rho::P,stack,ws,0} \end{aligned}$$

From equation 4.17, the definition of CLVMI and the induction hypothesis we get the equality

$$\begin{aligned} &CI[\![l' +\!\!+ [\texttt{LetEnd}] +\!\!+ l_2]\!]_{\gamma,\rho::P,stack,ws,0} = \\ &CI[\![l_2]\!]_{\gamma,\rho::P,T_M::stack,ws,0} \end{aligned}$$

For some T_M such that

$$Tt(T_M) = T$$

Proving the case

Case: $c = e \times c'$

In the case of $c = e \times c'$, it follows from the assumption $Com(e \times c') = \texttt{Some}\ l_1$ and the definition of Com, that

$$\begin{aligned} ECom(e) &= \texttt{Some}\ l_e \\ Com(c) &= \texttt{Some}\ l' \end{aligned}$$

l_1 is then defined as

$$l' +\!\!+ [\texttt{Scale}\ l_e] +\!\!+ l_2$$

From the assumption $C[\![e \times c']\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ T$, and the definition of the CL denotational semantics, we get that

$$\begin{aligned}
&\varepsilon[\![e]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ (\text{ZVal}\ z)\\
&C[\![c']\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ T'\\
&C[\![e \times c']\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ scale(z,T') = T
\end{aligned} \tag{4.18}$$

By equation (4.18) and the induction hypothesis we get the equality

$$\begin{aligned}
&CI[\![l' +\!\!+ [\texttt{Scale}\ \ l_e] +\!\!+ l_2]\!]_{\gamma,\rho::P,stack,ws,0} =\\
&CI[\![[\texttt{Scale}\ \ l_e] +\!\!+ l_2]\!]_{\gamma,\rho::P,T'_M::stack,ws,0}
\end{aligned}$$

For some T'_M such that

$$Tt(T'_M) = T' \tag{4.19}$$

From the definition of CLVMI, Theorem 2 and equation 4.18 we get the equality

$$\begin{aligned}
&CI[\![[\texttt{Scale}\ \ l_e] +\!\!+ l_2]\!]_{\gamma,\rho::P,T'_M::stack,ws,0} =\\
&CI[\![l_2]\!]_{\gamma,\rho::P,scalem(z,T'_M)::stack,ws,0}
\end{aligned}$$

Now we need to show that $Tt(scalem(z,T'_M)) = scale(z,T')$. From equation 4.19 we can rewrite this as the equality

$$Tt(scalem(z,T'_M)) = scale(z,Tt(T'_M))$$

Which follows from the Lemma 9, proving the case

Case: $c = c_1\&c_2$

In the case of $c = c_1\&c_2$ it follows from the assumption $Com(c_1\&c_2) = \texttt{Some}\ \ l_1$ and the definition of Com, that

$$\begin{aligned}
&Com(c_1) = \texttt{Some}\ \ l'_1\\
&Com(c_2) = \texttt{Some}\ \ l'_2
\end{aligned}$$

l_1 is then defined as

$$l'_1 +\!\!+ l'_2 +\!\!+ [\text{Both}]$$

From the assumption $C[\![c_1\&c_2]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ T$, and the definition of the CL denotational semantics, we get that

$$\begin{aligned}
&C[\![c_1]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ T_1\\
&C[\![c_2]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ T_2\\
&C[\![c_1\&c_2]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ add(T_1,T_2) = T
\end{aligned} \tag{4.20}$$

From the definition of CLVMI, equation 4.20 and the induction hypothesis we get the equality:

$$\begin{aligned}
&CI[\![l'_1 +\!\!+ l'_2 +\!\!+ [\text{Both}] +\!\!+ l_2]\!]_{\gamma,\rho::P,stack,ws,0} =\\
&CI[\![l_2]\!]_{\gamma,\rho::P,addM(T_{M1},T_{M2})::stack,ws,0}
\end{aligned}$$

For T_{M1}, T_{M2} such that

$$\begin{aligned} Tt(T_{M1}) &= T_1 \\ Tt(T_{M2}) &= T_2 \end{aligned} \tag{4.21}$$

Now we need to show the equality

$$add(T_1, T_2) = Tt(addM(T_{M1}, T_{M2}))$$

Which follows from Lemma 10 and equation 4.21, proving the case.

Case: $c = \texttt{if}\ e\ \texttt{within}\ d\ \texttt{then}\ c_1\ \texttt{else}\ c_2$

For the case of $c = \texttt{if}\ e\ \texttt{within}\ d\ \texttt{then}\ c_1\ \texttt{else}\ c_2$, from the assumption $Com(c) = \texttt{Some}\ \ l_1$ and the definition of Com we get the equalities

$$\begin{aligned} &ECom(e) = \texttt{Some}\ \ l_e \\ &Com(c_1) = \texttt{Some}\ \ l_1' \\ &Com(c_2) = \texttt{Some}\ \ l_2' \\ &l_1 = [\texttt{If}\ \ l_e\ \ d] +\!\!+\, l_2' +\!\!+\, [\texttt{Then}] +\!\!+\, l_1' +\!\!+\, [\texttt{IfEnd}] \end{aligned} \tag{4.22}$$

From the definition of CLVMI we get the following equality for any list of CLVM instructions l'

$$CI[\![[\texttt{If}\ \ l_e\ \ d] +\!\!+\, l']\!]_{\gamma,\rho::P,stack,ws,0} = \begin{cases} CI[\![l']\!]_{\gamma,\rho/(n-d_l)::\rho::P,stack,(n-d_l)::ws,1} & \text{if } wsem(l,n,\gamma,\rho) = \texttt{Some}\ \ (\texttt{True}, d_l) \\ CI[\![l']\!]_{\gamma,\rho/(n-d_l)::\rho::P,stack,(n-d_l)::ws,0} & \text{if } wsem(l,n,\gamma,\rho) = \texttt{Some}\ \ (\texttt{False}, d_l) \\ \texttt{None} & \text{otherwise} \end{cases} \tag{4.23}$$

We consider each case of $wsem(l,n,\gamma,\rho)$ separately

Case: $c = \texttt{if}\ e\ \texttt{within}\ d\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \wedge wsem(l,n,\gamma,\rho) = (\texttt{True}, d_l)$

For the case of $wsem(l,n,\gamma,\rho) = (\texttt{True}, d_l)$, the following equality follows from Lemma 17 and the definition of the denotational semantics

$$C[\![c]\!]_{\gamma,Te(\rho)} = \begin{cases} delay((n-d_l), T') & \text{If } C[\![c_1]\!]_{\gamma,Te(\rho)/(n-d_l)} = T' \\ None & \text{Otherwise} \end{cases}$$

From the assumption $C[\![c]\!]_{\gamma,Te(\rho)} = \texttt{Some}\ \ T$ it follows that

$$\begin{aligned} C[\![c_1]\!]_{\gamma,Te(\rho)/(n-d_l)} &= T' \\ C[\![c]\!]_{\gamma,Te(\rho)} &= delay(T', (n-d_l)) \end{aligned} \tag{4.24}$$

From equation 4.23 and the assumption $wsem(l,n,\gamma,\rho) = (\texttt{True}, d_l)$ we get the equality

$$\begin{aligned} &CI[\![[\texttt{If}\ \ l_e\ \ d] +\!\!+\, l_2' +\!\!+\, [\texttt{Then}] +\!\!+\, l_1' +\!\!+\, [\texttt{IfEnd}] +\!\!+\, l_2]\!]_{\gamma,\rho::P,stack,ws,0} = \\ &CI[\![l_2' +\!\!+\, [\texttt{Then}] +\!\!+\, l_1' +\!\!+\, [\texttt{IfEnd}] +\!\!+\, l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,stack,(n-d_l)::ws,1} = \end{aligned}$$

By Lemma 20 and equation 4.22 we get the equality

$$CI[\![l_2' ++ [\texttt{Then}] ++ l_1' ++ [\texttt{IfEnd}] ++ l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,stack,(n-d_l)::ws,1} =$$
$$CI[\![[\texttt{Then}] ++ l_1' ++ [\texttt{IfEnd}] ++ l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,stack,(n-d_l)::ws,1} =$$

From the definition of CLVMI we get the equality

$$CI[\![[\texttt{Then}] ++ l_1' ++ [\texttt{IfEnd}] ++ l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,stack,(n-d_l)::ws,1} =$$
$$CI[\![l_1' ++ [\texttt{IfEnd}] ++ l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,stack,(n-d_l)::ws,0} =$$

From the induction hypothesis, Lemma 1 and equation 4.24 we get the equality

$$CI[\![l_1' ++ [\texttt{IfEnd}] ++ l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,stack,(n-d_l)::ws,1} =$$
$$CI[\![[\texttt{IfEnd}] ++ l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,T_M'::stack,(n-d_l)::ws,0} =$$

Where T_M' is a CLVM trace such that

$$Tt(T_M') = T' \tag{4.25}$$

From the definition of CLVMI it follows that

$$CI[\texttt{IfEnd}] ++ l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,T_M'::stack,(n-d_l)::ws,1} =$$
$$CI[\![l_2]\!]_{\gamma,\rho::P,delaym((n-d_l),T_M')::stack,ws,0} =$$

Now we need to show that

$$delay((n-d_l),T') = Tt(delaym((n-d_l),T_M'))$$

From equation 4.25 it follows that

$$delay((n-d_l),Tt(T_M')) = Tt(delaym((n-d_l),T_M'))$$

Which follows from Lemma 8. Proving the case.

Case: $c = \textbf{if}\ e\ \textbf{within}\ d\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \wedge wsem(l,n,\gamma,\rho) = (\texttt{False}, d_l)$

For the case of $wsem(l,n,\gamma,\rho) = (\texttt{False}, d_l)$, the following equality follows from Lemma 18 and the definition of the denotational semantics

$$C[\![c]\!]_{\gamma,Te(\rho)} = \begin{cases} delay((n-d_l),T') & \text{If } C[\![c_2]\!]_{\gamma,Te(\rho)/(n-d_l)} = T' \\ None & \text{Otherwise} \end{cases}$$

From the assumption $C[\![c]\!]_{\gamma,Te(\rho)}$ it follows that

$$C[\![c_2]\!]_{\gamma,Te(\rho)/(n-d_l)} = T'$$
$$C[\![c]\!]_{\gamma,Te(\rho)} = delay(T',(n-d_l)) \tag{4.26}$$

From equation 4.23 and the assumption $wsem(l,n,\gamma,\rho) = (\texttt{False}, d_l)$ we get the equality

$$CI[\![[\texttt{If}\ \ l_e\ \ d] + l_2' + [\texttt{Then}] + l_1' + [\texttt{IfEnd}] + l_2]\!]_{\gamma,\rho::P,stack,ws,0} =$$
$$CI[\![l_2' + [\texttt{Then}] + l_1' + [\texttt{IfEnd}] + l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,stack,(n-d_l)::ws,0} =$$

From the induction hypothesis, Lemma 1 and equation 4.26 we get the equality

$$CI[\![l_2' + [\texttt{Then}] + l_1' + [\texttt{IfEnd}] + l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,stack,(n-d_l)::ws,0} =$$
$$CI[\![[\texttt{Then}] + l_1' + [\texttt{IfEnd}] + l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,T_M'::stack,(n-d_l)::ws,0} =$$

Where T_M' is a CLVM trace such that

$$Tt(T_M') = T' \tag{4.27}$$

From the definition of CLVMI we get the equality

$$CI[\![[\texttt{Then}] + l_1' + [\texttt{IfEnd}] + l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,T_M'::stack,(n-d_l)::ws,0} =$$
$$CI[\![l_1' + [\texttt{IfEnd}] + l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,T_M'::stack,(n-d_l)::ws,1} =$$

By Lemma 20 it follows that

$$CI[\![l_1' + [\texttt{IfEnd}] + l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,T_M'::stack,(n-d_l)::ws,1} =$$
$$CI[\![[\texttt{IfEnd}] + l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,T_M'::stack,(n-d_l)::ws,1} =$$

And from the definition of CLVMI we get

$$CI[\![[\texttt{IfEnd}] + l_2]\!]_{\gamma,\rho/(n-d_l)::\rho::P,T_M'::stack,(n-d_l)::ws,1} =$$
$$CI[\![l_2]\!]_{\gamma,\rho::P,delaym((n-d_l),T_M')::stack,ws,0} =$$

Now we need to show that

$$delay((n-d_l,T')) = Tt(delaym((n-d_l),T_M'))$$

From equation 4.27 it follows that

$$delay((n-d_l),Tt(T_M')) = Tt(delaym((n-d_l),T_M'))$$

Which follows from Lemma 8. Proving the case.

Case: $c = \texttt{if}\ e\ \texttt{within}\ d\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \wedge wsem(l,n,\gamma,\rho) = \texttt{None}$

In the case of $wsem(l,n,\gamma,\rho) = \texttt{None}$ we get the following equality by Lemma 19

$$C[\![c]\!]_{\gamma,Te(\rho)} = \texttt{None}$$

Which contradicts our assumption proving the case.

This concludes the induction proof of Lemma 15. Theorem 1 follows as special case of Lemma 15 Q.E.D

4.5 Evaluating the certification

The previous sections presented the proof of Theorem 1. The theorem states that compilation and CLVMI interpretation of CL contracts preserves the denotational semantics of CL. We consider this Coq formalized and verified theorem to be the certification of our compilation scheme. In this section we discuss the strength of this certification. Which guarantees do we actually provide to the end user?

The CL paper [7] defines a CL contract as closed if it is well-typed with the empty variable assignment. The denotational semantics of a closed contract with the environment ρ is written $C[\![c]\!]_\rho$. The paper also introduce the principle of causality. They define the equivalence relation $=_t$ for $t \in \mathbb{Z}$, as $\rho_1 =_t \rho_2$ iff. $\rho_1(l,s) = \rho_2(l,s)$ for any $s \leq t$. A closed contract c is then defined as causal iff. $\rho_1 =_t \rho_2$ implies $C[\![c]\!]_{\rho_1}(t) = C[\![c]\!]_{\rho_2}(t)$

As mentioned previously the use case of our compilation scheme is as follows: The user defines a CL contract and uses the temporal type-system of CL to ensure that the contract is well-formed, closed and causal. The contract is then compiled to CLVM. The user can then deploy a contract manager which will use the financial-contract evaluator which we defined, using the compiled CL contract as the representation of the financial contract. The manager will then query the evaluator, as required, with an environment and some natural number q. q specifies the offset from deployment, up to which the contract manager requires the stipulated transfer. The evaluator then evaluates the CLVM contract using CLVMI, and returns the trace of transfers from the CLVMI output trace, which are stipulated between the last queried time and q.

Since the contract is well-formed, the denotational semantics of CL will always specify some output for any provided environment. By Theorem 1, the output specified by the financial-contract evaluator will always adhere to these semantics. Since the proof is verified in Coq the trusted computing base is the Coq kernel and the extraction procedure of ConCert. It is the responsibility of the contract manager to provide an environment which contain all the required observables for the desired evaluation of the contract. Since the contract is causal, the transfers stipulated by the denotational semantics of CL at time q will never depend on an observable value at a time later than q. Since CLVMI interpretation of compiled CL contracts follow the denotational semantics, the contract manager only needs to provide environments with correct observables up to time q to get the desired trace of stipulations. We consider this to be a strong certification of the compilation scheme.

There are two possible improvements to the certification, which we consider as possible future work. While it might seem intuitive from the definition of the interpreter we do not provide a proof that all well-typed CL contracts will compile, we only certify the evaluation of the contracts that do. We suspect this proof could be provided without much effort. Secondly, if the contract evaluator is given an environment which does not contain all observables required for c up to time q, the evaluator will treat the value of a missing observables simply as `false` or 0, depending on the observable label, yielding a potentially undesired trace. A contract manager should never want an evaluation by a faulty environment, so providing run-time exceptions when encountering a missing observable would be preferable over a default value.

Chapter 5

Conclusion

In this work we set out to examine whether or not it is possible to develop a certified compilation scheme from a domain specific language (DSL) for financial contracts with a formalized semantics, to functional smart contract languages. And if by utilizing such a compilation scheme, we could define a smart-contract financial contract evaluator, which evaluates a financial contract DSL in accordance with a formalized semantics.

As described in chapter 3 we designed an imperative intermediate language for CL, called CLVM along with a definitional interpreter CLVMI. We argue that CLVM can be represented and evaluated by CLVMI in a smart-contract of the most common functional smart contract languages. Utilizing this compilation scheme we implemented a financial-contract interpreter for compiled CL as a smart-contract in the ConCert framework. While we were not able to extract our Coq formalization, we argue that this smart-contract is extractable to Liquidity by the extraction procedure provided by ConCert.

We certified the compilation scheme by proving that interpretation of compiled CL contracts follow the denotational semantics of the source CL contract. We formalized and verified this proof in Coq. This guarantees that our financial-contract evaluator will evaluate the compiled CL contract in accordance with the formalized semantics of CL, relying only on the trusted computing base of the Coq verification kernel and the extraction to Liquidity. The proof is presented in chapter 4. This is the first financial contract evaluator with a formalized semantics which can run as a smart-contract on a blockchain. Furthermore the compilation scheme allows users to specify contracts directly in the CL language. We consider this to be a significant contribution to the field of financial contract evaluation on blockchain.

Chapter 6

Future work

While we argue that our compilation scheme and financial-contract evaluator is extractable to Liquidity, we did not manage to prove this. Extracting the code could be a first step in furthering this work. In addition to the denotational semantics, the paper [7] also presents a reduction semantics for CL. The reduction semantics continuously stipulates the transfers obligated by the contract and shrinks it to a contract representing the remaining obligations. Implementing a CLVM interpreter which follows the reduction semantics of CL could potentially lead to a more efficient financial-contract evaluator. In this project we do not consider any specific contract manager for use with our contract evaluator. However, we do require the contract manager to provide environments with all observables need by the contract up to the time of querying. This is a significant requirement and if it is not met, it could potentially lead to erroneous evaluation. Adding a runtime exception, for missing observables, to CLVMI could solve this problem. By implementing the smart contract in ConCert we also allow for future investigation of the interaction between different contract managers and our contract evaluator.

Bibliography

[1] The coq proof assistant. `https://coq.inria.fr/`. Accessed: 2020-14-06.

[2] The formal coq specification of cl. `https://github.com/HIPERFIT/contracts`. Accessed: 2020-27-05.

[3] The github page of clvm. `https://github.com/malthelange/CLVM`. Accessed: 2020-14-06.

[4] The github page of the concert framework. `https://github.com/AU-COBRA/ConCert`. Accessed: 2020-14-06.

[5] The liquidity smart-contract language. `https://www.liquidity-lang.org/`. Accessed: 2020-14-06.

[6] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 215–228, 2020.

[7] Patrick Bahr, Jost Berthold, and Martin Elsman. Certified symbolic management of financial multi-party contracts. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP'2015, pages 315–327, September 2015.

[8] Benjamin Egelund-Müller, Martin Elsman, Fritz Henglein, and Omry Ross. Automated execution of financial contracts on blockchains. *Business and Information Systems Engineering*, 59(6):457–467, 12 2017.

[9] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: An adventure in financial engineering (functional pearl). *SIGPLAN Not.*, 35(9):280–292, September 2000.

[11] Pablo Lamela Seijas and Simon Thompson. Marlowe: Financial contracts on blockchain. In *International Symposium on Leveraging Applications of Formal Methods*, pages 356–375. Springer, 2018.

[12] V. U. Wickramarachchi, C. I. Keppitiyagama, and K. G. Gunawardana. Efficiently transform contracts written in peyton jones contract descriptive language to solidity. In *2019 19th International Conference on Advances in ICT for Emerging Regions (ICTer)*, volume 250, pages 1–8, 2019.