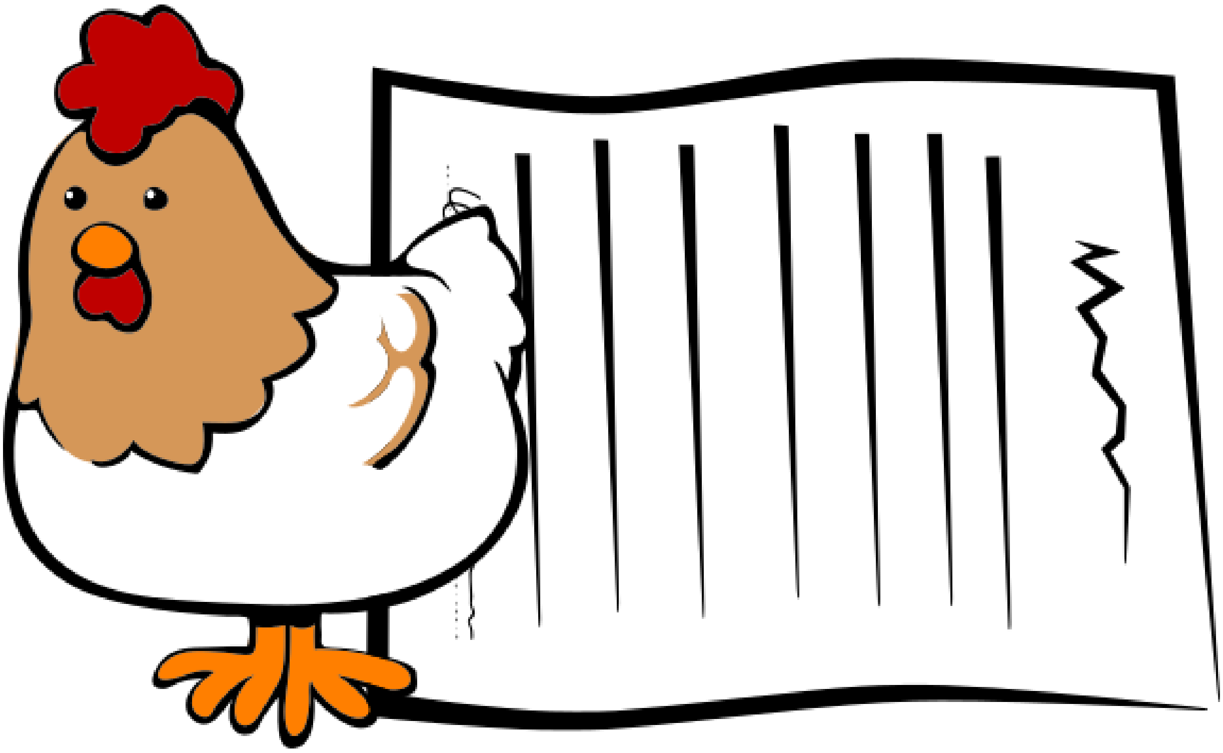
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Masters thesis

Christian Kjær Larsen — `c.kjaer@di.ku.dk`

Declarative Contracts

Mechanized semantics and analysis

Supervisors: Fritz Henglein and Agata Murawska

September 2nd, 2019

Abstract

In this thesis we explore formal verification of a contract specification language (CSL) for general purpose business contracts. For this we use the Coq proof assistant. CSL is a declarative language that specifies a valid set of event sequences satisfying a contract. Properties of contracts can be proved using the formal semantics of CSL, but with the help of a proof assistant we can gain confidence that our proofs are actually true.

The three main contributions of this thesis are the following:

First, we mechanize 4 different semantics of CSL in Coq. A big step semantics, a denotational semantics and two different reduction semantics. We then prove a selection of meta-theorems relating them in different ways. This mechanization helped us find mistakes in the original operational semantics for CSL.

Second, we design an abstract interpretation framework for CSL. This framework is designed to infer properties of traces that satisfy a given contract. We then use this framework to define two concrete analyses. Participation analysis will give an approximation of the participating agents and how they transfer resources in a contract. Fairness analysis will approximate the gains and losses different agents have by participating in a contract. The analysis framework is proven sound with respect to the formal semantics of CSL. The analysis framework can be instantiated by proving abstract domains for the analysis and a few combinators. If the combinators have certain properties, then the analysis is sound.

Finally, we mechanize the abstract interpretation framework in Coq and mechanize its soundness proof. We also mechanize some abstract domains and use them to mechanize both participation analysis and fairness analysis and thereby prove their soundness with respect to the semantics of CSL.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contributions . . . 3
1.3 Roadmap . . . 3

2 A Compositional Contract Language **5**
2.1 Syntax . . . 5
2.2 Typing . . . 7
2.3 Contract satisfaction . . . 7
2.4 Denotational semantics . . . 10
2.5 Reduction semantics . . . 10
2.6 Resources and agents . . . 16

3 Proofs and programs **18**
3.1 Propositions as types . . . 18
3.2 The Coq proof assistant . . . 19
3.3 An interval abstraction . . . 20

4 Mechanization of CSL in Coq **26**
4.1 Bindings . . . 26
4.2 Expressions . . . 27
4.3 Syntax . . . 28
4.4 Contract satisfaction . . . 29
4.5 Denotational semantics . . . 29
4.6 Substitution . . . 32
4.7 Residuation . . . 33

5 Abstract interpretation **37**
5.1 Collecting semantics . . . 37
5.2 Galois connections . . . 38
5.3 Fixed point algorithms . . . 39

6 Contract analysis **41**
6.1 Abstract interpretation of CSL . . . 42
6.2 Analysis of expressions . . . 43
6.3 A collecting semantics for CSL . . . 46
6.4 An abstract interpreter . . . 56
6.5 A general framework for contract analyses . . . 57
6.6 Participation analysis . . . 58

6.7 Fairness analysis 59

7 Contract analyses in Coq 64

7.1 Generic analysis 64

7.2 Analysis instances 66

8 Discussion 71

8.1 Conclusion 71

8.2 Related work 72

8.3 Future work 75

Bibliography 76

A Overview of the source code 80

A.1 The basic mechanization 80

A.2 Verification of the analysis 81

A.3 Prototype implementation 82

B Additional background 83

B.1 Lattices 83

B.2 Lattices in Coq 85

Chapter 1

Introduction

In this chapter we will motivate the thesis, and we will give an overview of the contributions and the structure.

1.1 Motivation

In recent years there has been much interest in distributed ledger technologies (DLT) where blockchain is the most popular type. Many DLTs support smart contracts which are computer protocols that facilitate, verify or enforce contracts.

The most popular smart contract platform at the moment is Ethereum. Its smart contracts run on the Ethereum Virtual Machine (EVM). Ethereum contracts, and many other smart contracts, are written in imperative or object oriented languages that are then compiled to low level virtual machines. Ethereum contracts have had a lot of security issues, one famous example being the DAO hack where hackers stole around 50 million dollars from a single smart contract.

Current research [Bha+16] suggests that both the openness of Ethereum contracts and their intricate semantics makes writing correct and secure contracts very difficult. It is possible for an attacker to carefully analyze the contract source code (it is publicly available on the blockchain network) and make a sophisticated attack. Furthermore contracts are difficult to patch, since there is no way to update the code of a contract once it is deployed.

Another approach to smart contracts is the Contract Specification Language (CSL) currently under development by Deon Digital[1], which is based on early work by Andersen, Elsborg, Henglein, Simonsen and Stefansen [And+06].

CSL

CSL is a process-calculus inspired domain-specific language for writing business contracts. Where Ethereum contracts are essentially Turing-complete programs that are very hard to predict the behaviour of, CSL contracts are specifications. They describe sequences of events that are matched by a contract. The strategies of parties can, contrary to Ethereum contracts, be kept private and can not be analyzed by possible attackers. This is true to what the purpose of a traditional contract is. It specifies the rules that govern legal behaviors among consenting parties, not force how they act. For an overview

[1] https://docs.deondigital.com/latest/

of formal contract modelling approaches, Hvitved [Hvi10] provides a comprehensive survey.

One semantics of CSL is given by a relation $\delta \vdash^D s : c$, which states that the trace (sequence of events) s satisfies contract c in the environment δ. As an example we can write a simplified loan contract with 12 recurring payments in CSL as follows:

```
letrec repay[amount, payments] =
    Transfer("alice", "bob", amount, _ | payments = 1).Success
  + Transfer("alice", "bob", amount, _ | payments > 1).repay(amount, payments - 1)
in Transfer("bob", "alice", 12000 EUR, _).repay(1000 EUR, 12)
```

The Transfer is at the core of CSL. Every syntactic occurence of a Transfer represents a commitment to transfer a resource before finishing the remaining contract. The $+$ represents a choice between two subcontracts.

A satisfying trace for the loan contract contract will look like

$$\langle \mathsf{transfer}(\texttt{bob}, \texttt{alice}, 12000 \text{ EUR}, t_0), \mathsf{transfer}(\texttt{alice}, \texttt{bob}, 1000 \text{ EUR}, t_1), \ldots, \\ \mathsf{transfer}(\texttt{alice}, \texttt{bob}, 1000 \text{ EUR}, t_{12})\rangle.$$

Where Bob first transfers 12000 euros to Alice, and then Alice makes 12 repayments of 1000 euros to Bob.

In this thesis we mechanize the semantics of CSL in Coq. This mechanized semantics can be used to prove interesting properties about CSL. For instance we can prove that the contract $(c; c_1) + (c; c_2)$ is equivalent to $c; (c_1 + c_2)$. First we define contract equivalence in Coq using our mechanized contract satisfaction relation:

```
Definition contract_equiv Γ Δ (c c' : contract Γ Δ) :=
  ∀ (D : template_env Γ) (δ : env Δ) (s : trace),
    csat δ D s c ↔ csat δ D s c'.
```

contract_equiv c c′ encodes that c and c′ has the same satisfying traces. Now we prove that

```
Lemma seq_plus : ∀Γ Δ (c c1 c2 : contract Γ Δ),
  contract_equiv ((c ;; c1) :+: (c ;; c2)) (c ;; (c1 :+: c2)).
```

There are more semantics of CSL that we mechanize in this thesis, and we will prove much more interesting properties about them than this example.

Contract analysis

One claim in the original paper by Andersen et al. was that their contract specification language was in some way analyzable. In the paper they go on to define two very simple analyses that are basically syntax directed. They also explain that their analyses are correct but that they do not work for all contracts. For a contract language it would be very beneficial to have analysis methods that can be used both when developing new contracts and when monitoring the performance of deployed contracts.

In this thesis we develop a sound general analysis method that can infer properties of satisfying traces for contracts. If we can describe a property of a trace with a function $\beta : Tr \to L$ to some set L of properties, and if β has some some special properties, then we can in general find some ℓ that is a sound over-approximation for $\beta(s)$ for all s satisfying the contract.

For instance we develop participation analysis where we can infer what parties transfer resources between each other in the contract. For the loan contract we can infer a rather trivial property, namely that only Bob and Alice communicate. We also develop fairness analysis which bounds the gains and losses by participating in a contract. For the loan contract we can infer that neither Bob or Alice gains anything from participating in the contract. In a satisfying trace, Bob pays 12000 euros, but also receives them again.

When developing analyses, their correctness is crucial. If the result of the analyses are relied on then one should really trust that they are correct. For this, proof verification is important. We also verify the soundness of the analysis methods with respect to the mechanized semantics of CSL.

To summarize, the goal of this thesis is to build a theoretical framework necessary for the formal analysis of digital contracts. We plan to do the following:

1. Mechanize the formal semantics of CSL using a proof assistant.

2. Build a theoretical framework for formal analysis of CSL.

3. Use the mechanized semantics of CSL to verify the correctness of the analysis method.

1.2 Contributions

The main novel contributions of this thesis are the following:

- A mechanization of the syntax and semantics of CSL in Coq. We have also mechanized a number of proofs about the meta-theory of CSL. The mechanization was developed in collaboration with Murawska.

- A general approach to static analysis of CSL by means of abstract interpretation. This is done by deriving a collecting semantics from the natural semantics of CSL and then abstracting it using the concept of Galois connections. We have also developed some concrete analyses that have some practical interest.

- Mechanization of parts of the static analyses in Coq which includes the mechanization of a number of abstract domains and the definition of a generic analysis using type classes.

1.3 Roadmap

- In Chapter 2 we introduce the Contract Specification Language (CSL), give the formal semantics and describe a resource model that will be useful for the analysis.

- In Chapter 3 we give a short introduction to proofs as programs and how we can use the Coq proof assistant for mechanization of programming language theory.

- In Chapter 4 we describe our Coq mechanization of CSL and show the most important theorems that we have mechanized.

- In Chapter 5 we introduce the general idea of abstract interpretation.

- In Chapter 6 we develop an approach for static analysis of CSL by means of an abstract collecting semantics and use it to specify 2 concrete static analyses and prove their correctness. We also show how to implement an abstract interpreter for CSL using abstract semantics trees.

- In Chapter 7 we show how we mechanized the analyses and proved their correctness in Coq.
- In Chapter 8 we evaluate our work, look at related work and touch upon future work.

Chapter 2

A Compositional Contract Language

In this section we will describe a compositional contract language for writing general multi-party contracts [And+06]. Except for the last section on agent and resource models, this chapter does not contain any contributions and is just meant to give an overview of CSL though some definitions will differ a bit from the original paper.

The contract specification language (CSL) is based on Peyton-Jones and Eber′s compositional specification for financial contracts [Jon01] but generalized to work for multiple parties and not only for financial contracts involving currencies.

The language is inspired by the Resources-Events-Agents (REA) ontology by McCarthy [McC82] who describe an accounting model based only on the fact that events happen where scarce resources are transferred between agents, created, transformed or consumed.

At the heart of the REA model is an exchange where a resource changes ownership from one agent to another. This is modeled in CSL by the $\mathsf{transfer}(a_1, a_2, r, t)$ event where a_1 sends resource r to a_2 at time t. Creation, transformation and consummation is not modeled by CSL. This event model is different from the the model in the language described by Peyton-Jones and Ebers in the sense that they only specify financial contracts where the agents are implicit and the scarce resources are currencies.

The language is built on top of a domain of agents $\mathcal{A}$, resources $\mathcal{R}$ and times $\mathcal{T}$ to model the transfer of resources. Agents are the entities transferring resources. It could be entities like persons, companies, cryptographic keys in a blockchain system and so on. Resources could be, say, currencies or physical assets. Times could be represented as times with time zones (think ISO time) or as a time difference since some epoch (think UNIX time).

We start by describing the syntax of CSL and describing the constructs of the language.

2.1 Syntax

The entire syntax is shown in Figure 2.1. c is the syntax of contracts, td is the syntax of a single template declaration and cd is the syntax for a contract specification with a set of mutually recursive contract templates. We describe the syntax constructors for c.

- $\mathsf{Success}$ and $\mathsf{Failure}$ are the basic contracts. They represent the successful and the breached contract respectively.

$$
\begin{array}{rcll}
c & ::= & \textsf{Success} & \text{(The completed contract)} \\
 & \mid & \textsf{Failure} & \text{(The breached contract)} \\
 & \mid & \textsf{Transfer}(A_1, A_2, R, T \mid P).c & \text{(Commitment to transfer)} \\
 & \mid & c_1 + c_2 & \text{(Alternatively)} \\
 & \mid & c_1 \parallel c_2 & \text{(Concurrently)} \\
 & \mid & c_1 \ ; \ c_2 & \text{(Sequentially)} \\
 & \mid & f(a_1, \ldots, a_n) & \text{(Template application)} \\
 & & & \\
td & ::= & f[x_1, \ldots, x_n] = c & \text{(Template declaration)} \\
cd & ::= & \textsf{letrec}\ td_1 \ldots td_n\ \textsf{in}\ c & \text{(Top level contract declaration)}
\end{array}
$$

Figure 2.1: The core CSL syntax

- $\textsf{Transfer}(A_1, A_2, R, T \mid P).c$ represents the contract that to be successful commits to a transfer event to occur before requiring c to be successful. A_1, A_2, R and T are variables where A_1 represents the sender and A_2 the receiver of the event. R is the resource transferred in the event and T is the time the event occurred. P is a predicate where the variables are bound and if the predicate is true, the event is matched. The variables are bound in c as well. This constructor is the basic building block of contracts and is the only place transfers are represented. We will introduce syntactic sugar for when we are matching literals. For $a_1, a_2 \in \mathcal{A}$, $r \in \mathcal{R}$ we let $\textsf{Transfer}(a_1, a_2, r, T \mid P).c$ abbreviate

  $$\textsf{Transfer}(A_1, A_2, R, T \mid A_1 = a_1 \wedge A_2 = a_2 \wedge R = r \wedge P).c,$$

  where A_1, A_2 and R are fresh.

- $c_1 + c_2$ composes c_1 and c_2 by making a choice possible. For the contract to be successful, exactly one of c_1 and c_2 should be successful.

- $c_1 \parallel c_2$ composes c_1 and c_2 and is successful if c_1 and c_2 are both successful but both can accept events concurrently. This can be used to specify independent contracts.

- $c_1; c_2$ composes c_1 and c_2 and is successful if both subcontracts are successful and c_1 happens entirely before c_2. This can be used to specify contracts where it is important that one contracts is successful before another one happens.

- $f(a_1, \ldots, a_n)$ is the application of a contract template. Where f is the name of the template. It is successful if the body of the template is successful with the formal parameters bound to their values. $a_1, \ldots, a_n$ are expressions from an expression language that we will describe later. The contract templates are mutually recursive, which allows recurring contracts to be specified.

Note that we did not specify the syntax of expressions to be used in predicates and arguments to templates. Later in this thesis we will discuss what consequences the design of the expression language has on the expressibility and analyzability of the contracts.

We will assume that for the expression language $\mathcal{P}$ with expressions e that there exists valid typing judgments $\Delta \vdash e : \tau$, where $\tau \in \{\textsf{Agent}, \textsf{Resource}, \textsf{Time}, \textsf{Boolean}\}$, and a denotation $\mathcal{Q}[\![e]\!]^{\delta}$ that maps expressions to values of type τ given that $\Delta \vdash e : \tau$ and δ agrees with Δ. We write $\delta \models P$ to denote that $\mathcal{Q}[\![P]\!]^{\delta} = \textsf{true}$ and $\delta \not\models P$ to denote that $\mathcal{Q}[\![P]\!]^{\delta} = \textsf{false}$.

2.2 Typing

To define typing rules for CSL, we start by defining two different typing contexts. Δ maps variables to types and Γ maps template names to typing signatures, in this case represented by Δ.

$$\Delta ::= [x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n]$$
$$\Gamma ::= [f_1 \mapsto \Delta_1, \dots, f_n \mapsto \Delta_n]$$

We let $\Delta[x \mapsto \tau]$ denote removing all occurrences of x from the list Δ, and adding $x \mapsto \tau$ at the and. We let $\Delta[x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n]$ abbreviate $\Delta[x_1 \mapsto \tau_1] \cdots [x_n \mapsto \tau_n]$. We let $\Gamma(f)$ denote finding the last occurrence of $f \mapsto \Delta$ in Γ and returning Δ. The typing rules of CSL can be seen in Figure 2.2.

The typing rules for CSL are pretty simple, and most of the work is used on making sure that the contract templates are well-typed and that variables of the correct type are bound in the Transfer along with requiring that the predicate evaluates to a boolean.

2.3 Contract satisfaction

Now it is time to describe what a contract actually means. In this section we give a big-step semantics for contract satisfaction. It characterizes the satisfying traces of a contract specification.

We mentioned events as part of the REA ontology, so they are surely going to be important in the semantics. We only allow one kind of event which is the transfer of a resource between two agents. For $a_1, a_2 \in \mathcal{A}$, $r \in \mathcal{R}$ and $t \in \mathcal{T}$ we describe the transfer event by the following syntax:

$$e ::= \textsf{transfer}(a_1, a_2, r, t)$$

where $\textsf{transfer}(a_1, a_2, r, t)$ represents a transfer of resource r at time t from a_1 to a_2. An event trace is a finite sequence $Tr = e^*$ of events such that the timestamps occur in non-decreasing order. We write $\langle\rangle$ for the empty trace, $\langle e_1, \dots, e_n\rangle$ for a sequence of events.

We define an environment of contract templates D as a mapping from template names to their template declarations. We define environment of values δ as a list of mappings from variables to values.

$$D ::= [f_1 \mapsto td_1, \dots, f_n \mapsto td_n] \tag{2.1}$$
$$\delta ::= [x_1 \mapsto v_1, \dots, x_n \mapsto v_n] \tag{2.2}$$

We define environment look-ups and updates in a similar fashion for D and δ as we did for typing environments in the previous section. To construct D For a set of template declarations $t_1, \dots, t_n$ we write

$$td_1, \dots, td_n \checkmark [f_1 \mapsto td_1, \dots f_n \mapsto td_n]$$

$\boxed{\Delta, \Gamma \vdash c}$ Contract c is well typed in Δ with template typings Γ

$$\text{T-Success}\;\frac{}{\Delta, \Gamma \vdash \textsf{Success}} \qquad \text{T-Failure}\;\frac{}{\Delta, \Gamma \vdash \textsf{Failure}}$$

$$\text{T-TemplateApp}\;\frac{\forall i \in \{1, \ldots, n\} :\; \Delta \vdash a_i : \tau_i}{\Delta, \Gamma \vdash f(a_1, \ldots, a_n)}\,(\Gamma(f) = [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n])$$

$$\text{T-Transfer}\;\frac{\begin{array}{c}\Delta' = \Delta[A_1 \mapsto \textsf{Agent}, A_2 \mapsto \textsf{Agent}, R \mapsto \textsf{Agent}, T \mapsto \textsf{Time}] \\ \Delta' \vdash P : \textsf{Boolean} \quad \Delta', \Gamma \vdash c\end{array}}{\Delta, \Gamma \vdash \textsf{Transfer}(A_1, A_2, R, T \mid P).c}$$

$$\text{T-Alternative}\;\frac{\Delta, \Gamma \vdash c_1 \quad \Delta, \Gamma \vdash c_2}{\Delta, \Gamma \vdash c_1 + c_2} \qquad \text{T-Concurrent}\;\frac{\Delta, \Gamma \vdash c_1 \quad \Delta, \Gamma \vdash c_2}{\Delta, \Gamma \vdash c_1 \parallel c_2}$$

$$\text{T-Sequence}\;\frac{\Delta, \Gamma \vdash c_1 \quad \Delta, \Gamma \vdash c_2}{\Delta, \Gamma \vdash c_1 \;;\; c_2}$$

$\boxed{\Gamma \vdash td}$ Template declaration td is well typed in Γ

$$\text{T-Decl}\;\frac{[x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n], \Gamma \vdash c}{\Gamma \vdash f[x_1, \ldots, x_n] = c}\,(\Gamma(f) = [\tau_1, \ldots, \tau_n])$$

$\boxed{\Gamma \vdash cd}$ Top level declaration cd is well typed in Γ

$$\text{T-Letrec}\;\frac{[], \Gamma \vdash c \quad \forall i \in 1, \ldots, n :\; \Gamma \vdash t_i}{\Gamma \vdash \textsf{letrec}\; t_1 \ldots t_n \;\textsf{in}\; c}$$

Figure 2.2: Typing rules for contracts

if $td_i = f_i[x_1, \ldots, x_m] = c_i$ for all $i \in 1 \ldots n$. This definition simply constructs the template environment by destructing the template declarations and binding the names of the templates to the declarations.

The judgments for contract satisfaction can be seen on Figure 2.3. In contrast to [And+06] we do not include the global environment. This is for easier formalization and to make the analyses easier to state. They will be trivial to add but will only make the definitions more complex. We describe the rules for contract satisfaction:

S-Success Only the empty trace satisfies the successful contract, since the successful contract has no more obligations.

S-Transfer The contract that commits to a transfer is satisfied if we have some event at the head of the the trace and the predicate evaluates to true with the values in the event bound and if the subcontract is satisfied with the tail of the event trace.

S-AltLeft, S-AltRight $c_1 + c_2$ is satisfied if either c_1 or c_2 is satisfied with the event trace.

$\boxed{\delta \vdash^D s : c}$ s is a satisfying trace for c in δ with templates D.

$$\text{S-Success}\;\frac{}{\delta \vdash^D \langle\rangle : \mathsf{Success}}$$

$$\text{S-Template}\;\frac{\begin{array}{c}\forall i \in \{1,\ldots,n\} : v_i = \mathcal{Q}[\![a_i]\!]^{\delta} \\ [x_1 \mapsto v_1, \ldots x_n \mapsto v_n] \vdash^D s : c\end{array}}{\delta \vdash^D s : f(a_1,\ldots,a_n)}\;(D(f) = (f[x_1,\ldots,x_n] = c))$$

$$\text{S-Transfer}\;\frac{\begin{array}{c}\delta' = \delta[A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t] \\ \delta' \vdash^D s : c \quad \delta \models P\end{array}}{\delta \vdash^D \mathsf{transfer}(a_1,a_2,r,t)s : \mathsf{Transfer}(A_1,A_2,R,T \mid P).c}$$

$$\text{S-AltLeft}\;\frac{\delta \vdash^D s : c_1}{\delta \vdash^D s : c_1 + c_2} \qquad \text{S-AltRight}\;\frac{\delta \vdash^D s : c_2}{\delta \vdash^D s : c_1 + c_2}$$

$$\text{S-Concurrent}\;\frac{\delta \vdash^D s_1 : c_1 \quad \delta \vdash^D s_2 : c_2 \quad (s_1,s_2) \rightsquigarrow s}{\delta \vdash^D s : c_1 \parallel c_2}$$

$$\text{S-Sequence}\;\frac{\delta \vdash^D s_1 : c_1 \quad \delta \vdash^D s_2 : c_2 \quad s_1 +\!\!+\, s_2 = s}{\delta \vdash^D s : c_1 ; c_2}$$

$\boxed{s : td}$ s is a satisfying trace for the top level contract specification td.

$$\text{S-Letrec}\;\frac{t_1,\ldots,t_n \checkmark D \qquad [] \vdash^D s : c}{s : \mathsf{letrec}\ t_1,\ldots,t_n\ \mathsf{in}\ c}$$

Figure 2.3: Rules for contract satisfaction

S-Concurrent $c_1 \parallel c_2$ is satisfied for s if there is an interleaving (written $(s_1, s_2) \rightsquigarrow s$) such that s_1 satisfies c_1 and s_1 satisfies c_2.

S-Sequence $c_1; c_2$ is satisfied for s if we can split the event trace s into s_1 s_2 such that s_1 satisfies c_1 and s_2 satisfies c_2. We write this as an appending $s_1 +\!\!+\, s_2 = s$.

S-Template A template application is satisfied if the trace satisfies the body of the template with the formal parameters bound to their value when evaluated.

S-Letrec For a top level contract specification we construct the template environment D and we then require that the contract c is satisfied with the templates in the empty environment.

Note that there are no rules for $\mathsf{Failure}$, since a breached contract cannot be satisfied.

2.4 Denotational semantics

We are not going to explain the denotational semantics of CSL in detail, so an interested reader can read more in [And+06]. The main idea of the denotational semantics is that contracts are denoted by trace sets $S \in \mathcal{P}(Tr)$ by a denotation function $\mathcal{C}$. We specify this informally here as a function mapping contracts to semantic functions from denotations of template environments and environment to sets of traces:

$$\mathcal{C} : \mathsf{Contract} \to (Dom(D) \to Dom(\Delta) \to \mathcal{P}(Tr))$$

And contract templates are denoted by a function mapping template environments to meanings of their bodies.

$$\mathcal{D} : D \to (TName \to Dom(\Delta) \to \mathcal{P}(Tr))$$

We write $\mathcal{C}[\![c]\!]^{\mathcal{D}[\![D]\!];\delta} : \mathcal{P}(Tr)$ for the denotation of a contract c in an environment δ with denoted templates D. We can prove that the denotation of a contract coincides with all the derivable trace satisfactions.

Theorem 2.1. $\mathcal{C}[\![c]\!]^{\mathcal{D}[\![D]\!];\delta} = \{s \mid \delta \vdash^D s : c\}$

2.5 Reduction semantics

Now we want to go from just checking that a trace satisfies a contract to an operational semantics, where we can monitor contract execution in time as events occur. We will not go into depth, and a more detailed explanation can be found in [And+06].

Conceptually we can do the following to monitor a contract:

- Compute the set of all satisfying traces for a contract S_0. If $S_0 = \emptyset$, output that the contract is breached.
- For events $e_n \in \{e_1, e_2, \ldots\}$ do
  - Compute $S_n = \{s \mid e\, s \in S_{n-1}\}$
  - If $S_n = \emptyset$ output that the contract is breached.
  - If at any time we want to conclude the contract, we just check whether $\langle\rangle \in S_n$, and we can stop contract monitoring.

But we cannot simply compute the set of all satisfying traces of a contract. To work around this we now conceptually extend the language with a residuation operator $e\backslash c$ to mean the residual contract of c given the event e. The traces that satisfy this residual contract are the following in an environment δ and with templates D:

$$\frac{\delta \vdash^D e\, s : c}{\delta \vdash^D s : e\backslash c}$$

We can use this insight to define equality of a residual contract

Definition 2.1 (Residuation equality). *To denote equality of residuation we write $D \vdash c' = e\backslash c$ to mean that*

$$\forall s, \delta . \delta \vdash^D e\ s : c' \iff \delta \vdash^D s : c.$$

$\boxed{D \vdash c \text{ nullable}}$

$$\frac{}{D \vdash \mathsf{Success}\text{ nullable}} \qquad \frac{D \vdash c \text{ nullable}}{D \vdash c + c' \text{ nullable}} \qquad \frac{D \vdash c' \text{ nullable}}{D \vdash c + c' \text{ nullable}}$$

$$\frac{D \vdash c \text{ nullable} \qquad D \vdash c' \text{ nullable}}{D \vdash c \parallel c' \text{ nullable}} \qquad \frac{D \vdash c \text{ nullable} \qquad D \vdash c' \text{ nullable}}{D \vdash c; c' \text{ nullable}}$$

$$\frac{D \vdash c \text{ nullable}}{D \vdash f(a_1, \ldots, a_n) \text{ nullable}} D(f) = (f[x_1, \ldots, x_n] = c)$$

Figure 2.4: Syntactic nullability

We also need to state that a contract has at least the behavior of another contract

Definition 2.2 (Contract subset). *To denote that a contract has at least the satisfying traces of another one we write $D \vdash c' \subseteq c$ to mean that*

$$\forall s, \delta . \delta \vdash^D s : c' \Rightarrow \delta \vdash^D s : c.$$

We also characterize that a contract has at least the behavior of a residual contract.

Definition 2.3 (Residuation subset). *We write $D \vdash c' \subseteq e \backslash c$ to mean that*

$$\forall s, \delta . \delta \vdash^D e\ s : c' \Rightarrow \delta \vdash^D s : c$$

The idea now is to define a reduction semantics that defines what the syntactic residual contract after an event e is. To do this we now need a way to decide whether a contract can be concluded successfully. We call this a nullable contract.

2.5.1 Nullability

We characterize semantic nullability by a relation $D \models c$ nullable that is true if for all δ, $\delta \vdash^D \langle\rangle : c$. This is differs from the paper, since the paper says the empty trace should satisfy the contract in some environment, and then states that semantic nullability is independent from the choice of δ. The only difference is in the case where the type of environments is not inhabited for all typing environments, but we will not encounter this case. We will prove in the mechanization that we can construct environments for all typing environments.

We can also characterize nullability syntactically. This can be seen in Figure 2.4. Success is nullable, and then for $+$, just one subcontract should be nullable. A Transfer is of course not nullable, and for the rest of the combinators, all subcontracts should be nullable.

We can prove that the two notions of nullability are equivalent.

Lemma 2.1. *$D \vdash c$ nullable $\Longleftrightarrow D \models c$ nullable*

Proof. For $\Rightarrow$ by induction on the derivation of nullability for c. For $\Leftarrow$ by induction on the trace satisfaction derivation for c. □

$\boxed{D \vdash c\ \text{guarded}}$

$$\frac{}{D \vdash \mathsf{Success}\ \text{guarded}} \qquad \frac{}{D \vdash \mathsf{Failure}\ \text{guarded}}$$

$$\frac{D \vdash c\ \text{guarded}}{D \vdash f(a_1, \ldots, a_n)\ \text{guarded}}\, D(f) = (f[x_1, \ldots, x_n] = c)$$

$$\frac{}{D \vdash \mathsf{Transfer}(A_1, A_2, R, T \mid P).c\ \text{guarded}}$$

$$\frac{D \vdash c\ \text{guarded} \qquad D \vdash c'\ \text{guarded}}{D \vdash c + c'\ \text{guarded}} \qquad \frac{D \vdash c\ \text{guarded} \qquad D \vdash c'\ \text{guarded}}{D \vdash c \parallel c'\ \text{guarded}}$$

$$\frac{D \vdash c\ \text{nullable} \qquad D \vdash c\ \text{guarded} \qquad D \vdash c'\ \text{guarded}}{D \vdash c; c'\ \text{guarded}} \qquad \frac{D \not\vdash c\ \text{nullable} \qquad D \vdash c\ \text{guarded}}{D \vdash c; c'\ \text{guarded}}$$

Figure 2.5: Syntactic guardedness

2.5.2 Guardedness

The goal of residuation is the reduce contracts given events. Unproductive contract templates like

```
letrec f[] = g()
       g[] = f() in f()
```

can be hard to reduce, since a residuation algorithm might loop. To make sure the residuation is possible we require guardedness. Guardedness basically ensures that we do not have contracts that are unproductive, which means that any template application must be guarded by a Transfer. The rules for guardedness can be seen on Figure 2.5. We also need guardedness for template environments.

Definition 2.4. *We say that $D = [f_1 \mapsto f_1[\mathbf{x_1}] = c_1, \ldots, f_n \mapsto f_n[\mathbf{x_n}] = c_n]$ is guarded if it is the case for all $c_i \in \{c_1, \ldots, c_n\}$ that $D \vdash c_i$ guarded.*

It turns out that this is a sufficient condition to ensure that contracts are guarded.

Lemma 2.2. *If D is guarded then for all c, $D \vdash c$ guarded.*

Proof. By induction on c. □

2.5.3 Deterministic reduction by delayed matching

In the sketch for the contract monitoring algorithm we needed to compute $S_n = \{s \mid e\, s \in S_{n-1}\}$ given an event e, but computing the initial set of satisfying traces for a contract is hard. We therefore take a different approach. We do a syntactic reduction of a contract

$\boxed{D \vdash_{\mathrm{D}} c \xrightarrow{e} c'}$ c reduces deterministically to c' by e.

$$\text{D-Success}\,\frac{}{D \vdash_{\mathrm{D}} \mathsf{Success} \xrightarrow{e} \mathsf{Failure}} \qquad \text{D-Failure}\,\frac{}{D \vdash_{\mathrm{D}} \mathsf{Failure} \xrightarrow{e} \mathsf{Failure}}$$

$$\text{D-TransferTrue}\,\frac{\{A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t\} \models P}{D \vdash_{\mathrm{D}} \mathsf{Transfer}(A_1, A_2, R, T \mid P).c \xrightarrow{\mathsf{transfer}(a_1,a_2,r,t)} c[a_1/A_1, a_2/A_2, r/R, t/T]}$$

$$\text{D-TransferFalse}\,\frac{\{A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t\} \not\models P}{D \vdash_{\mathrm{D}} \mathsf{Transfer}(A_1, A_2, R, T \mid P).c \xrightarrow{\mathsf{transfer}(a_1,a_2,r,t)} \mathsf{Failure}}$$

$$\text{D-TApp}\,\frac{\begin{array}{c} D \vdash_{\mathrm{D}} c[v_1/x_1, \ldots, v_n/x_n] \xrightarrow{e} c' \quad D(f) = (f[x_1, \ldots, x_n] = c) \\ \forall i \in \{1, \ldots, n\} : v_i = \mathcal{Q}[\![a_1]\!]^{\emptyset} \end{array}}{D \vdash_{\mathrm{D}} f(a_1, \ldots, a_n) \xrightarrow{e} c'}$$

$$\text{D-Alt}\,\frac{D \vdash_{\mathrm{D}} c \xrightarrow{e} d \quad D \vdash_{\mathrm{D}} c' \xrightarrow{e} d'}{D \vdash_{\mathrm{D}} c + c' \xrightarrow{e} d + d'} \qquad \text{D-Con}\,\frac{D \vdash_{\mathrm{D}} c \xrightarrow{e} d \quad D \vdash_{\mathrm{D}} c' \xrightarrow{e} d'}{D \vdash_{\mathrm{D}} c \parallel c' \xrightarrow{e} c \parallel d' + d \parallel c'}$$

$$\text{D-SeqNull}\,\frac{D \vdash c \text{ nullable} \quad D \vdash_{\mathrm{D}} c \xrightarrow{e} d \quad D \vdash_{\mathrm{D}} c' \xrightarrow{e} d'}{D \vdash_{\mathrm{D}} c; c' \xrightarrow{e} (d; c') + d}$$

$$\text{D-SeqNotNull}\,\frac{D \not\vdash c \text{ nullable} \quad D \vdash_{\mathrm{D}} c \xrightarrow{e} d}{D \vdash_{\mathrm{D}} c; c' \xrightarrow{e} d; c'}$$

Figure 2.6: Deterministic reduction

given an event. Instead of keeping track of a set of traces, we keep track of a residual contract.

The first operational semantics that we will describe is the delayed matching semantics. This semantics delays the commitments to events by committing to all possible transfers in parallel.

The operational semantics rewrites a contract given an event e. It will be rewritings on the form $D \vdash_{\mathrm{D}} c \xrightarrow{e} c'$ meaning that a contract c is rewritten to c' given an event e. In Figure 2.6 we have written the rules. The basic matching rule D-TransferTrue matches a single commitment, but the problem is that it might be matched in multiple subcontracts. To fix this we match the commitment in all subcontracts and then form a syntactic alternative of the possible reductions.

The semantics is fully deterministic, but the consequence of using the rewrite rules will be that we keep track of all possible reductions as a big explicit syntactic alternative. We can then at the end check whether the contract has the behavior of $\mathsf{Success}$ and therefore can be concluded. Note that we reduce closed contracts to closed contracts since we always substitute in values for each $\mathsf{Transfer}$.

We can show that this semantics implements residuation.

Theorem 2.2. *For any c, c', e and D: if $D \vdash_{\mathrm{D}} c \xrightarrow{e} c'$ then $D \vdash e \backslash c = c'$*

And that the residual contract is uniquely determined

Theorem 2.3. *For all c and guarded D, there exists a unique c' such that $D \vdash_{\mathrm{D}} c \xrightarrow{e} c'$ and $D \vdash c'$ guarded.*

2.5.4 Non-deterministic reduction by eager matching

The previous deterministic reduction semantics faithfully implements residuation, but the residual contract is not entirely natural. In accounting it is customary to match events eagerly, and choose the matching subcontract as events arrive. To reflect this in the semantics, we change the deterministic semantics into a non-deterministic semantics where alternatives are represented by rules at the meta-level. This semantics can be seen in Figure 2.7. Meta-level choices are represented by a special reduction $D \vdash_{\mathrm{N}} c \xrightarrow{\tau} c'$ that does not correspond to an actual event, and is used to choose commitments eagerly. We let λ range over both τ and actual events. For instance we have added the rule N-AltL to choose to match the left alternative in a contract $c + c'$. We have also added a transitivity rule N-Trans to be able to make a sequence of τ-reductions along with an actual event reduction to reduce say $c + \mathsf{Transfer}(A_1, A_2, R, T \mid \mathsf{True}).\mathsf{Success}$ to $\mathsf{Success}$ given an event e.

We can show that this non-deterministic reduction semantics is sound. That is if we reduce on an event we remain sound, and if we do a τ-reduction we also remain sound.

Theorem 2.4. *We show soundness for τ-reductions and actual event reductions separately.*

1. *If $D \vdash_{\mathrm{N}} c \xrightarrow{e} c'$ then $D \vdash c' \subseteq e \backslash c$.*

2. *If $D \vdash_{\mathrm{N}} c \xrightarrow{\tau} c'$ then $D \vdash c' \subseteq c$.*

One single reduction is not complete, but if we take all the possible reductions for a contract and put them in a big alternative, then we are complete. We state that with the following theorem

Theorem 2.5. *If $D \vdash_{\mathrm{D}} c \xrightarrow{e} c'$ then there exist contracts $c_1, \ldots, c_n$ for some $n \geq 1$ such that $D \vdash_{\mathrm{N}} c \xrightarrow{e} c_i$ for all $i \ldots n$ and $D \vdash c' \subseteq \Sigma_{i=1}^{n} c_i$.*

2.5.5 Controlled reduction semantics

In [And+06] there is also a third operational semantics, which is also an eager matching semantics, but instead of allowing arbitrary τ-reductions, events carry routing information. The routing information denotes what commitment the event is meant to match.

We will not go into detail about it here, since we are not mechanizing it in Coq. This is because the original formulation in the paper is not complete with respect to the non-deterministic reduction semantics. Not every reduction in the non-deterministic reduction semantics can be mimicked by one in the controlled reduction semantics.

$\boxed{D \vdash_{\mathrm{N}} c \xrightarrow{e} c'}$ c reduces non-deterministically to c' by e.

$$\text{N-Success}\ \frac{}{D \vdash_{\mathrm{N}} \mathsf{Success} \xrightarrow{e} \mathsf{Failure}} \qquad \text{N-Failure}\ \frac{}{D \vdash_{\mathrm{N}} \mathsf{Failure} \xrightarrow{e} \mathsf{Failure}}$$

$$\text{N-TransferTrue}\ \frac{\{A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t\} \models P}{D \vdash_{\mathrm{N}} \mathsf{Transfer}(A_1, A_2, R, T \mid P).c \xrightarrow{\mathsf{transfer}(a_1,a_2,r,t)} c[a_1/A_1, a_2/A_2, r/R, t/T]}$$

$$\text{N-TransferFalse}\ \frac{\{A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t\} \not\models P}{D \vdash_{\mathrm{N}} \mathsf{Transfer}(A_1, A_2, R, T \mid P).c \xrightarrow{\mathsf{transfer}(a_1,a_2,r,t)} \mathsf{Failure}}$$

$$\text{N-TApp}\ \frac{D(f) = (f[x_1, \dots, x_n] = c) \quad \forall i \in \{1, \dots, n\} : v_i = \mathcal{Q}[\![a_i]\!]\emptyset}{D \vdash_{\mathrm{N}} f(a_1, \dots, a_n) \xrightarrow{\tau} c[v_1/x_1, \dots, v_n/x_n]}$$

$$\text{N-AltL}\ \frac{}{D \vdash_{\mathrm{N}} c + c' \xrightarrow{\tau} c} \qquad \text{N-AltR}\ \frac{}{D \vdash_{\mathrm{N}} c + c' \xrightarrow{\tau} c'}$$

$$\text{N-Con1}\ \frac{D \vdash_{\mathrm{N}} c \xrightarrow{\lambda} d}{D \vdash_{\mathrm{N}} c \parallel c' \xrightarrow{\lambda} d \parallel c'} \qquad \text{N-Con2}\ \frac{D \vdash_{\mathrm{N}} c' \xrightarrow{\lambda} d'}{D \vdash_{\mathrm{N}} c \parallel c' \xrightarrow{\lambda} c \parallel d'}$$

$$\text{N-Con3}\ \frac{}{D \vdash_{\mathrm{N}} \mathsf{Success} \parallel c \xrightarrow{\tau} c} \qquad \text{N-Con4}\ \frac{}{D \vdash_{\mathrm{N}} c \parallel \mathsf{Success} \xrightarrow{\tau} c}$$

$$\text{N-Seq1}\ \frac{}{D \vdash_{\mathrm{N}} \mathsf{Success}; c \xrightarrow{\tau} c} \qquad \text{N-Seq2}\ \frac{D \vdash_{\mathrm{N}} c \xrightarrow{\lambda} d}{D \vdash_{\mathrm{N}} c; c' \xrightarrow{\lambda} d; c'}$$

$$\text{N-Trans}\ \frac{D \vdash_{\mathrm{N}} c \xrightarrow{\tau} c' \quad D \vdash_{\mathrm{N}} c' \xrightarrow{e} c''}{D \vdash_{\mathrm{N}} c \xrightarrow{e} c''}$$

Figure 2.7: Non-deterministic reduction

In particular if we can reduce $D \vdash_{\mathrm{N}} c \xrightarrow{e} c'$, then in the non-deterministic semantics we can reduce $D \vdash_{\mathrm{N}} (c_1 + c_2) \parallel c \xrightarrow{e} c_1 \parallel c'$. This is not possible to mimic in the controlled semantics in the paper.

There is currently ongoing work by Murawska on developing a complete reduction semantics with explicit control.

2.6 Resources and agents

When describing CSL we did not describe the base domains, we only required the existence of $\mathcal{A}, \mathcal{R}$ and $\mathcal{T}$. For the analysis of CSL we need to be more specific about agents and resources. We will use the definitions about resources and agents in Section 6.6 and Section 6.7 about participation and fairness analysis.

For agents the story is pretty simple. We only distinguish between two cases. Either we have some (basically) countably infinite set of agents. It could be public keys in a blockchain system or arbitrary user names (strings). This means that we do not have access to the entire universe of possible agents for a transfer. The other case is that we have a fixed set of agents for a particular contract. It could be all registered users in a business application.

For resources the story is a bit more complex. We will now give an example of a resource domain that will fit into the REA ontology. The definitions here are less formal than what is done by Henglein et al. in the POETS paper [Hen+09], and we avoid involving linear algebra.

We define a countable infinite set of resource types X

$$X = \{\text{DKK}, \text{iPhone}, \text{Vesterbrogade 142, st. th.}, \ldots\}$$

to denote scarce resources, they might be unique like in the case of "Vesterbrogade 142, st. th", or they might be fungible like "DKK" or "USD" and so on. We can now write resources $\mathcal{R}$ as a finite map from resource types to real numbers denoting how much of a certain resource type goes into a resource. Implicitly resources types not defined in the finite map will map to 0.

$$\mathcal{R} = X \xrightarrow{\text{fin}} \mathbb{R}$$

For easier notation we write a resource as a sum of scaled resource types, for instance the resource consisting of 1.5 litres of milk and 15 iPhones can be written as:

$$r = 1.5 \cdot \text{litres of milk} + 15 \cdot \text{iPhone}.$$

We can define addition and subtraction pretty easily by just performing it element-wise. We can also scale resources by a constant. We can describe ownership of resources O as a finite mapping from agents to resources denoting which resources which agents own

$$O = \mathcal{A} \xrightarrow{\text{fin}} \mathcal{R}.$$

A transfer of resources is a change of ownership such that the amount of resources in total is preserved.

$$T = \mathcal{A} \xrightarrow{\text{fin}} \mathcal{R}$$

For an event $e = \mathsf{transfer}(a_1, a_2, r, t)$, the associated transfer is

$$t_e = \begin{cases} \{a_1 \mapsto -r, a_2 \mapsto r\} & \text{if } a_1 \neq a_2 \\ \emptyset & \text{otherwise} \end{cases}$$

For more general transfers $t' \in T$ where multiple parties can be involved it must also be the case that

$$\sum_{a \in \mathcal{A}} t'(a) = 0$$

Which is trivially true for the simple two-way transfer.

We can also denote the effect of an entire trace with a function $E : Tr \to T$ that computes the combined transfer for all the events.

$$E(\langle e_1, \ldots, e_n \rangle) = \sum_{i=1}^{n} t_{e_i}$$

Example 2.1. *Consider the event trace*

$$s = \langle \textsf{\textit{transfer}}(a, b, 10 \cdot \textit{DKK}, t_1), \textsf{\textit{transfer}}(b, a, 1 \cdot \textit{litres of milk}, t_2) \rangle$$

The effect of the trace is

$$E(s) = \{a \mapsto 1 \cdot \textit{litres of milk} - 10 \cdot \textit{DKK}, b \mapsto 10 \cdot \textit{DKK} - 1 \cdot \textit{litres of milk}\}$$

We can check that the amount of resources is preserved, and we can convince ourselves that this holds in general.

This concludes the description of CSL, and we will quickly introduce the background needed for mechanization of CSL.

Chapter 3

Proofs and programs

In the last chapter we gave some background on CSL and defined the semantics and some meta-theorems on paper. In this chapter we will give the background for using programming languages with higher-order type systems for formalizing mathematics and programming language theory. It will not be a complete introduction to type theory, but will mainly give an intuition into how we can encode mathematical statements using the Coq proof assistant. Readers that already have experience with Coq can skip this section.

3.1 Propositions as types

The notion of propositions as types describes a correspondence between a constructive logic and a programming language. The idea is that for each proposition A in the logic, there is a corresponding type $|A|$ in the programming language. Determining provability of A corresponds to the checking that type $|A|$ is inhabited (there exists a program of the type $|A|$). Proving A corresponds to constructing a term t of type $|A|$. Checking the proof of A corresponds to checking that t has type $|A|$.

In propositional logic we can informally describe the correspondence as a translation from propositions to types in a functional language with simple types:

$$\begin{aligned} [\![A \Rightarrow B]\!] &= [\![A]\!] \rightarrow [\![B]\!] \\ [\![A \wedge B]\!] &= [\![A]\!] \times [\![B]\!] \\ [\![A \vee B]\!] &= [\![A]\!] + [\![B]\!] \\ [\![\bot]\!] &= \emptyset \end{aligned}$$

We write $\neg A$ is an abbreviation for $A \Rightarrow \bot$ to complete the definition of propositional logic. In the translation, $+$ is the disjoint union defined as

$$P + Q = \{(\texttt{left}, p) \mid p \in P\} \cup \{(\texttt{right}, q) \mid q \in Q\}.$$

$\emptyset$ is an empty type that has no elements.

Now to prove a statement like $A \Rightarrow A \vee B$, we have to construct a program with the type $|A| \rightarrow |A| + |B|$. This could be the program $\lambda x : |A|.(\texttt{left}, x)$. Notice the similarity between the lambda abstraction and the implication introduction rule in propositional logic.

To extend the correspondence to predicate logic we have to give a translation for quantifiers. Quantifiers bind variables in propositions, so in the propositions as types interpretations we have to include binding structures in types. This gives rise to dependent types where types depend on values. We introduce the Π-type to model universal quantification and the Σ-type to model existential quantification in the propositions as types interpretation. This was also the original motivation when Howard and De Bruijn introduced dependent types in the 60's [BD09].

Intuitively the universal quantifier $\forall x \in X.A\ x$ joins a family of propositions based on the set X, so we can view it as a big conjunction

$$\bigwedge_{x \in X} A\ x.$$

In dependent type theory we describe this type as the Π-type $\Pi x : X.A\ x$. This type will allow us to describe a family of types indexed by a set X. In a program this might be the set of n-tuples described with the Π-type

$$NTup = \Pi n : \mathbb{N}. \overbrace{\mathbb{N} \times \cdots \times \mathbb{N}}^{n}.$$

Now $(42, 7, 9)$ is a member of the type $NTup$ 3. The Π-type is essentially a type-level function.

If we look at the propositions as types translation for the conjunction, then the big conjunction can be though of as a big product, and elements of the type are members of this product.

For the existential quantifier $\exists x \in X.A\ x$ we can dually view it as a big disjunction

$$\bigvee_{x \in X} A\ x$$

So for this we introduce the Σ-type $\Sigma x : X.A\ x$. By the proposition as types translation, we can look at this as a big disjoint union. So elements of this Π-type are tuples with the first element choosing which element of $x \in X$ we came from, and the second element is a proof of A that depends on x. If we look at the statement $\exists x \in \mathbb{N}.x \geq 10$ then the proposition as types interpretation is $\Sigma x : \mathbb{N}.x \geq 10$. An element of this type is a tuple $(x, P\ x)$ where x is an element greater than 10, and $P\ x$ is a proof that x is greater than 10.

Now we can complete the propositions as types interpretation for predicate logic:

$$[\![\forall x \in X.A\ x]\!] = \Pi x : |X|.[\![A\ x]\!]$$
$$[\![\exists x \in X.A\ x]\!] = \Sigma x : |X|.[\![A\ x]\!]$$

We will now look at the proof assistant Coq, which is a practical implementation of a dependently typing programming language where this view of propositions as types is used for encoding propositions and proofs of them.

3.2 The Coq proof assistant

The Coq proof assistant is a piece of software that can verify proofs of mathematical theorems. At its core is the language Gallina which is based on the higher-order type theory, the Calculus of Inductive Constructions. By the proofs as programs interpretation we can use it both for writing programs by viewing it at a typed programming language and using it for doing constructive predicate logic.

On top of Gallina is a tactic language, Ltac, that lets us prove theorems interactively, implement proof search and decision algorithms and much more. This is the feature that makes Coq a good option for verifying large scale proofs about both mathematics and software. A lot of the tedious work can be automated.

When we write Coq terms directly we write := after the type of the term and give the term like this:

```
Definition important_def : type := implementation.
```

When we construct the term interactively using Ltac we give a sequence of proof steps:

```
Definition also_important_def : type.
  tactic1. tactic2.
Defined.
```

To make it clear when we are writing proofs, we also can use keywords `Lemma` and `Theorem` to make it clear that we are doing a proof and not constructing a program:

```
Theorem important_proof : proposition_we_need_to_prove.
Proof. tactic1.
       tactic2. Qed.
```

And instead of the type, we write the proposition we want to prove. There are small distinctions between proofs and programs in Coq, but we will not explore it in this thesis, since we are interested in practical proof developments.

Now as a small tutorial to Coq, we describe the implementation of a library for intervals. This example will show the use of Coq to formalize some mathematics that later will be used in the context of an analysis of CSL. This will be a nice example of the flexibility of the Coq system to combine both programming language logics and more regular mathematics. For a brief introduction to Coq, a recommended reading is the notes “Coq in a Hurry” by Bertot [Ber06]. A good complete textbook on Coq is “Coq’Art” by Bertot and Casteran [BC04].

3.3 An interval abstraction

We define an interval on the integers as the following set:

$$\mathcal{I}_{\mathbb{Z}} = \{\bot\} \cup \{[x_1, x_2] \mid x_1 \leq x_2, x_1 \in \mathbb{Z} \cup \{-\infty\}, x_2 \in \mathbb{Z} \cup \{\infty\}\}$$

We start by identifying the 5 different types of intervals. For $a, b \in \mathbb{Z}$ we have the following statements about intervals:

1. The empty interval, $i = \bot$ is an interval.
2. The interval that is bounded to the right $i = [-\infty, b]$ is a valid interval.
3. The interval that is bounded to the left $i = [a, \infty]$ is a valid interval.
4. The fully bounded interval $i = [a, b]$ with $a < b$ is a valid interval.
5. The entire number line $i = [-\infty, \infty]$ is a valid interval
6. Nothing else is an interval.

We want to encode valid intervals, and we do that with an inductive datatype. All except case 4 are easy to encode in Coq. They either take one or zero arguments and every way to construct them results in a valid interval. In case 4 there are different strategies for encoding that $a < b$. We choose to make the intervals well-typed by construction. For this we can use the Σ-type in Coq. The final definition of valid intervals is:

```
Inductive interval : Set :=
| EmptyInterval : interval
| Interval : { ' (x, y) | Z.le x y } → interval
| Below : Z → interval
| Above : Z → interval
| FullInterval : interval.
```

Here `{ ' (x, y) | Z.le x y }` is the sigma type $\Sigma(x, y) : \mathbb{Z}^2 . x \leq y$, where members are pairs of integers where the first coordinate is smaller than the second. This is exactly what we need for a valid interval. The definitions for the inductive type are parameterized by the type of their constructor. For instance `EmptyInterval` takes no arguments and `Interval` takes a member of the subset type as an argument.

To construct a singleton interval we use the `exist` constructor that constructs an element of a Σ-type. For the equality proof we already have a proof that $x \leq x$ in the standard library:

```
Z.le_refl : ∀ n : Z, (n <= n)%Z
```

Here we see the proofs as programs interpretation in action. The program `Z.le_refl` represents a proof that $x \leq x$. We can also view it as function with a Π-type. In Coq we write Π-types with the $\forall$ quantifier or with a named argument in the type. To construct a singleton interval we do:

```
Definition singleton_interval (x : Z) : interval := Interval (exist _ (x, x) Z.le_refl x)).
```

There are multiple ways to implement this function. Consider another option:

```
Definition singleton_interval' (x : Z) : interval :=
  Interval (exist _ (x, x)
    Z.ge_le x x (Z.le_ge x x (Z.le_refl x)).
```

Where the proof now rewrites to $x \geq x$ and back. We now try to prove that intervals contructed with the two function are equal. We want this to be the case, since the intervals are supposed to be equal by having equal bounds:

```
Example singleton_interval_correct : singleton_interval 0 = singleton_interval' 0.
Proof. unfold singleton_interval, singleton_interval'.
       f_equal. f_equal.
Admitted.
```

We cannot prove this, since we need to prove that

```
  Z.le_refl 0 = Z.ge_le 0 0 (Z.le_ge 0 0 (Z.le_refl 0))
```

which they are not (they are different proof terms). We need some other notion of equality of intervals. We want bounded intervals to be equal if the first coordinates of the Σ-types are equal. We do this with the `proj1_sig` function that projects the witness out of a Σ-type:

```
Definition eq_interval (a b : interval) : Prop :=
  match a, b with
  | Interval i1, Interval i2 ⇒ proj1_sig i1 = proj1_sig i2
  | _, _ ⇒ a = b
  end.
```

We define membership in intervals as you would expect, but again we have to project out the bounds in the case of the fully bounded interval. Also inclusion is defined as an implication, and not by comparing bounds:

```
Definition In_interval (z : Z) (i : interval) : Prop :=
  match i with
  | EmptyInterval ⇒ False
  | Interval i' ⇒ let (a, b) := proj1_sig i' in (a <= z)%Z ∧ (z <= b)%Z
  | Below b ⇒ Z.le z b
  | Above a ⇒ Z.le a z
  | FullInterval ⇒ True
  end.
Definition Incl_interval (i1 i2 : interval) : Prop :=
  ∀ z, In_interval z i1 → In_interval z i2.
```

With both definitions we can prove that equality of intervals actually encodes equality of intervals, namely that they have the same elements.

```
Lemma In_interval_eq : ∀ (i1 i2 : interval),
    eq_interval i1 i2 → ∀ z, In_interval z i1 ↔ In_interval z i2.
```

The nice thing about the dependently typed intervals is that whenever we have a function on intervals for instance join,

```
Definition join_interval (i1 i2 : interval) : interval.
```

Then we know that the output of this function is a valid interval with upper bounds actually being upper bounds. We then explicitly prove that join has the properties that we want. For instance that the empty interval is an identity element

```
Lemma join_interval_l_id : ∀ i, eq_interval (join_interval EmptyInterval i) i.
```

One crucial lemma we want to show is the monotonicity of join, $i_1 \subseteq i_2 \wedge i_3 \subseteq i_4 \Rightarrow i_1 \cup i_3 \subseteq i_2 \cup i_4$.

```
Lemma join_interval_monotone : ∀ i1 i2 i3 i4,
    Incl_interval i1 i2 ∧ Incl_interval i3 i4 →
    Incl_interval (join_interval i1 i3) (join_interval i2 i4).
```

To prove a statement like this we need a different characterization of interval inclusion that relates inclusion to lower and upper bounds of intervals. For instance for two bounded intervals, inclusion implies an order of the lower and upper bounds.

```
Lemma Incl_interval_bounds_ii : ∀ s1 s2,
    Incl_interval (Interval s1) (Interval s2) →
    (lb s2 <= lb s1 ∧ ub s1 <= ub s2)%Z.
```

We prove similar lemmas for open intervals. Proofs of lemmas like these require a lot of reasoning about linear arithmetic. In Coq there is a tactic `lia` that decides a large part of linear arithmetic, and it is very useful. We define a tactic that automatically proves a lot of statements about intervals.

```
Ltac interval_killer :=
  repeat match goal with
        | [ s : {'(_, _) : Z * Z | (_ <= _)%Z } |- _ ] ⇒ destruct s as [x ?]; destruct x as [? ?]
        end; simpl in *; f_equal; auto; try lia.
```

The tactic basically destructs the bounded intervals and gives us all the inequality proofs. It then tries to prove the goal automatically. For instance we could define a lemma that states something about inclusion when adding a singleton interval:

```
Lemma In_interval_plus_singleton : ∀ z1 z2 i,
    In_interval z1 (plus_interval (singleton_interval z2) i) ↔
    In_interval (z1 - z2) i.
Proof.
  destruct i; interval_killer.
Qed.
```

Which we now can prove with automation. This small example shows that Coq is also a good choice for mechanizations of other things than just logics about programming languages. The entire implementation can be found in `CSL/Analysis/Interval.v`.

We will also briefly introduce two features of Coq that are not so common in Coq developments that we use in our development: Type classes and co-inductive types.

3.3.1 Type classes

Type classes in Coq are based on dependent records, which is a generalization of regular records such that types of fields can depend on previous values defined in the record. It is commonly used to structure, say, data types with proofs about them. For instance we can package a natural number with a proof that it is bigger than 10 inside a dependent record where the proof depends on the value.

```
Record Stuff : Type := mkStuff { thing : nat; proof : thing >= 10 }.
```

We can construct an element of `Stuff` for the natural number 10

```
Definition ten := mkStuff (le_refl 10).
```

By providing a proof that $10 >= 10$, which is a proof from the standard library `le_refl`, that we can instantiate with 10. Note that Coq makes `thing` implicit, since it can be determined from `proof`.

Type classes share the syntax with dependent records both for class declarations and instance specifications. As an example we can specify the class of injective functions from `A` to `B` as the type class

```
Class Injective A B := {
  f : A → B;
  f_injective : ∀ a b, f a = f b → a = b
}.
```

Now we give the instances with the `Instance` keyword. We can compose type classes, and derive instances from other instances. For instance we can derive that the composition of two injective functions is injective.

```
Instance compInjective A B C `(Injective A B) `(Injective B C) : Injective A C :=
  {
    f a := f (f a)
  }.
intros; apply f_injective; apply f_injective; auto.
Defined.
```

Note that if we do not specify the member in the record, we can give a proof script that constructs the term after the fact. Also the inner `f` is from the parameter `Injective A B` and the outer `f` is from the parameter `Injective B C`. We will use the type classes heavily when defining generic analyses for CSL in a later section.

3.3.2 Co-inductive types

In the development we are also going to use co-inductive types which makes it possible to define infinite objects and prove properties about them.

For inductive types, elements are obtained from finite applications of the constructors in the definition. For co-inductive types, elements are obtained from both finite and infinite applications of the constructors in the definition. This means that if we need to have an infinite branching structure in sub-terms, then we have to use co-inductive types.

We noted before that propositions can be seen as types, and proofs can be seen as inhabitants of these types. Co-inductive types of sort Prop describe co-inductive predicates, and proofs of these statements can be infinite proof terms. Constructing these proofs in Coq is a little tricky, and not as widely supported as inductive proofs.

As an example of a co-inductive type we implement infinite trees:

```
CoInductive ltree (A : Set) : Set :=
| LLeaf : ltree A
| LBin : A → ltree A → ltree A → ltree A.
```

We can describe the tree that infinitely branches to the right with zeros on all nodes by the co-recursive function:

```
CoFixpoint right_zero_tree : ltree nat := LBin 0 LLeaf right_zero_tree.
```

So trees can be finite or infinite, and some branches can be finite while other branches are infinite.

One major difficulty with co-inductive types is that equality is inductively defined in Coq, and therefore too strong to be used to reason about elements of co-inductive types. Usually we then define another notion of equivalence, bi-similarity. The idea is for say, infinite trees, that two trees are bi-similar if the root is equal, and all the sub-trees are bi-similar. We describe this as a co-inductive predicate:

```
CoInductive ltree_bisim A : ltree A → ltree A → Prop :=
| LLeafBisim : ltree_bisim LLeaf LLeaf
| LBinBisim : ∀ a t1 t1' t2 t2',
    ltree_bisim t1 t1' → ltree_bisim t2 t2' → ltree_bisim (LBin a t1 t2) (LBin a t1' t2').
```

So if we want to show that two transformations `f g : ltree A → ltree A` on the same tree are bi-similar we show

```
Theorem fg_bisim A : ∀ (t : ltree A), ltree_bisim (f t) (g t).
```

For instance we can define the map function on infinite trees (similarly to how you would define it for finite trees) that maps a function on each element:

```
CoFixpoint ltree_map (A B : Set) (f : A → B) (t : ltree A) : ltree B :=
  match t with
  | LLeaf ⇒ LLeaf
  | LBin v t1 t2 ⇒ LBin (f v) (ltree_map f t1) (ltree_map f t2)
  end.
```

The only restriction on writing co-recursive functions in Coq is that every co-recursive call needs to be guarded by a constructor. In the previous function, calls to `ltree_map` are guarded by `LBin`.

Now we want to prove that mapping with two functions composed together is the same as mapping them individually. We cannot do it with equality, but with bi-simulation we can say that no matter how we explore the tree, then the nodes will always be similar.

```
Theorem ltree_map_comp_bisim : ∀ (A B C : Set) (f : B → C) (g : A → B) (t : ltree A),
  ltree_bisim (ltree_map (f ∘ g) t) (ltree_map f (ltree_map g t)).
Proof.
```

```
  intros A B C f g.
  cofix H.
  (* Proof steps here *)
Qed.
```

And for the proof by co-induction we also have the guardedness condition. All uses of the co-inductive hypothesis

```
H : ∀ t : ltree A, ltree_bisim (ltree_map (f ○ g) t) (ltree_map f (ltree_map g t))
```

has to be guarded by a constructor.

We will now describe the mechanization of CSL in Coq.

Chapter 4

Mechanization of CSL in Coq

In this chapter we will describe our mechanization of CSL. This includes the encoding of the syntax and semantics, and all the meta-theorems from [And+06] except the control semantics. We will use an intrinsic approach to mechanization which means that the terms that we construct will be well-typed by construction and we will therefore have no explicit typing rules. This encoding will rely heavily on the dependently typed programming capabilities of the Coq proof assistant. An overview of the source code can be found in Appendix A.

4.1 Bindings

When one has to mechanize any serious piece of programming language theory one has to make a choice of how to represent binding structures. They are usually relatively easy to work with on paper, but are notoriously hard to deal with in proof assistants [AZW09].

In this development we are going to use dependently typed De Bruijn indices as used by Benton et al. [Ben+11] and further explored by Chlipala [Chl14] to make variables well-typed by construction and therefore fit in our intrinsically typed approach.

A De Bruijn index is a natural number representing an occurrence of a variable, and the value denotes the number of binders in scope between the occurrence and the corresponding binder. For CSL we have binding structures in the case of Transfer and template declarations. Take for instance the following contract specification

$$\mathsf{letrec}\ f[x,y] = \mathsf{Transfer}(A_1, A_2, R, T \mid A_1 = \text{"}alice\text{"} \land R = \text{iPhone}).f(x,y)\ \mathsf{in}\ f(\text{2019-09-02}, \mathsf{True}) \quad (4.1)$$

With De Bruijn indices we could represent this as

$$\mathsf{letrec}\ f[2] = \mathsf{Transfer}(4 \mid x_3 = \text{"}alice\text{"} \land x_5 = \text{iPhone}).f(x_1,x_2)\ \mathsf{in}\ f(\text{2019-09-02}, \mathsf{True}) \quad (4.2)$$

Where we write $f[2]$ to denote that f takes two arguments and x_n to denote a variable with De Bruijn index n. One major advantage of using De Bruijn indices is that we do not have to worry about α-equivalence. That is defining equality of terms up to renaming of variables. With De Bruijn indices there is a canonical encoding for a Transfer and a template declaration.

Now to make the indices dependently typed in Coq we define a De Bruijn index as a membership predicate on a list:

```
Inductive member A x : list A → Type :=
| HFirst : ∀ ls, member x (x :: ls)
| HNext : ∀ x' ls, member x ls → member x (x' :: ls).
```

Which can be used to prove that a certain type is a member of a typing environment. For instance if we have a typing environment for the previous contract

```
Example Δ : list ty := [Timestamp; Bool; Agent, Agent; Resource, Timestamp].
```

then the variable x_3 will be encoded as

```
Example x3 : member Agent Δ := HNext (HNext HFirst).
```

Representing the fact that x3 is a variable with De Bruijn index 3 and type Agent.

For the actual environment we use a heterogeneous list. It will be a dependent type indexed by a typing environment list A and a denotation function B that denotes each type in the typing environment.

```
Inductive hlist A (B : A → Type) : list A → Type :=
| HNil : hlist B nil
| HCons : ∀ x ls, B x → hlist ls → hlist (x :: ls).
```

In this definition B makes the hlist very generic. It makes it possible to parameterize the list with a type-level function that defines the types of elements in the list. This will be useful for reusing the typing environment for different objects in our mechanization. In addition to variable bindings, we are using it for template environments, arguments to templates and for abstract environments for the analyses in a later chapter.

To retrieve elements from the heterogeneous list we have a function

```
hget : ∀ A (B : A → Type) x ls, hlist B ls → member x ls → B x.
```

Which, given a De Bruijn index, will retrieve an object of the correct type. If we take an actual environment with a type level function tyDenote : ty → A then for an actual environment δ : hlist tyDenote Δ we can retrieve the value of type A with hget δ x3

There are also other functions for hlists for appending, updating and destructing which can be found in CSL/HList.v. In the next sections we will use the hlist for all our environments and the member predicate for all our bindings.

4.2 Expressions

In spirit of the original paper we could have parameterized the mechanization by an expression language making the development very generic, but for ease of mechanization we have fixed the syntax of expressions to be the following

```
Inductive exp Δ : ty → Set :=
| Var : ∀ τ, member τ Δ → exp Δ τ
| Lit : ∀ τ, tyDenote τ → exp Δ τ
| Binop : ∀ τ1 τ2, binop τ1 τ2 → exp Δ τ1 → exp Δ τ1 → exp Δ τ2
| Unop : ∀ τ1 τ2, unop τ1 τ2 → exp Δ τ1 → exp Δ τ2.
```

With binop and unop being the usual set of binary and unary operators on booleans and integers. Note that the expression language is intrinsically typed with a type τ and parameterized with a typing environment Δ for free variables. Variables are defined using the member predicate from before, and a correctly typed De Bruijn index will result in a correctly typed expression.

To denote types of expressions we write a type level function. For our simple mechanization we denote agents by strings, resources and timestamps by natural numbers and bools by bools. In this way we are not including any complicated domains for resources and timestamps. We can think of the number denoting a resource as the value, or an index into the universe of resources. The timestamp can be thought of as the time difference since some epoch.

```
Fixpoint tyDenote (t : ty) : Set := match t with
  | Agent ⇒ string | Resource ⇒ nat | Timestamp ⇒ nat | Bool ⇒ bool end.
```

For mappings from variables to values, we use an hlist for the environment.

```
Definition env Δ := hlist tyDenote Δ.
```

The argument tyDenote encodes that elements of the hlist are values corresponding to the types of Δ. To denote expressions with values we write a simple recursive interpreter.

```
Fixpoint expDenote Δ τ (e : exp Δ τ) (δ : env Δ) : tyDenote τ.
```

This demonstrates one of the strengths of using intrinsically typed syntax. We have that the interpreter for expressions is inherently type preserving by construction. The only thing we prove for now is the decidability of expression evaluation.

```
Lemma expDenote_dec : ∀ Δ τ (e : exp Δ τ) (δ : env Δ) (v : tyDenote τ),
  {expDenote e δ = v} + {expDenote e δ <> v}.
```

Which means that for any value, either an expression evaluates to it, or it does not. We can now describe how we encoded the syntax of CSL in Coq.

4.3 Syntax

We write the syntax of CSL as an inductive type indexed by a template environment Γ and the a typing environment Δ.

```
Inductive contract Γ Δ : Set :=
| success : contract Γ Δ
| failure : contract Γ Δ
| tapp : ∀ ts, arg_env ts Δ → member ts Γ → contract Γ Δ
| transfer : exp (new_var Δ) Bool → contract Γ (new_var Δ) → contract Γ Δ
| alternate : contract Γ Δ → contract Γ Δ → contract Γ Δ
| sequence : contract Γ Δ → contract Γ Δ → contract Γ Δ
| concurrent : contract Γ Δ → contract Γ Δ → contract Γ Δ.
```

The interesting definitions are the ones for template applications and transfers. When applying a contract template, we have a list of expressions with different types. We use a hlist indexed with the type signature of the template to store expressions with those types.

```
Definition arg_env ts Δ:= hlist (exp Δ) ts.
```

We then use De Bruijn indices to specify which function template from the global template environment Γ we are instantiating. In the case of transfers we extend the global environment with 4 new variables that are bound in both the predicate and the subcontract. The definition of new_var is

```
Definition new_var Δ := Agent :: Agent :: Resource :: Timestamp :: Δ.
```

Note that with this definition of the syntax we have faithfully encoded the typing rules of Figure 2.2. This is what makes the encoding intrinsically typed, since we do not need an explicit typing judgment. All terms are well-typed by construction.

We also define a function that evaluates all expressions passed as arguments to a template for use in the semantics.

```
Fixpoint arg_envDenote ts Δ (δ : env Δ) (ae : arg_env ts Δ) : env ts.
```

Finally we use the notation feature of Coq where we can add custom syntax to make contracts easier to write

```
Notation "A ;; B" := (sequence A B) (at level 80, right associativity).
Notation "A :+: B" := (alternate A B) (at level 79, left associativity).
Notation "A :||: B" := (concurrent A B) (at level 78, left associativity).
```

4.4 Contract satisfaction

Before mechanizing the contract satisfaction relation we start by providing denotations for events and traces. They are pretty basic, an event $\textsf{transfer}(a_1, a_2, r, t)$ is just a simple inductive data type

```
Inductive event := Event : Agent → Agent → Resource → Timestamp → event.
```

and traces can simply be denoted as finite lists of events

```
Definition trace := list event.
```

The environment of templates D is a `hlist` with contracts.

```
Definition template_env Γ := hlist (contract Γ) Γ.
```

The definition partially applies the contract type to the template typing environment Γ fixing it for all template bodies. The last parameter to `hlist` specifies the the argument types to each contract template. This definition makes the templates mutually recursive and provides argument types for all contract templates in one single definition. We do not encode the top level contract declaration *td*, and we assume that D has already been constructed. Therefore we only encode the judgment $\delta \vdash^D s : c$.

We translate the contract satisfaction judgment directly into Coq by encoding the rules as an inductive datatype.

```
Inductive csat : ∀Γ Δ, env Δ → template_env Γ → trace → contract Γ Δ → Prop.
```

We are now forced to formalize what it means to interleave and append two traces. We define this using two relations `interleave` and `appends`. They are defined as you would expect, and can be found in the accompanying code. Most rules are as you would expect, but we show the rules for template application and transfers:

```
| AppF : ∀ts Γ Δ δ (D : template_env Γ) (ae : arg_env ts Δ) f t,
   csat (arg_envDenote δ ae) D t (hget D f) → csat δ D t (tapp ae f)
| Transfer : ∀Γ Δ (D : template_env Γ) δ e (p : exp (new_var Δ) Bool) t c,
   csat (addEvent e δ) D t c →
   expDenote p (addEvent e δ) = true →
   csat δ D (e :: t) (transfer p c)
```

For transfers `expDenote p δ' = true` encodes $\delta \models P$ when `p` encodes P and δ' encodes δ. For template applications we see that the De Bruijn indexing also works nicely for templates. `f` is a De Bruijn index proving that there is a template in `D` with the correct type. Everything in these definitions fits nicely together, and it is clearly readable without any auxiliary typing proofs.

4.5 Denotational semantics

In Section 2.4 we hinted the existence of a denotational semantics for CSL, and in this section we will try to encode a form of denotational semantics in Coq.

The denotations of contract specifications are sets of traces, which we need to encode in Coq. The sets of traces are interesting since they are possibly infinite. We choose to encode them using characteristic functions. Given a set $A \subseteq \mathcal{U}$, then a characteristic function

$$f_A : \mathcal{U} \to \{\texttt{true}, \texttt{false}\}$$

is defined such $f_A(x) = \texttt{true}$ if and only if $x \in A$. Sets as characteristic functions in Coq are provided by `Uniset` from the standard library[1]. They are simply function from a set `A` with decidable equality to `bool`.

```
Inductive uniset : Set :=
  Charac : (A → bool) → uniset.
```

Using `uniset` we can then say that the denotation of a contract is

```
Definition contract_denotation := uniset trace.
```

There is an interesting consequence of using characteristic functions to define the denotational semantics of CSL. Whenever we denote a contract we build a function from traces to booleans. Whenever we denote, say $c_1 \parallel c_2$ we have to construct the function that given a trace t computes all possible interleavings and then returns true if and only if for at least one interleaving $(t_1, t_2) \rightsquigarrow t$ it is the case that t_1 is in the denotation of c_1 and t_1 is in the denotation of c_2.

To construct all of these pairs we implement functions that compute them:

```
Fixpoint interleavings (T : Set) (l : list T) : list (list T * list T).
Fixpoint appendings (T : Set) (l : list T) : list (list T * list T).
```

And we prove that they compute the same interleavings and appendings that we can construct with the relations used in the trace satisfaction relation. For instance we show the following lemma for interleavings:

```
Lemma interleavings_interleave : ∀ (T : Set) (t t1 t2 : list T),
   List.In (t1, t2) (interleavings t) ↔ interleave t1 t2 t.
```

Now we can try to write a denotation function that maps contract specifications to sets of traces:

```
Fixpoint contractDenote Γ Δ (c : contract Γ Δ) (D : template_env Γ) (δ : env Δ)
  : uniset trace :=
| success _ _ ⇒ Singleton _ _ nil
| failure _ _ ⇒ Emptyset _
| (c1 ;; c2) ⇒ Charac (fun t ⇒
                  let tc1 := contractDenote n c1 D δ in
                  let tc2 := contractDenote n c2 D δ in
                  existsb (fun pt ⇒
                             match pt with
                             | (t1, t2) ⇒ charac tc1 t1 && charac tc2 t2
                             end) (appendings t))
| fapp ae f ⇒ contractDenote (hget D f) D (argEnvDenote δ ae)
| _ ⇒ _ (* Rest of cases ignored for now *)
end.
```

But we have a problem. Coq rejects this definition since the it is not structurally recursive over any of its arguments. Coq cannot prove that `c` gets smaller, and in that way ensure termination. This is also correct, since any `c′` that we look up in the function environment may be larger.

[1] https://coq.inria.fr/stdlib/Coq.Sets.Uniset.html

To solve this problem we now need a way of modelling possible non-termination in Coq. We are going to follow the approach taken in [Chl14] where Chlipala uses to notion of approximation levels among computation results to model non-termination. We define an approximation level as a natural number n that corresponds to the depth of the recursion on the syntax tree.

Chlipala also lifts the domain that he is working with to include $\bot$ to denote possible non-termination. In our case with sets of traces we have a complete lattice partially ordered by set inclusion, and we do not have to lift the domain with $\bot$ to denote possible non-termination since $\bot = \emptyset$ is already present. This feature of our domain makes things a lot simpler for us, but also makes it less natural since we cannot distinguish between non-terminating contracts and contracts with no satisfying trace.

Now to guarantee termination of our interpreter we provide the approximation level as an argument which we then decrease by 1 in each recursive call. If the approximation level is zero, we indicate possible non-termination:

```
Fixpoint contractDenote n Γ Δ (c : contract Γ Δ) (D : template_env Γ) (δ : env Δ)
  : uniset trace := match n with
  | O ⇒ Emptyset _
  | S n ⇒ match c with
    | success _ _ ⇒ Singleton _ _ []
    | failure _ _ ⇒ Emptyset _
    | tapp ae f ⇒ contractDenote n (hget D f) D (arg_envDenote δ ae)
    (* And so on *)
    end end.
```

Coq accepts this definition, but now we have to convince ourselves that this encoding captures the meaning of the denotational semantics. To do this we try to prove that our interpreter is continuous in the sense of domain theory. That is running the interpreter at a higher approximation level will only give us a larger set of valid traces for the contract:

```
Lemma contractDenoteContinuous :
  ∀ n Γ Δ (c : contract Γ Δ) (D : template_env Γ) (δ : environment Δ),
    incl (contractDenote n c D δ) (contractDenote (S n) c D δ).
```

which is a corollary from a stronger statement, namely that it holds for any higher approximation level.

We now have to define what is means that a contract denotes a trace set. There is one obvious candidate where for a trace to be included in the denotation of a contract, there has to exist an approximation level such that the trace is included in the denotation when run at that approximation level:

```
Definition denotes' Γ Δ (δ : env Δ) (D : template_env Γ) c t :=
  ∃ n, In (contractDenote n c D Δ) t.
```

It turns out that we need a stronger definition in some proofs. Intuitively we can also say that when we reach some approximation level where a trace is in the denotation, then no matter now much we increase the approximation level, the trace will still be included in the denotation of the contract.

```
Definition denotes Γ Δ (δ : env Δ) (D : template_env Γ) c t :=
  ∃ n, ∀ m, n ≤ m → In (contractDenote m c D δ) t.
```

It turns out that the definitions are the equivalent

```
Lemma denotes_denotes'_equiv: ∀ Γ Δ (δ : env Δ) (D : template_env Γ) c t,
    denotes δ D c t ↔ denotes' δ D c t.
```

Which again follows from continuity. This result is nice since we can use the most appropriate definition for proving the property at hand. We now have to convince ourselves that this definitional interpreter is correct with respect to the trace satisfaction relation.

4.5.1 Equivalence with satisfaction relation

Where Theorem 2.1 relate the trace satisfaction relation and the denotational semantics, we prove the corresponding statement with respect to our definitional interpreter.

```
Theorem csat_denotes : ∀ Γ Δ (δ : env Δ) (D : template_env Γ) t c,
    csat δ D t c ↔ denotes δ D c t.
```

Where the left-to-right direction is proven by induction on the satisfaction derivation, and the right-to-left direction is proven by induction on the approximation level and then by case analysis of c. It is not possible to prove this on the syntax of c since the interpreter is not structurally recursive, and therefore the induction would not go through in the case of template application.

This completes the mechanization of the trace satisfaction semantics and the denotational semantics. Before we can mechanize the reduction semantics, we need to define substitution for contract specifications.

4.6 Substitution

In the style of Benton et al. [Ben+11], we define a typed substitution as a map from variables to expressions.

```
Definition Sub Δ Δ' := ∀ τ, member τ Δ → exp Δ' τ.
```

So all variables in Δ should map to an expression typed in Δ'. We also provide a denotation of a substitution as a transformation from one environment to another:

```
Fixpoint subDenote Δ Δ' : Sub Δ' Δ → env Δ → env Δ'.
```

It takes the expressions from the substitution for all variables and evaluates them and puts them in the environment typed with Δ'. And now we define a function that applies a substitution to a contract specification.

```
Fixpoint SubstContract Γ Δ Δ' (s : Sub Δ Δ') (c : contract Γ Δ) : contract Γ Δ'.
```

For use in the correctness for the reduction semantics, we need to prove that substitution commutes. This means that applying the substitution to the contract has the same satisfying traces as the contract does in the denoted substitution:

```
Lemma substCommCsat :
    ∀ Δ Γ (D : template_env Γ) (c : contract Γ Δ) Δ′ (s : Sub Δ Δ′) t,
    ∀ δ, csat (subDenote s δ) D t c ↔
      csat δ D t (SubstContract s c).
```

This is actually proven by using the denotational semantics, but by the previous theorem we also have this for the trace satisfaction semantics. This substitution lemma corresponds to Lemma 2 in Andersen et al. [And+06].

One thing that is very nice about the intrinsic encoding is that we can directly state in the type of a contract whether it is closed. A closed contract will have the type `contract Γ []` stating that there are no free variables since nothing has membership in `[]`.

For the reduction semantics we are going to use two special substitutions. One substitutes in the arguments to a template when reducing a template application. This will be a substitution from the environment indexed by the typing signature `ts` to the empty environment:

```
Fixpoint argEnvSub ts (ae : arg_env ts []) : Sub ts [].
```

We prove that this substitution commutes with evaluating the arguments to a template

```
Lemma argEnvDenoteArgEnvSub : ∀ ty (ae : arg_env ty []),
   arg_envDenote HNil ae = subDenote (argEnvSub ae) HNil.
```

The other substitution that we need for the reduction semantics substitutes in the variables from an event, and gets rid of the free variables when reducing a Transfer.

```
Definition eventSub Δ (e : event) : Sub (new_var Δ) Δ.
```

4.7 Residuation

In this section we are going to mechanize residuation and the reduction semantics of CSL. For this we will start by defining what it means to be a residual contract. We say that c' is the residual contract for c given an event e if the residual contract denotes the rest of the trace. We write this in Coq exactly like we did in Section 2.5:

```
Definition residuates Γ Δ (D : template_env Γ) (c c' : contract Γ Δ) e :=
  ∀ t δ, csat δ D t c' ↔ csat δ D (e :: t) c.
```

4.7.1 Nullability

To mechanize both guardedness and the reduction semantics we need to encode nullability. We define syntactic nullability from Figure 2.4 as a relation:

```
Inductive nullable : ∀ Γ Δ, template_env Γ → contract Γ Δ → Prop.
```

We prove that a nullable contract actually admits the empty trace. This is the semantic characterization of nullability.

```
Theorem nullability : ∀ Γ Δ (δ : env Δ) (D : template_env Γ) c,
  nullable D c ↔ denotes δ D c [].
```

We might need to talk about closed contracts as well as open contracts so we prove that if we have any substitution then it does not change the nullability of a contract.

```
Lemma nullableSub : ∀ Γ Δ Δ' (D : template_env Γ) (c : contract Γ Δ) s,
  nullable D c ↔ @nullable Γ Δ' D (SubstContract s c).
```

Finally we differed from the paper formalization a bit in the case of semantic nullability. In the original paper they state semantic nullability by saying that there exists an environment where the contract denotes the empty trace. We said that is should be the case for all environments. For our definition to imply the existence, we just need to show that environments are inhabited. We do this by proving the following lemma:

```
Lemma env_inhabited : ∀ Δ, inhabited (env Δ).
```

Where `inhabited A` just requires us to give a term of type `A`.

4.7.2 Guardedness

Just like for nullability we state guardedness as a relation on contracts. This is a direct encoding of the rules in Figure 2.5.

```
Inductive guarded : ∀ Γ Δ, template_env Γ → contract Γ Δ → Prop.
```

We define guardedness for template environments by requiring all bodies in templates to be guarded.

```
Definition fguarded Γ (D : template_env Γ) :=
  ∀ ts (f : member ts Γ), guarded D (hget D f).
```

This is very easy to state with the dependently typed De Bruijn indexing of template names. It is very easy to make a statement for all valid templates.

It turns out that we need guardedness for closed contracts without free variables. Note that for a template body to be closed, we need to substitute in values for arguments. We make a similar relation as for open contracts, but we make sure that it is between closed contracts.

```
Inductive guarded0 : ∀ Γ template_env Γ → contract Γ [] → Prop.
Definition fguarded0 Γ (D : template_env Γ) := ∀ ts (f : member ts Γ) ae,
    guarded0 D (SubstContract (argEnvSub ae) (hget D f)).
```

We show that guardedness for templates implies guardedness for all contract specifications with those templates. We only show it for open contracts here, but we have also shown it for closed contracts.

```
Lemma fguarded_guarded : ∀ Γ Δ (D : template_env Γ) (c : contract Γ Δ),
    fguarded D → guarded D c.
```

We also show that guardedness for open contracts implies guardedness for closed contracts

```
Lemma fguarded_equiv : ∀ Γ (D : template_env Γ),
    fguarded D → fguarded0 D.
```

These lemmas are used when showing properties about the deterministic reduction semantics, and it turns out that it is much easier to use the closed definition.

4.7.3 Delayed matching

To encode the delayed matching semantics from Figure 2.6 we make an inductive datatype encoding the rules directly. It encodes an inductive relation between two closed contracts and an event.

```
Inductive delayed_matching Γ :
  template_env Γ → contract Γ [] → event → contract Γ [] → Prop.
| transferDelay1 : ∀ e D (p : exp (new_var []) Bool) c,
    expDenote p (addEvent e HNil) = true →
    delayed_matching D (transfer p c) e (SubstContract (eventSub e) c)
| transferDelay2 : ∀ D e (p : exp (new_var []) Bool) c,
    expDenote p (addEvent e HNil) <> true →
    delayed_matching D (transfer p c) e (failure _ _)
(* Additional rules *)
```

For transfers we write `SubstContract (eventSub e) c` to substitute the values from the event into the subcontract `c`. Then we reduce to the remaining contract if the predicate is true, otherwise we reduce to failure. This is exactly what we write in Figure 2.6. The rest of the rules are as you would expect.

We now prove that if a contract matches an event and transforms into a residual contract, then it is a valid residuation.

```
Theorem delayed_matching_res : ∀Γ (D : template_env Γ) (c c' : contract Γ []) e,
   delayed_matching D c e c' → residuates D c c' e.
```

We also prove that if D is guarded, then there is a unique residuation which is also guarded. Note that we can write ∃! in Coq to denote unique existence. This requires us to prove uniqueness after claiming that the c' exists.

```
Theorem unique_residuation : ∀Γ (D : template_env Γ) e (c : contract Γ []),
   fguarded D → ∃! c', delayed_matching D c e c' ∧ guarded D c'.
```

This proof relies of determinism of the expression language to make a case analysis whether the predicate will be true or not given the event. This concludes the mechanization of the delayed matching semantics. We will now continue with the eager matching semantics.

4.7.4 Eager matching

Just like before we encode the eager semantics from Figure 2.7 as an inductive data type:

```
Inductive eager_matching Γ :
  template_env Γ → contract Γ [] → option event → contract Γ [] → Prop.
```

Where `Some e` corresponds to an actual event and `None` corresponds to a spontaneous τ-transition.

We show the soundness of eager matching by considering τ-transitions and transitions on actual events separately. First we show that if we make a τ-transition we remain sound. In this case it means that after a τ-transition, we should not denote more traces than before.

```
Theorem eager_sound_tau : ∀Γ (D : template_env Γ) c c' t,
   eager_matching D c None c' → (denotes HNil D c' t → denotes HNil D c t).
```

Then when reducing on a concrete event `Some e` we show that we are sound with respect to the residual contract

```
Theorem eager_sound_event : ∀Γ (D : template_env Γ) c c' t e,
   eager_matching D c (Some e) c' → (denotes HNil D c' t → denotes HNil D c (e :: t)).
```

For the completeness of eager matching, we start by defining what it means to τ-step between two contracts. This is an inductively defined relation from one closed contract to another by making a sequence of τ-reductions. We say that either a contract τ-steps to itself, or it makes one additional τ-reduction.

```
Inductive tau_steps_eager Γ (D : template_env Γ) : contract Γ [] → contract Γ [] → Prop.
| step0 : ∀c, tau_steps_eager D c c
| stepS : ∀c c' c'',
   eager_matching D c None c' → tau_steps_eager D c' c'' →
   tau_steps_eager D c c''.
```

We prove that this relation is transitive, and that it is injective with respect to both branches in parallel composition:

```
  Lemma tau_steps_inj_conc : ∀Γ (D : template_env Γ) c1 c1' c2 c2',
   tau_steps_eager D c1 c1' →
   tau_steps_eager D c2 c2' →
   tau_steps_eager D (c1 :||: c2) (c1' :||: c2').
```

And that it is injective with respect to the first contract in sequential composition:

```
Lemma tau_steps_inj_seq : ∀ Γ (D : template_env Γ) c1 c1' c,
    tau_steps_eager D c1 c1' →
    tau_steps_eager D (c1 ;; c) (c1' ;; c).
```

These statements are needed to prove that a nullable contract τ-steps to Success.

```
Lemma nullable_tau_steps : ∀ Γ Δ (D : template_env Γ) (c : contract Γ Δ) s,
  nullable D c →
  tau_steps_eager D (SubstContract s c) (success _ _).
```

Now that we have this machinery defined, we are ready to prove the completeness of eager matching. The corresponding theorem in Andersen et al. [And+06] states that if $D \vdash_{\mathrm{D}} c \xrightarrow{e} c'$ then there exists a non-empty set of contracts $\{c_1, \ldots, c_n\}$ such that $D \vdash_{\mathrm{N}} c \xrightarrow{e} c_i$ for all $i \in 1 \ldots n$ and $D \vdash c' \subseteq \Sigma_{i=1}^{n} c_i$.

When encoding this we have to make some choices. There are multiple ways of encoding the last statement. Either we could say that c' is included in each of the alternatives, or that we actually construct one large syntactic alternative. We have chosen to interpret it as a syntactic alternative, meaning that the sum is actually an object-level sum in CSL.

$$\Sigma_{i=1}^{n} c_i = c_1 + (c_2 + (\cdots + (c_n + \mathsf{Failure}) \cdots))$$

In Coq this is a fold with syntactic $+$.

```
Definition alternatives cs := fold_right (fun c1 c2 ⇒ c1 :+: c2) (failure _ _) cs.
```

So the statement $D \vdash c' \subseteq \Sigma_{i=1}^{n} c_i$ gets encoded as

```
∀ t, denotes D HNil c' t → denotes D HNil (alternatives cs) t
```

We faithfully encode the statement of the theorem like this:

```
Theorem eager_complete : ∀ Γ (D : template_env Γ) (c c' : contract Γ []) e,
    delayed_matching D c e c' →
    ∃ cc cs, (* non-empty list *)
      Forall (fun c'' ⇒ eager_matching D c (Some e) c'') (cc :: cs) ∧
      ∀ t, (csat HNil D t c' → csat HNil D t (alternatives (cc :: cs))).
```

Note that we require one additional element `cc` to ensure that the list is non-empty. Now if we have a delayed matching, then there exists a non-empty list of contracts, such that all of them eagerly matches the event, and if the original residual contract from the delayed matching is satisfies a trace, then the big syntactic alternative of all the possible eager matches also satisfies the trace. Note that `Forall` is a predicate that extends a predicate `p : A → Prop` onto a list and holds whenever `p a` holds for all elements `a` in the list.

This proof is a bit of a pain. If we take one of the cases, say the delayed matching of sequential composition with the first contract being nullable.

$$\frac{D \vdash c \text{ nullable} \qquad D \vdash_{\mathrm{D}} c \xrightarrow{e} d \qquad D \vdash_{\mathrm{D}} c' \xrightarrow{e} d'}{D \vdash_{\mathrm{D}} c; c' \xrightarrow{e} (d; c') + d'}$$

Then we discharge the induction hypothesis on both delayed matchings and we get two lists of alternatives $d_1, \ldots, d_n$ and $d'_1, \ldots, d'_m$. The resulting list of alternatives that will make the case work out will be $d_1; c' + \cdots + d_n; c' + d'_1 + \cdots + d'_m$. Constructing this with lists in Coq requires a lot of small lemmas about lists and alternatives, sequence and so forth and it is very technical.

We will now move on to the second part of the thesis which is contract analysis. We will start by giving some background on abstract interpretation.

Chapter 5

Abstract interpretation

In this short chapter we will introduce the concept of abstract interpretation. For this we will use some lattice theory. The subset of lattice theory that we will use in the following chapters is described in Appendix B.1.

In computability theory, Rice's theorem states that all non-trivial semantic properties of programs are undecidable. This is also true for CSL even when the expression language is fairly simple. This means that answering any non-trivial question about a contract is in general undecidable.

Abstract interpretation [CC92a] is a framework for defining sound analyses that approximate the behavior of computer programs. The main application of abstract interpretation is static analysis where we compute properties of programs without running them.

The classical abstract interpretation approach starts with an operational semantics for the programming language to be analyzed. Then a collecting semantics is defined as the strongest set of static properties of interest. Then an abstraction of the collecting semantics is performed to make an analysis effectively computable. This abstraction is typically described by a Galois connection to a complete lattice. To find a concrete analysis result, fixed point algorithms are used to find a fixed point on the analysis lattice.

5.1 Collecting semantics

The collecting semantics (or static semantics in some sources) have to be designed in a way that it captures all possible properties of interest. Some properties can be forgotten if they are not interesting to us. For instance non-termination is described by some semantics, but not others. The collecting semantics is often taken to be the fixed point of some operational semantics. In the fixed point we have all executions of programs, terminating and non-terminating.

There is no single definition of what a collecting semantics is, and it might vary depending on the nature of the language and how its semantics is defined. It also depends on the properties we are interested in. If we only care about terminating programs, then a big-step semantics or denotational semantics might be OK, but if we also care about diverging programs then a small step semantics might be better suited.

In this thesis we are going to think of a collecting semantics as the semantics that collects all possible behaviors of a program, where the operational semantics might only describe one particular execution. So if one execution of the program is described by a value $\ell \in L$ then all the possible behaviors are described by a set $S \subseteq L$. For an imperative program, a particular execution can be described by a final state of the memory, and the collecting semantics is a set of possible states that might be the result of running the program.

5.2 Galois connections

The purpose of the Galois connection is to abstract the collecting semantics into an abstract semantics. The collection semantics is usually defined as a fixed point of the concrete semantics resulting in a set of possible executions. The Galois connection then expresses a correspondence between the collection semantics and the abstract properties as a pair of functions:

Definition 5.1. *A Galois connection between to lattices $(A, \subseteq)$ and $(B, \sqsubseteq)$ is a pair of monotone functions $\alpha : A \to B$ and $\gamma : B \to A$ such that*

$$\forall a \in A, b \in B. \alpha(a) \sqsubseteq b \iff a \subseteq \gamma(b).$$

In a Galois connection, elements of A are typically sets of semantic values and elements of B are properties describing them. If we consider arithmetic expressions, a collecting semantics might define sets of integers that an expression might evaluate to. A classic example is abstracting this domain into a domain of signs.

Example 5.1. *$\mathcal{P}(\mathbb{Z})$ is a complete lattice. We can abstract this into a lattice of signs $S = \{\bot, \top, 0, +, -\}$ where $\bot < 0 < \top$, $\bot < + < \top$ and $\bot < - < \top$. by the following abstraction function:*

$$\begin{aligned}
\alpha(\{0\}) &= 0 \\
\alpha(\{\}) &= \bot \\
\alpha(S) &= + \textit{ if all elements of S are strictly positive} \\
\alpha(S) &= - \textit{ if all elements of S are strictly negative} \\
\alpha(S) &= \top \textit{ otherwise}
\end{aligned}$$

Now we can define the concretization function as:

$$\begin{aligned}
\gamma(0) &= \{0\} \\
\gamma(+) &= \{z \in \mathbb{Z} \mid z > 0\} \\
\gamma(-) &= \{z \in \mathbb{Z} \mid z < 0\} \\
\gamma(\bot) &= \{\} \\
\gamma(\top) &= \mathbb{Z}
\end{aligned}$$

It is routine to check that γ and α are monotone. We can prove that it is a Galois connection by considering the cases for α and γ. For instance let $a \subseteq \mathcal{P}(Z)$ be a set with all strictly positive elements and now assume that $\alpha(a) \leq b$. By the ordering on S, b is $+$ or $\top$. In both cases it is the case that $a \subseteq \gamma(b)$. We can do similar proofs for the rest of the cases. This will prove that we have a Galois connection.

We are not going to formulate Galois connection as Cousot and Cousot does it, but we are going to use representation functions that are maps between values and the best properties describing them. A representation function is on the form $\beta : V \rightarrow L$, and we are going to derive a Galois connection between $\mathcal{P}(V)$ and L from it. As described in [NNH99], the following functions describe a valid Galois connection using representation functions:

- $\gamma : L \rightarrow \mathcal{P}(A) = \lambda\ell.\{v \mid \beta(v) \sqsubseteq \ell\}$
- $\alpha : \mathcal{P}(A) \rightarrow L = \lambda V. \bigsqcup_{v \in V} \beta(v)$

To find an analysis of a program in the abstract domain we are going to exploit the fact that we required L to be a complete lattice. This means that if the functions from our abstract semantics are monotone, then we can find a fixed point using relatively simple algorithms.

5.3 Fixed point algorithms

Given a complete lattice $(L, \sqsubseteq)$, then in an analysis setting, we can think of a program p as transforming one property ℓ into another property ℓ'. That is $\ell = f(\ell')$. In the case of recursive programs, the analysis of a function might depend on itself, so in some sense $\ell = f(\ell)$. We typically require f to be monotone, and to find a solution we hope that upwards iteration from $\bot$ finds a fixed point. That is $f^n(\bot) = f^{n+1}(\bot)$ for some n. When L has finite ascending chains, then $f^n(\bot)$ always terminate for some n with $f^n(\bot) = \mathrm{lfp}(f)$, since the iteration of f forms an ascending chain. We describe this in the appendix. For the analyses in this thesis we are going to use abstract domains that do have infinite ascending chains. For this we will use the technique of widening to ensure termination of the fixed point method.

5.3.1 Widening

The widening technique as described by Cousot and Cousot [CC92b] is used to approximate fixed points on complete lattices. A widening operator $\triangledown : L \times L \rightarrow L$ is used to force every ascending chain in the lattice to be finite. We write

$$\ell_n^{\triangledown} = \begin{cases} \ell_n & \text{if } n = 0 \\ \ell_{n-1}^{\triangledown} \triangledown \ell_n & \text{if } n > 0 \end{cases}$$

to denote that we put the widening operator between each value of a chain. We require the following properties of the widening operator:

- $\ell_1 \sqsubseteq \ell_1 \triangledown \ell_2 \sqsupseteq \ell_1$, for all $\ell_1, \ell_2 \in L$.
- For all ascending chains $(\ell_n)_n$, the ascending chain $(l_n^{\triangledown})_n$ eventually stabilizes.

Instead of naively iterating from $\bot$ we define a new iteration strategy for f:

$$f_{\triangledown}^{n} = \begin{cases} \bot & \text{if } n = 0 \\ f_{\triangledown}^{n-1} & \text{if } n > 0 \wedge f(f_{\triangledown}^{n-1}) \sqsubseteq f_{\triangledown}^{n-1} \\ f_{\triangledown}^{n-1} \triangledown f(f_{\triangledown}^{n-1}) & \text{otherwise} \end{cases}$$

where we essentially put the widening operator between every application of f. And now Nielson and Nielson [NNH99] prove that if $\triangledown$ is a widening operator, then $(f_{\triangledown}^{n})_n$

eventually stabilizes. We can then use this iteration strategy when finding fixed points on complete lattices that has infinite ascending chains.

For the interval lattice one widening operator is:

$$\begin{aligned}
\bot \mathbin{\triangledown} y &= y \\
x \mathbin{\triangledown} \bot &= x \\
[a_1, b_1] \mathbin{\triangledown} [a_2, b_2] &= [a, b] \\
\text{where } a &= \begin{cases} a_1 & \text{when } a_1 \leq a_2 \\ -\infty & \text{otherwise} \end{cases} \\
b &= \begin{cases} b_1 & \text{when } b_2 \leq b_1 \\ \infty & \text{otherwise} \end{cases}
\end{aligned}$$

Intuitively it only allows us to make the bounds larger once, and then afterwards we will just go to ∞ or $-\infty$. Take for instance the infinite chain

$$[0, 1] \sqsubset [1, 1] \sqsubset [1, 2] \sqsubset [1, 3] \cdots$$

If we apply widening between every element, we get the chain

$$[0, 1] \sqsubset [1, 1] \sqsubset [1, \infty]$$

which now stabilizes. There is a large loss of precision here, and more complicated widening operators can be defined. One simple change is making a fixed number of iterations before applying widening. There is also the technique of narrowing which can recover some precision. We will not use narrowing for our analysis of CSL.

Chapter 6

Contract analysis

In this chapter we will develop a framework for analyzing compositional contracts, and we will use this framework to develop a few simple analyses and prove their correctness.

We motivate the need for contract analysis by looking at a simple multiparty contract simulating an escrow. Here `alice` wants to buy a bike from the shop, but to make sure that she gets the bike, she gives the money to a trusted third party which gives it to the shop whenever the bike is delivered to `alice`.

```
letrec escrow[trusted, seller, buyer, goods, payment, deadline] =
  Transfer(buyer, trusted, payment, _).
    (Transfer(seller, buyer, goods, T | T < deadline).
     Transfer(trusted, seller, payment, T’ | True).Success
   + Transfer(trusted, buyer, payment, T | T > deadline).Success)

    in escrow("3rd", "shop", "alice", 1 bike, 1000 EUR, 2019-09-01)
```

Remember that `Transfer(a, b, r, _ )` is an abbreviation for

```
Transfer(A, B, R, T | A = a ∧ B = b ∧ R = r)
```

In this multiparty contract we might want to know how the involved parties are sending resources. We might want to check that the trusted third party never receives money from the seller, and is only handling resources from the buyer. We call this participation analysis, and we are interested in inferring a relation between agents. For the escrow contract this relation is

$$R_c = \{(\texttt{3rd} \rightarrow \texttt{shop}), (\texttt{shop} \rightarrow \texttt{alice}), (\texttt{alice} \rightarrow \texttt{3rd}), (\texttt{3rd} \rightarrow \texttt{alice})\}$$

where $(\texttt{3rd} \rightarrow \texttt{shop})$ is read as `3rd` transfers a resource to `shop`. So it should be the case for all traces satisfying this contract that transfers only happen as described by this relation.

We might also be interested in the fairness of this contract. That is whether any participant might benefit too much from participating in the contract in relation to others. We will call this fairness analysis, and we are interested in inferring a set of parties and a bound on the cost/benefit of participating in the contract.

As an input to the analysis we will provide a map of resources to real numbers which provide a valuation of resources. In the case of the escrow contract the valuation could be:

$$V = \{\texttt{bike} \mapsto 900, \texttt{EUR} \mapsto 1\}.$$

Then if we analyze the contract there are two outcomes. If the shop does not deliver the bike, the shop and alice have no gain or loss. If the shop delivers the bike, the shop gains 100 and alice loses 100 because of the difference between value and purchase price. The set we would like to infer is

$$R_q = \{(\texttt{3rd}, [0, 0]), (\texttt{shop}, [0, 100]), (\texttt{alice}, [-100, 0])\}.$$

$(\texttt{shop}, [0, 100])$ is read as the shop gains between 0 and 100 by participating in this contract. We will see that the escrow contract is pretty simple to analyze, since it does not contain any recursion or transfers that can accept a wide array of events. If we look at a contract like

```
letrec f[a] = Success + Transfer(S, R, 1 EUR, T | S = a ∨ R = a).f(a)
    in f("alice")
```

Then it looks a lot harder. For the participation analysis we would like to be able to infer that

$$R_c = \{(\top \to \texttt{alice}), (\texttt{alice} \to \top)\}$$

or in other words: All transfers will have any sender and alice as a receiver or alice as a sender and any receiver. The best result for the fairness analysis in this case will be

$$R_q = \{(\top, [-\infty, \infty])\}$$

Meaning that all the agents collectively involved in contract execution (which may include Alice) might receive or pay an arbitrarily large amount by participating in the contract.

In the spirit of abstract interpretation we are focusing on analyses that work for all contracts. To make this viable we are going to accept over-approximations in some cases. We will now explain how to phrase abstract interpretation of CSL.

6.1 Abstract interpretation of CSL

The goal of contract analysis is to extract as much information about the possible behaviour of the contract without actually having an actual trace to check for satisfiability. We would like to answer questions like: "How much will it cost me to participate in this contract?", or "Who are the participating parties in this contract?". These are properties that are not obvious how to compute just looking at the contract.

We are going to take the classic approach to define an abstract analysis for CSL. First we will define a collecting semantics for CSL and then we will abstract it using the Galois connection described below.

6.1.1 Galois connections for CSL

In the case of CSL the Galois connection will be between $(\mathcal{P}(Tr), \subseteq)$ and some lattice $(L, \sqsubseteq)$ describing properties of traces. On Figure 6.1 we have visualized the Galois connection. Intuitively α abstracts trace sets to properties, and γ maps properties to the trace sets that are described by them. The figure shows that if we apply the abstraction, we will always remain sound with respect to the concrete semantics. We will not define this Galois connection explicitly, but we will use representation functions $\beta : Tr \to L$ to map traces to the best properties describing them. We will take this approach mainly because the correctness is going to be based on the trace satisfaction relation, and it

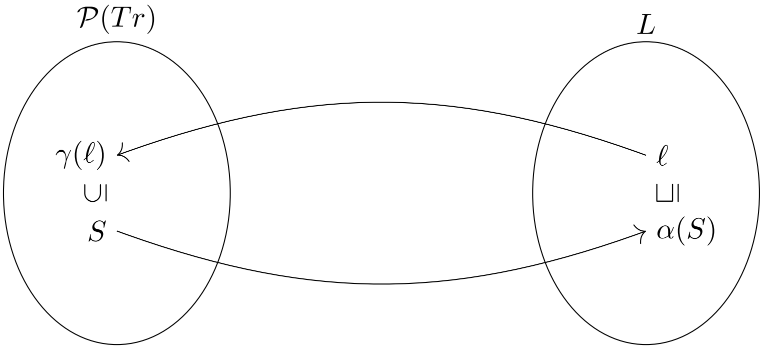


Figure 6.1: Visualizing the Galois connection

describes individual traces. We will leave the actual representation function unspecified, since we are interested in defining the most general analysis possible.

The goal is now to have an analysis $[\![c]\!]^\sharp \in L$ that describes the possible satisfying traces of a contract. Then we will say that if we have a trace $s \in Tr$ satisfying the contract $\delta \vdash^D s : c$ then it should be the case that $\beta(s) \sqsubseteq [\![c]\!]^\sharp$. In other words, the analyzed behavior should describe all possible satisfying traces. Before constructing the analysis, we will tackle analysis of expressions. This is separate from the actual abstract analysis just like the definition of the expression language for CSL was almost independent from the CSL definition.

6.2 Analysis of expressions

In CSL, the only way to accept events is through the transfer construct:

$$\mathsf{Transfer}(A_1, A_2, R, T \mid P).c$$

From the semantics we know that an event is matched by the Transfer if the predicate P evaluates to true. Therefore to analyze CSL in any meaningful way, we need to be able to extract information from P. We want to extract information about variables given that $\delta \models P$ and propagate this information to the analysis of the subcontract c. This is not unlike the proof rule in Hoare logic for if-statements where in the body, we can assume the condition to be true.

For the analysis of predicates we require an abstract environment $M : Var \to A$ with a mapping from variables to a complete lattice of abstract values $(A, \sqsubseteq)$. We also require an abstraction function α_A that lifts actual values into their abstract counterpart. If we are dealing with a power set lattice then we can define $\alpha_A(x) = \{x\}$. We will see later that the choice of abstract environment will have a large impact on the precision and complexity of the analysis. For two abstract environments m_1, m_2 we define an ordering:

$$m_1 \sqsubseteq m_2 \iff \forall x \in Var. m_1(x) \sqsubseteq m_2(x).$$

The actual analysis will refine an $m \in M$ given that the predicate must be true. We write this as the analysis of P

$$[\![P]\!]^\sharp : M \to M_\perp.$$

Lifting the result into a domain including $\bot$ means that we also have the possibility of returning $\bot$ if we are certain that the predicate cannot be satisfied. This makes it possible to be much more precise in, say, the case of a recursive contract with a simple condition limiting the number of recursive invocations. This could for instance be a loan where the number of recurring payments is fixed. For instance the contract

```
letrec repay[amount, payments] =
    Transfer("alice", "bob", amount, _ | payments = 1).Success
  + Transfer("alice", "bob", amount, _ | payments > 1).repay(amount, payments - 1)
in Transfer("bob", "alice", 12000 EUR, _).repay(1000 EUR, 12)
```

specifying a loan with 12 payments of 1000 euros. Intuitively at some point, `payments` will be 1 and the predicate analysis should be able to return $\bot$ for the right branch of the alternative. For the first 11 payments it should be able to return $\bot$ for the left branch.

In general we can make a trivial analysis for the predicate by returning the input directly, this also gives us a sound way to skip analyzing predicates that are too hard.

To make the expression analysis complete we also need to analyze template arguments for an application $f(a_1, \dots, a_n)$ of a template $f[x_1, \dots, x_n] = c$. Here we want a transformation

$$\llbracket (a_1, x_1), \dots, (a_n, x_n) \rrbracket^\sharp : M \to M$$

that changes the current abstract environment into one that can be used to analyze the body of the template. It could do different things depending on the analysis we are defining, but a simple implementation can just look up arguments in the abstract environment and abstract literals. A more complicated analysis might do computations in order to be more precise (this is necessary for the loan contract `repay`).

in the following section we describe what we require about the analysis of expressions in order to use it in an analysis for contract specifications.

6.2.1 Correctness

We relate the actual environment to an abstract environment by saying that an abstract environment describes and actual environment if all variables are described by one in the abstract environment. We will denote this by a relation $\mathcal{R}_M$ between δ's and m's such that

$$\delta \; \mathcal{R}_M \; m \iff \forall x \in Var. \alpha_A(\delta(x)) \sqsubseteq m(x).$$

We can use this to describe what we require for the correctness of a predicate analysis. We want it to preserve the overapproximation given that the predicate is satisfiable:

$$\delta \; \mathcal{R}_M \; m \wedge \delta \models P \implies \delta \; \mathcal{R}_M \; \llbracket P \rrbracket^\sharp m$$

Thereby preserving the invariant that m is correct. It should also have the property that if the analysis signals unsatisfiability, then the predicate is not satisfiable:

$$\delta \; \mathcal{R}_M \; m \wedge \llbracket P \rrbracket^\sharp m = \bot \Rightarrow \delta \not\models P$$

Furthermore when the predicate is satisfiable, then it should be satisfiable in a larger environment, and the resulting environments should be monotone:

$$m_1 \sqsubseteq m_1' \wedge \llbracket P \rrbracket^\sharp m_1 = m_2 \neq \bot \wedge \llbracket P \rrbracket^\sharp m_1' = m_2' \neq \bot \Rightarrow m_2 \sqsubseteq m_2'$$

$$\begin{array}{rcll} v & ::= & Variable & \text{(Value expressions)} \\ & \mid & Literal & \\ b & ::= & b_1 \wedge b_2 & \text{(Boolean expressions)} \\ & \mid & b_1 \vee b_2 & \\ & \mid & \neg b & \\ & \mid & v_1 = v_2 & \end{array}$$

Figure 6.2: Syntax for simplified predicates

And if the predicate is not satisfiable in a large environment it should not be satisfiable in a smaller one.

$$m \sqsubseteq m' \wedge \llbracket P \rrbracket^{\sharp} m' = \bot \Rightarrow \llbracket P \rrbracket^{\sharp} m = \bot$$

We also require that the abstract evaluation of arguments is an over-approximation. For an application $f(a_1, \ldots a_n)$ of a template $f[x_1, \ldots x_n] = c$ we want that

$$\delta \; \mathcal{R}_M \; m \Rightarrow \{x_1 \mapsto \mathcal{Q}\llbracket a_1 \rrbracket^{\delta}, \ldots, x_n \mapsto \mathcal{Q}\llbracket a_n \rrbracket^{\delta}\} \; \mathcal{R}_M \; \llbracket (a_1, x_1), \ldots, (a_n, x_n) \rrbracket^{\sharp} m$$

And we also require that it is monotone, so evaluating the arguments in a larger environment will result in a larger environment.

$$m \sqsubseteq m' \Rightarrow \llbracket (a_1, x_1), \ldots, (a_n, x_n) \rrbracket^{\sharp} m \sqsubseteq \llbracket (a_1, x_1), \ldots, (a_n, x_n) \rrbracket^{\sharp} m'$$

Such an analysis will be useful as a starting point for defining an analysis for contracts. We will now hint a simple way to phrase such an expression analysis.

6.2.2 An example predicate analysis

In this example we use the power set lattice for variables to describe their possible values in a non-relational way.

$$M = Var \rightarrow \mathcal{P}(\mathcal{D})$$

where $\mathcal{D} = \mathcal{A} \cup \mathcal{R} \cup \mathcal{T}$ is the domain of all values. When we write an environment, for instance $\{a \mapsto A\}$, then it is implicit that all other variables than a maps to $\top$.

We start by assuming that the predicate is written in a specific form. We include only the usual boolean operators and variables and literals. The syntax of predicates can be seen on Figure 6.2. For arguments to functions we will also include basic arithmetic. Since this is a fairly informal treatment, we will also assume that the predicates are well typed. Now there is a simple and very naive algorithm to extract at least some constraints from these predicates in Figure 6.3. It is a form of unification, where we start with some constraints and then we generate a stronger set of constraints from the equalities in the predicate. We let M^C denote the point-wise complement of each member of M. We also let $\sqcup$ and $\sqcap$ denote point-wise union and intersection respectively.

For template arguments a simple algorithm just works for variables and literals and looks them up or abstracts them. If we do any arithmetic, we just perform it on the values in the set. We now look at a very simple example of the predicate analysis.

$$
\begin{aligned}
[\![e_1 \wedge e_2]\!]^\sharp m &= [\![e_1]\!]^\sharp m \sqcap [\![e_2]\!]^\sharp m \\
[\![e_1 \vee e_2]\!]^\sharp m &= [\![e_1]\!]^\sharp m \sqcup [\![e_2]\!]^\sharp m \\
[\![\neg e]\!]^\sharp m &= ([\![e]\!]^\sharp m)^C \\
[\![v = a]\!]^\sharp m &= m[v \mapsto (m(v) \cap \{a\})] \\
[\![a = v]\!]^\sharp m &= m[v \mapsto (m(v) \cap \{a\})] \\
[\![v = v']\!]^\sharp m &= m[v \mapsto (m(v) \cap m(v')), v' \mapsto (m(v) \cap m(v'))] \\
[\![a = a']\!]^\sharp m &= \bot \text{ if } a \neq a' \\
[\![\_]\!]^\sharp m &= m
\end{aligned}
$$

Figure 6.3: Algorithm for extracting constraints from predicates

Example 6.1 (Predicate analysis). *If we look at a very simple predicate that might appear as a predicate in a contract:*

$$P = ((a = \text{"}alice\text{"} \vee a = b) \wedge r = 2 \text{ iPhone})$$

then the result of using the algorithm could be

$$[\![P]\!]^\sharp \{b \mapsto \{\text{"}bob\text{"}\}\} = \{a \mapsto \{\text{"}alice\text{"}, \text{"}bob\text{"}\}, b \mapsto \{\text{"}bob\text{"}\}, r \mapsto \{2 \text{ iPhone}\}\}$$

We also show a short example of how the analysis of template arguments works.

Example 6.2 (Argument analysis). *The analysis of the arguments in an application* $f(x, y - 7 \text{ days}, 1 \text{ iPhone})$ *where* $D(f) = (f[a_1, a_2, a_3] = c)$ *could be*

$$
\begin{aligned}
&[\![(x, a_1), (y - 7 \text{ days}, a_2), (1 \text{ iPhone}, a_3)]\!]^\sharp \{y \mapsto \{\textit{2019-10-10}, \textit{2019-09-10}\} = \\
&\qquad\qquad \{a_2 \mapsto \{\textit{2019-10-03}, \textit{2019-09-03}\}, a_3 \mapsto \{1 \text{ iPhone}\}\}
\end{aligned}
$$

More complex algorithms could be developed, and the use of a relational domain could be very beneficial for especially agents in predicates. Take for instance a transfer

$$\mathsf{Transfer}(A_1, A_2, R, T \mid A_1 = \text{"}alice\text{"} \vee A_2 = \text{"}alice\text{"}).c$$

The previous analysis would conclude that both A_1 and A_2 could be anything, which is also true if we look at the variables individually, but for any transfer we will have Alice as either the sender or receiver. This is not sufficient for the participation analysis that we hinted in the introduction to be precise.

We will now use the general expression analysis framework described here to develop a general framework for contract analysis that we can use to phrase analyses that will infer properties on satisfying traces.

6.3 A collecting semantics for CSL

In the true spirit of abstract interpretation, we are going to “calculate” an analysis for CSL. We are going to do this by systematically transforming the trace satisfaction semantics into an abstract semantics for CSL. We are going to differ in one place though. We are not going to prove the correctness of the collecting semantics with respect to the original semantics. We are going to wait until we have defined the abstract analysis.

As a first step towards an analysis for CSL we are going to transform the contract satisfaction semantics into a collecting semantics for CSL. The strongest static property for a contract is the set of traces that satisfies it, so the collecting semantics are going to state which traces $S \subseteq Tr$ satisfy a contract.

We are then going to abstract the collecting semantics using the Galois connection described in Section 5.2 to get a abstract collecting semantics for CSL where we will end up with a computationally simpler domain. The development in this section is inspired by the work of Schmidt [Sch95] on doing abstract interpretation based on big step semantics for a functional programming language.

The choice of basing the analysis on the contract satisfaction semantics has the consequence of only letting us infer properties of satisfying traces. If a contract is going to be breached at some point in the future, then no properties that we infer will be valid. This could be a problem with a practical contract monitoring system, where we accept events and reduce contracts with the reduction semantics eagerly. We might also be interested in properties of traces where some prefix does not breach the contract. We will not investigate these analyses in this thesis.

To get started with the analysis we are going to make an extreme simplification. We are going to swap out the $\mathsf{Transfer}(A_1, A_2, R, T \mid P).c$ construct with one with actual values instead of variables, $\mathsf{Transfer}(a_1, a_2, r, t).c$. Think of this as just translating

$$\mathsf{Transfer}(a_1, a_2, r, t).c \text{ to } \mathsf{Transfer}(A_1, A_2, R, T \mid a_1 = A_1 \wedge a_2 = A_2 \wedge r = R \wedge t = T).c.$$

This restricts the language to a great degree, but it serves the purpose of understanding how we can go from classifying satisfying traces to generating all satisfying traces.

The previous restriction of transfers implies that the variables are going to be useless, so we also ignore the environment. An attempt at describing all possible traces of a contract is shown in Figure 6.4. The judgment $D \rhd c : S$ states that a contract c has the satisfying traces S. The condition that makes this possible is that fact that there can only be one possible transfer event for any given syntactic $\mathsf{Transfer}$.

Now informally it should be the case that all the satisfying traces are collected into S:

$$\emptyset \vdash^D s : c \wedge D \rhd c : S \Rightarrow s \in S.$$

Note that now the analysis is algorithmic in the sense that there is only one rule for each syntactic constructor of CSL. This removes the inherent non-determinism of the $+$-combinator.

As an example derivation of the hypothetical collecting semantics consider the simple CSL contract

```
letrec f[] = Success + Transfer(a, b, r, t).f() in f()
```

and let $e = \mathsf{transfer}(a, b, r, t)$. We want the collecting semantics for this contract to be the infinite tree $\mathcal{T} =$

$$\dfrac{\dfrac{\dfrac{}{D \rhd \mathsf{Success} : \{\langle\rangle\}} \qquad \dfrac{\dfrac{\mathcal{T}}{D \rhd f() : \{\langle\rangle, \langle e\rangle, \langle e, e\rangle, \ldots\}}}{D \rhd \mathsf{Transfer}(a, b, r, t).f() : \{\langle e\rangle, \langle e, e\rangle, \ldots\}}}{D \rhd \mathsf{Success} + \mathsf{Transfer}(a, b, r, t).f() : \{\langle\rangle, \langle e\rangle, \langle e, e\rangle, \ldots\}}}{D \rhd f() : \{\langle\rangle, \langle e\rangle, \langle e, e\rangle, \ldots\}}$$

$\boxed{D \rhd c : S}$ Contract specification c has trace set S

$$\frac{}{D \rhd \textsf{Success} : \{\langle\rangle\}} \qquad \frac{}{D \rhd \textsf{Failure} : \emptyset} \qquad \frac{D \rhd c_1 : S_1 \quad D \rhd c_2 : S_2}{D \rhd c_1 + c_2 : S_1 \cup S_2}$$

$$\frac{D \rhd c_1 : S_1 \quad D \rhd c_2 : S_2}{D \rhd c_1 \parallel c_2 : \{s \mid (s_1, s_2) \in S_1 \times S_2, (s_1, s_2) \rightsquigarrow s\}}$$

$$\frac{D \rhd c_1 : S_1 \quad D \rhd c_2 : S_2}{D \rhd c_1 ; c_2 : \{s_1 +\!\!+ s_2 \mid (s_1, s_2) \in S_1 \times S_2\}}$$

$$\frac{D \rhd c : S}{D \rhd \textsf{Transfer}(a_1, a_2, r, t).c : \{\textsf{transfer}(a_1, a_2, r, t)s \mid s \in S\}}$$

$$\frac{D \rhd c : S}{D \rhd f() : S} D(f) = (f[] = c)$$

Figure 6.4: Hypothetical collecting semantics for simple CSL

There are details about the co-inductive nature of $D \rhd c : S$ that we will get into later when actually proving the correctness of the abstract semantics. There we will give formal rules on how to construct these trees and rigorously prove properties about them.

In this case we convince ourselves that this collecting semantics somehow captures all traces described by the trace satisfaction semantics, since every node in the tree for the collecting semantics will contain the trace in the corresponding derivation for the trace satisfaction semantics. The correctness of the collecting semantics is not so important prove rigorously at the moment. The collecting semantics is just a stepping stone towards an abstract semantics.

Now we want to add the environment δ to the collecting semantics and remove the restriction on transfers. We will then have a judgment $D, \delta \rhd c : S$. This change has some interesting consequences. Imagine making a rule for Transfer:

$$\frac{D, \delta' \rhd c : S \quad \delta' = ?}{D, \delta \rhd \textsf{Transfer}(A_1, A_2, R, T \mid P).c : \{\textsf{transfer}(?, ?, ?, ?)s \mid s \in S\}}$$

It's not obvious what δ' should be and the values in the transfer at the head of all traces. Given the trace satisfaction rule for Transfer, we know that $\delta' \models P$. We can then think of the predicate in the Transfer as a function p' that transforms a δ to a set $\mathcal{D}$ of δ's with the property that for all the δ's, $\delta' \models P$. Formally we write this as a function:

$$p'(P, \delta) = \{\delta' \mid \delta' \models P\}$$

We extend p' to sets of environments by unioning all the resulting sets of environments:

$$p(P, \mathcal{D}) = \bigcup \{p'(P, \delta) \mid \delta \in \mathcal{D}\}$$

We change the judgment to $D, \mathcal{D} \triangleright c : S$ such that $\mathcal{D}$ is a set of possible environments. Now we can finally write the rule for Transfer:

$$\frac{D, \mathcal{D}' \triangleright c : S \quad \mathcal{D}' = p(P, \mathcal{D})}{D, \mathcal{D} \triangleright \mathsf{Transfer}(A_1, A_2, R, T \mid P).c : t(S, \mathcal{D}', A_1, A_2, R, T)}$$

where the effect of a Transfer is described by the function t:

$$t(S, \mathcal{D}, A_1, A_2, R, T) = \{\mathsf{transfer}(\delta(A_1), \delta(A_2), \delta(R), \delta(T))s \mid s \in S, \delta \in \mathcal{D}\}$$

The satisfying transfers have actual values from all the possible environments. For template applications we basically have to evaluate the arguments in all the possible environments $\mathcal{D}$. This results in a set of environments that can be used in the body:

$$\frac{D, \mathcal{D}' \triangleright c : S \qquad \mathcal{D}' = \{x_1 \mapsto \mathcal{Q}[\![a_1]\!]^\delta, \ldots, x_n \mapsto \mathcal{Q}[\![a_n]\!]^\delta \mid \delta \in \mathcal{D}\}}{D, \mathcal{D} \triangleright f(a_1, \ldots, a_n) : S} (D(f) = f[x_1, \ldots, x_n] = c)$$

This basically concludes the construction of a collecting semantics for CSL. The collecting semantics is not easily computable, since the function p that computes the possible environments for the subcontract after a Transfer might be very hard to compute and will return an infinite set in a lot of cases.

We will from this idealized collecting semantics derive an abstract version that will be computable by regular fixed point methods.

6.3.1 Abstract collecting semantics

We translate the collecting semantics for CSL into an abstract collecting semantics by abstracting sets of traces into values ℓ of a complete lattice $(L, \sqsubseteq)$. We abstract sets of environments as an abstract environment $m \in M : Var \to A$ where $(A, \sqsubseteq)$ is a complete lattice as well.

Instead of using p to compute possible environments for the remaining contract after a Transfer we are going to use the predicate analysis defined in Section 6.2:

$$[\![P]\!]^\sharp : M \to M_\perp.$$

The predicate analysis will refine abstract environments given the fact the P will evaluate to true in any of the $\mathcal{D}'$s that was the result of p. If it turns out that the predicate is not satisfiable, we will return $\perp$. We will also reuse the abstract semantics for evaluating template arguments for an application $f(a_1, \ldots, a_n)$ of a contract template $f[x_1, \ldots x_n] = c$ as a function that returns a new abstract environment for use in the body.

$$[\![(a_1, x_1), \ldots, (a_n, x_n)]\!]^\sharp : M \to M$$

We need a value L_{Success} for the analysis of the successful contract Success which might be different from $\perp$ that should somehow describe the empty trace. Since we want to define the analysis compositionally, we need functions for combining the resulting analyses in the case of $+$, ; and $\|$:

$$C_+, C_;, C_\| : L \times L \to L$$

For $+$ the only option is setting $C_+ = \sqcup$. This is because the join of the lattice is the least element that describes both alternatives. For the other operations we might do

$\boxed{D, m \rhd c : \ell}$ Contract specification c has abstract trace ℓ

$$\frac{}{D, m \rhd \textsf{Success} : L_{\textsf{Success}}} \qquad \frac{}{D, m \rhd \textsf{Failure} : \bot}$$

$$\frac{D, m \rhd c_1 : \ell_1 \quad D \rhd c_2 : \ell_2}{D, m \rhd c_1 \parallel c_2 : C_{\parallel}(\ell_1, \ell_2)} \qquad \frac{D, m \rhd c_1 : \ell_1 \quad D \rhd c_2 : \ell_2}{D \rhd c_1; c_2 : C_{;}(\ell_1, \ell_2)}$$

$$\frac{D, m \rhd c_1 : \ell_1 \quad D \rhd c_2 : \ell_2}{D, m \rhd c_1 + c_2 : \ell_1 \sqcup \ell_2} \qquad \frac{[\![P]\!]^{\sharp} m = \bot}{D, m \rhd \textsf{Transfer}(A_1, A_2, R, T).c : \bot}$$

$$\frac{D, m' \rhd c : \ell \quad m' = [\![P]\!]^{\sharp} m \neq \bot}{D, m \rhd \textsf{Transfer}(A_1, A_2, R, T).c : C_{\textsf{Transfer}}(\ell, m', A_1, A_2, R, T)}$$

$$\frac{m' = [\![(a_1, x_1), \ldots, (a_n, x_n)]\!]^{\sharp} m \quad D, m' \rhd c : \ell}{D, m \rhd f(a_1, \ldots, a_n) : \ell} D(f) = (f[x_1, \ldots, x_n] = c)$$

Figure 6.5: Abstract collecting semantics

something different. It might be that we somehow can exploit the order of events or that we know that both subcontracts happen.

And now the interesting case for $\textsf{Transfer}(A_1, A_2, R, T \mid P).c$ where we must combine the result for c with the result of analyzing P given the bound variables. We describe this as a function which calculates an analysis given the updated environment and the result of analyzing the subcontract:

$$C_{\textsf{Transfer}} : L \times M \times Var^4 \to L$$

These functions will be abstract versions of the combination functions in the hypothetical collecting semantics in Figure 6.4. We have added a rule for Transfer in the case that $[\![P]\!]^{\sharp} m = \bot$, just to make the analysis more precise to exploit obviously unsatisfiable contracts. This will also require that the predicate analysis is decidable, but we do not consider that requirement unreasonable. The abstract collecting semantics can be seen in Figure 6.5. If we make sure to construct combination functions such that they respect the ordering on both $\mathcal{P}(Tr)$ and L then we have a correct abstract collecting semantics with respect to the concrete collecting semantics. This is not something we will prove though, since we will eventually prove soundness with respect to the trace satisfaction semantics. We will now investigate what we should require of the combinators and β to make sure that we have a correct analysis.

6.3.2 Correctness of the combinators

We want to put restrictions on β with respect to the combinators such that (α, γ) is a valid Galois connection. This is in our case if

$$\forall S \in \mathcal{P}(Tr), \ell \in L. \alpha(S) \sqsubseteq \ell \Longleftrightarrow S \subseteq \gamma(\ell).$$

The soundness requirement for L_{Success} that gives us a valid Galois connection is

$$\beta(\langle\rangle) \sqsubseteq L_{\mathsf{Success}}. \tag{6.1}$$

For sequential composition it is natural to require that appending the two satisfying traces should produce something included in the combination of both subcontracts. This is also what gives a valid Galois connection

$$\beta(t_1) \sqsubseteq l_1 \land \beta(t_2) \sqsubseteq l_2 \Rightarrow \beta(t_1 +\!\!+ \, t_2) \sqsubseteq C_;(l_1, l_2). \tag{6.2}$$

For parallel composition we require something similar, but for an interleaving instead of an appending.

$$\beta(t_1) \sqsubseteq l_1 \land \beta(t_2) \sqsubseteq l_2 \land (t_1, t_2) \rightsquigarrow t \Rightarrow \beta(t) \sqsubseteq C_{\|}(l_1, l_2) \tag{6.3}$$

For an event $e = \mathsf{transfer}(a_1, a_2, r, t)$ matched by a $\mathsf{Transfer}(A_1, A_2, R, T \mid P).c$ we require that if we have a correct environment with the values from the event added, then the result of the combination function should be at least as large as the representation of the trace with the event in front. Here we reuse the correctness relation between the abstract environment and the concrete environment from the expression analysis to express that the abstract values that we extract from the environment are sound approximations of the actual ones in the transfer.

$$\begin{aligned}\delta[A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t] \; \mathcal{R}_M \; m \land \beta(t) \sqsubseteq l \\ \Rightarrow \beta(e\ t) \sqsubseteq C_{\mathsf{Transfer}}(l, m, A_1, A_2, R, T)\end{aligned} \tag{6.4}$$

Also to facilitate widening of the environment and to be able to use standard fixed point algorithms we require that all analysis combinators are monotone in their arguments

$$l_1 \sqsubseteq l'_1 \land l_2 \sqsubseteq l'_2 \Rightarrow C_{\mathsf{op}}(l_1, l_2) \sqsubseteq C_{\mathsf{op}}(l'_1, l'_2) \tag{6.5}$$

And the C_{Transfer} combinator should be monotone in the abstract environment as well as the analysis of subcontracts

$$m \sqsubseteq m' \land l_1 \sqsubseteq l_2 \Rightarrow C_{\mathsf{Transfer}}(l, m, A_1, A_2, R, T) \sqsubseteq C_{\mathsf{Transfer}}(l', m', A_1, A_2, R, T) \tag{6.6}$$

We can use these correctness criteria to prove the correctness of the abstract collecting semantics. But before that we need to define what the derivation trees of the abstract collecting semantics actually mean.

6.3.3 Abstract semantics trees

As in Schmidt [Sch95] we want to define abstract semantics trees co-inductively from the rules in Figure 6.5. This is to describe both finite and infinite derivation trees. If we define the trees only inductively we will limit ourselves to finite derivations. Then the analysis will fail to describe a lot of interesting recurring contracts.

Let the universe $\mathcal{U}_A$ be the set of ω-depth[1] finitely branching trees whose nodes are labelled by $D, m \triangleright c : \ell$ or by Δ (used when one tree might explore more subcontracts than another).

[1] Countably infinite depth

Figure 6.5 defines a set of rule schemes Φ that have the form

$$\frac{\{D, m_i \triangleright c_i : \ell_i\}_{i \in 1 \ldots n}}{D, m \triangleright op(c_i)_{i \in 1 \ldots n} : f(\ell_i)_{i \in 1 \ldots n}}$$

Like in Schmidt [Sch95] we define the corresponding functional of Φ, $\bar{\Phi}$ and take the well-formed abstract semantics trees to be $\mathrm{gfp}(\bar{\Phi})$. This includes both finite-depth derivations and ω-depth trees.

We say that the abstract semantics of the contract specification c in an abstract environment m is a $t \in \mathrm{gfp}(\bar{\Phi})$ such that $root(t) = D, m \triangleright c : \ell$ for some $\ell \in L$. Intuitively abstract semantics trees are built using the rules from the abstract collecting semantics, but can have possibly infinite paths.

Now we want to show soundness of the abstract semantics. This is simpler than in [Sch95], since we only have to consider finite derivations of the trace satisfaction semantics. For this we then avoid co-induction, and can prove soundness only using structural induction. We now want to show that abstract semantics trees accurately predicts satisfying traces:

Theorem 6.1. *If $\delta \vdash^D s : c$ by $\mathcal{H}$, $\delta\ \mathcal{R}_M\ m$ and we have a tree $t \in \mathcal{U}_A$ with $root(t) = D, m \triangleright c : \ell$ then $\beta(s) \sqsubseteq \ell$.*

Intuitively we want to show that the concrete trace satisfaction derivation is included in the abstract tree. In particular in the case of $+$, the trace satisfaction derivation has to make a choice whether to pick the left or the right alternative. The abstract tree is forced to explore both branches. This is where we use the property of $\sqcup$ to show that we preserve the over-approximation.

Proof. On the structure of $\mathcal{H}$.

S-Success It can only be the case that $root(t) = D, m \triangleright \mathsf{Success} : L_{\mathsf{Success}}$, so we need to show that $\beta(\langle\rangle) \sqsubseteq L_{\mathsf{Success}}$ which we have by requirement 6.1.

S-AltLeft We have derivation $\mathcal{H}_1$ of $\delta \vdash^D s : c_1$. It must be the case that t has sub-trees t_1, t_2 with $root(t_1) = D, m \triangleright c_1 : \ell_1$ and $root(t_2) = D, m \triangleright c_2 : \ell_2$. Now by the induction hypothesis on $\mathcal{H}_1$ and t_1 we have that $\beta(s) \sqsubseteq \ell_1$ and since L is a complete lattice, we have that $\beta(s) \sqsubseteq \ell_1 \sqcup \ell_2$.

S-AltRight This case is symmetric.

S-Concurrent We have a derivation $\mathcal{H}_1$ of $\delta \vdash^D s_1 : c_1$ and $\mathcal{H}_2$ of $\delta \vdash^D s_2 : c_2$. It must be the case that t has sub-trees t_1, t_2 with $root(t_1) = D, m \triangleright c_1 : \ell_1$ and $root(t_2) = D, m \triangleright c_2 : \ell_2$. Now by the induction hypothesis on $\mathcal{H}_1$ and t_1 we have that $\beta(s_1) \sqsubseteq \ell_1$ and similarly for $\mathcal{H}_2$ and t_2 we have that $\beta(s_2) \sqsubseteq \ell_2$. Now by requirement 6.3 we have what we require.

S-Sequence This case is similar to S-Concurrent.

S-Template We have a derivation $\mathcal{H}'$ of $\delta' \vdash^D s : c$ for a contract template $D(f) = (f[x_1, \ldots, x_n] = c)$ where $\delta' = \{x_1 \mapsto \mathcal{Q}[\![a_1]\!]^\delta, \ldots, x_n \mapsto \mathcal{Q}[\![a_n]\!]^\delta\}$. t has a sub-tree t' with $root(t') = D, [\![(a_1, x_1), \ldots, (a_n, x_n)]\!]^\sharp m \triangleright c : \ell$. By the induction hypothesis and the soundness requirement for the expression analysis we have what is required.

S-Transfer This proof is similar and the result follows from the induction hypothesis and the soundness requirement (6.4) of the $\mathsf{Transfer}$ combinator.

This concludes the proof. □

We also want to show that if we perform the analysis with a larger environment, then the analysis should remain sound. This is to be able to have a practical implementation of the analysis for recursive contracts with changing environments. This is only a property of the analysis, and does not have anything to do with the trace satisfaction semantics. Now we have to do some more work, since we are dealing with possibly infinite trees.

Like in [Sch95] we define a binary relation $\preceq_{\mathcal{U}_A} \subseteq \mathcal{U}_A \times \mathcal{U}_A$ as the largest binary relation satisfying the following properties:

- $t \preceq_{\mathcal{U}_A} t'$ if $t' = \Delta$.
- $t \preceq_{\mathcal{U}_A} t'$ if $root(t) = D, m \rhd c : \ell, root(t') = D, m' \rhd c : \ell'$, $m \sqsubseteq m'$, $\ell \sqsubseteq \ell'$ and for all sub-trees i of t there exists a sub-tree j of t' such that $t_i \preceq_{\mathcal{U}_A} t'_j$.

Informally this is a relation between trees such that if we explore them in the same way, then it will always be the case that the abstract trace and the abstract environment will be more precise in t than in t'. For instance the two trees below are related if $m \sqsubseteq m'$

$$\frac{[\![P]\!]^\sharp m = \bot}{D, m \rhd \mathsf{Transfer}(A_1, A_2, R, T \mid P).c : \bot} \preceq_{\mathcal{U}_A}$$

$$\frac{m'' = [\![P]\!]^\sharp m' \neq \bot \qquad \dfrac{\Delta}{D, m'' \rhd c : \ell}}{D, m' \rhd \mathsf{Transfer}(A_1, A_2, R, T \mid P).c : C_{\mathsf{Transfer}}(\ell, m'', A_1, A_2, R, T)}$$

since $\bot \sqsubseteq a$ for all $a \in L$. In fact this is also the only place where we lose precision, since we have the possibility of exploring less of the abstract trees when m is more precise. This is because whenever m is more precise, we could have that the predicate analysis can signal unsatisfiability of the predicate.

We can use the relation $\preceq_{\mathcal{U}_A}$ to state the soundness of widening:

Theorem 6.2. *If $m \sqsubseteq m'$, t_1, t_2 with $root(t_1) = D, m \rhd c : \ell_1$ and $root(t_2) = D, m' \rhd c : \ell_2$ then $t_1 \preceq_{\mathcal{U}_A} t_2$.*

Proof. We will not be super rigorous in proving this theorem, but in the same way as in [Sch95] can show that the relation on trees is closed, and the only remaining goal is by induction on c. The induction amounts to examining each rule scheme to verify that each pair of trees are related by $\preceq_{\mathcal{U}_A}$. In most cases there is only one way to construct the two trees, and the result will simply follow by the monotonicity of the combinators.

- Case $c = \mathsf{Success}$: There is only one way to construct these two trees, and $L_{\mathsf{Success}} \sqsubseteq L_{\mathsf{Success}}$.
- Case $c = \mathsf{Failure}$: There is only one way to construct these two trees, and $\bot \sqsubseteq \bot$.
- Case $c = c_1 + c_2$: This follows since $\sqcup$ is monotone.
- Case $c = c_1 \parallel c_2$: This follows since $C_{\parallel}$ is required to be monotone.

- Case $c = c_1; c_2$: This follows since $C_{;}$ is required to be monotone.
- Case $c = f(a_1, \dots, a_n)$: This follows since abstract expression evaluation is monotone.
- Case $c = \mathsf{Transfer}(A_1, A_2, R, T \mid P).c$: We have 4 ways to construct the two trees in this case.
  1. When $\ell_1 := \bot$ and $\ell_2 := \bot$ we have that $\bot \sqsubseteq \bot$.
  2. When $\ell_1 := \bot$ and $\ell_2 := C_{\mathsf{Transfer}}(\ell_2, m', A_1, A_2, R, T)$ we have that $\bot \sqsubseteq \ell_2$.
  3. When $\ell_1 := C_{\mathsf{Transfer}}(\ell_1, m, A_1, A_2, R, T)$ and $\ell_2 := \bot$ is vacuously true due to the requirements of the predicate analysis that a predicate now satisfiable in a large environment is definitely not satisfiable in a smaller environment.
  4. When $\ell_1 := C_{\mathsf{Transfer}}(\ell_2, m', A_1, A_2, R, T)$ and $\ell_2 := C_{\mathsf{Transfer}}(\ell_2, m', A_1, A_2, R, T)$ we have the statement of the theorem due to the monotonicity requirement of the predicate analysis and the monotonicity of C_{Transfer} combinator (requirement 6.6).

This concludes this proof, and we can say that every possible way to construct these two trees keeps $\ell_1 \sqsubseteq \ell_2$ which is exactly what we need. □

6.3.4 Computing a least solution

Now that we have a sound abstract semantics for CSL we will describe how to compute an analysis for a contract specification. For a contract specification c with templates D we have a tree $t \in \mathcal{U}_A$ such that $root(t) = D, \top \rhd c : \ell$.

In the case of a contract where we can write an finite abstract semantics tree (this is true in cases without recursion, or when the predicate analysis can infer termination), we can compute an analysis trivially by just unfolding the tree and reading off ℓ.

Example 6.3. *For the contract $c = \mathsf{Success} + \mathsf{Transfer}(A_1, A_2, R, T \mid P).\mathsf{Success}$ we can easily write the analysis as*

$$\frac{\emptyset, \top \rhd \mathsf{Success} : L_{\mathsf{Success}} \qquad \dfrac{[\![P]\!]^{\sharp}\top = m \qquad \emptyset, m \rhd \mathsf{Success} : L_{\mathsf{Success}}}{\emptyset, \top \rhd \mathsf{Transfer}(A_1, A_2, R, T \mid P).\mathsf{Success} : \ell}}{\emptyset, \top \rhd c : L_{\mathsf{Success}} \sqcup \ell}$$

with $\ell = C_{\mathsf{Transfer}}(L_{\mathsf{Success}}, m, A_1, A_2, R, T)$. It is easy to see that we can do this with all contracts without recursion.

For a contract with recursion the abstract semantics tree is possibly infinite, and we cannot simply unfold the tree to find a solution in finite time. Now if there on every infinite path is a node where we analyze the same contract with the same environment we can use a standard fixed point algorithm (with widening in the case of infinite abstract domains) to solve for the desired value. We can do this computation since we required all the combinators to be monotone. We will show how to do this by an example.

Example 6.4. *In an environment $D = \{f \mapsto f[] = \mathsf{Success} + f()\}$ we want to analyze $c = f()$. This not a very productive contract, but we want to be able to analyze all contracts.*

Our intuition is that this contract should have the behaviour L_{Success}, since only the empty trace satisfies this contract.

The idea is to return $\bot$ for the analysis of any repeating node in the tree for the first iteration. This means that the second time we encounter $f()$ with environment $\top$, we return $\bot$.

$$\dfrac{\dfrac{D, \top \triangleright \mathsf{Success} : L_{\mathsf{Success}} \qquad D, \top \triangleright f() : \bot}{D, \top \triangleright \mathsf{Success} + f() : L_{\mathsf{Success}} \sqcup \bot}}{D, \top \triangleright f() : L_{\mathsf{Success}}}$$

Now to reach a fixed point we now return L_{Success} for the second encounter of $f()$ in the second iteration of the algorithm.

$$\dfrac{\dfrac{D, \top \triangleright \mathsf{Success} : L_{\mathsf{Success}} \qquad D, \top \triangleright f() : L_{\mathsf{Success}}}{D, \top \triangleright \mathsf{Success} + f() : L_{\mathsf{Success}} \sqcup L_{\mathsf{Success}}}}{D, \top \triangleright f() : L_{\mathsf{Success}}}$$

We have reached a fixed point, so the analysis of c is L_{Success}. This corresponds to finding a least solution of ℓ to the equation $\ell = L_{\mathsf{Success}} \sqcup \ell$ where we can easily check that L_{Success} is a solution.

We still have a problem though. Consider the recursive contract

```
letrec f[x] = Success + Transfer(a1, a2, r, t | true).f(x-1) in f(0)
```

Now if we try to analyze this contract we see that there is an issue. If we keep track of the possible values of x, we always analyze the body of f in a different environment. The first time with $\{x \mapsto \{0\}\}$ then with $\{x \mapsto \{-1\}\}$ and so on. At some point when unfolding the tree we will see the infinite path:

$$\begin{array}{c} D, \{x \mapsto \{-2\}\} \triangleright f(x-1) : ? \\ \vdots \\ D, \{x \mapsto \{-1\}\} \triangleright f(x-1) : ? \\ \vdots \\ D, \{x \mapsto \{0\}\} \triangleright f(x-1) : ? \end{array}$$

where we never will have a repeating node. Here we can use environment widening to force a repetition along the path by widening the entry for x in the abstract environment to $\top$. We can do this after some fixed number of iterations. It might be when we encounter the contract for the second time or after some larger n to increase the precision of the analysis.

There is still one catch though. Whenever we have an infinite abstract domain like intervals, we are not guaranteed that the fixed point algorithm used in the example will terminate. It might be the case since we have a Transfer in the recursive contract that say adds something to the resulting interval, and does it for every iteration. Here we can apply the widening operator between iterations of the fixed point algorithm by using the iteration strategy f_{∇}^{n} from Section 5.3.1.

Note that the complexity of the analysis depends heavily on the precision of the predicate analysis. It is fair to assume that the more complicated the expression language is, the harder it is to analyze. Furthermore, if we are very precise, we might be able to infer termination of a contract, but also if we are precise, we might have to employ environment widening to ensure termination of the analysis. There are trade-offs in designing both a predicate analysis and an expression language for CSL.

6.4 An abstract interpreter

We are going to sidestep from the formal treatment of the analysis a little bit and look at how to implement an abstract interpreter that finds the least solution for any abstract semantics tree of the CSL analysis. We are going to use the approach taken by Darais, Labich, Nguyen and Van Horn [Dar+17] where they implement an abstract interpreter for a higher order functional language in Scheme. We use Haskell instead of Coq to not have to prove termination for the abstract interpreter. We will assume that we have all the abstract operations available (CSeq, CTrans, etc.) and that we can perform lattice operations (joining, widening).

We start by building a naive interpreter that will unfold the abstract semantics trees described in the previous section. The interpreter will read the abstract environment from a reader monad.

```
analyze ::
     (MonadReader (AEnv Value) m)
  ⇒ Contract → m AnalysisResult
...
analyze (Alternate c1 c2) = liftM2 join (analyze c1) (analyze c2)
analyze (Sequence c1 c2) = liftM2 Cseq (analyze c1) (analyze c2)
...
```

This of course does not terminate in the case of recursive templates. To be able to detect whether we have encountered a contract before we factor out the recursion, so that we can check for each recursive call whether we should abort and widen the environment or use the previously calculated result:

```
analyze ::
     (MonadReader (AEnv Value) m)
  ⇒ (Contract → m AnalysisResult)
  → Contract
  → m AnalysisResult
...
analyze f (Sequence c1 c2) = liftM2 Cseq (f c1) (f c2)
...
```

The rules are for the interpreter are directly translated from the abstract collecting semantics in Figure 6.5. To detect convergence we construct a map that caches earlier analysis results, and when we encounter them again we can just return them.

```
type SolveState a = Map (Contract, AEnv a) AnalysisResult
```

AEnv a corresponds to M, and AnalysisResult corresponds to L. We make a monad that we can read abstract environments and earlier solve states from. We add state that we can write the result of recently analyzed contracts to.

```
type AnalysisMonad a = ReaderT (AEnv a) (ReaderT (SolveState a) (State (SolveState a)))
```

We can use this monad to shortcut the recursion and return the earlier result if we already computed it. So instead of calling the analyzer recursively we use analyzeCache to detect convergence before calling analyze by mutual recursion:

```
analyzeCache :: Contract → AnalysisMonad Value AnalysisResult
analyzeCache c = do
  out ← get; env ← ask
  case Map.lookup (c, env) out of -- Have we already computed the result?
    Just r → return r
    Nothing → do
      cin ← askCacheIn
      let res = fromMaybe bottom (Map.lookup (c, env) cin) -- Get the old result
      modify (Map.insert (c, env) res) -- Update the result cache with the old result
      res′ ← analyze analyzeCache c -- Run the analyzer
      modify (Map.insertWith widen (c, env) res′) -- Insert the new result and apply widening
      return res′
```

This algorithm performs one step of the fixed point algorithm. We can initialize the result of all analyses of all contracts in all environments to be ⊥, and then analyze the contract until we find a fixed point on our lattice. We do with with the `fixCache` function

```
fixCache :: Contract → AnalysisMonad Agent AnalysisResult
fixCache ev c = do
  env ← ask
  ac′ ←
    mlfp
      (\ac → do
         put Map.empty -- Initialize the result cache
         localCacheIn ac (analyzeCache c) -- Update the reader cache
         get) -- return updated cache
  return (fromMaybe bottom (Map.lookup (c, cache env) ac′))
```

Here `mlfp` (monadic least fixed point) will iterate `analyzeCache` until it finds a fixed point by swapping out the caches. It initializes the state to be ⊥, and the reader to be the result of the last iteration by `localCacheIn`, and then it returns the resulting analyses by `get`. `mlfp` will terminate whenever nothing in the caches changes and we have found a fixed point.

We can change it to also perform environment widening by checking how many times we have encountered c with different environments, and we can then widen them and continue the analysis. We will not describe this here.

A prototype implementation of this can be found in the accompanying code in the folder `csl-impl`.

6.5 A general framework for contract analyses

Now that we have defined an abstract analysis for CSL we want to use it as a framework for defining concrete analyses. We want to infer properties of satisfying traces which is exactly what the abstract analysis does. Think of L as the set of properties.

Examples of important properties could be:

- Bob never transfers resources to Alice, because export restrictions prohibit him from doing so.
- Alice is not obliged to send more than what she has available on her bank account added to her credit limit.
- Charlie does not benefit too much from any series of events satisfying the contract.

It turns out that we can use the abstract collecting semantics of CSL to infer these properties. By instantiating the lattice, expression analysis and combinators we can get a specific analysis for CSL. We will now describe two simple instances that compute something relevant about a contract.

6.6 Participation analysis

One thing of interest to us is figuring out which agents can participate in a contract. We are interested in inferring a relation of the parties transferring resources. The intended meaning of the analysis is that if a pair of agents (a, b) is in the result, there may be a transfer of resources from a to b in any trace satisfying the contract specification. To make the interpretation of the result more clear we write $(a \to b)$ instead to emphasize that it is a transfer from a to b. For the analysis we only track the agent variables in the abstract environment.

$$L_p = \mathcal{P}(\mathcal{A} \times \mathcal{A}), \quad M_p = Var_{\mathsf{agent}} \to \mathcal{P}(\mathcal{A})$$

This participation analysis is of course very crude, and in a practical setting a much more precise statement is needed. We are interested in this only since it demonstrates the analysis framework very nicely. To be an analysis in the context of the abstract framework we need to define a representation function. In this case it accumulates all the agents in the events of a trace.

$$\beta(\mathsf{transfer}(a_1, b_1, r_1, t_1), \ldots, \mathsf{transfer}(a_n, b_n, r_n, t_n)) = \{(a_1, b_1), \ldots, (a_n, b_n)\}$$

The correctness of the analysis relies on the fact that β is a homomorphism with respect to append or interleavings and union.

Lemma 6.1. *If $s_1 +\!\!+\, s_2 = s$ or $(s_1, s_2) \rightsquigarrow s$ then $\beta(s) = \beta(s_1) \cup \beta(s_2)$.*

The analysis $\mathsf{Success}$ is simple since no one is communicating, so $L_{\mathsf{Success}} = \bot = \emptyset$.

For all the contract combinators we just union the results of the subcontracts $C_; = C_{\|} = \cup$, since in the case of sequential and parallel composition we know that all the transfers from both subcontracts will happen.

For $\mathsf{Transfer}$ we take all the possible values for the senders and receivers and we construct the cartesian product. This is exactly the over-approximation of the communicating agents that we are looking for.

$$C_{\mathsf{Transfer}}(l, m, a_1, a_2, r, t) = l \cup (m(a_1) \times m(a_2))$$

Remember that m contains a valid over-approximation of mappings from variables to sets of agents. We can prove the important correctness criteria pretty easily.

Lemma 6.2. *If $\delta[A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t]\ \mathcal{R}_M\ m \wedge \beta(s) \subseteq l$ then $\beta(\textsf{transfer}(a_1, a_2, r, t)\ s) \subseteq l \cup (m(A_1) \times m(A_2))$*

Proof. By the correctness of the abstract environment $a_1 \in m(A_1)$ and $a_2 \in m(A_2)$ and by the definition of β, $\beta(\mathsf{transfer}(a_1, a_2, r, t)\ s) = \{(a_1, a_2\} \cup \beta(s)$. Now since $(a_1, a_2) \in m(A_1) \times m(A_2)$ the statement of the lemma follows. □

The monotonicity of the combinators is trivial, since union is monotone. The proof for monotonicity of a $\mathsf{Transfer}$ is also pretty easy.

Lemma 6.3. *If $m \sqsubseteq m'$ and $l \subseteq l'$ then $l \cup (m(A_1) \times m(A_2)) \subseteq l' \cup (m'(A_1) \times m'(A_2))$*

Proof. By definition of $\sqsubseteq$ for environments and monotonicity of $\cup$. □

We will now show a short example of an analysis of a simple contract.

6.6.1 Example

Lets look at the escrow contract again:

```
letrec escrow[trusted, seller, buyer, goods, payment, deadline] =
  Transfer(buyer, trusted, payment, _).
    (Transfer(seller, buyer, goods, T | T < deadline).
     Transfer(trusted, seller, payment, T’ | True).Success
   + Transfer(trusted, buyer, payment, T | T > deadline).Success)

    in escrow("3rd", "shop", "alice", 1 bike, 1000 EUR, 2019-09-01)
```

We will just look at the body of the escrow template. It will be analyzed in an environment

$$m = \{trusted \mapsto \{"3rd"\}, seller \mapsto \{"shop"\}, buyer \mapsto \{"alice"\}\}$$

if we only consider agent variables. The template environment is D, which we will not write out. Now if we write up the analysis for the body of escrow, c where the two alternatives after the first transfer are c_1 and c_2 that has analyses ℓ_1 and ℓ_2 respectively.

$$\frac{m' = [\![P]\!]m \qquad \dfrac{\cdots}{D, m' \triangleright c_1 + c_2 : \ell_1 \cup \ell_2}}{D, m \triangleright c : (\ell_1 \cup \ell_2) \cup \{"alice" \rightarrow "3rd"\}}$$

And we can continue by calculating the results for $c_1 + c_2$ which we will not do here. At some point we will reach Success in all branches and we will be done. If we had recursion, we should apply fixed point techniques just like in Section 6.3.4.

6.6.2 Possible improvements

Consider a contract

```
Transfer(A, B, R, T | true).Transfer(C, D, R’, T’ | C = B ∧ D = A).Success
```

Then the result of the current analysis will be $\{(\top \rightarrow \top)\}$ meaning that anybody might transfer resources to anybody which is true. We can say something a bit more precise though. A possible improvement is introducing an existential for every unknown value when analyzing a transfer. Then we can identify whenever that variable is matched later. This can make it possible to make conditional statements like: If Alice is the sender of the first event, then she is the receiver of the second. This will require a more complex abstract domain, which we will leave for future work.

6.7 Fairness analysis

An analysis of much more practical interest than participation analysis is fairness analysis. In fairness analysis we are interested in analyzing the cost of participating in a contract for all agents. We will reuse the model of resources described in Section 2.6.

We are in essence trying to approximate the combined transfer of a trace $E(t) : \mathcal{A} \xrightarrow{\text{fin}} \mathcal{R}$ by just looking at the contract specification. For the analysis we choose an abstract domain $L_f = \mathcal{A} \rightarrow \mathcal{I}_{\mathbb{R}}$ that abstracts resources into intervals on the real number line. This means that the property that we are inferring is some bound on the value of a contract specification for the participating agents. To do this we are going to introduce the concept of a valuation. A valuation is a function $V : \mathcal{R} \rightarrow \mathbb{R}$ that computes some real valued measure of a particular resource. This measure is unitless, but one can think of it as the native currency of the party performing the analysis.

The valuations could vary depending on who performs the analysis. A German might value 1 EUR at 1 but a Dane might value it at 7.5. Also a customer could value an item differently than a seller, since there is a different between purchase and selling price.

To abstract traces onto this domain we are defining a representation function for a single transfer parameterized by the valuation function V:

$$\beta'_V(\mathsf{transfer}(a_1, a_2, r, t)) \\ = \begin{cases} \{a_1 \mapsto [-V(r), -V(r)], a_2 \mapsto [V(r), V(r)]\} & \text{when } a_1 \neq a_2 \\ \{a_1 \mapsto [0, 0]\} & \text{when } a_1 = a_2 \end{cases}$$

which is very similar to the effect of a $\mathsf{Transfer}$ defined in Section 2.6, but with intervals of valuations instead of resources. The representation function for an entire trace $s \in Tr$ will just map with this function and fold with interval addition $\oplus$.

$$\beta_V(s) = \text{fold}(\oplus, \{v \mapsto [0, 0] \mid v \in Var\}, \text{map}(\beta'_V, s)).$$

Example 6.5. *Consider the event trace from Section 2.6*

$$s = \langle \textsf{\textit{transfer}}(a, b, 10 \cdot \textit{DKK}, t_1), \textsf{\textit{transfer}}(b, a, 1 \cdot \textit{litres of milk}, t_2)\rangle$$

With the valuation $\{\textit{litres of milk} \mapsto 8, \textit{DKK} \mapsto 1\}$, then we can calculate

$$\beta_V(s) = \{a \mapsto [-2, -2], b \mapsto [2, 2]\}$$

assuming that $a \neq b$.

We are going to extend the valuation function to sets of resources by joining all of them together.

$$V_L(R) = \bigsqcup \{[V(r), V(r)] \mid r \in R\}.$$

The abstract environment will at least require a mapping from agent and resource variables to sets of agents or resources.

$$M_f = Var_{\text{agent}} \cup Var_{\text{resource}} \to \mathcal{P}(\mathcal{A}) \cup \mathcal{P}(\mathcal{R})$$

6.7.1 Combinators

We are now going to describe the combinators that will instantiate the abstract analysis framework. The analysis of the successful contract maps every agent to the singleton interval of 0 representing that nothing is transferred.

$$L_{\mathsf{Success}} = \{a \mapsto [0, 0] \mid a \in \mathcal{A}\}$$

This is different to $\perp$, where every agent would map to the empty interval. This choice makes intuitive sense, since no transfers will leave any agent with a net gain of 0.

In the case of sequential and parallel composition we know that both subcontracts are satisfied, so we can add all the intervals together making the analysis precise, so

$C_; = C_{\|} = \oplus$. We do not track the order of transfers, so the combinators are the same for ; and $\|$.

In the correctness of fairness analysis we again need a result relating appendings, interleavings and $\oplus$

Lemma 6.4. *If $s_1 +\!\!+ s_2 = s$ or $(s_1, s_2) \rightsquigarrow s$ then for all valuations V, $\beta(s) = \beta_V(s_1) \oplus \beta_V(s_2)$.*

Now the analysis of the transfer adds intervals for senders and receivers. There is a small catch though. If there is exactly one sender or receiver for the event, we can be precise. Otherwise we will have to widen to interval to include $[0, 0]$, since we do not know the actual agent. We make a helper function to a transfer of an abstract resource with respect to a single abstract agent:

$$V_{\mathsf{Transfer}}(A, R) = \begin{cases} \emptyset & \text{when } A = \emptyset \\ \{a \mapsto V_L(R)\} & \text{when } A = \{a\} \\ \{a \mapsto [0, 0] \sqcup V_L(R) \mid a \in A\} & \text{otherwise} \end{cases}$$

We can then use this to define the analysis of the $\mathsf{Transfer}$ with different signs for senders and receivers.

$$C_{\mathsf{Transfer}}(l, m, a_1, a_2, r, t) = l \oplus V_{\mathsf{Transfer}}(m(a_1), -m(r)) \oplus V_{\mathsf{Transfer}}(m(a_2), m(r))$$

We state the correctness requirement for this combinator by just substituting the definitions into the abstract framework.

Lemma 6.5. *If $\delta[A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t]\ \mathcal{R}_M\ m \wedge \beta(s) \sqsubseteq l$ then*

$$\beta(\mathsf{transfer}(a_1, a_2, r, t)\ s) \sqsubseteq l \oplus V_{\mathsf{Transfer}}(m(a_1), -m(r)) \oplus V_{\mathsf{Transfer}}(m(a_2), m(r))$$

We will not prove this lemma here, but we will do it later in the Coq formalization. Now we have what is required for this to be an analysis of CSL, and we can safely say that if we satisfy the specification with these combinators, then we will have a sound approximation of the combined transfer of a trace.

6.7.2 Example

One can verify pretty easily that we can achieve the analysis of the escrow contract claimed in the introduction. It is simply a matter of calculating the result from the abstract tree since it contains no recursion. We also look at the recursive contract:

```
letrec f[] =
   Transfer("alice", A, 1 USD, T | A = "bob" ∨ A = "charlie").(Success + f())
in f()
```

We analyze this contract with the valuation map $V = \{\text{USD} \mapsto 6\}$. By writing up the abstract tree we find that we need to solve the following equation for ℓ:

$$\ell = (L_{\mathsf{Success}} \sqcup \ell) \oplus \{\text{"}alice\text{"} \mapsto [6, 6], \text{"}bob\text{"} \mapsto [-6, 0], \text{"}charlie\text{"} \mapsto [-6, 0]\}$$

To try to solve this we apply the standard fixed point algorithm by setting $\ell_0 = \bot$ and iterating.

$$\begin{aligned} \ell_1 = (L_{\mathsf{Success}} \sqcup \ell_0) &\oplus \{\text{"}alice\text{"} \mapsto [6, 6], \text{"}bob\text{"} \mapsto [-6, 0], \text{"}charlie\text{"} \mapsto [-6, 0]\} \\ &= \{\text{"}alice\text{"} \mapsto [6, 6], \text{"}bob\text{"} \mapsto [-6, 0], \text{"}charlie\text{"} \mapsto [-6, 0]\} \end{aligned}$$

$$\begin{aligned}\ell_2 = (L_{\textsf{Success}} \sqcup \ell_1) \oplus \{\text{"}alice\text{"} \mapsto [6,6], \text{"}bob\text{"} \mapsto [-6,0], \text{"}charlie\text{"} \mapsto [-6,0]\}\\ = \{\text{"}alice\text{"} \mapsto [6,12], \text{"}bob\text{"} \mapsto [-12,0], \text{"}charlie\text{"} \mapsto [-12,0]\}\end{aligned}$$

$$\begin{aligned}\ell_3 = (L_{\textsf{Success}} \sqcup \ell_2) \oplus \{\text{"}alice\text{"} \mapsto [6,6], \text{"}bob\text{"} \mapsto [-6,0], \text{"}charlie\text{"} \mapsto [-6,0]\}\\ = \{\text{"}alice\text{"} \mapsto [6,18], \text{"}bob\text{"} \mapsto [-18,0], \text{"}charlie\text{"} \mapsto [-18,0]\}\end{aligned}$$

This will never terminate and find a fixed point due to the infinite abstract domain of intervals. We can apply point-wise widening to get a sound over-approximation

$$\ell_{\approx} = \{\text{"}alice\text{"} \mapsto [6,\infty], \text{"}bob\text{"} \mapsto [-\infty,0], \text{"}charlie\text{"} \mapsto [-\infty,0]\}.$$

We can check that it is a solution by substituting in into the original equation

$$\begin{aligned}\ell_{\approx} = (L_{\textsf{Success}} \sqcup \ell_{\approx}) \oplus \{\text{"}alice\text{"} \mapsto [6,6], \text{"}bob\text{"} \mapsto [-6,0], \text{"}charlie\text{"} \mapsto [-6,0]\}\\ = \{\text{"}alice\text{"} \mapsto [6,\infty], \text{"}bob\text{"} \mapsto [-\infty,0], \text{"}charlie\text{"} \mapsto [-\infty,0]\}\end{aligned}$$

6.7.3 Possible improvements

As described in Section 2.6, transfers have a net effect of 0. We do not exploit this in the formulation of fairness analysis described here. It would be very beneficial to be able to infer relationships between the amount of resources transferred for different agents. We will explore this a little bit in this section, but it will be fairly informal, and we will leave out a lot of technicalities.

Given that the transfers are two-way, it makes sense to have a domain that can relate two variables. In our case variables will be the expected gain of one agent in a contract. We write $eg(a)$ for the expected gain for agent $a \in \mathcal{A}$. For one particular transfer $\textsf{transfer}(a, b, r, t)$ we know that

$$eg(a) = -V(r) \text{ and } eg(b) = V(r)$$

Therefore also that

$$eg(a) + eg(b) = 0$$

This establishes a relation between a and b. We would like to exploit this somehow in the analysis of CSL. In the analysis of the transfer we now have to give a better implementation of $C_{\textsf{Transfer}}$. We know that from the abstract environment we get a set of possible agents for both the sender and the receiver of the event:

$$m(A_1) = \{a_1, \dots, a_n\}, m(A_2) = \{b_1, \dots, b_{n'}\}$$

We can also calculate a bound $V_L(m(R)) = [x, x']$ for the valuation of the resource. We can then write inequalities

$$eg(b_i) \leq x', eg(b_i) \geq 0$$

and

$$eg(a_i) \geq -x', eg(a_i) \leq 0$$

which are essentially what we also get with the intervals in the analysis. Similarly to the analysis we can make this more precise in the case where there is only one possible agent for each variable in the Transfer.

Now we can also exploit the fact that only one of the possible agents can be the sender and only one can be the receiver in the satisfying trace:

$$eg(a_1) + \ldots + eg(a_n) + eg(b_1) + \ldots + eg(b_{n'}) = 0$$

An abstract domain that can represent such constraints is the domain of convex polyhedra [CH78]. We will not explore this further, but we will leave it as future work to phrase fairness analysis with a more complex domain.

Chapter 7

Contract analyses in Coq

In this chapter we will briefly describe how we verified the correctness of the analysis specification in Coq using the mechanization of CSL described in Chapter 4. We have mechanized the abstract collecting semantics and mechanized the participation analysis and the fairness analysis. For the two concrete analyses, we have also mechanized some abstract domains which is described in Appendix B.2.

7.1 Generic analysis

To encode the abstract collecting semantics for contract analyses we use type classes to specify the requirements for an analysis of CSL. We start by defining a type class for the predicate analysis that also includes the analysis of arguments to templates. For the predicate analysis we require a `SetLattice` to map variables to. We will show some of the requirements here an an example. The remaining requirements are exactly the ones specified in Section 6.2.

```
Class PredicateAnalysis (A : ty → Type) ‘(L : SetLattice A) := {
    arg_eval : ∀ Δ τ, exp Δ τ → hlist A Δ → A τ;

    aenv_correct {Δ} (δ : env Δ) (m : hlist A Δ) :=
      ∀ τ (x : member τ Δ), In (hget δ x) (hget m x);

    arg_env_aenv := fix F {Δ ts} (m : hlist A Δ) (ae : arg_env ts Δ) : hlist A ts :=
      match ae with
      | HNil ⇒ HNil
      | HCons e ae’ ⇒ HCons (arg_eval e m) (F m ae’)
      end;

    analysis : ∀ Δ,
      hlist A (new_var Δ) → exp (new_var Δ) Bool → option (hlist A (new_var Δ));

    analysis_Some : ∀ Δ (p : exp (new_var Δ) Bool) m m’ δ,
      aenv_correct δ m ∧ analysis m p = Some m’ ∧ expDenote p δ = true → aenv_correct δ m’;

    (* Remaining requirements *)
}
```

The parameter `A : ty → Type` determines the abstract domain type of variables. `A t` is the abstract type of variables of type `t`. The `hlist A Δ` in the definitions is the abstract environment that encodes $M = Var \to A$. Note that with `arg_env_aenv` we provide an implementation for evaluating all arguments to a template application from the instance function `arg_eval` that just evaluates one. `aenv_correct` is the encoding of the relation $\mathcal{R}_M$. We can now prove a lemma about the correctness of evaluating all the arguments to a template just from the type class specification:

```
Lemma arg_env_aenv_correct A ‘{P : PredicateAnalysis A} :
  ∀ Δ (δ : env Δ) ts (ae : arg_env ts Δ) (m : hlist A Δ),
    aenv_correct δ m →
    aenv_correct (arg_envDenote δ ae) (arg_env_aenv m ae).
```

This definition of the predicate analysis is nice, since it makes it possible just to specify exactly what we need, and the rest can be left generic.

For the analysis specification we also create a type class that specifies the requirements of a CSL analysis. This requires an abstract domain for traces `L` and a predicate analysis over the abstract domain of variables `A`. The requirements are exactly the same as in Section 6.3.2.

```
Class CSLAnalysis (L : Type) (A : ty → Type) ‘(Lattice L) ‘(PredicateAnalysis A) := {
    β : trace → L;

    C_transfer : ∀ Δ, hlist A (new_var Δ) → L → L;

    β_transfer : ∀ Δ (δ : env Δ) (m’ : hlist A (new_var Δ)) l e t,
        aenv_correct (addEvent e δ) m’ →
        Incl (β t) l → Incl (β (cons e t)) (C_transfer m’ l);

    monotone_C_transfer : ∀ Δ (m m’ : hlist A (new_var Δ)) l l’,
        aenv_Incl m m’ → Incl l l’ → Incl (C_transfer m l) (C_transfer m’ l’)

   (* Remaining requirements *)
}
```

Here `aenv_Incl` is from the `PredicateAnalysis` class and `Incl` is from the `Lattice` class.

We can use this type class definition to encode the abstract collecting semantics from Figure 6.5. Like for abstract semantics trees we want to describe both finite and infinite derivations and like the definition for infinite trees in Section 3.2, we use the keyword `CoInductive` to include both finite and infinite derivations using the rules in the definition. We just show one case, and the rest are very similar to the rules on paper.

```
CoInductive csl_analysis L A ‘(CA : CSLAnalysis L A) :
∀ Γ Δ, contract Γ Δ → template_env Γ → hlist A Δ → L → Prop :=
(* Definitions for Success, Failure, Sequence, Alternative and Concurrent *)
| TransferASucc : ∀ Γ Δ (D : template_env Γ) (m : hlist A Δ)
                   (m’ : hlist A (new_var Δ)) c (p : exp (new_var Δ) Bool) l,
    analysis (add_event_aenv m) p = Some m’ →
    csl_analysis CA c D m’ l →
    csl_analysis CA (transfer p c) D m (C_transfer m’ l)
(* Rest of definitions *)
```

In the case for Transfer where the predicate analysis does not return $\bot$, we use `add_event_aenv` to extend the abstract environment with $\top$ for variables that are bound. We then use the `C_transfer` function to combine the result of the predicate analysis with the result of analyzing the subcontract. Since we require a type class instance for the CSL analysis, we have access to all the combinators and definitions about the predicate analysis as well.

We can prove that the co-inductive interpretation of the analysis given the generic analysis definition is sound with respect to the trace satisfaction semantics. This is essentially the same theorem as Theorem 6.1 and it is also proven by induction on the trace satisfaction derivation:

```
Theorem csl_analysis_sound L A ‘(CSLAnalysis L A) :
  ∀ Γ Δ (D : template_env Γ) (δ : env Δ) (m : hlist A Δ) (c : contract Γ Δ) l s,
  aenv_correct δ m ∧ csl_analysis c D m l ∧ csat δ D t c → Incl (β t) l.
```

We also prove that environment widening is sound, which corresponds to Theorem 6.2. There is one catch though. Co-induction is not widely used in the Coq community, so we had some problems proving this. We therefore only proved a weaker version by induction using the least fixed point interpretation of the rules instead (achieved by just replacing `CoInductive` with `Inductive`). This change means that the theorem is only valid for finite derivations of the abstract collecting semantics. This essentially means that the result is only valid for non-recursive contracts, or contracts where the expression analysis can infer termination.

```
Theorem env_widening_sound L A ‘(CSLAnalysis L A) :
  ∀ Γ Δ (D : template_env Γ) (m m’ : hlist A Δ) (c : contract Γ Δ) s s’,
    aenv_Incl m m’ ∧ ind_csl_analysis c D m s ∧ ind_csl_analysis c D m’ s’ → Incl s s’.
```

To prove the theorem using co-induction one would have to define the relation $\preceq_{\mathcal{U}_A}$ co-inductively and then show using the `cofix` tactic that if `aenv_Incl m m′` then the relation holds for abstract derivations in those environments. This is a variation on bi-similarity where we require inclusion instead of equality to hold at every node in the abstract tree. But where we in the case of bi-similarity of trees only needed two definitions in the co-inductive relation (one for leaves and one for nodes), we need to have one for every syntax constructor and combinations for the different ways of combining analyses for transfers.

In general the approach with type classes seems very nice, since we only need to focus on the important steps when defining a new analysis for CSL. We do not need to set up the entire machinery for every analysis. We will now briefly explain the different instantiations of the type classes that defines participation analysis and fairness analysis.

7.2 Analysis instances

The previous generic analysis is useless unless we show that there are implementations of the type classes. We now provide instances for the different type classes, and we start with the predicate analysis.

7.2.1 Predicate analysis

We claimed earlier that the identity analysis of predicates should be a valid analysis. We can define it as an instance of the predicate analysis class. We define the evaluation function for arguments by just looking up variables and abstracting literals. For all other expressions we just return ⊤.

```
Instance id_predicate_analysis_abstract_set :
  @PredicateAnalysis abstract_set _ abstract_set_setlattice := {
    arg_eval {Δ t} (e : exp Δ t) (m : value_map Δ) := match e with
      | Var v ⇒ hget m v
      | @Lit _ t l ⇒ ActualSet (singleton (tyDenote_dec_eq t) l)
      | _ ⇒ FullSet
      end;

    analysis _ m _ := Some m
  }.
  (* Correctness proofs go here *)
```

We have also started implementing the predicate analysis from Section 6.2.2. We have had some challenges with this given the dependently typed syntax. We want to define a function

```
Definition possibleValues Δ τ (old : value_map Δ) (e : exp Δ τ) : option (value_map Δ).
```

that given a abstract environment and an expression returns the possible mappings of variables as a more refined environment. This function then traverses the syntax of the predicate and unifies equalities using the function

```
Definition unify τ Δ (old : value_map Δ) : exp Δ τ → exp Δ τ → option (value_map Δ).
```

`unify` unifies values to variables and variables to variables using the rules in Figure 6.3. We have not completed the correctness proofs of these functions, mostly because of problems using the intrinsically typed syntax. In particular it is proving the following theorem that causes problems:

```
Lemma put_intersect_var_var : ∀ Δ (δ : env Δ) (m : value_map Δ) τ (v0 v1 : member τ Δ),
    value_map_correct δ m →
    hget δ v0 = hget δ v1 →
    value_map_correct δ (hput (intersect_abstract_set (hget m v0) (hget m v1))
                         (hput (intersect_abstract_set (hget m v0) (hget m v1)) m v0) v1).
```

Here we are given a sound environment and two variables that have that same value in the concrete environment. Then updating these two values to be their intersection should also be sound. Given that we think that this theorem holds, then the unification analysis is also sound. We leave it as future work to complete the correctness proof of this analysis.

7.2.2 Participant analysis

For the participation analysis we want an abstract domain of agent pairs. We start by defining a type for values that can be unknown:

```
Inductive abstract_value τ : Type :=
| AnyValue : abstract_value τ
| ActualValue : tyDenote τ → abstract_value τ.
```

It is either unknown, or it is a value from the expression language. We use this definition to define pairs of abstract agents:

```
Definition abstract_agent_pair := (abstract_value Agent * abstract_value Agent)%type.
```

We can use abstract pairs to encode sets like $\{(a, b) \mid b \in \mathcal{A}\}$ as $\{(a, \texttt{AnyValue})\}$. Inclusion is now a bit more complex since we have to check whether these special values are in there. We can define a lattice instance for pairs of abstract agents:

```
Instance aap_set_lattice : Lattice (set abstract_agent_pair) :=
  { top := singleton dec_eq_aap (AnyValue, AnyValue);
    bot := empty_set _;
    Incl := aap_set_Incl;
    join := set_union dec_eq_aap
  }.
```

We can reuse the standard definition of set union, but we have defined a new notion of inclusion for sets of abstract agents to check if one or two agents in a pair is unknown. The encoding of sets with abstract values also gives us a nice definition for $\top$, namely the set of abstract pairs $\{(\top, \top)\}$.

For the actual analysis instance, the definition of β is just a fold that extracts agents and adds them to the set. We prove that this function is a homomorphism with respect to both interleavings and appendings.

```
Lemma β_append : ∀ t1 t2 t l1 l2,
    appends t1 t2 t → aap_set_Incl (β_participation t1) l1 ∧ aap_set_Incl (β_participation t2) l2 →
    aap_set_Incl (β_participation t) (set_union dec_eq_aap l1 l2).
Lemma β_interleave : ∀ t1 t2 t l1 l2,
    interleave t1 t2 t → aap_set_Incl (β_participation t1) l1 ∧ aap_set_Incl (β_participation t2) l2 →
    aap_set_Incl (β_participation t) (set_union dec_eq_aap l1 l2).
```

For Transfer we write the definition of C_{Transfer} exactly as on paper. Note that the mechanization uses De Bruijn indices, so A_1 corresponds to `HFirst`, and A_2 corresponds to `HNext HFirst`:

```
Definition transfer_analysis Δ (m : value_map (new_var Δ)) l :=
  set_union dec_eq_aap (cross (hget m HFirst) (hget m (HNext HFirst))) l.
```

For correctness, we prove that it is correct with respect to the representation function. This is exactly the statement of Lemma 6.2.

```
Lemma transfer_correct ‘{@PredicateAnalysis _ _ abstract_set_setlattice} :
  ∀ Δ (δ : env Δ) (m’ : value_map (new_var Δ)) l e t,
        aenv_correct (addEvent e δ) m’ →
        aap_set_Incl (β_consent t) l →
        aap_set_Incl (β_consent (e :: t)) (transfer_analysis m’ l).
```

We also prove that is is monotone with respect to both the environment and the subcontract. This is exactly the statement of Lemma 6.3.

```
Lemma transfer_analysis_monotone ‘{@PredicateAnalysis _ _ abstract_set_setlattice} :
  ∀ (Δ : list ty) (m m’ : value_map (new_var Δ)) l l’,
    aenv_Incl m m’ → Incl l l’ → Incl (transfer_analysis m l) (transfer_analysis m’ l’).
```

This is a very technical proof that has a lot of cases due to the definition of abstract sets. This contrast a paper proof that is almost trivial.

For the definition of the analysis, we only require that the predicate analysis operates on power sets for agents, but for simplicity we just say that all of them are abstract sets.

```
Instance participant_analysis ‘{P : @PredicateAnalysis _ _ abstract_set_setlattice} :
  CSLAnalysis aap_set_lattice P := {
    β := β_consent;
    L_succ := bot;
    C_par := join; C_seq := join;
    β_transfer := transfer_analysis;
    β_transfer := transfer_correct;
    β_par := β_interleave;
    β_seq := β_append;
    monotone_C_par := participant_join_monotone;
    monotone_C_seq := participant_join_monotone;
    monotone_C_transfer := transfer_analysis_monotone
  }.
```

This instance declaration proves along with `csl_analysis_sound` and `env_widening_sound` that the participation analysis is sound.

7.2.3 Fairness analysis

For the fairness analysis we start by defining the abstract domain that the analysis is performed on. Just like in the definition on paper it is a map from agents to intervals. We describe this as a function from agent values to intervals:

```
Definition fairness_result := tyDenote Agent → interval.
```

Now we have an automatic lattice instance by combining the function lattice with the interval lattice. Note that we use integers instead of real numbers, because the support in Coq is much better. This is mainly a technicality. For now we define the valuation simply as a function from resources to integers parameterized by a direction:

```
Definition valuation (v : tyDenote Resource) (d : direction) : Z.
```

The direction determines whether to value the resource as a sender or a receiver

```
Inductive direction := Sender | Receiver.
```

The representation function is a function from traces to fairness results and is simply implemented as a right fold:

```
Definition β_fairness (t : trace) : fairness_result.
```

It uses addition on fairness results defined as element-wise addition:

```
Definition add_fr (f1 f2 : fairness_result) : fairness_result :=
  λ a ⇒ plus_interval (f1 a) (f2 a).
```

And the effect of a single transfer as a fairness result corresponding to β'_V in the paper description:

```
Definition ev_to_fr (e : event) : fairness_result.
  match e with
  | Event a1 a2 r _ ⇒
    λ a ⇒ if String.eqb a1 a2 then singleton_interval 0 else
       if String.eqb a a1 then singleton_interval (valuation r Sender) else
         if String.eqb a a2 then singleton_interval (valuation r Receiver) else
           singleton_interval 0
  end.
```

We provide the basis for the analysis, $L_{\textsf{Success}}$, which is the fairness result where all agents map to $[0, 0]$. This is the analysis of Success:

```
Definition success_fr : fairness_result := λ _ ⇒ singleton_interval 0.
```

The combinators for $+$ and $||$ will just be `add_fr`, so we prove that for both append and interleave, β is a homomorphism with respect to `add_fr`:

```
Lemma β_fairness_appends_add_eq : ∀ (t1 t2 t3 : trace) (a : tyDenote Agent),
    appends t1 t2 t3 →
    ∀ a, eq_interval ((β_fairness t3) a) ((add_fr (β_fairness t1) (β_fairness t2)) a).
Lemma β_fairness_interleaves_add_eq : ∀ (t1 t2 t3 : trace) (a : tyDenote Agent),
    interleave t1 t2 t3 →
    ∀ a, eq_interval ((β_fairness t3) a) ((add_fr (β_fairness t1) (β_fairness t2)) a).
```

Note that we have the required property of the representation function since `eq_interval` implies `Incl_interval`.

Now for the analysis of the Transfer, we translate the definition of $V_{\textsf{Transfer}}$ almost directly. The only change is that we include the direction of the transfer.

```
Definition value_transfer
  (sr : abstract_set Resource) (sa : abstract_set Agent) (d : direction) : fairness_result.
```

This definition is surprisingly complicated since we are dealing with an abstraction of finite sets, so there are a lot of cases to cover.

We now have the building blocks to analyze Transfer. Remember that for a

$$\textsf{Transfer}(A_1, A_2, R, T \mid P).c,$$

A_1 is encoded as `HFirst` and A_2 as `HNext HFirst` and so forth. We use the previous function to value the transfer both for possible senders and receivers:

```
Definition transfer_analysis Δ (m : value_map (new_var Δ)) : fairness_result :=
  let a1 := hget m HFirst in
  let a2 := hget m (HNext HFirst) in
  let r := hget m (HNext (HNext HFirst)) in
  add_fr (value_transfer r a1 Sender) (value_transfer r a2 Receiver).
```

We prove the required correctness statement

```
Lemma transfer_analysis_correct : ∀ Δ e (δ : env Δ) (m' : value_map (new_var Δ)) a,
    value_map_correct (addEvent e δ) m' →
    Incl_interval (ev_to_fr e a) (transfer_analysis m' a).
```

This proof is particularly nasty due to the enormous number of cases due to the complex definition of valuations. We have to take care that we distinguish the cases where an agent can be exactly one agent and all the other cases. In addition to the cases for actual sets we also have cases for unknowns. We could probably automate large parts of the proof, but we did not have time to explore it.

For monotonicity of the transfer, we want to prove the following lemma:

```
Lemma value_transfer_monotone : ∀ sa sa' sr sr' d a,
    abstract_set_Incl sa sa' →
    abstract_set_Incl sr sr' →
    Incl_interval (value_transfer sr sa d a) (value_transfer sr' sa' d a).
```

But we did not manage to complete it due to time constraints. Just like for `transfer_analysis_correct`, we have a very large number of cases to prove and a lot of them are manual due to having to work with `eq_interval` instead of regular equality in Coq.

With (almost) all the proofs completed, we can then provide an instance of the analysis framework for fairness analysis:

```
Instance fairness_analysis `{P : @PredicateAnalysis _ _ abstract_set_setlattice } :
  CSLAnalysis (@fun_lattice (tyDenote Agent) interval _) P :=
  {
    β := β_fairness;
    L_succ := success_fr;
    C_par := add_fr; C_seq := add_fr;
    C_transfer {Δ} m l := add_fr (transfer_analysis m) l
  }.
(* Proofs for correctness using the lemmas above *)
Defined.
```

This concludes the mechanization of the generic analysis framework and the two analyses. By providing class instances for the `CSLAnalysis` class we prove the correctness of both participation and fairness analysis.

Chapter 8

Discussion

In this chapter we are going to discuss to what degree we reached the objectives of the thesis. We are then going to discuss related work, and in the end suggest directions for future work.

8.1 Conclusion

The primary objective of this thesis was to build a theoretical framework necessary for the formal analysis of digital contracts. To achieve this we planned to do three things:

1. Mechanize the formal semantics of CSL using a proof assistant.

2. Build a theoretical framework for formal analysis of CSL.

3. Use the mechanized semantics of CSL to verify the correctness of the analysis method.

We achieved all these things to some degree, and we will now evaluate how well we reached the different objectives separately:

8.1.1 Mechanization of CSL semantics

We managed to complete the mechanization of the syntax and most of the semantics. The mechanization of CSL in Coq helped us figure out that the originally presented control semantics for CSL was not actually complete as originally stated in the paper. This indicates that it is somehow easier to fool a human that the Coq proof checker. This was a example of why proof assistants should be pervasive in all of computer science to give us confidence that proofs actually hold.

The choice of an intrinsic encoding for the mechanization seemed like a good idea, since the well-typedness of contracts carried all the way through to the analyses. This was very nice. Working with the intrinsic typed syntax was not equally pleasant everywhere, and especially pattern matching on dependently typed syntax is complicated in Coq. This made the analysis of expressions a bit hard to express.

Mechanizing the operational semantics of CSL was not strictly necessary for verifying the analysis, but we did it to "kick the tires" of the mechanization and make sure that it was actually useful for proving properties before using it to verify the analyses.

8.1.2 Analysis method for CSL

We find the abstract collections semantics elegant and it seems to capture some of the properties of CSL contracts that we are interested in. We have intentionally left the implementation of the analyses very open, and only stated what is necessary to have a correct analysis. It is still open whether the analysis method is actually useful for CSL. We have defined two very simple analyses, but it is not clear how well the analysis method will capture other properties of interest. The fairness analysis seems novel, and it is very useful to infer quantitative properties of contracts.

The analysis of expressions needs a lot more work to also capture relationships between variables, which is crucial for an effective analysis of predicates. A lot of properties that we are interested in are relational, so inferring relations from predicates is also quite important. We have intentionally not focused that much on expression analysis here since in a real implementation of CSL, the expression language is much more complicated, and the work done here would not be of much use.

One problem with the current algorithm for finding a solution to the analysis is that we interleave constraint generation and iteration strategy. Another approach might be to construct the constraints about the result of the analysis, and then use an external solver to find a solution. This is generally the approach taken by Nielson et al. [NNH99].

8.1.3 Correctness verification of the analysis

The encoding of the generic analysis as a type class in Coq seems very natural. We were able to specify the requirements of the analysis very concisely, and they were very easy to compare to those on paper. We had some problems with respect to co-induction in Coq though, and we think that it is due to the complex encoding with both intrinsic syntax and type class constraints that interferes with the progress checker of the `cofix` tactic, but we will have to explore this further. This left us with a weaker proof of soundness of environment widening.

There are also some really ugly proofs in the Coq mechanization in the case of intervals and the valuation of transfers. This is probably mainly due to lack of Coq programming experience, and more routine in developing large Coq proofs would probably have helped.

8.2 Related work

There is a huge body of work related to smart contract languages, but we are going to focus on related work that is relevant for this thesis. We are going to look at previous work on mechanizing contract languages and analysis of contract languages. We are also briefly going to look at certified abstract interpretation.

8.2.1 Mechanization of contract languages

KEVM In the work by Hildenbrandt et al. [Hil+18], the formal semantics of EVM, the virtual machine in the Ethereum blockchain, is defined using the $\mathbb{K}$-framework. From this mechanized semantics they are able to generate a reference interpreter for EVM with reasonable performance.

They are also able to use the specification of the formal semantics to construct a program verifier that can be used to verify the functional correctness of contracts and predict their gas usage.

Solidity in F* In this work by Bhargavan et al. [Bha+16] they translate Solidity contracts to F*, a programming language for program verification. They then show that it is possible to verify run-time safety and functional correctness of contracts in F*, and they guarantee that the correctness carries through when executed on the blockchain.

Certified Symbolic Management of Financial Multi-party Contracts In this work by Bahr et al. [BBE15] they present a multi-party contract language and mechanize it using the Coq proof assistant. Their mechanization allows formalizing domain-specific analyses and transformations. They also show that they can extract a Haskell implementation of the DSL from the Coq definitions along with certified contract management functionalities.

Scilla Scilla by Sergey et al. [SKH18] is an intermediate language for verified smart contracts. It models contracts as communicating automata. The automata makes transitions based on incoming messages.

Their main idea is that any in-contract computation does not involve any other parties. It is in some sense a pure computation. This distinction makes it possible to separate communication and computation and therefore make analysis and verification much simpler.

They leave their expression language free, and do not settle on any specifics. They disallow looping, but plan to support well-founded recursion to be able to prove termination statically. They have mechanized Scilla in Coq by a shallow embedding, where the language of in-contract computation is just Gallina (from Coq). They then translate Scilla contracts into Coq and verify properties of contracts manually.

Michelson The smart contract language of the Tezos blockchain [Goo14] has a mechanization in Coq[1]. The purpose of this mechanization is to prove properties of actual smart contracts in Coq. This is very similar to the approach of Scilla.

Plutus Core For the Cardano blockchain, their smart contract language Plutus compiles into Plutus Core, which is based on System F_ω. Plutus Core has a full mechanization in Agda[2] where they mechanize meta-theorems about progress and preservation.

Towards a Smart Contract Verification Framework in Coq In this work by Annenkov and Spitters [AS19] they explore an approach for verification of Oak smart contracts for the Concordium blockchain. They have developed a framework for developing the meta-theory of the smart contract language using a deep embedding and reasoning about the correctness of concrete contracts by a shallow embedding.

[1] https://gitlab.com/nomadic-labs/mi-cho-coq

[2] https://github.com/input-output-hk/plutus/tree/master/metatheory

8.2.2 Analysis of contract languages

Composing contracts In this seminal work by Jones et al. [Jon01] that CSL is based on, they translate compositional contracts to value processes. Value processes are partial functions from time to a random variable of some domain A. The details are rather math-heavy, but the idea is that this random variable represents the value of the contract at some time in the future. We see this as an automatic analysis method. This analysis is in use in the pricing and life-cycle management part of the LexiFi product[3] which is a portfolio management system for financial contracts.

Quantitative analysis of smart contracts In this work by Chatterjee et al. [CGV18], they provide a simplified programming language for smart contracts which can be translated to Solidity contracts. They then define a translation from smart contracts to state-based games. They use this formulation of contracts as games to analyze the expected payoff of participating in a contract. This in not unlike the fairness analysis that we have developed for CSL.

Static analysis of Ethereum contracts Due to the large number of security problems with Ethereum smart contracts, there has been a lot of work on static analysis of Ethereum smart contracts.

Luu et al. [Luu+16] developed a static analysis tool based on symbolic execution to flag possible security vulnerabilities in Ethereum smart contracts. Their tool flagged almost half of the contracts deployed at the time as vulnerable including some very well known vulnerable contracts.

Grossman et al. [Gro+17] developed an analysis to detect non-modular callbacks in Ethereum and shows that it could have been used to detect problems wrt. the DAO hack where callbacks were exploited to steal 150M $ in Ether.

Because of the unintuitive semantics of Ethereum smart contract, a lot of work has been put into verifying safety properties of contracts, and not into analysis of qualitative properties.

8.2.3 Certified abstract interpretation

Verasco In this major work by Jourdan et al. [Jou+15], they implement a static analyzer for C based on abstract interpretation in Coq. They reuse the formally verified C semantics from the CompCert project. They implement a large selection of advanced abstract domains, and prove the soundness of the analyzer with respect to the CompCert C semantics.

Certified Abstract Interpretation with Pretty-Big-Step Semantics In this work by Bodin et al. [BJS15] they develop certified abstract interpretation of an imperative language also based on the work of Schmidt [Sch95]. They use pretty-big-step semantics which is a variant of big-step semantics that aims to capture diverging programs. They also mechanize their work in Coq and prove soundness of their abstract interpreter.

[3] `https://www.lexifi.com/`

8.3 Future work

In this section we highlight 5 directions for future work. Some of them are theoretical and some of them are more practical.

8.3.1 Improvements of the abstract domains

As we hinted in the description of both of the concrete analyses, we are not as precise as we could be. There are domains which can express relationships between objects, so that we can make conditional statements about contracts like: "If Alice pays x dollars, then Bob will have to pay at least $2x$ dollars to the contract". We would like to explore the possibility of extending the analyses that we have defined to more complex domains that are more precise in these cases.

8.3.2 More analyses

One obvious direction of future work could be to develop more concrete analyses for CSL to investigate whether the framework is actually useful for a wide array of analyses. For instance it would be beneficial to also try to phrase analyses that infer temporal properties of contracts. All the analyses that we describe are non-temporal, meaning that the order of events do not matter for the final result. An example of a temporal property could be: "All obligations of Bob will be completed before Alice can participate in the contract".

8.3.3 Verified contract monitoring

One area of possible further investigation is the construction of a verified contract monitor. With a mechanized sound and complete reduction semantics, it should not be that complicated to construct a verified implementation of a contract monitoring system. It would basically be an interpreter for reducing contracts given events that would be proven to implement one of the reduction semantics.

8.3.4 Verified abstract interpretation of CSL

At the moment we have a prototype interpreter in Haskell that implements the abstract semantics of CSL. It would be great to have a verified abstract interpreter in Coq that guarantees that the result of the analysis is sound with respect to the semantics of CSL. There is one major problem with this. Termination of the abstract interpreter is hard to prove since it depends of properties of the lattice on which the analysis is performed. There has been work on data-flow analysis in Coq by Carchere et al. [Cac+05] where they prove termination of their analyzer using the ascending chain property of the analysis lattice.

8.3.5 Certification of CSL contracts

For high-risk contracts, it might be beneficial to verify and formally prove that a contract has certain properties. This approach is pursued in many other new contract languages, where one uses a program logic to prove properties about concrete contracts instead of relying on automatic inference of predetermined properties like we do. This is useful to prove that a contract conforms to a specification. This direction is motivated by the fact that a lot of the related work is within the domain of smart contract verification.

Bibliography

[And+06] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. “Compositional specification of commercial contracts”. In: *International Journal on Software Tools for Technology Transfer* 8.6 (2006), pp. 485–516. DOI: 10.1007/s10009-006-0010-1.

[AS19] Danil Annenkov and Bas Spitters. “Towards a Smart Contract Verification Framework in Coq”. In: *CoRR* abs/1907.10674 (2019). arXiv: 1907.10674. URL: http://arxiv.org/abs/1907.10674.

[AZW09] Brian Aydemir, Stephan A Zdancewic, and Stephanie Weirich. “Abstracting syntax”. In: *Technical Reports (CIS)* (2009), p. 901.

[BBE15] 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. Vancouver, BC, Canada: ACM, 2015, pp. 315–327. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784747. URL: http://doi.acm.org/10.1145/2784731.2784747.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer Berlin Heidelberg, 2004. DOI: 10.1007/978-3-662-07964-5.

[BD09] Ana Bove and Peter Dybjer. “Dependent Types at Work”. In: *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*. Ed. by Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 57–99. ISBN: 978-3-642-03153-3. DOI: 10.1007/978-3-642-03153-3_2. URL: https://doi.org/10.1007/978-3-642-03153-3_2.

[Ben+11] Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor McBride. “Strongly Typed Term Representations in Coq”. In: *Journal of Automated Reasoning* 49.2 (2011), pp. 141–159. DOI: 10.1007/s10817-011-9219-0.

[Ber06] Yves Bertot. “Coq in a Hurry”. In: *CoRR* abs/cs/0603118 (2006). arXiv: cs/0603118. URL: http://arxiv.org/abs/cs/0603118.

[Bha+16] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. “Formal Verification of Smart Contracts: Short Paper”. In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. PLAS ’16. Vienna, Austria: ACM, 2016, pp. 91–96. ISBN: 978-1-4503-4574-

3. DOI: 10.1145/2993600.2993611. URL: http://doi.acm.org/10.1145/2993600.2993611.

[BJS15] Martin Bodin, Thomas Jensen, and Alan Schmitt. "Certified Abstract Interpretation with Pretty-Big-Step Semantics". In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*. CPP '15. Mumbai, India: ACM, 2015, pp. 29–40. ISBN: 978-1-4503-3296-5. DOI: 10.1145/2676724.2693174. URL: http://doi.acm.org/10.1145/2676724.2693174.

[Cac+05] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. "Extracting a data flow analyser in constructive logic". In: *Theoretical Computer Science* 342.1 (2005), pp. 56–78. DOI: 10.1016/j.tcs.2005.06.004.

[CC92a] Patrick Cousot and Radhia Cousot. "Abstract interpretation and application to logic programs". In: *The Journal of Logic Programming* 13.2-3 (1992), pp. 103–179.

[CC92b] Patrick Cousot and Radhia Cousot. "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation". In: *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*. PLILP '92. Berlin, Heidelberg: Springer-Verlag, 1992, pp. 269–295. ISBN: 3-540-55844-6. URL: http://dl.acm.org/citation.cfm?id=646448.692441.

[CGV18] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. "Quantitative Analysis of Smart Contracts". In: *CoRR* abs/1801.03367 (2018). arXiv: 1801.03367. URL: http://arxiv.org/abs/1801.03367.

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic discovery of linear restraints among variables of a program". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '78*. ACM Press, 1978. DOI: 10.1145/512760.512770.

[Chl14] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, Feb. 11, 2014. 440 pp. ISBN: 0262026651.

[Dar+17] David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. "Abstracting definitional interpreters (functional pearl)". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), pp. 1–25. DOI: 10.1145/3110256.

[Goo14] LM Goodman. "Tezos—a self-amending crypto-ledger White paper". In: (2014).

[Gro+17] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. "Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 48:1–48:28. ISSN: 2475-1421. DOI: 10.1145/3158136. URL: http://doi.acm.org/10.1145/3158136.

[Hen+09] Fritz Henglein, Ken Friis Larsen, Jakob Grue Simonsen, and Christian Stefansen. "POETS: Process-oriented event-driven transaction systems". In: *The Journal of Logic and Algebraic Programming* 78.5 (2009), pp. 381–401. DOI: 10.1016/j.jlap.2008.08.007.

[Hil+18] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. "KEVM: A Complete Semantics of the Ethereum Virtual Machine". In: *2018 IEEE 31st Computer Security Foundations Symposium*. IEEE, 2018, pp. 204–217.

[Hvi10] Tom Hvitved. "A survey of formal languages for contracts". In: *Fourth workshop on formal languages and analysis of contract–oriented software*. Citeseer, 2010.

[Jon01] Simon Peyton Jones. "Composing Contracts: An Adventure in Financial Engineering". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2001, pp. 435–435. DOI: 10.1007/3-540-45251-6_24.

[Jou+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. "A Formally-Verified C Static Analyzer". In: *SIGPLAN Not.* 50.1 (Jan. 2015), pp. 247–259. ISSN: 0362-1340. DOI: 10.1145/2775051.2676966. URL: http://doi.acm.org/10.1145/2775051.2676966.

[Luu+16] 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. Vienna, Austria: ACM, 2016, pp. 254–269. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978309. URL: http://doi.acm.org/10.1145/2976749.2978309.

[McC82] William E. McCarthy. "The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment". In: *The Accounting Review* 57.3 (1982), pp. 554–578. ISSN: 00014826. URL: http://www.jstor.org/stable/246878.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 1999. ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6.

[Sch95] David A. Schmidt. "Natural-semantics-based abstract interpretation (preliminary version)". In: *Static Analysis*. Springer Berlin Heidelberg, 1995, pp. 1–18. DOI: 10.1007/3-540-60360-3_28.

[SKH18] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. "Scilla: a Smart Contract Intermediate-Level LAnguage". In: *CoRR* abs/1801.00687 (2018). arXiv: 1801.00687. URL: http://arxiv.org/abs/1801.00687.

Appendix A

Overview of the source code

Here we will briefly give an overview of the included source code.

First we describe the mechanization in Coq in the directory `csl-formalization`. The proofs are checked with version 8.9.1 of the Coq proof assistant on Linux. To check all the proofs, run `make` in the `CSL` folder. To interactively step through the proofs, we recommend Proof General[1], which is an interface for Coq based on the Emacs text editor.

A.1 The basic mechanization

The first couple of files are used to mechanize the semantics of CSL and proof properties of the semantics.

A.1.1 `CSL/HList.tex`

In this file we have the implementation of the `HList` that we use for environments.

A.1.2 `CSL/Definitions.tex`

In this file we have the definitions for domains, events and traces.

A.1.3 `CSL/Exp.tex`

This file includes the syntax for expressions and an interpreter for expressions. Futhermore it includes auxillary definitions for substitutions in expressions.

A.1.4 `CSL/Syntax.tex`

This file includes the syntax of CSL and auxillary definitions for substitutions in contracts.

A.1.5 `CSL/Satisfaction.tex`

This file includes definitions for interleave and append relations and the trace satisfaction semantics of CSL.

[1] `https://proofgeneral.github.io/`

A.1.6 `CSL/Tactics`

This file includes some tactics that are used in the proofs for the definitional interpreter and substitution.

A.1.7 `CSL/Denotational.tex`

This file includes functions that calculate interleavings and appendings and proofs of their correctness with respect to the relations used in the trace satisfaction semantics. It also includes the definitional interpreter for CSL and the proof of equivalence to the trace satisfaction semantics.

A.1.8 `CSL/Subst.tex`

This file includes the proof that substitution commutes with respect to both semantics.

A.1.9 `CSL/Guarded.tex`

This file includes both definitions for nullability and guardedness. It also includes proofs about the equivalence of semantic and syntactic nullability and that guardedness for template environments imply guardedness for all contracts.

A.1.10 `CSL/Residuation.tex`

This file includes the delayed matching and the eager matching semantics. It also includes soundness and completeness proofs for them. It also includes the definition of the control semantics from the paper, but without completed proofs.

A.1.11 `CSL/Examples.tex`

This file includes some example derivations of satisfying traces for contracts.

A.2 Verification of the analysis

The next couple of files include definitions we need for the analysis of CSL

A.2.1 `CSL/Analysis/Lattice.tex`

This file includes the type class definitions for lattices and the inferred instance for function lattices.

A.2.2 `CSL/Analysis/AbstractValues.tex`

This file includes data types for abstract values and abstract sets. These are used to give type class instances for abstract sets.

A.2.3 `CSL/Analysis/Interval.tex`

This file includes the implementation of the interval abstract domain along with a type class instance for lattices. It also includes a wide array of helpful lemmas about intervals that are used for proving the fairness analysis correct.

A.2.4 CSL/Analysis/Generic.tex

This file includes type class instances for both the predicate analysis and the CSL analysis. It also includes specifications of the inductive and co-inductive interpretation of the abstract collecting semantics for CSL. Finally it includes the soundness proofs for the abstract collecting semantics given the type class instances for both the CSL analysis and predicate analysis.

A.2.5 CSL/Analysis/IdPredicateAnalysis.tex

This file includes the trivial predicate analysis that does no refinement and proofs that it is correct.

A.2.6 CSL/Analysis/Algorithms.tex

This file includes an attempt at implementing an algorithm for predicate analysis that does unification. Proofs of its correctness is almost complete.

A.2.7 CSL/Analysis/Participation.tex

This file includes definitions of the abstract domain and representation function for participation analysis. It also includes the important proofs for the correctness of the combinators.

A.2.8 CSL/Analysis/ParticipationGeneric.tex

This file includes the type class instance for the participation analysis that proves that it is correct. It also includes the lattice instance for the abstract domain of the analysis.

A.2.9 CSL/Analysis/Fairness.tex

This file includes definitions of the abstract domain and representation function for fairness analysis. It also includes the important proofs for the correctness of the combinators. Finally it includes an incomplete proof of monotonicity of the transfer combinator.

A.2.10 CSL/Analysis/FairnessGeneric.tex

This file includes the type class instance for the fairness analysis that proves that it is correct.

A.3 Prototype implementation

We have also included the prototype implementation of an abstract interpreter for CSL. It is included in the `csl-impl` directory. It requires `stack`[2] for compiling and running.

When run, the program will analyze a few example contracts from the `Examples.hs` file.

[2] https://docs.haskellstack.org/en/stable/README/

Appendix B

Additional background

B.1 Lattices

We will in this section describe lattices, and the subset of lattice theory that we will use for the thesis.

Definition B.1 (Lattice). *A lattice is a set L equipped with a partial order $(\sqsubseteq)$ written $(L, \sqsubseteq)$. Furthermore we require that every pair (a, b) of elements has both a least upper bound and a greatest lower bound. We write $a \sqcup b$ for the least upper bound of two elements and $a \sqcap b$ for the greatest lower bound of two elements. These are also called join and meet respectively.*

Definition B.2 (Complete lattice). *A complete lattice is a lattice where every subset $S \subseteq L$ has a least upper bound (written $\bigsqcup S$) and a greatest lower bound ($\sqcap S$). We write $(L, \sqsubseteq, \bigsqcup, \sqcap)$ for a complete lattice. We write $\top = \bigsqcup L$ and $\bot = \bigsqcap L$ for the greatest and smallest element of the complete lattice respectively.*

A widely used complete lattice in abstract interpretation in the powerset lattice.

Example B.1. *The powerset lattice $(\mathcal{P}(A), \subseteq, \bigcup, \bigcap)$ is a complete lattice on any set where $\subseteq$ is ordinary set inclusion. We note that every subset $S \subseteq \mathcal{P}(A)$ has a least upper bound $\bigcup S$ and a greatest lower bound $\bigcap S$. We also note that $\top = \mathcal{P}(A)$ and $\bot = \emptyset$.*

Note also that $(\mathcal{P}(A), \supseteq, \bigcap, \bigcup)$ is a complete lattice.

Another complete lattice that is often used in analyses is the complete lattice on total functions. This lattice is used to model things like abstract values stored in variables.

Example B.2. *Let $(L', \sqsubseteq', \bigsqcup', \bigsqcap')$ be a complete lattice and let S be any set, now $(S \to L', \sqsubseteq, \bigsqcup, \bigsqcap)$ is also a complete lattice where*

$$
\begin{aligned}
f \sqsubseteq g &\iff \forall s \in S. f(s) \sqsubseteq' g(s) \\
(\bigsqcup X)(s) &= \bigsqcup{}'\{f(s) \mid f \in X\} \\
(\bigsqcap X)(s) &= \bigsqcap{}'\{f(s) \mid f \in X\}
\end{aligned}
$$

Note that $\bot(s) = \bot'$ and $\top(s) = \top'$.

We will also quickly describe the complete lattice that we will use in fairness analysis to bound gains and losses of a contract. It will be the intervals on the real number line.

Example B.3 (Intervals).

$$\mathcal{I}_{\mathbb{R}} = \{\bot\} \cup \{[x_1, x_2] \mid x_1 \leq x_2, x_1 \in \mathbb{R} \cup \{-\infty\}, x_2 \in \mathbb{R} \cup \{\infty\}\}$$

where the ordering is extended such that $-\infty < x$ and $x < \infty$ for all $x \in \mathbb{R}$. $i_1 \sqsubseteq i_2$ will be true iff any number in i_1 is also in i_2. $(\mathcal{I}_{\mathbb{R}}, \sqsubseteq, \sqcup, \sqcap)$ is a complete lattice where $\sqcup$ joins intervals and $\sqcap$ intersects intervals.

B.1.1 Ascending chains on lattices

For a lattice $(L, \sqsubseteq)$, a sequence $(\ell_n)_n$ of elements is an ascending chain if

$$n \leq m \Rightarrow \ell_n \sqsubseteq \ell_m$$

We say that the chain $(\ell_n)_n$ eventually stabilizes if and only if

$$\exists n_0. \forall n. n \geq n_0 \Rightarrow \ell_n = \ell_{n_0}.$$

Now we can define the Ascending Chain Condition. We say that a complete lattice $(L, \sqsubseteq)$ has the Ascending Chain Condition if and only if all ascending chains eventually stabilize.

B.1.2 Fixed points on complete lattices

We are going to compute fixed points of complete lattices, and to compute fixed points we are going to use monotone functions:

Definition B.3 (Monotonicity). *f is monotone if*

$$l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2)$$

If we have a monotone $f : L \to L$ on a complete lattice $(L, \sqsubseteq)$, then a fixed point of f is an element $l \in L$ such that $f(l) = l$. We write $\mathrm{Fix}(f) = \{l \mid f(l) = l\}$ for the set of fixed points of f on l. The least fixed point of f is the smallest of such fixed points.

$$\mathrm{lfp}(f) = \bigsqcap \mathrm{Fix}(f)$$

And the greatest fixed point of f is the largest of such fixed points.

$$\mathrm{gfp}(f) = \bigsqcup \mathrm{Fix}(f)$$

The Knaster Tarski shows the existence of fixed points on complete lattices:

Theorem B.1 (Knaster-Tarski Theorem for fixed points). *Let $(L, \sqsubseteq)$ be a complete lattice and $f : L \to L$ be a monotone function. Define*

$$m = \bigsqcap \{x \in L \mid f(x) \sqsubseteq x\}$$

and

$$m' = \bigsqcup \{x \in L \mid x \sqsubseteq f(x)\}.$$

Now m is the least fixed point of f and m' is the greatest fixed point of f.

Now one can show that

$$f^n(\bot) \sqsubseteq \bigsqcup_n f^n(\bot) \sqsubseteq \mathrm{lfp}(f) \sqsubseteq \mathrm{gfp}(f) \sqsubseteq \bigsqcap_n f^n(\top) \sqsubseteq f^n(\top).$$

If now L satisfies the Ascending Chain Condition, then one can also show that there exists n such that $f^n(\bot) = f^{n+1}(\bot) = \mathrm{lfp}(f)$. This will be useful for finding the solution to analyses of programs.

B.2 Lattices in Coq

For the abstract domain of traces, we are going to define a type class for lattices that we will use for all our analyses. It will not be a complete lattice, since we are not going to use ⊓, hence we formalize a complete upper semilattice.

```
Class Lattice (L : Type) :=
  {
    top : L; bot : L;
    Incl : L → L → Prop;
    Incl_bot : ∀ (x : L), Incl bot x;
    Incl_top : ∀ (x : L), Incl x top;
    Incl_refl : ∀ (x : L), Incl x x;
    Incl_trans : ∀ (x y z : L), Incl x y → Incl y z → Incl x z;
    join : L → L → L;
    join_correct : ∀ (x y : L), Incl x (join x y) ∧ Incl y (join x y);
    join_monotone : ∀ (x x' y y' : L), Incl x x' → Incl y y' → Incl (join x y) (join x' y')
  }.
```

For the abstract domain that we are going to use for the environment, we are going to subclass the lattice into a set-like lattice that has the possibility of lifting a single value into the lattice.

```
Class SetLattice (A : ty → Type) `{L : ∀ t : ty, Lattice (A t)} :=
  {
    In {t} : tyDenote t → A t → Prop;
    lift {t} : tyDenote t → A t;
    InIncl : ∀ t (v : tyDenote t) l, In v l ↔ Incl (lift v) l
  }.
```

This definition encodes the α function implicitly, so any value has a representation in the lattice automatically. The typeclass constraint for `L` requires us to give a lattice instance for all the return types of `A`. For instance we can define different abstractions for different types:

```
Definition good_env_denote (t : ty) :=
  match t with
  | Resource ⇒ interval
  | Timestamp ⇒ interval
  | _ ⇒ power_set
  end.
```

then `@SetLattice good_env_denote _` will make a set like lattice for the environment that has, say resource variables mapped to intervals and agent variables mapped to power sets. We will later use this to make the abstract environment defined as part of the predicate analysis.

This is a nice example of programming with dependent types in Coq that makes it possible to flexible definitions that can capture a wide array of use cases.

B.2.1 Instances

We will now describe the different lattice instances, and what they will be used for.

Functions

To model maps for use in the fairness analys we have made a lattice instance for total functions. The function lattice has a very nice definition in Coq using type classes where we only require the codomain to be a lattice.

```
Instance fun_lattice {A B} `(LB : Lattice B) : Lattice (A → B) :=
  {
    top := fun _ ⇒ top;
    bot := fun _ ⇒ bot;
    Incl x y := ∀ a, Incl (x a) (y a);
    join x y := λ a ⇒ join (x a) (y a)
  }.
```

Power set

There is a pretty large design space for encoding sets in Coq. To encode power sets we have chosen to lift finite sets by including an abstraction of the full set $\top$. Actual sets will be sets of values of the base type τ, tyDenote τ.

```
Inductive abstract_set t : Type :=
| FullSet : abstract_set t
| ActualSet : set (tyDenote t) → abstract_set t.
```

We can then make a lattice and set like instance for it.

```
Instance abstract_set_lattice t : Lattice (abstract_set t) :=
  {
    top := FullSet;
    bot := ActualSet (empty_set (tyDenote t));
    Incl x y := abstract_set_Incl x y;
    join x y := union_abstract_set x y
  }.
Instance abstract_set_setlattice : SetLattice abstract_set :=
  {
    In := abstract_set_In;
    lift {t} v := ActualSet (singleton (tyDenote_dec_eq t) v)
  }.
```

Intervals

We have described the implementation of intervals in Section 3.3. The lattice instance just uses the defintions from this library:

```
Instance interval_Lattice : Lattice interval :=
  {
    top := FullInterval;
    bot := EmptyInterval;
    Incl := Incl_interval;
    join := join_interval;
  }.
  (* Proofs of correctness *)
Defined.
```