A Formally Verified Static Analysis Framework for Compositional Contracts

Anonymous Authors[1]

[1]Unknown University
{anonymous1, anonymous2}@email.com

Abstract. A (commercial or financial) contract is a joint commitment to exchange resources such as money, goods, services amongst multiple parties. It expresses which actions may, must and must not be performed by its parties at which time, location and under which other conditions.
We present a general framework for statically analyzing *digital contracts*, formal specifications of such contracts. They are expressed in *Contract Specification Language* (CSL). Semantically, a CSL contract classifies traces of events into compliant (complete and successful) and noncompliant (incomplete or manifestly breached) ones.
Our analysis framework is a compositional abstract interpretation that soundly approximates the set of traces denoted by a contract by an abstract value in a lattice. The framework is parameterized by the lattice, the interpretation of contract primitives and combinators and requirements these must satisfy. It treats recursion by unrestricted unfolding. Employing Schmidt's natural semantics approach we interpret our inference system coinductively to account for infinite derivation trees and prove their sound abstract interpretation.
Finally, we show some example applications: participation analysis (who is possibly involved in a transfer to whom; who is possibly obliged to perform an action under some execution) and fairness analysis (bounds on how much is gained by each participant under any compliant execution of the contract).
The semantics of CSL, the abstract interpretation framework and its correctness theorem, the analyses and their proofs of satisfying the abstract interpretation framework requirements have been mechanized in the Coq proof assistant.

1 Introduction

Rising interest in distributed ledger technology spawned a subsequent increase in development of smart contract languages. Most ledger-specific smart contract languages are inherently resource-centric, with limited or no support for direct interaction between users as a first-class concept. A party participating in an exchange on a ledger can invoke some of the smart contract code (Ethereum) as the trusted orchestrator amongst them, or propose a change of state of the resource which is acceptable according to a given validation logic (Corda). More involved exchanges are of course possible to code, but they typically lack an explicit specification of the contractual rules behind them. Smart contract languages are typically Turing-complete (languages reminiscent of) full-fledged complex programming languages and are thus hard to analyze in principle and in practice.

In contrast to this, *Contract Specification Language* [2], used by Deon Digital [6] for specifying contracts in in a number of domains, is a relatively simple, CSP-like domain-specific language with few constructs for composing contracts from subcontracts.

We find it helpful to think of CSL as another layer of abstraction in a blockchain stack that supports installing user-defined protocols (contracts) that can then be managed by a choice of contract managers [7] in multiple ways. In programming language terms, contract managers are interpreters of digital contracts; specializing a particular contract manager with a fixed contract yields a specific smart contract in the programming language the particular contract manager is written in. Keeping contracts and contract managers separate supports portability of contracts, analysis of contracts without having to analyze the full language of the contract manager, it and facilitates multiple ways of executing the same contract by different contract managers, e.g. with or without support for regulatory auditing.

What kind of properties do we care about as a participant in a contract, assuming we have established certain level of trust in the security of platform on which the transaction will take place? If we are buying a CryptoKitties collectible, we might be interested in ensuring that if the transaction succeeds, the kitty is owned by us, and if it fails, we keep the ETH we would have paid for it. Depending on the complexity of the transaction, and of the language involved, it may or may not be an *obvious* property to verify.

To facilitate this style of analysis for any exchange of resources, we need to be more precise about what such an "exchange" can be. It is almost never just a single transfer, but multiple transfers; financial contracts such as derivatives, bonds, pensions, etc, can run over many years and involve combinations of options and payments depending on a complex set of conditions and choices. To fully describe a transaction we need to make certain assumptions not only about our own behaviour, but also about the behaviour of our counterparties. Agreeing on these assumptions constitutes specifying a *contract*, one that is perhaps closer to the legal sense of the word. Only once all participants agree on what the protocol of exchange— a contract —is, we are able to look at it critically in search for a loophole which allows our counterparty to gain take our Ether and keep the collectible we were about to buy from them.

We claim the following novel contributions.

- We provide a semantic framework for digital contract and a novel abstract interpretation framework for soundly analyzing contracts written in the contract specification language CSL, which includes support specifying contracts using general recursion.
- We provide illustrative analyses that represent important information and properties about a contract: Who is transferring resources to whom? Who may be involved (participate) in the contract? Is the contract always roughly fair (say under a mark-to-market valuation of all resources exchanged) under *any* execution? Is there somebody who may end up being obliged to do something even though they have never themselves signed up to the contract (clearly this is not legal, in practice such a contract would be stuck).
- We specify containment semantics, the abstract interpretation framework of CSL and formally verify the soundness of the general framework, as well as the correctness of presented example analyses, in the Coq proof assistant.

Our approach is based on CSL's containment semantics, which is formulated as a proof of compliance for a *complete* event trace. Intuitively, this is like asking only at the very end whether the events occurred as expected. In practice, CSL contracts are *monitored* on line, processing one event at a time. Here, we crucially exploit the powerful meta-theoretic property that the monitoring semantics and the containment semantics are

equivalent (proved and mechanized elsewhere). If we consider the monitoring semantics as the primary semantics, the containment semantics crucially provides a (co)induction principle that is fit for compositional analysis of contracts. It facilitates a powerful, but also deceptively simple way of formulating abstract interpretations and proving their soundness.

The remainder of the paper is organized as follows. In Section 2 we present a couple of examples of contracts in CSL and discuss the analysis we would like to perform on them. This informal presentation of CSL is followed by a proper introduction in Section 3. We then proceed to present our general framework of analysis and examples of concrete analyses of participation and fairness in Section 4. Details regarding Coq mechanization of the presented theory follow in Section 5. Section 6 concludes with related work and discussion of future work.

2 Preview

We begin by looking at a few multi-party contracts and the types of analyses we might be interested in applying to them.

The first example is a sales contract, where we use a trusted third party (escrow) to make sure that a seller of an item delivers it before receiving the payment. Here we first expect a payment from a buyer to the escrow. Then we have a choice of either delivering the bike and getting the money from the escrow before the deadline, or returning the money after the deadline. In CSL this can be written in the following slightly simplified way:

```
letrec sale[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 sale("3rd", "shop", "alice", 1 bike, 1000 EUR, 2019-09-01)
```

In this multiparty contract we are interested in the possible resource flows between the involved parties. For instance, we 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 the result of it is a relation between pairs of agents. For the escrow sale contract this relation is $R_p = \{(\texttt{3rd} \to \texttt{shop}), (\texttt{shop} \to \texttt{alice}), (\texttt{alice} \to \texttt{3rd}), (\texttt{3rd} \to \texttt{alice})\}$

We might also be interested in checking how much each agent can gain (or lose) by participating in the contract. *Fairness analysis* infers a set of parties and bounds on the utility of participating in the contract according. As an input to the analysis we provide a valuation function, which maps of a unit of some resource type to a real number representing its value in some base currency, for instance: $V = \{\texttt{bike} \mapsto 900, \texttt{EUR} \mapsto 1\}$. Looking at the contract, there are two possible outcomes. If the shop does not deliver the bike, neither the shop nor Alice have any gain or loss. If the shop delivers the bike, it gains 100 and Alice loses 100 because of the difference between value and purchase price. The result of the analysis we would like to obtain is $R_q = \{(\texttt{3rd}, [0, 0]), (\texttt{shop}, [0, 100]), (\texttt{alice}, [-100, 0])\}$.

The `sale` contract is fairly simple to analyze, since it does not contain any recursion or transfers with complicated acceptance conditions. However, things can quickly get harder, for instance if we look at this loan contract:

```
letrec repay[amount, interest, payments, from, to] =
   Transfer(from, to, R, _ | R = amount * payments + interest).Success
   + Transfer(from, to, R, _ | payments > 1 ∧ R = amount + interest).
     repay(amount, interest, payments - 1, from, to)

     in Transfer("bob", "alice", 1200 EUR, _) .
        repay(100 EUR, 10 EUR, 12, "alice", "bob")
```

The participation analysis is still easy, returning $R_p = \{(\texttt{bob} \to \texttt{alice}), (\texttt{alice} \to \texttt{bob})\}$. Analyzing the fairness is a bit more tricky, but it is still possible to infer that in this case, $R_q = \{(\texttt{alice}, [-120, -10]), (\texttt{bob}, [10, 120])\}$. Now let us combine combine `sale` with `repay` in the following way:

```
letrec sale[trusted, seller, buyer, item, payment, deadline] = ...
       repay[amount, interest, payments, from, to] = ...
in sale("3rd", "shop", "bob", 1 bike, 1000 EUR, 2019-09-01);
   (sale("3rd", "bob", "alice", 1 bike, 1 EUR, 2019-09-08)
   || repay(100 EUR, 5 EUR, 10, "alice", "bob"))
```

It may not be immediately obvious, but this is an extremely unfair contract, since the second `sale` contract (or both of them) may be canceled, and yet `alice` is obliged to pay back the `1000 EUR` (with interest!). In this case, the potential gains and losses of the contract participants are much more widespread:

$$R_q = \{(\texttt{shop}, [0, 100]), (\texttt{3rd}, [0, 0]), (\texttt{alice}, [-1050, -6]), (\texttt{bob}, [-94, 1050])\}.$$

These examples show that while contracts are compositional, their properties might not be. Indeed, cleverly combining two relatively fair contracts results in a contract where one of the parties can cheat the other. Our goal in this paper is to make it relatively easy to build analyses like the ones above.

3 Contract Specification Language

We now give a formal introduction to CSL, a domain-specific language for compositional contract specification. We note that the presentation of the language is limited to features required for the contract analysis introduced in the next section. For a more detailed overview, see Andersen et al. [2].

3.1 Syntax

CSL is used to describe possible interactions between *agents* exchanging *resources*. It supports *contract templates*, i.e. potentially mutually recursive contracts, which may further depend on a vector of formal parameters. A contract can therefore depend on both expression variables and contract template variables. We denote the context containing the former as Δ, and the latter as Γ. The basic syntax for contracts is given by the following grammar:

$$\begin{aligned} c &::= \mathsf{Success} \mid \mathsf{Failure} \mid c_1 + c_2 \mid c_1 \parallel c_2 \mid c_1; c_2 \mid \\ &\quad\ \mathsf{Transfer}(A_1, A_2, R, T \mid P).c \mid f(\boldsymbol{a}) \\ D &::= \{f_i[\boldsymbol{X_i}] = c_i\}_i \\ r &::= \mathsf{letrec}\ D\ \mathsf{in}\ c \end{aligned}$$

The first two constructs represent finished contracts: Success denotes the successfully completed contract, whereas Failure indicates an unfulfillable contract or a manifest contract breach. The following three are contract combinators: an alternative of executing contract c_1 or c_2 is expressed as $c_1 + c_2$; if the goal is to execute two contracts in parallel, $c_1 \parallel c_2$ is used; and finally, $c_1; c_2$ represents sequential composition of contracts. Next, $\mathsf{Transfer}(A_1, A_2, R, T \mid P).c$ is a *resource transfer* between two agents, the most basic form of resource exchange, indicating that agent A_1 is obliged to send resource R to agent A_2 at some time T such that the predicate[1] P is true; the contract then continues as c. Here A_1, A_2, R and T are binding occurrences of variables whose scope is P and c. The variables are bound when the contract is matched against a concrete event $e = \mathsf{transfer}(a_1, a_2, r, t)$. We use concrete values in place of binders to indicate equality constraints, e.g. $\mathsf{Transfer}(\mathtt{alice}, \mathtt{bob}, R, T \mid P).c$ is a shortening for $\mathsf{Transfer}(A, B, R, T \mid P \wedge A = \mathtt{alice} \wedge B = \mathtt{bob}).c$. $f(\boldsymbol{a})$ is an instantiation of a contract template named f with concrete arguments $\boldsymbol{a}$. Contract templates are collected in an environment $D = \{f_i[\boldsymbol{X_i}] = c_i\}_i$, where each c_i is a contract depending on formal arguments $\boldsymbol{X_i}$. Upon instantiation, these arguments become concrete values from the expression language. Lastly, contract c using a collection of contract templates D is written as $\mathsf{letrec}\ D\ \mathsf{in}\ c$.

$$\frac{}{\Gamma; \Delta \vdash \mathsf{Success} : \mathsf{Contract}} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{Failure} : \mathsf{Contract}}$$

$$\frac{(\Gamma; \Delta \vdash c_i : \mathsf{Contract})_{i=1,2} \quad \mathrm{op} \in \{+, \|, ;\}}{\Gamma; \Delta \vdash c_1\ \mathrm{op}\ c_2 : \mathsf{Contract}} \qquad \frac{f : \Delta' \to \mathsf{Contract} \in \Gamma \quad \Delta \vdash \boldsymbol{a} : \Delta'}{\Gamma; \Delta \vdash f(\boldsymbol{a}) : \mathsf{Contract}}$$

$$\frac{\begin{array}{l}(\Delta' = \Delta, A_1 : \mathsf{Agent}, A_2 : \mathsf{Agent}, R : \mathsf{Resource}, T : \mathsf{Time}) \\ \Gamma; \Delta' \vdash c : \mathsf{Contract} \qquad \Delta' \vdash P : \mathsf{Boolean}\end{array}}{\Gamma; \Delta \vdash \mathsf{Transfer}(A_1, A_2, R, T \mid P).c : \mathsf{Contract}}$$

$$\frac{(\Gamma; \Delta'_i \vdash c_i : \mathsf{Contract})_i \qquad \Gamma = \{f_i : \Delta'_i \to \mathsf{Contract}\}_i}{\vdash \{f_i[\boldsymbol{X_i}] = c_i\}_i : \Gamma}$$

$$\frac{\vdash D : \Gamma \qquad \Gamma; \cdot \vdash c : \mathsf{Contract}}{\vdash \mathsf{letrec}\ D\ \mathsf{in}\ c : \mathsf{Contract}}$$

Fig. 1. Well-formedness of contracts

Figure 1 presents a simple type system ensuring well-formedness of contracts. This type system relies on a typed expression language with a typing judgment $\Delta \vdash a : \tau$, which can be generalized to vectors of expressions: $\Delta \vdash \boldsymbol{a} : \Delta'$. In the remainder of this paper, we will assume all contracts are well-formed.

Events and traces. The execution of the interactions specified in a contract takes the form of a sequence of *events*, which are external to the specification. We typically refer to this sequence as a *trace*.[2] Since in CSL there is only one type of basic interaction between agents – described in the contract as $\mathsf{Transfer}(A_1, A_2, R, T \mid P).c$ – we

[1] "Predicate" in the sense of formula denoting a Boolean-valued function.

[2] Traces are just event sequences here.

accordingly have one type of events that can occur in a trace: $e ::= \mathsf{transfer}(a_1, a_2, r, t)$. A $\mathsf{transfer}(a_1, a_2, r, t)$ event indicates that agent a_1 has sent resource r to agent a_2 at a time t. A trace s is then a finite sequence of these events in the order in which they occurred. The language can be extended to support user-defined business events [1].

Expression language. CSL is parametric in the choice of the expression language; however, types Boolean, Agent, Resource and Time need to be present as those are used to decide whether an event $e = \mathsf{transfer}(a_1, a_2, r, t)$ is accepted by contract $\mathsf{Transfer}(A_1, A_2, R, T \mid P).c$. This is done by checking the value of expression P under assignment $\{A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t\}$. As the value of an expression may also depend on expression variables listed in context Δ, we need a concrete environment δ corresponding to it. We denote as $Q[\![a]\!]^\delta$ a mapping of expression a to a concrete value. For convenience, we write $\delta \models P$ if $Q[\![P]\!]^\delta = \mathsf{true}$ and $\delta \not\models P$ if $Q[\![P]\!]^\delta = \mathsf{false}$.

3.2 Contract Satisfaction

A CSL contract specifies the expected behaviour of participating parties. Above we have provided some informal intuitions for accepting a single event by matching it against a Transfer contract. Here we make these intuitions more formal, and generalize accepting a single event to a trace satisfying a contract. The complete rules of the contract satisfaction relation for traces are presented in Fig. 2.

$$\frac{}{\delta \vdash_D \epsilon : \mathsf{Success}} \qquad \frac{\delta \vdash_D s : c_1}{\delta \vdash_D s : c_1 + c_2} \qquad \frac{\delta \vdash_D s : c_2}{\delta \vdash_D s : c_1 + c_2}$$

$$\frac{\delta \vdash_D s_1 : c_1 \quad \delta \vdash_D s_2 : c_2 \quad (s_1, s_2) \rightsquigarrow s}{\delta \vdash_D s : c_1 \parallel c_2} \qquad \frac{\delta \vdash_D s_1 : c_1 \quad \delta \vdash_D s_2 : c_2}{\delta \vdash_D s_1\, s_2 : c_1 ; c_2}$$

$$\frac{Q[\![P]\!]^{\delta'} = \mathsf{true} \quad \delta' \vdash_D s : c \quad (\delta' = \delta, \{A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t\})}{\delta \vdash_D \mathsf{transfer}(a_1, a_2, r, t)\, s : \mathsf{Transfer}(A_1, A_2, R, T \mid P).c}$$

$$\frac{\boldsymbol{X} \mapsto \boldsymbol{v} \vdash_D s : c \quad f(\boldsymbol{X}) = c \in D \quad v = Q[\![\boldsymbol{a}]\!]^\delta}{\delta \vdash_D s : f(\boldsymbol{a})}$$

Fig. 2. Contract satisfaction

An empty trace (ϵ) satisfies a Success contract, matching the intuition that Success denotes a completed contract. To satisfy a contract offering an alternative $c_1 + c_2$, the trace must satisfy one of its components, c_1 or c_2, resulting in the next two rules. To satisfy a parallel composition of contracts $c_1 \parallel c_2$, trace s must be decomposed into s_1 and s_2, satisfying, respectively, c_1 and c_2. This decomposition may be an arbitrary interleaving, denoted by $(s_1, s_2) \rightsquigarrow s$. By contrast, in sequential composition $c_1 ; c_2$ we require that trace s is cut into two, $s = s_1\, s_2$, as c_1 must be satisfied before anything happens in c_2. Matching an event $\mathsf{transfer}(a_1, a_2, r, t)$ against contract $\mathsf{Transfer}(A_1, A_2, R, T \mid P).c$ is the crucial case of contract satisfaction. Concrete values a_1, a_2, r and t are provided for formal arguments A_1, A_2, R and T, respectively, which extend the existing concrete environment δ. In this extended environment, we check

that the expression P evaluates to true and that the remainder of the trace, s, satisfies contract c. Finally, for a trace to satisfy a contract template instantiation, we must change the concrete environment δ to be the evaluation of arguments $\boldsymbol{a}$ passed to the template f. We then check the definition of template f, and verify that indeed, trace s satisfies that contract.

4 Static Analysis

In this section we define a general framework for analysis of compositional contracts, and discuss requirements on its components that will guarantee the soundness of resulting analyses. We follow by providing some concrete instances: possible and definite participation in a contract and fairness analysis.

4.1 General Analysis Framework

CSL as a language can be decomposed into two components: contracts and a predicates. The former are a fixed, predefined set of operations that describe interactions between participants. The latter provide a basis to make decisions regarding acceptance or rejection of events submitted by participants. Naturally, analysis of compositional contracts specified in CSL will, correspondingly, consist of two parts.

The overall objective is of static analysis is to infer properties of a program (here: contract) without the need to "run" it on all inputs. This typically involves, among others, keeping track of how the abstract environment changes throughout execution. In CSL, the list of contracts one is static; there are no contract variables. However, the expression environment is affected by both incoming events, which introduce new binders and restrictions on values; as well as contract template calls, which alter the local environment. The expression analysis is used to make these changes and restrictions explicit.

In this section, we specify requirements on both predicate and contract analysis that guarantee the soundness of the analysis results with respect to the contract satisfaction relation (containment) defined in Section 3.2.

Predicate analysis. To capture bindings we require an abstract environment with abstract values $M : Var \to A$ with an abstraction function $\alpha : \mathcal{D} \to A$ from concrete values to abstract values. We often choose A to be the power-set lattice of values, in which case $\alpha(v) = \{v\}$. For two abstract environments m_1, m_2 we write $m_1 \sqsubseteq m_2$ iff $\forall x. m_1(x) \sqsubseteq m_2(x)$.

Whether to accept or reject an incoming event is determined by the predicate P in $\mathsf{Transfer}(X \mid P).c$. With a concrete environment δ, we can simply check whether $\delta \models P$ holds. Working with abstract values, we want to extract the restrictions on variables that make P evaluate to true, and use them to refine the abstract environment. We describe this transformation as a function $[\![P]\!]^\sharp : M \to M_\bot$. As the type suggests, this analysis also has the choice of returning $\bot$ to signal unsatisfiability, making the analysis much more precise when we can determine that a $\mathsf{Transfer}$ will never accept any events. We also require an abstract expression semantics for evaluating arguments to contract templates $[\![a]\!]^\sharp_m : A$, which in most cases is a simple lookup in m.

The properties that we require of a predicate analysis and abstract environment are gathered on Figure 3. They include relating abstract and concrete environment, as

$$\delta \sim m := \forall x \in Var.\alpha(\delta(x)) \sqsubseteq m(x) \tag{1}$$
$$\delta \sim m \wedge \delta \models P \to \delta \sim [\![P]\!]^\sharp(m) \tag{2}$$
$$\delta \sim m \wedge [\![P]\!]^\sharp(m) = \bot \to \delta \not\models P \tag{3}$$
$$m_1 \sqsubseteq m_2' \wedge [\![P]\!]^\sharp(m_1) = m_2 \neq \bot \wedge [\![P]\!]^\sharp(m_1') = m_2' \to m_2 \sqsubseteq m_2' \tag{4}$$
$$m \sqsubseteq m' \wedge [\![P]\!]^\sharp(m') = \bot \to [\![P]\!]^\sharp(m) = \bot \tag{5}$$

Fig. 3. Constraints for predicate analysis

specified in Equation 1, which we abbreviate $\delta \sim m$. When the abstract and concrete environments are related, we expect the abstract one to preserve the overapproximation when the predicate is satisfiable, as expressed by Equation 2. Similarly, we expect that if the predicate analysis signals unsatisfiability, then the predicate is indeed not satisfied, as given by Equation 3. Predicate analysis transformation $[\![P]\!]^\sharp$ should in general be monotone and failure-preserving, as specified by Equations 4 and 5. Similar requirements regarding over-approximation and monotonicity preservation can be stated for the $[\![a]\!]^\sharp$ function.

Depending on the choice of expression language, predicate analysis may get costly and complicated. It is therefore important to ensure that an "identity analysis", which performs no refinements, is an allowed instance we can use as the analysis of last resort. Most implementations will also rely on some form of unification for analysis of equality predicates, as in practice we often specify e.g. "the sender of the first event is the same, as the receiver of the second one".

Abstract collecting semantics. To define abstract collecting semantics for contract analysis, we begin with a complete lattice $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ describing properties of traces. We also need a representation function $\beta : Tr \to L$ mapping traces to the best properties describing them. We will use this representation function to later relate the abstract constraints to the trace satisfaction relation shown in Figure 2. Our goal is to define an analysis $[\![c]\!]^\sharp_m \in L$ describing all possible traces. In other words, the following is our approximation of soundness:

$$\forall s \in Tr, (\delta \vdash^D s : c) \wedge \delta \sim m \Rightarrow \beta(s) \sqsubseteq [\![c]\!]^\sharp_m \tag{6}$$

Since we want the analysis to be compositional, we need combination functions for $+$, ; and $\|$, which can only combine the results for subcontracts, $C_+, C_;, C_\| : L \times L \to L$. Further, to analyze contract $\mathsf{Transfer}(A_1, A_2, R, T \mid P).c$ we must combine the result for c with the result of analyzing P given the bound variables: $C_{\mathsf{Transfer}} : L \times M \times Var^4 \to L$. This time, the combinator might depend on the newly introduced bound variables, the result of the subcontract and the predicate analysis. We also require a designated lattice element $L_{\mathsf{Success}} \in L$ for the analysis of the successful contract.

The generic abstract collecting semantics for CSL can be seen on Figure 4. The analysis for both $C_;$ and $C_\|$ are left unspecified, however for C_+ we have no choice but to use the $\sqcup$ operator of the underlying lattice. We note that as we explicitly distinguish between the predicate analysis returning $\bot$ or a concrete value, we require that analysis to be decidable. There are some further restrictions on the relationship between β and the abstract collecting semantics, needed to reconstruct a valid Galois

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

$$\frac{}{D, m \triangleright \mathsf{Success} : L_{\mathsf{Success}}} \qquad \frac{}{D, m \triangleright \mathsf{Failure} : \bot}$$

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

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

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

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

Fig. 4. Abstract collecting semantics

connection from the representation function:

$$\beta(\langle\rangle) \sqsubseteq L_{\mathsf{Success}}$$
$$\beta(t_1) \sqsubseteq \ell_1 \wedge \beta(t_2) \sqsubseteq \ell_2 \to \beta(t_1 +\!\!+ t_2) \sqsubseteq C_{;}(\ell_1, \ell_2)$$
$$\beta(t_1) \sqsubseteq \ell_1 \wedge \beta(t_2) \sqsubseteq \ell_2 \wedge (t_1, t_2) \rightsquigarrow t \to \beta(t) \sqsubseteq C_{\parallel}(\ell_1, \ell_2)$$
$$\delta[A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t] \sim m \wedge \beta(s) \sqsubseteq \ell \to$$
$$\beta(\mathsf{transfer}(a_1, a_2, r, t)\, s) \sqsubseteq C_{\mathsf{Transfer}}(\ell, m, A_1, A_2, R, T)$$

Finally, we require all the C_{op}, as well as C_{Transfer} to be monotone. This allows using widening techniques for both the environment and trace approximations.

Infinite abstract trees. Before we discuss the soundness of our analysis, we have to think about what kind of derivation trees can we encounter when analyzing arbitrary contracts. While it is true that all concrete traces of any contract will be finite, the language still allows recursive contracts to be defined. This results in the possibility of constructing an infinite derivation tree using rules from Figure 4. To address this, we will now treat the $D, m \triangleright c : \ell$ judgment as coinductive.

Let $\mathcal{U}_A$ be the set of ω-deep, finitely branching trees with nodes labeled by either $D, m \triangleright c : \ell$ or Δ. We follow Schmidt [10] in defining the well-formed abstract semantic trees to be the greatest fixed point of a functorial $\bar{\Phi}$ corresponding to the judgments from Figure 4.

We then say that the abstract semantics of the contract specification c in an abstract environment $m \in M$ is a $t \in \mathsf{gfp}(\bar{\Phi})$ such that the root of the tree is a judgment: $root(t) = D, m \triangleright c : \ell$ 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.

Soundness. We can now state that abstract semantic trees soundly predict satisfying traces.

Theorem 1 (Soundness of approximation). *If* $\mathcal{H} :: \delta \vdash_D s : c$, $\delta \sim m$ *and we have a tree* $t \in \mathcal{U}_A$ *with* $root(t) = D, m \triangleright c : \ell$ *then* $\beta(s) \sqsubseteq \ell$.

Proof. Structural induction on the derivation of trace satisfaction, $\mathcal{H}$.

We again follow Schmidt [10] in our approach of defining a binary relation on trees, $\preceq_{\mathcal{U}_A} \subseteq \mathcal{U}_A \times \mathcal{U}_A$ as the largest binary relation satisfying:

- $t \preceq_{\mathcal{U}_A} t'$ if $t' = \Delta$.
- $t \preceq_{\mathcal{U}_A} t'$ if $root(t) = D, m \triangleright c : \ell, root(t') = D, m' \triangleright c : \ell'$, $m \sqsubseteq m'$, $\ell \sqsubseteq \ell'$ and for all subtrees i of t there exists a subtree 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, t will be more precise than t'.

Theorem 2 (Soundness of widening). *If* $m \sqsubseteq m'$, t_1, t_2 *with* $root(t_1) = D, m \triangleright c : \ell_1$ *and* $root(t_2) = D, m' \triangleright c : \ell_2$ *then* $t_1 \preceq_{\mathcal{U}_A} t_2$.

Proof. The relation $\preceq_{\mathcal{U}_A}$ on trees is closed; the remaining cases are by induction on c.

4.2 Example Analyses

We finish this section by showing some example instantiations of the framework. For space-efficiency reasons we omit the statements of required properties, as they are simply concretisations of the properties mentioned in the general framework description, this time with concrete lattices. The proofs of all these properties can be found in the accompanying Coq development.

Potential participation. We are interested in inferring a relation on the parties transferring resources. The intended meaning of the analysis is that if a pair of agents (a, b) is in the result, there might be a transfer of resources from a to b in some satisfying trace. For this analysis, the abstract environment will only track the agent variables: $L_c = \mathcal{P}(\mathcal{A} \times \mathcal{A})$, $M_c = Var_{\text{agent}} \to \mathcal{P}(\mathcal{A})$

The representation function β simply accumulates all the agents participating in the events of a given 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, interleaving and union.

Lemma 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 analyses of Failure and Success are simple, since in both cases no one is communicating, so $L_{\mathsf{Success}} = L_{\mathsf{Failure}} = \bot = \emptyset$. For all the contract combinators we just union the results of the subcontracts $C_{\mathsf{op}} = \cup$ for $\mathsf{op} \in \{+, ;, \|\}$, since in the case of choice we don't know statically which of the subcontracts will be satisfied. For Transfer we take all the possible pairs values for sender and receiver: $C_{\mathsf{Transfer}}(l, m, a_1, a_2, r, t) = l \cup (m(a_1) \times m(a_2))$. If we assume that the expression language only allow testing agents for equality we can use a simple unification algorithm for the predicate analysis.

Fairness. In this analysis we are interested in estimating the cost of participating in a contract for all agents. This time, the lattice is the total function lattice on the intervals on the real number line augmented with $\pm\infty$. The abstract environment this time is a mapping from variables to sets of agents or resources.

$$L_c = Var \to \mathcal{I}_{\mathbb{R}}, \quad M_c = Var_{\text{agent}} \cup Var_{\text{resource}} \to \mathcal{A} \cup \mathcal{R}$$

We will also need $V : \mathcal{R} \to \mathbb{R}$, a valuation function that computes value of a unit of any resource. We can extend it to sets of resources by joining the resulting singleton intervals:

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

Let $\oplus$ be addition on intervals, extended pointwise to maps. We make an entry with the negative value of the resource for the sender, and an entry with the value of the resource for the receiver.

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

The representation function is simply a fold over the trace, parameterized by valuation function V:

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

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

Lemma 2. *If $s_1 +\!\!+ s_2 = s$ or $(s_1, s_2) \rightsquigarrow s$ then for all valuations V, $\beta(s) = \beta_V(s_1) \oplus \beta_V(s_2)$.*

The analysis of the successful contract maps every agent to the singleton interval of 0, representing that nothing is transferred: $L_{\mathsf{Success}} = \{v \mapsto [0, 0] \mid v \in Var\}$. In the case of $+$ we have no other option than to join the intervals to accommodate both alternatives, $C_+ = \sqcup$. For sequential and parallel composition we know that both subcontracts are satisfied, so we can add all the intervals: $C_; = C_\| = \oplus$.

The Transfer analysis has to distinguish between two cases. If there is exactly one sender or one 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:

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

We can then use this to define the analysis of the Transfer:

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

Definite participation. Where in the first example we wanted to know about pairs of agents who *might* participate in the contract, here we want to calculate the set of agents who *definitely* participate as the sender.

Formally, agent a is definitely participating (as a sender) in contract c if for every trace s such that $\delta \vdash_D s : c$, there exist s_1, s_2, b, r, t such that $s = s_1\, \mathsf{transfer}(a, b, r, t)\, s_2$.

Like in the potential participation example, the abstract environment will only track the agent variables: $L_c = \mathcal{P}(\mathcal{A})$, $M_c = Var_{\text{agent}} \to \mathcal{P}(\mathcal{A})$. Interestingly, the representation function also has to be (almost) identical: $\beta(\mathsf{transfer}(a_1, b_1, r_1, t_1), \ldots, \mathsf{transfer}(a_n, b_n, r_n, t_n)) = \{a_1, \ldots, a_n\}$. This is of course a huge overapproximation, but indeed any agent who is definitely participating in the contract, will be captured by β. The requirement for C_+ to be the $\sqcup$ of the lattice gives away that, compared to the potential participation analysis, we will have to invert the ordering on the lattice to get the required structure. We can then set $C_;$ and $C_{\|}$ to be $\cup$ (which is $\sqcap$), and define the L_{Success} as $\emptyset$, or the $\top$ of the lattice.

The analysis for Transfer is, as usual, the most interesting. We only want to include a sender of a transfer in the result, if the predicate identifies them uniquely – in other words, if the abstract value corresponding to the sender is a singleton.

$$C_{\mathsf{Transfer}}(l, m, a_1, a_2, r, t) = \begin{cases} \{a_1\} \cup l & \text{when } m(a_1) \text{ is a singleton} \\ l & \text{otherwise} \end{cases}$$

5 Coq Mechanization

Both the trace semantics of CSL and the abstract collecting semantics have been mechanized in Coq proof assistant[3]. We have also mechanically verified the argument that the concrete analyses mentioned in the previous section are indeed correct instantiations of the general contract analysis framework. While the specifics of the implementation are best understood by looking at the code, this section provides a general overview of what – and how – has been mechanized.

5.1 Mechanized Semantics of CSL

The formalization of CSL uses dependently typed De Bruijn indices in the style of Benton et al. [3].

```
Inductive ty : Set := Agent | Resource | Timestamp | Bool.
Inductive contract (Γ : list (list ty)) (Δ : list ty) : Type
```

To represent a concrete environment, we use a heterogeneous list indexed by the corresponding typing environment. As the language of expressions we have picked for the mechanization is extremely simple, we can denote the base types using the corresponding Coq types. To capture contract templates, we again use heterogeneous lists.

```
Definition tyDenote (τ : ty) : Set := (...).
Definition env Δ := hlist tyDenote Δ.
Definition template_env Γ := hlist (contract Γ) Γ.
```

Traces are represented as lists of events of appropriate types (i.e. quadruples of concrete values). The trace satisfaction semantics from Figure 2 is encoded very naturally as an inductive definition.

```
Definition trace := list event.
Inductive csat :
  ∀ Γ Δ, env Δ → template_env Γ → trace → contract Γ Δ → Prop
```

[3] A zip file can be found here: https://gofile.io/?c=Az3a0K

5.2 Generic Analysis Framework

To implement the analysis framework as described in the previous section, we make use of Coq's type classes. We first define a type class describing requirements for predicate and template arguments' analysis.

```
Class PredicateAnalysis (A : ty → Type) `(L : SetLattice A)
```

Next, we define contract analysis relying on the predicate analysis being provided.

```
Class CSLAnalysis (L : Type) (A : ty → Type) `(Lattice L) `(PredicateAnalysis A)
```

Finally, we specify a coinductive type for the analysis, and prove its soundness, corresponding to Theorem 1.

```
CoInductive csl_analysis L A `(CA : CSLAnalysis L A) :
  ∀ Γ Δ, contract Γ Δ → template_env Γ → hlist A Δ → L → Prop := (...)

Theorem csl_analysis_sound L A `(CA : CSLAnalysis L A) :
  ∀ Γ Δ (D : template_env Γ) (δ : env Δ) (m : hlist A Δ) (c : contract Γ Δ) r t,
    aenv_correct δ m ∧ csl_analysis CA c D m r ∧ csat δ D t c → Incl (β t) r.
```

We also show that the environment widening is sound, corresponding to Theorem 2. This time we are using the inductive version of the CSL analysis type.

```
Inductive csl_analysis L A `(CA : CSLAnalysis L A) :
  ∀ Γ Δ, contract Γ Δ → template_env Γ → hlist A Δ → L → Prop := (...)
Lemma 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'.
```

Concrete analyses. The provided Coq sources contain three instances of the `CSLAnalysis` class, corresponding to the examples described in the previous section. Due to space considerations, we only give more details about the potential participation analysis.

One key difference between the definitions on paper and in the definitions in Coq is the formalization of sets. For the predicate analysis we use finite sets to describe analysis results. We use a minor generalization of sets to approximate the power-set domain of values indexed by the base type.

```
Inductive abstract_set τ : Type :=
| FullSet : abstract_set τ
| ActualSet : set (tyDenote τ) → abstract_set τ.
```

For this particular analysis, the abstract domain consists of pairs of agents. As we sometimes might not know anything about one of them, we must distinguish between concrete values and "any value" placeholders.

```
Inductive abstract_value τ : Type :=
| AnyValue : abstract_value τ
| ActualValue : tyDenote τ → abstract_value τ.
Definition abstract_agent_pair := (abstract_value Agent * abstract_value Agent).
```

We can then show that abstract sets form a lattice, and so do abstract values. With those instances at hand, we still need to show all the properties required by the contract analysis type class. Finally, the resulting declaration of `CSLAnalysis` instance can be given:

```
Program Instance participation_analysis
  `{P : ·PredicateAnalysis _ _ abstract_set_setlattice }
  : CSLAnalysis aap_set_lattice P :=
{
  β := β_participation;
  L_succ := bot;
  C_par := join; C_seq := join; C_transfer := (..);
  β_par := (..); β_seq := (..); β_transfer := (..);
  monotone_C_par := (..); monotone_C_seq := (..); monotone_C_transfer := (..);
}.
```

Note that we do not specify what concrete predicate analysis is used in this particular contract analysis, only what is its abstract domain.

6 Conclusion

In this paper we have outlined a framework for analysis of CSL contracts, showing a few small example instances. While these examples are relatively simple, we find the generality of this system promising enough to capture more complex contract properties, including ones of legal consequence, e.g. agent obligation in contracts, relative gains and losses of participants, recognition of single-sided contracts (i.e. ones where only one of the parties has any obligations remaining), etc. We found that using type classes in the mechanization of our framework makes it relatively easy to experiment with new analyses in a formal setting.

Given the difficulty of disallowing unintended behaviors of smart contracts after they have been deployed, including preventing unwanted chains of interactions, we feel like this work can contribute to the trustworthiness of smart contracts. By making the participant interaction layer separate and largely orthogonal to the underlying resource manager underneath it, we are able to focus on analyzing desired properties of interactions.

Related work. There is a rich literature on declarative contract languages going back 30 years, and more recent interest brought on by smart contracts, programs executing on a blockchain system, and their verification. A full literature will be provided in the final version of this paper.

Harz and Knottenbelt [8] provide an excellent overview of smart contract languages and techniques. Within their classification, CSL can be placed in the high-level language tier, closest perhaps to DAML [5], a modeling language by Digital Asset, and Marlowe [11], a Cardano ledger domain-specific language for financial contracts. In spirit, CSL is closer to Marlowe in that they both base the language on existing proposals ([9, 2]) from the financial and commercial domains. In execution, like DAML, it aims to be ledger-agnostic.

Most tools for analyzing smart contracts focus on security properties of Ethereum-style smart contracts, programs that manage account balances and are written in an expressive imperative language.

For quantitative analysis of existing contract languages, Chatterjee, Goharshady and Velner [4] present game-theoretic analysis of Solidity-like contracts, including expected payoffs. Compared to our more direct approach, a translation from programs to state-based games is required.

Future work. The directions for future investigation include finding more examples of properties to be verified using the proposed general technique, including analysis of temporal properties. One interesting case, briefly mentioned before, is the relational analysis of relative gains of contract participants: instead of estimating intervals of gains and losses for each participant independently, we would relate gains of one party relative to those of others. This would allow us to perform a more sophisticated fairness analysis.

We recognize that the style of analysis presented here has its limitations. While we can define an analysis of a universally-quantified property (definite participation can be one example), the approximation we get might not always be satisfactory. It might therefore be worth investigating a more direct approach, defining properties for whole sets of traces. Another limitation worth addressing in future developments is the inability to reason about failing traces. While we can quite often work around this caveat by using dual statements, we again risk loosing precision.

An orthogonal line of work is to get the existing analysis incorporated into Deon Digital's [6] contract specification language, a more expressive variant of CSL allowing, among other, for user-defined events beyond a simple Transfer.

References

1. J. Andersen, P. Bahr, F. Henglein, and T. Hvitved. Domain-specific languages for enterprise systems. In T. Margaria and B. Steffen, editors, *Proc. 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 8802 of *Lecture Notes in Computer Science (LNCS)*, pages 73–95. Springer-Verlag Berlin Heidelberg, 2014.
2. J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen. Compositional specification of commercial contracts. *STTT*, 8(6):485–516, 2006.
3. N. Benton, C.-K. Hur, A. J. Kennedy, and C. McBride. Strongly typed term representations in coq. *Journal of Automated Reasoning*, 49(2):141–159, Aug 2012.
4. K. Chatterjee, A. K. Goharshady, and Y. Velner. Quantitative analysis of smart contracts. *CoRR*, abs/1801.03367, 2018.
5. DAML. Digital Asset Modelling Language. `https://daml.com/`.
6. Deon Digital. CSL Language Guide. `https://deondigital.com/docs/v0.39.0/`.
7. B. Egelund-Müller, M. Elsman, F. Henglein, and O. Ross. Automated execution of financial contracts on blockchains. *Business & Information Systems Engineering*, 59(6):457–467, 2017. Nominated for Association of Information Systems Best Paper Award 2017.
8. D. Harz and W. J. Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods. *CoRR*, abs/1809.09805, 2018.
9. S. Peyton Jones, J.-M. Eber, and J. Seward. Composing contracts: An adventure in financial engineering (functional pearl). *SIGPLAN Not.*, 35(9):280–292, Sept. 2000.
10. D. A. Schmidt. Natural-semantics-based abstract interpretation (preliminary version). In *Static Analysis*, pages 1–18. Springer Berlin Heidelberg, 1995.
11. P. L. Seijas and S. J. Thompson. Marlowe: Financial contracts on blockchain. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, volume 11247 of *Lecture Notes in Computer Science*, pages 356–375. Springer, 2018.