Liquidity analysis in resource-aware programming

Silvia Crafa[1][0000−0003−0993−4734] and Cosimo Laneve[2][0000−0002−0052−4061]

[1] Department of Mathematics, University of Padova, Italy
silvia.crafa@unipd.it

[2] Department of Computer Science and Engineering, University of Bologna, Italy
cosimo.laneve@unibo.it

Abstract. Liquidity is a liveness property of programs managing resources that pinpoints those programs not freezing any resource forever. We consider a simple stateful language whose resources are assets (digital currencies, non fungible tokens, etc.). Then we define a type system that tracks in a symbolic way the input-output behaviour of functions with respect to assets. These types and their composition, which define types of computations, allow us to design an algorithm for liquidity whose cost is exponential with respect to the number of functions. We also demonstrate its correctness.

Keywords: Resource-aware programming · assets · liquidity · type systems · symbolic analysis.

1 Introduction

The proliferation of programming languages that explicitly feature resources has become more and more significant in the last decades. Cloud computing, with the need of providing an elastic amount of resources, such as memories, processors, bandwidth and applications, has pushed the definition of a number of formal languages with explicit primitives for acquiring and releasing them (see [1] and the references therein). More recently, a number of smart contracts languages have been proposed for managing and transferring resources that are assets (usually, in the form of digital currencies, like Bitcoin), such as the Bitcoin Scripting [5], Solidity [8], Vyper [10] and Scilla [13]. Even new programming languages are defined with (linear) types for resources, such as Rust [11].

In all these contexts, the efficient analysis of properties about the usage of resources is central to avoid flaws and bugs of programs that may also have relevant costs at runtime. In this paper, we focus on the *liquidity property*: a program is liquid when no resource remains frozen forever inside it, i.e. it is not redeemable by any party interacting with the program. For example, a program is not liquid if the body of a function does not use the resources transferred during the invocation by the caller. A program is also not liquid if, when it terminates, there is a resource that has not been emptied.

Functions	$F ::=$	$@\mathtt{Q}\ \mathtt{A}:\ \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}]\,\{\,S\,\} \Rightarrow @\mathtt{Q}'$
Prefixes	$P ::=$	$E \rightarrow \mathtt{x} \;\mid\; E \rightarrow \mathtt{A} \;\mid\; c \times \mathtt{h} \multimap \mathtt{h}' \;\mid\; c \times \mathtt{h} \multimap \mathtt{A} \quad // \, 0 \leqslant c \leqslant 1$
Statements	$S ::=$	$\_ \;\mid\; P\ S \;\mid\; \mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S\,\}\ S$
Expressions	$E ::=$	$v \;\mid\; X \;\mid\; E\,\mathtt{op}\,E \;\mid\; \mathtt{uop}\,E$
Values	$v ::=$	$c \;\mid\; \mathtt{false} \;\mid\; \mathtt{true}$

Table 1. Syntax of *Stipula* (X are assets, fields and parameters names)

We analyze liquidity for a simple programming language, a lightweight version of *Stipula*, which is a domain-specific language that has been designed for programming *legal contracts* [7]. In *Stipula*, programs are *contracts* that transit from state to state and a control logic specifies what functionality can be invoked by which caller; the set of callers is defined when the contract is instantiated. Resources are assets (digital currencies, smart keys, non-fungible tokens, etc.) that may be moved with ad-hoc operators from one to another.

Our analyzer is built upon a type system that records the effects of functions on assets by using symbolic names. Then a *correctness* property, whereby the (liquidity) type of the final state of a computation is always an over-approximation of the actual state, allows us to safely reduce our analysis arguments to verifying if liquidity types of computations have assets that are empty.

We identify the liquidity property: *if an asset becomes not-empty in a state then there is a continuation where all the assets are empty in one of its states.* The algorithm for assessing liquidity looks for computations whose liquidity types $\Xi \rightarrow \Xi'$ are such that the contract's assets in Ξ' are empty. The crucial issue of the analysis is termination, given that computations may be *infinitely many* because contracts may have cycles. For this reason we restrict to computations whose length is bound by a value. Actually we found more reasonable computations where every function can be invoked a bounded number of times. It turns out that the computational cost of the algorithm is exponential with respect to the number of functions.

The structure of the paper is as follows. The lightweight *Stipula* language is introduced in Section 2 and the semantics is defined in Section 3. Section 4 reports the theory underlying our liquidity analyzer and Section 5 illustrates the algorithm for verifying liquidity. We end our contribution by discussing the related work in Section 6 and delivering our final remarks in Section 7. Due to page limits, the technical material has been omitted and can be found in the full paper.

2 The *Stipula* language

We use disjoint sets of names: *contract names*, ranged over by C, C′, $\cdots$; names referring to digital identities, called *parties*, ranged over by A, A′, $\cdots$; *function names* ranged over by f, g, $\cdots$; *asset names*, ranged over by h, k, $\cdots$, to be used both as contract's assets and function's asset parameters; *non asset names*, ranged over by x, y, $\cdots$, to be used both as contract's fields and function's non asset parameters. Assets and generic contract's fields are syntactically set apart since they have different semantics; similarly for functions' parameters. Names of assets, fields and parameters are generically ranged over by X. Names `Q`, `Q'`, $\cdots$ will range over contract states. To simplify the syntax, we often use the vector notation $\overline{x}$ to denote possibly empty *sequences* of elements. With an abuse of notation, in the following sections, $\overline{x}$ will also represent the *set* containing the elements in the sequence.

The code of a *Stipula* contract is [3]

$$\texttt{stipula C} \; \{ \; \texttt{parties} \; \overline{\texttt{A}} \quad \texttt{fields} \; \overline{\texttt{x}} \quad \texttt{assets} \; \overline{\texttt{h}} \quad \texttt{init Q} \quad \overline{F} \; \}$$

where `C` identifies the *contract name*; $\overline{\texttt{A}}$ are the *parties* that can invoke contract's functions, $\overline{\texttt{x}}$ and $\overline{\texttt{h}}$ are the *fields* and the *assets*, respectively, and `Q` is the *initial state*. The contract body also includes the sequence $\overline{F}$ of functions, whose syntax is defined in Table 1. It is assumed that the names of parties, fields, assets and functions do not contain duplicates and functions' parameters do not clash with the names of contract's fields and assets.

The declaration of a function highlights the state `Q` when the invocation is admitted, who is the caller party `A`, and the list of parameters. Function's parameters are split in two lists: the *formal parameters* $\overline{\texttt{y}}$ in brackets and the *asset parameters* $\overline{\texttt{k}}$ in square brackets. The *body* $\{ \; S \; \} \Rightarrow \texttt{@Q}'$ specifies the *statement part* S and the state `Q'` of the contract when the function execution terminates. We write $\texttt{@Q}\,\texttt{A} : \texttt{f}(\overline{\texttt{y}})\,[\overline{\texttt{k}}]\; \{ S \} \Rightarrow \texttt{@Q}' \in \texttt{C}$ when $\overline{F}$ contains $\texttt{@Q}\,\texttt{A} : \texttt{f}(\overline{\texttt{y}})\,[\overline{\texttt{k}}]\; \{ S \} \Rightarrow \texttt{@Q}'$ and we will often shorten the above predicate by writing $\texttt{Q}\;\texttt{A.f}\;\texttt{Q}' \in \texttt{C}$. *Stipula* does not admit *internal nondeterminism*: for every `Q`, `A` and `f`, there is at most a `Q'` such that $\texttt{Q}\;\texttt{A.f}\;\texttt{Q}' \in \texttt{C}$.

Statements S include the empty statement $\_$ and different prefixes followed by a continuation. Prefixes P use the two symbols $\rightarrow$ (*update*) and $\multimap$ (*move*) to differentiate operations on fields and assets, respectively. The prefix $E \rightarrow \texttt{x}$ updates the field or the parameter `x` with the value of E – the old value of `x` is lost – it is *destroyed*; $E \rightarrow \texttt{A}$ sends the value of E to the party `A`. The move operations $c \times \texttt{h} \multimap \texttt{h}'$ and $c \times \texttt{h} \multimap \texttt{A}$ define actions that *never destroy resources.* In particular, $c \times \texttt{h} \multimap \texttt{h}'$ subtracts the value of $c \times \texttt{h}$ to the asset `h` and adds this value to `h'`, where c is a constant between 0 and 1. Notice that, because of this constraint, $c \times \texttt{h}$ is always smaller or equal to `h` – therefore assets never have negative values. It is also worth to notice that, according to the syntax, the right-hand side of $\rightarrow$ in $E \rightarrow \texttt{x}$ is always a field or a non-asset function parameter, while the right-hand side of $\multimap$ in $c \times \texttt{h} \multimap \texttt{h}'$ is always an asset (the left-hand side is an expression that indicates part of an asset). The operation $c \times \texttt{h} \multimap \texttt{A}$ subtracts the value of $c \times \texttt{h}$ to the asset `h` and transfers it to `A`.

Statements also include *conditionals* $\texttt{if}\;(E)\;\{ \; S \; \}\;\texttt{else}\;\{ \; S' \; \}$ with the standard semantics. In the rest of the paper we will always abbreviate $1 \times \texttt{h} \multimap \texttt{h}'$ and $1 \times \texttt{h} \multimap \texttt{A}$ (which are very usual, indeed) into $\texttt{h} \multimap \texttt{h}'$ and $\texttt{h} \multimap \texttt{A}$, respectively.

Expressions E include constant values v, names X of either assets, fields or parameters, and both binary and unary operations. Constant values are

[3] Actually this is a lightweight version of the language in [7].

– real numbers n, that are written as nonempty sequences of digits, possibly followed by "`.`" and by a sequence of digits (*e.g.* `13` stands for `13.0`). The number may be prefixed by the sign `+` or `-`. Reals come with the standard set of binary arithmetic operations (`+`, `-`, $\times$) and the unary division operation E/c where $c \neq 0$, in order to avoid 0-division errors.
– boolean values `false` and `true`. The operations on booleans are conjunction `&&`, disjunction `||`, and negation `!`.
– asset values that represent *divisible* resources (*e.g.* digital currencies). Divisible asset constants are assumed to be identical to positive real numbers (assets cannot have negative values).

Relational operations (`<`, `>`, `<=`, `>=`, `==`) are available between any expression.

The standard definition of *free names* of expressions, statements and functions is assumed and will be denoted $fn(E)$, $fn(S)$ and $fn(F)$, respectively. A contract `stipula C` $\{$ `parties` $\overline{\texttt{A}}$ `fields` $\overline{\texttt{x}}$ `assets` $\overline{\texttt{h}}$ `init Q` $\overline{F}$ $\}$ is *closed* if, for every $F \in \overline{F}$, $fn(F) \subseteq \overline{\texttt{A}} \cup \overline{\texttt{x}} \cup \overline{\texttt{h}}$.

We illustrate relevant features of *Stipula* by means of few examples. Consider the `Fill_Move` contract

```
stipula Fill_Move { parties Alice,Bob   assets h1,h2   init Q0
    @Q0 Alice: fill()[k]{   k ⊸ h2   } ⇒ @Q1
    @Q1 Bob: move()[]{   h2 ⊸ h1   } ⇒ @Q0
    @Q0 Bob: end()[]{   h1 ⊸ Bob   } ⇒ @Q2
}
```

that regulates interactions between `Alice` and `Bob`. It has two assets and three states `Q0`, `Q1` and `Q2`, with initial state `Q0`. In `Q0`, `Alice` may move part of her asset by invoking `fill`; the asset is stored in the formal parameter `k`. That is, the party `Alice` is assumed to own some asset and the invocation, *e.g.* `Alice.fill()[5.0]`, is removing *5* units from `Alice`'s asset and storing them in `k` (and then in `h2`). Said otherwise, the total assets of the system is invariant during the invocation; similarly during the operations `h` $\multimap$ `h'`. The execution of `fill` moves the assets in `k` to `h2`) and makes the contract transit to the state `Q1`. In this state, the unique admitted function is `move` by which `Bob` accumulates in `h1` the assets sent by `Alice`. The contract's state becomes `Q0` again and the behaviour may *cycle*. `Fill_Move` terminates when, in `Q0`, `Bob` decides to grab the whole content of `h1`. Notice the *nondeterministic* behaviour when `Fill_Move` is in `Q0`: according to a `fill` or an `end` function is invoked, the contract may transit in `Q1` or `Q2` (this is called *external* nondeterminism in the literature). Notice also that *Stipula* overlooks the details of the interactions with the parties (usually an asset transfer between the parties and the contract is mediated by a bank).

`Fill_Move` has the property that assets are eventually emptied (whatever it is the state of the contract – *the contract is liquid*). *Stipula* contracts do not always retain this property. For example, the following `Ping_Pong` contract

```
stipula Ping_Pong { parties Amy,Mary   assets h,k   init Q0
    @Q0 Mary: ping()[u]{   h ⊸ Mary   u ⊸ k   } ⇒ @Q1
    @Q1 Amy: pong()[v]{   k ⊸ Amy   v ⊸ h   } ⇒ @Q0
}
```

has a cyclic behaviour where `Mary` and `Amy` exchange asset values. By invoking `ping`, `Mary` moves her asset into `u` (and then into the asset field `k`) and grabs the value stored in `h`; conversely, by invoking `pong`, `Amy` moves her asset into `v` (and then into the asset field `h`) and grabs the value stored in `k`. (Since asset fields are initially empty, the first invocation of `ping` does not deliver anything to `Mary`.) Apart the initial state, `Ping_Pong` never reaches a state where `h` and `k` are both empty at the same time (if `Mary` and `Amy` invocations carry nonempty assets) – in the terminology of the next section, *the contract is not liquid.* States have either `k` empty – `Q0` – or `h` empty – `Q1`.

3 Semantics

Let a *configuration*, ranged over by $\mathbb{C}$, $\mathbb{C}'$, $\cdots$, be a tuple $\mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,\Sigma)$ where

- $\mathtt{C}$ is the contract name and $\mathtt{Q}$ is one of its states;
- ℓ, called *memory*, is a mapping from names (parties, fields, assets and function's parameters) to values. The values of parties are noted $A, A', \cdots$. These values cannot be passed as function's parameters and cannot be hard-coded into the source contracts, since they do not belong to expressions. We write $\ell[\mathtt{h} \mapsto u]$ to specify the memory that binds $\mathtt{h}$ to u and is equal to ℓ otherwise;
- Σ is the (possibly empty) residual of a function body, *i.e.* Σ is either $\_$ or a term $S \Rrightarrow @\mathtt{Q}$.

Configurations such as $\mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,\_)$, *i.e.* there is no statement to execute, are called *idle*.

We will use the auxiliary *evaluation function* $[\![E]\!]_\ell$ that returns the value of E in the memory ℓ such that:

- $[\![v]\!]_\ell = v$ for real numbers and asset values, $[\![\mathtt{true}]\!]_\ell = \mathtt{1.0}$ and $[\![\mathtt{false}]\!]_\ell = \mathtt{0.0}$ (booleans are converted to reals); $[\![X]\!]_\ell = \ell(X)$ for names of assets, fields and parameters. i
- let *uop* and *op* be the semantic operations corresponding to `uop` and `op`, then $[\![\mathtt{uop}\, E]\!]_\ell = \mathit{uop}\, v$, $[\![E \,\mathtt{op}\, E']\!]_\ell = v\; \mathit{op}\; v'$ with $[\![E]\!]_\ell = v, [\![E']\!]_\ell = v'$. In case of boolean operations, every non-null real corresponds to `true` and `0.0` corresponds to `false`; the operations return the reals for `true` and `false`. Because of the restrictions on the language, `uop` and `op` are always defined.

The semantics of *Stipula* is defined by a *transition relation*, noted $\mathbb{C} \xrightarrow{\mu} \mathbb{C}'$, that is given in Table 2, where μ is either empty or $A : \mathtt{f}(\overline{u})[\overline{v}]$ or $v \rightarrow A$ or $v \multimap A$. Rule [Function] defines invocations: the label specifies the party A performing the invocation and the function name $\mathtt{f}$ with the actual parameters. The transition may occur provided (i) the contract is in the state $\mathtt{Q}$ that admits

$$\textsc{[Function]}\quad \dfrac{\begin{array}{c}\texttt{@Q}\ \texttt{A} : \texttt{f}\,(\overline{\texttt{y}})\,[\overline{\texttt{k}}]\ \{\, S\,\} \Rightarrow \texttt{@Q}' \in \texttt{C} \\ \ell(\texttt{A}) = A \qquad \ell' = \ell[\overline{\texttt{y}} \mapsto \overline{u}, \overline{\texttt{k}} \mapsto \overline{v}]\end{array}}{\texttt{C}(\texttt{Q}\,,\, \ell\,,\, \_) \xrightarrow{A:\texttt{f}(\overline{u})[\overline{v}]} \texttt{C}(\texttt{Q}\,,\, \ell'\,,\, S \Rightarrow \texttt{@Q}')}$$

$$\textsc{[State-Change]}\quad \texttt{C}(\texttt{Q}\,,\, \ell\,,\, \_ \Rightarrow \texttt{@Q}') \longrightarrow \texttt{C}(\texttt{Q}'\,,\, \ell\,,\, \_)$$

$$\textsc{[Value-Send]}\quad \dfrac{[\![E]\!]_\ell = v \quad \ell(\texttt{A}) = A}{\texttt{C}(\texttt{Q}\,,\, \ell\,,\, E \to \texttt{A}\ \varSigma) \xrightarrow{v \to A} \texttt{C}(\texttt{Q}\,,\, \ell\,,\, \varSigma)}$$

$$\textsc{[Asset-Send]}\quad \dfrac{[\![c \times \texttt{h}]\!]_\ell = v \quad \ell(\texttt{A}) = A \quad [\![\texttt{h} - v]\!]_\ell = v'}{\texttt{C}(\texttt{Q}\,,\, \ell\,,\, c \times \texttt{h} \multimap \texttt{A}\ \varSigma) \xrightarrow{v \multimap A} \texttt{C}(\texttt{Q}\,,\, \ell[\texttt{h} \mapsto v']\,,\, \varSigma)}$$

$$\textsc{[Field-Update]}\quad \dfrac{[\![E]\!]_\ell = v}{\texttt{C}(\texttt{Q}\,,\, \ell\,,\, E \to \texttt{x}\ \varSigma) \longrightarrow \texttt{C}(\texttt{Q}\,,\, \ell[\texttt{x} \mapsto v]\,,\, \varSigma)}$$

$$\textsc{[Asset-Update]}\quad \dfrac{\begin{array}{c}[\![c \times \texttt{h}]\!]_\ell = v \quad [\![\texttt{h} - v]\!]_\ell = v' \quad [\![\texttt{h}' + v]\!]_\ell = v'' \\ \ell' = \ell[\texttt{h} \mapsto v', \texttt{h}' \mapsto v'']\end{array}}{\texttt{C}(\texttt{Q}\,,\, \ell\,,\, c \times \texttt{h} \multimap \texttt{h}'\ \varSigma) \longrightarrow \texttt{C}(\texttt{Q}\,,\, \ell'\,,\, \varSigma)}$$

$$\textsc{[Cond-true]}\quad \dfrac{[\![E]\!]_\ell = \texttt{true}}{\begin{array}{r}\texttt{C}(\texttt{Q}\,,\, \ell\,,\, \texttt{if}\ (E)\ \{\, S\,\}\ \texttt{else}\ \{\, S'\,\}\ \varSigma) \\ \longrightarrow \texttt{C}(\texttt{Q}\,,\, \ell\,,\, S\ \varSigma)\end{array}}$$

$$\textsc{[Cond-false]}\quad \dfrac{[\![E]\!]_\ell = \texttt{false}}{\begin{array}{r}\texttt{C}(\texttt{Q}\,,\, \ell\,,\, \texttt{if}\ (E)\ \{\, S\,\}\ \texttt{else}\ \{\, S'\,\}\ \varSigma) \\ \longrightarrow \texttt{C}(\texttt{Q}\,,\, \ell\,,\, S'\ \varSigma)\end{array}}$$

Table 2. The transition relation of *Stipula*

invocations of `f` from A such that $\ell(\texttt{A}) = A$ and (ii) the configuration is *idle*. Rule [State-Change] says that a contract changes state when the execution of the statement in the function's body terminates. To keep simple the operational semantics of *Stipula*, we do not remove garbage names in the memories (the formal parameters of functions once the functions have terminated). Therefore memories retain such names and the formal parameters keep the value they have at the end of the function execution. These values are lost when the function is called again (*c.f.* rule [Function]: in ℓ', the assets $\overline{\texttt{k}}$ are updated with $\overline{v}$). A function that does not empty asset formal parameters is clearly incorrect and the following analysis will catch such errors.

Regarding statements, we only discuss [Asset-Send] and [Asset-Update] because the other rules are standard. Rule [Asset-Send] delivers part of an asset `h` to A. This part, named v, is removed from the asset, *c.f.* the memory of the right-hand side state in the conclusion. In a similar way, [Asset-Update] moves a part v of an asset `h` to an asset $\texttt{h}'$. For this reason, the final memory becomes $\ell[\texttt{h} \mapsto v', \texttt{h}' \mapsto v'']$, where $v' = \ell(\texttt{h}) - v$ and $v'' = \ell(\texttt{h}') + v$.

A contract `stipula C` $\{$ `parties` $\overline{\texttt{A}}$ `fields` $\overline{\texttt{x}}$ `assets` $\overline{\texttt{h}}$ `init Q` $\overline{F}$ $\}$ is invoked by $\texttt{C}(\overline{A}, \overline{u})$ that corresponds to the initial configuration

$$\texttt{C}(\texttt{Q}\,,\, [\overline{\texttt{A}} \mapsto \overline{A}, \overline{\texttt{x}} \mapsto \overline{u}, \overline{\texttt{h}} \mapsto \overline{0}]\,,\, \_)\ .$$

We remark that no field and asset is left uninitialized, which means that no *undefined-value* error can occur during the execution by accessing to field and assets. Notice that the initial value of assets is 0. In order to keep the notation light we always assume that parties $\overline{\texttt{A}}$ are always instantiated by the corresponding names $\overline{A}$ written with italic fonts.

Below we use the following notation and terminology:

– We write $\mathbb{C} \overset{A.\mathtt{f}(\overline{u})[\overline{v}]}{\Longrightarrow} \mathbb{C}'$ if $\mathbb{C} \xrightarrow{A.\mathtt{f}(\overline{u})[\overline{v}]} \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_n} \mathbb{C}'$ and μ_i are either empty or $v \multimap A$ or $v \to A$ and $\mathbb{C}'$ is idle.
– We write $\mathbb{C} \Longrightarrow \mathbb{C}'$ if $\mathbb{C} \overset{A_1.\mathtt{f}_1(\overline{u_1})[\overline{v_1}]}{\Longrightarrow} \cdots \overset{A_n.\mathtt{f}_n(\overline{u_n})[\overline{v_n}]}{\Longrightarrow} \mathbb{C}'$, for some $A_1.\mathtt{f}_1(\overline{u_1})[\overline{v_1}]$, $\cdots, A_n.\mathtt{f}_n(\overline{u_n})[\overline{v_n}]$. $\mathbb{C} \Longrightarrow \mathbb{C}'$ will be called *computation.*

An important property of closed contracts guarantees that the invocation of a function never fails. This property immediately follows by the fact that, in such contracts, the evaluation of expressions and statements can never rise an error (operations are total, names are always bound to values and type errors cannot occur because values are always converted to reals).

Theorem 1 (Progress). *Let* `C` *be a closed Stipula contract with fields* $\overline{\mathtt{x}}$*, assets* $\overline{\mathtt{h}}$*, parties* $\overline{\mathtt{A}}$ *and* `@Q A:f(`$\overline{\mathtt{y}}$`)[`$\overline{\mathtt{k}}$`]{` S `}`$\Rightarrow$`@Q'` $\in$ `C`*. For every* ℓ *such that* $\overline{\mathtt{x}}, \overline{\mathtt{h}}, \overline{\mathtt{A}} \subseteq$ *dom*(ℓ)*, there is* ℓ' *such that* $\mathtt{C}(\mathtt{Q}, \ell, \_) \overset{A.\mathtt{f}(\overline{u})[\overline{v}]}{\Longrightarrow} \mathtt{C}(\mathtt{Q}', \ell', \_)$.

We conclude with the definition of liquidity. We use the following notation:

– we write $\ell(\overline{\mathtt{h}}) > \overline{0}$ if and only if there is $\mathtt{k} \in \overline{\mathtt{h}}$ such that $\ell(\mathtt{k}) > 0$; similarly $\ell(\overline{\mathtt{h}}) = \overline{0}$ if and only if, for every $\mathtt{k} \in \overline{\mathtt{h}}$, $\ell(\mathtt{k}) = 0$.

Definition 1 (Liquidity). *A Stipula contract* `C` *with assets* $\overline{\mathtt{h}}$ *and initial configuration* $\mathbb{C}$ *is* liquid *if, for every computation* $\mathbb{C} \Longrightarrow \mathtt{C}(\mathtt{Q}, \ell, \_)$*, then*

1. $\ell(\overline{\mathtt{h}'}) = \overline{0}$ *with* $\overline{\mathtt{h}'} = dom(\ell) \backslash \overline{\mathtt{h}}$*;*
2. *if* $\ell(\overline{\mathtt{h}}) > \overline{0}$ *then there is* $\mathtt{C}(\mathtt{Q}, \ell, \_) \Longrightarrow \mathtt{C}(\mathtt{Q}', \ell', \_)$ *such that* $\ell'(\overline{\mathtt{h}}) = \overline{0}$.

We notice that Progress is critical for reducing liquidity to some form of reachability analysis (otherwise we should also deal with function invocations that terminate into a stuck state because of an error). In the following sections, using a symbolic technique, we define an algorithm for assessing liquidity and demonstrate its correctness.

4 The theory of liquidity

We begin with the definition of the *liquidity type system* that returns an abstraction of the input-output behaviour of functions with respect to assets. These abstractions record whether an asset is zero – notation $\mathbb{0}$ – or not – notation $\mathbb{1}$. The values $\mathbb{0}$ and $\mathbb{1}$ are called *liquidity values* and we use the following notation:

– *liquidity expressions* $\mathbb{e}$ are defined as follows, where ξ, ξ', $\cdots$ range over (symbolic) liquidity names:

$$\mathbb{e} ::= \mathbb{0} \quad | \quad \mathbb{1} \quad | \quad \xi \quad | \quad \mathbb{e} \sqcup \mathbb{e} \quad | \quad \mathbb{e} \sqcap \mathbb{e}\,.$$

They are ordered as $\mathbb{0} \leqslant \mathbb{e}$ and $\mathbb{e} \leqslant \mathbb{1}$; the operations $\sqcup$ and $\sqcap$ respectively return the maximum and the minimum value of the two arguments; they are monotone with respect to $\leqslant$ (that is $\mathbb{e}_1 \leqslant \mathbb{e}_1'$ and $\mathbb{e}_2 \leqslant \mathbb{e}_2'$ imply $\mathbb{e}_1 \sqcup \mathbb{e}_2 \leqslant \mathbb{e}_1' \sqcup \mathbb{e}_2'$ and $\mathbb{e}_1 \sqcap \mathbb{e}_2 \leqslant \mathbb{e}_1' \sqcap \mathbb{e}_2'$). A liquidity expression that does not contain liquidity names is called *ground.* Two tuples are ordered $\leqslant$ if they are element-wise ordered by $\leqslant$.

$$\text{[L-SEND]}\quad \frac{\mathtt{A}, \mathit{fn}(E) \subseteq \mathtt{X} \cup \mathit{dom}(\varXi)}{\varXi \vdash_{\mathtt{X}} E \to \mathtt{A} : \varXi} \qquad \text{[L-UPDATE]}\quad \frac{\mathtt{x}, \mathit{fn}(E) \subseteq \mathtt{X} \cup \mathit{dom}(\varXi)}{\varXi \vdash_{\mathtt{X}} E \to \mathtt{x} : \varXi}$$

$$\text{[L-ASEND]}\quad \frac{\mathtt{h} \in \mathit{dom}(\varXi) \quad \mathtt{A} \in \mathtt{X}}{\varXi \vdash_{\mathtt{X}} \mathtt{h} \multimap \mathtt{A} : \varXi[\mathtt{h} \mapsto \mathbb{0}]} \qquad \text{[L-EXPASEND]}\quad \frac{\mathtt{h} \in \mathit{dom}(\varXi) \quad c \neq 1 \quad \mathtt{A} \in \mathtt{X}}{\varXi \vdash_{\mathtt{X}} c \times \mathtt{h} \multimap \mathtt{A} : \varXi}$$

$$\text{[L-AUPDATE]}\quad \frac{\mathtt{h}, \mathtt{h}' \in \mathit{dom}(\varXi) \quad \mathbb{e} = \varXi(\mathtt{h}) \sqcup \varXi(\mathtt{h}')}{\varXi \vdash \mathtt{h} \multimap \mathtt{h}' : \varXi[\mathtt{h} \mapsto \mathbb{0}, \mathtt{h}' \mapsto \mathbb{e}]} \qquad \text{[L-EXPAUPD]}\quad \frac{\mathtt{h}, \mathtt{h}' \in \mathit{dom}(\varXi) \quad c \neq 1 \quad \mathbb{e} = \varXi(\mathtt{h}) \sqcup \varXi(\mathtt{h}')}{\varXi \vdash c \times \mathtt{h} \multimap \mathtt{h}' : \varXi[\mathtt{h}' \mapsto \mathbb{e}]}$$

$$\text{[L-ZERO]}\quad \varXi \vdash_{\mathtt{X}} \_ : \varXi \qquad \text{[L-SEQ]}\quad \frac{\varXi \vdash_{\mathtt{X}} P : \varXi' \quad \varXi' \vdash_{\mathtt{X}} S : \varXi''}{\varXi \vdash_{\mathtt{X}} P\ S : \varXi''} \qquad \text{[L-COND]}\quad \frac{\begin{array}{c} \mathit{fn}(E) \subseteq \mathtt{X} \cup \mathit{dom}(\varXi) \\ \varXi \vdash_{\mathtt{X}} S : \varXi' \quad \varXi \vdash_{\mathtt{X}} S' : \varXi'' \\ \varXi' \sqcup \varXi'' \vdash_{\mathtt{X}} S'' : \varXi''' \end{array}}{\varXi \vdash_{\mathtt{X}} \mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S'\,\}\ S'' : \varXi'''}$$

$$\text{[L-FUNCTION]}\quad \frac{\mathtt{A}, \mathit{fn}(S) \subseteq \mathtt{X} \cup \overline{\mathtt{y}} \qquad \overline{\xi'}\ \mathit{fresh} \qquad \varXi[\overline{\mathtt{k}} \mapsto \overline{\xi'}] \vdash_{\mathtt{X} \cup \overline{\mathtt{y}}} S : \varXi'}{\varXi \vdash_{\mathtt{X}} @\mathtt{Q}\ \mathtt{A} : \mathtt{f}(\,\overline{\mathtt{y}}\,)[\,\overline{\mathtt{k}}\,]\{\,S\,\} \Rightarrow @\mathtt{Q}' :\ \mathtt{Q}\ \mathtt{A.f}\ \mathtt{Q}' : \varXi[\overline{\mathtt{k}} \mapsto \overline{\mathbb{1}}] \to \varXi'\{^{\overline{\mathbb{1}}}/_{\overline{\xi'}}\}}$$

$$\text{[L-CONTRACT]}\quad \frac{\overline{\xi}\ \mathit{fresh} \quad \left([\overline{\mathtt{h}} \mapsto \overline{\xi}] \vdash_{\overline{\mathtt{A}} \cup \overline{\mathtt{x}}} F : \mathcal{L}_F\right)^{F \in \overline{F}}}{\vdash \mathtt{stipula}\ \mathtt{C}\ \{\mathtt{parties}\ \overline{\mathtt{A}}\ \mathtt{fields}\ \overline{\mathtt{x}}\ \mathtt{assets}\ \overline{\mathtt{h}}\ \mathtt{init}\ \mathtt{Q}\ \ \overline{F}\ \} : \bigcup_{F \in \overline{F}} \mathcal{L}_F}$$

Table 3. The Liquidity type system of *Stipula*

- *environments* $\varXi$ map contract's assets and asset parameters to liquidity expressions. Environments that map names to ground liquidity expressions are called *ground environments.*
- *liquidity function types* $\mathtt{Q}\,\mathtt{A.f}\,\mathtt{Q}' :\ \varXi \to \varXi'$ where $\varXi \to \varXi'$ records the liquidity effects of fully executing the body of $\mathtt{Q}\,\mathtt{A.f}\,\mathtt{Q}'$.
- *judgments* $\varXi \vdash_{\mathtt{X}} S : \varXi'$ for statements and $\varXi \vdash_{\mathtt{X}} @\mathtt{Q}\ \mathtt{A} : \mathtt{f}(\,\overline{\mathtt{x}}\,)[\,\overline{\mathtt{h}'}\,]\ \{\,S\,\} \Rightarrow @\mathtt{Q}' : \mathcal{L}$ for function definitions, where $\mathcal{L}$ is a liquidity function type. The set $\mathtt{X}$ contains party and field names.

The liquidity type system is defined in Table 3; below we discuss the most relevant rules. Asset movements have four rules – [L-ASEND], [L-EXPASEND] [L-AUPDATE] and [L-EXPAUPD] – according to whether the constant factor is 0 or not and whether the asset is moved to an asset or a party. According to [L-AUPDATE], the final asset environment of $\mathtt{h} \multimap \mathtt{h}'$ (which is an abbreviation for $1 \times \mathtt{h} \multimap \mathtt{h}'$) has $\mathtt{h}$ that is emptied and $\mathtt{h}'$ that gathers the value of $\mathtt{h}$, henceforth the liquidity expression $\varXi(\mathtt{h}) \sqcup \varXi(\mathtt{h}')$. Notice that, when both $\mathtt{h}$ and $\mathtt{h}'$ are $\mathbb{0}$, the overall result is $\mathbb{0}$. In the rule [L-EXPAUPD], the asset $\mathtt{h}$ is decreased by an amount that is moved to $\mathtt{h}'$. Since $c \neq 1$, the static analysis (which is independent of the runtime value of $\mathtt{h}$) can only safely assume that the asset $\mathtt{h}$ is not emptied by this operation (if it was not empty before). Therefore, after the withdraw, the liquidity value of $\mathtt{h}$ has not changed. On the other hand, the asset $\mathtt{h}'$ is increased of some amount if both c and $\mathtt{h}$ have a non zero liquidity value, henceforth the expression $\Xi(\mathtt{h}) \sqcup \Xi(\mathtt{h}')$. In particular, as before, when both $\Xi(\mathtt{h})$ and $\Xi(\mathtt{h}')$ are $\mathbb{0}$, the overall result is $\mathbb{0}$.

The rule for conditionals is [L-COND], where the operation $\sqcup$ on environments is defined pointwise by $(\Xi' \sqcup \Xi'')(\mathtt{h}) = \Xi'(\mathtt{h}) \sqcup \Xi''(\mathtt{h})$. That is, the liquidity analyzer over-approximates the final environments of $\mathtt{if}\ (E)\ \{\ S\ \}\ \mathtt{else}\ \{\ S'\ \}$ by taking the maximum values between the results of parsing S (that corresponds to a true value of E) and those of S' (that corresponds to a false value of E). Regarding E, the analyzer only verifies that its names are bound in the contract.

The rule for *Stipula* contracts is [L-CONTRACT]; it collects the liquidity labels $\mathcal{L}_i$ that describe the liquidity effects of each contract's function; each function assumes injective environments that respectively associate contract's assets with fresh symbolic names. In turn, the type produced by [L-FUNCTION] says that the complete execution of $\mathtt{Q\ A.f\ Q'}$ has liquidity effects $\Xi[\overline{\mathtt{h}'} \mapsto \overline{\mathbb{1}}] \to \Xi'\{^{\overline{\mathbb{1}}}/_{\overline{\xi'}}\}$, assuming that the body S of the function is typed as $\Xi[\overline{\mathtt{h}'} \mapsto \overline{\xi'}] \vdash S : \Xi'$. That is, in the conclusion of [L-FUNCTION] we replace the symbolic values of the liquidity names representing formal parameters with $\mathbb{1}$, because they may be any value when the function will be called.

Example 1. The set $\mathcal{L}$ of the $\mathtt{Fill\_Move}$ contract contains the following liquidity types:

$$\begin{array}{r}
\mathtt{Q0\ Alice.fill\ Q1} : [\mathtt{h1} \mapsto \xi_1, \mathtt{h2} \mapsto \xi_2, \mathtt{k} \mapsto \mathbb{1}] \to [\mathtt{h1} \mapsto \xi_1, \mathtt{h2} \mapsto \xi_2 \sqcup \mathbb{1}, \mathtt{k} \mapsto \mathbb{0}] \\
\mathtt{Q1\ Bob.move\ Q0} : [\mathtt{h1} \mapsto \xi_1, \mathtt{h2} \mapsto \xi_2] \to [\mathtt{h1} \mapsto \xi_1 \sqcup \xi_2, \mathtt{h2} \mapsto \mathbb{0}] \\
\mathtt{Q0\ Bob.end\ Q2} : [\mathtt{h1} \mapsto \xi_1, \mathtt{h2} \mapsto \xi_2] \to [\mathtt{h1} \mapsto \mathbb{0}, \mathtt{h2} \mapsto \xi_2]
\end{array}$$

In the following we will always shorten $\vdash \mathtt{stipula\ C}\ \{\mathtt{parties}\ \overline{\mathtt{A}}\ \mathtt{fields}\ \overline{\mathtt{x}}\ \mathtt{assets}\ \overline{\mathtt{h}}\ \mathtt{init\ Q}\ \overline{F}\ \} : \mathcal{L}$ into $\vdash \mathtt{C} : \mathcal{L}$. A first property of the liquidity type system is that typed contracts are closed.

Proposition 1. *If* $\vdash \mathtt{C} : \mathcal{L}$ *then* $\mathtt{C}$ *is closed.*

Therefore typed contracts own the progress property (Theorem 1). The correctness of the system in Table 3 requires the following notions:

- A *(liquidity) substitution* is a map from liquidity names to liquidity expressions (that may contain names, as well). Substitutions will be noted either σ, σ', $\cdots$ or $\{^{\overline{\mathbb{e}}}/_{\overline{\chi}}\}$. A substitution is *ground* when it maps liquidity names to ground liquidity expressions. For example $\{^{\mathbb{0},\mathbb{1}}/_{\chi,\xi}\}$ and $\{^{\mathbb{0}\sqcup\mathbb{1},\mathbb{1}\sqcap\mathbb{0}}/_{\chi,\xi}\}$ are ground substitutions, $\{^{\mathbb{0}\sqcup\chi'}/_{\chi}\}$ is not.
  We let $\sigma(\Xi)$ be the environment where $\sigma(\Xi)(\mathtt{x}) = \sigma(\Xi(\mathtt{x}))$.
- Let $[\![\mathbb{e}]\!]$ be the *partial evaluation* of $\mathbb{e}$ by applying the commutativity axioms of $\sqcup$ and $\sqcap$ and the axioms $\mathbb{0} \sqcup \mathbb{e} = \mathbb{e}$, $\mathbb{0} \sqcap \mathbb{e} = \mathbb{0}$, $\mathbb{1} \sqcup \mathbb{e} = \mathbb{1}$, $\mathbb{1} \sqcap \mathbb{e} = \mathbb{e}$. More

precisely

$$[\![\mathbb{e}]\!] = \begin{cases} \mathbb{e} & \textit{if} \;\; \mathbb{e} = \mathbb{0} \;\; \textit{or} \;\; \mathbb{e} = \mathbb{1} \;\; \textit{or} \;\; \mathbb{e} = \xi \\ [\![\mathbb{e}']\!] & \textit{if} \;\; (\mathbb{e} = \mathbb{e}' \sqcup \mathbb{e}'' \;\; \textit{or} \;\; \mathbb{e} = \mathbb{e}'' \sqcup \mathbb{e}') \;\; \textit{and} \;\; [\![\mathbb{e}'']\!] = \mathbb{0} \\ [\![\mathbb{e}']\!] & \textit{if} \;\; (\mathbb{e} = \mathbb{e}' \sqcap \mathbb{e}'' \;\; \textit{or} \;\; \mathbb{e} = \mathbb{e}'' \sqcap \mathbb{e}') \;\; \textit{and} \;\; [\![\mathbb{e}'']\!] = \mathbb{1} \\ \mathbb{0} & \textit{if} \;\; \mathbb{e} = \mathbb{e}' \sqcap \mathbb{e}'' \;\; \textit{and either} \;\; [\![\mathbb{e}']\!] = \mathbb{0} \;\; \textit{or} \;\; [\![\mathbb{e}'']\!] = \mathbb{0} \\ \mathbb{1} & \textit{if} \;\; \mathbb{e} = \mathbb{e}' \sqcup \mathbb{e}'' \;\; \textit{and either} \;\; [\![\mathbb{e}']\!] = \mathbb{1} \;\; \textit{or} \;\; [\![\mathbb{e}'']\!] = \mathbb{1} \\ [\![\mathbb{e}']\!] \# [\![\mathbb{e}'']\!] & \textit{otherwise} \;\; (\# \;\; \textit{is either} \;\; \sqcup \;\; \textit{or} \;\; \sqcap) \end{cases}$$

Notice that if $\mathbb{e}$ is ground then $[\![\mathbb{e}]\!]$ is either $\mathbb{0}$ or $\mathbb{1}$.

- We let $[\![\Xi]\!]$ be the environment where $[\![\Xi]\!](\mathtt{x}) = [\![\Xi(\mathtt{x})]\!]$. Therefore, when Ξ is ground, $[\![\Xi]\!]$ is ground as well. The converse is false.
- When Ξ and Ξ' are ground, we write $\Xi \leqslant \Xi'$ if and only if, for every $\mathtt{h} \in \mathit{dom}(\Xi)$, $[\![\Xi(\mathtt{h})]\!] \leqslant [\![\Xi'(\mathtt{h})]\!]$. Observe that this implies that $\mathit{dom}(\Xi) \subseteq \mathit{dom}(\Xi')$.
- $\Xi|_{\overline{\mathtt{h}}}$ is the environment Ξ restricted to the names $\overline{\mathtt{h}}$, defined as follows

$$\Xi|_{\overline{\mathtt{h}}}(\mathtt{k}) \;=\; \begin{cases} \Xi(\mathtt{k}) & \text{if } \mathtt{k} \in \overline{\mathtt{h}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

- let $\ell = [\overline{\mathtt{A}} \mapsto \overline{A}, \overline{\mathtt{x}'} \mapsto \overline{u}, \overline{\mathtt{h}'} \mapsto \overline{v}]$ be a memory, where $\overline{\mathtt{x}'}$ are contract's fields and non-asset parameters, while $\overline{\mathtt{h}'}$ are contract's assets and the asset parameters. We let $\mathbb{E}(\ell)$ be the ground environment defined as follows:

$$\mathbb{E}(\ell)(\mathtt{k}) \;=\; \begin{cases} \mathbb{0} & \text{if } \;\mathtt{k} \in \overline{\mathtt{h}'} \;\text{ and }\; \ell(\mathtt{k}) = 0 \\ \mathbb{1} & \text{if } \;\mathtt{k} \in \overline{\mathtt{h}'} \;\text{ and }\; \ell(\mathtt{k}) \neq 0 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Liquidity types are correct, as stated by the following theorem.

Theorem 2 (Correctness of liquidity labels). *Let* $\vdash \mathtt{C} : \mathcal{L}$ *and* $\mathtt{@Q}\;\mathtt{A} : \mathtt{f}\,(\overline{\mathtt{y}})\,[\overline{\mathtt{k}}]\;\{\, S \,\} \Rightarrow \mathtt{@Q}' \in \mathtt{C}$ *and* $\mathtt{Q}\;\mathtt{A.f}\;\mathtt{Q}' : \Xi \to \Xi'$ *in* $\mathcal{L}$. *If* $\mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,\_) \overset{A.\mathtt{f}(\overline{u})[\overline{v}]}{\Longrightarrow} \mathtt{C}(\mathtt{Q}'\,,\,\ell'\,,\,\_)$ *then there are* $\mathtt{X}$ *and* Ξ'' *such that:*

1. $\mathbb{E}(\ell[\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}]) \vdash_{\mathtt{X}} S : \Xi''$*;*
2. $\mathbb{E}(\ell[\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}])|_{\mathit{dom}(\Xi)} \leqslant [\![\sigma(\Xi)]\!]$ *and* $[\![\Xi''|_{\mathit{dom}(\Xi)}]\!] \leqslant [\![\sigma(\Xi')]\!]$*, for a ground substitution* σ*;*
3. $\mathbb{E}(\ell')|_{\mathit{dom}(\Xi)} \leqslant [\![\Xi'']\!]$*.*

For example, consider the transition

$$\mathtt{Fill\_Move}(\mathtt{Q1}\,,\,\ell\,,\,\_) \overset{Bob.\mathtt{move}()[]}{\longrightarrow} \mathtt{Fill\_Move}(\mathtt{Q0}\,,\,\ell'\,,\,\_)$$

of $\mathtt{Fill\_Move}$ where $\ell = [\mathtt{h1} \mapsto 25.0, \mathtt{h2} \mapsto 5.0]$ and $\ell' = [\mathtt{h1} \mapsto 30.0, \mathtt{h2} \mapsto 0.0]$. By definition, $\mathbb{E}(\ell) = [\mathtt{h1} \mapsto \mathbb{1}, \mathtt{h2} \mapsto \mathbb{1}]$. Letting $\mathtt{X} = \{\mathtt{Alice}, \mathtt{Bob}\}$, by the liquidity type system we obtain $\mathbb{E}(\ell) \vdash_{\mathtt{X}} \mathtt{h2} \multimap \mathtt{h1} : \Xi''$, $\Xi'' = [\mathtt{h1} \mapsto \mathbb{1} \sqcup \mathbb{1}, \mathtt{h2} \mapsto \mathbb{0}]$. Since $\mathtt{Q1}\;\mathtt{Bob.move}\;\mathtt{Q0} : [\mathtt{h1} \mapsto \xi_1, \mathtt{h2} \mapsto \xi_2] \to [\mathtt{h1} \mapsto \xi_1 \sqcup \xi_2, \mathtt{h2} \mapsto \mathbb{0}]$ (see Example 1), the ground substitution σ that satisfies Theorem 2.2 is $[\xi_1 \mapsto$

$\mathbb{1}, \xi_2 \mapsto \mathbb{1}]$ (actually, in this case, the "$\leqslant$" are equalities). Regarding the last item, $\mathbb{E}(\ell') = [\mathtt{h1} \mapsto \mathbb{1}, \mathtt{h2} \mapsto \mathbb{0}]$ and $\mathbb{E}(\ell') \leqslant [\![\Xi'']\!]$ follows by definition.

A basic notion of our theory is the one of abstract computation and its liquidity type.

Definition 2. *An* abstract computation *of a Stipula contract, ranged over by $\varphi, \varphi', \cdots$, is a finite sequence $\mathtt{Q}_1\ \mathtt{A}_1.\mathtt{f}_1\ \mathtt{Q}_2\ ;\ \cdots\ ;\ \mathtt{Q}_n\ \mathtt{A}_n.\mathtt{f}_n\ \mathtt{Q}_{n+1}$ of contract's functions, shortened into $\{\ \mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1}\ \}^{i \in 1..n}$. We use the notation $\mathtt{Q} \overset{\varphi}{\rightsquigarrow} \mathtt{Q}'$ to highlight the initial and final states of φ and we let $\{\ \mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1}\ \}^{i \in 1..n}$ be the abstract computation of $\big(\mathtt{C}(\mathtt{Q}_i\ ,\ \ell_i\ ,\ \_) \xrightarrow{A_i:\mathtt{f}_i(\overline{u_i})[\overline{v_i}]} \mathtt{C}(\mathtt{Q}_{i+1}\ ,\ \ell_{i+1}\ ,\ \_)\big)^{i \in 1..n}$.*

*An abstract computation φ is κ-*canonical *if functions occur at most κ-times in φ.*

We notice that abstract computations do not mind of memories. Regarding canonical computations, every prefix of a κ-canonical computation is κ-canonical as well, including the empty computation.

Definition 3 (Liquidity type of an abstract computation). *Let $\vdash \mathtt{C} : \mathcal{L}$ and $\overline{\mathtt{h}}$ be the assets of $\mathtt{C}$. Let also $\mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1} : \Xi_i \to \Xi'_i \in \mathcal{L}$ for every $i \in 1..n$. The* liquidity type *of $\varphi = \{\ \mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1}\ \}^{i \in 1..n}$, noted $\mathtt{L}_\varphi$, is $\Xi_1^{(b)}|_{\overline{\mathtt{h}}} \to \Xi_n^{(e)}|_{\overline{\mathtt{h}}}$ where $\Xi_1^{(b)}$ and $\Xi_n^{(e)}$ ("b" stays for* begin*, "e" stays for* end*) are defined as follows*

$$\Xi_1^{(b)} = \Xi_1 \qquad \Xi_{i+1}^{(b)} = \Xi_{i+1}\{^{\Xi_i^{(e)}(\overline{\mathtt{h}})}/_{\overline{\xi}}\} \qquad \Xi_i^{(e)} = \Xi'_i\{^{\Xi_i^{(b)}(\overline{\mathtt{h}})}/_{\overline{\xi}}\}\ .$$

Notice that, by definition, the initial environment of the i-th type is updated so that it maps assets to the values computed at the end of the $(i-1)$-th transition. These values are also propagated to the final environment of the i-th transitions by substituting the occurrence of a liquidity name with the computed value of the corresponding asset. Notice also that the domains of the environments $\Xi_i^{(b)}$, $1 \leqslant i \leqslant n$, are in general different because they are also defined on the asset parameters of the corresponding function. However, formal parameters are not relevant because they are always replaced by $\mathbb{1}$ and are therefore dropped in the liquidity types of computations.

For example, consider the computation of the `Fill_Move` contract

$$\varphi = \mathtt{Q0\ Alice.fill\ Q1\ ;\ Q1\ Bob.move\ Q0\ ;\ Q0\ Bob.end\ Q2}$$

(we refer to Example 1 for the types of the contract). Let $H = \{\mathtt{h1}, \mathtt{h2}\}$ and $\Xi = [\mathtt{h1} \mapsto \xi_1, \mathtt{h2} \mapsto \xi_2]$. φ has liquidity type $\Xi_1^{(b)}|_H \to \Xi_3^{(e)}|_H$ where:

$$\begin{aligned}
\Xi_1^{(b)} &= \Xi[\mathtt{k} \mapsto \mathbb{1}] & \Xi_1^{(e)} &= \Xi[\mathtt{h}_2 \mapsto \xi_2 \sqcup \mathbb{1}, \mathtt{k} \mapsto \mathbb{0}] \\
\Xi_2^{(b)} &= \Xi\{^{\xi_2 \sqcup \mathbb{1}}/_{\xi_2}\} & \Xi_2^{(e)} &= [\mathtt{h1} \mapsto \xi_1 \sqcup \xi_2, \mathtt{h2} \mapsto \mathbb{0}]\{^{\xi_2 \sqcup \mathbb{1}}/_{\xi_2}\} \\
&= \Xi[\mathtt{h}_2 \mapsto \xi_2 \sqcup \mathbb{1}] & &= [\mathtt{h1} \mapsto \xi_1 \sqcup \xi_2 \sqcup \mathbb{1}, \mathtt{h2} \mapsto \mathbb{0}] \\
\Xi_3^{(b)} &= \Xi\{^{\xi_1 \sqcup \xi_2 \sqcup \mathbb{1}, \mathbb{0}}/_{\xi_1, \xi_2}\} & \Xi_3^{(e)} &= [\mathtt{h1} \mapsto \mathbb{0}, \mathtt{h2} \mapsto \xi_2]\{^{\xi_1 \sqcup \xi_2 \sqcup \mathbb{1}, \mathbb{0}}/_{\xi_1, \xi_2}\} \\
&= [\mathtt{h1} \mapsto \xi_1 \sqcup \xi_2 \sqcup \mathbb{1}, \mathtt{h2} \mapsto \mathbb{0}] & &= [\mathtt{h1} \mapsto \mathbb{0}, \mathtt{h2} \mapsto \mathbb{0}]
\end{aligned}$$

Therefore $\mathtt{L}_\varphi = \Xi \to [\mathtt{h1} \mapsto \mathbb{0}, \mathtt{h2} \mapsto \mathbb{0}]$. That is, whatever they are the initial values of h1 and h2, which are represented in Ξ by the liquidity names ξ_1 and ξ_2, respectively, their liquidity values after the computation φ are $\mathbb{0}$ (henceforth they are 0 by the following Theorem 3). Notice also the differences between $\mathtt{L}_\varphi$ and $\Xi \to [\mathtt{h1} \mapsto \mathbb{0}, \mathtt{h2} \mapsto \xi_2]$, which is the type of `Bob.end`: from this last type we may derive that h1 is $\mathbb{0}$ in the final environment, while the value of h2 is the same of the initial environment (in fact, `Bob.end` only empties h1 and does not access to h2).

We recall that the operational semantics of *Stipula* in Table 2 does not remove garbage names in the memories (the formal parameters of functions once the functions have terminated, see Section 3). However, these names do not exist in environments of the liquidity types of abstract computations. For this reason, in the following statement, we restrict the inequalities to the names of the contract's assets.

Theorem 3 (Correctness of an abstract computation). *Let* $\vdash \mathtt{C} : \mathcal{L}$ *and* $\big(\mathtt{C}(\mathtt{Q}_i\,,\ \ell_i\,,\ \_) \overset{A_i:\mathtt{f}_i(\overline{u_i})[\overline{v_i}]}{\Longrightarrow} \mathtt{C}(\mathtt{Q}_{i+1}\,,\ \ell_{i+1}\,,\ \_)\big)^{i\in 1..n}$ *and the abstract computation* $\varphi = \{\ \mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1}\ \}^{i\in 1..n}$ *have liquidity type* $\mathtt{L}_\varphi = \Xi \to \Xi'$.

Then there is a substitution σ *such that* $\mathbb{E}(\ell_1)|_{\overline{\mathtt{h}}} \leqslant [\![\sigma(\Xi)]\!]$ *and* $\mathbb{E}(\ell_{n+1})|_{\overline{\mathtt{h}}} \leqslant [\![\sigma(\Xi')]\!]$.

5 The algorithm for liquidity

Analyzing the liquidity of a *Stipula* contract amounts to verifying the two constraints of Definition 1. Checking constraint 1 is not difficult: for every transition $\mathtt{Q}\,\mathtt{A.f}\,\mathtt{Q}'$ of the contract with assets $\overline{\mathtt{h}}$, we consider its liquidity type $\Xi \to \Xi'$ and verify whether, for every parameter $\mathtt{k} \notin \overline{\mathtt{h}}$, $[\![\Xi'(\mathtt{k})]\!] = \mathbb{0}$. Since there are finitely many transitions, this analysis is exhaustive. The correctness is the following: if $\mathtt{k} \notin \overline{\mathtt{h}}$ implies $[\![\Xi'(\mathtt{k})]\!] = \mathbb{0}$ then, for every substitution σ, $[\![\sigma(\Xi')]\!](\mathtt{k}) = \mathbb{0}$. Specifically for the substitution σ' such that $\mathbb{E}(\ell') \leqslant [\![\sigma'(\Xi')]\!]$, which is guaranteed by Theorem 2.

On the contrary, verifying the constraint 2 of Definition 1 is harder because the transition system of a *Stipula* contract may be complex (cycles, absence of final states, nondeterminism).

We first define the notions of reachable function and reachable state of a *Stipula* contract C. Let $\vdash \mathtt{C} : \mathcal{L}$; $\mathbb{T}^\kappa_{\mathtt{Q}}$ is *the set of κ-canonical liquidity types* $\mathtt{Q} \overset{\varphi}{\rightsquigarrow} \mathtt{Q}' : \mathtt{L}_\varphi$ where φ is a κ-canonical abstract computation starting at Q in the contract (the contract is left implicit). Notice that, by definition, the empty computation $\mathtt{Q} \overset{\varepsilon}{\rightsquigarrow} \mathtt{Q} : \Xi \to \Xi$ belongs to $\mathbb{T}^\kappa_{\mathtt{Q}}$. We say that $\mathtt{Q}'$ *is reachable* from Q if there is φ such that $\mathtt{Q} \overset{\varphi}{\rightsquigarrow} \mathtt{Q}' : \mathtt{L}_\varphi$ is in $\mathbb{T}^\kappa_{\mathtt{Q}}$. For example, in the `Fill_Move` contract, the set $\mathbb{T}^\kappa_{\mathtt{Q0}}$ is the following

$$\begin{aligned}\mathbb{T}^\kappa_{\mathtt{Q0}} = & \textstyle\bigcup_{i\leqslant\kappa, j\leqslant 1}\Big(\mathtt{Q0\,Alice.fill\,Q1}\ ;\ \mathtt{Q1\,Bob.move\,Q0}\Big)^i ; \Big(\mathtt{Q0\,Bob.move\,Q2}\Big)^{j\leqslant 1} : \mathtt{L}_{i,j} \\ & \cup \textstyle\bigcup_{i\leqslant\kappa-1}\Big(\mathtt{Q0\,Alice.fill\,Q1}\ ;\ \mathtt{Q1\,Bob.move\,Q0}\Big)^i ; \mathtt{Q0\,Alice.fill\,Q1} : \mathtt{L}_i\end{aligned}$$

Let $\mathtt{Q}$ be the initial state of $\mathtt{C}$ whose assets are $\overline{\mathtt{h}}$.

step 1. Compute $\mathbb{T}^{\kappa}_{\mathtt{Q}'}$ for every $\mathtt{Q}'$ reachable from $\mathtt{Q}$; let $\mathcal{Z} = \varnothing$.

step 2. For every $\mathtt{Q}'$ and $\mathtt{Q}' \stackrel{\varphi}{\rightsquigarrow} \mathtt{Q}'' : \Xi \to \Xi' \in \mathbb{T}^{\kappa}_{\mathtt{Q}'}$ and $\varnothing \subsetneq \overline{\mathtt{k}} \subseteq \overline{\mathtt{h}}$ such that
  (a) for every $\mathtt{k}' \in \overline{\mathtt{k}}$, $[\![\Xi'(\mathtt{k}')]\!] \neq \mathbb{0}$ and $[\![\Xi'(\mathtt{k}')]\!] \neq [\![\Xi(\mathtt{k}')]\!]$
  (b) for every $\mathtt{k}'' \in \overline{\mathtt{h}} \backslash \overline{\mathtt{k}}$, $[\![\Xi'(\mathtt{k}'')]\!] = [\![\Xi(\mathtt{k}'')]\!]$
  (c) $(\mathtt{Q}'', \overline{\mathtt{k}}) \notin \mathcal{Z}$:

  2.1 If there is no $\mathtt{Q}'$, $\mathtt{Q}' \stackrel{\varphi}{\rightsquigarrow} \mathtt{Q}'' : \Xi \to \Xi'$ and $\overline{\mathtt{k}}$ then exit: *the contract is liquid.*

  2.2 otherwise verify whether there is $\mathtt{Q}'' \stackrel{\varphi'}{\rightsquigarrow} \mathtt{Q}''' : \Xi'' \to \Xi''' \in \mathbb{T}^{\kappa}_{\mathtt{Q}''}$ such that $[\![\Xi'''(\overline{\mathtt{k}})]\!] = \mathbb{0}$ and, for every $\mathtt{k}' \in \overline{\mathtt{h}} \backslash \overline{\mathtt{k}}$, either $[\![\Xi'''(\mathtt{k}')]\!] = \mathbb{0}$ or $[\![\Xi'''(\mathtt{k}')]\!] = [\![\Xi''(\mathtt{k}')]\!]$. If this is the case, add $(\mathtt{Q}'', \overline{\mathtt{k}})$ to $\mathcal{Z}$ and reiterate **step 2**, otherwise exit: *the contract is not liquid.*

Table 4. The algorithm for liquidity – $\mathcal{Z}$ contains pairs $(\mathtt{Q}, \overline{\mathtt{k}})$

(with suitable $\mathtt{L}_{i,j}$ and $\mathtt{L}_i$).

Below, without loss of generality, we assume that every state in the contract is reachable from the initial state. A straightforward optimization allows us to reduce to this case. We also assume that our contracts satisfy item 1 of liquidity. Therefore we focus on item 2.

Verifying liquidity is complex because a single asset or a tuple of assets may become 0 during *a computation*, rather than just one transition. Let us discuss the case with an example. Consider the `Ugly` contract with assets `w1` and `w2` and functions:

```
@Q0 Mark: get()[u]{   u ⊸ w2   } ⇛ @Q1
@Q1 Sam: shift()[]{   w1 ⊸ Sam   w2 ⊸ w1   } ⇛ @Q1
@Q1 Sam: end()[]{ } ⇛ @Q2
```

In this case there is no single transition that empties all the assets. However there is a liquid computation (a computation that empties all the assets), which is the one invoking `shift` two times: `Q1 Sam.shift Q1 ; Q1 Sam.shift Q1`. In particular, we have

$$\mathtt{Q1\ Sam.shift\ Q1} : [\mathtt{w1} \mapsto \xi_1, \mathtt{w2} \mapsto \xi_2] \to [\mathtt{w1} \mapsto \mathbb{0} \sqcup \xi_2, \mathtt{w2} \mapsto \mathbb{0}]$$
$$\mathtt{Q1\ Sam.shift\ Q1\ ;\ Q1\ Sam.shift\ Q1} : [\mathtt{w1} \mapsto \xi_1, \mathtt{w2} \mapsto \xi_2] \to [\mathtt{w1} \mapsto \mathbb{0}, \mathtt{w2} \mapsto \mathbb{0}]$$

(we have simplified the final environment). That is, in this case, liquidity requires the analysis of 2-canonical computations to be assessed. (When the contract has no cycle, 1-canonical computations are sufficient to verify liquidity.) Since we have to consider cycles, in order to force termination, we restrict our analysis to κ-canonical abstract computations (with a finite value of κ).

The algorithm uses the set $\mathbb{T}^{\kappa}_{\mathtt{Q}'}$, for every state $\mathtt{Q}'$ of the contract that is reachable from $\mathtt{Q}$ – see step 1 of Table 4. Step 2 identifies the "critical pairs" $(\mathtt{Q}'', \overline{\mathtt{k}})$ such that there is a computation updating the assets $\overline{\mathtt{k}}$ and terminating in the state $\mathtt{Q}''$. Assume that $(\mathtt{Q}'', \overline{\mathtt{k}}) \notin \mathcal{Z}$. Then we must find $\mathtt{Q}'' \stackrel{\varphi'}{\rightsquigarrow} \mathtt{Q}''' : \Xi'' \to \Xi'''$

in $\mathbb{T}^{\kappa}_{\mathtt{Q}''}$ such that $\Xi'''(\overline{\mathtt{k}}) = \overline{\mathbb{0}}$ and the other assets in $\overline{\mathtt{h}}\backslash\overline{\mathtt{k}}$ are either $\mathbb{0}$ or equal to the corresponding value in Ξ''. That is, as for the efficient algorithms, assets $\overline{\mathtt{h}}\backslash\overline{\mathtt{k}}$ have not been modified by φ' and may be overlooked. Notice that these checks are exactly those defined in step 2.2. If no liquidity type $\mathtt{Q}'' \overset{\varphi'}{\leadsto} \mathtt{Q}''' : \Xi'' \to \Xi'''$ is found in $\mathbb{T}^{\kappa}_{\mathtt{Q}''}$ such that $\Xi'''(\overline{\mathtt{k}}) = \mathbb{0}$, the liquidity cannot be guaranteed and the algorithm exits stating that the contract is not liquid (which might be a false negative because the liquidity type might exist in $\mathbb{T}^{\kappa+1}_{\mathtt{Q}''}$).

For example, in case of the `Fill_Move` contract, the liquidity algorithm spots $\mathtt{Q0} \overset{\mathtt{Alice.fill}}{\leadsto} \mathtt{Q1} : \Xi \to \Xi'$ because $[\![\Xi'(\mathtt{h1})]\!] \neq [\![\Xi(\mathtt{h1})]\!]$. Therefore it parses the liquidity types in $\mathbb{T}^{1}_{\mathtt{Q1}}$ and finds the type $\mathtt{Q1} \overset{\varphi'}{\leadsto} \mathtt{Q2} : [\mathtt{h1} \mapsto \xi_1, \mathtt{h2} \mapsto \xi_2] \to [\mathtt{h1} \mapsto \mathbb{0}, \mathtt{h2} \mapsto \mathbb{0}]$, where $\varphi' = \mathtt{Q1\ Bob.move\ Q0\ ;\ Q0\ Bob.end\ Q2}$. Henceforth $(\mathtt{Q1}, \mathtt{h1})$ is added to $\mathcal{Z}$. There is also another problematic type: $\mathtt{Q1} \overset{\mathtt{Bob.move}}{\leadsto} \mathtt{Q0} : \Xi'' \to \Xi'''$, because $[\![\Xi'''(\mathtt{h1})]\!] \neq [\![\Xi''(\mathtt{h1})]\!]$. In this case, the liquidity type of the abstract computation `Q0 Bob.end Q2` (still in $\mathbb{T}^{1}_{\mathtt{Q0}}$) satisfies the liquidity constraint. We leave this check to the reader.

The correctness of the algorithm follows from Theorem 3. For example, assume that step 2.2 returns $\mathtt{Q} \overset{\varphi}{\leadsto} \mathtt{Q}' : \Xi \to \Xi'$ where $\Xi'(\overline{\mathtt{k}}) = \overline{\mathbb{0}}$ and $\Xi'(\overline{\mathtt{h}}\backslash\overline{\mathtt{k}}) = \Xi(\overline{\mathtt{h}}\backslash\overline{\mathtt{k}})$. Then, for every initial memory ℓ, the concrete computation corresponding to φ ends in a memory ℓ' such that $\mathbb{E}(\ell')|_{\overline{\mathtt{k}}} \leqslant [\![\sigma(\Xi'|_{\overline{\mathtt{k}}})]\!] = [\![\sigma([\overline{\mathtt{k}} \mapsto \overline{\mathbb{0}}])]\!] = [\overline{\mathtt{k}} \mapsto \overline{\mathbb{0}}]$ (for every σ) and $\mathbb{E}(\ell')|_{\overline{\mathtt{h}}\backslash\overline{\mathtt{k}}} \leqslant [\![\sigma(\Xi'|_{\overline{\mathtt{h}}\backslash\overline{\mathtt{k}}})]\!] = [\![\sigma([\overline{\mathtt{h}}\backslash\overline{\mathtt{k}} \mapsto \overline{\xi}])]\!] = [\overline{\mathtt{k}} \mapsto \overline{\mathbb{1}}]$, by taking a $\sigma = [\overline{\xi} \mapsto \mathbb{1}]$. As regards termination, the set $\mathcal{Z}$ increases at every iteration. When no other pair can be added to $\mathcal{Z}$ (and we have not already exited) the algorithm terminates by declaring the contract as liquid.

Proposition 2. *Let* $\vdash \mathtt{C} : \mathcal{L}$. *If the algorithm of Table 4 returns that* $\mathtt{C}$ *is liquid then it is liquid. Additionally, the algorithm always terminates.*

The computational cost of liquidity is the following. Let n be the size of the *Stipula* contract (the number of functions, prefixes and conditionals in the code), h be the number of assets, m be the number of states and m' be the number of functions. Then

- the cost of the inference of liquidity types is linear with respect to the size of the contract, *i.e.* $O(n)$;
- the length of κ-canonical traces starting in a state is less than $\kappa \times m'$; therefore the cardinality of $\mathbb{T}^{\kappa}_{\mathtt{Q}}$ is bounded by $\sum_{0 \leqslant i \leqslant \kappa \times m'} i!$. The cost of computing $\mathbb{T}^{\kappa}_{\mathtt{Q}}$, for every $\mathtt{Q}$, and the liquidity types of the elements therein is proportional to the number of κ-canonical traces, which are $N = m \times (\sum_{0 \leqslant i \leqslant \kappa \times m'} i!)$;
- the cost for verifying steps 2 and 3 of the algorithm is $O(N \times N \times 2^h)$ because, for every κ-canonical trace and every subset of $\overline{\mathtt{h}}$, we must look for a κ-canonical trace satisfying step 3.

Therefore the overall cost of the algorithm is $O(n + N + N^2 \times 2^h)$, which means it is $O(N^2)$, *i.e.* exponential with respect to m', assuming the other values are in linear relation with m'.

6 Related works

Liquidity properties have been put forward by Tsankov et al. in [14] as the property of a smart contract to always admit a trace where its balance is decreased (so, the funds stored within the contract do not remain frozen). Later, Bartoletti and Zunino in [3] discussed and extended this notion to a general setting – the Bitcoin language – that takes into account the strategy that a participant (which is possibly an adversary) follows to perform contract actions. More precisely, they observe that there are many possible flavours of liquidity, depending on which participants are assumed to be honest and on what are the strategies. In the taxonomy of [3, 2], the notion of liquidity that we study in this work is the so-called *multiparty strategyless liquidity*, which assumes that all the contract's parties cooperate by actually calling the functions provided by the contract. We notice that, without cooperation, there is no guarantee that a party that has the permission to call a function will actually call it.

Both [14] and [3, 2] adopt a model checking technique to verify properties of contracts. However, while [14] uses finite state models and the Uppaal model checker to verify the properties, [3, 2] targets infinite state system and reduces them to finite state models that are consistent and complete with respect to liquidity. This mean that the technique of [3, 2] is close to ours (we also target infinite state models and reduce to finite sets of abstract computations that over-approximate the real ones), even if we stick to a symbolic approach. Last, the above contributions and the ones we are aware of in the literature always address programs with one asset only (the contract balance). In this work we have understood that analyzing liquidity in programs with several different assets is way more complex than the case with a single asset.

A number of research projects are currently investigating the subject of resource-aware programming, as the prototype languages Obsidian [6] Nomos [9, 4], Marlowe [12] and Scilla [13]. As discussed in the empirical study [6], programming with linear types, ownership and assets is difficult and the presence of strong type systems can be an effective advantage. In fact, the above languages provide type systems that guarantee that assets are not accidentally lost, even if none of them address liquidity. More precisely, Obsidian uses types to ensure that owning references to assets cannot be lost unless they are explicitly disowned by the programmer. Nomos uses a linear type system to prevent the duplication or deletion of assets and amortized resource analysis to statically infer the resource cost of transactions. Marlowe [12], being a language for financial contracts, does not admit that money be locked forever in a contract. In particular, Marlowe's contracts have a finite lifetime and, at the end of the lifetime, any remaining money is returned to the participants. In other terms, all contracts are liquid by construction. In the extension of *Stipula* with events, the finite lifetime constraint can be explicitly programmed: a contract issues an event at the beginning so that at the timeout all the contract's assets are sent to the parties. Finally, Scilla is an intermediate-level language for safe smart contracts that is based on System F and targets a blockchain. It is unquestionable that a blockchain implementation of *Stipula* would bring in the advantages of a public and decentralised platform, such as traceability and the enforcement of contractual conditions. Being Scilla a minimalistic language with a formal semantics and a powerful type system, it seems an excellent candidate for implementing *Stipula*.

7 Conclusions

We have studied liquidity, a property of programs managing resources that pinpoints those programs not freezing any resource forever. In particular we have designed an algorithm for liquidity and demonstrated its correctness.

We are currently prototyping the algorithm. Our prototype takes in input an integer value κ and verifies liquidity by sticking to types in $\mathbb{T}^{\kappa}_{\mathtt{Q}}$. This allows us to tune the precision of the analysis according to the contract to verify. We are also considering optimisations that improve both the precision of the algorithms and the performance. For example, the precision of the checks $[\![\Xi'(\mathtt{k})]\!] \neq \mathbb{0}$ and $[\![\Xi'(\mathtt{k})]\!] \neq [\![\Xi(\mathtt{k})]\!]$ may be improved by noticing that the algebra of liquidity expressions is a distributive lattice with min ($\mathbb{0}$) and max ($\mathbb{1}$). This algebra has a complete axiomatization that we may implement (for simplicity sake, in this paper we have only used min-max rules – see definition of $[\![\mathbb{e}]\!]$). Other optimizations we are studying allow us to reduce the number of canonical computations to verify (such as avoiding repetition of cycles that modify only one asset).

Another research objective addresses the liquidity analysis in languages featuring *conditional transitions* and *events*, such as the full *Stipula* [7]. These primitives introduce *internal nondeterminism*, which may undermine state reachability and, for this reason, they have been dropped in this paper. In particular, our analysis might synthesize a computation containing a function whose execution depends on values of fields that never hold. Therefore the computation will never be executed (it is a false positive) and must be discarded (and the contract might be not liquid). To overcome these problems, we will try to complement our analysis with an (off-the-shelf) constraint solver technique that guarantees the reachability of states of the computations synthesized by our algorithms.

Acknowledgments. We thank the FACS 2022 referees for their careful reading and many constructive suggestions on the submitted paper.

References

1. Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, and Cosimo Laneve. Engineering virtualized services. In *NordiCloud '13*, volume 826 of *ACM International Conference Proceeding Series*, pages 59–63. ACM, 2013.
2. Massimo Bartoletti, Stefano Lande, Maurizio Murgia, and Roberto Zunino. Verifying liquidity of recursive bitcoin contracts. *Log. Methods Comput. Sci.*, 18(1), 2022.
3. Massimo Bartoletti and Roberto Zunino. Verifying liquidity of Bitcoin contracts. In *Principles of Security and Trust*, pages 222–247. Springer International Publishing, 2019.

4. Sam Blackshear, David L. Dill, Shaz Qadeer, Clark W. Barrett, John C. Mitchell, Oded Padon, and Yoni Zohar. Resources: A safe language abstraction for money. *CoRR*, abs/2004.05106, 2020. URL: `https://arxiv.org/abs/2004.05106`.
5. Harris Brakmić. *Bitcoin Script*, pages 201–224. Apress, Berkeley, CA, 2019.
6. Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Can advanced type systems be usable? An empirical study of ownership, assets, and typestate in Obsidian. *Proc. ACM Program. Lang.*, 4(OOPSLA):132:1–132:28, 2020.
7. Silvia Crafa, Cosimo Laneve, and Giovanni Sartor. Pacta sunt servanda: legal contracts in Stipula. Technical report, arXiv:2110.11069, 10 2021.
8. Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, USA, 2017.
9. A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar. Resource-aware session types for digital contracts. In *IEEE 34th CSF*, pages 111–126. IEEE Computer Society, 2021.
10. Mudabbir Kaleem, Anastasia Mavridou, and Aron Laszka. Vyper: A security comparison with Solidity based on common vulnerabilities. In *BRAINS 2020*, pages 107–111. IEEE, 2020.
11. Steve Klabnik and Carol Nichols. *The RUST programming language*. No Starch Press, 2019.
12. Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon J. Thompson. Marlowe: Implementing and analysing financial contracts on blockchain. In *Financial Cryptography and Data Security*, volume 12063 of *LNCS*, pages 496–511. Springer, 2020.
13. Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with scilla. *Proc. ACM Program. Lang.*, 3(OOPSLA):185:1–185:30, 2019.
14. Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical Security Analysis of Smart Contracts. In *Proc. ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82. ACM, 2018.