Liquidity analysis in resource-aware programming

Silvia Crafa
Dept. of Computer Science and Mathematics, University of Padova, Italy

Cosimo Laneve ORCID
Dept. of Computer Science and Engineering, University of Bologna, Italy

Abstract

Liquidity is a sensible security 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. By composing the input-output behaviours we derive liquidity types of computations that we use for designing two algorithms that verify two liquidity properties. We also demonstrate the correctness of the algorithms.

2012 ACM Subject Classification Theory of computation → Formalisms

Keywords and phrases Resource-aware programming, assets, separate-liquidity and liquidity, symbolic analysis, type inference, *Stipula*.

Digital Object Identifier 10.4230/LIPIcs..2022.

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 qualitative and quantitative service level specifications (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 [4], Solidity [7], Vyper [9]. Even new programming languages are defined with (linear) types for resources, such as Rust [10].

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 completely consumes 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.

In this paper we design a symbolic analyzer for liquidity. To this aim, we commit to a simple programming language where resources are assets (digital currencies, smart keys, non-fungible tokens, etc.) that may be moved with ad-hoc operators from one place (*e.g.* a wallet) to another. Programs 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. The target language of our study is *Stipula*, a domain-specific language that has been designed for programming *legal contracts* [6].

Our analyzer is built upon a type system that records the effects of functions on assets by using symbolic names. Since these effects detect whether a function empties an asset or not, it is therefore possible to design a simple algorithm verifying *separate-liquidity*: *if an asset* `h` *becomes not-empty in a state* `Q` *then there is a computation starting at* `Q` *and ending in* `Q'`

Functions	F	$::=$	$\_$	$\mid$	$@\mathtt{Q}\ \mathtt{A} : \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}]\{S\} \Rightarrow @\mathtt{Q}'\ F$			
Prefixes	P	$::=$	$E \to \mathtt{x}$	$\mid$	$E \to \mathtt{A}$	$\mid$	$c \times \mathtt{h} \multimap \mathtt{h}'$	$\mid$ $c \times \mathtt{h} \multimap \mathtt{A}$ $\quad // 0 \leqslant c \leqslant 1$
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)

whereby h *is empty in* Q′. It turns out that the separate-liquidity property can be assessed by collecting all the functions that may be invoked in any computation starting at Q, which are finitely many, and scanning their liquidity types.

While separate-liquidity is satisfactory in programs where assets are separated (no asset field is moved to another asset field, for instance when assets have different types), it is inadequate in unrestricted programs moving assets between fields. In this case, a liquidity analyzer should spot programs that empty assets' fields by moving the value from one field to another without never returning it. For these programs, assessing liquidity is more complex because one cannot restrict to single functions but has to consider *computations* because (different) assets may be emptied in several steps. As an example, the function that withdraws w1 (by sending its value to a party), moves w2 to w1, and w3 to w2 in sequence. This function empties w1, w2 and w3 if it is invoked three times (the first time it empties w3, the second time w2 and the third time w1).

The crucial issue is therefore designing a terminating algorithm for liquidity given that computations may be *infinitely many* because programs may have cycles. Instead of restricting to computations whose length is bound by a value, we have found more sensible the notion of κ-*canonical computations*, *i.e.* computations where functions occur at most κ-times (which are clearly finitely many, as well). This because a program might have so many steps that a function modifying assets could be invoked after a bound one might fix, say n, and therefore not be considered in the computations of length up-to n, thus obtaining a less precise result.

The correctness of separate-liquidity and liquidity relies on a *monotony* argument by which the (liquidity) type of the final state of a computation is always an over-approximation of the actual state. Therefore we can safely reduce our analysis to verifying if liquidity types of (κ-canonical) computations have an asset (for separate-liquidity) or all the assets (for liquidity) that are empty.

The structure of the paper is as follows. The simplified *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 algorithms for verifying separate-liquidity and liquidity. We end our contribution by discussing the related work in Section 6 and delivering our final remarks in Section 7.

2 The *Stipula* language

We use disjoint sets of names: *contract names*, ranged over by C, C′, ⋯; names referring to digital identities, called *parties*, ranged over by A, A′, ⋯; *function names* ranged over f, g, ⋯; *asset names*, ranged over by h, k, ⋯, to be used both as contract's assets and function's asset parameters; *non asset names*, ranged over x, y, ⋯, to be used both as contact'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′, ⋯ 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

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

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 the *initial state* is set to `Q`. The contract body also includes the sequence F of functions, whose syntax is defined in Table 1. It is assumed that the names of parties, fields, and assets 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 who is the caller party `A`, the state `Q` when the invocation is admitted 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 `}` ⇒ `@Q′` specifies the *statement part* S and the state `Q′` of the contract when the function execution terminates.

Statements S include the empty statement `_` and different prefixes followed by a continuation. Prefixes P use the two symbols $\rightarrow$ and $\multimap$ to differentiate operations on fields and on assets, respectively. The prefix $E \rightarrow \texttt{x}$ updates the field or the parameter `x` with the value of E; $E \rightarrow \texttt{A}$ sends the value of E to the party `A`; $c \times \texttt{h} \multimap \texttt{h}'$ subtracts the value of $c \times \texttt{h}$ to the asset `h` and adds it to `h′`, where c is a constant between 0 and 1 – *resources stored in assets can be moved but cannot be destroyed* –, $c \times \texttt{h} \multimap \texttt{A}$ subtracts the value of $c \times \texttt{h}$ to the asset `h` and transfers it to `A`. It follows that assets never have negative values. Statements also include a *conditionals* `if` $(E)\,\{\,S\,\}$ `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 include

- 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 comes 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 `!`.
- constant values of type asset that represent *divisible* resources (*e.g.* digital currencies). For simplicity, divisible asset constants are assumed to be identical to positive real numbers and, as discussed above, cannot assume negative values.

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

As an example, consider the `Fill_Move` contract

```
stipula C { parties Alice, Bob   assets wallet1, wallet2   init Q0
            @Q0 Alice: fill()[u,v]{     @Q1 Bob: move()[]{         @Q0 Bob: end()[]{
                u ⊸ wallet1                 wallet2 ⊸ wallet1          wallet1 ⊸ Bob
                v ⊸ wallet2             } ⇒ @Q0                    } ⇒ @Q2
            } ⇒ @Q1
}
```

where `Alice` fills `wallet1` and `wallet2` (with her own assets `u` and `v`, respectively) and `Bob` accumulates the overall values in `wallet1`. The contract terminates when `Bob` decides to grab the whole content of `wallet1`.

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

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

$$\text{[Value-Send]}\quad \frac{[\![E]\!]_\ell = v \qquad \ell(\texttt{A}) = A}{\texttt{C}(\texttt{Q}\,,\,\ell\,,\,E \to \texttt{A}\;\Sigma) \xrightarrow{v \to A} \texttt{C}(\texttt{Q}\,,\,\ell\,,\,\Sigma)}$$

$$\text{[Asset-Send]}\quad \frac{[\![c \times \texttt{h}]\!]_\ell = v \qquad \ell(\texttt{A}) = A \qquad [\![\texttt{h} - v]\!]_\ell = v'}{\texttt{C}(\texttt{Q}\,,\,\ell\,,\,c \times \texttt{h} \multimap \texttt{A}\;\Sigma) \xrightarrow{v \multimap A} \texttt{C}(\texttt{Q}\,,\,\ell[\texttt{h} \mapsto v']\,,\,\Sigma)}$$

$$\text{[Field-Update]}\quad \frac{[\![E]\!]_\ell = v}{\texttt{C}(\texttt{Q}\,,\,\ell\,,\,E \to \texttt{x}\;\Sigma) \longrightarrow \texttt{C}(\texttt{Q}\,,\,\ell[\texttt{x} \mapsto v]\,,\,\Sigma)}$$

$$\text{[Asset-Update]}\quad \frac{[\![c \times \texttt{h}]\!]_\ell = v \qquad [\![\texttt{h} - v]\!]_\ell = v' \qquad [\![\texttt{h}' + v]\!]_\ell = v'' \qquad \ell' = \ell[\texttt{h} \mapsto v', \texttt{h}' \mapsto v'']}{\texttt{C}(\texttt{Q}\,,\,\ell\,,\,c \times \texttt{h} \multimap \texttt{h}'\;\Sigma) \longrightarrow \texttt{C}(\texttt{Q}\,,\,\ell'\,,\,\Sigma)}$$

$$\text{[Cond-true]}\quad \frac{[\![E]\!]_\ell = \texttt{true}}{\texttt{C}(\texttt{Q}\,,\,\ell\,,\,(\texttt{if}\,(E)\,\{\,S\,\}\,\texttt{else}\,\{\,S'\,\}\;\Sigma) \longrightarrow \texttt{C}(\texttt{Q}\,,\,\ell\,,\,S\;\Sigma)}$$

$$\text{[Cond-false]}\quad \frac{[\![E]\!]_\ell = \texttt{false}}{\texttt{C}(\texttt{Q}\,,\,\ell\,,\,\texttt{if}\,(E)\,\{\,S\,\}\,\texttt{else}\,\{\,S'\,\}\;\Sigma) \longrightarrow \texttt{C}(\texttt{Q}\,,\,\ell\,,\,S'\;\Sigma)}$$

Table 2 The transition relation of *Stipula*

A contract named `C` is invoked by $\texttt{C}(\overline{A}, \overline{u})$ that corresponds to a configuration (see below) where the initial state is the one specified in the `init` clause, the parties are $\overline{A}$, the fields' values are $\overline{u}$ and the assets' values are 0.

3 Semantics

The meaning of *Stipula* primitives is defined operationally by means of a transition relation between configurations. *Configurations*, ranged over by $\mathbb{C}$, $\mathbb{C}'$, $\cdots$, are tuples $\texttt{C}(\texttt{Q}\,,\,\ell\,,\,\Sigma)$ where

- `C` is the contract name and `Q` is a state of its;
- ℓ, called *memory*, is a mapping from names (parties, fields, assets and function's parameters) to values. The values of parties are noted with italic fonts $A, A', \cdots$. These names cannot be passed as function's parameters and cannot be hard-coded into the source contracts, since they do not belong to expressions;
- Σ is the (possibly empty) residual of a function body, *i.e.* Σ is either `_` or a term $S \Rightarrow \texttt{@Q}$.

The definition of the transition relation uses the auxiliary *total evaluation function* $[\![E]\!]_\ell$ that returns the value of E in the memory ℓ such that:

- $[\![v]\!]_\ell = v$ for values, $[\![X]\!]_\ell = \ell(X)$ for names of assets, fields and parameters (X always belongs to the domain of ℓ).
- let *uop* and *op* be the semantic operations corresponding to `uop` and `op`, then $[\![\texttt{uop}\,E]\!]_\ell = uop\,v$, $[\![E\,\texttt{op}\,E']\!]_\ell = v\;op\;v'$ with $[\![E]\!]_\ell = v, [\![E']\!]_\ell = v'$. Because of the restrictions on the language, `uop` and `op` are always defined.

The formal definition of the *Stipula* transition relation, noted $\mathbb{C} \xrightarrow{\mu} \mathbb{C}'$, is given in Table 2 where μ is either empty or $A : \texttt{f}(\overline{u})[\overline{v}]$ or $v \to A$ or $v \multimap A$. The most relevant rules are spelt out below. Rule [Function] defines invocations: the label specifies the party A performing the invocation and the function name `f` with the actual parameters. The transition may occur provided (*i*) the contract is in the state `Q` that admits invocations of `f` from A and (*ii*) it is *idle*. Rule [State-Change] says that a contract changes state when the execution of the statement in the function's body terminates.

As regards statements, we only discuss [Asset-Send] and [Asset-Update] because the other rules are standard. Rule [Asset-Send] delivers part of an asset $\mathtt{h}$ 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 $\mathtt{h}$ to an asset $\mathtt{h}'$. For this reason, the final memory becomes $\ell[\mathtt{h} \mapsto v', \mathtt{h}' \mapsto v'']$, where $v' = \ell(\mathtt{h}) - v$ and $v'' = \ell(\mathtt{h}') + v$.

Given the invocation $\mathtt{C}(\overline{A}, \overline{u})$ of a contract with parties $\overline{\mathtt{A}}$, fields $\overline{\mathtt{x}}$ and assets $\overline{\mathtt{h}}$, its *initial configuration* is

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

where $\mathtt{Q}$ is the state specified in the `init` clause. Notice that the initial value of assets is 0. Below we use the following notation and terminology:

- We write $\mathbb{C} \Longrightarrow \mathbb{C}'$ if there is a *sequence of transitions* such that $\mathbb{C} \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_n} \mathbb{C}'$; $\mathbb{C} \Longrightarrow \mathbb{C}'$ will be called *computation*.
- Configurations such as $\mathtt{C}(\mathtt{Q}, \ell, \_)$, *i.e.* there is no statement to execute, are called *idle*.

An important property of our language guarantees that the invocation of a function never fails. This property is actually an immediate consequence of the restrictions we have (according to which the evaluation of expressions can never rise an error).

▸ **Theorem 1** (Progress). *Let* $\mathtt{C}$ *be a Stipula contract and* $\mathbb{C}$ *be the initial configuration. If* $\mathbb{C} \Longrightarrow \mathbb{C}' \xrightarrow{A.\mathtt{f}(\overline{u})[\overline{v}]} \mathtt{C}(\mathtt{Q}, \ell, S)$ *then there is* $\mathtt{C}(\mathtt{Q}', \ell', \_)$ *such that* $\mathtt{C}(\mathtt{Q}, \ell, S) \Longrightarrow \mathtt{C}(\mathtt{Q}', \ell', \_)$.

We conclude with the definition of liquidity. Let $\ell(\overline{\mathtt{h}}) > \overline{0}$ 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 2** (Liquidity). *A Stipula contract with assets* $\overline{\mathtt{h}}$ *and initial configuration* $\mathbb{C}$ *is* separate-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* $\mathtt{k} \in \overline{\mathtt{h}}$ *and* $\ell(\mathtt{k}) > 0$ *then there is* $\mathtt{C}(\mathtt{Q}', \ell', \_)$ *such that* $\mathtt{C}(\mathtt{Q}, \ell, \_) \Longrightarrow \mathtt{C}(\mathtt{Q}', \ell', \_)$ *and* $\ell'(\mathtt{k}) = 0$.

The contract 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* $\mathtt{C}(\mathtt{Q}', \ell', \_)$ *such that* $\mathtt{C}(\mathtt{Q}, \ell, \_) \Longrightarrow \mathtt{C}(\mathtt{Q}', \ell', \_)$ *and* $\ell'(\overline{\mathtt{h}}) = \overline{0}$.

The properties of separate-liquidity and liquidity are incomparable: a separate-liquid contract may be not liquid and, conversely, a liquid contract may be not separate-liquid. For instance, take a flip-flop contract with two assets `u` and `v` and two functions: the first one moves `u` to `v` and the second one moves `v` to `u`. This contract is separate-liquid but not liquid. A contract that is liquid but not separate-liquid is the `Ugly` contract in Section 5. It turns out that separate-liquidity and liquidity coincide on contracts with so-called separate assets, *i.e.* contracts that do not move asset fields to other asset fields (for example because assets have different types).

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

$$\text{[L-SEND]}\ \Xi \vdash E \to \mathtt{A} : \Xi \qquad \text{[L-UPDATE]}\ \Xi \vdash E \to \mathtt{x} : \Xi$$

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

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

$$\text{[L-ZERO]}\ \Xi \vdash \_ : \Xi \qquad \text{[L-SEQ]}\ \frac{\Xi \vdash P : \Xi' \quad \Xi' \vdash S : \Xi''}{\Xi \vdash P\ S : \Xi''} \qquad \text{[L-COND]}\ \frac{\Xi \vdash S : \Xi' \quad \Xi \vdash S' : \Xi'' \quad \Xi' \sqcup \Xi'' \vdash S''' : \Xi'''}{\Xi \vdash \mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S'\,\}\ S''' : \Xi'''}$$

$$\text{[L-FUNCTION]}\ \frac{\overline{\xi'}\ \mathit{fresh} \qquad \Xi[\overline{\mathtt{h}'} \mapsto \overline{\xi'}] \vdash S : \Xi'}{\Xi \vdash @\mathtt{Q}\ \mathtt{A} : \mathtt{f}(\overline{\mathtt{x}'})[\overline{\mathtt{h}'}]\{\,S\,\} \Rightarrow @\mathtt{Q}' : \mathtt{Q}\ \mathtt{A.f}\ \mathtt{Q}' : \Xi[\overline{\mathtt{h}'} \mapsto \overline{\mathbb{1}}] \to \Xi'\{\overline{\mathbb{1}}/_{\overline{\xi'}}\}}$$

$$\text{[L-PROGRAM]}\ \frac{\overline{\xi}\ \mathit{fresh} \qquad \left([\overline{\mathtt{h}} \mapsto \overline{\xi}] \vdash \mathtt{F}_i : \mathcal{L}_i\right)^{i \in 1..n}}{\vdash \mathtt{stipula\ C}\ \{\mathtt{parties}\ \overline{\mathtt{A}}\ \mathtt{fields}\ \overline{\mathtt{x}}\ \mathtt{assets}\ \overline{\mathtt{h}}\ \mathtt{init}\ \mathtt{Q}\ \mathtt{F}_1 \cdots \mathtt{F}_n\ \} : \bigcup_{i \in 1..n} \mathcal{L}_i}$$

Table 3 The Liquidity type system of *Stipula*

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:

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

  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*.
- *environments* Ξ 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* $\mathtt{Q\,A.f\,Q}' : \Xi \to \Xi'$ where $\Xi \to \Xi'$ records the liquidity effects of fully executing the body of $\mathtt{Q\,A.f\,Q}'$.
- *judgments* $\Xi \vdash E : \mathbb{e}$ for expressions, $\Xi \vdash S : \Xi'$ for statements and $\Xi \vdash @\mathtt{Q\ A} : \mathtt{f}(\overline{\mathtt{x}})[\overline{\mathtt{h}'}]\ \{\,S\,\} \Rightarrow @\mathtt{Q}' : \mathcal{L}$ for function definitions, where $\mathcal{L}$ is a liquidity function type.

The liquidity type system is defined in Table 3; below we discuss the most relevant rules.

Asset movements have four rules – [L-AUPDATE], [L-EXPAUPD], [L-ASEND] and [L-EXPASEND] – 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 $\Xi(\mathtt{h}) \sqcup \Xi(\mathtt{h}')$. 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). The expression E is overlooked by the analyzer.

The rule for *Stipula* contracts is [L-PROGRAM]; 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. For example, the set $\mathcal{L}$ of the `Fill_Move` contract contains the following liquidity types:

$$\begin{array}{rl}
\mathtt{Q0\ Alice.fill\ Q1}: & [\mathtt{wallet1} \mapsto \xi_1, \mathtt{wallet2} \mapsto \xi_2, \mathtt{u} \mapsto \mathbb{1}, \mathtt{v} \mapsto \mathbb{1}] \\
& \qquad \to [\mathtt{wallet1} \mapsto \xi_1 \sqcup \mathbb{1}, \mathtt{wallet2} \mapsto \xi_2 \sqcup \mathbb{1}, \mathtt{u} \mapsto \mathbb{0}, \mathtt{v} \mapsto \mathbb{0}] \\
\mathtt{Q1\ Bob.move\ Q0}: & [\mathtt{wallet1} \mapsto \xi_1, \mathtt{wallet2} \mapsto \xi_2] \to [\mathtt{wallet1} \mapsto \xi_1 \sqcup \xi_2, \mathtt{wallet2} \mapsto \mathbb{0}] \\
\mathtt{Q0\ Bob.end\ Q2}: & [\mathtt{wallet1} \mapsto \xi_1, \mathtt{wallet2} \mapsto \xi_2] \to [\mathtt{wallet1} \mapsto \mathbb{0}, \mathtt{wallet2} \mapsto \xi_2]
\end{array}$$

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 $\{{}^{\overline{\mathtt{e}}}/_{\overline{\chi}}\}$. 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 $[\![\mathtt{e}]\!]$ be the *partial evaluation* of $\mathtt{e}$ by applying the commutativity axioms of $\sqcup$ and $\sqcap$ and the axioms $\mathbb{0} \sqcup \mathtt{e} = \mathtt{e}$, $\mathbb{0} \sqcap \mathtt{e} = \mathbb{0}$, $\mathbb{1} \sqcup \mathtt{e} = \mathbb{1}$, $\mathbb{1} \sqcap \mathtt{e} = \mathtt{e}$. More precisely

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

  Notice that if $\mathtt{e}$ is ground then $[\![\mathtt{e}]\!]$ 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:

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

▸ **Theorem 3** (Correctness of liquidity labels). *Let* $\mathtt{C}$ *be a Stipula contract with assets* $\overline{\mathtt{h}}$ *and liquidity function types* $\mathcal{L}$*. If* $\mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,\_) \xrightarrow{A.\mathtt{f}(\overline{u})[\overline{v}]} \mathtt{C}(\mathtt{Q}\,,\,\ell'\,,\,S \Rightarrow \mathtt{Q}') \Longrightarrow \mathtt{C}(\mathtt{Q}'\,,\,\ell''\,,\,\_ \Rightarrow @\mathtt{Q}'')$ *where the transitions in* $\Longrightarrow$ *are not instances of rule* [FUNCTION] *and* $\mathtt{Q}\ \mathtt{A}.\mathtt{f}\ \mathtt{Q}' : \Xi \to \Xi' \in \mathcal{L}$ *then:*

1. $\mathbb{E}(\ell)|_{\overline{\mathtt{h}}} = \mathbb{E}(\ell')|_{\overline{\mathtt{h}}}$;
2. $\mathbb{E}(\ell') \vdash S : \Xi''$;
3. *there is a ground substitution* σ *such that* $\mathbb{E}(\ell')|_{dom(\Xi)} \leqslant [\![\sigma(\Xi)]\!]$ *and* $[\![\Xi''|_{dom(\Xi)}]\!] \leqslant [\![\sigma(\Xi')]\!]$;
4. $\mathbb{E}(\ell'')|_{dom(\Xi)} \leqslant [\![\Xi'']\!]$.

A basic notion of our theory is the one of abstract computation.

▸ **Definition 4.** *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 will use the notation* $\mathtt{Q} \overset{\varphi}{\rightsquigarrow} \mathtt{Q}'$ *to highlight the initial and final states of* φ*. The abstract computation of* $\big(\mathtt{C}(\mathtt{Q}_i\,,\,\ell_i\,,\,\_) \xrightarrow{A_i:\mathtt{f}_i(\overline{u_i})[\overline{v_i}]} \Longrightarrow \mathtt{C}(\mathtt{Q}_{i+1}\,,\,\ell_{i+1}\,,\,\_)\big)^{i\in 1..n}$ *where the transitions in* $\Longrightarrow$ *are not instances of* [FUNCTION] *is* $\{\ \mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1}\ \}^{i\in 1..n}$.

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

Abstract computation have liquidity types, as well; we define them by merging the final environment of a function with the initial environment of the next one.

▸ **Definition 5** (Liquidity type of an abstract computation). *Let* $\overline{\mathtt{h}}$ *be the assets of a Stipula contract and* $\mathcal{L}$ *be the set of liquidity function types. 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 drop in the liquidity types of computations.

To keep simple the operational semantics of *Stipula* in Table 2, we do not remove garbage names in the memories (the formal parameters of functions once the functions have terminated). Therefore a memory ℓ retains such names, which do not exist in environments of the liquidity types of abstract computations. For this reason, in the following statement, we restrict the inequalities to names in the domain of environments.

▸ **Theorem 6** (Correctness of an abstract computation). *Let* $\mathtt{C}$ *be a Stipula contract with liquidity function types* $\mathcal{L}$ *and* $\mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1} : \Xi_i \to \Xi'_i \in \mathcal{L}$ *for every* $i \in 1..n$*. If*

$$\Big(\mathtt{C}(\mathtt{Q}_i\,,\,\ell_i\,,\,\_) \xrightarrow{A_i:\mathtt{f}_i(\overline{u_i})[\overline{v_i}]} \Longrightarrow \mathtt{C}(\mathtt{Q}_{i+1}\,,\,\ell_{i+1}\,,\,\_)\Big)^{i\in 1..n}$$

where

Let $\mathtt{Q}$ be the initial state of $\mathtt{C}$ and let $\overline{\mathtt{h}}$ be the assets of $\mathtt{C}$:

step 1. Compute $\mathbb{Q}_{\mathtt{Q}'}$ for every $\mathbb{Q}_{\mathtt{Q}'}$ reachable from $\mathtt{Q}$; set $\mathcal{Z} = \varnothing$.

step 2. For every $\mathtt{Q}'\ \mathtt{A.f}\ \mathtt{Q}'' \in \mathbb{Q}_{\mathtt{Q}}$, take the liquidity type $\mathtt{Q}'\ \mathtt{A.f}\ \mathtt{Q}'' : \Xi \to \Xi'$ and verify whether there is $\mathtt{k} \in \overline{\mathtt{h}}$ such that (i) $[\![\Xi'(\mathtt{k})]\!] \neq \mathbb{0}$ and $[\![\Xi'(\mathtt{k})]\!] \neq [\![\Xi(\mathtt{k})]\!]$ and (ii) $(\mathtt{Q}'', \mathtt{k}) \notin \mathcal{Z}$:
- if there is $(\mathtt{Q}'', \mathtt{k}) \notin \mathcal{Z}$ then go to **step 3**, otherwise exit: *the contract is separate-liquid.*

step 3. verify whether there is $\mathtt{Q1}\ \mathtt{B.g}\ \mathtt{Q2} \in \mathbb{Q}_{\mathtt{Q}''}$ such that $\mathtt{Q1}\ \mathtt{B.g}\ \mathtt{Q2} : \Xi_1 \to \Xi_2$ and $[\![\Xi_2(\mathtt{k})]\!] = \mathbb{0}$. If this is the case, add $(\mathtt{Q}'', \mathtt{k})$ to $\mathcal{Z}$ and reiterate **step 2**, otherwise exit: *the contract is not separate-liquid.*

Table 4 The separate-liquidity algorithm – $\mathcal{Z}$ contains pairs $(\mathtt{Q}, \mathtt{h})$

- *the transitions in* $\Longrightarrow$ *are not instances of rule* [FUNCTION]
- *and the abstract computation* $\varphi = \{\ \mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1}\ \}^{i\in 1..n}$ *has 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 algorithms for separate-liquidity and liquidity

Analyzing the liquidity of a *Stipula* contract amounts to verifying the two constraints of Definition 2 (both for separate-liquidity and liquidity). In both cases, checking constraint 1 is not difficult: for every transition $\mathtt{Q}\ \mathtt{A.f}\ \mathtt{Q}'$ of the abstract automaton of a contract with assets $\overline{\mathtt{h}}$, we consider its liquidity type $\Xi \to \Xi'$ and verify whether, for every $\mathtt{k} \notin \overline{\mathtt{h}}$, $[\![\Xi'(\mathtt{k})]\!] = \mathbb{0}$. Since there are finitely many transitions, this analysis is exhaustive. The correctness of this analysis 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 3.

On the contrary, verifying the constraints 2 of Definition 2 is harder because the transition system of a *Stipula* contract may be complex (cycles, absence of final states, nondeterminism). Before presenting the two algorithms, we define the notions of reachable function and reachable state. Let $\mathbb{Q}_{\mathtt{Q}}$ be the least set such that

1. if $\mathtt{Q}\ \mathtt{A.f}\ \mathtt{Q}'$ is a function in $\mathtt{C}$ then $\mathtt{Q}\ \mathtt{A.f}\ \mathtt{Q}' \in \mathbb{Q}_{\mathtt{Q}}$;
2. if $\mathtt{Q}'\ \mathtt{B.g}\ \mathtt{Q}'' \in \mathbb{Q}_{\mathtt{Q}}$ and $\mathtt{Q}''\ \mathtt{B}'.\mathtt{g}'\ \mathtt{Q}'''$ is a function in $\mathtt{C}$ then $\mathtt{Q}''\ \mathtt{B}'.\mathtt{g}'\ \mathtt{Q}''' \in \mathbb{Q}_{\mathtt{Q}}$.

That is $\mathbb{Q}_{\mathtt{Q}}$ contains all the functions $\mathtt{Q}'\ \mathtt{B.g}\ \mathtt{Q}''$ *reachable* from computations starting at $\mathtt{Q}$. In this case, we also say that both $\mathtt{Q}'$ and $\mathtt{Q}''$ are *reachable* from $\mathtt{Q}$. Notice that, (i) $\mathbb{Q}_{\mathtt{Q}}$ is finite, (ii) if $\mathbb{Q}_{\mathtt{Q}'}$ is reachable from $\mathtt{Q}$, then $\mathbb{Q}_{\mathtt{Q}'} \subseteq \mathbb{Q}_{\mathtt{Q}}$, (iii) if no function starts at $\mathtt{Q}$ then $\mathbb{Q}_{\mathtt{Q}} = \varnothing$ and (iv) if $\mathtt{Q}$ is the initial state and every state is reachable, then $\mathbb{Q}_{\mathtt{Q}}$ contains all the functions of the contract. For example, in the `Fill_Move` contract,

$$\mathbb{Q}_{\mathtt{Q0}} = \{\ \mathtt{Q0\ Alice.fill\ Q1},\ \mathtt{Q1\ Bob.move\ Q0},\ \mathtt{Q0\ Bob.end\ Q2}\ \} \qquad \mathbb{Q}_{\mathtt{Q2}} = \varnothing\ .$$

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.

Separate-liquidity

The algorithm for verifying separate-liquidity is defined in Table 4. The set $\mathcal{Z}$ contains all the verifications we have done; that is, if $(\mathtt{Q}, \mathtt{h}) \in \mathcal{Z}$ then we have already verified that there is a function in $\mathbb{Q}_{\mathtt{Q}}$ such that $\mathtt{h}$ is $\mathbb{0}$ in its final environment. The analysis ends when $\mathcal{Z}$ does not increase anymore. We use the liquidity type of functions, *e.g.* $\mathtt{Q}\ \mathtt{A.f}\ \mathtt{Q}' : \Xi \to \Xi'$, because Theorem 3 guarantees that, if $[\![\Xi'(\mathtt{h})]\!] = \mathbb{0}$ then every execution of $\mathtt{A.f}$ will lead to a memory ℓ with $\ell(\mathtt{h}) = 0$.

It is worth to notice that, in the step 2, the algorithm verifies $[\![\Xi'(\mathtt{k})]\!] \neq \mathbb{0}$ and $[\![\Xi'(\mathtt{k})]\!] \neq [\![\Xi(\mathtt{k})]\!]$ to check whether a function $\mathtt{Q'\,A.f\,Q''}$ *updates* the value of an asset (with a value different from $\mathbb{0}$), while the definition of separate-liquidity looks for assets' values greater than 0. To explain the reason, we observe that our arguments are *local*: it is possible that $[\![\Xi'(\mathtt{k})]\!] \neq \mathbb{0}$ because the function has not updated the asset (therefore $[\![\Xi'(\mathtt{k})]\!] = [\![\Xi(\mathtt{k})]\!] = \xi$, for some symbolic name ξ). In this case, separate-liquidity issues regarding `k` must be searched elsewhere, rather than in $\mathtt{Q'\,A.f\,Q''}$. It is also worth to notice that we are taking $[\![\Xi'(\mathtt{h})]\!]$ instead of $\Xi'(\mathtt{h})$ in order to discard ground expressions that are syntactically different but semantically equivalent to $\mathbb{0}$.

Once a state $\mathtt{Q''}$ is found where an asset `k` has been updated and the pair $(\mathtt{Q''}, \mathtt{k})$ does not belong to $\mathcal{Z}$, separate-liquidity amounts to look for a function in $\mathbb{Q}_{\mathtt{Q''}}$ that empties `k`. This is what is specified in step 3. For example, consider the `Fill_Move` contract in Section 2. There are two problematic liquidity types `Q0 Alice.fill Q1` : $\Xi_1 \to \Xi_2$ and `Q1 Bob.move Q0` : $\Xi'_1 \to \Xi'_2$ (the environments have been computed in Section 4). In fact $\Xi_2(\mathtt{wallet1}) \neq \Xi_1(\mathtt{wallet1})$, $\Xi_2(\mathtt{wallet2}) \neq \Xi_1(\mathtt{wallet2})$ and $\Xi'_2(\mathtt{wallet1}) \neq \Xi'_1(\mathtt{wallet1})$ (all the values are not-$\mathbb{0}$). Therefore we have to find two liquidity types of functions that are reachable from `Q1` and one that is reachable from `Q0` that empty the corresponding assets. The reader is invited to verify that these are the types of `Q1 Bob.move Q0`, `Q0 Bob.end Q2` and `Q0 Bob.end Q2`, respectively.

The correctness of the algorithm for separate-liquidity is a consequence of the Progress Theorem and Theorem 6. Termination follows by the fact that $\mathcal{Z}$ may contain a finite number of pairs and every $\mathbb{Q}_{\mathtt{Q}}$ is finite.

Liquidity

Verifying the liquidity requirement – *i.e.* there is a reachable configuration whose memory ℓ is such that $\ell(\overline{\mathtt{h}}) = \overline{0}$ – is more complex than the one for separate-liquidity because a tuple of assets may become 0 during *a computation*, rather than just one transition. For example, in the `Fill_Move` contract, the transition `Q0 Alice.fill Q1` modifies the (symbolic) values of `wallet1` and `wallet2`. In order to determine a state where the two assets become $\mathbb{0}$ we must consider the computation `Q1 Bob.move Q0 ; Q0 Bob.end Q2`, where `Q1 Bob.move Q0` turns `wallet2` to $\mathbb{0}$ and `Q0 Bob.end Q2` turns `wallet1` to $\mathbb{0}$. However, the liquidity analysis cannot disregard computations where functions are repeated (because of cycles). Consider, for instance, the `Ugly` contract that has two assets `w1` and `w2`, and the following functions:

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

This contract has the problematic transition `Q0 Mark.get Q1` because the asset `w2` is filled with a parameter value. While `w2` is emptied by `Q1 Sam.shift Q1`, this function fill `w1` and there is no other function emptying `w1`. That is, the contract is not separate-liquid. 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, the liquidity requires the analysis of 2-canonical computations. (If the contract has no cycle, 1-canonical computations are sufficient to assess liquidity.) Since we have to consider repetitions, in order to force

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

step 1. Compute $\mathbb{T}^{\kappa}_{\mathtt{Q}'}$ for every $\mathtt{Q}'$; let $\mathbb{S}$ be set of $(\mathtt{Q}'', \overline{\mathtt{k}})$ such that $\varnothing \subsetneq \overline{\mathtt{k}} \subseteq \overline{\mathtt{h}}$ and, for every $\mathtt{k}' \in \overline{\mathtt{k}}$, $[\![\Xi'(\mathtt{k}')]\!] \neq \mathbb{0}$ and $[\![\Xi'(\mathtt{k}')]\!] \neq [\![\Xi(\mathtt{k}')]\!]$ and, for every $\mathtt{k}'' \in \overline{\mathtt{h}}\backslash\overline{\mathtt{k}}$, $[\![\Xi'(\mathtt{k}'')]\!] = [\![\Xi(\mathtt{k}'')]\!]$; set $\mathcal{Z} = \varnothing$.

step 2.
- if $\mathcal{Z} = \mathbb{S}$ then exit: *the contract is liquid.*
- otherwise take $(\mathtt{Q}'', \overline{\mathtt{k}}) \in \mathbb{S}\backslash\mathcal{Z}$ and go to **step 3**.

step 3. verify whether there is a *liquidity type* $\Xi'' \to \Xi'''$ 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 5 The liquidity algorithm – $\mathcal{Z}$ contains pairs $(\mathtt{Q}, \overline{\mathtt{k}})$

termination of the liquidity algorithm, we restrict our analysis to κ-canonical abstract computations (with a finite value of κ). Let $\mathbb{T}^{\kappa}_{\mathtt{Q}}$ be *the set of liquidity types* of the κ-canonical computations starting at $\mathtt{Q}$ in the contract (which is implicit).

The algorithm for liquidity is defined in Table 5. It uses the set $\mathbb{T}^{\kappa}_{\mathtt{Q}'}$, for every state $\mathtt{Q}$ of the contract – see step 1. Step 1 also collect the "critical pairs" $(\mathtt{Q}'', \overline{\mathtt{k}})$ such that there is a function updating the assets $\overline{\mathtt{k}}$ and terminating in the state $\mathtt{Q}''$. This is the set $\mathbb{S}$; comments about it are similar to the step 2 of the separate-liquidity; therefore they are omitted. Next, assume that $(\mathtt{Q}'', \overline{\mathtt{k}}) \in \mathbb{S}\backslash\mathcal{Z}$. Then we must find a liquidity type $\Xi'' \to \Xi'''$ 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 separate-liquidity, our arguments are *local*: assets $\overline{\mathtt{h}}\backslash\overline{\mathtt{k}}$ have not been modified by the function that has produced $(\mathtt{Q}'', \overline{\mathtt{k}})$ and may be overlooked (if they where not empty in input then there must be other pairs in $\mathbb{S}$ where the assets appear). Notice that the foregoing checks are exactly those defined in step 3. We also notice that, if $\Xi'' \to \Xi'''$ exists, let φ be a concrete computation corresponding to (the abstract computation of) this type and ℓ' be its final memory. By Theorem 6, $\mathbb{E}(\ell')|_{\overline{\mathtt{h}}} \leqslant [\![\Xi''']\!]$. Therefore, if $[\![\Xi'''(\overline{\mathtt{k}})]\!] = \mathbb{0}$ then $\ell'(\overline{\mathtt{k}}) = \overline{0}$, this the correctness of the algorithm. On the other hand, if no liquidity type $\Xi'' \to \Xi'''$ is found in $\mathbb{T}^{\kappa}_{\mathtt{Q}''}$ such that $\Xi'''(\overline{\mathtt{k}}) = \mathbb{0}$, we cannot guarantee that the contract is liquid and we exit stating that the contract is not liquid (which might be a false negative because the liquidity type might exist in $\mathbb{T}^{\kappa+1}_{\mathtt{Q}''}$).

At each iteration $\mathcal{Z}$ increases. 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. For example, in case of the `Fill_Move` contract, the liquidity algorithm spots $\mathtt{Q0\ Alice.fill\ Q1} : \Xi \to \Xi'$ because $[\![\Xi'(\mathtt{wallet1})]\!] \neq [\![\Xi(\mathtt{wallet1})]\!]$. Therefore it parses the liquidity types in $\mathbb{T}^{1}_{\mathtt{Q1}}$ and finds the type of $\mathtt{Q1\ Bob.move\ Q0\ ;\ Q0\ Bob.end\ Q2}$, which is $[\mathtt{wallet1} \mapsto \xi_1, \mathtt{wallet2} \mapsto \xi_2] \to [\mathtt{wallet1} \mapsto \mathbb{0}, \mathtt{wallet2} \mapsto \mathbb{0}]$. There is also another problematic function, $\mathtt{Q1\ Bob.move\ Q0} : \Xi'' \to \Xi'''$, because $[\![\Xi'''(\mathtt{wallet1})]\!] \neq [\![\Xi''(\mathtt{wallet1})]\!]$. In this case, the liquidity type of the abstract computation $\mathtt{Q0\ Bob.end\ Q2}$ (still in $\mathbb{T}^{1}_{\mathtt{Q0}}$) satisfies the liquidity constraint. We leave this check to the reader.

6 Related works

Liquidity properties have been put forward by Tsankov et al. in [12] 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 [2] 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 [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's protocol. Notice that the fact that each function is only called at the right time is guaranteed by the state-based programming of *Stipula*, however nothing ensures that a party that has the permission to call a function will actually call it. Hence we target the multiparty strategyless property checking that, for all contract runs, there exists a collaborative strategy of all participants that never freezes funds.

Both the works [12, 2] adopt a model checking technique to verify properties of contracts. However, while [12] uses finite state models and the Uppaal model checker to verify the properties, [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 [2] is close to our one (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 [5] Nomos [8, 3] and Marlowe [11]. As discussed in the empirical study [5], 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. Finally, Marlowe [11], 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.

7 Conclusions

We have studied a property of programs managing resources that pinpoints those programs not freezing any resource forever, called liquidity. In particular we have designed two algorithms that verify two different liquidity properties. The correctness of the algorithms has also been addressed.

We are currently prototyping the two algorithms. In case of liquidity, our prototype takes in input an integer value κ and verifies liquidity by sticking to types in $\mathbb{T}^{\kappa}_{\mathbb{Q}}$. 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 $[\![\mathtt{e}]\!]$). As regards performance, the liquidity algorithm may be improved by using the pairs $(\mathtt{Q}', \overline{\mathtt{k}})$ that are already in $\mathcal{Z}$: if we have to check that $(\mathtt{Q}, \{\mathtt{k}_1, \mathtt{k}_2\})$ and we have a computation $\mathtt{Q} \overset{\varphi}{\rightsquigarrow} \mathtt{Q}'$ with liquidity type $\Xi \to \Xi'$ and $\Xi'(\mathtt{k}_1) = \mathbb{0}$ and $(\mathtt{Q}', \{\mathtt{k}_2\}) \in \mathcal{Z}$ then we can safely add $(\mathtt{Q}, \{\mathtt{k}_1, \mathtt{k}_2\})$ to $\mathcal{Z}$ because a computation emptying $\mathtt{k}_2$ that starts at $\mathtt{Q}'$ has been already found. Again, the current algorithm has been chosen because of its simplicity. 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* [6]. These primitives introduce *internal nondeterminism*, which may undermine state reachability and, for this reason, they have been drop 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 intend to investigate how 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.

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 and Roberto Zunino. Verifying liquidity of Bitcoin contracts. In *Principles of Security and Trust*, pages 222–247. Springer International Publishing, 2019.
3. 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*, 2020. `arXiv:2004.05106`.
4. Harris Brakmić. *Bitcoin Script*, pages 201–224. Apress, Berkeley, CA, 2019.
5. 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.
6. Silvia Crafa, Cosimo Laneve, and Giovanni Sartor. Pacta sunt servanda: legal contracts in Stipula. Technical report, arXiv:2110.11069, 10 2021.
7. Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, USA, 2017.
8. 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.
9. 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.
10. Steve Klabnik and Carol Nichols. *The RUST programming language*. No Starch Press, 2019.
11. 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.
12. 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.

A The theory of liquidity: technical material

▸ **Lemma 7.** *The following properties hold true:*

*(**Weakening**) Let $\Xi \vdash S : \Xi'$ and Ξ_w be such that $\Xi_w|_{dom(\Xi)} = \Xi$. Then there exists Ξ'_w such that $\Xi_w \vdash S : \Xi'_w$ and $\Xi'_w|_{dom}(\Xi') = \Xi'$. Similarly, if $\Xi \vdash v : \mathbb{e}$ and $\Xi_w|_{dom(\Xi)} = \Xi$, then $\Xi_w \vdash v : \mathbb{e}$.*

*(**Substitution**) Let σ be a substitution mapping liquidity names to liquidity expressions. If $\Xi \vdash S : \Xi'$ then $\sigma(\Xi) \vdash S : \sigma(\Xi')$; if $\Xi \vdash v : \mathbb{e}$ then $\sigma(\Xi) \vdash v : \sigma(\mathbb{e})$.*

*(**Monotonicity**) Let Ξ, Ξ'' be ground environments such that $\Xi \leqslant \Xi''$. If $\Xi \vdash S : \Xi'$ then there is Ξ''' such that $\Xi'' \vdash S : \Xi'''$ and $\Xi' \leqslant \Xi'''$. Similarly for v: if $\Xi \vdash v : \mathbb{e}$ then $\Xi'' \vdash v : \mathbb{e}'$ and $[\![\mathbb{e}]\!] \leqslant [\![\mathbb{e}']\!]$.*

Proof. The proofs of Weakening and Substitution are standard and therefore omitted; we discuss the Monotonicity. The proof about expressions is omitted. As regards statements, the proof is by induction on the structure of S. The basic case is immediate. The inductive cases are $P\ S$ and $\texttt{if}\ (E)\ \{\ S_t\ \}\ \texttt{else}\ \{\ S_e\ \}\ S$, assuming the Monotonicity holds on S, S_t and S_e. We discuss $P\ S$ when P is (i) $E \to \texttt{x}$ and (ii) $c \times \texttt{h} \multimap \texttt{h}'$.

In case (i), we must prove monotony for $\Xi \vdash E \to \texttt{x}\ \ S : \Xi'$ and $\Xi \leqslant \Xi''$. By [L-SEQ], we have $\Xi \vdash E \to \texttt{x} : \Xi_1$ and $\Xi_1 \vdash S : \Xi'$. By [L-UPDATE] applied to $\Xi \vdash E \to \texttt{x} : \Xi_1$ we derive $\Xi_1 = \Xi$. Therefore Monotonicity follows by inductive hypotheses on S.

In case (ii), monotony must be proved for $\Xi \vdash c \times \texttt{h} \to \texttt{h}'\ \ S : \Xi'$ and $\Xi \leqslant \Xi''$. By [L-SEQ], we have $\Xi \vdash c \times \texttt{h} \to \texttt{h}' : \Xi_1$ and $\Xi_1 \vdash S : \Xi'$. By [L-EXPAUPD], $\Xi_1 = \Xi[\texttt{h}' \mapsto \mathbb{e}']$, where $\mathbb{e}' = (\mathbb{e} \sqcap \Xi(\texttt{h})) \sqcup \Xi(\texttt{h}')$ and $\Xi \vdash c : \mathbb{e}$. Taking an environment Ξ'' such that $\Xi \leqslant \Xi''$ and applying [L-EXPAUPD], we obtain $\Xi'' \vdash c \times \texttt{h} \to \texttt{h}' : \Xi'_1$ such that $\Xi'_1 = \Xi''[\texttt{h}' \mapsto \mathbb{e}'']$, where $\mathbb{e}'' = (\mathbb{e} \sqcap \Xi''(\texttt{h})) \sqcup \Xi''(\texttt{h}')$ and $\Xi'' \vdash c : \mathbb{e}$ (by [L-ZERO] and [L-VALUE] we obtain the same $\mathbb{e}$ as before). Therefore, by monotony of $\sqcap$ and $\sqcup$, $\Xi_1 \leqslant \Xi'_1$. Monotonicity follows again by inductive hypotheses on S. ◂

Theorem 3 (Correctness of liquidity labels)**.** Let $\texttt{C}$ be a *Stipula* contract with assets $\overline{\texttt{h}}$ and liquidity function types $\mathcal{L}$. If

$$\texttt{C}(\texttt{Q}\,,\,\ell\,,\,\_) \xrightarrow{A.\texttt{f}(\overline{u})[\overline{v}]} \texttt{C}(\texttt{Q}\,,\,\ell'\,,\,S \Rightarrow \texttt{Q}') \Longrightarrow \texttt{C}(\texttt{Q}'\,,\,\ell''\,,\,\_ \Rightarrow @\texttt{Q}'')$$

where the transition in $\Longrightarrow$ are not instances of rule [FUNCTION] and $\texttt{Q}\ \texttt{A.f}\ \texttt{Q}' : \Xi \to \Xi'$ in $\mathcal{L}$ then:

1. $\mathbb{E}(\ell)|_{\overline{\texttt{h}}} = \mathbb{E}(\ell')|_{\overline{\texttt{h}}}$;
2. $\mathbb{E}(\ell') \vdash S : \Xi''$;
3. there is a ground substitution σ such that $\mathbb{E}(\ell')|_{dom(\Xi)} \leqslant [\![\sigma(\Xi)]\!]$ and $[\![\Xi''|_{dom(\Xi)}]\!] \leqslant [\![\sigma(\Xi')]\!]$.
4. $\mathbb{E}(\ell'')|_{dom(\Xi)} \leqslant [\![\Xi'']\!]$.

Proof. Item 1 is an immediate consequence of definitions of $\xrightarrow{A.\texttt{f}(\overline{u})[\overline{v}]}$ and of $\mathbb{E}(\cdot)$.

As regards the statement 2, by $\texttt{Q}\ \texttt{A.f}\ \texttt{Q}' : \Xi \to \Xi' \in \overline{\mathcal{L}}$ and [L-FUNCTION] and assuming $\overline{\texttt{k}}$ be the list of asset parameters of $\texttt{A.f}$, we have that $\Xi = \Xi_0[\overline{\texttt{k}} \mapsto \overline{\mathbb{1}}]$ and $\Xi_0[\overline{\texttt{k}} \mapsto \overline{\xi'}] \vdash S : \Xi''$ ($\overline{\xi'}$ are a tuple of fresh liquidity names). We first notice that $dom(\Xi) = dom(\Xi_0) \cup \overline{\texttt{k}}$ and that $dom(\ell')$ may have asset names that do not occur in $dom(\Xi_0[\overline{\texttt{k}} \mapsto \overline{\xi'}])$. These names are

the formal parameters of previous invocations, which have not been garbage-collected by the semantics. However, by definition of $\mathbb{E}(\ell)$: $dom(\Xi_0[\overline{\mathtt{k}} \mapsto \overline{\xi'}]) \subseteq dom(\mathbb{E}(\ell'))$. Additionally, by definition of $\Xi_0[\overline{\mathtt{k}} \mapsto \overline{\xi'}]$, there exists a ground substitution σ such that

$$\mathbb{E}(\ell)|_{dom(\Xi)} \;=\; \sigma(\Xi[\overline{\mathtt{k}} \mapsto \overline{\xi'}]) \;.$$

Therefore, by the Substitution lemma, we have

$$\mathbb{E}(\ell)|_{dom(\Xi)} \vdash S : \Xi''' \tag{1}$$

for some Ξ'''. We obtain 2. by (1) and the Weakening Lemma.

We demonstrate 3. Observe that $\Xi = \Xi[\overline{\mathtt{k}} \mapsto \overline{\xi'}]\{{}^{\overline{\mathbb{1}}}/_{\overline{\xi'}}\}$. Therefore, using the foregoing substitution σ we have

$$\begin{array}{rcl} \mathbb{E}(\ell)|_{dom(\Xi)} & = & [\![\sigma(\Xi[\overline{\mathtt{k}} \mapsto \overline{\xi'}])]\!] \\ & \leqslant & [\![\sigma(\Xi)]\!] \end{array}$$

(notice the "$\leqslant$". It is so because Ξ does not contain $\overline{\xi'}$ being replaced by $\mathbb{1}$, while σ might map these names to $\mathbb{0}$). Next, by Monotonicity we obtain $\Xi''' \leqslant [\![\sigma(\Xi')]\!]$. The statement 3 follows by noticing that $\Xi''|_{dom(\Xi)} = \Xi'''$ and using Weakening.

To demonstrate 4, we proceed by induction on the length of the computation. The basic case is immediate. Assuming the property holds for computations of length n, we demonstrate the case of computations of length $n+1$ by reasoning on the first transition and applying inductive hypotheses to the continuation. We discuss the case when the first transition is an instance of [ASSET-UPDATE] with $c \neq 1$. Let

$$\begin{array}{rcl} \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,c \times \mathtt{h} \multimap \mathtt{h}'\;S \Rrightarrow @\mathtt{Q}') & \longrightarrow & \mathtt{C}(\mathtt{Q}\,,\,\ell[\mathtt{h} \mapsto v', \mathtt{h}' \mapsto v'']\,,\,S \Rrightarrow @\mathtt{Q}') \\ & \Longrightarrow & \mathtt{C}(\mathtt{Q}\,,\,\ell'\,,\,\_ \Rrightarrow @\mathtt{Q}') \end{array}$$

where $[\![c \times \mathtt{h}]\!]_\ell = v$ and $v' = [\![\mathtt{h} - v]\!]_\ell$ and $v'' = [\![\mathtt{h}' + v]\!]_\ell$. By $\mathbb{E}(\ell) \vdash c \times \mathtt{h} \multimap \mathtt{h}'\;S : \Xi''$ and by [L-SEQ] and [L-UPDATE], we obtain

$$\mathbb{E}(\ell) \vdash c \times \mathtt{h} \multimap \mathtt{h}' : \mathbb{E}(\ell)[\mathtt{h}' \mapsto \mathtt{e}] \qquad (\mathtt{e} = \mathbb{E}(\ell)(\mathtt{h}) \sqcup \mathbb{E}(\ell)(\mathtt{h}')) \tag{2}$$

$$\mathbb{E}(\ell)[\mathtt{h}' \mapsto \mathtt{e}] \vdash S : \Xi'' \tag{3}$$

By definition of v', v'' and $\mathtt{e}$, we observe that $\mathbb{E}(\ell[\mathtt{h} \mapsto v', \mathtt{h}' \mapsto v'']) \leqslant \mathbb{E}(\ell)[\mathtt{h}' \mapsto \mathtt{e}]$. Next, if $\mathbb{E}(\ell[\mathtt{h} \mapsto v', \mathtt{h}' \mapsto v'']) \vdash S : \Xi'$ then, by Lemma 7(Monotony), $\Xi' \leqslant \Xi''$. We conclude by inductive hypothesis that give $\mathbb{E}(\ell') \leqslant \Xi'$. ◂

Theorem 6 (Correctness of an abstract computation) Let $\mathtt{C}$ be a *Stipula* contract with liquidity function types $\mathcal{L}$ and $\mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1} : \Xi_i \to \Xi'_i \in \mathcal{L}$ for every $i \in 1..n$. If

$$\Big(\mathtt{C}(\mathtt{Q}_i\,,\,\ell_i\,,\,\_) \xrightarrow{A_i:\mathtt{f}_i(\overline{u_i})[\overline{v_i}]} \mathtt{C}(\mathtt{Q}_i\,,\,\ell'_i\,,\,S_i \Rrightarrow @\mathtt{Q}_{i+1}) \Longrightarrow \mathtt{C}(\mathtt{Q}_{i+1}\,,\,\ell_{i+1}\,,\,\_)\Big)^{i \in 1..n}$$

where

- $\ell'_i = \ell_i[\overline{\mathtt{y}_i} \mapsto \overline{u_i}, \overline{\mathtt{k}_i} \mapsto \overline{v_i}])$ with $\overline{\mathtt{y}_i}$ and $\overline{\mathtt{k}_i}$ being the formal parameters of the function $\mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1}$;
- the transitions in $\Longrightarrow$ are not instances of rule [FUNCTION];
- and the abstract computation $\varphi = \{\ \mathtt{Q}_i\ \mathtt{A}_i.\mathtt{f}_i\ \mathtt{Q}_{i+1}\ \}^{i \in 1..n}$ has 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 $\mathbb{E}(\ell_{n+1})|_{\overline{\mathtt{h}}} \leqslant [\![\sigma(\Xi')]\!]$.

Proof. Let $\varphi = \{\mathtt{Q}_i\, A_i.\mathtt{f}_i\, \mathtt{Q}_{i+1}\}^{i\in 1..n}$ and, for every i, $\mathtt{Q}_i\, A_i.\mathtt{f}_i\, \mathtt{Q}_{i+1} : \Xi_i \to \Xi'_i \in \mathcal{L}$. Let also $\mathtt{L}_\varphi = \Xi_1^{(b)}|_{\overline{\mathtt{h}}} \to \Xi_n^{(e)}|_{\overline{\mathtt{h}}}$ such that

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

For every $i \in 1..n$, by definition of $\Xi_i^{(b)} \to \Xi_i^{(e)}$ and the Substitution Lemma, we have $\Xi_i^{(b)} \vdash S_i : \Xi_i^{(e)}$. Let $\overline{\mathtt{h}'} = dom(\Xi_i^{(b)})$; if there is a ground substitution σ_i such that $\mathbb{E}(\ell'_i)|_{\overline{\mathtt{h}'}} \leqslant [\![\sigma_i(\Xi_i^{(b)})]\!]$ then, by Theorem 3(2), $\mathbb{E}(\ell_{i+1})|_{\overline{\mathtt{h}'}} \leqslant [\![\sigma_i(\Xi_i^{(e)})]\!]$.

Below we demonstrate that, for every $i \in 1..n$, σ_i do exist and are all equal.

Let $\overline{\mathtt{h}_1} = dom(\Xi_1^{(b)})$. By definition of $\Xi_1^{(b)}$, there is σ such that

$$\mathbb{E}(\ell_1)|_{\overline{\mathtt{h}_1}} \leqslant [\![\sigma(\Xi_1^{(b)})]\!] \tag{4}$$

$$\mathbb{E}(\ell_2)|_{\overline{\mathtt{h}_1}} \leqslant [\![\sigma(\Xi_1^{(e)})]\!] \qquad \text{(by Theorem 3)} \tag{5}$$

$$\mathbb{E}(\ell_1)|_{\overline{\mathtt{h}}} \leqslant [\![\sigma(\Xi_1^{(b)})]\!]|_{\overline{\mathtt{h}}} \qquad \text{(by (4) and } \overline{\mathtt{h}} \subseteq \overline{\mathtt{h}_1}) \tag{6}$$

$$\mathbb{E}(\ell_2)|_{\overline{\mathtt{h}}} \leqslant [\![\sigma(\Xi_1^{(e)})]\!]|_{\overline{\mathtt{h}}} \qquad \text{(by (5) and } \overline{\mathtt{h}} \subseteq \overline{\mathtt{h}_1}) \tag{7}$$

By definition, $\ell'_2 = \ell_2[\overline{\mathtt{y}_2} \mapsto \overline{u_2}, \overline{\mathtt{k}_2} \mapsto \overline{v_2}]$. Let $dom(\Xi_2^{(b)}) = \overline{\mathtt{h}_2} = \overline{\mathtt{h}} \cup \overline{\mathtt{k}_2}$. Then $\mathbb{E}(\ell'_2)|_{\overline{\mathtt{h}_2}} \leqslant \mathbb{E}(\ell_2)|_{\overline{\mathtt{h}}}[\overline{\mathtt{k}_2} \mapsto \overline{\mathbb{1}}] \leqslant [\![\sigma(\Xi_1^{(e)})]\!]|_{\overline{\mathtt{h}}}[\overline{\mathtt{k}_2} \mapsto \overline{\mathbb{1}}]$ (because of (7)). We conclude by observing that $\Xi_2^{(b)} = \Xi_1^{(e)}|_{\overline{\mathtt{h}}}[\overline{\mathtt{k}_2} \mapsto \overline{\mathbb{1}}]$ and that $\sigma(\Xi_2^{(b)}) = \sigma(\Xi_1^{(e)}|_{\overline{\mathtt{h}}}[\overline{\mathtt{k}_2} \mapsto \overline{\mathbb{1}}]) = \sigma(\Xi_1^{(e)})|_{\overline{\mathtt{h}}}[\overline{\mathtt{k}_2} \mapsto \overline{\mathbb{1}}]$.

The theorem follows by repeating the arguments on every function in the abstract computation. ◂