Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria

Corso di Laurea Triennale in Informatica

Analisi della Liquidità nei contratti legali in Stipula

Relatore:
Chiar.mo Prof.
Cosimo Laneve

Presentata da:
Matteo Panicciari

Sessione Marzo 2026
Anno Accademico 2025/2026

Abstract

I contratti legali definiscono clausole che regolano le interazioni tra più parti e disciplinano la gestione di risorse patrimoniali. In questo contesto, la proprietà di liquidità garantisce che le risorse gestite da un contratto non rimangano permanentemente bloccate e che il loro valore possa tornare nella disponibilità di una delle parti coinvolte. In questa tesi studiamo tale proprietà nei contratti legali scritti in `Stipula`. Ne introduciamo la sintassi e la semantica operazionale, quindi formalizziamo le proprietà di *k-Separate Liquidity* e *Liquidity*, insieme a un sistema di tipi capace di descrivere simbolicamente il comportamento del contratto in esecuzione rispetto alle risorse gestite.

Su questa base presentiamo due algoritmi per la verifica automatica della liquidità e realizziamo, in Python, un analizzatore che li implementa. Lo strumento consente di stabilire se un contratto soddisfa tali proprietà e di individuare eventuali situazioni di blocco permanente del valore.

Indice

Elenchi

Elenco delle tabelle

Elenco degli algoritmi

Elenco delle definizioni

Elenco dei contratti

Capitolo 1

Introduzione

1.1 Contratti Legali

Un contratto legale è un accordo tra due o più parti che, una volta raggiunto il consenso sui suoi termini (il cosiddetto *meeting of the minds* [2]), dà origine a un rapporto giuridicamente vincolante. Stipulando il contratto, le parti assumono obblighi reciproci, acquisiscono diritti e possono modificare le proprie posizioni giuridiche, ad esempio trasferendo beni, concedendo l'uso di una risorsa o stabilendo condizioni economiche tra loro. Tali rapporti si sviluppano nel tempo e possono produrre effetti patrimoniali e giuridici sulle parti coinvolte.

Generalmente i contratti vengono espressi attraverso il linguaggio naturale. Questo aspetto è fondamentale: il linguaggio naturale ha il vantaggio di essere molto espressivo e vicino a chi effettivamente redige i contratti, ma allo stesso tempo presenta lo svantaggio di essere soggetto ad ambiguità e possibili interpretazioni differenti. Tale ambiguità può generare incertezze nell'applicazione delle clausole contrattuali e rendere complessa l'analisi sistematica del contenuto di un contratto, soprattutto quando esso diventa particolarmente articolato o include numerose condizioni ed eccezioni.

Per questo motivo, negli ultimi anni è emersa l'esigenza di rappresentare i contratti legali in una forma più strutturata, che ne consenta non solo la descrizione, ma anche l'analisi automatica. In questo contesto sono nati dei linguaggi di dominio specifico [1] progettati per la digitalizzazione dei contratti legali, tra cui *Stipula* [2]. L'utilizzo di un linguaggio di programmazione per la stesura di un contratto è reso possibile dal principio di *libertà di forma* riconosciuto dai moderni ordinamenti giuridici, secondo

[1] Identifica un linguaggio di programmazione dedicato a un dominio ristretto di problemi

il quale le parti sono libere di scegliere il mezzo con cui formalizzare il proprio accordo [4, 14].

I linguaggi di programmazione hanno il vantaggio di offrire al contratto una definizione formale e non ambigua, permettendo di rappresentarne il comportamento in modo preciso. In questa prospettiva, `Stipula` si propone di fare da ponte tra l'espressività richiesta in ambito legale, il rigore garantito da una sintassi formale e la necessità di mantenere il linguaggio comprensibile a utenti non esperti di programmazione.

Un ulteriore vantaggio della rappresentazione dei contratti come programmi è la possibilità di sottoporli ad analisi automatiche. Grazie alla loro definizione formale, è infatti possibile sviluppare strumenti che verificano proprietà del contratto in modo deterministico. Un esempio è l'analisi della raggiungibilità delle clausole: una clausola è detta raggiungibile quando esiste una serie di azioni che, a partire dallo stato iniziale del contratto, conduce alla sua esecuzione. È quindi stato possibile sviluppare strumenti [4, 5] in grado di verificare automaticamente questa proprietà nei contratti scritti in `Stipula`.

1.2 Liquidità nei Contratti Legali

Nel contesto dei contratti legali rappresentati come programmi emerge però una nuova problematica: eventuali *errori nella definizione del contratto* non sono più ambiguità interpretative, ma diventano veri e propri bug del programma che possono compromettere la gestione dei fondi.

Numerosi errori nella scrittura di contratti hanno infatti mostrato come incongruenze tra il comportamento atteso e quello effettivo del codice possano causare la perdita o il blocco permanente di ingenti quantità di risorse digitali. Un esempio significativo è rappresentato da un incidente avvenuto su Ethereum, in cui 160 milioni di dollari sono rimasti bloccati all'interno di un contratto a causa di un bug in una libreria [1,9].

Questo aspetto motiva lo studio della proprietà di *liquidità*, che assume un ruolo centrale nell'analisi della correttezza dei contratti.

Intuitivamente, un contratto è *liquido* se tutte le risorse che esso gestisce possono, prima o poi, tornare in possesso di una delle parti coinvolte nella sua esecuzione. Al contrario, in un contratto non liquido può accadere che alcune risorse rimangano permanentemente intrappolate nel sistema, senza che alcun partecipante abbia la possibilità e il potere di recuperarle [7].

L'analisi della liquidità può essere vista come una particolare forma di proprietà di *liveness* nei sistemi concorrenti: non riguarda soltanto ciò che può accadere durante l'esecuzione del contratto, ma garantisce che certe situazioni desiderabili — come il recupero delle risorse — possano effettivamente verificarsi lungo l'evoluzione del sistema. Stabilire formalmente questa proprietà richiede quindi di modellare il comportamento del contratto e le possibili interazioni tra le parti che lo eseguono [7].

1.3 Il linguaggio Stipula

`Stipula` è un linguaggio di programmazione progettato per rappresentare contratti legali come programmi eseguibili. L'idea alla base del linguaggio è quella di modellare un contratto come un sistema per regolare le interazioni tra più soggetti e la gestione delle risorse condivise. In questo modo è possibile descrivere in maniera precisa quali attori partecipano al contratto, quali risorse vengono gestite e quali operazioni possono essere eseguite durante la sua esecuzione [2].

Dal punto di vista concettuale, il comportamento di un contratto può essere interpretato come quello di una macchina a stati finiti. Una macchina a stati finiti è un modello matematico costituito da un insieme finito di stati e da un insieme di transizioni che descrivono come il sistema può transire da uno stato all'altro in seguito a determinate azioni. Questo modello si adatta naturalmente alla rappresentazione dei contratti, poiché consente di descrivere le diverse fasi dell'accordo e le operazioni consentite in ciascuna di esse.

In `Stipula`, gli stati del contratto corrispondono agli stati della macchina, mentre le operazioni che possono essere eseguite dalle parti rappresentano le transizioni tra tali stati. L'evoluzione del contratto nel tempo è quindi descritta attraverso una sequenza di stati e di azioni che ne determinano il passaggio da una fase all'altra. Questa struttura rende espliciti i possibili comportamenti del contratto e consente di studiarne staticamente le possibili esecuzioni e le relative proprietà.

1.4 Struttura dell'elaborato

In questa tesi vengono descritte l'analisi e la definizione di algoritmi per l'individuazione di *contratti che non garantiscono liquidità*. Il fulcro dell'elaborato consiste nell'implementazione di un analizzatore basato su tali algoritmi.

Nel Capitolo 2 viene descritto il linguaggio `Stipula` e le sue caratteristiche. La comprensione della struttura e del significato di un contratto `Stipula` è un requisito

fondamentale per comprendere le scelte effettuate nell'implementazione dell'analizzatore.

Nel Capitolo 3 viene presentata la teoria della liquidità. Questo capitolo definisce il concetto di liquidità, soffermandosi sul formalizzare la presenza della proprietà nei contratti `Stipula`. Vengono riportati alcuni contratti di esempio allo scopo di spiegarne il comportamento e di esaminarne casi limite che rendono l'analisi più complicata. All'interno del capitolo vengono infine descritti gli algoritmi per l'individuazione di contratti liquidi, fondamentali nel capitolo successivo.

Nel Capitolo 4 viene presentata l'implementazione dell'analizzatore. Vengono riportate le tecnologie utilizzate, la struttura del progetto e le scelte effettuate. Segue poi un approfondimento sulle componenti dell'analizzatore con un focus sulle classi create per modellare i dati necessari. Infine vengono descritte le modalità d'uso dello strumento e l'output atteso dall'analizzatore.

Nel Capitolo 5 vengono riassunti i punti più importanti di tutta la tesi, presentando le conclusioni di questo lavoro.

Capitolo 2

Stipula

`Stipula` è un linguaggio di programmazione *domain-specific* progettato per scrivere ed eseguire contratti. L'obiettivo del linguaggio è fornire una rappresentazione formale dei contratti sufficientemente espressiva da modellare gli effetti giuridici tra le parti coinvolte e, allo stesso tempo, abbastanza strutturata da permettere l'analisi automatica delle loro proprietà [2].

Un contratto in `Stipula` descrive gli attori coinvolti nel contratto, detti *parti* o *partecipanti*, le risorse gestite dal contratto, dette *assets*, le variabili interne utilizzate per le informazioni sullo stato del contratto, dette *fields*, e le operazioni che possono essere effettuate durante la sua esecuzione, definite attraverso *funzioni* o *eventi*. Gli asset rappresentano risorse trasferibili, ad esempio valute digitale o token, mentre i campi memorizzano dati che influenzano il comportamento del contratto ma non rappresentano risorse economiche.

In `Stipula`, ogni *funzione* rappresenta una possibile transizione tra stati del contratto. Una funzione può essere invocata dalla parte autorizzata solo quando il contratto si trova nello stato specificato nella sua definizione. La sua esecuzione può aggiornare i campi del contratto, trasferire valori tra asset diversi e consente alle parti di ritirare i valori rimuovendoli dal contratto. Al termine dell'esecuzione della funzione, il contratto transita in un nuovo stato, modificando così l'insieme delle operazioni disponibili. Inoltre, il linguaggio consente di definire eventi temporizzati che vengono eseguiti automaticamente al verificarsi di una determinata condizione temporale, permettendo di modellare obblighi e scadenze tipiche dei contratti legali. Anche gli eventi, come le funzioni, fungono quindi da transizioni tra gli stati del contratto.

2.1 Sintassi di Stipula

Nella definizione della sintassi di `Stipula`, utilizzeremo diversi insiemi di nomi: *nomi di contratti*, indicati con C, C', ...; *nomi delle parti*, indicati con A, A', ...; *nomi di funzioni*, indicati con f, g, ...; *nomi degli asset*, indicati con h, k, ..., usati sia per gli asset globali che per i parametri asset; *nomi di fields*, indicati con x, y, ..., usati sia per i campi del contratto che per i parametri non-asset; *nomi degli stati*, indicati con $\mathtt{Q}$, $\mathtt{Q}'$, ...; i nomi degli asset, dei campi e dei parametri in generale verranno indicati con X. Per semplificare la sintassi utilizzeremo la notazione $\bar{x}$ per indicare una sequenza, eventualmente vuota, di elementi x.

Un contratto `Stipula` è composto inizialmente da alcune sezioni principali: la dichiarazione del nome del contratto, la definizione degli *asset* e dei *fields* utilizzati, e una clausola di `agreement`. Quest'ultima rappresenta il momento in cui le parti raggiungono l'accordo sui termini del contratto e ne attivano l'esecuzione. A partire da questo punto, il comportamento del contratto è modellato tramite una serie di transizioni tra stati, definite mediante *funzioni* o *eventi*. Il codice di un contratto `Stipula` avrà quindi la seguente forma:

$$\mathtt{stipula}\ C\ \{\ \mathtt{assets}\ \bar{h}\quad \mathtt{fields}\ \bar{x}\quad \mathtt{agreement}(\bar{A})\{\ \overline{\bar{A} : \bar{x}}\ \} \Rightarrow \mathtt{@Q}\quad \bar{F}\ \} \tag{2.1}$$

dove $\bar{F}$ indica una sequenza di dichiarazioni di *funzioni*, come definito nella *Sintassi di Stipula* descritta nella Tabella 2.1 [2, 4, 7].

Functions	$F ::=$	$\mathtt{@Q}\ A : \mathtt{f}(\bar{y})[\bar{k}]G\{\ S\ \bar{W}\ \} \Rightarrow \mathtt{@Q}'$
Events	$W ::=$	$t \gg \mathtt{@Q}\ \{\ S\ \} \Rightarrow \mathtt{@Q}'$
Guards	$G ::=$	$(E) \mid \_$
Statements	$S ::=$	$\_ \mid P\ S \mid \mathtt{if}(E)\{\ S\ \}\mathtt{else}\{\ S\ \}\ S$
Prefixes	$P ::=$	$E \to x \mid E \to A \mid h \multimap h'$
		$h \multimap A \mid E \multimap h, h' \mid E \multimap h, A$
Expressions	$E ::=$	$v \mid X \mid E\ \mathtt{op}\ E \mid \mathtt{uop}\ E$
Values	$v ::=$	$n \mid s \mid \mathtt{true} \mid \mathtt{false}$
Time Expr.	$t ::=$	$\mathtt{now} \mid \mathtt{t} + x \mid \mathtt{t} + n \mid \mathtt{t} + s$

Tabella 2.1: Sintassi di `Stipula`

$n \in \mathbb{R}$, s è una stringa.

2.1.1 Dichiarazioni iniziali

Come descritto nella regola 2.1, il contratto prevede un'iniziale dichiarazione degli asset e delle variabili utilizzati nel contratto. Viene quindi definita una lista - non

vuota - di identificatori per gli asset e una lista - non vuota - di identificatori per i campi del contratto.

Quindi, è prevista la fase di accordo tra le parti attraverso parola chiave `agreement` in cui vengono definiti gli identificatori di coloro che sono coinvolti nel contratto. Al suo interno troviamo poi l'assegnamento di ogni campo del contratto a una o più parti che possono farne uso. Viene poi definito lo *stato iniziale del contratto*, da cui può avere inizio l'esecuzione del contratto.

La scrittura del contratto si chiude poi con la definizione delle funzioni.

2.1.2 Funzioni, Eventi

La lista di *funzioni* definisce le transizioni possibili tra gli *stati del contratto*. Un'altra possibilità per definire queste transizioni è la definizione di *eventi*, dichiarati e descritti all'interno del corpo di una funzione. La dichiarazione degli stati avviene nel momento del loro utilizzo come *stato iniziale* o *stato finale* di una funzione o di un evento.

Funzione Una funzione F appartiene ad un partecipante A che è l'unica entità che può invocare l'esecuzione della funzione. L'invocazione può avvenire solo nel caso in cui il contratto si trovi nello stato iniziale definito nella dichiarazione della funzione (`Q`). La funzione accetta due liste di parametri, accessibili solo all'interno del suo corpo: parametri che rappresentano *campi* ($\bar{y}$) e parametri che rappresentano *asset* ($\bar{k}$). Prima del corpo della funzione può inoltre essere definita una *precondizione*, attraverso un'espressione booleana, chiamata *guardia*. Nel caso in cui, nel momento dell'invocazione della funzione, la guardia dovesse avere valore `false`, allora il corpo della funzione non verrà eseguito. Nel corpo della funzione è possibile definire una lista di *istruzioni* (S) e una lista di *eventi* (W). Le istruzioni possono essere strutture condizionali (if-else) oppure *prefissi* (P). Al termine del corpo della funzione, viene indicato lo stato (`Q'`) del contratto in cui transire in caso di esecuzione della funzione [2].

Evento Un evento W, in modo simile alle funzioni, definisce una transizione tra uno stato iniziale (`Q`) e uno stato finale (`Q'`) la cui invocazione esegue le istruzioni (S) definite all'interno del suo corpo. L'evento non accetta parametri o guardie e non appartiene a nessun partecipante. L'invocazione è schedulata e viene innescata quando l'orologio globale corrisponde al valore dell'espressione temporale (t). Quan-

do innescato, il corpo dell'evento verrà eseguito se il contratto si trova nello stato `Q` [2].

2.1.3 Prefissi

I prefissi P definiscono le operazioni di *assegnamento* ($\rightarrow$) o *movimento* ($\multimap$) degli asset e dei campi del contratto.

Assegnamento L'operazione di assegnamento $E \rightarrow x$ aggiorna il campo o il parametro (x) con il valore dell'espressione E. L'operazione di assegnamento $E \rightarrow A$ invia il valore contenuto nell'espressione (E) al partecipante A [2].

Movimento Le operazioni di movimento indicano spostamenti di valore patrimoniale da un *asset* (h) ad un altro asset (h') o ad un partecipante (A) del contratto. In particolare:

1. $E \multimap h, h'$ con $(E = c * h) \wedge (1 \geq c \geq 0)$, sposta parte del valore $(c \times h)$ contenuto nell'asset h, nell'asset h';
2. $E \multimap h, h'$ con $E \in X$, sposta il valore E dall'asset h all'asset h';
3. $E \multimap h, A$ con $(E = c * h) \wedge (1 \geq c \geq 0)$, ritira dal contratto parte del valore $(c \times h)$ contenuto nell'asset h, depositandolo nel patrimonio del partecipante A;
4. $E \multimap h, A$ con $E \in X$, ritira dal contratto il valore E dall'asset h depositandolo nel patrimonio del partecipante A;
5. $h \multimap h'$ è equivalente all'espressione (1) con $c = 1$, l'asset h risulterà vuoto;
6. $h \multimap A$ è equivalente all'espressione (3) con $c = 1$, l'asset h risulterà vuoto.

Le risorse contenute negli asset possono essere trasferite ma non distrutte. La semantica operazionale impedisce che un asset assuma valori negativi.

2.1.4 Espressioni

Le Expressions E includono valori costanti (numeri reali n, booleani, stringhe s); identificatori di asset, campi e parametri indicati con X; operazioni tra espressioni, che possono essere unarie (`uop` E, cioè: $!E \mid -E$) o binarie (E `op` E):

- binarie aritmetiche ($+ \mid - \mid * \mid /$);
- binarie booleane ($\&\& \mid \| \mid == \mid != \mid >= \mid <= \mid > \mid <$).

2.2 Semantica di Stipula

La semantica di Stipula è definita tramite una *relazione di transizione*. Tali transizioni sono definite su oggetti chiamati *configurazioni* [4, 7].

Definizione 1 (Configurazione)**.** *Definiamo una* configurazione*, indicata con* $\mathbb{C}$*,* $\mathbb{C}'$*, ..., come la tupla* `C(Q`, ℓ, Σ, Ψ`)` *dove:*

- `C` *è il* nome *del contratto;*
- `Q` *è uno* stato *del contratto;*
- ℓ*, chiamata* memoria*, è una mappatura dai* nomi *(partecipanti, campi, asset, parametri delle funzioni) ai rispettivi* valori*;*
- Σ *è il* residuo del corpo *di una funzione, i.e.* _ *(vuoto) oppure* $S \Rightarrow$ `@Q;`
- Ψ *è un* multiinsieme di eventi *in attesa, che sono già stati schedulati per un'esecuzione futura ma non sono ancora stati innescati.*

In particolare Ψ può essere _, quando non ci sono eventi in attesa, oppure $W_1 \mid ... \mid W_n$ dove $W_i ::= \mathbb{t}_i \gg_{\texttt{ev}_i} \texttt{Q}_i\{S_i\} \Rightarrow \texttt{Q}'$. In cui, per l'identificazione dell'elemento, useremo $\texttt{ev}_i$ che identifica la linea di codice in cui è stato dichiarato l'evento ed è introdotto dalla regola [FUNCTION].

Il tempo $\mathbb{t}_i$ è un tempo assoluto e si ottiene valutando, al momento della schedulazione, l'espressione temporale t_i di W_i dove `now` viene sostituito con il valore dell'orologio globale nel momento di generazione dell'evento.

Utilizzeremo una *funzione di valutazione* $[\![E]\!]_\ell$ per indicare il valore dell'espressione E nella memoria ℓ, tale che:

- $[\![v]\!]_\ell = v$ per numeri reali e valori di asset;
- $[\![\texttt{true}]\!]_\ell = 1$, $[\![\texttt{false}]\!]_\ell = 0$;
- $[\![X]\!]_\ell = \ell(X)$ per nomi di asset, campi, partecipanti o parametri;
- $[\![\texttt{uop}\ E]\!]_\ell = \texttt{uop}\ v$ con $[\![E]\!]_\ell = v$;
- $[\![E\ \texttt{op}\ E']\!]_\ell = v\ \texttt{op}\ v'$ con $[\![E]\!]_\ell = v$ e $[\![E']\!]_\ell = v'$.

Sia ℓ la *memoria* di una configurazione del contratto `C` e $\bar{h}$ un insieme di asset che fanno parte di `C`. Scriveremo $\ell(\bar{h}) > \bar{0}$ se e solo se $\exists k \in \bar{h}$ tale che $\ell(k) > 0$. Allo stesso modo scriveremo $\ell(\bar{h}) = \bar{0}$ se e solo se $\forall k \in \bar{h} \Rightarrow \ell(k) = 0$ [7].

[Function]

$$\frac{\texttt{@Q A} : \texttt{f}(\bar{y})[\bar{k}]G\{S\bar{W}\} \Rightarrow \texttt{@Q}' \in \texttt{C} \qquad \ell(G) = 1 \qquad \Psi, \mathbb{t} \nrightarrow \\ \ell(\texttt{A}) = A \qquad \Psi \mid \bar{W}\{\mathbb{t}/\texttt{now}\} = \Psi' \qquad \ell' = \ell[\bar{y} \mapsto \bar{u},\, \bar{k} \mapsto \bar{v}]}{\texttt{C}(\texttt{Q},\ \ell,\ \_,\ \Psi),\ \mathbb{t} \xrightarrow{A:\texttt{f}(\bar{u})[\bar{v}]} \texttt{C}(\texttt{Q},\ \ell',\ S \Rightarrow \texttt{@Q}',\ \Psi'),\ \mathbb{t}}$$

[Event]

$$\frac{\Psi = \{\mathbb{t} \gg_{ev_i} \texttt{@Q}\{S\} \Rightarrow \texttt{@Q}'\} \mid \Psi'}{\texttt{C}(\texttt{Q},\ \ell,\ \_,\ \Psi),\ \mathbb{t} \xrightarrow{\texttt{ev}_i} \texttt{C}(\texttt{Q},\ \ell,\ S \Rightarrow \texttt{@Q}',\ \Psi'),\ \mathbb{t}}$$

[Tick]

$$\frac{\Psi, \mathbb{t} \nrightarrow}{\texttt{C}(\texttt{Q},\ \ell,\ \_,\ \Psi),\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q},\ \ell,\ \_,\ \Psi),\ \mathbb{t} + 1}$$

[State-Change]

$$\texttt{C}(\texttt{Q},\ \ell,\ \_ \Rightarrow \texttt{@Q}',\ \Psi),\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q}',\ \ell,\ \_,\ \Psi'),\ \mathbb{t}$$

[Field-Update]

$$\frac{[\![E]\!]_\ell = v}{\texttt{C}(\texttt{Q},\ \ell,\ E \to x\ \Sigma,\ \Psi),\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q},\ \ell[x \mapsto v],\ \Sigma,\ \Psi),\ \mathbb{t}}$$

[Asset-Send]

$$\frac{[\![c \times h]\!]_\ell = v \qquad \ell(\texttt{A}) = A \qquad [\![h - v]\!]_\ell = v'}{\texttt{C}(\texttt{Q},\ \ell,\ c \times h \multimap h, \texttt{A}\ \Sigma,\ \Psi),\ \mathbb{t} \xrightarrow{v \multimap A} \texttt{C}(\texttt{Q},\ \ell[h \mapsto v'],\ \Sigma,\ \Psi),\ \mathbb{t}}$$

[Value-Send]

$$\frac{[\![E]\!]_\ell = v \qquad \ell(\texttt{A}) = A}{\texttt{C}(\texttt{Q},\ \ell,\ E \to \texttt{A}\ \Sigma,\ \Psi),\ \mathbb{t} \xrightarrow{v \to A} \texttt{C}(\texttt{Q},\ \ell,\ \Sigma,\ \Psi),\ \mathbb{t}}$$

[Asset-Update]

$$\frac{[\![c \times h]\!]_\ell = v \qquad [\![h - v]\!]_\ell = v' \qquad [\![h' + v]\!]_\ell = v'' \qquad \ell' = \ell[h \mapsto v',\, h' \mapsto v'']}{\texttt{C}(\texttt{Q},\ \ell,\ c \times h \multimap h, h'\ \Sigma,\ \Psi),\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q},\ \ell',\ \Sigma,\ \Psi),\ \mathbb{t}}$$

[Cond-True]

$$\frac{[\![E]\!]_\ell \neq 0}{\texttt{C}(\texttt{Q},\ \ell,\ \textbf{if}(E)\{S\}\textbf{else}\{S'\}\ \Sigma,\ \Psi),\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q},\ \ell,\ S\ \Sigma,\ \Psi),\ \mathbb{t}}$$

[Cond-False]

$$\frac{[\![E]\!]_\ell = 0}{\texttt{C}(\texttt{Q},\ \ell,\ \textbf{if}(E)\{S\}\textbf{else}\{S'\}\ \Sigma,\ \Psi),\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q},\ \ell,\ S'\ \Sigma,\ \Psi),\ \mathbb{t}}$$

Tabella 2.2: Relazione di transizione di `Stipula`

Relazione di transizione La relazione di transizione è indicata da $\mathbb{C}, \mathbb{t} \xrightarrow{\mu} \mathbb{C}', \mathbb{t}'$ ed è definita nella Tabella 2.2 [7]. L'etichetta di transizione μ può essere $\_$, $A : \texttt{f}(\bar{u})[\bar{v}]$, $\texttt{ev}_i$, $v \to A$ oppure $v \multimap A$.

Prendiamo in considerazione la regola [FUNCTION]. L'etichetta μ indica che il partecipante A invoca la funzione di nome `f` con parametri attuali $\bar{u}$ e $\bar{v}$. La transizione avviene se

- il contratto si trova nello stato `Q`, che ammette l'invocazione di `f` da parte di `A`, con $\ell(\texttt{A}) = A$;
- la configurazione corrente ha codice residuo _;
- se il predicato $\Psi, \mathbb{t} \nrightarrow$ è vero.

Dato $\Psi = \{\mathbb{t}_i \gg_{ev_j} \texttt{@Q}_i\{S_i\} \Rightarrow \texttt{@Q}'\}^{i \in 1..k}$, il predicato $\Psi, \mathbb{t} \nrightarrow$ è definito come segue [4]:

$$\Psi, \mathbb{t} \nrightarrow = \begin{cases} \texttt{true} & \text{se } \forall i \in 1..k \ , \ \mathbb{t}_i \neq \mathbb{t} \\ \texttt{false} & \text{altrimenti} \end{cases}$$

Cioè $\Psi, \mathbb{t} \nrightarrow$ è vero se nessun evento in Ψ può essere innescato al tempo $\mathbb{t}$. Questo da priorità alle transizioni innescate da un'invocazione di evento rispetto a quelle causate da un'invocazione di funzione.

Commentiamo anche la regola [ASSET-SEND]; le altre regole sono simili. La regola [ASSET-SEND] calcola il valore v tramite $c \times h$, poi rimuove il valore v dall'asset h e assegna il valore v al partecipante A, controllando che $\ell(\texttt{A}) = A$.

Configurazione Iniziale La *configurazione iniziale* di un contratto `C` sarà quindi la tupla $\texttt{C}(\texttt{Q}_0, \ell, \_, \_)$ in cui $\texttt{Q}_0$ è lo stato iniziale del contratto, $\bar{h}$ è l'insieme di asset globali definiti e ℓ è una memoria in cui $\forall k \in \bar{h}.[\![k]\!]_\ell = \mathbb{0}$.

2.3 Computazioni Astratte

La semantica operazionale descritta definisce l'evoluzione di un contratto come una sequenza di transizioni tra configurazioni. Una *computazione concreta* è quindi una sequenza di configurazioni collegate dalla relazione di transizione.

Per le analisi fatte in seguito non è tuttavia necessario tenere traccia delle configurazioni nel loro totale. Informazioni quali: la memoria ℓ, i valori dei parametri delle funzioni o l'orologio globale del contratto non verranno prese in considerazione. È sufficiente considerare la sequenza delle invocazioni di funzione e gli stati del contratto attraversati durante l'esecuzione.

Introduciamo quindi il concetto di *computazione astratta*.

Definizione 2 (Computazione Astratta)**.** *Una* computazione astratta φ *di un contratto in* `Stipula` *è una sequenza finita* $\mathtt{Q}_1\ \mathtt{c}_1\ \mathtt{Q}_2\ \mathtt{;}\ \ldots\mathtt{;}\ \mathtt{Q}_n\ \mathtt{c}_n\ \mathtt{Q}_{n+1}$ *di funzioni* ($c_i = \mathtt{A}_i.f_i$) *o di eventi* ($c_i = \mathtt{ev}_i$) *del contratto. Usiamo la notazione* $\mathtt{Q} \overset{\varphi}{\rightsquigarrow} \mathtt{Q'}$ *per indicare lo stato iniziale e finale di* φ*.*

Indichiamo con $\{\ \mathtt{Q}_i\ \mathtt{c}_i\ \mathtt{Q}_{i+1}\ \}^{i\in 1..n}$ *la computazione astratta di:*

$$(\quad \mathcal{C}\,(\mathtt{Q}_i, \ell_i, \_, \Psi_i), \mathbb{t}_i \xRightarrow{A_i:f_i(\bar{u_i})[\bar{v_i}]\ \mid\ \mathtt{ev}_i} \mathcal{C}\,(\mathtt{Q}_{i+1}, \ell_{i+1}, \_, \Psi_{i+1}), \mathbb{t}_{i+1}\quad)^{i\in 1..n}$$

Una computazione astratta φ *è* κ*-canonica se le funzioni occorrono al più* κ *volte in* φ*.*

Le *computazioni astratte* saranno fondamentali nella nostra analisi e nei nostri algoritmi. Sarà quindi necessario creare un algoritmo per calcolare tutte le computazioni di un contratto `C` raggiungibili dallo stato iniziale $\mathtt{Q}_0$.

Algoritmo 2.1 Calcolo delle Computazioni Astratte

Poni $\mathcal{AC} \leftarrow \varnothing$.

Passo 1. Per ogni funzione f di `C` che ha stato iniziale $\mathtt{Q}_0$, crea φ tale che $\varphi = [f]$. Poni $\varphi_\Psi = ev(f)$. Inserisci quindi φ in $\mathcal{AC}$.

Passo 2. Poni $\mathcal{N} \leftarrow \varnothing$.

Passo 3. Per ogni computazione astratta $\varphi \in \mathcal{AC}$, considera tutte le funzioni f di `C` tali che lo stato iniziale di f coincida con lo stato finale di φ.
Se f compare in φ meno di κ volte: poni $\varphi' = \varphi + f$; poni $\varphi'_\Psi = \varphi_\Psi \cup ev(f)$; inserisci φ' in $\mathcal{N}$.

Passo 4. Per ogni computazione astratta $\varphi \in \mathcal{AC}$, considera tutti gli eventi e disponibili in φ_Ψ tali che lo stato iniziale di e coincida con lo stato finale di φ. Poni $\varphi' = \varphi + e$; poni $\varphi'_\Psi = \varphi_\Psi \setminus \{e\}$; inserisci φ' in $\mathcal{N}$.

Passo 5. Se $\mathcal{N} \setminus \mathcal{AC} = \varnothing$, allora **termina**: $\mathcal{AC}$ contiene tutte le computazioni astratte generabili dal contratto `C` partendo dallo stato $\mathtt{Q}_0$. Altrimenti aggiorna $\mathcal{AC} \leftarrow \mathcal{AC} \cup \mathcal{N}$ e ripeti il **Passo 2**.

L'Algoritmo 2.1 utilizza una strategia di *costruzione incrementale mediante punto fisso*[1]. Si parte da un insieme iniziale di computazioni astratte e lo si estende iterativamente generando nuove computazioni ottenute come estensione di quelle

[1]Una procedura iterativa in cui, a partire da una base iniziale, si applica ripetutamente una funzione di espansione monotona fino al raggiungimento di un punto fisso, ossia di uno stato stabile in cui nessun nuovo elemento può più essere aggiunto.

già note. A ogni iterazione vengono considerate tutte le computazioni astratte già costruite e si producono possibili estensioni compatibili con lo stato finale della computazione corrente. Le nuove computazioni generate vengono raccolte in un insieme temporaneo e aggiunte all'insieme principale solo al termine dell'iterazione. Il procedimento continua finché non è più possibile generare nuove computazioni, ovvero quando l'insieme risultante non cresce ulteriormente.

Dal punto di vista implementativo, ogni computazione astratta φ mantiene, oltre alla sequenza delle funzioni ed eventi già eseguiti, anche un attributo φ_Ψ che rappresenta l'insieme degli eventi attualmente schedulati. Questo insieme viene aggiornato dinamicamente durante l'estensione della computazione: quando viene aggiunta una funzione, gli eventi dichiarati all'interno della funzione vengono inseriti tra quelli disponibili; quando invece viene eseguito un evento, esso viene rimosso dall'insieme degli eventi disponibili. Si utilizza una funzione $ev(f)$ che restituisce l'insieme di eventi dichiarati all'interno della funzione f. L'utilizzo di un insieme separato $\mathcal{N}$ per raccogliere le computazioni generate in una iterazione permette di verificare facilmente se l'insieme delle computazioni è cresciuto e quindi se l'algoritmo deve proseguire oppure può terminare.

Capitolo 3

Liquidità

3.1 Liquidità in Stipula

Diamo ora la definizione di *liquidità* per contratti scritti in linguaggio **Stipula**. Il nostro obiettivo è verificare staticamente se un contratto garantisce sempre l'assenza di fondi bloccati perennemente al suo interno. Quindi, introdurremo dapprima una proprietà locale sugli asset, detta *k-Separate Liquidity*, e successivamente una proprietà più forte, detta *Liquidity*, necessaria nei casi in cui il valore patrimoniale possa essere trasferito tra asset distinti del contratto.

Definizione 3 (k-Separate Liquidity)**.** *Un contratto* $\mathbf{C}$ *scritto in* ***Stipula*** *con asset* $\bar{h}$ *e configurazione iniziale* $\mathbb{C}$ *ha la proprietà* k-Separate Liquidity *(*$k \in \bar{h}$*) se, per ogni* $\mathbb{C} \Rightarrow \mathbf{C}(\mathsf{Q}, \ell, \_, \Psi)$*, vale:*

$$\ell(\bar{h'}) = \bar{0} \quad con \quad \bar{h'} = dom(\ell) \setminus \bar{h} \tag{1.1}$$

$$se \;\; \ell(k) > 0 \;\; allora \;\; \exists\, \mathbf{C}(\mathsf{Q}, \ell, \_, \Psi) \Rightarrow \mathbf{C}(\mathsf{Q}', \ell', \_, \Psi') \;\; t.c. \;\; \ell'(k) = 0 \tag{1.2}$$

Diremo che il contratto $\mathbf{C}$ *ha la proprietà* Separate Liquidity *se,* $\forall k \in \bar{h}$*, vale la* k-Separate Liquidity*.*

Per analizzare come un contratto risponde a questa proprietà, indichiamo quali sono i valori che può assumere un asset attraverso un'astrazione: un asset è vuoto - notazione $\mathbb{0}$ - o non vuoto - notazione $\mathbb{1}$. I valori $\mathbb{0}$ e $\mathbb{1}$ sono chiamati *valori di liquidità* e utilizzeremo le *espressioni di liquidità* per descrivere lo stato di un asset [7].

Definizione 4 (Espressione di Liquidità)**.** *Un'*espressione di liquidità *è definita*

come segue:

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

dove $\xi, \xi', \ldots$ *indicano* nomi di liquidità *(simbolici).*

Valore delle espressioni di liquidità Le espressioni di liquidità sono **ordinate** in questo modo: $\mathbb{0} \leq \mathbb{e}$ e $\mathbb{1} \geq \mathbb{e}$. Le operazioni $\sqcap$ e $\sqcup$ ritornano rispettivamente il **minimo** e il **massimo** valore tra i due argomenti e sono **monotoniche** rispetto al $\leq$ (quindi se $\mathbb{e}_1 \leq \mathbb{e}'_1$ e $\mathbb{e}_2 \leq \mathbb{e}'_2$ allora $\mathbb{e}_1 \sqcup \mathbb{e}_2 \leq \mathbb{e}'_1 \sqcup \mathbb{e}'_2$ e $\mathbb{e}_1 \sqcap \mathbb{e}_2 \leq \mathbb{e}'_1 \sqcap \mathbb{e}'_2$). Due tuple sono ordinate da $\leq$ se sono ordinate elemento per elemento da $\leq$.

La definizione di *Separate Liquidity* risponde alla nostra necessità: verificare, per ogni asset, che se in una configurazione risulta non vuoto, esista una configurazione successiva in cui esso possa diventare vuoto.

Consideriamo il contratto `Ping_Pong` [7].

```
stipula Ping_Pong {
    assets hA, hM
    fields x
    agreement(Amy, Mary) { Mary : x } => @Q0

    @Q0 Mary: ping()[u]{
        hM -o Mary
        u -o hA
    } => @Q1

    @Q1 Amy: pong()[v]{
        hA -o Amy
        v -o hM
    } => @Q0
}
```

Contratto 3.1: Ping_Pong

Il contratto presenta un ciclo in cui `Mary` e `Amy` si scambiano valori patrimoniali. Invocando `ping`, `Mary` muove parte del suo patrimonio in `u`, raccoglie il valore depositato in `hM`, poi muove il valore spostato in `u` verso `hA`. A questo punto `Amy` può invocare `pong`, muovendo valori in modo simile a quanto fatto prima da `Mary`.

Lo scambio di beni tra `Amy` e `Mary` può andare avanti all'infinito e sappiamo che nessun valore inserito negli asset del contratto rimarrà bloccato.

È facile verificare che la *Separate Liquidity* viene rispettata: ad esempio, al termine di un `ping`, ci troveremo nella configurazione $\texttt{Ping\_Pong(Q1,}\ell\texttt{[hA} \mapsto \mathbb{1}, \texttt{hM} \mapsto \mathbb{0}\texttt{]}, \_, \_\texttt{)}$,

quindi con hA non vuoto. Trovandoci nello stato Q1, Amy potrà subito svuotarlo con un pong.

Valutiamo adesso una variante di Ping_Pong, il contratto Ping_Pong_Sink.

```
stipula Ping_Pong_Sink {
    assets hA,hM
    fields x

    agreement(Amy,Mary){
        Mary : x
    } => @Q0

    @Q0 Mary: ping()[u]{
        hM -o hA
        u -o hA
    } => @Q1

    @Q1 Amy: pong()[v]{
        hA -o hM
        v -o hM
    } => @Q0
}
```

Contratto 3.2: Ping_Pong_Sink

Questo contratto differisce da Ping_Pong perché, quando Mary invoca ping, il valore contenuto in hM non viene ritirato e inserito nel patrimonio di Mary, ma viene spostato in un altro asset.

Valutiamo adesso la *Separate Liquidity* di Ping_Pong_Sink. Abbiamo due asset (hA e hM) e due possibili configurazioni con codice residuo vuoto[1]:

1. $\texttt{Ping\_Pong\_Sink(Q0,}\ell\texttt{[hA} \mapsto \mathbb{0}, \texttt{hM} \mapsto \mathbb{1}\texttt{]}, \_, \_\texttt{)}$;
2. $\texttt{Ping\_Pong\_Sink(Q1,}\ell'\texttt{[hA} \mapsto \mathbb{1}, \texttt{hM} \mapsto \mathbb{0}\texttt{]}, \_, \_\texttt{)}$.

Per hA vediamo che solo nella configurazione (2) vale che $\ell'(\texttt{hA}) > \mathbb{0}$, ma esiste una possibile configurazione successiva (1) in cui $\ell(\texttt{hA}) = \mathbb{0}$. La hA-*Separate Liquidity* è quindi verificata. Allo stesso modo (con configurazioni invertite), sarà verificata la hM-*Separate Liquidity*.

Nonostante valga la *Separate Liquidity*, osserviamo che i valori inseriti nei due asset non potranno mai essere ritirati dalle due parti. In altre parole, una volta che un

[1] $\mathbb{C} = \texttt{C(Q, } \ell\texttt{, } \Sigma\texttt{, } \Psi\texttt{)}$ tale che $\Sigma = \_$

bene è inserito in uno degli asset del contratto, questo non potrà mai essere prelevato e rimarrà permanentemente all'interno del contratto. Questo comportamento è causato dallo spostamento di valore tra i due asset (`hA -o hM` e `hM -o hA`), che rende il contratto *Separate Liquid* in quanto, nonostante al termine di ogni configurazione, uno dei due asset risulti svuotato, il suo valore viene semplicemente trasferito nell'altro asset del contratto.

Questo esempio mostra che non è sempre sufficiente analizzare gli asset separatamente, occorre talvolta considerare insiemi di asset tra loro collegati da operazioni di movimento.

Cluster Dato un contratto C, definiamo una relazione tra asset globali dicendo che due asset appartengono allo stesso *cluster* se esiste una sequenza di operazioni di movimento che li collega. I cluster sono quindi le classi di equivalenza indotte da tale relazione.

In altre parole, due asset appartengono allo stesso cluster se il valore può circolare dall'uno all'altro, eventualmente attraverso altri asset del contratto. In questi casi non è sufficiente analizzare gli asset separatamente: per escludere la presenza di fondi permanentemente bloccati è necessario verificare che tutti gli asset appartenenti allo stesso cluster possano essere svuotati nella medesima configurazione.

In generale, sarebbe più accurato verificare la proprietà di $\bar{c}$-*Separate Liquidity*, dove $\bar{c}$ è un cluster di asset. Nel presente lavoro adottiamo tuttavia una semplificazione conservativa: consideriamo un unico cluster costituito da tutti gli asset del contratto e richiediamo quindi l'esistenza di una configurazione in cui essi risultino simultaneamente vuoti.

Definizione 5 (Liquidity)**.** *Un contratto* $\mathbf{C}$ *scritto in* ***Stipula*** *con asset* $\bar{h}$ *e configurazione iniziale* $\mathbb{C}$ *ha la proprietà* Liquidity *se, per ogni* $\mathbb{C} \Rightarrow \mathbf{C}(\mathsf{Q}, \ell, \_, \Psi)$*, vale:*

$$\ell(\bar{h}') = \bar{0} \quad con \quad \bar{h}' = dom(\ell) \setminus \bar{h} \tag{1.1}$$

$$se \quad \ell(\bar{h}) > \bar{0} \quad allora \quad \exists\, \mathbf{C}(\mathsf{Q}, \ell, \_, \Psi) \Rightarrow \mathbf{C}(\mathsf{Q}', \ell', \_, \Psi') \quad t.c. \quad \ell'(\bar{h}) = \bar{0} \tag{1.2}$$

La *Separate Liquidity* è una proprietà più debole di *Liquidity*, ma verificarne la presenza sarà meno costoso. Infatti la *Separate Liquidity* richiede, che per ogni asset k, sia sempre possibile svuotare k in qualche istante; mentre *Liquidity* richiede che nello stesso istante tutti gli asset siano vuoti.

Nei contratti `Ping_Pong` e `Ping_Pong_Sink` non c'è mai una configurazione in cui `hA` e `hM` sono contemporaneamente vuoti (tolta quella *iniziale* prima che `Mary` chiami `ping`). Entrambi i contratti non soddisfano quindi la proprietà di *Liquidity*. Tuttavia, mentre `Ping_Pong_Sink` può effettivamente mantenere valore intrappolato all'interno del contratto, `Ping_Pong` mostra che la mancata soddisfazione di tale proprietà non coincide sempre con l'intuizione di "assenza di fondi permanentemente bloccati".

Osserviamo quindi che la proprietà di *Liquidity* qui adottata è volutamente più forte dell'intuizione informale di "assenza di fondi bloccati". Essa richiede infatti la possibilità di raggiungere una configurazione in cui tutti gli asset risultino simultaneamente vuoti; per questo motivo può fallire anche in contratti che, intuitivamente, permettono comunque alle parti di recuperare il valore nel tempo.

Relazione tra cluster e *Separate Liquidity* Sia `C` un contratto. Se ogni cluster generato dalle computazioni astratte di `C` contiene un solo asset globale, allora la proprietà di *Separate Liquidity* è sufficiente a garantire che nessun valore rimanga bloccato nel contratto. Se invece esiste un cluster che contiene almeno due asset globali, *Separate Liquidity* non è più sufficiente, ed è necessario verificare la proprietà di *Liquidity*.

Nel primo caso, infatti, non essendoci movimenti tra asset distinti del contratto, lo svuotamento di un asset può avvenire solo tramite trasferimento del suo valore a una delle parti. Di conseguenza, se `C` soddisfa la *Separate Liquidity*, ogni asset non vuoto può essere reso vuoto solo facendo uscire il relativo valore dal contratto.

Nel secondo caso, invece, la possibilità di spostare valore tra asset appartenenti allo stesso cluster impedisce di concludere, a partire dalla sola *Separate Liquidity*, che tale valore esca effettivamente dal contratto. Per escludere questa possibilità è quindi necessario richiedere la proprietà più forte di *Liquidity*, che garantisce l'esistenza di una configurazione in cui tutti gli asset risultano simultaneamente vuoti.

3.2 Casi Limite

Analizziamo adesso alcuni aspetti particolari nel calcolo della liquidità attraverso dei contratti in `Stipula` che meritano un approfondimento.

3.2.1 Falsi Risultati

Seguendo la sintassi di `Stipula`, vediamo come sia possibile dichiarare delle clausole sotto forma di *evento* o di *funzione* (Sezione 2.1.2).

Gli **eventi** sono attivati da un'espressione temporale: quando ci troviamo nello stato iniziale dell'evento e raggiungiamo la quantità di *tick* dichiarata, vengono eseguite le azioni al suo interno. In questo contesto, è possibile dichiarare eventi temporalmente irraggiungibili [4]. Ad esempio, nella stessa funzione potremmo dichiarare due eventi:

1. `now+1D` $\gg$ `@Q0 {...} => @Q1`
2. `now` $\gg$ `@Q1 {...} => @Q2`

Poniamo che l'evento (1) sia l'unica clausola del contratto che porta allo stato `Q1`. L'evento (2) sarà quindi attivabile solamente in seguito all'evento (1). Quest'ultimo, tuttavia, verrà attivato un giorno `(+1D)` dopo `now`; di conseguenza, nel momento in cui ci troviamo per la prima volta in `Q1`, non sarà più possibile attivare l'evento (2), che risulterà quindi irraggiungibile (irraggiungibilità temporale) [4].

Le **funzioni** possono essere soggette a una *guardia*: in questo caso, quando una parte richiama la funzione, viene prima valutata l'espressione booleana dichiarata nella guardia. Se l'espressione ha valore `False`, le operazioni all'interno della funzione non vengono eseguite. In questo contesto, è quindi possibile dichiarare funzioni con guardie sempre false.

Vediamo quindi che possono esistere configurazioni irraggiungibili causate della presenza di qualche *clausola non eseguibile*. Queste stesse configurazioni possono portare, nella nostra analisi di liquidità, a falsi positivi (se all'interno della *clausola* era presente un'operazione che svuotava l'asset analizzato) o a falsi negativi (se all'interno della *clausola* era presente un'operazione che riempiva l'asset analizzato).

Analizziamo ciò che succede nel contratto `False_Negative`.

```
stipula False_Negative {
    assets h1, h2
    fields x

    agreement(Matteo) { Matteo: x } => @Q0

    @Q0 Matteo: init()[u]{
        u -o h2

        now + 1D >> @Q1 {} => @Q2
        now >> @Q2 {
```

```
            h2 -o h1
        } => @Q2
    } => @Q1

    @Q2 Matteo: end()[]{
        h2 -o Matteo
    } => @Q3
}
```

Contratto 3.3: False_Negative

Valutando la `h1`-*Separate Liquidity*, osserviamo che l'unica configurazione in cui `h1` risulterebbe pieno è quella successiva all'esecuzione dell'evento `now >> @Q2 {...} => @Q2`. Non esistono poi configurazioni successive che possano svuotare `h1`. Come analizzato in precedenza, però, quella clausola non sarebbe mai eseguibile, in quanto raggiungibile solo dopo l'esecuzione dell'evento `now + 1D >> @Q1 {...} => @Q2`.

Il contratto quindi possiede solamente la proprietà di `h2`-*Separate Liquidity* (data la presenza di `@Q2 Matteo:end @Q3`) e risulterà non liquido, anche se non è possibile che dei fondi rimangano bloccati perennemente in `h1`, poiché `h1` non potrà mai assumere un valore > 0.

3.2.2 Funzioni cicliche

Seguendo la sintassi di `Stipula`, vediamo come sia possibile dichiarare clausole che portano a cicli tra funzioni o eventi (Sezione 2.1.2). Nell'analisi della liquidità ci troveremmo quindi davanti a un numero infinito di configurazioni possibili, che causerebbero la non terminazione dell'analisi. È quindi necessario definire un *numero massimo di occorrenze* κ di una funzione all'interno della stessa computazione, valutando quindi solo computazioni κ-canoniche.

La scelta di questo valore è molto importante. Vediamo perché analizzando il contratto `Ugly`.

```
stipula Ugly {
    assets w1, w2
    fields x

    agreement(Mark, Sam) {
        Mark: x
    } => @Q0

    @Q0 Mark: get()[u]{
```

```
        u -o w2
    } => @Q1

    @Q1 Sam: shift()[]{
        w1 -o Sam
        w2 -o w1
    } => @Q1
}
```

Contratto 3.4: Ugly

Il contratto con $\kappa = 1$, cioè considerando il caso in cui una funzione non può comparire due volte nella stessa computazione (le computazioni astratte saranno *1-canoniche*), genera le seguenti configurazioni con codice residuo vuoto:

1. $\texttt{Ugly(Q1,}\ \ell[\texttt{w1} \mapsto \mathbb{0}, \texttt{w2} \mapsto \mathbb{1}]\texttt{,}\ \_\texttt{,}\ \_\texttt{)}$;
2. $\texttt{Ugly(Q1,}\ \ell[\texttt{w1} \mapsto \mathbb{1}, \texttt{w2} \mapsto \mathbb{0}]\texttt{,}\ \_\texttt{,}\ \_\texttt{)}$.

Per la configurazione (2), generata dopo le chiamate di `get` e `shift`, risulta impossibile svuotare l'asset `w1`. Il contratto sarà quindi non liquido, anche se `Sam` potrebbe svuotare l'asset effettuando una nuova chiamata a `shift`, senza generare alcun blocco permanente di valori patrimoniali.

Valutiamo adesso lo stesso contratto con $\kappa = 2$. Come anticipato, richiamando due volte di seguito la funzione `shift`, si verifica che il contratto è liquido. Infatti, inizialmente l'unica azione possibile è chiamare la funzione `get`, alla quale può seguire la funzione `shift`. A questo punto ci troveremmo nella configurazione (2) precedentemente analizzata. Con una nuova chiamata a `shift`, `Sam` può ritirare il valore immagazzinato in `w1`, rendendo tutte le configurazioni prive di fondi bloccati perennemente.

Il contratto `Ugly` risulta quindi liquido, secondo la nostra definizione, solo per $\kappa > 1$.

3.3 Algoritmi per il calcolo della liquidità

Prima di studiare gli algoritmi per il calcolo delle proprietà di *Separate Liquidity* e *Liquidity*, dobbiamo introdurre alcune notazioni che ci saranno utili [7].

Ambiente Un *ambiente* Ξ associa agli asset di un contratto, sia globali sia passati come parametro, la loro *espressione di liquidità*.

Tipo di liquidità di una funzione Associamo a ogni funzione un *tipo di liquidità* `Q A.f Q'`: $\Xi \to \Xi'$, in cui $\Xi \to \Xi'$ descrive gli effetti della completa esecuzione di `A.f` sulle espressioni di liquidità dell'ambiente Ξ.

Tipo di liquidità di un evento Associamo a ogni evento un *tipo di liquidità* `Q` $\texttt{ev}_i$ `Q'`: $\Xi \to \Xi'$, in cui $\Xi \to \Xi'$ descrive gli effetti della completa esecuzione di $\texttt{ev}_i$ sulle espressioni di liquidità dell'ambiente Ξ.

Giudizio Un *giudizio* è un'espressione che attribuisce a un costrutto del linguaggio il suo effetto sui tipi di liquidità. Un giudizio può avere una delle seguenti forme:

- $\Xi \vdash_{\texttt{X}} S : \Xi'$ per le istruzioni;
- $\Xi \vdash_{\texttt{X}} \texttt{@Q}\ A : f(\bar{x})[\bar{h'}]\{S\} \Rightarrow \texttt{@Q}' : \mathcal{L}$ per le definizioni di funzione;
- $\Xi \vdash_{\texttt{X}} \texttt{t >> @Q}\ \{S\} \Rightarrow \texttt{@Q}' : \mathcal{L}$ per le definizioni di evento.

Qui $\mathcal{L}$ denota il tipo di liquidità $\Xi \to \Xi'$ della funzione o dell'evento. L'insieme `X` contiene i nomi delle parti e dei campi del contratto.

Definiamo ora un *sistema di tipi di liquidità*, basato su giudizi $\vdash_{\texttt{X}}$, per i costrutti di `Stipula`. Tale sistema consente di assegnare tipi di liquidità alle *istruzioni*, alle *funzioni* e agli *eventi*, e quindi di determinare il comportamento delle *computazioni astratte* dal punto di vista della liquidità.

L'intero sistema di regole è riportato nella Tabella 3.1 [7].

Approfondiamo ora alcune regole; le altre hanno un *comportamento analogo*. La regola [L-Function] specifica le seguenti premesse:

- `A`, $fn(S) \subseteq \texttt{X} \cup \bar{y}$ richiede che i nomi usati nella funzione, cioè `A` e quelli contenuti in $fn(S)$, siano leciti, ossia appartengano a `X` oppure ai campi passati come parametro;
- $\bar{\xi}'$ *fresh* richiede che per i parametri asset $\bar{k}$ vengano utilizzati nuovi nomi di variabile;
- $\Xi[\bar{k} \mapsto \bar{\xi}'] \vdash_{\texttt{X} \cup \bar{y}} S : \Xi'$ richiede che il corpo della funzione S venga valutato nell'ambiente Ξ, esteso con i parametri asset $\bar{k}$.

La regola [L-Function] stabilisce poi che, verificate tali premesse, alla funzione `@Q A:f(`$\bar{y}$`)[`$\bar{k}$`]{S}`$\Rightarrow$`@Q'` viene assegnato il tipo di liquidità `Q A.f Q'` : $\Xi[\bar{k} \mapsto \bar{\mathbb{1}}] \to \Xi'\{\bar{\mathbb{1}}/\bar{\xi}'\}$, dove Ξ' è l'ambiente risultante dalla valutazione di S.

$$\frac{\texttt{A},\ fn(E)\ \subseteq\ \texttt{X} \cup dom(\Xi)}{\Xi\ \vdash_{\texttt{X}}\ E \to \texttt{A}\ : \Xi}\ [\text{L-Send}] \qquad \frac{\texttt{x},\ fn(E) \subseteq\ \texttt{X} \cup dom(\Xi)}{\Xi\ \vdash_{\texttt{X}}\ E \to \texttt{x}\ : \Xi}\ [\text{L-Update}]$$

$$\frac{h \in dom(\Xi) \qquad \texttt{A} \in \texttt{X}}{\Xi\ \vdash_{\texttt{X}}\ h \multimap h, \texttt{A}\ : \Xi[h \mapsto \mathbb{0}]}\ [\text{L-aSend}] \qquad \frac{h, h' \in dom(\Xi) \qquad \mathbb{e} = \Xi(h) \sqcup \Xi(h')}{\Xi\ \vdash_{\texttt{X}}\ h \multimap h, h'\ : \Xi[h \mapsto \mathbb{0},\ h' \mapsto \mathbb{e}]}\ [\text{L-aUpdate}]$$

$$\frac{h \in dom(\Xi) \qquad E \neq h \qquad \texttt{A} \in \texttt{X}}{\Xi\ \vdash_{\texttt{X}}\ E \multimap h, \texttt{A}\ : \Xi}\ [\text{L-ExpASend}] \qquad \frac{\mathbb{e} = \Xi(h) \sqcup \Xi(h') \quad h, h' \in dom(\Xi) \qquad E \neq h}{\Xi\ \vdash_{\texttt{X}}\ E \multimap h, h'\ : \Xi[h' \mapsto \mathbb{e}]}\ [\text{L-ExpAUpd}]$$

$$\frac{fn(E) \subseteq \texttt{X} \cup dom(\Xi) \qquad \Xi\ \vdash_{\texttt{X}}\ S : \Xi' \qquad \Xi\ \vdash_{\texttt{X}}\ S' : \Xi'' \qquad \Xi' \sqcup \Xi''\ \vdash_{\texttt{X}}\ S'' : \Xi'''}{\Xi\ \vdash_{\texttt{X}}\ \texttt{if}(E)\{S\}\texttt{else}\{S'\}\ S'' : \Xi'''}\ [\text{L-Cond}]$$

$$\Xi\ \vdash_{\texttt{X}}\ \_\ :\ \Xi\ [\text{L-Zero}] \qquad \frac{\Xi\ \vdash_{\texttt{X}}\ P : \Xi' \qquad \Xi'\ \vdash_{\texttt{X}}\ S : \Xi''}{\Xi\ \vdash_{\texttt{X}}\ P\ S : \Xi''}\ [\text{L-Seq}]$$

$$\frac{\texttt{A},\ fn(S) \subseteq \texttt{X} \cup \bar{y} \qquad \bar{\xi}'\ fresh \qquad \Xi[\bar{k} \mapsto \bar{\xi}']\ \vdash_{\texttt{X}\cup\bar{y}}\ S : \Xi'}{\Xi\ \vdash_{\texttt{X}}\ \texttt{@Q A} : \texttt{f}(\bar{y})[\bar{k}]\{S\} \Rightarrow \texttt{@Q}' : \texttt{Q A.f Q}'\ : \Xi[\bar{k} \mapsto \bar{\mathbb{1}}] \to \Xi'\{\bar{\mathbb{1}}/\bar{\xi}'\}}\ [\text{L-Function}]$$

$$\frac{fn(S) \subseteq \texttt{X} \qquad \Xi\ \vdash_{\texttt{X}}\ S : \Xi'}{\Xi\ \vdash_{\texttt{X}}\ \mathbb{t} \gg_{ev_i} \texttt{@Q}\{S\} \Rightarrow \texttt{@Q}' : \texttt{Q ev}_i\ \texttt{Q}' : \Xi \to \Xi'}\ [\text{L-Event}]$$

$$\frac{\bar{\xi}\ fresh \qquad \left([\bar{h} \mapsto \bar{\xi}]\ \vdash_{\bar{\texttt{A}}\cup\bar{\texttt{x}}}\ F : \mathcal{L}_F\right)^{F\in\bar{F}}}{\vdash\ \texttt{stipula C}\ \{\texttt{assets}\ \bar{h}\ \ \texttt{fields}\ \bar{\texttt{x}}\ \texttt{agreement}(\bar{\texttt{A}})\{...\}\texttt{=>@Q}\ \ \bar{F}\} : \bigcup_{F\in\bar{F}} \mathcal{L}_F}\ [\text{L-Contract}]$$

Tabella 3.1: Il sistema di tipi di liquidità di Stipula

Le regole [L-ASend], [L-ExpASend], [L-AUpdate] e [L-ExpAUpd] mostrano invece come le operazioni di movimento influiscano sul tipo di liquidità. Ad esempio, nella regola [L-ASend] vediamo che l'intero valore dell'asset h viene trasferito a A; di conseguenza, nell'ambiente risultante il valore associato a h deve essere posto uguale a $\mathbb{0}$. Al contrario, nella regola [L-ExpASend] si assume che $E \neq h$. Ricordando che la semantica operazionale impedisce che il valore di E sia maggiore del valore contenuto in h, possiamo concludere che non tutto il valore dell'asset h viene trasferito a A; per questo motivo, il valore di h non deve essere modificato

nell'ambiente risultante [7].

Sulla base delle nozioni di *ambiente*, di *tipo di liquidità* per funzioni ed eventi, di *giudizio* e del relativo *sistema di tipi di liquidità*, possiamo ora formulare la definizione di tipo di liquidità di una computazione astratta [7].

Definizione 6 (Tipo di Liquidità di una Computazione Astratta)**.** *Sia* $\vdash C : \mathcal{L}^2$ *e siano* $\bar{h}$ *gli asset di* C*. Sia inoltre* $\mathtt{Q}_i\ \mathtt{c}_i\ \mathtt{Q}_{i+1} : \Xi_i \to \Xi'_i \in \mathcal{L}$ *per ogni* $i \in 1..n$*, con* $\mathtt{c}_i = \mathtt{A}_i\mathtt{.f}_i \mid \mathtt{ev}_i$*.*

Il tipo di liquidità *di* $\varphi = \{\mathtt{Q}_i\ \mathtt{c}_i\ \mathtt{Q}_{i+1}\}^{i\in 1..n}$*, indicato con* $\mathtt{L}_\varphi$*, è*

$$\Xi_1^{(b)}|_{\bar{h}} \to \Xi_n^{(e)}|_{\bar{h}} \quad [3]$$

dove $\Xi_1^{(b)}$ *e* $\Xi_n^{(e)}$ *sono definiti come segue*[4]*:*

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

La Definizione 6 mostra che il tipo di liquidità di una computazione astratta si ottiene componendo, in sequenza, i tipi di liquidità delle clausole che la formano. Ogni clausola $\mathtt{Q}_i\ \mathtt{c}_i\ \mathtt{Q}_{i+1}$ contribuisce infatti con il proprio tipo di liquidità $\Xi_i \to \Xi'_i$, e il calcolo complessivo ne propaga gli effetti lungo tutta la computazione. Il contesto iniziale della prima clausola coincide con il suo ambiente iniziale, cioè $\Xi_1^{(b)} = \Xi_1$. Per le clausole successive, invece, l'ambiente di partenza viene aggiornato utilizzando le informazioni prodotte dal passo precedente sugli asset globali $\bar{h}$. In modo analogo, anche l'ambiente finale di ciascun passo si ottiene adattando il tipo di liquidità in uscita della clausola al contesto effettivo in cui essa viene eseguita. Il tipo finale $\mathtt{L}_\varphi$ riassume quindi l'effetto complessivo della computazione sui soli asset globali del contratto.

Introduciamo ora un'ultima notazione che sarà utilizzata negli algoritmi.

Definizione 7 ($\mathbb{T}^\kappa_{\mathtt{Q}}$)**.** *Definiamo* $\mathbb{T}^\kappa_{\mathtt{Q}}$ *come l'insieme degli elementi* $\mathtt{Q} \overset{\varphi}{\curvearrowright} \mathtt{Q}' : \mathcal{L}_\varphi$*, dove* φ *è una computazione* κ*-canonica del contratto che inizia dallo stato* $\mathtt{Q}$*.*

A partire da queste definizioni, possiamo ora descrivere gli algoritmi utilizzati per verificare le proprietà di liquidità.

[2] Forma breve per: $\vdash$ `stipula` C {`assets` $\bar{h}$ `fields` $\bar{x}$ `agreement(`$\bar{A}$`){...}=>@Q` $\bar{F}$} : $\mathcal{L}$.

[3] $\Xi|_{\bar{h}}(k) = \begin{cases} \Xi(k) & \text{se } k \in \bar{h} \\ undefined & \text{altrimenti} \end{cases}$

[4] "*b*" sta per "begin", mentre "*e*" sta per "end".

Algoritmo 3.1 Calcolo della Liquidity

Sia $\mathtt{Q}$ lo stato iniziale di C, i cui asset sono $\bar{h}$.

Passo 1. Calcola $\mathbb{T}^{\kappa}_{\mathtt{Q}'}$ per ogni $\mathtt{Q'}$ raggiungibile da $\mathtt{Q}$; poni $\mathcal{Z} \leftarrow \varnothing$.

Passo 2. Per ogni $\mathtt{Q}'$ e ogni $\mathtt{Q}' \overset{\varphi}{\curvearrowright} \mathtt{Q} : \Xi \to \Xi' \in \mathbb{T}^{\kappa}_{\mathtt{Q}'}$ e $\varnothing \subsetneq \bar{k} \subseteq \bar{h}$ è l'insieme più grande tale che:
(a) $\forall k' \in \bar{k}, [\![\Xi'(k')]\!] \neq \mathbb{0}$ e $[\![\Xi'(k')]\!] \neq [\![\Xi(k')]\!]$
(b) $(\mathtt{Q}'', \bar{k}) \notin \mathcal{Z}$

Passo 2.1 Se non esistono tali $\mathtt{Q}'$, $\mathtt{Q}' \overset{\varphi}{\curvearrowright} \mathtt{Q}''' : \Xi \to \Xi'$ e $\bar{k}$, allora **termina**: il contratto è *Liquid*.

Passo 2.2 Altrimenti verifica se esiste $\mathtt{Q}'' \overset{\varphi'}{\curvearrowright} \mathtt{Q}''' : \Xi'' \to \Xi''' \in \mathbb{T}^{\kappa}_{\mathtt{Q}''}$ tale che $[\![\Xi'''(\bar{k})]\!] = \bar{\mathbb{0}}$ e, per ogni $k' \in \bar{h} \setminus \bar{k}$, vale $[\![\Xi'''(k')]\!] = \mathbb{0}$ o $[\![\Xi'''(k')]\!] = [\![\Xi''(k')]\!]$. Se questo vale, aggiungi $(\mathtt{Q}'', \bar{k})$ in $\mathcal{Z}$ e ripeti il **Passo 2**; altrimenti **termina**: il contratto non è *Liquid*.

Algoritmo 3.2 Calcolo della *k-Separate Liquidity*

Sia $\mathtt{Q}$ lo stato iniziale di C, i cui asset sono $\bar{h}$.

Passo 1. Calcola $\mathbb{T}^{\kappa}_{\mathtt{Q}'}$ per ogni $\mathtt{Q}'$ raggiungibile da $\mathtt{Q}$; poni $\mathcal{Z} \leftarrow \varnothing$.

Passo 2. Per ogni $\mathtt{Q}'$ e ogni $\mathtt{Q}' \overset{\varphi}{\curvearrowright} \mathtt{Q}'' : \Xi \to \Xi' \in \mathbb{T}^{\kappa}_{\mathtt{Q}'}$ tali che:
(a) $[\![\Xi'(k)]\!] \neq \mathbb{0}$ e $[\![\Xi'(k)]\!] \neq [\![\Xi(k)]\!]$
(b) $(\mathtt{Q}'', k) \notin \mathcal{Z}$

Passo 2.1 Se non esistono tali $\mathtt{Q}'$ e $\mathtt{Q}' \overset{\varphi}{\curvearrowright} \mathtt{Q}'' : \Xi \to \Xi'$, allora **termina**: il contratto è *k-Separate Liquid*.

Passo 2.2 Altrimenti verifica se esiste $\mathtt{Q}'' \overset{\varphi'}{\curvearrowright} \mathtt{Q}''' : \Xi'' \to \Xi''' \in \mathbb{T}^{\kappa}_{\mathtt{Q}''}$ tale che $[\![\Xi'''(k)]\!] = \mathbb{0}$. Se questo vale, aggiungi $(\mathtt{Q}'', k)$ in $\mathcal{Z}$ e ripeti il **Passo 2**; altrimenti **termina**: il contratto non è *k-Separate Liquid*.

L'Algoritmo 3.1 usa l'insieme $\mathbb{T}^{\kappa}_{\mathtt{Q}'}$ (per ogni stato $\mathtt{Q'}$ raggiungibile da $\mathtt{Q}$) per identificare delle "coppie critiche" ($\mathtt{Q"}$,$\bar{k}$) tali che esista una computazione che modifichi gli asset appartenenti all'insieme $\bar{k}$ e che termini nello stato $\mathtt{Q"}$. L'insieme $\mathcal{Z}$ contiene coppie ($\mathtt{Q}$,$\bar{k}$) ed è utilizzato per non riesaminare la stessa computazione astratta più volte. Assumiamo che ($\mathtt{Q"}$,$\bar{k}$)$\notin \mathcal{Z}$. Allora dovremmo trovare $\mathtt{Q}'' \overset{\varphi'}{\curvearrowright} \mathtt{Q}''' : \Xi'' \to \Xi'''$ in $\mathbb{T}^{\kappa}_{\mathtt{Q}''}$ tale che $[\![\Xi'''(\bar{k})]\!] = \bar{\mathbb{0}}$ e tale che gli altri asset $\bar{h} \setminus \bar{k}$ siano uguali a $\mathbb{0}$ oppure non siano stati modificati (rispetto al valore in Ξ''. Se non troviamo l'elemento in $\mathbb{T}^{\kappa}_{\mathtt{Q}''}$, la liquidità **non potrà essere garantita** e l'algoritmo risponderà in modo negativo (il che potrebbe essere un falso negativo perché l'elemento $\mathtt{Q}'' \overset{\varphi'}{\curvearrowright} \mathtt{Q}''' : \Xi'' \to \Xi'''$

potrebbe essere trovato in $\mathbb{T}^{\kappa+1}_{\mathtt{Q}''}$).

L'Algoritmo 3.2 è simile al precedente, anzi più semplice in quanto esamina un solo asset k alla volta nella sua esecuzione.

Gli algoritmi 3.1 e 3.2 hanno rispettivamente costo computazionale $O(n+N+N^2 \times 2^h)$ e $O(n+N+N^2)$, dove: n è la dimensione del contratto `Stipula` (numero di funzioni, prefissi e condizionali nel codice); h è il numero degli asset; m è il numero degli stati; m' è il numero delle funzioni; $N = m \times (\Sigma_{0 \leq i \leq \kappa \times m'} i!)$. Assumendo che il numero di asset h sia limitato ad una costante (nei contratti realistici) e che gli altri valori siano in relazione lineare a m', possiamo dire che entrambi i costi computazionali siano $O(N^2)$, *i.e.* esponenziale rispetto a m' [7].

Capitolo 4

Implementazione

4.1 Introduzione

Il progetto è stato interamente pubblicato sulla piattaforma GitHub [8] sotto licenza GNU GPL-3.0 [6].

L'implementazione è realizzata prevalentemente in linguaggio **Python** [13], con l'eccezione della grammatica di Stipula e delle componenti prodotte automaticamente tramite il tool **ANTLR** [10].

La parte di codice composta dai file riguardanti ANTLR, così come il loro utilizzo di base, sono stati completamente ereditati, senza apportare modifiche dal progetto Stipula Analyzer [5].

4.2 Tecnologie Usate

ANTLR (ANother Tool for Language Recognition) è un generatore di parser per linguaggi di programmazione. A partire dalla definizione formale della grammatica di un linguaggio, fornita come file di input, il software consente di generare automaticamente gli strumenti necessari all'analisi dei programmi, quali lexer, parser e visitor. L'output prodotto da ANTLR è disponibile per i principali linguaggi di programmazione e fornisce uno scheletro su cui implementare le funzionalità desiderate. Nel nostro caso, il codice generato da ANTLR è stato prodotto in Python.

Python è un linguaggio di programmazione ad alto livello, orientato agli oggetti e ampiamente utilizzato sia in ambito accademico sia industriale. La scelta di Python come linguaggio per l'implementazione del software è motivata da diversi fattori.

In primo luogo, Python è multi-piattaforma: può essere installato ed eseguito su numerosi sistemi operativi, tra cui Linux, macOS, Windows e Android, garantendo così una buona portabilità del progetto. Inoltre, il linguaggio si distingue per la sua facilità d'uso: una volta installato l'interprete, i programmi possono essere eseguiti direttamente senza la necessità di fasi di compilazione complesse.

Un ulteriore vantaggio è rappresentato dalla disponibilità di un package manager dedicato, *pip*, che consente di installare e gestire in modo semplice le librerie esterne non incluse nella distribuzione standard di Python.

In questo senso, i pacchetti esterni a noi necessari per utilizzare il *Liquidity Analyzer* sono racchiusi nel file *requirements.txt* nella directory del progetto:

- **antlr4-python3-runtime** [12]: contiene le classi base di ANTLR implementate in Python;
- **click** [15]: permette di gestire uno script Python come comando da terminale strutturando le opzioni e i parametri in input.

4.3 Struttura

Il progetto, includendo solo file o directory utili al software, è così strutturato:

- **main.py** dato in input un contratto scritto in Stipula (o una cartella di programmi), permette di analizzare la liquidità del contratto (o dei contratti);
- **classes/** contiene tutte le classi create per questo software:
  - **liquidity_analyzer.py** contiene la classe *LiquidityAnalyzer*;
  - **liquidity_visitor.py** contiene la classe *LiquidityVisitor* (sottoclasse di *StipulaVisitor*);
  - **data/** contiene le classi relative ai dati complessi necessari a *liquidity_analyzer.py* e *liquidity_visitor.py*, cioè:
    - **abstract_computation.py** contiene la classe *AbsComputation*;
    - **asset_types.py** contiene la classe *AssetTypes*;
    - **liquidity_expression.py** contiene le classi *LiqConst* e *LiqExpr*;
    - **visitor_entry.py** contiene le classi *VisitorEntry*, *EventVisitorEntry* e *FunctionVisitorEntry*;

- **generated/** contiene i file (*StipulaLexer.py*, *StipulaParser.py*, *StipulaVisitor.py*) generati automaticamente da ANTLR attraverso il comando:

```
antlr4 -Dlanguage=Python3 Stipula.g4
```

Nelle prossime sezioni ci concentriamo nel commentare gli script e le classi principali del progetto [8].

4.4 main.py

Il file *main.py* è lo script in Python che funge da punto di ingresso per l'utilizzo del software per l'analisi delle liquidità di un contratto in Stipula. Lo script è eseguibile da linea di comando e prende in input un percorso ad un file o ad una cartella, un numero e un'opzione attraverso il pacchetto *click* [15].

Quindi controlla se il percorso è associato a un contratto, cioè un file con estensione `".stipula"`, in questo caso richiama la funzione `run()` con il percorso al file. In caso negativo, controlla se il percorso è associato ad una cartella, richiamando la funzione `run()` per ogni file con estensione `".stipula"` al suo interno. Se non è neanche questo il caso, viene sollevata un'eccezione.

La funzione `run()` accetta tre parametri: il *percorso* al file `".stipula"`, il *numero massimo* di occorrenze di una funzione all'interno di una computazione e un *flag* `is_verbose`. Questa variabile booleana è impostata a `True` se lo script era stato eseguito con opzione `-v` o `--verbose`, in questo caso l'output mostrato sarà composto anche dalle motivazioni del risultato e da informazioni generali sul contratto (o sui contratti) da analizzare. La funzione si sviluppa quindi con:

1. **lettura del file di input**, attraverso la funzione `antlr4.FileStream(file_path)`;
2. **generazione del lexer**, istanziando un oggetto di classe `StipulaLexer`, contenuta nei file generati automaticamente da ANTLR;
3. **generazione della lista di token**, attraverso la funzione `antlr4.CommonTokenStream(lexer)`;
4. **generazione del parser**, istanziando un oggetto di classe `StipulaParser`, contenuta nei file generati automaticamente da ANTLR;
5. **generazione dell'albero di sintassi astratta**, attraverso la funzione `parser.stipula()`;

6. **controllo errori di sintassi**, attraverso la funzione `parser.getNumberOfSyntaxErrors()`;
7. **generazione del visitor**, creando un'istanza di `LiquidityVisitor`;
8. **visita dell'albero di sintassi astratta**, attraverso il metodo `visit(tree)` contenuta all'interno della classe `LiquidityVisitor`.

Con l'esecuzione della visita dell'albero, il controllo passa quindi all'oggetto di classe `LiquidityVisitor`.

4.5 LiquidityVisitor

Come anticipato, la classe `LiquidityVisitor` estende la classe `StipulaVisitor` generata da ANTLR e visita l'albero di computazione astratta generato dal parser partendo con il metodo `visitStipula()` dalla radice dell'albero, cioè dal nodo corrispondente alla regola di parsing `stipula`.

È importante specificare che l'invocazione di `visitStipula()` non avviene direttamente tramite una chiamata esplicita, ma attraverso il meccanismo del visitor pattern implementato da ANTLR [11].

Quando viene eseguita l'istruzione `LiquidityVisitor().visit(tree)` in `main.py`, il metodo `visit()` (definito nella classe base `ParseTreeVisitor`) delega l'elaborazione al nodo radice attraverso una chiamata a `tree.accept(visitor)`.

Il metodo `accept()` è generato automaticamente da ANTLR per ciascun possibile nodo del parse tree ed invoca il metodo del visitor corrispondente alla regola grammaticale rappresentata dal nodo. Nel caso di un nodo associato alla regola `stipula` (radice dell'albero in quanto regola iniziale della nostra grammatica), il metodo `accept()` richiama `visitor.visitStipula(ctx)`. Essendo questo metodo ridefinito in LiquidityVisitor, l'invocazione viene risolta dinamicamente secondo le regole del dynamic binding di Python.

Attraverso l'oggetto `ctx` (context, generato da ANTLR, rappresenta un nodo specifico del parse tree) si accede alle informazioni sintattiche relative a quel nodo, inclusi eventuali sotto-nodi (figli) e token riconosciuti, e viene passato ai metodi del visitor per esplorare l'intero albero come desiderato.

Il metodo `visitStipula()` definito nella classe `LiquidityVisitor` costituisce il punto di ingresso per la visita dell'albero di sintassi (parse tree) generato da AN-

TLR. Esaminiamolo quindi per mostrare nel particolare uno dei metodi presenti in `LiquidityVisitor`.

Come specificato nella sintassi di *Stipula*, definita nel file `Stipula.g4` [8], la regola di parsing iniziale è l'eq. (2.1) della Sezione 2.1, definita nella grammatica come:

```
stipula : 'stipula' contractId=ID '{' assetsDecl? fieldsDecl?
   agreement functionDecl+ '}' EOF ;
```

La regola `stipula` descrive la struttura sintattica di un contratto: dopo la parola chiave `'stipula'` e l'identificatore del contratto (`ID`), sono previste opzionalmente le dichiarazioni di `assets` e `fields`, seguite obbligatoriamente dalla dichiarazione di `agreement` e da una o più dichiarazioni di funzione (`functionDecl+`).

Nel metodo `visitStipula()`, il parametro `ctx` è un'istanza di `StipulaContext`, classe generata automaticamente da ANTLR e associata alla parser rule `stipula`. Tale oggetto rappresenta il nodo radice corrispondente nel parse tree e fornisce metodi di accesso sia ai token terminali (ad esempio `ID`) sia ai nodi figli corrispondenti alle sotto-regole (`assetsDecl`, `fieldsDecl`, `agreement`, `functionDecl`).

Ad esempio, verifichiamo la presenza della sotto-regola `assetsDecl`, qualora essa sia presente, viene invocato il metodo `visitAssetsDecl()`, anch'esso ridefinito in `LiquidityVisitor`. Quest'ultimo visiterà il sotto-albero corrispondente, seguendo la definizione della regola `assetsDecl` nella grammatica, accedendo così agli identificatori degli assets dichiarati nel contratto.

Oltre ad esplorare le sue sotto-regole, `visitStipula()` si occupa anche di richiamare le funzioni `compute_results()` e `compute_results_verbose()` (Sezione 4.6.2), presenti in `LiquidityAnalyzer`, per calcolare e mostrare il risultato di liquidità del contratto. Nei restanti metodi del *visitor*, sono presenti interazioni con l'*analyzer* solo attraverso i metodi di tipo *getter/setter*, funzionali alla fase di calcolo finale eseguita al termine della visita, eseguito appunto al termine della visita dell'albero da `visitStipula()`.

Il resto della classe *LiquidityVisitor* è composto dai restanti metodi per la visita dell'albero, uno per ogni regola di parsing definita in `Stipula.g4`. Questi metodi effettuano una visita in profondità dell'albero e interagiscono con l'oggetto di classe *LiquidityAnalyzer* per fornire tutte le informazioni necessarie al calcolo della liquidità.

4.6 LiquidityAnalyzer

La classe `LiquidityAnalyzer` rappresenta il fulcro di calcolo e l'output dell'analizzatore. Un'istanza di `LiquidityAnalyzer` è un oggetto di cui vengono popolati gli attributi dal *visitor*; successivamente, su tale istanza viene avviata la computazione vera e propria tramite il metodo `compute_results()`.

4.6.1 Attributi

Analizziamo e descriviamo i principali attributi della classe `LiquidityAnalyzer`:

- `K`, numero massimo di occorrenze di una funzione in una computazione astratta (Sezione 3.2.2);
- `Q0`, stringa che identifica lo *stato iniziale* del contratto, questo attributo viene impostato dal *visitor* attraverso il metodo `set_q0()`;
- `global_assets`, insieme di identificatori degli *assets* definiti nell'intestazione del contratto, l'insieme viene popolato dal *visitor* attraverso il metodo `add_global_asset()`;
- `functions`, insieme di oggetti (uno per ogni funzione definita all'interno del contratto) di classe `FunctionVisitorEntry` (Sezione 4.7.4), l'insieme viene popolato dal *visitor* attraverso il metodo `add_visitor_function()`;
- `events`, insieme di oggetti (uno per ogni evento definito all'interno del contratto) di classe `EventVisitorEntry` (Sezione 4.7.3), l'insieme viene popolato dal *visitor* attraverso il metodo `add_visitor_event()`;
- `abs_computations`, insieme di oggetti di `AbsComputation` (Sezione 4.7.1), le computazioni vengono calcolate dal metodo `compute_abs_computation()`;
- `states`, insieme di identificatori degli *stati* del contratto, l'insieme viene popolato dal metodo `compute_states()`;
- `reachable_states`, insieme di identificatori degli *stati raggiungibili* del contratto, l'insieme viene popolato dal metodo `compute_reachable_states()`;
- `Tqk`, dizionario che associa ad ogni stato un set di *computazioni astratte* che partono da quello stato, il dizionario è popolato dal metodo `compute_tqk()` e implementa l'insieme $\mathbb{T}^{\kappa}_{\mathtt{Q}}$ (Definizione 7);

- `has_events`, flag che indica che all'interno del contratto era presente almeno un evento, il valore viene settato a `True` ad ogni invocazione del metodo `add_visitor_event()`;
- `has_guards`, flag che indica che all'interno del contratto era presente almeno una guardia (Sezione 2.1.2), il valore viene settato a `True` dal *visitor* utilizzando un parametro booleano del metodo `add_visitor_function()`.

4.6.2 Metodi

Di seguito viene riportata una breve descrizione dei principali metodi della classe `LiquidityAnalyzer`.

`compute_results()` Questo metodo permette al *visitor* di avviare il calcolo delle proprietà di *liquidità* del contratto e restituisce quattro risultati.

Il primo risultato è un dizionario che associa a ogni *asset* k una coppia `(bool, str)`[1], che rappresenta l'esito della proprietà di *k-Separate Liquidity*. Il secondo è una coppia `(bool, str)`[1] relativa alla proprietà di *complete liquidity*.

Infine, vengono restituiti due valori booleani che indicano rispettivamente la presenza di eventi dichiarati in una delle funzioni del contratto e la presenza di guardie sulle funzioni.

La funzione si occupa di richiamare gli altri metodi della classe per calcolare gli *stati*, le *computazioni astratte*, gli *stati raggiungibili* e il dizionario `Tqk`. Inoltre controlla, richiamando `compute_function_local_liquidity()`, che ogni funzione rispetti la liquidità locale. In caso negativo, termina la sua computazione seguendo il punto (1.1) delle Definizioni 3, 5 di *k-Separate Liquidity* e di *Liquidity*.

Quindi esegue *Costly algorithm for k-Separate Liquidity* (Algoritmo 3.2) e *Costly algorithm for Liquidity* (Algoritmo 3.1) chiamando rispettivamente i metodi `costly_algorithm_k_separate()` e `costly_algorithm_complete()` per poi ritornare tutti i risultati ottenuti.

`compute_results_verbose()` È il metodo che consente al *visitor* di mostrare le informazioni aggiuntive per la modalità *verbosa*. Verranno stampate le informazioni (cluster generati, tipo di liquidità) riguardanti le *funzioni* e gli *eventi* del contratto e

[1] Il valore booleano è `True` se il contratto soddisfa la proprietà, altrimenti è `False`; in questo caso la stringa contiene la motivazione della risposta negativa.

quelle (cluster risultanti, tipo di liquidità calcolato) riguardanti tutte le *computazioni astratte* generate.

`compute_function_local_liquidity()` Per ogni oggetto nell'insieme delle funzioni, controlla che la liquidità locale sia rispettata richiamando il metodo `compute_local_liquidity()` dell'oggetto di classe `FunctionVisitorEntry` (Sezione 4.7.4).

`compute_states()` Calcola tutti gli stati presenti nel contratto in base allo *stato iniziale* e *finale* di ogni funzione e evento dichiarato.

`compute_abs_computation()` Calcola tutte le possibili computazioni astratte partendo dallo stato iniziale del contratto seguendo l'Algoritmo 2.1.

`compute_reachable_states()` Calcola, tra gli stati del contratto, quali sono quelli raggiungibili dallo stato iniziale seguendo le computazioni astratte calcolate.

`compute_tqk()` Calcola, per ogni stato del contratto, tutti i segmenti di computazioni astratte che partono da quello stato, seguendo la Definizione 7. Quindi per ogni computazione astratta c (tra quelle già ottenute partendo dallo stato iniziale), ne esamina ogni configurazione d: sia s lo stato iniziale della configurazione, l'algoritmo aggiunge all'insieme `Tqk[s]` la computazione c troncata in modo che parta da d.

`costly_algorithm_k_separate()`, `costly_algorithm_k_separate_asset()`
Il metodo `costly_algorithm_k_separate()` richiama, per ogni asset, il metodo `costly_algorithm_k_separate_asset()`, il quale implementa l'Algoritmo 3.2. Viene quindi verificata la *Separate Liquidity*. In caso di risultato negativo, assieme al valore booleano `False` viene riportata la motivazione da stampare nella modalità *verbosa*.

`costly_algorithm_complete()` Il metodo implementa l'Algoritmo 3.1. Quindi viene verificata la *complete liquidity*. In caso di risultato negativo, assieme al valore booleano `False` viene riportata la motivazione da stampare nella modalità *verbosa*.

Oltre ai metodi citati, è importante sottolineare la presenza dei metodi *getter*/*setter*:

- `set_q0()`;

- `add_visitor_function();`
- `add_visitor_event();`
- `add_global_asset();`
- `get_global_asset();`

4.7 Classi di dati

Di seguito, viene riportata una breve descrizione (scopo, struttura e metodi principali) per ognuna delle classi di dati utilizzate nel *visitor* e nell'*analyzer*.

4.7.1 AbsComputation

La classe modella le *computazioni astratte* (Sezione 2.3). Un oggetto di classe `AbsComputation` è formato da una lista ordinata di funzioni o eventi che formano una possibile computazione astratta del contratto.

Ad ogni istanza è associata inoltre una lista di eventi (rappresentazione di Ψ della Definizione 1), un'istanza di `AssetTypes` (Sezione 4.7.2) e due dizionari che descrivono il tipo di liquidità della computazione (Definizione 6).

Il metodo più importante è `insert_configuration()` che l'*analyzer* richiama per manipolare la computazione. Il metodo, oltre ad aggiungere la clausola passata come parametro in coda alla lista, si occupa di aggiornare il tipo di liquidità della computazione, l'istanza di `AssetTypes` e la lista degli eventi. Questa viene modificata secondo i seguenti casi:

- se A è una funzione del contratto e l'evento E è definito all'interno di A, allora E viene aggiunto alla lista degli eventi;
- se A è un evento del contratto, allora A viene rimosso dalla lista. Per poterlo reinserire sarà necessario invocare prima `insert_configuration(`B`)`, dove B è la funzione che definisce A.

4.7.2 AssetTypes

La classe implementa una struttura dati che modella una partizione dinamica di un insieme finito di asset. Internamente, la partizione è rappresentata come un insieme di *cluster* (Sezione 3.1). Due asset inclusi in un'operazione di *movimento* (`-o`) appartengono allo stesso *cluster*.

La classe non ha l'obiettivo di tenere traccia dei tipi concreti degli asset (valute digitali, NFTs, ecc.), bensì di evidenziare la presenza di eventuali movimenti di beni tra asset all'interno di una computazione astratta, così da segnalare all'utente l'utilità dell'analisi della *Separate Liquidity*. (Sezione 3.1).

Dal punto di vista implementativo, la classe rappresenta una versione semplificata della struttura Union-Find. Il metodo `add_singleton()` crea un nuovo insieme all'interno della partizione, generando un nuovo componente *singleton* costituito dall'identificatore dell'asset con cui viene invocato il metodo. Il metodo `merge_types()` realizza invece l'unione tra gli insiemi contenenti gli asset passati come parametro. Se entrambi gli elementi sono presenti nella struttura, i due sottoinsiemi vengono rimossi e sostituiti dal loro insieme unione.

4.7.3 EventVisitorEntry

La classe estende `VisitorEntry` (Sezione 4.7.7) e rappresenta un *evento* del contratto. Oltre agli attributi e i metodi ereditati, introduce il campo `trigger`, utile ad identificare l'evento nella stampa dei risultati. Questo valore non è altro che l'espressione temporale nella dichiarazione dell'*evento* che scatena la transizione dallo stato *iniziale* allo stato *finale*.

4.7.4 FunctionVisitorEntry

La classe estende `VisitorEntry` (Sezione 4.7.7) e rappresenta una *funzione* del contratto. Oltre agli attributi ereditati, introduce il nome della *funzione*, il nome della *parte* che la può invocare, l'insieme dei parametri asset, un flag che indica la presenza di una guardia e una lista di *eventi* definiti al suo interno.

Gli asset *locali* (passati come parametro) vengono inizializzati con valore $\mathbb{1}$, come definito in Tabella 3.1 [7].

Oltre a quelli ereditati, la classe mette a disposizione un metodo per verificare se tutti gli asset locali risultano svuotati al termine dell'esecuzione e un metodo per aggiungere un elemento alla lista dedicata di *eventi* dichiarati all'interno della funzione.

4.7.5 LiqConst

La classe definisce un insieme di costanti simboliche utilizzate nel progetto.

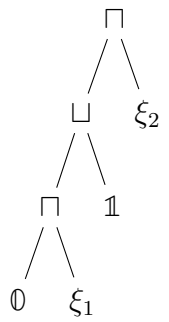
4.7.6 LiqExpression

La classe modella il concetto di *espressione di liquidità*. Nella Definizione 4 vediamo come un'espressione $\mathbb{e}$ possa essere:

- un valore costante ($\mathbb{0}$, $\mathbb{1}$);
- un valore variabile (ξ);
- il risultato di un'operazione ($\mathbb{e}' \sqcup \mathbb{e}''$, $\mathbb{e}' \sqcap \mathbb{e}''$).

Per rappresentare questa varietà, un'*espressione di liquidità* è caratterizzata da un albero binario in cui ogni nodo è un'istanza di `LiqExpression`. I nodi interni rappresentano gli operatori ($\sqcup$ e $\sqcap$), mentre le foglie rappresentano costanti o variabili. In questo modo è possibile rappresentare un'espressione complessa come una struttura composta da più istanze di `LiqExpression` annidate. Ogni istanza della classe contiene quindi tre informazioni: il valore del nodo e i riferimenti ai sotto-alberi sinistro e destro (eventualmente nulli nel caso di foglie).

Ad esempio, l'espressione $((\mathbb{0} \sqcap \xi_1) \sqcup \mathbb{1}) \sqcap \xi_2$ è rappresentata come segue:



Tra i metodi della classe troviamo `add_operation()`, che consente di espandere un'espressione creando una nuova operazione tra l'espressione corrente e una nuova espressione. L'istanza su cui viene invocato il metodo viene preventivamente copiata e assegnata come sotto-albero sinistro, l'espressione passata come parametro diventa il sotto-albero destro, il valore del nodo viene quindi aggiornato con l'operatore specificato. In questo modo si preserva la struttura precedente come operando sinistro della nuova operazione, mantenendo coerente la rappresentazione ad albero dell'espressione.

Il metodo `replace_value()` permette invece di sostituire ricorsivamente tutte le occorrenze di un determinato valore atomico (costante o variabile) con una nuova

espressione. Se il nodo corrente rappresenta un operatore, la sostituzione viene propagata ai sotto-alberi. Se invece il valore del nodo coincide con quello da sostituire, il nodo viene rimpiazzato con una copia dell'espressione fornita come parametro.

Infine, il metodo `resolve_partial_eval()` semplifica l'espressione passata come parametro, ritornando un'altra espressione più concisa con identico valore di liquidità, applicando la proprietà commutativa di $\sqcap$ e $\sqcup$ e gli assiomi $\mathbb{0} \sqcup \mathbb{e} = \mathbb{e}$, $\mathbb{0} \sqcap \mathbb{e} = \mathbb{0}$, $\mathbb{1} \sqcup \mathbb{e} = \mathbb{1}$, $\mathbb{1} \sqcap \mathbb{e} = \mathbb{e}$.

4.7.7 VisitorEntry

La classe rappresenta una generica transizione tra due stati del contratto (*start state*, *end state*), tenendo traccia della liquidità degli asset locali e globali all'inizio e alla fine della transizione. Essa mantiene due ambienti: un ambiente di ingresso (`input_env`), che associa a ciascun asset globale un'espressione di liquidità iniziale variabile (ξ_i), e un ambiente di uscita (`output_env`). Quest'ultimo è strutturato come una hashtable di liste che consente di gestire livelli annidati di blocchi mappando ogni asset ad una pila di valori.

La classe fornisce inoltre metodi per gestire i livelli dell'ambiente (aggiunta e rimozione di blocchi annidati), metodi per ottenere una rappresentazione chiara dell'ambiente iniziale e finale, metodi *getter*/*setter* per gli attributi della classe e metodi per conservare e manipolare l'istanza di `AssetTypes` (Sezione 4.7.2) della configurazione.

4.8 Utilizzo del Software

L'analizzatore è utilizzabile attraverso lo script Python `main.py` (Sezione 4.4), invocabile da console. La sintassi del comando è:

```
python main.py [-v | --verbose] <PATH> <FREQ>
```

In cui:

- `[-v | --verbose]` sono due parametri opzionali con lo scopo di aumentare la quantità di informazioni fornite all'utente nell'output, mostrando, oltre al risultato richiesto, anche le motivazioni degli esiti;
- `<PATH>` è un percorso nel filesystem locale. Il percorso può puntare al file contenente il contratto scritto in Stipula da analizzare oppure a una cartella, nel qual caso verranno analizzati tutti i file con estensione `".stipula"` al suo interno;

- `<FREQ>` è un numero intero positivo che rappresenta il numero massimo di occorrenze di una funzione all'interno di una computazione (Sezione 3.2.2).

4.8.1 Integrazione con Stipula-Workbench

Il progetto, oltre all'uso indipendente da linea di comando, è stato integrato nel progetto `Stipula-Workbench` [3].

Quando il contratto creato nel *workbench* viene sottoposto all'analisi di *liquidità*, il server riceve una richiesta HTTP e crea un file temporaneo con estensione `.stipula` contenente il contratto da analizzare. Viene quindi costruito il comando per eseguire lo script `main.py` utilizzando l'interprete Python del *virtual environment* del progetto e passando come argomenti il percorso del file temporaneo e la frequenza massima delle funzioni (numero preso in input nel *workbench*), insieme all'eventuale flag `--verbose`.

L'analizzatore viene eseguito tramite un processo figlio in modo asincrono. Il server raccoglie l'output prodotto sugli stream `stdout` e `stderr`; al termine dell'esecuzione il file temporaneo viene eliminato. In presenza di errori (`stderr` non vuoto) viene restituita una risposta HTTP con stato 500, altrimenti l'output dell'analisi viene restituito con stato 200 in formato JSON.

Il risultato viene infine visualizzato nell'interfaccia del *workbench*.

4.9 Analisi dei Risultati

4.9.1 Output dell'analizzatore

L'analizzatore, nella modalità non *verbosa*, indica se il contratto analizzato possiede la proprietà di *Separate Liquidity* (fornendo, per ogni asset k, il risultato della k-*Separate Liquidity*) e la proprietà di *Liquidity*.

Ad esempio, analizzando il Contratto 3.3 `False_Negative`, si ottiene:

```
False_Negative
  - is NOT Separate Liquid.
    - is NOT h1-Separate Liquid.
    - is h2-Separate Liquid.
  - is NOT Liquid.

  INFO:
    False_Negative contains transfers between distinct assets.
      - Separate Liquidity does not imply contract liquidity.
```

```
10      - Please refer to the Liquidity result.
11
12  WARNING:
13    False_Negative has events.
14      - h1-Separate Liquidity result could be a false negative.
15      - h2-Separate Liquidity result could be a false positive.
16      - Liquidity result could be a false negative.
```

L'output è composto da tre parti principali: il risultato dell'analisi delle proprietà di liquidità, eventuali informazioni aggiuntive (`INFO`) e possibili avvisi (`WARNING`) riguardo all'affidabilità del risultato.

4.9.2 Interpretazione dei messaggi

INFO Nella sezione `INFO` è presente un avviso all'utente riguardo quale proprietà di liquidità considerare in base ai *cluster* di asset risultanti dalle computazioni calcolate, secondo quanto stabilito nella Sezione 3.1.

Se nelle computazioni ottenute dal contratto non è presente alcun movimento tra due asset distinti, allora la *Separate Liquidity* è sufficiente a garantire che il contratto sia privo di fondi bloccati. Questo è il caso del Contratto 3.1 `Ping_Pong`.

Al contrario, se sono presenti movimenti tra asset distinti, un risultato positivo della *Separate Liquidity* non è sufficiente a garantire la liquidità del contratto. In questo caso è necessario considerare anche la proprietà di *Liquidity*. Questo è il caso del Contratto 3.2 `Ping_Pong_Sink`.

WARNING Nella sezione `WARNING` vengono segnalati possibili falsi risultati. Se il contratto contiene *eventi* o *guardie*, potrebbero esistere computazioni generate e valutate dall'analizzatore che in realtà non possono mai verificarsi (a causa della raggiungibilità temporale di un evento [4] o di una guardia di funzione sempre falsa) (Sezione 3.2.1).

Una computazione di questo tipo potrebbe essere valutata come non liquida, rendendo l'intero contratto non liquido e producendo quindi un falso negativo (proprio come nell'esempio `False_Negative`). Al contrario, una computazione resa liquida esclusivamente da una funzione con guardia sempre falsa o da un evento non raggiungibile potrebbe portare a un falso positivo.

Per questo motivo, in presenza di *guardie* o *eventi* nel contratto, l'analizzatore si limita a segnalare la possibilità di falsi risultati senza verificarne l'effettiva occorren-

za. In particolare, per la verifica della raggiungibilità temporale degli eventi è già disponibile un analizzatore dedicato [5] integrato nel *workbench*.

4.10 Analisi dei Risultati in modalità verbosa

In modalità *verbosa*, oltre al risultato finale delle proprietà di *Separate Liquidity* e *Liquidity*, l'analizzatore fornisce informazioni aggiuntive utili a comprendere come i risultati negativi siano stati ottenuti.

Ad esempio, analizzando il Contratto 3.3 `False_Negative`, si ottiene:

```
False_Negative
  - is NOT Separate Liquid.
    - is NOT h1-Separate Liquid.
       comp: Q1 now+1D Q2; Q2 now Q2;
    - is h2-Separate Liquid.
  - is NOT Liquid.
     {'h1'} in comp: Q1 now+1D Q2; Q2 now Q2;

  INFO:
    False_Negative contains transfers between distinct assets.
      - Separate Liquidity does not imply contract liquidity.
      - Please refer to the Liquidity result.

  WARNING:
    False_Negative has events.
      - h1-Separate Liquidity result could be a false negative.
      - h2-Separate Liquidity result could be a false positive.
      - Liquidity result could be a false negative.
======================================================================
VERBOSE INFORMATION
  Functions, Events:
    Q0 Matteo:init Q1
      LIQUIDITY TYPE:
        {'h1': xi_1, 'h2': xi_2, 'u': 1} ->
                    {'h1': xi_1, 'h2': 1, 'u': 0}
      ASSET TYPES:
        (h1); (h2, u)
      EVENTS LIST:
        [Q1 now+1D Q2, Q2 now Q2]
    [ ... ]
  Abstract Computations:
    Q0 Matteo:init Q1; Q1 now+1D Q2; Q2 Matteo:end Q3;
      LIQUIDITY TYPE:
```

```
34          {'h1': xi_1, 'h2': xi_2} -> {'h1': xi_1, 'h2': 0}
35        ASSET TYPES:
36          (h1); (h2)
37        ARE ALL ASSET TYPES SINGLETON:
38          True
39      [ ... ]
```

Viene quindi mostrata la porzione di computazione che ha prodotto un esito negativo. Nel caso della proprietà di *Liquidity*, viene inoltre mostrato l'insieme di asset che risultavano bloccati.

Oltre alle sezioni `INFO` e `WARNING`, descritte in precedenza, viene mostrata una sezione aggiuntiva denominata `VERBOSE INFORMATION`. Questa è suddivisa in due parti principali: la descrizione delle *transizioni* del contratto e l'elenco delle *computazioni astratte* prodotte durante l'analisi.

Functions, Events Per ogni configurazione del contratto viene mostrato il relativo *Liquidity Type*, che descrive come la funzione o l'evento trasforma le espressioni di liquidità associate ai vari asset. In particolare, il tipo di liquidità è rappresentato come una trasformazione tra lo stato iniziale e lo stato finale degli asset.

Ad esempio, per la funzione `Q0 Matteo:init Q1` abbiamo:

$$\{\text{'h1': } \xi_1, \text{ 'h2': } \xi_2, \text{ 'u': } 1\} \rightarrow \{\text{'h1': } \xi_1, \text{ 'h2': } 1, \text{ 'u': } 0\}$$

da cui assumiamo che consideri gli asset globali `h1`, `h2` e il parametro asset `u`. Al suo interno avviene il trasferimento dei fondi dell'asset `u` nell'asset `h2`; di conseguenza, al termine della funzione `u` risulta svuotato, mentre `h2` assume il valore massimo tra il suo valore iniziale (ξ_2) e il valore di `u` ($\mathbb{1}$), cioè $\mathbb{1}$.

Per ciascuna configurazione vengono inoltre mostrati gli *asset types*, ovvero i cluster di asset che risultano correlati a seguito dell'esecuzione della funzione. Se la configurazione è una funzione, viene anche riportata la lista degli eventuali eventi dichiarati al suo interno.

Abstract Computations La seconda parte della sezione riporta la descrizione di tutte le *computazioni astratte* generate dall'analizzatore a partire dal contratto. Ogni computazione è rappresentata come una sequenza di configurazioni (funzioni o eventi) che descrive un possibile percorso di esecuzione del contratto.

Per ciascuna computazione vengono mostrate la sequenza di configurazioni che la compongono, il *Liquidity Type* risultante dalla composizione delle configurazioni e i

cluster di asset ottenuti al termine della computazione. Inoltre viene riportata l'informazione `ARE ALL ASSET TYPES SINGLETON`, che indica se tutti gli asset risultano indipendenti tra loro, cioè se tutti i cluster corrispondono a singoletti di asset.

Capitolo 5

Conclusioni

In questo lavoro abbiamo affrontato il problema dell'analisi della liquidità nei contratti legali scritti in `Stipula`. In tale contesto, chiamiamo *asset* una risorsa patrimoniale gestita dal contratto. L'obiettivo era individuare staticamente i contratti in cui il valore contenuto negli asset potesse rimanere permanentemente bloccato. Dopo aver introdotto il linguaggio, la sua sintassi e la sua semantica operazionale, abbiamo formalizzato due proprietà rilevanti per l'analisi: la *k-Separate Liquidity*, che considera la possibilità di svuotare un singolo asset, e la *Liquidity*, che richiede invece di poter raggiungere uno stato in cui tutti gli asset risultino simultaneamente vuoti.

Su questa base abbiamo presentato due algoritmi per la verifica automatica di tali proprietà e ne abbiamo realizzato un'implementazione concreta in Python sotto forma di analizzatore. L'analizzatore consente di esaminare contratti scritti in `Stipula`, analizzarne l'effetto delle istruzioni, verificarne le proprietà di liquidità e fornire indicazioni utili per interpretare il risultato ottenuto.

L'analisi sviluppata mostra che la nozione intuitiva di assenza di fondi bloccati non coincide sempre con la proprietà più forte di *Liquidity*: possono infatti esistere contratti in cui il valore continua a circolare correttamente senza che tutti gli asset risultino mai contemporaneamente vuoti.

Allo stesso tempo, il lavoro mette in evidenza alcuni limiti dell'analisi statica di un contratto `Stipula`. In particolare, la presenza di clausole non concretamente raggiungibili durante l'esecuzione del contratto può portare a considerare computazioni solo apparentemente ammissibili, con la conseguenza di produrre possibili falsi positivi o falsi negativi. Inoltre, per garantire la terminazione dell'analisi, è necessario imporre un limite al numero di occorrenze delle funzioni coinvolte in comportamenti

ciclici. Questa scelta rende l'analisi dipendente da un parametro fissato a priori: un valore troppo basso potrebbe infatti escludere computazioni rilevanti, mentre un valore più alto consente un'esplorazione più ampia ma aumenta il costo dell'analisi.

Come sviluppo futuro, sarebbe interessante integrare nello stesso analizzatore l'analisi della liquidità con strumenti, già esistenti, per la verifica della raggiungibilità delle clausole, così da ridurre l'impatto di tali limitazioni. Si potrebbe inoltre raffinare l'analisi nei casi in cui il valore circoli soltanto all'interno di particolari insiemi di asset, detti cluster.

Nel complesso, questa tesi mostra che la presenza della liquidità nei contratti legali può essere verificata in modo automatico grazie alla loro rappresentazione come programmi, combinando definizioni teoriche e strumenti concreti di verifica.

Bibliografia

[1] Massimo Bartoletti and Roberto Zunino. Verifying liquidity of Bitcoin contracts. In *Proceedings of the 8th International Conference on Principles of Security and Trust (POST)*. Springer, 2019.

[2] Stefano Crafa, Cosimo Laneve, Giovanni Sartor, and Andrea Veschetti. Pacta sunt servanda: Legal contracts in Stipula. *Science of Computer Programming*, 2023.

[3] Erik Dervishi, Cosimo Laneve, and Matteo Panicciari. Stipula workbench. `https://github.com/stipula-language/stipula-workbench`, 2026.

[4] Simone Evangelisti. *Analisi di contratti legali in Stipula*. PhD thesis, Università di Bologna, 2024.

[5] Simone Evangelisti and Cosimo Laneve. Stipula analyzer. `https://github.com/stipula-language/stipula-analyzer`, 2024.

[6] Free Software Foundation. Gnu general public license, version 3. `https://www.gnu.org/licenses/gpl-3.0.html`, 2007.

[7] Cosimo Laneve. Liquidity analysis in resource-aware programming. *THE JOURNAL OF LOGICAL AND ALGEBRAIC METHODS IN PROGRAMMING*, 2023.

[8] Matteo Panicciari. Stipula liquidity analyzer. `https://github.com/MatteoPanicciari/StipulaLiquidityAnalyzer`, 2026.

[9] Parity Technologies. Parity wallet security alert. `https://paritytech.io/blog/security-alert.html`, 2017.

[10] Terence Parr. Antlr. `https://www.antlr.org/`.

[11] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[12] Terence Parr, Sam Harwell, and Eric Vergnaud. Antlr 4.13.2 runtime for python 3. `https://pypi.org/project/antlr4-python3-runtime/`.

[13] Python Software Foundation. Python. `https://www.python.org/`.

[14] Research Group on EC Private Law (Acquis Group) Study Group on a European Civil Code. *Principles, Definitions and Model Rules of European Private Law: Draft Common Frame of Reference (DCFR), Outline Edition.* Sellier, 2009.

[15] Armin Ronacher. Click. `https://pypi.org/project/click/`.