Programming legal contracts
– a beginners guide to *Stipula* –

Silvia Crafa[1] and Cosimo Laneve[2]

[1] University of Padova
[2] University of Bologna – Inria FOCUS

Abstract. We discuss the design principles of *Stipula*, a domain specific language that assists lawyers in programming legal contracts through specific patterns. *Stipula* is based on a small set of primitives that are amenable to be prototyped on either centralized or distributed systems. We also outline two techniques that have been defined for *Stipula* contracts: a type inference system that allows to derive types for fields, assets and contract's functions, and an analyzer of liquidity that pinpoints those contracts that do not freeze any asset forever.

1 Introduction

The legal field is one of the domains that are currently most influenced by the so-called digital revolution. A large number of legal texts, ranging from laws, regulations, administrative procedures, to contractual agreements, court judgements and jurisprudence, might considerably benefit from a sensible digitalisation. The advantages are not only in terms of efficiency, like speed-up and automatic execution of fully defined procedures, but also in terms of data organisation and transparency of processes. As a cons, computationally dealing with laws is difficult because of the complexity of the legal texts: human judgement is often required to interpret the natural language since it is, at the same time, very expressive and quite ambiguous.

In this article we focus on *legal contracts*, a specific subset of the legal field that define "those agreements which are intended to give rise to a binding legal relationship or to have some other legal effect" [11]. These agreements basically are *protocols* that regulate the *relationships* between parties in terms of permissions, obligations, prohibitions, escrows and securities. In turn, according to the principle of *freedom of form*, which is shared by the contractual law of modern legal systems, the agreements can be expressed by the parties using the language and medium they prefer, including a programming language. When a programming language is chosen, it is mandatory that the language be high level enough so that

writing and inspecting a software contract do not require proficiency in computer science. In fact, only if the parties (which are usually lawyers, notaries, ordinary citizens, etc.) are fully aware of the computational effects of their code there may be a genuine agreement over the content of the contract, thus reducing or possibly eliminating applications to courts for either misinterpretations or misunderstandings.

In [6] we have defined a new domain-specific language, called *Stipula*, that uses few selected, concise and intelligible (to unskilled users) primitives that have a precise correspondence with the distinctive elements of legal contracts. The language is equipped with a formal operational semantics, so that the behaviour is fully specified and amenable to automatic verification. Overall, the basic motivation underneath the design of *Stipula* is that a contractual agreement actually defines an *interaction protocol* between parties. As a consequence, the underlying theory of the language is influenced by principles of concurrent systems. Interestingly, the contract law imposes an intrinsic *openness* to contractual interactions, since triggering conditions may depend on contextual situations and third party authorities may dynamically (and timely) add further requirements to deal with litigations.

In this paper we give a gentle introduction to *Stipula*, by motivating its distinctive features with contractual elements taken from two paradigmatic legal contracts: a rental contract and a bet contract. We then discuss two formal analysis techniques that have been defined for *Stipula* and that can bee seen as useful tools that support a safe programming of legal contracts. First, since *Stipula* is untyped, we illustrate a type inference system that allows to automatically derive types for fields, assets and contracts function. This system is useful to type check the correctness of operations, thus preventing basic errors with contract's data and assets. Then we illustrate an analyzer of liquidity that statically checks the presence of executions of the legal contract leaving assets frozen into the contract without being redeemable by any party. We conclude the paper with a number of final remarks about the implementation of *Stipula* and a discussion of related work.

2 Legal contracts' elements as *Stipula* building blocks

Contractual agreements are generally written as combinations of distinctive elements, such as permissions, prohibitions, obligations, fungible and non fungible assets exchanges, and aleatory or real-world data retrieval. These elements are combined into common legal patterns that either es-

tablish new obligations, rights, powers and liabilities between the parties, or transfer rights (such as rights to property) from one party to another, often subject to specific conditions and by taking advantage of escrows and securities.

A first distinctive feature of a legal contract is the "*meeting of the minds*", *i.e.* the moment when, after the possible negotiation of the contractual content, the parties express consent on the terms of the agreement and the contract produces its legal effects. *Stipula* provides an ad-hoc primitive, called *agreement*, which marks that contracts' parties have reached a consensus on the contractual arrangement they want to create. As an example, consider a contract regulating a bike rental service: the following *Stipula* code

```
agreement (Lender, Borrower)(rentingTime, cost){
   _
   } ⇒ @Inactive
```

is meeting a `Lender` and a `Borrower` to agree on both the `rentingTime` and on its `cost`. After the agreement the contract starts and it goes into a state `@Inactive` that expresses that no rent will occur until the payment (some money has been transferred from `Borrower` to `Lender`). The "`_`" in the body of the agreement is an abbreviation when all the parties (that are written inside the first curved brackets) agree on all the values (that are written inside the second curved brackets). This is not always the case. To illustrate the verbose alternative, consider a variant of the contract also including an `Authority` that is charged to monitor contextual constraints, such as obligations of diligent storage and care, or the obligations of using goods only as intended, taking care of litigations and dispute resolution. In this case, the agreement would be

```
agreement (Lender, Borrower, Authority)(rentingTime, cost){
   Lender, Borrower: rentingTime, cost
   } ⇒ @Inactive
```

expressing the fact that only the `Lender` and the `Borrower` agree on both `rentingTime` and `cost`, while the `Authority`, which also engage in the meeting of minds, is the pointer to a *third party* that will supervise `Lender` and `Borrower` behaviours.

A second distinctive feature of legal contracts is that the set of normative elements, namely permissions, prohibitions and obligations, usually changes over time according to the actions that have been done (or not). To model these changes, *Stipula* commits to a *state-machine programming* style, inspired by the state machine pattern that is supported by almost every programming language (with ad-hoc libraries and/or modules). For

instance, in a bike rental, once the lender and the borrower have agreed on the rental period and cost, the lender is prohibited from preventing the borrower from paying for the service and (afterwards) using the bike. *Stipula* expresses this feature by letting the contract play a proactive role: assuming that the bike can be used if the borrower has a temporary access code, the contract stores the temporary code in a field, thus disallowing the lender to withdraw from the rental. The following code defines the *function* `offer` that can be invoked by `Lender` when the contract is in state `@Inactive` to send an access code to be used by the `Borrower`.

```
@Inactive Lender : offer(x) {
   x → code
   } ⇒ @Payment
```

Of course, the code is not disclosed to the `Borrower` before the payment for the service. In other terms, the above fragment is giving *permission* to the `Lender` to invoke `offer` in the state `@Inactive` and, if no further function is defined in `@Inactive`, the contract is *prohibiting* other parties to do any action at this stage. Once `code` has been received, the contract moves to a state `@Payment` where presumably the `Borrower` will pay (actually, he is allowed to pay) for the rental.

It is worth to notice that the foregoing code also asserts that `Lender` is trusting the contract to act as intermediary that can store sensible informations and, as we will see below, assets. In fact, `x → code` stores the value sent by the `Lender` into a contract's field called `code` that cannot be accessed outside the contract.

A further distinctive feature of legal contracts is the management of *assets*: currency is required for payments and escrows, tokens, both fungible and non-fungible, are useful to model securities and to provide a digital handle on a physical good. For instance, in the case of the bike rental, instead of a simple numeric `code`, a more innovatory IoT technique would be to rely on a unique token that grants access to the bike's smart lock. Moreover, in the traditional setting, the `Borrower` pays the `Lender` with a credit card before he can use the bike and the money transaction is only specified by the contract through a normative clause. Its occurrence is not guaranteed (in case of dispute, one party has to appeal to a court). On the other hand, *Stipula* admits digital legal contracts that automatically deal with assets transfers, so to remove intermediation even from the payments. In *Stipula* assets can be also temporarily retained by legal contracts, which may decide to redistribute them when particular conditions occur. To this aim, the language promotes an explicit, and thus

conscious, management of assets by regarding them as first-class values with ad-hoc operations. For example, the function

```
@Payment Borrower : pay[h]
   (h == cost) {
      h ⊸ wallet
      code → Borrower
   } ⇒ @Using
```

is defining the payment of the rental by `Borrower`, which sends an asset `h` – the argument is in square brackets – to the contract. The function call has a precondition – operation `h == cost` – that checks whether the borrower pays the correct fee or not. The semantics of the operation `h ⊸ wallet`, which is an abbreviation for `h ⊸ h, wallet`, is that, after the execution, `h` is not owned by `Borrower` anymore and is taken by the contract that stores it in the *asset field* `wallet`. The design choice of explicitly marking asset movements with the ad hoc operator "⊸" (thus separating it from "→") promotes a safer, asset-aware, programming discipline that reduces the risk of the so-called double spending, the accidental loss or the locked-in assets. Notice that the contract does not immediately forward the payment to the `Lender`, rather it is retained for some time until the rental period is terminated (in this way, in case of disputes, neither the `Borrower` nor the `Lender` can access/use the asset while the dispute is in progress). Once the fee has been payed, the `Borrower` gets the access code to the bike and the contract transits into a `@Using` state.

There is a fourth distinctive feature of legal contracts: the *obligations*, namely operations that must be done, typically within a deadline, by some party. In *Stipula*, obligations are recast into commitments that are checked at a specific time limit and the corresponding programming abstraction is the *event* primitive. For example, the foregoing `pay` function may be refined by issuing an event that terminates the renting service when the time limit is reached. The code becomes

```
@Payment Borrower : pay[v]
   (v == cost) {
      v ⊸ wallet
      code → Borrower
      now + rentingTime ≫              //end-of-time usage
         @Using {
            "End_Reached" → Borrower
            wallet ⊸ Lender
         } ⇒ @End
   } ⇒ @Using
```

asserting that the bike can be used until the renting period terminates. The time limit is expressed by `now + rentingTime` and, at that moment,

if the bike has not been already returned (the state of the contract is still `@Using`), a message of returning the bike is sent to the `Borrower` (`"End_Reached" → Borrower`) and the fee that was stored in `wallet` is delivered to the `Lender` (`wallet ⊸ Lender`). We remark that events are not triggered by any party: they are automatically executed when the time condition is met. Since the statements in the body of events will be executed in the future, we assume for simplicity that the event's body is outside of the scope of functions's parameters, both assets and non assets. A more complex alternative would be to save for future execution the *closure* of the event statements, that captures the local values of functions' parameters.

The foregoing codes do not address disputes, *e.g.* contentions because the bike is returned, or initially was, broken or damaged. These are common elements of legal contracts, that are usually assessed by means of *third party enforcements*, typically by a court. Disputes have a simple modelling in *Stipula* that does not require any new ad-hoc feature, and somehow mimic the behaviour of a court. In fact, when contract's violations cannot be fully checked by the software, such as the damage or misuse of the bike, or the renting of a broken bike, then it is necessary the presence of a trusted third party, the `Authority`, to supervise the dispute and to provide a trusted resolution mechanism. The code below illustrates the encoding of the off-chain monitoring and enforcement mechanism by means of an `Authority` (which must have been included in the agreement) in *Stipula*.

```
@Using Lender,Borrower : dispute(x) {
   x → _
   } ⇒ @Dispute

@Dispute Authority : verdict(x,y)
   (y >= 0 && y <= 1) {
      x → Lender, Borrower
      y×wallet ⊸ wallet, Lender
      wallet ⊸ Borrower
   } ⇒ @End
```

The function `dispute` may be invoked either by the `Lender` or by the `Borrower` and carries the reasons for kicking the dispute off (`x` is intended to be a string). Once the reasons are communicated to every party (we use the abbreviation "`_`" instead of writing three times the sending operation) the contract transits into a state `@Dispute` where the `Authority` will analyze the issue and emit a verdict. This is performed by permitting in the state `@Dispute` only the invocation of the `verdict` function, that has

legal contracts	***Stipula* contracts**
meeting of the minds	agreement primitive
permissions, prohibitions	state-aware programming
currency and tokens	asset-aware (linear) programming
obligations	event primitive
judicial enforcement	explicit Authority and ad-hoc pattern
exceptional behaviors	explicit Authority and ad-hoc pattern

Fig. 1. Correspondence between legal elements and *Stipula* features

two arguments: a string of motivations `x`, and a coefficient `y` that denotes the part of the wallet that will be delivered to `Lender` as reimbursement; the `Borrower` will get the remaining part. It is worth to spot this point: the statement `y`×`wallet` ⊸ `wallet, Lender` *takes* the `y` part of `wallet` (`y` is in [0..1]) and sends it to `Lender`; *at the same time* the `wallet` is reduced correspondingly. The remaining part is sent to `Borrower` with the statement `wallet` ⊸ `Borrower` (which is actually a shortening for `1` × `wallet` ⊸ `wallet, Borrower`) and the `wallet` is emptied.

There is a last distinctive element in legal contracts that deserves a comment: the management of *exceptional behaviours*, *i.e.* all those behaviours that cannot be anticipated due to the occurrence of unforeseeable and extraordinary events. As in the above case, *Stipula* does not use any new ad-hoc feature, rather a simple pattern is provided that defines a template:

```
~@End _ : block(x) {
   x → _
   } ⇒ @Exception

@Exception Authority : handle(x,y) //similar to verdict(x,y)
```

According to the above pattern, the function `block` may be invoked by any party (notation "`_`") provided the lifetime of the contract is *not terminated* (the contract is not in the state `End`). The management of the exception is similar to that of disputes and therefore omitted.

Figure 1 recaps the normative elements of a legal contract and the corresponding modellings in *Stipula*. Figure 2 uses coloured boxes to highlight the correspondence between the normative elements of a standard bike rental contract and the corresponding editing in *Stipula*. In this case the *Stipula* code is a bit more complex than the one discussed above: `Borrower`

pays the double of the fee in order to safeguard `Lender` from damages, late returns, etc. Accordingly, the termination of the rental requires the `Borrower` to call the function `end`, after which the `Lender` has to confirm the absence of damages by invoking `rentalOK`. Only this sequence of actions, which is enforced by the additional state `@Return`, allows the lender to be payed and the borrower to get back the money deposited as security.

It is worth to notice that a *Stipula* contract begins with the keyword `stipula` and define *assets* and *fields* that are used therein. We also observe that the code is untyped because types seem harsh to lawyers. However, a type inference system that allows one to derive types is discussed in Section 4. Finally, the code of Figure 2 is also *liquid*: at the end of any contract execution, in the final state `@End`, the asset `wallet` is empty, *i.e.* `Bike_Rental` has no locked-in value (see the discussion in Section 5).

3 Example: the bet contract

An example for testing the expressivity of *Stipula* is a contract ruling a bet. This is a legal contract that contains an element of randomness (*alea*, such as a future, aleatory event, such as the winner of a football match, the delay of a flight, the future value of a company's stock) that is entirely independent of the will of the parties.

A digital encoding of a bet contract requires that the parties explicitly agree on the source of data that will determine the final value of the aleatory event – the `DataProvider` –, which is usually a specific online service, an accredited institution, or any trusted third party. It is also important that the digital contract defines precise time limits for accepting payments and for providing the actual value of the aleatory event. Indeed there can be a number of issues: the aleatory event does not happen, *e.g.* the football match has been cancelled, or the data provider fails to deliver the required value, *e.g.* the online service is down.

The *Stipula* code in Listing 1.1 corresponds to the case where `Better1` and `Better2` respectively place in `val1` and `val2` their bets, while the agreed `amount` of currency is stored in the contract's assets `wallet1` and `wallet2` [3]. Observe that both bets must be placed within an (agreed) time limit `t_before` (line 15), to ensure that the legal bond is established before the occurrence of the aleatory event. The second timeout, scheduled in line 22, is used to ensure the contract termination even if the `DataProvider`

[3] For simplicity, this code requires `Better1` to place its bet before `Better2`. It is easy to extend the code to let the two bets be placed in any order.

Fig. 2. A standard Bike Rental contract and its modelling in *Stipula*

```
stipula Bet {
  assets wallet1, wallet2
  fields val1, val2, source, alea, amount, t_before, t_after

  agreement (Better1, Better2, DataProvider)
                (source, alea, amount, t_before, t_after){
     DataProvider, Better1, Better2 : source, alea, t_after
     Better1, Better2 : amount, t_before
  } ⇒ @Init

  @Init Better1 : place_bet(x)[v]
    (v == amount){
        v ⊸ wallet1
        x → val1
        t_before ≫ @First { wallet1 ⊸ Better1 } ⇒ @Fail
    } ⇒ @First

  @First Better2: place_bet(x)[v]
    (v == amount){
        v ⊸ wallet2
        x → val2
        t_after ≫ @Run {
            wallet1 ⊸ Better1
            wallet2 ⊸ Better2 } ⇒ @Fail
  } ⇒ @Run

  @Run DataProvider : data(x,y)[]
    (x==alea){
        if (y==val1 && y==val2){        // Better1 and Better2 win
            wallet1 ⊸ Better1
            wallet2 ⊸ Better2
        } else if (y==val1 && y!=val2){ // The winner is Better1
            wallet2 ⊸ Better1
            wallet1 ⊸ Better1
        } else if (y!=val1 && y==val2){ // The winner is Better2
            wallet1 ⊸ Better2
            wallet2 ⊸ Better2
        } else {                        // No winner
            wallet1 ⊸ DataProvider
            wallet2 ⊸ DataProvider
        }
   } ⇒ @End
}
```

Listing 1.1. The contract for a bet

fails to provide the expected data, through the call of the function `data`. When the function `data` is called and the first argument `x` is the `alea` of the bet, the betters are rewarded according to the result `y`. For simplicity we assume that the data-provider service gets the two bets when they loose.

Compared to the Bike Rental in Section 2, the role of the `DataProvider` here is less pivotal than that of the `Authority`. While it is expected that `Authority` will play its part, `DataProvider` is much less than a peer of the contract. It is sufficient that it is an independent party that is entitled to call the contract's function to supply the expected external data that will extract from `source`. In case `DataProvider` behaves incorrectly, *e.g.* it supplies an incorrect value through the function `data`, the betters can appeal against the data provider since they agreed upon the data emitted by the `source`. As usual, any dispute that might render the contract voidable or invalid, *e.g.* one better knew the result of the match in advance, can be handled by including an `Authority`, according to the pattern illustrated in the Bike Rental example.

4 Type inference in *Stipula*

Stipula is type-free: types have been drop in order to ease the understanding of the language and of the codes, in particular to lawyers. This does not mean that types are neglected. Indeed, the language comes with a type inference system that allows one to derive types of assets, fields and functions' arguments. In this section we discuss the main design principles of the system.

Stipula has the following *primitive types* `T`

- `real`, representing real numbers, written as nonempty sequences of digits, possibly followed by "`.`" and by a sequence of digits (*e.g.* `13` stands for `13.0`);. The number may be prefixed by the sign `+` or `-`.
- `bool` representing the boolean values `false` and `true`.
- `string`, representing strings: sequences of characters that are pre- and post-fixed by "`"`".
- `asset` representing assets that can be accumulated, such as cryptocurrencies.

We remark that an entity of type `asset` can be safely cast to the numeric type `real`, so that, *e.g.*, the boolean expression `h == 5` is well typed when `h` is of `asset` type. We use the following notation

- *type terms* α, α', $\cdots$, which are either *type variables* X, Y, Z, $\cdots$, or primitive types;
- *environments* Γ that maps fields and non-asset functions' arguments to type variables, and Δ maps assets and assets functions' arguments to type variables. They are both injective functions, and the notation $\Gamma[\mathtt{x} \mapsto X]$, resp. $\Delta[\mathtt{v} \mapsto V]$, stands for either the update or the extension of the environment, depending on whether $\mathtt{x}$, resp. $\mathtt{v}$, belongs to the domain of the environment.
- *constraints* Υ, Υ', $\cdots$, which are conjunctions of equations $\alpha = \alpha'$;
- *judgments* $\Gamma, \Delta \vdash E : \alpha, \Upsilon$ for expressions E and $\Gamma, \Delta \vdash S : \Upsilon$ for statements S.

The inference system of *Stipula* is almost standard: it associates pairwise different type variables to the names of a program and parses the code by collecting constraints. At the end of the parsing process, the constraints are solved by means of a unification technique and the type variables are replaced by the resulting values (see [9] for details of the technique).

The complete type inference system of *Stipula* is reported in Appendix A, we just discuss here the most relevant rules. The simplest rule of the inference system is the typing of a value κ:

$$\frac{\kappa \in \mathtt{T}}{\Gamma, \Delta \vdash \kappa : \mathtt{T}, \mathit{true}}$$

That is, assuming that κ belongs to $\mathtt{T}$ we derive that κ has type $\mathtt{T}$ without any constraint (the term *true*) in *every* environments Γ and Δ. A simple rule that generates constraints is the assignment of a value to a field:

$$\frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Upsilon' = (\Gamma(\mathtt{x}) = \alpha) \wedge \Upsilon}{\Gamma, \Delta \vdash E \to \mathtt{x} : \Upsilon'}$$

As usual, statements $E \to \mathtt{x}$ have no type: the typing system returns a constraint imposing that the typing of E is equal to the type of $\mathtt{x}$, *i.e.,* $\Gamma(\mathtt{x}) = \alpha$. For example, the typing of $\mathtt{"hello"} \to \mathtt{x}$ in the environments Γ, Δ returns the constraint $\Gamma(\mathtt{x}) = \mathtt{string}$.

The following rule is the typing of the asset withdraw:

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \mathtt{h}, \mathtt{h}' \in \mathit{dom}(\Delta) \\ \Upsilon' = (\alpha = \mathtt{real}) \wedge (\Delta(\mathtt{h}) = \Delta(\mathtt{h}')) \wedge \Upsilon\end{array}}{\Gamma, \Delta \vdash E \multimap \mathtt{h}, \mathtt{h}' : \Upsilon'}$$

The rule defines the type of the *withdraw* of the value of E from the asset $\mathtt{h}$ and the corresponding *addition* to $\mathtt{h}'$. According to the semantics of *Stipula*, this operation is not performed if $\mathtt{h}$ does not own enough assets. The rule constraints the assets $\mathtt{h}$ and $\mathtt{h}'$ to have the same type.

Finally, the typing of a *Stipula* contract is given in the following rule, where $\vdash \mathtt{G}$ stands for the syntactic check that the agreement $\mathtt{G}$ is well formed, and $\Upsilon \Vdash \sigma$ means that the type variable substitution σ satisfies the constraints Υ:

$$\frac{\overline{X}, \overline{Z} \textit{ fresh} \qquad \vdash \mathtt{G} \qquad \Big([\overline{\mathtt{x}} \mapsto \overline{X}], [\overline{\mathtt{h}} \mapsto \overline{Z}] \vdash \mathtt{F}_i : \mathtt{A_i}.\mathtt{f_i}(\overline{Y_i})[\overline{V_i}], \Upsilon_i\Big)^{i \in 1..n} \quad \bigwedge_{i \in 1..n} \Upsilon_i \Vdash \sigma}{\vdash \ \mathtt{stipula\ C\ \{\ assets\ \overline{h}\ \ fields\ \overline{x}\ \ G\ \ f_1 \cdots f_n\ \}} \ : \ [\,\overline{\mathtt{h}} : \sigma(\overline{Z}),\ \overline{\mathtt{x}} : \sigma(\overline{X}),\ \sigma\Big(\mathtt{A_i}.\mathtt{f_i}(\overline{Y_i})[\overline{V_i}]\Big)^{i \in 1..n}\,]}$$

(we assume that the pair $\mathtt{A_i}.\mathtt{f_i}$ identifies exactly one function). Whenever the inference system is not able to derive a ground type, that is $\sigma(X)$ is a type variable, it means that there are no type constraints. In particular, as regards assets, the type system only collects assets type identities and he type variable can be safely instantiated to the ground type `asset`. For instance, in the case of the Bike Rental contract, the inference system identifies the types of `wallet` with the formal parameter of the function `pay` and the user is asked to instantiate it. As regards the Bet contract, the types of `wallet1` and `wallet2` are not identified (they may be different) and, as before, the types of the asset parameter of `Better1.place_bet` and of `wallet1` are identified; similarly for `Better2.place_bet` and of `wallet2`.

5 An analyzer of liquidity

Liquidity is a sensible security property of every program managing assets: a contract is *liquid* when no assets remain frozen forever inside it, *i.e.* it is not redeemable by any party [2]. The liquidity analyzer of *Stipula* has four phases. Below we detail every phase and the techniques we use.

In the first phase, the analyzer extracts the underlying finite state automaton of a contract. The states of the contract are those written at the end of the agreement – this is the *initial state* – and those written at the beginning and at the end of functions and of events. The transitions correspond to functions and events (the correspondence is one-to-one) and are labelled by the function name and the event (event + codeline

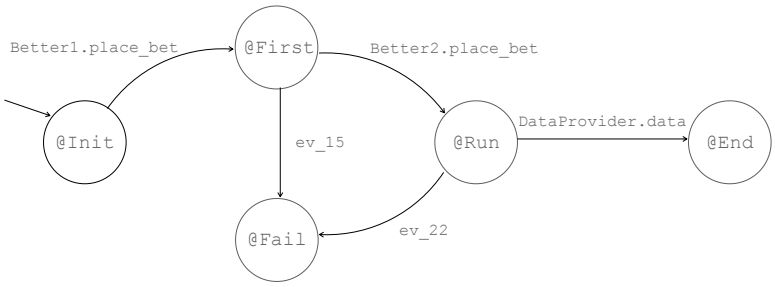

Fig. 3. The finite-state automata of the Bet contract

of the initial instruction) respectively. Figure 3 reports the finite state automata of the Bet contract in Section 3.

The second phase is, technically, the most complex one. For every transition of the automata, it returns an over-approximation of the balances of the assets that expresses whether an asset is empty – notation $\mathsf{0}$ – or not empty – notation ∞. The values $\mathsf{0}$ and ∞ are called *liquidity values*; they are ordered as $\mathsf{0} < \infty$, and the operations $\sqcup$, resp. $\sqcap$, return the maximum, resp. minumum, value of two liquidity values. We use

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

$$\mathbb{e} ::= \mathsf{0} \mid \infty \mid \xi \mid \mathbb{e} \sqcup \mathbb{e} \mid \mathbb{e} \sqcap \mathbb{e}$$

– *environments* Φ, which map fields and non-asset parameters into liquidity expressions, and Ξ, which map contract's assets and asset parameters to liquidity expressions;
– *judgments* $\Phi, \Xi \vdash E : \mathbb{e}$ for expressions, $\Phi, \Xi \vdash S : \Phi', \Xi'$ for statements and $\Phi, \Xi \vdash \texttt{@Q A :f}(\overline{\texttt{x}})[\overline{\texttt{h}'}]\ (E)\{\, S\, W \,\} \Rightarrow \texttt{@Q}' : \mathcal{L}$ for function definitions, where $\mathcal{L}$ is a set of labels (see below).

We discuss few critical rules. As regards expressions, every constant has liquidity value ∞ but for the constant 0. Since any other expression involves an operation, the analysis computes whether the result of this operation is $\mathsf{0}$ or ∞. In particular, for the arithmetic operation $\times$, the analysis returns the lower-bound of the arguments (because when one argument is 0, the liquidity analysis returns $\mathsf{0}$) while it returns the upper-bound for any other operation:

$$\Phi, \Xi \vdash 0 : \mathtt{0} \qquad \frac{\kappa \neq 0}{\Phi, \Xi \vdash \kappa : \infty}$$

$$\frac{\Phi, \Xi \vdash E : \mathbb{e} \quad \Phi, \Xi \vdash E' : \mathbb{e}'}{\Phi, \Xi \vdash E \ \times \ E' : \mathbb{e} \sqcap \mathbb{e}'} \qquad \frac{\Phi, \Xi \vdash E : \mathbb{e} \quad \Phi, \Xi \vdash E' : \mathbb{e}' \quad \mathtt{op} \neq \times}{\Phi, \Xi \vdash E \, \mathtt{op} \, E' : \mathbb{e} \sqcup \mathbb{e}'}$$

As regard statements, we have two rules for asset movements:

$$\frac{\mathbb{e} = \Xi(\mathtt{h}) \sqcup \Xi(\mathtt{h}')}{\Phi, \Xi \vdash \mathtt{h} \multimap \mathtt{h}' : \Phi, \Xi[\mathtt{h} \mapsto \mathtt{0}, \mathtt{h}' \mapsto \mathbb{e}]} \qquad \frac{\begin{array}{c} E \neq \mathtt{h} \quad \Phi, \Xi \vdash E : \mathbb{e} \\ \mathbb{e}' = (\mathbb{e} \sqcap \Xi(\mathtt{h})) \sqcup \Xi(\mathtt{h}') \end{array}}{\Phi, \Xi \vdash E \multimap \mathtt{h}, \mathtt{h}' : \Phi, \Xi[\mathtt{h}' \mapsto \mathbb{e}']}$$

According to the rule on the left, the final asset environment of $\mathtt{h} \multimap \mathtt{h}'$ (which is an abbreviation for $\mathtt{h} \multimap \mathtt{h}, \mathtt{h}'$) has $\mathtt{h}$ that is emptied and $\mathtt{h}'$ that gathers the value of $\mathtt{h}$, henceforth the liquidity expression $\Xi(\mathtt{h}) \sqcup \Xi(\mathtt{h}')$. Notice that, when both $\mathtt{h}$ and $\mathtt{h}'$ are $\mathtt{0}$, the overall result is $\mathtt{0}$. In the rule on the right, the asset $\mathtt{h}$ is decreased by an amount that is moved to $\mathtt{h}'$. Since E is not $\mathtt{h}$, the static analysis (which is independent of the runtime value of E) can only safely assume that the asset $\mathtt{h}$ is not emptied by this operation (if it was not empty before). Therefore, after the withdraw, the liquidity value of $\mathtt{h}$ has not changed. On the other hand, the asset $\mathtt{h}'$ is increased of some amount if both E and $\mathtt{h}$ have a non zero liquidity value, henceforth the expression $(\mathbb{e} \sqcap \Xi(\mathtt{h})) \sqcup \Xi(\mathtt{h}')$. In particular, when both $\Xi(\mathtt{h})$ and $\Xi(\mathtt{h}')$ are $\mathtt{0}$, the overall result is $\mathtt{0}$.

The rule for conditionals is

$$\frac{\Phi, \Xi \vdash S : \Phi', \Xi' \qquad \Phi, \Xi \vdash S' : \Phi'', \Xi''}{\Phi, \Xi \vdash \mathtt{if} \ (E) \ \{ \ S \ \} \ \mathtt{else} \ \{ \ S' \ \} : \Phi' \sqcup \Phi'', \Xi' \sqcup \Xi''}$$

where the operation $\sqcup$ on environments is defined pointwise as follows:

$$(\Xi' \sqcup \Xi'')(\mathtt{h}) = \Xi'(\mathtt{h}) \sqcup \Xi''(\mathtt{h})$$

That is, the liquidity analyzer over-approximates the final environments of $\mathtt{if} \ (E) \ \{ \ S \ \} \ \mathtt{else} \ \{ \ S' \ \}$ by taking the maximum values between the results of parsing S (that corresponds to a true value of E) and those of S' (that corresponds to a false value of E). The expression E is overlooked by the analyzer (it is not difficult to refine the current analyzer by pre-computing `E` and, in case it is always true or false, avoid any maximum operation).

The rule for *Stipula* contracts collects the *liquidity labels* $\mathcal{L}_i$ that describe the liquidity effects of each contract's function; each function

(which can be potentially called in any order and any number of times) assumes injective environments that just associate contracts fields, resp. assets, with symbolic names:

$$\frac{\overline{\chi},\overline{\xi}\ \textit{fresh} \quad \left([\overline{\mathtt{x}}\mapsto\overline{\chi}],[\overline{\mathtt{h}}\mapsto\overline{\xi}]\vdash \mathtt{F}_i:\mathcal{L}_i\right)^{i\in 1..n}}{\vdash \mathtt{stipula\ C}\ \{\ \mathtt{assets}\ \overline{\mathtt{h}}\ \ \mathtt{fields}\ \overline{\mathtt{x}}\ \mathtt{G}\ \mathtt{F}_1\cdots\mathtt{F}_n\ \}:\mathcal{L}_1,\cdots,\mathcal{L}_n}$$

In turn, the rule for function definitions is:

$$\frac{\begin{array}{c} W=\left(E_i\gg @\mathtt{Q}_i\{\ S_i\ \}\Rightarrow @\mathtt{Q}'_i\right)^{i\in I} \\ \Phi[\overline{\mathtt{x}'}\mapsto\overline{\infty}],\Xi[\overline{\mathtt{h}'}\mapsto\overline{\infty}]\vdash S:\Phi',\Xi' \qquad \left(\Phi,\Xi\vdash S_i:\Phi'_i,\Xi'_i\right)^{i\in I}\end{array}}{\Phi,\Xi\vdash @\mathtt{Q}\ \mathtt{A}:\mathtt{f}(\overline{\mathtt{x}'})[\overline{\mathtt{h}'}](E)\{\ S\ W\}\Rightarrow @\mathtt{Q}':\ \begin{array}{c} @\mathtt{Q}\ \mathtt{A.f}\ @\mathtt{Q}':\Phi,\Xi\to\Phi',\Xi' \\ \left(@\mathtt{Q}_i\ \mathtt{ev\_}i\ @\mathtt{Q}'_i:\ \Phi,\Xi\to\Phi'_i,\Xi'_i\ \right)^{i\in I}\end{array}}$$

This rule produces a set $\mathcal{L}$ of *liquidity labels* associated to transitions of the finite state automaton. The main label is that of the function, namely $@\mathtt{Q}\ \mathtt{A.f}\ @\mathtt{Q}':\Phi,\Xi\to\Phi',\Xi'$, saying that the transition named $@\mathtt{Q}\ \mathtt{A.f}\ @\mathtt{Q}'$ has liquidity effects $\Phi,\Xi\to\Phi',\Xi'$. As explained above, Φ and Ξ just associate contract's fields, resp. assets, with symbolic names. The analysis of the function's body S additionally assumes that the function parameters $\overline{\mathtt{x}'}$ and $\overline{\mathtt{h}'}$ are bound to ∞, because they may be any value. The liquidity effects of S are recorded by the environments Φ' and Ξ'. In particular, if $\Xi'(\mathtt{h}')=\infty$, where $\mathtt{h}'$ is an asset parameter, *i.e.* $\mathtt{h}'\notin dom(\Xi)$, then the asset $\mathtt{h}'$ has not been emptied by S. Therefore the asset is frozen and the program is not liquid. Similarly for the events' body S_i, but in this case the initial environments are not extended with function parameters because the syntax of *Stipula* imposes that events are out of their scope. In the above rule, I is intended to be the set of (the initial) code lines of the events.

Once the liquidity labels of the automaton's transitions t (both function calls and events) have been computed, it is possible to begin the third phase, *i.e.* calculating the effects that a computation has on the assets' balances. A *computation* is a finite sequences of labelled transitions $\{t_i:\Phi,\Xi\to\Phi_i,\Xi_i\}^{i=1,\ldots,n}$. The *effects* of the ith transition are the environments $\Phi_i^{\mathit{eff}},\Xi_i^{\mathit{eff}}$ recursively defined as follows

$$\Phi_i^{\mathit{eff}}=\left[\left(\mathtt{x}\mapsto[\![\ \Phi_i(\mathtt{x})\ \{{}^{\Phi_{i-1}^{\mathit{eff}}(\mathtt{x})}/_{\Phi(\mathtt{x})}\ \}\ ]\!]\right)^{\mathtt{x}\in dom(\Phi_i)}\right]$$

$$\Xi_i^{\mathit{eff}}=\left[\left(\mathtt{h}\mapsto[\![\ \Xi_i(\mathtt{h})\ \{{}^{\Xi_{i-1}^{\mathit{eff}}(\mathtt{h})}/_{\Xi(\mathtt{h})}\}\ ]\!]\right)^{\mathtt{h}\in dom(\Xi_i)}\right]$$

where

– we let the effects of the *initial environments* be $\Phi_0^{\mathit{eff}}, \Xi_0^{\mathit{eff}} = [\ \overline{\mathtt{x}} \mapsto \overline{\infty}\ ], [\ \overline{\mathtt{h}} \mapsto \overline{0}\ ]$, indicating that the initial value of fields is ∞ and that of assets $\overline{\mathtt{h}}$ is empty;
– $\Phi_i(\mathtt{x})\,\{^{\Phi_{i-1}^{\mathit{eff}}(\mathtt{x})}/_{\Phi(\mathtt{x})}\ \}$ is the liquidity expression obtained from $\Phi_i(\mathtt{x})$ by substituting the occurrences of the symbolic value $\Phi(\mathtt{x})$ with the effects of the previous transition $\Phi_{i-1}^{\mathit{eff}}(\mathtt{x})$. Similarly for assets.
– $[\![\mathtt{e}]\!]$ computes the liquidity value of the expression (which has no symbolic values) by computing the result of the operations $\sqcap$ and $\sqcup$.

As an example, consider the Bet contract and the compuation

$$
\begin{aligned}
t_1 &= \texttt{@Init Better1.place\_bet @First} : \Phi, \Xi \to \Phi_1, \Xi_1 \\
t_2 &= \texttt{@First Better2.place\_bet @Run} : \Phi, \Xi \to \Phi_2, \Xi_2 \\
t_3 &= \texttt{@Run DataProvider.data @End} : \Phi, \Xi \to \Phi_3, \Xi_3
\end{aligned}
$$

with the environments

$$
\begin{aligned}
\Phi \ &= \quad [\ \mathtt{val1} \mapsto \chi_1, \mathtt{val2} \mapsto \chi_2, \mathtt{source} \mapsto \chi_3, \mathtt{alea} \mapsto \chi_4, \mathtt{amount} \mapsto \chi_5, \\
&\qquad \mathtt{t\_before} \mapsto \chi_6, \mathtt{t\_after} \mapsto \chi_7\ ], \\
\Xi \ &= \quad [\ \mathtt{wallet1} \mapsto \xi_1, \mathtt{wallet2} \mapsto \xi_2\ ]
\end{aligned}
$$

$$
\begin{aligned}
\Phi_1, \Xi_1 &= \Phi[\mathtt{x} \mapsto \infty, \mathtt{val1} \mapsto \infty], \Xi[\mathtt{wallet1} \mapsto \xi_1 \sqcup \infty, \mathtt{y} \mapsto 0] \\
\Phi_2, \Xi_2 &= \Phi[\mathtt{x} \mapsto \infty, \mathtt{val2} \mapsto \infty], \Xi[\mathtt{wallet2} \mapsto \xi_2 \sqcup \infty, \mathtt{y} \mapsto 0] \\
\Phi_3, \Xi_3 &= \Phi[\mathtt{x} \mapsto \infty, \mathtt{z} \mapsto \infty], \Xi[\mathtt{wallet1} \mapsto 0, \mathtt{wallet2} \mapsto 0]
\end{aligned}
$$

The effects of the computation are the following ones:

$$
\begin{aligned}
\Phi_0^{\mathit{eff}} &= [\ \mathtt{val1} \mapsto \infty, \mathtt{val2} \mapsto \infty, \mathtt{source} \mapsto \infty, \mathtt{alea} \mapsto \infty, \mathtt{amount} \mapsto \infty, \\
&\qquad \mathtt{t\_before} \mapsto \infty, \mathtt{t\_after} \mapsto \infty\ ] \\
\Xi_0^{\mathit{eff}} &= [\ \mathtt{wallet1} \mapsto 0, \mathtt{wallet2} \mapsto 0\ ]
\end{aligned}
$$

$$
\begin{aligned}
\Phi_1^{\mathit{eff}} &= \Phi_0[\mathtt{x} \mapsto \infty] & \Xi_1^{\mathit{eff}} &= \Xi_0[\mathtt{wallet1} \mapsto \infty, \mathtt{y} \mapsto 0] \\
\Phi_2^{\mathit{eff}} &= \Phi_1^{\mathit{eff}}[\mathtt{x} \mapsto \infty] & \Xi_2^{\mathit{eff}} &= \Xi_1^{\mathit{eff}}[\mathtt{wallet2} \mapsto \infty, \mathtt{y} \mapsto 0] \\
\Phi_3^{\mathit{eff}} &= \Phi_2^{\mathit{eff}}[\mathtt{x} \mapsto \infty, \mathtt{z} \mapsto \infty] & \Xi_3^{\mathit{eff}} &= \Xi_2^{\mathit{eff}}[\mathtt{wallet1} \mapsto 0, \mathtt{wallet2} \mapsto 0]
\end{aligned}
$$

The last phase of the liquidity analysis amounts to considering the set of computations between two states of the automata. If the automata *has no cycle*, this set is *finite*. Therefore, let $\{\varphi_1, \cdots, \varphi_r\}$ be the set of computations from the initial state and the final states (assume there is at least one) and let $\Xi_{\varphi_1}^{\mathit{eff}}, \cdots, \Xi_{\varphi_r}^{\mathit{eff}}$ be the corresponding final effects. Then we take the environment $\Xi^{\mathit{eff}} = \Xi_{\varphi_1}^{\mathit{eff}} \sqcup \cdots \sqcup \Xi_{\varphi_r}^{\mathit{eff}}$. The contract is liquid if

1. there is no asset $\mathtt{h}$ such that $\Xi^{\mathit{eff}}(\mathtt{h}) = \infty$;
2. if, in every transition of the computations with effects Ξ_i^{eff}, there is no $\mathtt{h} \in \mathit{dom}(\Xi_i^{\mathit{eff}}) \backslash \mathit{dom}(\Xi)$ (a formal parameter that is an asset) such that $\Xi_i^{\mathit{eff}}(\mathtt{h}) = \infty$.

In the case of the Bet contract, there are three computations, corresponding to the following sequence of function calls and events

φ_1 = Better1.place_bet. Better2.place_bet. DataProvider.data
φ_2 = Better1.place_bet. Better2.place_bet. ev_22
φ_3 = Better1.place_bet. ev_15

It is easy to verify that the final effects of these computations are always $[\ \mathtt{wallet1} \mapsto 0, \mathtt{wallet2} \mapsto 0\ ]$ and every transition empties the formal parameters that are assets. Therefore the Bet contract is liquid.

When the automata has cycles, let $\varphi = \varphi_1(\varphi_2)^n\varphi_3$ be the computation with a cycle φ_2 (for simplicity, assume that φ_2 has no repetition; we are discussing a basic case, the general case is similar). Let $\Phi, \Xi \to \Phi_\varphi, \Xi_\varphi$ be the composition of the labels of φ_2. Let also $\Phi^{\mathit{eff}}, \Xi^{\mathit{eff}}$ be the environments computed at the end of φ_1. The environments at the end of the cycle are defined by means of a *fixpoint technique*, *i.e.* $\mathit{FIX}_{\Phi,\Xi \to \Phi_\varphi,\Xi_\varphi}(\Phi^{\mathit{eff}}, \Xi^{\mathit{eff}})$. We notice that, in this setting, a (least) fixpoint always exists because the domains of environments are finite partial orders and the judgments $\Phi, \Xi \vdash S : \Phi', \Xi'$ are monotone. Once the final environment of the cycle has been computed, we continue as above to determine the final environment of φ.

6 Conclusions

We have presented *Stipula*, a simple domain-specific language featuring a distilled number of operations that enable the formalisation of the main elements of juridical acts, such as permissions, prohibitions, and obligations. A number of related projects [12, 10, 7] have put forward legal markup languages, to wrap logic and other contextual information around traditional legal prose, and providing templates for common contracts that can be customized by setting template's parameters with appropriate values. In *Stipula*, rather than software templates, it is possible to define specific programming patterns that can be used to encode the building blocks that can be used to describe, analyse and execute (thus enforce) legal agreements (see the Table 1).

This is similar to what has been done in [8] where the authors have defined a set of combinators expressing financial and insurance contracts,

together with a denotational semantics and algebraic properties that says what such contracts are worth. These ideas have been implemented by the Marlowe and Findel languages [1, 3], which are (small) domain specific languages featuring constructs like participants, tokens, currency and timeouts to wait until a certain condition becomes true (similarly to *Stipula*).

We remark that legal contracts are more general and expressive than financial contracts. Accordingly, languages like Marlowe and Findel are built around a fixed set of contract's *combinators*, and they can be implemented using an interpreter, that is a single program that handles any financial contract by evaluating its (most external) combinator. The case of *Stipula* is more complex: agreement, assets, events, named states and named functions are programming primitives rather than combinators. Therefore each *Stipula* contract must be implemented, actually compiled, into a suitable running software, and the parties must collaborate by invoking the contract's functions to make the contract progress.

Being a principled high-level language, *Stipula* is implementation-agnostic, and does not commit to any architecture. In [6] we provided a detailed discussion about the implementation of the main elements of *Stipula* on top of either a centralized Java application or a distributed system such as a blockchain. In particular, *Stipula* might actually be implemented in terms of smart contracts written in Solidity or Obsidian [5, 4], which is based on state-oriented programming and explicit management of typed linear assets. This would bring in the advantages of a public and decentralized blockchain platform. However, we think that *Stipula*'s software/digital contracts are more general and encompass smart contracts: they provide benefits in terms of automatic execution and enforcement of contractual conditions, traceability, and outcome certainty even without using a blockchain. Their implementation might be more flexible, allowing a suitable level of privacy, reversibility and intermediation. Additionally, the intrinsic open nature of legal contracts is another challenge for smart contracts, that can hardly deal with the off-chain world: external data can enter the blockchain only through oracles, which are problematic in many senses, and the dynamic change of behaviour conflicts with the rigidity of smart contracts definition. Time is another big issue in blockchains.

Overall, we think that *Stipula* provides a programming model that is simple and rigorous, which are, in our opinion, fundamental criteria for reasoning about legal contracts and for understanding their basic principles. In our mind *Stipula*, and its toolset of formal methods, is the back-

bone of a framework where addressing and studying other, more complex features that are drawn from juridical acts.

References

1. Cardano Documentation. `https://docs.cardano.org/`, 2020.
2. Massimo Bartoletti and Roberto Zunino. Verifying liquidity of bitcoin contracts. In *Principles of Security and Trust - 8th International Conference, POST 2019*, volume 11426 of *Lecture Notes in Computer Science*, pages 222–247. Springer, 2019.
3. Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. Findel: Secure derivative contracts for ethereum. In *Financial Cryptography and Data Security - FC 2017*, volume 10323 of *Lecture Notes in Computer Science*, pages 453–467. Springer, 2017.
4. Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Can advanced type systems be usable? an empirical study of ownership, assets, and typestate in obsidian. *Proc. ACM Program. Lang.*, 4(OOPSLA):132:1–132:28, 2020.
5. Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *ACM Trans. Program. Lang. Syst.*, 42(3):14:1–14:82, 2020.
6. Silvia Crafa, Cosimo Laneve, and Giovanni Sartor. Pacta sunt servanda: legal contracts in Stipula. Technical report, arXiv:2110.11069, 10 2021.
7. Lexon Foundation. Lexon Home Page. `http://www.lexon.tech`, 2019.
8. Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering, functional pearl. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 280–292. ACM, 2000.
9. Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
10. Open Source Contributors. The Accord Project. `https://accordproject.org`, 2018.
11. 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.
12. Aaron Wright, David Roon, and ConsenSys AG. OpenLaw Web Site. `https://www.openlaw.io`, 2019.

A *Stipula*: the set of type inference rules

$$\text{[T-CONSTANT]}\ \frac{\kappa \in \mathtt{T}}{\Gamma, \Delta \vdash \kappa : \mathtt{T}, \mathit{true}} \qquad \text{[T-NOW]}\ \Gamma, \Delta \vdash \mathtt{now} : \mathtt{real}, \mathit{true} \qquad \text{[T-FIELD]}\ \Gamma, \Delta \vdash \mathtt{x} : \Gamma(\mathtt{x}), \mathit{true} \qquad \text{[T-ASSET]}\ \Gamma, \Delta \vdash \mathtt{h} : \mathtt{real}, \mathit{true}$$

$$\text{[T-AOP]}\ \frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Gamma, \Delta \vdash E' : \alpha', \Upsilon' \qquad \Upsilon'' = (\alpha, \alpha' = \mathtt{real}) \wedge \Upsilon \wedge \Upsilon'}{\Gamma, \Delta \vdash E\ \mathtt{aop}\ E' : \mathtt{real}, \Upsilon''} \qquad \text{[T-ROP]}\ \frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Gamma, \Delta \vdash E' : \alpha', \Upsilon' \qquad \Upsilon'' = (\alpha = \alpha') \wedge \Upsilon \wedge \Upsilon'}{\Gamma, \Delta \vdash E\ \mathtt{rop}\ E' : \mathtt{bool}, \Upsilon''}$$

$$\text{[T-BOP]}\ \frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Gamma, \Delta \vdash E' : \alpha', \Upsilon' \qquad \Upsilon'' = (\alpha, \alpha' = \mathtt{bool}) \wedge \Upsilon \wedge \Upsilon'}{\Gamma, \Delta \vdash E\ \mathtt{bop}\ E' : \mathtt{bool}, \Upsilon''}$$

$$\text{[T-SEND]}\ \frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon}{\Gamma, \Delta \vdash E \to \mathtt{A} : \Upsilon} \qquad \text{[T-UPDATE]}\ \frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Upsilon' = (\Gamma(\mathtt{x}) = \alpha) \wedge \Upsilon}{\Gamma, \Delta \vdash E \to \mathtt{x} : \Upsilon'} \qquad \text{[T-ASEND]}\ \frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \mathtt{h} \in \mathit{dom}(\Delta) \qquad \Upsilon' = (\alpha = \mathtt{real}) \wedge \Upsilon}{\Gamma, \Delta \vdash E \multimap \mathtt{h}, \mathtt{A} : \Upsilon'}$$

$$\text{[T-AUPDATE]}\ \frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \mathtt{h}, \mathtt{h}' \in \mathit{dom}(\Delta) \qquad \Upsilon' = (\alpha = \mathtt{real}) \wedge (\Delta(\mathtt{h}) = \Delta(\mathtt{h}')) \wedge \Upsilon}{\Gamma, \Delta \vdash E \multimap \mathtt{h}, \mathtt{h}' : \Upsilon'}$$

$$\text{[T-COND]}\ \frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Gamma, \Delta \vdash S : \Upsilon' \qquad \Gamma, \Delta \vdash S' : \Upsilon'' \qquad \Upsilon''' = (\alpha = \mathtt{bool}) \wedge \Upsilon \wedge \Upsilon' \wedge \Upsilon''}{\Gamma, \Delta \vdash \mathtt{if}\ (E)\ \{\ S\ \}\ \mathtt{else}\ \{\ S'\ \} : \Upsilon'''} \qquad \text{[T-COMPSTM]}\ \frac{\Gamma, \Delta \vdash S : \Upsilon' \qquad \Gamma, \Delta \vdash S' : \Upsilon''}{\Gamma, \Delta \vdash S\ S' : \Upsilon' \wedge \Upsilon''}$$

$$\text{[T-EVENT]}\ \frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Gamma, \Delta \vdash S : \Upsilon' \qquad \Upsilon'' = (\alpha = \mathtt{real}) \wedge \Upsilon \wedge \Upsilon'}{\Gamma, \Delta \vdash E \gg \mathtt{@Q}\{\ S\ \} \Rightarrow \mathtt{@Q}' : \Upsilon''} \qquad \text{[T-COMPEVNT]}\ \frac{\Gamma, \Delta \vdash \mathtt{W} : \Upsilon' \qquad \Gamma, \Delta \vdash \mathtt{W}' : \Upsilon''}{\Gamma, \Delta \vdash \mathtt{W}\ \mathtt{W}' : \Upsilon' \wedge \Upsilon''}$$

$$\text{[T-AGREEMENT]}\ \frac{\bigcup_{i\in 1..n} \overline{\mathtt{A}_i} \subseteq \overline{\mathtt{A}} \quad \bigcup_{i\in 1..n} \overline{\mathtt{v}_i} \subseteq \overline{\mathtt{v}} \quad \bigcap_{i\in 1..n} \overline{\mathtt{v}_i} = \varnothing}{\vdash \mathtt{agreement}\ (\overline{\mathtt{A}})(\overline{\mathtt{v}})\{\overline{\mathtt{A}_1} : \overline{\mathtt{v}_1}\ \cdots\ \overline{\mathtt{A}_n} : \overline{\mathtt{v}_n}\}}$$

$$\text{[T-FUNCTION]}\ \frac{\begin{array}{c}\overline{Y}, \overline{V}\ \mathit{fresh} \qquad \Gamma' = \Gamma[\overline{\mathtt{y}} \mapsto \overline{Y}] \qquad \Delta' = \Delta[\overline{\mathtt{v}} \mapsto \overline{V}] \\ \Gamma', \Delta' \vdash E : \alpha, \Upsilon \qquad \Gamma', \Delta' \vdash S : \Upsilon' \qquad \Gamma', \Delta' \vdash \mathtt{W} : \Upsilon'' \\ \Upsilon''' = (\alpha = \mathtt{bool}) \wedge \Upsilon \wedge \Upsilon' \wedge \Upsilon''\end{array}}{\Gamma, \Delta \vdash @\mathtt{Q}\ \mathtt{A}\ :\ \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{v}}](E)\,\{\,S\,W\,\} \Rightarrow @\mathtt{Q}' : \mathtt{A}.\mathtt{f}(\overline{Y})[\overline{V}], \Upsilon'''}$$

$$\text{[T-PROGRAM]}\ \frac{\begin{array}{c}\overline{X}, \overline{Z}\ \mathit{fresh} \quad \vdash \mathtt{G} \quad \left([\overline{\mathtt{x}} \mapsto \overline{X}], [\overline{\mathtt{h}} \mapsto \overline{Z}] \vdash \mathtt{F}_i : \mathtt{A}_\mathtt{i}.\mathtt{f}_\mathtt{i}(\overline{Y_i})[\overline{V_i}], \Upsilon_i\right)^{i\in 1..n} \\ \bigwedge_{i\in 1..n} \Upsilon_i \Vdash \sigma\end{array}}{\vdash \mathtt{stipula}\ \mathtt{C}\ \{\,\mathtt{assets}\ \overline{\mathtt{h}}\ \ \mathtt{fields}\ \overline{\mathtt{x}}\ \mathtt{G}\ \mathtt{F}_1 \cdots \mathtt{F}_n\ \} : [\,\overline{\mathtt{x}} : \sigma(\overline{X}),\ \overline{\mathtt{h}} : \sigma(\overline{Z}),\ \sigma\left(\mathtt{A}_\mathtt{i}.\mathtt{f}_\mathtt{i}(\overline{Y_i})[\overline{V_i}]\right)^{i\in 1..n}\,]}$$

Table 1. The type inference system of *Stipula* [It is assumed that the pair `A.f` uniquely identifies a function]

B *Stipula*: the set of liquidity rules

$$\text{[L-ZERO]}\quad \Phi, \Xi \vdash 0 : \mathbb{0} \qquad \text{[L-CONSTANT]}\quad \frac{\kappa \neq 0}{\Phi, \Xi \vdash \kappa : \infty} \qquad \text{[L-VAR]}\quad \frac{\mathtt{x} \in dom(\Phi)}{\Phi, \Xi \vdash \mathtt{x} : \Phi(\mathtt{x})} \qquad \text{[L-ASSET]}\quad \frac{\mathtt{h} \in dom(\Xi)}{\Phi, \Xi \vdash \mathtt{h} : \Xi(\mathtt{h})}$$

$$\text{[L-TIMES]}\quad \frac{\Phi, \Xi \vdash E : \mathbb{e} \quad \Phi, \Xi \vdash E' : \mathbb{e}'}{\Phi, \Xi \vdash E \times E' : \mathbb{e} \sqcap \mathbb{e}'} \qquad \text{[L-OP]}\quad \frac{\Phi, \Xi \vdash E : \mathbb{e} \quad \Phi, \Xi \vdash E' : \mathbb{e}' \quad \mathtt{op} \neq \times}{\Phi, \Xi \vdash E \,\mathtt{op}\, E' : \mathbb{e} \sqcup \mathbb{e}'}$$

$$\text{[L-SEND]}\quad \Phi, \Xi \vdash E \multimap \mathtt{A} : \Phi, \Xi \qquad \text{[L-UPDATE]}\quad \frac{\Phi, \Xi \vdash E : \mathbb{e}}{\Phi, \Xi \vdash E \multimap \mathtt{x} : \Phi[\mathtt{x} \mapsto \mathbb{e}], \Xi} \qquad \text{[L-AUPDATE]}\quad \frac{\mathbb{e} = \Xi(\mathtt{h}) \sqcup \Xi(\mathtt{h}')}{\Phi, \Xi \vdash \mathtt{h} \multimap \mathtt{h}' : \Phi, \Xi[\mathtt{h} \mapsto 0, \mathtt{h}' \mapsto \mathbb{e}]}$$

$$\text{[L-EXPAUPD]}\quad \frac{E \neq \mathtt{h} \quad \Phi \vdash E : \mathbb{e} \quad \mathbb{e}' = (\mathbb{e} \sqcap \Xi(\mathtt{h})) \sqcup \Xi(\mathtt{h}')}{\Phi, \Xi \vdash E \multimap \mathtt{h}, \mathtt{h}' : \Phi, \Xi[\mathtt{h}' \mapsto \mathbb{e}']} \qquad \text{[L-ASEND]}\quad \Phi, \Xi \vdash \mathtt{h} \multimap \mathtt{A} : \Phi, \Xi[\mathtt{h} \mapsto 0] \qquad \text{[L-EXPASEND]}\quad \frac{E \neq \mathtt{h}}{\Phi, \Xi \vdash E \multimap \mathtt{h}, \mathtt{A} : \Phi, \Xi}$$

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

$$\text{[L-FUNCTION]}\quad \frac{\begin{array}{c} W = \left(e_i \gg @\mathtt{Q}_i\{\ S_i\ \} \Rightarrow @\mathtt{Q}'_i\right)^{i \in I} \\ \Phi[\overline{\mathtt{x}'} \mapsto \overline{\infty}], \Xi[\overline{\mathtt{h}'} \mapsto \overline{\infty}] \vdash S : \Phi', \Xi' \qquad \left(\Phi, \Xi \vdash S_i : \Phi'_i, \Xi'_i\right)^{i \in I} \end{array}}{\Phi, \Xi \vdash @\mathtt{Q}\ \mathtt{A} : \mathtt{f}(\overline{\mathtt{x}'})[\overline{\mathtt{h}'}](E)\{\ S\, W\} \Rightarrow @\mathtt{Q}' : \begin{array}{c} @\mathtt{Q}\ \mathtt{A.f} : \Phi, \Xi \to \Phi', \Xi'\ @\mathtt{Q}' \\ \left(@\mathtt{Q}_i\ \mathtt{ev\_}i : \Phi, \Xi \to \Phi'_i, \Xi'_i\ @\mathtt{Q}'_i\right)^{i \in I} \end{array}}$$

$$\text{[L-PROGRAM]}\quad \frac{\overline{\chi}, \overline{\xi}\ \textit{fresh} \quad \left([\overline{\mathtt{x}} \mapsto \overline{\chi}], [\overline{\mathtt{h}} \mapsto \overline{\xi}] \vdash \mathtt{F}_i : \mathcal{L}_i\right)^{i \in 1..n}}{\vdash \mathtt{stipula\ C}\ \{\, \mathtt{assets}\ \overline{\mathtt{h}}\ \ \mathtt{fields}\ \overline{\mathtt{x}}\ \mathtt{G}\ \mathtt{F}_1 \cdots \mathtt{F}_n\ \} : \mathcal{L}_1, \cdots, \mathcal{L}_n}$$

Table 2. The Liquidity static analysis of *Stipula*.