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

Silvia Crafa[1] and Cosimo Laneve[2]

[1] University of Padova
[2] University of Bologna

Abstract. We discuss the design principles of *Stipula*, a domain specific language that can assists lawyers in programming legal contracts through specific software patterns. The language is based on a small set of primitives, that precisely correspond to the distinctive elements of legal contracts, and that are amenable to be prototyped on both centralized or distributed systems. We also outline two formal techniques to reason about *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" [12]. These agreements are basically *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, in this domain, 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.

Therefore we decided to work in close connection with lawyers to select few concise and intelligible primitives that have a precise correspondence with the distinctive elements of legal contracts. The resulting language, called *Stipula*, is a new domain-specific language with a formal operational semantics, so that the behaviour is fully specified and amenable to automatic verification. Its actual adoption by legal practitioners still requires a human-readable interface, such as an IDE support or a visual language interface, but we think that the design and the theory of this *legal calculus* [3] provide interesting insights on the application of programming languages to the legal field.

The complete definition of *Stipula* can be found in [7]; 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 also discuss two 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' functions. This system is useful to type check the correctness of operations, thus preventing basic errors with contract's data and assets. Then we overview an analyzer that statically checks the presence of executions of the legal contract leaving assets frozen into the contract without being redeemable by any party (liquidity). 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 establish 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){
   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`). A variant of the contract might 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){
   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 value of "`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 highligths that `Lender` is trusting the contract to act as intermediary that can store relevant informations (such as, 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 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 a trusted third party, the `Authority`, is necessary to supervise the dispute and to provide a 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 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

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

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 coconnected 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 *Stipula* is untyped, to keep a simple syntax; however, a type inference system that allows one to derive types is discussed in Section 4. Finally, we notice that 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` 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 lose.

[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.

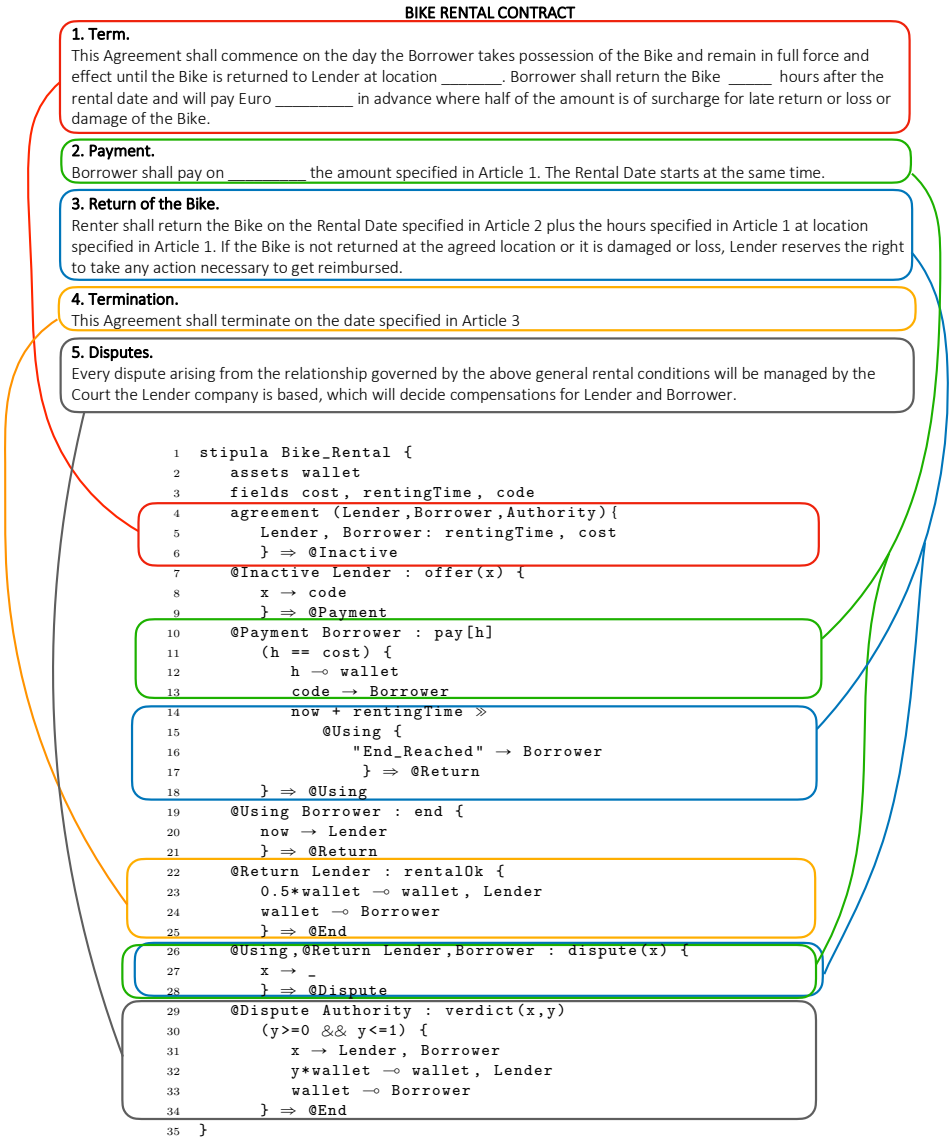

Fig. 2. A standard Bike Rental contract and its modelling in *Stipula*

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

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

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

  @First Better2: place_bet(x)[h]
    (h == amount){
        h ⊸ 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

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 dropped because there is no type annotation in standard legal contracts and therefore they may be initially obscure to unskilled users, such as lawyers. On the other hand, a lightweight and well designed typed syntax is acknowledged as an effective support to produce quality software and to enhance code comprehension. Therefore, we postpone the choice of a suitable typed syntax to the study of an appropriate programming interface that help legal practitioners to program in *Stipula*. Nevertheless the language comes with a type inference system that allows one to derive types of assets, fields and functions' arguments, so to statically prevent basic programming errors. In this section we discuss the main design principles of the system.

Stipula has the following *primitive types*

$$T ::= \quad \texttt{real} \;\mid\; \texttt{bool} \;\mid\; \texttt{string} \;\mid\; \texttt{time} \;\mid\; \texttt{asset}$$

that mirror the set of values of the language: real numbers, booleans, strings, time values, and assets. What is exactly a time value will be specified by the concrete implementation (either over a centralized system or a distributed platform such a s a blockchain), but in general *Stipula* admits both *absolute time* values (as the date `"2022/1/1:00:15"T`) and *relative time* expressions, like `now + 3`, standing for 3 days from now or after that at least 3 blocks have been appended to the underlying blockchain. Values of type asset can be *divisible* resources (e.g. (crypto)currencies) or *indivisible* assets (e.g. smart keys, or NFT tokens). In particular, divisible assets correspond to positive real numbers (therefore we admit a subsumption rule from assets to real numbers), while indivisible ones

must be considered as a whole and can be either empty asset (`0`) or a full asset (e.g. the constants `key1234` and `nft123`).

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 [10] for details of the technique). Here we just discuss the most relevant rules, that are based on 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. The notation $\Gamma[\mathtt{x} \mapsto X]$, resp. $\Delta[\mathtt{h} \mapsto V]$, stands for either the update or the extension of the environment, depending on whether `x`, resp. `h`, 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 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 the constant κ belongs to `T` we derive that κ has type `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 `x`, *i.e.,* $\Gamma(\mathtt{x}) = \alpha$. For example, the typing of the assignment `"hello"` $\to$ `x` in the environments Γ, Δ returns the constraint $\Gamma(\mathtt{x}) =$ `string`.

The following rule is the typing of the asset transfer:

$$\frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Upsilon' = (\alpha = \mathtt{real}) \wedge (\Delta(\mathtt{h}), \Delta(\mathtt{h}') = \mathtt{asset}) \wedge \Upsilon}{\Gamma, \Delta \vdash E \multimap \mathtt{h}, \mathtt{h}' : \Upsilon'}$$

The rule defines the type of the *withdraw* of the value of E from the asset $\mathtt{h}$ and the corresponding *addition* to $\mathtt{h}'$. Therefore, the expression E must have type `real`, since it corresponds to a quantity to be withdrawn form the asset $\mathtt{h}$ and added to the asset $\mathtt{h}'$. An additional Subsumption rule is introduced to promote assets to be real, so that assets, when used within expressions (*e.g.* $\mathtt{h} \multimap \mathtt{h}, \mathtt{wallet}$), are considered as reals. We also remark that the type system does not check the amount of assets that are withdrawn, but the operational semantics of *Stipula* prevents the (unsafe) execution of the transfer operation whenever $\mathtt{h}$ does not own enough assets, *e.g.* $2 * \mathtt{h} \multimap \mathtt{h}, \mathtt{wallet}$.

Given a contract's function $\mathtt{F}_i$, the judgment $\Gamma, \Delta \vdash \mathtt{F}_i : \Gamma_i, \Delta_i, \Upsilon_i$ collects the constraints Υ_i generated from the typing of the function body, and the type environments Γ_i, Δ_i that associate fresh type variables to the parameters names:

$$\frac{\begin{array}{c}\overline{Y}, \overline{V} \textit{ fresh} \qquad \Gamma' = \Gamma[\overline{\mathtt{y}} \mapsto \overline{Y}] \qquad \Delta' = \Delta[\overline{\mathtt{k}} \mapsto \overline{V}] \\ \Gamma', \Delta' \vdash E : \alpha, \Upsilon \qquad \Gamma', \Delta' \vdash S : \Upsilon' \qquad \Gamma, \Delta \vdash W : \Upsilon'' \\ \Upsilon''' = (\alpha = \mathtt{bool}) \wedge \Upsilon \wedge \Upsilon' \wedge \Upsilon''\end{array}}{\Gamma, \Delta \vdash @\mathtt{Q}\ \mathtt{A} : \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}](E)\,\{\, S\, W \,\} \Rightarrow @\mathtt{Q}' : [\overline{\mathtt{y}} \mapsto \overline{Y}], [\overline{\mathtt{k}} \mapsto \overline{V}], \Upsilon'''}$$

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:

$$\frac{\begin{array}{c}\overline{X}, \overline{Z} \textit{ fresh} \qquad \vdash \mathtt{G} \qquad \Gamma = [\overline{\mathtt{x}} \mapsto \overline{X}] \quad \Delta = [\overline{\mathtt{h}} \mapsto \overline{Z}] \\ \left(\Gamma, \Delta \vdash \mathtt{F}_i : \Gamma_i, \Delta_i, \Upsilon_i\right)^{i \in 1..n} \qquad \bigwedge_{i \in 1..n} \Upsilon_i \Vdash \sigma\end{array}}{\vdash\ \mathtt{stipula\ C}\,\{\,\mathtt{assets}\ \overline{\mathtt{h}}\ \ \mathtt{fields}\ \overline{\mathtt{x}}\ \ \mathtt{G}\ \ \mathtt{F}_1 \cdots \mathtt{F}_n\,\} : [\,\sigma(\Gamma, \Gamma_1 \cdots \Gamma_n), \sigma(\Delta\Delta_1 \cdots \Delta_n)\,]}$$

In the rule fresh type variables are associated to contracts fields and functions parameters, both assets and non assets. These associations are recorded in the type environments $\Gamma, \Gamma_1 \cdots \Gamma_n$ and $\Delta\Delta_1 \cdots \Delta_n$ (all contracts names are assumed to be different). Then the type of the contract is obtained by applying the substitution σ to these environments. Whenever the inference system is not able to derive a ground type for a contract name, that is $\sigma(X)$ is a type variable, it means that there are no type constraints for that name. In particular, as regards assets, the type system only collects assets type identities and the type variable can be safely instantiated to the ground type `asset`.

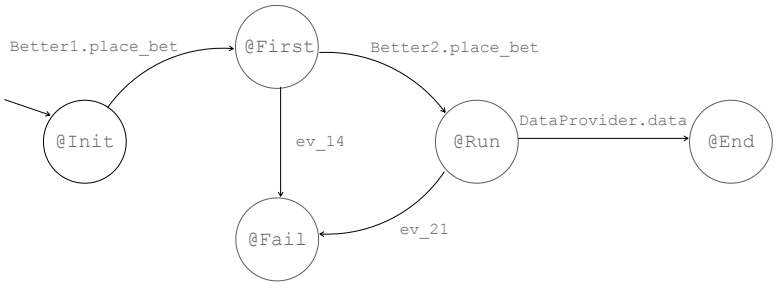

Fig. 3. The finite-state automata of the Bet contract

5 An analyzer of liquidity

Liquidity is a major security property of every program managing assets because it guarantees that assets are never frozen forever inside contracts [2]. In particular,

***liquidity*:** a *Stipula* contract is *liquid* if, whenever an asset becomes not-0, then there is a continuation that has a state where *every asset* is 0.

According to the definition, liquidity reduces to the analysis that a state is *reachable*, which is not trivial in Solidity because functions have guards that may disable invocations and events that may prevent invocations. What we discuss in this paper is a technique for pruning the space of analysis of reachability: our technique returns witnesses to be checked by an off-the-shelf reachability tool. For simplicity sake, in this section, we consider the sublanguage where statements $\mathtt{E} \multimap \mathtt{h}, \mathtt{A}$ and $\mathtt{E} \multimap \mathtt{h}, \mathtt{h}'$ have the shape $c{*}\mathtt{h} \multimap \mathtt{h}, \mathtt{A}$ and $c{*}\mathtt{h} \multimap \mathtt{h}, \mathtt{h}'$, respectively (every example in this paper matches this constraint). We also restrict our analysis to contracts such that computations do not pass through the same state twice. That is, consider the underlying finite state automaton of the contract where states are those specified by the contract and transitions are either functions or events – Figure 3 reports the finite state automaton of the Bet contract in Section 3. We are restricting to contracts where the underlying finite state automata have no cycle (the technique where this limitation is drop requires technicalities that are out of the scope of this paper).

The liquidity analyzer of a *Stipula* contract has three phases:

1. the liquidity effects of each transition of the automaton is statically computed by calculating (an over-approximation of) the assets and

the asset parameters of every function and event. More precisely, using a *type system*, we define the *liquidity label* of every function $\texttt{Q A.f Q}' : \Xi \to \Xi'$ and every event $\texttt{Q ev\_i Q}' : \Xi \to \Xi'$, where the initial environment Ξ associates contract's assets with symbolic names, while the final environments Ξ' records the effect of the execution of the corresponding bodies;

2. then we compute the liquidity effects of *computations* (*i.e.* sequences of transitions) of the automaton, by suitably merging the final environment of a transition with the initial environment of the next transition;
3. finally, we consider all the functions and events that either updates the assets or carry asset parameters and check whether (*i*) all asset parameters are emptied by functions' executions, and (*ii*) for every function/event modifying an asset, there is a continuation that empties all the assets of the contract. We remark the role of asset parameters: if a contract function is called by passing an asset parameter, like an amount of currency (as in the function `pay` of the `Bike_Rental` contract), that amount is no more available to the caller because of the linear semantics of assets. Therefore it is essential that the parameter is drained by the function and the currency is moved into a contract asset (*cif.* the instruction $\texttt{h} \multimap \texttt{wallet}$ in line 12 of the `Bike_Rental` contract in Table 2) or sent to a party, otherwise that currency is frozen and the program is not liquid. The condition (*ii*) above requires to trace the asset movements performed by computations and to verify that contract assets (*cif.* `wallet`) have been emptied.

Below we detail few critical aspects of the analysis. The liquidity type system returns, for every transition of the automaton, an over-approximation of the balances of the assets that expresses whether an asset is empty – notation $\mathbb{0}$ – or not empty – notation ∞. The values $\mathbb{0}$ and ∞ are called *liquidity values*. We use the following notation:

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

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

They are ordered as $\mathbb{0} < \infty$ and $0 \leqslant \xi$ and $\xi \leqslant \infty$; the operations $\sqcup$ and $\sqcap$ respectively return the maximum and the minimum value of the two arguments.

– *environments* Ξ, which map contract's assets and asset parameters to liquidity expressions.

– *liquidity labels* $t : \Xi \to \Xi'$ where t is either $\texttt{Q A.f Q}'$ (a function) or $\texttt{Q ev\_i Q}'$ (an event) and $\Xi \to \Xi'$ records the liquidity effects of fully executing the body of the transition t.
– *judgments* $\Xi \vdash E : \mathbb{e}$ for expressions, $\Xi \vdash S : \Xi'$ for statements and $\Xi \vdash \texttt{@Q A:f}(\overline{\texttt{x}})[\overline{\texttt{h}'}]\ (E)\{\, S\, W \,\} \Rightarrow \texttt{@Q}' : \mathcal{L}$ for function definitions, where $\mathcal{L}$ is a set of liquidity labels.

As regards expressions, we consider only constants (because of the restriction on the shape of asset expressions). In particular, the two rules

$$\Xi \vdash 0 : \mathbb{0} \qquad\qquad \frac{\kappa \neq 0}{\Xi \vdash \kappa : \infty}$$

assert that every constant has liquidity value ∞ but for the constant 0.

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

$$\frac{\mathbb{e} = \Xi(\texttt{h}) \sqcup \Xi(\texttt{h}')}{\Xi \vdash \texttt{h} \multimap \texttt{h}' : \Xi[\texttt{h} \mapsto \mathbb{0}, \texttt{h}' \mapsto \mathbb{e}]} \qquad \frac{c \neq 1 \quad \Xi \vdash c : \mathbb{e} \qquad \mathbb{e}' = (\mathbb{e} \sqcap \Xi(\texttt{h})) \sqcup \Xi(\texttt{h}')}{\Xi \vdash c * \texttt{h} \multimap \texttt{h}, \texttt{h}' : \Xi[\texttt{h}' \mapsto \mathbb{e}']}$$

According to the rule on the left, the final asset environment of $\texttt{h} \multimap \texttt{h}'$ (which is an abbreviation for $\texttt{h} \multimap \texttt{h}, \texttt{h}'$) has $\texttt{h}$ that is emptied and $\texttt{h}'$ that gathers the value of $\texttt{h}$, henceforth the liquidity expression $\Xi(\texttt{h}) \sqcup \Xi(\texttt{h}')$. Notice that, when both $\texttt{h}$ and $\texttt{h}'$ are $\mathbb{0}$, the overall result is $\mathbb{0}$. In the rule on the right, the asset $\texttt{h}$ is decreased by an amount that is moved to $\texttt{h}'$. Since c is not 1, the static analysis can only safely assume that the asset $\texttt{h}$ is not emptied by this operation (if it was not empty before). Therefore, after the withdraw, the liquidity value of $\texttt{h}$ has not changed. On the other hand, the asset $\texttt{h}'$ is increased of some amount if $\texttt{h}$ has a non zero liquidity value, henceforth the expression $(\mathbb{e} \sqcap \Xi(\texttt{h})) \sqcup \Xi(\texttt{h}')$. In particular, when both $\Xi(\texttt{h})$ and $\Xi(\texttt{h}')$ are $\mathbb{0}$, the overall result is $\mathbb{0}$.

The rule for conditionals is

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

where the operation $\sqcup$ on environments is defined pointwise: $(\Xi' \sqcup \Xi'')(\texttt{h}) = \Xi'(\texttt{h}) \sqcup \Xi''(\texttt{h})$. That is, the liquidity analyzer over-approximates the final environments of $\texttt{if}\ (E)\ \{\ S\ \}\ \texttt{else}\ \{\ S'\ \}$ by taking the maximum values between the results of parsing S (that corresponds to a true value of E) and those of S' (that corresponds to a false value of E). The expression E is overlooked by the analyzer.

The rule for *Stipula* contracts collects the *liquidity labels* that describe the liquidity effects of each contract's function; each function assumes

injective environments that just associate contracts' assets with symbolic names:

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

In turn, the rule for function definitions is:

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

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, saying that the transition named $\mathtt{Q\ A.f\ Q}'$ has liquidity effects $\Xi[\overline{\mathtt{h}'} \mapsto \overline{\infty}] \to \Xi'$. As explained above, Ξ just associates contract's assets, with symbolic names. The analysis of the function's body S additionally assumes that the function parameters $\overline{\mathtt{h}'}$ are bound to ∞, because they may be any value. The liquidity effects of S are recorded by the environment Ξ'. In particular, if $\Xi'(\mathtt{h}') = \infty$, where $\mathtt{h}'$ is an asset parameter, *i.e.* $\mathtt{h}' \notin \mathit{dom}(\Xi)$, then the asset $\mathtt{h}'$ has not been emptied by S. Therefore the asset is frozen into the parameter and the contract is not liquid. The premises also verifies the liquidity effects of events' bodies 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. The set I in the rule is intended to be the set of (the initial) code lines of the events scheduled by the function. For example, the liquidity types of the Bet contract are ($\Xi = [\ \mathtt{wallet1} \mapsto \xi_1, \mathtt{wallet2} \mapsto \xi_2\ ]$):

$$\begin{array}{lll} \mathtt{Init\ \ Better1.place\_bet\ \ First} & : & \Xi[\mathtt{h} \mapsto \infty] \to \Xi[\mathtt{wallet1} \mapsto \xi_1 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\ \mathtt{First\ \ Better2.place\_bet\ \ Run} & : & \Xi[\mathtt{h} \mapsto \infty] \to \Xi[\mathtt{wallet2} \mapsto \xi_2 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\ \mathtt{Run\ \ DataProvider.data\ \ End} & : & \Xi \to \Xi[\mathtt{wallet1} \mapsto \mathbb{0}, \mathtt{wallet2} \mapsto \mathbb{0}] \\ \mathtt{First\ \ ev\_14\ \ Fail} & : & \Xi \to \Xi[\mathtt{wallet1} \mapsto \mathbb{0}] \\ \mathtt{Run\ \ ev\_21\ \ Fail} & : & \Xi \to \Xi[\mathtt{wallet1} \mapsto \mathbb{0}, \mathtt{wallet2} \mapsto \mathbb{0}] \end{array}$$

To calculate the effects that a computation has on the assets' balances we use abstract computations. An *abstract computation* is a finite sequences of labelled transitions $\varphi = \{t_i : \Xi_i \to \Xi'_i\}^{i \in 1..n}$. We define the liquidity type of an abstract computation φ, noted $\mathtt{L}_\varphi$, by merging the

final environments of a transition with the initial environments of the next one. In particular, let $\overline{\mathtt{h}}$ be the assets of the contract and $\Xi|_{\overline{\mathtt{h}}}$ be the environment Ξ restricted to the domain $\overline{\mathtt{h}}$. Then

$$\mathrm{L}_\varphi = \ \Xi_1^{(b)}|_{\overline{\mathtt{h}}} \to \Xi_n^{(e)}|_{\overline{\mathtt{h}}}$$

where $\Xi_1^{(b)}$ and $\Xi_n^{(e)}$ ("b" stays for *begin*, "e" stays for *end*) are defined as follows

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

For example, consider the Bet contract computation

$$\varphi = \texttt{Init Better1.place\_bet First ; First Better2.place\_bet Run ; Run DataProvider.data End}$$

Then $\mathrm{L}_\varphi = \Xi_1^{(b)}|_{\{\texttt{wallet1},\texttt{wallet2}\}} \to \Xi_3^{(e)}|_{\{\texttt{wallet1},\texttt{wallet2}\}}$ where

$$\begin{array}{ll}
\Xi_1^{(b)} & = \Xi[\mathtt{h} \mapsto \infty] \\
\Xi_1^{(e)} & = \Xi[\texttt{wallet1} \mapsto \xi_1 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\
\Xi_2^{(b)} & = \Xi[\texttt{wallet1} \mapsto \xi_1 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\
\Xi_2^{(e)} & = \Xi[\texttt{wallet1} \mapsto \xi_1 \sqcup \infty, \texttt{wallet2} \mapsto \xi_2 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\
\Xi_3^{(b)} & = \Xi[\texttt{wallet1} \mapsto \xi_1 \sqcup \infty, \texttt{wallet2} \mapsto \xi_2 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\
\Xi_3^{(e)} & = \Xi[\texttt{wallet1} \mapsto \mathbb{0}, \texttt{wallet2} \mapsto \mathbb{0}, \mathtt{h} \mapsto \mathbb{0}]
\end{array}$$

The last phase of the liquidity analysis amounts to checking that (*i*) the execution of every function empties every asset parameter, and (*ii*) for every function or event modifying an asset field, there is a continuation (which is a computation) that empties all the assets. We notice that, as regards (*ii*), we cannot restrict to a local analysis (as in (*i*)) but we have to consider computations because assets may become 0 in several steps. For example, for the Bet contract, there are two problematic functions: `Init Better1.place_bet First` (that updates `wallet1`) and `First Better2.place_bet Run` (that updates `wallet2`). Our technique, by using liquidity types of computations, returns all the computations that start at `First` and at `Run` that empty the assets `wallet1` and `wallet2`. In particular, for `First`, it returns

```
First Better2.place_bet Run ; Run DataProvider.data End
First ev_14 Fail
First Better2.place_bet Run ; Run Ev_21 Fail
```

(the reader is invited to verify that, in the final environments are $[\texttt{wallet1} \mapsto \mathbb{0}, \texttt{wallet2} \mapsto \mathbb{0}]$). For `Run` we have

```
Run DataProvider.data End
Run ev_21 Fail .
```

Provided that, at least one of the computations starting at `First` and of those starting at `Run` can be actually executed (as we anticipated, our analysis needs to be complemented by a reachability analysis), the Bet contract is liquid. Actually, it turns out that this is the case because every foregoing computation can be performed.

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 [13, 11, 8] 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 [9] 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, 4], 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 [7] 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 [6, 5], 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 blockchain-based 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. On the other hand, we have shown that *Stipula* contracts may take advantage of an explicit Authority party and suitable programming patterns to flexibly deal with the exceptional behaviors occurring in the external context.

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 backbone 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. Shrutarshi Basu, Anshuman Mohan, James Grimmelmann, and Nate Foster. Legal calculi. Technical report, ProLaLa 2022 ProLaLa Programming Languages and the Law, 2022. At `https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi`.
4. 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.
5. Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Can advanced type systems be usable? an empirical study of ownership, assets, and typestate in obsidian. *Proc. ACM Program. Lang.*, 4(OOPSLA):132:1–132:28, 2020.
6. 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.
7. Silvia Crafa, Cosimo Laneve, and Giovanni Sartor. Pacta sunt servanda: legal contracts in Stipula. Technical report, arXiv:2110.11069, 10 2021.
8. Lexon Foundation. Lexon Home Page. `http://www.lexon.tech`, 2019.
9. 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.
10. Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
11. Open Source Contributors. The Accord Project. `https://accordproject.org`, 2018.
12. 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.
13. Aaron Wright, David Roon, and ConsenSys AG. OpenLaw Web Site. `https://www.openlaw.io`, 2019.