Pacta sunt servanda: Legal contracts in Stipula

Pacta sunt servanda: legal contracts in *Stipula*

Silvia Crafa

University of Padova

Cosimo Laneve

University of Bologna

Giovanni Sartor

University of Bologna, European University Institute

Adele Veschetti

University of Bologna

Abstract

We present *Stipula*, a domain specific language that may assist legal practitioners in programming legal contracts through specific patterns. The language is based on a small set of programming abstractions that correspond to common patterns in legal contracts. We illustrate the language by means of two paradigmatic legal contracts: a bike rental and a bet contract. *Stipula* comes with a formal semantics, an observational equivalence and a type inference system, that provide for a clear account of the contracts' behaviour and illustrate how several concepts from concurrency theory can be adapted to automatically verify the properties and the correctness of software-based legal contracts. We also discuss a prototype centralized implementation of *Stipula*.

Keywords: Legal contracts, assets, transition system, normative equality, type inference system, prototype.

1. Introduction

Law 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 decisions, to contractual agreements, and court judgements, might considerably benefit from a sensible digitalisation. However, it is extremely difficult to translate their legal texts into computable representation, since they are expressed through natural language, which is at the same time very expressive and ambiguous.

The target of our study is the digitalisation of *legal contracts*, which are "*agreements giving rise to a binding legal relationship between parties*" [35]. Since parties are free to express their agreement in the language and medium they choose (*freedom of form*, a principle shared by modern legal systems), drafting a contract by using a programming language (rather than, as usual, natural language) seems a valuable option. Advantages are in terms of speed-up, lack of ambiguity, and automatic and transparent enforcement of the contractual clauses. For this reason, several projects are being developed for defining programming languages to write legal contracts, *e.g.* [25, 34, 37, 20, 16, 3, 27]. However, finding the suitable abstraction level for contractual languages is still an open issue [9]. Indeed, a programming language for drafting legal contracts should be easy-to-use and to understand for legal practitioners and ordinary citizens, since complete awareness of the computational effects is necessary for a genuine agreement of the

parties over the content of the contract. At the same time, the language should be fairly expressive, have a running environment with a precise semantics, and possibly supply sensible analyzers.

In this article we define a new language, called *Stipula*, which aims to be an intermediate domain-specific language: more concrete than a user-friendly contract specification language, and more abstract than a full-fledged programming language. In this sense, *Stipula* can be considered a *legal calculus*[4], pivoted on few selected, concise and intelligible primitives, together with a precise formalization, influenced by principles of concurrent systems [28]. The preliminary research, conducted with the legal expertise of the third author, has identified some important features of legal contracts, that need to be preserved in their computational representation.

First of all, the formation of the contracts requires the agreement (the so called *meeting of the minds* of the parties, all of which must accept the terms of the contract). Only the achievement of the agreement will trigger the legal effects of the contract.

Secondly, a contract may produce multiple different legal effects, and in particular the following:

- the creation of normative positions, such as *permissions, prohibitions obligation*, and *powers*, which may dynamically evolve as time goes by (e.g., the permission to use a good, or the power to exercise an option, may only last until a deadline is met);
- the transfer of *currency* or other *assets* (e.g. the property of a physical or digital good, to be used for payments, escrows and securities)

Legal contracts may have the following features

- *Openness* to external conditions, which may trigger the initiation, change of termination of the contractual effects (e.g. the value of a stock at a given date may trigger its transfer);
- *Subsequent modification*, which may take place when the parties agree to modify the content of the contract;
- *Third party adjudication and enforcement*, i.e., the possibility that a dispute resolution mechanism is activated when a party challenges the content or the execution of the contract.

The basic primitives of Stipula have been designed in such a way as to meet the just listed requirements, providing template programs and design patterns that fit the building blocks of legal contracts. More precisely, (1) the `agreement` construct directly encodes the meeting of the minds. It does so, by requiring that the parties agree on all terms, before triggering the execution of the contract. (2) Permissions and prohibitions are modelled through the possibility and the impossibility for a party to invoke a function at a certain contract's stage. Obligations are enforced by scheduling an event that checks the fulfilment at a given date, issuing a corresponding penalty if the obligation has not been met. (3) Asset manipulation is syntactically distinguished, to stress the fact that assets cannot be destroyed or created but only transferred. (4) Contract clauses whose execution depends on external events are implemented by means of a party that takes the role of intermediary and assumes the legal responsibility of timely retrieving data from the external source agreed in the terms of the contract. This is different from relying on Oracles web services, to whom legal responsibilities can hardly be attributed. Finally, (5) dispute resolutions and enforcement of legal requirements are implemented in *Stipula* by including in the contract a party that takes the role of authority with corresponding functionalities. Authorities may also be invoked for the verification of non-automatically verifiable contextual circumstances, such as force majeure.

By combining these abstractions, many different legal contracts can be written in *Stipula*: we describe in this article a bike rental and a bet contract, and we refer to [11] for the encoding of the free rent of a physical good and a digital licence contract. Preliminary investigations show that simple, but realistic, contracts[1] can be easily expressed in *Stipula*. Additionally, our prototype

[1]For instance the bike rent in http://www.thebicyclecellar.com/wp-content/uploads/2013/10/Bike-Rental-Contract-BW.pdf can be written in Stipula [12].

demonstrates how the distinctive elements of the language can be implemented: the paper discusses a centralized implementation of features like agreements, assets and events that has permitted us to experiment our ideas and to run the *Stipula* contracts. Overall, in line with the literature about legal calculi [4, 17], we think that the theory of such a high level, domain-specific language is a first interesting step, that sheds some light on the digitalisation of legal texts.

The article is organized as follows. Section 2 gives a gentle introduction to *Stipula*, while Section 3 defines precisely its syntax. The semantics of the language is presented in Section 4. *Stipula* has been motivated by the remark that a legal contract actually defines an *interaction protocol* between parties (that regulates permissions, prohibitions, obligations). Therefore, in Section 5, taking inspiration from concurrency theory, we define an equational theory based on a *bisimulation*, which allows us to equate contracts that are syntactically different but are legally equivalent since they exhibit the same observable normative elements. On top of the formal semantics, in Section 6, we also define a type inference system that consents to derive types for fields, assets and contracts' functions. These are the basics to develop further formal techniques to automatically verify the properties and the correctness of *Stipula* legal contracts. Section 7 discusses the main issues of our prototype. We end our contribution by discussing the related work in Section 8 and delivering our final remarks in Section 9.

2. Getting started

The 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* has 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 the *Stipula* contract regulating a bike rental service defined in Figure 1; lines 1-3 define the name of the contract and the list of *assets* and *fields* that are used therein; the code in lines 5-7 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`).

Notice that the agreement also includes 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. The code in line 6 expresses 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.

The second distinctive feature of legal contracts is that the set of normative elements, namely permissions, prohibitions and obligations, usually change over time according to the actions that have been done (or not). To model these changes, *Stipula* commits to a *state-machine programming* style. 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. Lines 9-11 define 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`. Intuitively, this 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. It is worth to notice that this 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` (line 10) stores the value sent by the `Lender` into a contract's field called `code` that cannot be accessed outside the contract. Once `code` has been received, the contract moves to a state `@Payment` (line 11) where presumably the `Borrower` will eventually pay (actually, he is allowed to pay) for the rental.

```
1  stipula Bike_Rental {
2     assets wallet
3     fields cost, rentingTime, code
4
5     agreement (Lender,Borrower,Authority){
6        Lender, Borrower: rentingTime, cost
7        } ⇛ @Inactive
8
9     @Inactive Lender : offer(x) {
10       x → code
11       } ⇛ @Payment
12
13    @Payment Borrower : pay[h]
14       (h == cost) {
15          h ⊸ wallet
16          code → Borrower
17          now + rentingTime ≫
18             @Using {
19                "End_Reached" → Borrower
20                 } ⇛ @Return
21       } ⇛ @Using
22
23    @Using Borrower : end {
24       now → Lender
25       } ⇛ @Return
26
27    @Return Lender : rentalOk {
28        wallet ⊸ Lender
29       } ⇛ @End
30
31    @Using,@Return Lender,Borrower : dispute(x) {
32       x → _
33       } ⇛ @Dispute
34
35    @Dispute Authority : verdict(x,y)
36       (y>=0 && y<=1) {
37          x → Lender x → Borrower
38          wallet×y ⊸ wallet, Lender
39          wallet ⊸ Borrower
40       } ⇛ @End
41 }
```

Figure 1: The Bike Rent contract

A further feature is the management of *assets*: currency is required for payments and escrows, while 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 bikes smart lock [11]. Moreover, in the traditional setting of bike rental, 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 that contracts deal with assets transfers, so to remove intermediation even from the payments. In particular, 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 `pay` in lines 13-21 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` in line 15, 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`. Once the fee has been payed, the `Borrower` gets the access code to the bike and the contract transits into a `@Using` state.

Lines 17-20 illustrate the 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. Accordingly, the function `pay` issues an event

that terminates the renting service when the time limit is reached and the bike has not been returned yet. The time limit is expressed by `now + rentingTime` and, at that moment, if the bike has not been already returned (*i.e.*, the state of the contract is still `@Using`), a message of returning the bike is sent to the `Borrower` (`"End_Reached" → Borrower` in line 19) and the contract moves to the state `@Return`. We remark that events are not triggered by any party: they are automatically executed when the time condition is met. The termination of the rental further requires the `Lender` to confirm the absence of damages before receiving the fee (function `rentalOK` in lines 27-29). For the sake of simplicity this contract does not impose a penalty to the `Borrower` for late return, but it is not difficult to modify the rental contract by requiring the borrower to pay a higher fee that is deposited as security and is reimbursed in case of timely return [12].

The foregoing code does not address disputes, *e.g.* contentions because the bike is returned, or initially was, broken or damaged. Third-party adjudication and possibly enforcement is a distinctive aspect of legal contracts: disputes concerning contracts are usually decided by courts (unless the parties themselves decide to submit the case to arbitration). Disputes resolution has a simple modelling in *Stipula* that does not require new ad-hoc features, and somehow mimic the behaviour of a court. In fact, when contract's violations cannot be 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` (which must have been included in the agreement), to supervise the dispute and to provide a trusted resolution mechanism. More concretely, the party assuming the role of authority takes the legal responsibility of interfacing with a court or an Online Dispute Resolutions platform[2]. Lines 31-40 illustrate the encoding of the monitoring and enforcement mechanism: the function `dispute` may be invoked either by the `Lender` or by the `Borrower`, either in the state `@Using` or `@Return`, 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 (line 35), 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 `wallet×y —∘ wallet, Lender` in line 38 *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 `wallet×1 —∘ 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 the following simple template is used:

```
~@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.

As a further example for testing the expressivity of *Stipula*, we consider in Figure 2 a contract ruling a bet. This kind of legal contract contains an element of randomness (*alea*, which is a future, aleatory event, 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. Therefore, a digital encoding of a bet contract requires that the parties explicitly agree on the source of data, usually

[2]as The European ODR platform at `https://ec.europa.eu/consumers/odr`.

```
1  stipula Bet {
2    assets wallet1, wallet2
3    fields val1, val2, alea, source, amount, t_before, t_after
4
5    agreement (Better1, Better2, DataProvider){
6       DataProvider, Better1, Better2 : source, alea, t_after
7       Better1, Better2 : amount, t_before
8    } ⇛ @Init
9
10   @Init Better1 : place_bet(x)[h]
11     (h == amount){
12         h —∘ wallet1
13         x → val1
14         t_before ≫ @First { wallet1 —∘ Better1 } ⇛ @Fail
15     } ⇛ @First
16
17   @First Better2: place_bet(x)[h]
18     (h == amount){
19         h —∘ wallet2
20         x → val2
21         t_after ≫ @Run {
22             wallet1 —∘ Better1
23             wallet2 —∘ Better2 } ⇛ @Fail
24   } ⇛ @Run
25
26   @Run DataProvider : data(x,y,z)[]
27     (x==source && y==alea){
28         if (z==val1 && z==val2){        // Better1 and Better2 win
29             wallet1 —∘ Better1
30             wallet2 —∘ Better2
31         } else if (z==val1 && z!=val2){ // The winner is Better1
32             wallet2 —∘ Better1
33             wallet1 —∘ Better1
34         } else if (z!=val1 && z==val2){ // The winner is Better2
35             wallet1 —∘ Better2
36             wallet2 —∘ Better2
37         } else {                        // No winner
38             wallet1 —∘ DataProvider
39             wallet2 —∘ DataProvider
40         }
41    } ⇛ @End
42 }
```

Figure 2: The contract for a bet

an accredited web page or a specific online service – stored in the field `source` – that will publish the final value of the aleatory event. This value will be communicated by the party that assumes the role of `DataProvider`, taking the legal responsibility of supplying the correct data from the agreed source. In particular, it is not necessary that the actual data is directly provided by a trusted institution or an accredited online service, such as an Oracle service, who could hardly take an active legal responsibility in a bet contract. But two betters, say Alice and Bob, can agree to rely on a third party Carl for supplying data, or they can simply agree on the fact that Alice takes both the role of `Better1` and `DataProvider`.

The *Stipula* code in Figure 2 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]. Both bets must be placed within an (agreed) time limit `t_before` (line 14), to ensure that the legal bond is established before the occurrence of the aleatory event. The second timeout, scheduled in line 21, is used to ensure the contract termination even if the `DataProvider` fails to provide the expected data, through the call of the function `data`. 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. When the function `data` is called and the arguments `x` and `y` match the expected `source` and `alea` of the bet, the betters are rewarded according to the result `z`. For simplicity we assume

[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.

```
stipula C {
    assets h̄
    fields x̄
    agreement(Ā){
        Ā1 : x̄1
        ...                    // ⋃i∈1..n Āi ⊆ Ā ,  ⋃i∈1..n x̄i ⊆ x̄ ,  ⋂i∈1..n x̄i = ∅
        Ān : x̄n
    } ⇒ @Q
    F
}
```

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

Table 1: Syntax of *Stipula*

that the data-provider service gets the two bets when they lose. We notice that assets and fields are private: they cannot be directly accessed by the parties that can only interact with the contract by invoking the corresponding functions.

Compared to the Bike Rental, 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, that is entitled (and legally bound) to call the contract's function to supply the expected external data. As usual, any dispute that might render the contract voidable or invalid, *e.g.* one better knew the result of the match in advance, or the `DataProvider` supplied an incorrect value, can be handled by including an `Authority`, according to the pattern illustrated in the Bike Rental example.

3. The *Stipula* language

We use countable sets of *contract names*, ranged over by $\mathtt{C}$, $\mathtt{C}'$, ...; names referring to digital identities, called *parties*, ranged over by $\mathtt{A}$, $\mathtt{A}'$, ...; *function names* ranged over $\mathtt{f}$, $\mathtt{g}$, ... Assets and generic contract's fields are syntactically set apart since they have different semantics, similarly for functions' parameters. Then we assume a countable set of *asset names*, ranged over by $\mathtt{h}$, $\mathtt{k}$, ..., to be used both as contract's assets and function's asset parameters and of *non asset names*, ranged over $\mathtt{x}$, $\mathtt{y}$, ..., to be used both as contact's fields and function's non asset parameters. Finally, we will use $\mathtt{Q}$, $\mathtt{Q}'$, ..., to range over contract states. To simplify the syntax, we often use the vector notation $\bar{\mathtt{x}}, \bar{\mathtt{h}}, \bar{\mathtt{A}}$ to denote possibly empty sequences of elements.

A legal contract is written in *Stipula* according to the syntax of Table 1 where $\mathtt{C}$ identifies the contract; its body contains contract's *assets* and *fields*, the *agreement code*, and the sequence F of functions, written according to the syntax given in Table 1. Technically, the agreement is the constructor that specifies the terms of the contract (the initial values of a subset of the contract's fields) and the set $\bar{\mathtt{A}}$ of involved parties. The terms of the contract do not include assets, that are initially empty and that must be explicitly transferred afterwards. The agreement also specifies the initial state of the contract. It is assumed that fields, assets and parties' names do not contain duplicates.

Functions F are sequences of function definitions, where $\_$ stands for the empty sequence. The syntax highlights who is the party $\mathtt{A}$ that can invoke the function and the state $\mathtt{Q}$ when the invocation is admitted. Function's parameters are split in two lists: the *formal parameters* $\bar{\mathtt{y}}$ in brackets and the *asset parameters* $\bar{\mathtt{k}}$ in square brackets. The *precondition* E constrains the execution of the body of $\mathtt{f}$. Finally the *body* $\{\ S\ W\ \} \Rightarrow @\mathtt{Q}'$ specifies the *statement part* S, the *event part* W, and the state $\mathtt{Q}'$ of the contract when the function execution terminates.

Function's parameters are assumed without duplicates and empty lists of parameters are shortened by omitting empty parenthesis. With no loss of generality we assume that the names of functions' parameters do not clash with the names of contract's fields and assets. Additionally we assume that the parameters $\overline{\mathtt{y}}$ and $\overline{\mathtt{k}}$ do not occur in W to enforce that the event is correctly executed outside the scope of the function.

Statements S include the empty statement `_`, and different prefixes followed by a continuation. Prefixes P use the two symbols $\multimap$ and $\rightarrow$ to differentiate operations on assets and of fields, respectively. The prefix $E \rightarrow \mathtt{x}$ updates the field or the parameter `x` with the value of E; $E \rightarrow \mathtt{A}$ sends the value of E to the party `A`; $E \multimap \mathtt{h}, \mathtt{h}'$ subtracts the value of E to the asset `h` and adds it to $\mathtt{h}'$ – *resources stored in assets can be moved but cannot be destroyed* –, $E \multimap \mathtt{h}, \mathtt{A}$ subtracts the value of E to the asset `h` and transfers it to `A`. The operational semantics will prevent assets with negative values. Statements also include a *conditional* $\mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S'\,\}$ with the standard semantics. In the rest of the paper we will always abbreviate $\mathtt{h} \multimap \mathtt{h,h}'$ and $\mathtt{h} \multimap \mathtt{h,A}$ (which are very usual, indeed) into $\mathtt{h} \multimap \mathtt{h}'$ and $\mathtt{h} \multimap \mathtt{A}$, respectively. We also assume that the party names occurring in the code of a contract belong to the set of names mentioned in the agreement.

Events W are sequences of *timed continuations* that schedule some code for future execution. More precisely, the term $E \gg \mathtt{@Q}\,\{S\} \Rightarrow \mathtt{@Q}'$ schedules an execution that is triggered at a time $\mathbb{t}$ that is the value of E. When triggered, the continuation S will be automatically executed if the contract's state is `Q`. At the end of the execution of S, the contract transits to $\mathtt{Q}'$. The scheduling of the events is nondeterministic when several events are ready to be executed at the same time value – see the semantics in Section 4. Moreover, since the execution of the event scheduled for first may change the contract's state, this could either enable or prevent the execution of the continuation of the other ready events.

Expressions E include constant values ranged over by $v, u, ...$, names of assets, fields and parameters generically ranged over by X, the keyword `now` representing the current time (when the code is executed), and both binary and unary operations. The precise set of constant values and operations depend on the specific language implementation (see Section 7), but we assume that they include:

- real numbers n with the standard set of arithmetic operations ($+$, $-$, $\times$, $/$).
- boolean values `false` and `true` with boolean operations: conjunction (`&&`), disjunction (`||`) and negation (`!`).
- strings s are sequences of characters that are pre- and post-fixed by "`"`", with the string concatenation operation denoted by `^`.
- time values $\mathbb{t}$ represent global system's clock and are written as `"2022/1/1:00:15"`. We admit expressions like $\mathbb{t} + 5$, where 5 is intended as minutes, and expressions like $\mathtt{now} + 3$, where `now` represents the current global time (it will be replaced during the execution). We remark that time values can be passed as parameters to functions, so that events can be simply $\mathtt{x} \gg \mathtt{@Q}\,\{\,S\,\} \Rightarrow \mathtt{@Q}'$.
- asset values a and parties identities. The source code of *Stipula* does not contain these kind of values, which are instead introduced at runtime as actual parameters of functions' calls (the asset values) and during the agreement (the parties identities). We will discuss them in the next section.

Relational operations ($<, >, <=, >=, ==$) are available between any expression, and the meaning of operations involving assets will be discussed later, when introducing the operational semantics of the language. Moreover, for simplicity we let the keyword `now` only occur within expressions used as time guards of events.

In this paper we assume that a *Stipula* program consists of a single contract. We postpone to future work the study of language extensions allowing cross references between legal contracts using inheritance and composition.

4. Semantics

The meaning of *Stipula* primitives is defined operationally by means of a transition relation. First of all, the syntax of the source code must be extended to include terms that appear only at runtime, namely parties' identities and asset values.

Constants corresponding to parties' actual identities are denoted with italic fonts $A, A', ...$, *Alice*, *Bob*, *ItalyRent* to distinguish from parties formal names $\mathtt{A}, \mathtt{A}', ...$, `Borrower`, `Better1`, `Lender`. These constants correspond to digital identities, that are crucial in the legal setting (see Section 7), and they are introduced when agreements take place[4]. Indeed, parties' formal names appearing within the source code of a contract correspond to the *roles* of the parties in the contract, *e.g.*, the lender and the borrower of the bike, while at runtime these roles are assumed by real legal entities, *e.g.*, Alice Smith and Italy Rent s.r.l.. As discussed in Section 1, the same party can even assume two roles in the same contract: Alice and Bob may agree on a bet contract where Alice takes also the role of `DataProvider`.

Constant values of type asset are transferred to the contract as actual parameters of function invocations. We assume that contract's assets cover *currencies* and *tokens* like smart keys, NFTs or other kinds of unique tokens needed to access a digital good as an ebook or a software. A contract's asset holding currency, as `wallet` in Figure 1, actually contains zero or a positive real amount of coins, *e.g.*, euros, dollars, bitcoins, etc. For simplicity, currency values are represented by positive real numbers followed by a "`D`", e.g. `100.0D` (often shortened into `100D`). On the other hand a token is either present or absent; they are represented by positive integer numbers followed by "T", like *1234T*. The absence of a token is represented by the value *0T*.
Given the extended syntax, we let $\mathtt{C}(\mathbb{Q}, \ell, \Sigma, \Psi)$ be a *runtime contract* where

- $\mathtt{C}$ is the contract name;
- $\mathbb{Q}$ is the current state of the contract: it is either $\_$ (for no state) or a contract state $\mathtt{Q}$;
- ℓ is a mapping from names (parties, fields, assets and function's parameters) to values;
- Σ is the (possibly empty) residual of a function body or of an event handler, *i.e.* Σ is either $\_$ or a term $S\ W \Rrightarrow \mathtt{@Q}$;
- Ψ is a (possibly empty) multiset of *pending events* that have been already scheduled for future execution but not yet triggered. We let Ψ be $\_$ when there are no pending events, otherwise $\Psi = W_1 \mid \cdots \mid W_n$ such that each W_i is a single event expression (not a sequence), and its *time guard* is an expression that has already been evaluated into a time value $\mathbb{t}_i$.

Runtime contracts are ranged over by $\mathbb{C}$, $\mathbb{C}'$, $\cdots$. A *runtime configuration* is a pair $\mathbb{C}, \mathbb{t}$, where $\mathbb{t}$ is the time value of the system's global clock. The transition relation of *Stipula* is $\mathbb{C}, \mathbb{t} \xrightarrow{\mu} \mathbb{C}', \mathbb{t}'$, where

$$\mu \quad ::= \quad \_ \quad \mid \quad (\overline{A}, \overline{A_1} : \overline{v_1}, \cdots, \overline{A_n} : \overline{v_n}) \quad \mid \quad A : \mathtt{f}(\overline{u})[\overline{v}] \quad \mid \quad v \to A \quad \mid \quad v \multimap A\ .$$

That is the label μ is either empty, or denotes an initial agreement, or a function call, or a value send, or an asset transfer. Notice that these labels use the extended set of constants and values introduced in the runtime syntax. The formal definition of $\mathbb{C}, \mathbb{t} \xrightarrow{\mu} \mathbb{C}', \mathbb{t}'$ is given in Table 2, using the following auxiliary predicates and functions:

- $[\![E]\!]_\ell$ is a *partial function* that returns the value of E in the memory ℓ. The definition is standard, that is:

[4]Parties identities cannot be passed as function's parameters and they cannot be hard-coded into the source contracts, since they do not belong to the category of expressions.

– $[\![v]\!]_\ell = v$ for non-asset values, $[\![a]\!]_\ell = |a|$ for asset values, where $|\texttt{1234.0C}| = \texttt{1234.0}$ (that is the raw value of the currency) and $|\mathit{1234T}| = \texttt{1234}$, $[\![X]\!]_\ell = \ell(X)$ for fields and non-asset parameters, $[\![X]\!]_\ell = |\ell(X)|$ for assets and asset parameters. The evaluation of $[\![X]\!]_\ell$ is not defined when X is not in the domain of ℓ.

– let *uop* and *op* be the semantic operations corresponding to `uop` and `op`, then $[\![\texttt{uop}\, E]\!]_\ell = \mathit{uop}\, v$, $[\![E\ \texttt{op}\ E']\!]_\ell = v\ \mathit{op}\ v'$ with $[\![E]\!]_\ell = v, [\![E']\!]_\ell = v'$. Clearly `uop` and `op` are undefined when the corresponding semantic definition is undefined. For instance, the semantics is undefined when the type of arguments mismatch with the type of the semantic operation (e.g. `"hello" + 1`) and when `op` is a division and the divisor is 0.

- $[\![E]\!]^{\mathtt{a}}_\ell$ is the partial function returning asset values. It is defined as follows

  – for currencies ($+^{\mathtt{D}}$ and $-^{\mathtt{D}}$ are the semantic operations summing and subtracting currencies; $\times^{\mathtt{D}}$ multiplies a currency with a real number):

$$[\![E]\!]^{\mathtt{a}}_\ell = \begin{cases} E & \text{if } E \text{ is a currency value} \\ \ell(X) & \text{if } X \text{ is an asset name storing currencies} \\ [\![E']\!]^{\mathtt{a}}_\ell +^{\mathtt{D}} [\![E'']\!]^{\mathtt{a}}_\ell & \text{if } E = E' + E'' \text{ and } [\![E']\!]^{\mathtt{a}}_\ell \text{ and } [\![E'']\!]^{\mathtt{a}}_\ell \text{ are both currencies} \\ [\![E']\!]^{\mathtt{a}}_\ell -^{\mathtt{D}} [\![E'']\!]^{\mathtt{a}}_\ell & \text{if } E = E' - E'' \text{ and } [\![E']\!]^{\mathtt{a}}_\ell \text{ and } [\![E'']\!]^{\mathtt{a}}_\ell \text{ are both currencies} \\ & \text{and } |[\![E']\!]^{\mathtt{a}}_\ell| \geqslant |[\![E'']\!]^{\mathtt{a}}_\ell| \\ [\![E']\!]^{\mathtt{a}}_\ell \times^{\mathtt{D}} [\![E'']\!]_\ell & \text{if } E = E' \times E'' \text{ and } E'' \text{ is either a real number or a} \\ & \text{non-asset name} \end{cases}$$

  It is worth noticing that the second argument of $\times^{\mathtt{D}}$ is always a real number (in case E'' is an asset, we consider the corresponding `real`). For simplicity sake, we have not defined the operation of division in assets.

  – for tokens:

$$[\![E]\!]^{\mathtt{a}}_\ell = \begin{cases} E & \text{if } E \text{ is a token value} \\ \ell(X) & \text{if } X \text{ is an asset name storing tokens} \\ [\![E'']\!]^{\mathtt{a}}_\ell & \text{if } E = E' + E'' \text{ and } [\![E']\!]^{\mathtt{a}}_\ell = \mathit{0T} \\ \mathit{0T} & \text{if } E = E' - E'' \text{ and } [\![E']\!]^{\mathtt{a}}_\ell = [\![E'']\!]^{\mathtt{a}}_\ell \end{cases}$$

  That is, the operations we have defined on tokens are addition and subtraction. In case of addition, the operation is used for moving the token of the second argument in an asset, which is the first argument. Therefore we constrain the first argument to be empty, otherwise the move is meaningless. In case of subtraction, the operation must empty the first argument. Therefore the values of the two arguments must be equal and the result is *0T*.

- $[\![W\{^{\mathbb{t}}/_{\mathtt{now}}\}]\!]_\ell$ is the multiset of scheduled events obtained from the sequence W by replacing every `now` by $\mathbb{t}$ and evaluating every time expression in the time guards. That is $[\![\_]\!]_\ell = \_$ and $[\![E \gg \texttt{@Q}\ \{\ S\ \} \Rightarrow \texttt{@Q}'\ \ W']\!]_\ell = [\![E]\!]_\ell \gg \texttt{@Q}\ \{\ S\ \} \Rightarrow \texttt{@Q}' \mid [\![W']\!]_\ell$.

- Let Ψ be a multiset of pending events, and $\mathbb{t}$ a time value, then the predicate $\Psi\ ,\ \mathbb{t} \twoheadrightarrow$ is *true* whenever $\Psi = \mathbb{t}_1 \gg \texttt{@Q}_1\{S_1\} \Rightarrow \texttt{@Q}'_1 \mid \cdots \mid \mathbb{t}_n \gg \texttt{@Q}_n\{S_n\} \Rightarrow \texttt{@Q}'_n$ and, for every $1 \leqslant i \leqslant n$, $\mathbb{t}_i \neq \mathbb{t}$, *false* otherwise.

Rule [Agree] in Table 2 defines the agreement of the parties involved in the contract. This operation is a joint synchronization [18] where parties' formal names $\overline{\mathtt{A}}$ are replaced by actual parties identities $\overline{A}$ and some parties, namely $\overline{A_i}$, agree on the initial values $\overline{v_i}$ of the contract's fields $\overline{\mathtt{x}_i}$, for $i \in 1..n$. The resulting configuration moves the contract to the initial state `Q` and initializes the names $\overline{\mathtt{A}}$ and $\overline{\mathtt{x}}$ to the corresponding values, and the contract's assets to the empty value.

Rule [Function] defines invocations: the label specifies the party A performing the invocation and the function name `f` with the actual parameters. The transition may occur provided (*i*) the contract is in the state `Q` that admits invocations of `f` from A and (*ii*) the contract is *idle*,

[Agree]

$$\frac{\texttt{assets}\ \overline{\texttt{h}}\ \in\ \texttt{C} \qquad \texttt{agreement}(\overline{\texttt{A}})\{\ \overline{\texttt{A}_1 : \texttt{x}_1}\ \cdots\ \overline{\texttt{A}_n : \texttt{x}_n}\ \} \Rrightarrow \texttt{@Q}\ \in\ \texttt{C}}{\texttt{C}(\_, \varnothing, \_, \_)\ \texttt{,}\ \mathbb{t} \xrightarrow{(\overline{A},\ \overline{A_i : v_i}^{\,i \in 1..n})} \texttt{C}(\texttt{Q}, [\overline{\texttt{A}} \mapsto \overline{A}, \overline{\texttt{x}_i}\ \mapsto\ \overline{v_i}^{\,i \in 1..n}, \overline{\texttt{h}}\ \mapsto\ \overline{0}], \_, \_)\ \texttt{,}\ \mathbb{t}}$$

[Function]

$$\frac{\texttt{@Q A} : \texttt{f}(\overline{\texttt{y}})\,[\overline{\texttt{k}}]\ (E)\,\{\,S\,W\,\} \Rrightarrow \texttt{@Q}' \in \texttt{C} \qquad \Psi\ \texttt{,}\ \mathbb{t} \nrightarrow \qquad \ell(\texttt{A}) = A \quad \ell' = \ell[\overline{\texttt{y}} \mapsto \overline{u}, \overline{\texttt{k}} \mapsto \overline{v}] \quad [\![E]\!]_{\ell'} = \texttt{true}}{\texttt{C}(\texttt{Q}, \ell, \_, \Psi)\ \texttt{,}\ \mathbb{t} \xrightarrow{A : \texttt{f}(\overline{u})[\overline{v}]} \texttt{C}(\texttt{Q}, \ell', S\,W \Rrightarrow \texttt{@Q}', \Psi)\ \texttt{,}\ \mathbb{t}}$$

[State-Change]

$$\frac{[\![W\{^{\mathbb{t}}/_{\texttt{now}}\}]\!]_\ell = \Psi'}{\texttt{C}(\texttt{Q}, \ell, \_\,W \Rrightarrow \texttt{@Q}', \Psi)\ \texttt{,}\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q}', \ell, \_, \Psi' \mid \Psi)\ \texttt{,}\ \mathbb{t}}$$

[Event-Match]

$$\frac{\Psi = \mathbb{t} \gg \texttt{@Q}\ \{\ S\ \} \Rrightarrow \texttt{@Q}' \mid \Psi'}{\texttt{C}(\texttt{Q}, \ell, \_, \Psi)\ \texttt{,}\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q}, \ell, S \Rrightarrow \texttt{@Q}', \Psi')\ \texttt{,}\ \mathbb{t}}$$

[Tick]

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

[Value-Send]

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

[Asset-Send]

$$\frac{[\![E]\!]^{\texttt{a}}_\ell = a \quad \ell(\texttt{A}) = A \quad [\![\texttt{h} - a]\!]^{\texttt{a}}_\ell = a'}{\texttt{C}(\texttt{Q}, \ell, E \multimap \texttt{h}, \texttt{A}\ \Sigma, \Psi)\ \texttt{,}\ \mathbb{t} \xrightarrow{a \multimap A} \texttt{C}(\texttt{Q}, \ell[\texttt{h} \mapsto a'], \Sigma, \Psi)\ \texttt{,}\ \mathbb{t}}$$

[Field-Update]

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

[Asset-Update]

$$\frac{[\![E]\!]^{\texttt{a}}_\ell = a \quad [\![\texttt{h} - a]\!]^{\texttt{a}}_\ell = a' \quad [\![\texttt{h}' + a]\!]^{\texttt{a}}_\ell = a'' \quad \ell' = \ell[\texttt{h} \mapsto a', \texttt{h}' \mapsto a'']}{\texttt{C}(\texttt{Q}, \ell, E \multimap \texttt{h}, \texttt{h}'\ \Sigma, \Psi)\ \texttt{,}\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q}, \ell', \Sigma, \Psi)\ \texttt{,}\ \mathbb{t}}$$

[Cond-true]

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

[Cond-false]

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

Table 2: The transition relation of *Stipula*

i.e. the contract has no statement to execute – *c.f.* the left-hand side runtime contract – (*iii*) the precondition E is satisfied, and no event can be triggered – *c.f.* the premise $\Psi \ , \ \mathbb{t} \nrightarrow$. In particular, this last constraint expresses that events *have precedence* on possible function invocations. For example, if a payment deadline is reached and, at the same time, the payment arrives, it will be refused in favour of the event managing the deadline.

Rule [State-Change] says that a contract changes state when the execution of the statement in the function's body terminates. At this stage, the sequence of events W is added to the multiset of pending events once their time expressions have been evaluated (`now` is replaced by the current value of the clock).

Rule [Event-Match] specifies that event handlers may run provided there is no statement to perform and the time guard of the event has exactly the value of the global clock $\mathbb{t}$. Observe that the timeouts of the events are evaluated in an eager way when the event is scheduled – *c.f.* rule [State Change] – not when the event handler is triggered. Moreover, the state change performed at the end of the execution of the event handler is carried over again by the rule [State Change], with an empty sequence W.

Rule [Tick] defines the elapsing of time. This happens when the contract has no statement to perform (*i.e.* no function or event to conclude), and no event can be triggered. As a consequence, the complete execution of a function or of an event cannot last more than a single time unit. Even if the actual time management depends on the specific implementation of the language, we think that this semantics is appropriate, since in all the legal contracts we have analysed, functions and events contain few statements, there are no loops nor nested function calls. Additionally, even if ready events cannot be postponed because they encode legal deadlines, the implementation can reasonably work with a tolerance of few seconds (see Section 7). Nevertheless, it is worth to notice that the foregoing rules admit the paradoxical phenomenon that an endless sequence of function invocations does not make time elapse. An actual implementation of this semantics clearly prevents this behaviour, but this is fair, since no legal contract we analysed needed infinite sequences of function calls. Moreover, the contract equivalence studied in the next section shows that time is reasonably observed only through obligation deadlines; therefore in a contract behaviour what is relevant about a function is whether it can be executed before of after another function or an event, not its actual execution time, which can be freely postponed or anticipated by a few time units. Therefore, in this paper we have preferred to stick to the simpler semantics and accept a fair implementation.

As regards statements, the rule [Asset-Send] transfers part of an asset `h` to the party A. This part corresponds to the value of $[\![E]\!]^{\mathtt{a}}_{\ell}$; we remark that the premise $a = [\![E]\!]^{\mathtt{a}}_{\ell}$ is defined when the value of a is smaller or equal to the value of `h`. For example, if `h` holds `20D` then `25D` $\multimap$ `h, A` would not be executed, while `h` $\times$ `0.5` $\multimap$ `h, A` would remove `10D` from `h` and send them to `A`. In a similar way, [Asset_Update] moves a part a of an asset `h` to an asset `h'`. When assets are tokens, the premise of the rule [Asset-Send] is well defined only when `h` is moved as a whole, that is when the statement under evaluation is `h` $\multimap$ `h, A` (that we abbreviate `h` $\multimap$ `A`). Similarly, the premises of the rule [Asset-Update] are well defined only when a is the token in `h` and `h'` is empty. For example, when the statement is `h` $\multimap$ `h, h'` (and `h'` is empty). Rule [Value-Send] defines the sending of a value to a party. The value has to be either of type `real` or `string` or `bool`. The statement `1234D` $\rightarrow A$ is admitted: it means that A will receive the `real` value `1234` (*c.f.* the meaning of $[\![E]\!]_{\ell}$). Rule [Field-Update] defines updates of fields.

The initial configuration of a *Stipula* contract `C` is $\mathtt{C}(\_\ ,\ \varnothing\ ,\ \_\ ,\ \_)\mathtt{,t}$. The contract is *inactive* as long as no group of parties has interest to run it, *c.f.* rule [Agree]. The global clock can be any value because it corresponds to the absolute time.

Example.. We highlight the possible initial transitions of the `Bike_Rental` contract in Figure 1, where the actual identities of parties are *ItalyRent*, *Bob* and the third party authority is an Online Dispute Resolution service, say *ODR*. Let $\mu_0 = ((\mathit{ItalyRent}, \mathit{Bob}, \mathit{ODR}), (\mathit{ItalyRent},\ \mathit{Bob}) : (3600, 2))$, *i.e.* *ItalyRent* and *Bob* agree about renting a bike for 2 euro for 1 hour – the time is measured in seconds. Let also be $\ell = [\mathtt{Lender} \mapsto \mathit{ItalyRent}, \mathtt{Borrower} \mapsto \mathit{Bob}, \mathtt{Authority} \mapsto \mathit{ODR}, \mathtt{cost} \mapsto \mathtt{2c}, \mathtt{rentingTime} \mapsto 3600, \mathtt{wallet} \mapsto 0]$, and $\ell' = \ell[\mathtt{x} \mapsto \mathtt{123}, \mathtt{code} \mapsto \mathtt{123}]$ and $S\ W$

be the body of the function `pay`. Then

$$
\begin{array}{rll}
\texttt{Bike\_Rental}(\_\,,\,\varnothing\,,\,\_\,,\,\_)\;,\;0 & & \\
\xrightarrow{\mu_0} & \texttt{Bike\_Rental}(\texttt{Inactive}\,,\,\ell\,,\,\_\,,\,\_)\;,\;0 & [\textsc{Agree}] \\
\longrightarrow & \texttt{Bike\_Rental}(\texttt{Inactive}\,,\,\ell\,,\,\_\,,\,\_)\;,\;1 & [\textsc{Tick}] \\
\xrightarrow{\mathit{ItalyRent}:\texttt{offer(123)}} & \texttt{Bike\_Rental}(\texttt{Inactive}\,,\,\ell[\texttt{x}\mapsto 123]\,,\,\texttt{x}\to\texttt{code}\Rightarrow\texttt{@Payment}\,,\,\_)\;,\;1 & [\textsc{Function}] \\
\longrightarrow & \texttt{Bike\_Rental}(\texttt{Inactive}\,,\,\ell'\,,\,\_\Rightarrow\texttt{@Payment}\,,\,\_)\;,\;1 & [\textsc{Field-Update}] \\
\longrightarrow & \texttt{Bike\_Rental}(\texttt{Payment}\,,\,\ell'\,,\,\_\,,\,\_)\;,\;1 & [\textsc{State-Change}] \\
\longrightarrow & \texttt{Bike\_Rental}(\texttt{Payment}\,,\,\ell'\,,\,\_\,,\,\_)\;,\;2 & [\textsc{Tick}] \\
\longrightarrow & \texttt{Bike\_Rental}(\texttt{Payment}\,,\,\ell'\,,\,\_\,,\,\_)\;,\;3 & [\textsc{Tick}] \\
\xrightarrow{\mathit{Bob}:\texttt{pay[2D]}} & \texttt{Bike\_Rental}(\texttt{Payment}\,,\,\ell'[\texttt{h}\mapsto 2\texttt{D}]\,,\,\texttt{S W}\Rightarrow\texttt{@End}\,,\,\_)\;,\;3 & [\textsc{Function}]
\end{array}
$$

Errors in contract's execution correspond to stuck states, that is runtime configurations that do not transition anymore even if the contract's behavior is not terminated. Besides usual errors, like unsafe operations (*e.g.* `"hello"+5`), access to fields that have not been initialized and references to unknown (misspelled) identifiers, the distinctive errors of *Stipula* are the attempts to drain too much value from an asset, to forge new assets, and to accumulate tokens (*e.g.* $123T \multimap \texttt{h}$ where `h` already holds the token $567T$). All these errors correspond to stuck configurations, and we will later develop a type inference system to deal with some of them. We postone to future work a deeper analysis of contracts' errors, which requires an interdisciplinary analysis of the legal issues involved in the execution of the exceptional cases.

Before concluding, we observe that in *Stipula* legal contracts behave as follows:

1. the first action is always an agreement, which moves the contract to an idle state;
2. in an idle state, if there is a ready event with a matching state, it is triggered and completely executed, moving again to a (possibly different) idle state;
3. in an idle state, if there is no event to be triggered, *either* tick *or* call any permitted function (*i.e.* with matching state and preconditions). A function invocation amounts to execute its body until the end, which is again an idle state.

Stipula semantics has three sources of nondeterminism: (*i*) the order of the execution of ready event handlers, (*ii*) the order of the calls of permitted functions, and (*iii*) the delay of permitted function calls to a later time (thus, possibly, after other event handlers). For example, the contract `C` with two functions $\texttt{@Q A:f}\{\_\}\Rightarrow\texttt{@Q}$ and $\texttt{@Q A':g}\{\_\}\Rightarrow\texttt{@Q}$ behaves as either $\xrightarrow{A:\texttt{f}}\longrightarrow_n\xrightarrow{A':\texttt{g}}$ or $\xrightarrow{A':\texttt{g}}\longrightarrow_n\xrightarrow{A:\texttt{f}}$, where $\longrightarrow_n$ are n transitions that make the time elapse (rule [Tick]). As another example, consider a contract $\texttt{C}'$ with a function $\texttt{@Q A:f}\ \{\texttt{\_ now}\gg\texttt{@Q}'\{\ \texttt{"hello"}\to\texttt{A}\ \}\Rightarrow\texttt{@Q}'\ \}\Rightarrow\texttt{@Q}$ and a function $\texttt{@Q A':g}\ \{\_\}\Rightarrow\texttt{@Q}'$. Then it may behave as either $\xrightarrow{A:\texttt{f}}\xrightarrow{A':\texttt{g}}\xrightarrow{\texttt{"hello"}\to A}$ or as $\xrightarrow{A:\texttt{f}}\longrightarrow_n\xrightarrow{A':\texttt{g}}$, after which the action $\xrightarrow{\texttt{"hello"}\to A}$ is disabled, or as $\xrightarrow{A':\texttt{g}}$, which precludes the call of `f`. Given these sources of nondeterminism, we will show in the next section a bisimulation-based equivalence relation that clarifies when two contracts can be considered equivalent.

Remark.. The semantics of *Stipula* may be easily extended to configurations with several smart legal contracts. It is sufficient to consider configurations as consisting of sets of runtime contracts and to change the rule [Tick]. To illustrate the general case, let $\mathbb{C} = \texttt{C}_1(\mathbb{Q}_1\,,\,\ell_1\,,\,\Sigma_1\,,\,\Psi_1)$, $\cdots\,,\texttt{C}_n(\mathbb{Q}_n\,,\,\ell_n\,,\,\Sigma_n\,,\,\Psi_n)$. We define $\texttt{Events}(\mathbb{C}) \stackrel{\text{def}}{=} \Psi_1 \mid \cdots \mid \Psi_n$. The new rule [Tick] becomes

$$\frac{\texttt{Events}(\mathbb{C})\;,\;\mathbb{t}\nrightarrow}{\mathbb{C}\;,\;\mathbb{t}\longrightarrow\mathbb{C}\;,\;\mathbb{t}+1}\;[\textsc{Tick}+]$$

5. *Stipula* laws

Legal contracts written in natural language admit different writing styles for expressing the same binding legal relationship between parties. In this section we show that adopting a programming language instead of the natural language simplifies checking when two contracts are

legally equivalent. The operational semantics of Table 2 is indeed the base for defining a more coarse-grained semantics, that equates legal contracts that are syntactically different but a party using them cannot distinguish one from the other. We use a standard technique based on observations [28], where a party can differentiate two contracts if he can *observe different interactions* during its lifetime.

According to the intuition, a party can observe the following aspects of a contract's behaviour: the agreement she is going to sign, the dynamic set of permissions, prohibitions and obligations, and the assets she receives form the contract. Moreover, two contracts are legally equivalent if they involve the same parties observing the same interactions. Therefore, in *Stipula* the observations of contract are defined according to the following principles:

- the observation $(\overline{A},\ \overline{A}_i : \overline{v}_i\,^{i\in 1..n})$ observes the *agreement* that the parties are going to sign, that is who is taking the legal responsibility for which contract's role, and what are the terms of the contract, *i.e.* the initial values of the contract's fields.
- to observe the *permissions* or the *prohibitions*, we observe whether any party can invoke any functionality at a given time $\mathbb{t}$, *i.e.* whether any function call $A : \mathtt{f}(\overline{u})[\overline{v}]$ is possible at $\mathbb{t}$ or not;
- *obligations* are captured implicitly by shifting the observation at a specific point in time. Indeed, obligations are enforced in *Stipula* by scheduling an event that issues a corresponding penalty if the obligation has not been fulfilled at a given date. Therefore, observing an obligation corresponds to observing –in the future– the effects of executing the event that encodes the legal commitment, such as the issue of a sanction or the impossibility to do further actions.
- each party should finally observe whether at a give time $\mathbb{t}$ she can *receive* a value or an asset, *i.e.* whether $v \to A$ or $a \multimap A$ are possible at $\mathbb{t}$ or not.

Notice that, by focussing the contract's equivalence on the above observations, the contract's semantics can safely abstract away from the names of the contract's assets, fields and internal states. Moreover, the ordering of function invocations and assets transfers can be safely overlooked, as long as they are executed at the same global time. In particular, the system's clock needs not to be directly observed: by checking the set of permissions and prohibitions at any time units, and since only a contract's state change can modify the set of available permissions and prohibitions, we have that it is sufficient to observe whether a function can be executed before of after another function or an event, disregarding its precise execution time unit.

We will use the following notations:

- Let be $\alpha_1 = (\overline{A},\ \overline{A}_1{:}\overline{v}_1 \cdots \overline{A}_n{:}\overline{v}_n)$ and $\alpha_2 = (\overline{B},\ \overline{B}_1{:}\overline{w}_1 \cdots \overline{B}_n{:}\overline{w}_n)$ then we write $\alpha_1 \sim \alpha_2$ if $\overline{A}$ and $\overline{B}$ are equal up to reordering of sequences, and similarly for $\overline{A}_i : \overline{v}_i$ and $\overline{B}_{\kappa(i)} : \overline{w}_{\kappa(i)}$, where κ is a bijection (applying the same reordering of $\overline{B}_{\kappa(i)}$ to $\overline{w}_{\kappa(i)}$ we obtain $\overline{A}_i : \overline{v}_i$).
- Let be $\overset{\mu}{\Longrightarrow} \overset{\text{def}}{=} \Longrightarrow\!\!\overset{\mu}{\longrightarrow}\!\!\Longrightarrow$, where $\Longrightarrow$ stands for any number of $\longrightarrow$ transitions, possibly zero.

Definition 1 (Normative Equivalence). *A symmetric relation $\mathcal{R}$ is a bisimulation between two configurations at time $\mathbb{t}$, written $\mathbb{C}_1,\mathbb{t}\ \ \mathcal{R}\ \ \mathbb{C}_2,\mathbb{t}$, whenever*

1. *if $\mathbb{C}_1,\mathbb{t} \overset{\alpha}{\Longrightarrow} \mathbb{C}_1',\mathbb{t}$ then $\mathbb{C}_2,\mathbb{t} \overset{\alpha'}{\Longrightarrow} \mathbb{C}_2',\mathbb{t}$ for some α' such that $\alpha \sim \alpha'$ and $\mathbb{C}_1',\mathbb{t}\ \ \mathcal{R}\ \ \mathbb{C}_2',\mathbb{t}$;*
2. *if $\mathbb{C}_1,\mathbb{t} \overset{\mu_1}{\Longrightarrow} \cdots \overset{\mu_n}{\Longrightarrow} \mathbb{C}_1',\mathbb{t} \longrightarrow \mathbb{C}_1',\mathbb{t}+1$ then there exist $\mu_1' \cdots \mu_n'$ that is a permutation of $\mu_1 \cdots \mu_n$ such that $\mathbb{C}_2,\mathbb{t} \overset{\mu_1'}{\Longrightarrow} \cdots \overset{\mu_n'}{\Longrightarrow} \mathbb{C}_2',\mathbb{t} \longrightarrow \mathbb{C}_2',\mathbb{t}+1$ and $\mathbb{C}_1',\mathbb{t}+1\ \mathcal{R}\ \mathbb{C}_2',\mathbb{t}+1$.*

Let $\simeq$ be the largest bisimulation, called normative equivalence. *When the initial configurations of contracts $\mathtt{C}$ and $\mathtt{C}'$ are bisimilar, we simply write $\mathtt{C} \simeq \mathtt{C}'$.*

The name of the equivalence highlights the fact that it captures the normative elements of a contract. Indeed, being a symmetric relation, the normative equivalence compares both contracts' permissions and prohibitions: if $\mathtt{C}$ permits an action (*i.e.* exhibits an observation), then $\mathtt{C}'$ must permit the same action, and if $\mathtt{C}$ prohibits an action (*i.e.* does not exhibit a function call or an external communication), then also $\mathtt{C}'$ must not exhibit the corresponding observation. Moreover, the bisimulation game enforces a *transfer property* that shifts the time of observation to the future, so to capture and compare the changes of permissions/prohibitions and the (future) obligations. Observe that the equivalence abstracts away the ordering of the observations within the same time clock: if a party receives two messages in different order it might be due to delays of communications, rather to sensible differences in the contracts.

Nevertheless, specific orderings of function invocations are important in legal contracts, and the equivalence cannot overlook essential precedence constraints. For instance, the requirement that a function delivering a service can only be invoked after another specific function, say a payment. This is indeed the case for the bisimulation. To explain, consider the contract $\mathtt{C}$ with two functions $\mathtt{@Q\ A{:}f\{\_\} \Rightarrow @Q}$ and $\mathtt{@Q\ A'{:}g\{\_\} \Rightarrow @Q}$ and the contract $\mathtt{C}'$ with two functions $\mathtt{@Q\ A{:}f\{\_\} \Rightarrow @Q'}$ and $\mathtt{@Q'\ A'{:}g\{\_\} \Rightarrow @Q}$. In $\mathtt{C}$ the functions can be called in any order, while in $\mathtt{C}'$ the function $\mathtt{g}$ can be invoked only after $\mathtt{f}$. Accordingly, $\mathtt{C} \not\simeq \mathtt{C}'$ since there is a runtime configuration of $\mathtt{C}$ that exhibits the observation $\overset{A':\mathrm{g}}{\Longrightarrow}$, while it is not the case for the contract $\mathtt{C}'$, since at any time it can only exhibit $\overset{A:\mathrm{f}}{\Longrightarrow}\overset{A':\mathrm{g}}{\Longrightarrow}$.

The following theorem shows that the internal state of the contract is abstracted away by the normative equivalence, which only observes the external contract's behavior. The proof is omitted because it is standard.

Theorem 1 (Internal refactoring). *Let* $\mathtt{C}$ *and* $\mathtt{C}'$ *be two contracts that are equal up-to a bijective renaming of states. Then* $\mathtt{C} \simeq \mathtt{C}'$*. Similarly, for bijective renaming of assets, fields and contract names.*

The equivalence is also independent from *future* clock values. This allows us to garbage-collect events that cannot be triggered anymore because the time for their scheduling is already elapsed.

Theorem 2 (Time shift).

1. *If* $\mathbb{C}, \mathbb{t} \simeq \mathbb{C}', \mathbb{t}$ *and* $\mathbb{t} \leqslant \mathbb{t}'$*, then* $\mathbb{C}, \mathbb{t}' \simeq \mathbb{C}', \mathbb{t}'$.
2. *If* $\mathbb{t} < \mathbb{t}'$ *then* $\mathtt{C}(Q,\, \ell,\, \Sigma,\, \Psi \mid \mathbb{t} \gg \mathtt{@Q}\ \{\ S\ \} \Rightarrow \mathtt{@Q'})\ ,\ \mathbb{t}' \ \simeq\ \mathtt{C}(Q,\, \ell,\, \Sigma,\, \Psi)\ ,\ \mathbb{t}'$.

We finally put forward a set of algebraic laws formalizing that the ordering of communications can be safely overlooked, as long as they belong to the same time. The laws are defined over statements, therefore, let $\mathcal{C}[\ ]$ be a *context*, that is a contract that contains an hole where a statement may occur. We write $S \simeq S'$ if, for every context $\mathcal{C}[\ ]$, $\mathcal{C}[S] \simeq \mathcal{C}[S']$.

Theorem 3. *The following non-interference laws hold in* Stipula *(whenever they are applicable, we assume* $\mathtt{x}, \mathtt{h}, \mathtt{h}' \notin fv(E')$ *and* $\mathtt{x}', \mathtt{h}'', \mathtt{h}''' \notin fv(E)$*:*

$$
\begin{array}{rcl}
E \to \mathtt{A}\ \ E' \to \mathtt{A}' & \simeq & E' \to \mathtt{A}'\ \ E \to \mathtt{A} \\
E \to \mathtt{x}\ \ E' \to \mathtt{A} & \simeq & E' \to \mathtt{A}\ \ E \to \mathtt{x} \\
E \to \mathtt{x}\ \ E' \to \mathtt{x}' & \simeq & E' \to \mathtt{x}'\ \ E \to \mathtt{x} \\
E \multimap \mathtt{h}, \mathtt{A}\ \ E' \to \mathtt{A}' & \simeq & E' \to \mathtt{A}'\ \ E \multimap \mathtt{h}, \mathtt{A} \\
E \multimap \mathtt{h}, \mathtt{A}\ \ E' \to \mathtt{x}' & \simeq & E' \to \mathtt{x}'\ \ E \multimap \mathtt{h}, \mathtt{A} \\
E \multimap \mathtt{h}, \mathtt{h}'\ \ E' \to \mathtt{A} & \simeq & E' \to \mathtt{A}\ \ E \multimap \mathtt{h}, \mathtt{h}' \\
E \multimap \mathtt{h}, \mathtt{h}'\ \ E' \to \mathtt{x}' & \simeq & E' \to \mathtt{x}'\ \ E \multimap \mathtt{h}, \mathtt{h}' \\
E \multimap \mathtt{h}, \mathtt{A}\ \ E' \multimap \mathtt{h}'', \mathtt{A}' & \simeq & E' \multimap \mathtt{h}'', \mathtt{A}'\ \ E \multimap \mathtt{h}, \mathtt{A} \\
E \multimap \mathtt{h}, \mathtt{A}\ \ E' \multimap \mathtt{h}'', \mathtt{h}''' & \simeq & E' \multimap \mathtt{h}'', \mathtt{h}'''\ \ E \multimap \mathtt{h}, \mathtt{A} \\
E \multimap \mathtt{h}, \mathtt{h}'\ \ E' \multimap \mathtt{h}'', \mathtt{h}''' & \simeq & E' \multimap \mathtt{h}'', \mathtt{h}'''\ \ E \multimap \mathtt{h}, \mathtt{h}'
\end{array}
$$

Proof We prove the first equality. Let $S_1 = E \to \mathtt{A}\ \ E' \to \mathtt{A}'$ and $S_2 = E' \to \mathtt{A}'\ \ E \to \mathtt{A}$ and let the context $\mathcal{C}[\ ]$ be a contract that contains an hole where a statement may occur, then we need to prove that $\mathcal{C}[S_1] \simeq \mathcal{C}[S_2]$. Let be $\mathtt{C}_1 = \mathcal{C}[S_1]$ and $\mathtt{C}_2 = \mathcal{C}[S_2]$. Let also $\mathcal{C}_{(i)}[S_i] = \mathtt{C}_i(\mathbb{Q}, \ell, \_, \Psi[S_i])$, with $i = 1, 2$, be the runtime contract where the statement S_i occurs within a number of handlers of future events. We demonstrate that the symmetric closure of the following relation is a bisimulation:

$$\{\, (\mathtt{C}_1(\_, \varnothing, \_, \_), \mathbb{t}\ ,\ \mathtt{C}_2(\_, \varnothing, \_, \_), \mathbb{t})\,\} \bigcup \{\, (\mathcal{C}_{(1)}[S_1], \mathbb{t}'\ ,\ \mathcal{C}_{(2)}[S_2], \mathbb{t}')\ \mid\ \text{for every } \mathcal{C}_{(\cdot)}[\cdot] \text{ and } \mathbb{t}' \geqslant \mathbb{t}\,\}$$

Indeed, notice that the statement $\mathtt{E} \to \mathtt{A}\ \ \mathtt{E}' \to \mathtt{A}'$ can only contribute to the behavior of $\mathtt{C}$ with a couple of transitions during the evaluation of the body of a function or the evaluation of an event handler. Therefore the statement must be completely executed with the same time clock, possibly a number k of times due to the multiple function calls and event handlers that are executed during the same time clock.

Formally, if $\mathcal{C}_{(1)}[S_1], \mathbb{t}' \overset{\mu_1}{\Longrightarrow} \cdots \overset{\mu_n}{\Longrightarrow} \mathcal{C}'_{(1)}[S_1], \mathbb{t}' \longrightarrow \mathcal{C}'_{(1)}[S_1], \mathbb{t}' + 1$, then the sequence $\mu_1 \cdots \mu_n$ contains k occurrences of the pair $v \to A, v' \to A'$. Similarly, there exists $\mu'_1 \cdots \mu'_n$ and a configuration $\mathcal{C}'_{(2)}[S_2], \mathbb{t}'$ such that $\mathcal{C}_{(2)}[S_2], \mathbb{t}' \overset{\mu'_1}{\Longrightarrow} \cdots \overset{\mu'_n}{\Longrightarrow} \mathcal{C}'_{(2)}[S_2], \mathbb{t}' \longrightarrow \mathcal{C}'_{(2)}[S_2], \mathbb{t}' + 1$, where the sequence $\mu'_1 \cdots \mu'_n$ is identical to $\mu_1 \cdots \mu_n$ but for the k occurrences of the pair $v \to A, v' \to A'$ that has been swapped into $v' \to A', v \to A$. The argument also holds in the converse direction. This implies that $\mathcal{R}$ is a bisimulation, thus it is included in the normative equivalence $\simeq$, which is the largest bisimulation. □

6. The inference of types

The syntax of *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 legal practitioners. 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, in this section we present a type inference system that allows one to derive types of assets, fields and functions' arguments, and that can be used in the future to develop a user-friendly programming interface for *Stipula*. We show that the inferred types are sound with respect to the operational semantics (Subject Reduction), and that well typed contracts do not evolve to unsound states (Safety). More precisely, the Progress Theorem shows that well typed configurations just stuck on an attempt of doing an unsafe asset operation, an access to an uninitialized field or a division by 0. Even if this typing is quite simple, it prevents basic programming errors, and provides a basis to further develop stronger static analyses for safe asset usage.

Primitive types T are defined by the syntax

$$\textit{Primitive Types}\quad T ::= \quad \mathtt{real} \ \mid\ \mathtt{bool} \ \mid\ \mathtt{string} \ \mid\ \mathtt{time} \ \mid\ \mathtt{asset}$$

that mirrors the set of values: real numbers, booleans, strings, time values, and assets that include both currency and non-fungible asset values. 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 map fields and non-asset functions' arguments to type terms and Δ that map assets and assets functions' arguments to type terms. The notation $\Gamma[\mathtt{x} \mapsto \alpha]$, respectively $\Delta[\mathtt{h} \mapsto \alpha]$, stands for either the update or the extension of the environment, depending on whether $\mathtt{x}$, respectively $\mathtt{h}$, belongs to the domain of the environment. We recall that contract's names (fields, assets, function's parameters) have been assumed to be all pairwise distinct.
- *constraints* Υ, Υ', $\cdots$, which are conjunctions of equations $\alpha = \alpha'$;

[T-STRING] $\Gamma, \Delta \vdash s : \texttt{string}, \mathit{true}$ [T-TRUE] $\Gamma, \Delta \vdash \texttt{true} : \texttt{bool}, \mathit{true}$ [T-FALSE] $\Gamma, \Delta \vdash \texttt{false} : \texttt{bool}, \mathit{true}$ [T-TIME] $\Gamma, \Delta \vdash \mathbb{t} : \texttt{time}, \mathit{true}$

[T-NUMBER] $\Gamma, \Delta \vdash n : \texttt{real}, \mathit{true}$ [T-IASSET] $\Gamma, \Delta \vdash a : \texttt{asset}, \mathit{true}$ [T-NOW] $\Gamma, \Delta \vdash \texttt{now} : \texttt{time}, \mathit{true}$

[T-FIELD] $\Gamma, \Delta \vdash \texttt{x} : \Gamma(\texttt{x}), \mathit{true}$ [T-ASSET] $\Gamma, \Delta \vdash \texttt{h} : \Delta(\texttt{h}), \mathit{true}$ [T-SUBSUMPTION] $\dfrac{\Gamma, \Delta \vdash E : \texttt{asset}, \Upsilon}{\Gamma, \Delta \vdash E : \texttt{real}, \Upsilon}$

[T-ROP] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \Gamma, \Delta \vdash E' : \alpha', \Upsilon' \quad \Upsilon'' = (\alpha = \alpha') \wedge \Upsilon \wedge \Upsilon'}{\Gamma, \Delta \vdash E \,\texttt{rop}\, E' : \texttt{bool}, \Upsilon''}$ [T-BOP] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \Gamma, \Delta \vdash E' : \alpha', \Upsilon' \quad \Upsilon'' = (\alpha, \alpha' = \texttt{bool}) \wedge \Upsilon \wedge \Upsilon'}{\Gamma, \Delta \vdash E \,\texttt{bop}\, E' : \texttt{bool}, \Upsilon''}$

[T-AOP] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \Gamma, \Delta \vdash E' : \alpha', \Upsilon' \quad \Upsilon'' = (\alpha = \texttt{real}) \wedge (\alpha' = \texttt{real}) \wedge \Upsilon \wedge \Upsilon'}{\Gamma, \Delta \vdash E \,\texttt{aop}\, E' : \texttt{real}, \Upsilon''}$ [T-ASSET OP] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \Gamma, \Delta \vdash E' : \alpha', \Upsilon' \quad \Upsilon'' = (\alpha, \alpha' = \texttt{asset}) \wedge \Upsilon \wedge \Upsilon' \quad \texttt{op} \in \{+, -\}}{\Gamma, \Delta \vdash E \,\texttt{op}\, E' : \texttt{asset}, \Upsilon''}$

[T-ASSET MUL] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \Gamma, \Delta \vdash E' : \alpha', \Upsilon' \quad \Upsilon'' = (\alpha = \texttt{asset}) \wedge (\alpha' = \texttt{real}) \wedge \Upsilon \wedge \Upsilon'}{\Gamma, \Delta \vdash E \times E' : \texttt{asset}, \Upsilon''}$

[T-CONC] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \Gamma, \Delta \vdash E' : \alpha', \Upsilon' \quad \Upsilon'' = (\alpha, \alpha' = \texttt{string}) \wedge \Upsilon \wedge \Upsilon'}{\Gamma, \Delta \vdash E \mathbin{\hat{}} E' : \texttt{string}, \Upsilon''}$ [T-EXPTIME] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \Gamma, \Delta \vdash E' : \alpha', \Upsilon' \quad \Upsilon'' = (\alpha = \texttt{time}) \wedge (\alpha' = \texttt{real}) \wedge \Upsilon \wedge \Upsilon'}{\Gamma, \Delta \vdash E + E' : \texttt{time}, \Upsilon''}$

Table 3: Typing rules for Expressions, wehere $\texttt{rop}=\{<, >, <=, >=, ==\}$, $\texttt{bop}=\{\&\&, ||\}$, $\texttt{aop}=\{+, -, \times, /\}$.

[T-SEND] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon}{\Gamma, \Delta \vdash E \to \texttt{A} : \Upsilon}$ [T-ASEND] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \Upsilon' = (\alpha, \Delta(\texttt{h}) = \texttt{asset}) \wedge \Upsilon}{\Gamma, \Delta \vdash E \multimap \texttt{h}, \texttt{A} : \Upsilon}$ [T-UPDATE] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \alpha \neq \texttt{asset} \quad \Upsilon' = (\Gamma(\texttt{x}) = \alpha) \wedge \Upsilon}{\Gamma, \Delta \vdash E \to \texttt{x} : \Upsilon'}$

[T-AUPDATE] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \Upsilon' = (\alpha, \Delta(\texttt{h}), \Delta(\texttt{h}') = \texttt{asset})) \wedge \Upsilon}{\Gamma, \Delta \vdash E \multimap \texttt{h}, \texttt{h}' : \Upsilon'}$ [T-ZERO] $\Gamma, \Delta \vdash \_ : \mathit{true}$ [T-COMPSTM] $\dfrac{\Gamma, \Delta \vdash P : \Upsilon' \qquad \Gamma, \Delta \vdash S : \Upsilon''}{\Gamma, \Delta \vdash P\ S : \Upsilon' \wedge \Upsilon''}$

[T-COND] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \quad \Gamma, \Delta \vdash S : \Upsilon' \quad \Gamma, \Delta \vdash S' : \Upsilon'' \quad \Upsilon''' = (\alpha = \texttt{bool}) \wedge \Upsilon \wedge \Upsilon' \wedge \Upsilon''}{\Gamma, \Delta \vdash \texttt{if}\,(E)\,\{\,S\,\}\,\texttt{else}\,\{\,S'\,\} : \Upsilon'''}$ [T-EVENT] $\dfrac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Gamma, \Delta \vdash S : \Upsilon' \qquad \Gamma, \Delta \vdash W : \Upsilon'' \qquad \Upsilon'' = (\alpha = \texttt{time}) \wedge \Upsilon \wedge \Upsilon' \wedge \Upsilon''}{\Gamma, \Delta \vdash E \gg \texttt{@Q}\{\,S\,\} \Rightarrow \texttt{@Q}'\ W : \Upsilon'''}$

[T-AGREEMENT] $\dfrac{\bigcup_{i\in 1..n} \overline{\texttt{A}_i} \subseteq \overline{\texttt{A}} \quad \bigcup_{i\in 1..n} \overline{\texttt{x}_i} \subseteq \overline{\texttt{x}} \quad \bigcap_{i\in 1..n} \overline{\texttt{x}_i} = \varnothing}{\vdash_{\overline{\texttt{x}}} \texttt{agreement}\ (\overline{\texttt{A}})\{\overline{\texttt{A}_1} : \overline{\texttt{x}_1}\ \cdots\ \overline{\texttt{A}_n} : \overline{\texttt{x}_n}\}}$

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

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

Table 4: Typing rules for *Stipula*

- *judgments* $\Gamma, \Delta \vdash E : \alpha, \Upsilon$ for expressions and $\Gamma, \Delta \vdash S : \Upsilon$ for statements.

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, the constraints are solved by means of a unification technique and the type variables are replaced by the resulting values. According to this technique, if a constraint Υ has a *unifier*, *i.e.* a substitution σ replacing the type variables with type terms such that all the equations are identities, then there exists a *most general unifier* σ_{mgu} such that $\sigma = \sigma_{mgu} \circ \sigma'$, for some substitution σ' (see [30] for details of the technique). We will denote the most general unifier σ_{mgu} of a constraint Υ with $\Upsilon \Vdash \sigma_{mgu}$. We also say that σ is a *ground unifier* of Υ if it is a unifier and it maps type variables to primitive types. With an abuse of notation, we also write $\sigma(\Gamma)$ to denote the environment that, for each $\mathtt{x} \in dom(\Gamma)$, associates $\sigma(\Gamma(\mathtt{x}))$ to $\mathtt{x}$; similarly for $\sigma(\Delta)$.

The type inference system is given in Table 3 and 4, and the most relevant rules are commented below. The first set of rules infers the types of constants without any constraint in every environment Γ and Δ. Rule [T-SUBSUMPTION] promotes assets to be `real`, so that assets, when used within expressions, are considered as reals, according to the semantics that evaluates expressions by thaking their raw value. Rules for boolean operations (`bop`), relational operations (`rop`), string, time and arithmetic operations (`aop`) between standard values are straightforward and generate suitable constraints on the type of operands. The rules [T-ASSET OP] and [T-ASSET MUL] allow arithmetic operations also involving assets. This simple rules prevents typing errors like $\mathtt{5D} + \text{"}\mathtt{hello}\text{"}$, but it is not strong enough to prevent all the unsafe usage of assets. Indeed, expressions like $\mathtt{5D} + \mathit{1234}T$ or $\mathtt{5D} - \mathtt{10D}$ are well typed even if they result in runtime errors. The first kind of errors could be amended by introducing two different asset types, one for currency and a different one for tokens. On the other hand, statically preventing negative amounts of assets is more difficult, since the runtime value of an expression like $\mathtt{wallet} - \mathtt{10D}$ can be hardly approximated at static time. Therefore, we prefer to present here a simple type system and postpone to future work stronger typing disciplines for safe resource usage.

As usual, statements have no type: the typing system just returns constraints. For instance, rule [T-UPDATE] deals with the assignment to a field or a parameter by 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 $\text{"}\mathtt{hello}\text{"} \rightarrow \mathtt{x}$ in the environments Γ, Δ returns the constraint $\Gamma(\mathtt{x}) = \mathtt{string}$. Moreover, since the rule [T-UPDATE] requires the expression E to have a non-asset type, the statement $\mathtt{10D} \rightarrow \mathtt{x}$ is well typed by means of the combination of the rules [T-SUBSUMPTION] and [T-UPDATE].

The rules [T-ASEND] and [T-AUPDATE] type asset transfers. They impose that the expression E has type `asset`, whose raw value corresponds to a quantity to be withdrawn form the asset $\mathtt{h}$ and sent to $\mathtt{A}$ or, respectively, added to the asset $\mathtt{h}'$. As before, nothing ensures that a well typed transfer statement corresponds to a safe asset operation: for instance the unsafe statements $\mathtt{h1} \multimap \mathtt{h}, \mathtt{A}$, where $\mathtt{h1}$ holds a larger amount of currency than $\mathtt{h}$, and $\mathit{1234}T \multimap \mathtt{h}, \mathtt{h}'$, with non empty asset $\mathtt{h}'$, are well typed but their semantics results in a stuck contract execution. The remaining rules are standard.

The typing of a contract is shown in Table 4 and relies on three other kinds of judgements. $\vdash_{\overline{\mathtt{x}}} \mathtt{G}$ stands for the syntactic check that the contract's agreement $\mathtt{G}$ is well formed (see rule [T-AGREEMENT]). 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 (see rule [T-FUNCTION]). As discussed above, the judgement $\bigwedge_{i \in 1..n} \Upsilon_i \Vdash \sigma$ returns the most general unifier σ satisfying the constraints $\bigwedge_{i \in 1..n} \Upsilon_i$.

The typing of a *Stipula* contract is then given in rule [T-PROGRAM], that adopts the notation Γ, Γ' to denote the environment obtained by the concatenation of two disjoint environments Γ and Γ', and similarly for Δ, Δ'. In the rule fresh type variables are associated to contract's fields and functions' parameters, both assets and non assets. These associations are recorded in the type environments $\Gamma\Gamma_1 \cdots \Gamma_n$ and $\Delta\Delta_1 \cdots \Delta_n$ (remind that all contract's names have been assumed to be different; therefore Γ, Γ_i, Δ, Δ_i have disjoint domains). The whole contract has type $[\sigma(\Gamma\Gamma_1 \cdots \Gamma_n), \sigma(\Delta\Delta_1 \cdots \Delta_n)]$, where σ is the most general unifier. As an example, the

$$\frac{}{\Gamma, \Delta \vdash \_}\ [\text{T-ZERO-RUN}] \qquad \frac{\Gamma, \Delta \vdash S : true \qquad \Gamma, \Delta \vdash W : true}{\Gamma, \Delta \vdash S\ W \Rrightarrow @\mathtt{Q}}\ [\text{T-RUN}] \qquad \frac{\left(\Gamma, \Delta \vdash W_i : true\right)^{i \in 1..n}}{\Gamma, \Delta \vdash W_1 | \cdots | W_n}\ [\text{T-SCHED}]$$

$$\frac{\begin{array}{c} dom(\ell) = \{\overline{\mathtt{A}}, \overline{\mathtt{x}}, \overline{\mathtt{h}}\} \\ \left(\varnothing, \varnothing \vdash \ell(\mathtt{y}) : T_{\mathtt{y}}, true\right)^{\mathtt{y} \in \overline{\mathtt{x}}} \quad \left(\varnothing, \varnothing \vdash \ell(\mathtt{k}) : \mathtt{asset}, true\right)^{\mathtt{k} \in \overline{\mathtt{h}}} \\ \bigwedge_{\mathtt{y} \in \overline{\mathtt{x}}} (\Gamma(\mathtt{y}) = T_{\mathtt{y}}) \wedge \bigwedge_{\mathtt{k} \in \overline{\mathtt{h}}} (\Delta(\mathtt{k}) = \mathtt{asset}) \Vdash \sigma \end{array}}{\Gamma, \Delta \vdash \ell : \sigma(\Gamma), \sigma(\Delta)}\ [\text{T-MEMORY}]$$

$$\frac{\begin{array}{c} \vdash \mathtt{stipula\ C}\ \{...\} : [\Gamma, \Delta] \qquad \Gamma, \Delta \vdash \ell : \Gamma', \Delta' \\ \Gamma', \Delta' \vdash \Sigma \qquad \Gamma', \Delta' \vdash \Psi \end{array}}{\Gamma, \Delta \vdash \mathtt{C}(\mathtt{Q}\ ,\ \ell\ ,\ \Sigma\ ,\ \Psi)\ \mathtt{,}\ \mathbb{t}}\ [\text{T-RUNTIME}]$$

Table 5: Typing of Runtime Configurations

type inference of the `Bike_Rental` contract in Figure 1 collects the following constraints: $\Gamma(\mathtt{x}) = \Gamma(\mathtt{code})$ form the typing of the function `offer`; $\Delta(\mathtt{h}) = \Gamma(\mathtt{cost})$, $\Delta(\mathtt{h}) = \Delta(\mathtt{wallet}) = \mathtt{asset}$, $\Gamma(\mathtt{rentingTime}) = \mathtt{real}$ form the typing of the function `pay`, and form the typing of the function `verdict` infers that $\Gamma(\mathtt{y}) = \mathtt{real}$ and $\Gamma(\mathtt{x})$ is not an asset.,

In order to prove the soundness of the type inference system, we also need to type runtime terms with rules in Table 5. The key point is the typing of a memory ℓ that records the mapping from a contract's name to a value. The rule [T-MEMORY] defines the judgement $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$, that guarantees that the types of the values that ℓ maps to contract's names agree with the types assumed in the environments Γ, Δ. For instance, if $\Gamma, \Delta \vdash [\mathtt{x} \mapsto u, \mathtt{h} \mapsto v] : \Gamma', \Delta'$ then the constant values u, v have primitive types, say T_u and T_v, and it is possible to unify these types with the values bound to `x` and `h` by Γ and Δ. This means that `x` and `h` must belong to the domains of Γ and Δ, respectively. The rule [T-RUNTIME] states that the runtime configuration $\mathtt{C}(\mathtt{Q}\ ,\ \ell\ ,\ \Sigma\ ,\ \Psi)\ \mathtt{,}\ \mathbb{t}$ is well typed under the environments Γ, Δ whenever the source contract `C` is well typed, and the runtime memory ℓ agrees with the the contract type $[\Gamma, \Delta]$ (that satisfies all the type constraints collected by the inference process). The runtime terms Σ and Ψ must also be well typed using the updated environments Γ', Δ'.

Lemma 1 (Substitution Lemma). *Let σ be a unifier of Υ.*

- *If $\Gamma, \Delta \vdash S : \Upsilon$ then $\sigma(\Gamma), \sigma(\Delta) \vdash S : true$;*
- *If $\Gamma, \Delta \vdash W : \Upsilon$ then $\sigma(\Gamma), \sigma(\Delta) \vdash W : true$;*
- *If $\Gamma, \Delta \vdash E : \alpha, \Upsilon$ then $\sigma(\Gamma), \sigma(\Delta) \vdash E : \sigma(\alpha), true$.*

Observe that the semantics of *Stipula* is *open*, in the sense that it relies on actions that input values from the external context. Therefore, in order to prove the type preservation, it is essential to impose that the input values agree with the type constraints of the contract. We write $\Gamma, \Delta \vdash \mathtt{C}, \mu$ whenever:

- if $\mu = (\overline{A}, \overline{A_i} : \overline{v_i}^{\,i \in 1..n})$, then $\mathtt{agreement}\ (\overline{\mathtt{A}})\{\overline{\mathtt{A}_1} : \overline{\mathtt{x}_1}\ \cdots\ \overline{\mathtt{A}_n} : \overline{\mathtt{x}_n}\}$ is in `C` and $\Gamma, \Delta \vdash [\overline{\mathtt{x}_i} \mapsto \overline{v_i}^{\,i \in 1..n}, \overline{\mathtt{h}} \mapsto \overline{0}] : \Gamma', \Delta'$, for some Γ', Δ';
- if $\mu = A : \mathtt{f}(\overline{u})[\overline{v}]$ then $@\mathtt{Q}\ \mathtt{A} : \mathtt{f}(\overline{\mathtt{y}})\ [\overline{\mathtt{k}}]\ (E)\ \{\, S\ W \,\} \Rrightarrow @\mathtt{Q}'$ is in `C` and $\Gamma, \Delta \vdash [\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}] : \Gamma', \Delta'$, for some Γ', Δ';
- if $\mu = \varepsilon$ or $\mu = v \to A$ or $\mu = v \multimap A$ then $\Gamma, \Delta \vdash \mathtt{C}, \mu$ always holds.

In particular, we notice that $\Gamma, \Delta \vdash \mathtt{C}, A : \mathtt{f}(\overline{u})[\overline{v}]$ implies that $\overline{v}$ is a tuple of asset values. This follows by [T-MEMORY] that requires $\overline{v}$ have asset type.

Proposition 1. *Let $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$ and $\Gamma, \Delta \vdash [\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}] : \Gamma'', \Delta''$. Then $\Gamma, \Delta \vdash \ell[\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}] : \Gamma'\Gamma'', \Delta'\Delta''$.*

Proof If $\overline{\mathtt{y}}, \overline{\mathtt{k}} \notin dom(\ell)$, it is trivial. If $\mathtt{x} \in \overline{\mathtt{y}}$ and $\mathtt{x} \in dom(\ell)$ then the hypotheses guarantee that either (i) $\Gamma'(\mathtt{x})$ is a type variable or (ii) it is a primitive type. In case of (i), then there is a unifier that replaces $\Gamma'(\mathtt{x})$ with $\Gamma''(\mathtt{x})$. In case of (ii), $\mathtt{x}$ is overwritten by $[\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}]$. There are two subcases either $(ii.a)$ the primitive types of $\Gamma'(\mathtt{x})$ and $\Gamma''(\mathtt{x})$ are equal or $(ii.b)$ not. In the subcase $(ii.a)$, the unifier is the identity on $\Gamma'(\mathtt{x})$ and $\Gamma''(\mathtt{x})$. In the subcase $(ii.b)$, by the hypotheses, $\Gamma(\mathtt{x})$ must be a type variable; therefore the unifier for $\ell[\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}]$ has to map $\Gamma(\mathtt{x})$ to $\Gamma''(\mathtt{x})$. Similarly for $\mathtt{h} \in \overline{\mathtt{k}}$. □

Proposition 2. *Let* $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$.

(1) Let be $\Gamma', \Delta' \vdash E : T, true$. *If* $[\![E]\!]_\ell$ *is defined then* $\Gamma', \Delta' \vdash [\![E]\!]_\ell : T', true$, *where* $T' = T$ *if* $T \neq \mathtt{asset}$, *while* $T' = \mathtt{real}$ *if* $T = \mathtt{asset}$. *Moreover, if* $[\![E]\!]^{\mathtt{a}}_\ell$ *is defined then* $\Gamma', \Delta' \vdash [\![E]\!]^{\mathtt{a}}_\ell : \mathtt{asset}, true$.

(2) If $\Gamma', \Delta' \vdash W : true$ *and* $[\![W\{^{\mathtt{t}}/_{\mathtt{now}}\}]\!]_\ell$ *is defined then* $\Gamma', \Delta' \vdash [\![W\{^{\mathtt{t}}/_{\mathtt{now}}\}]\!]_\ell : true$.

Proof As regards (1), since $[\![E]\!]_\ell$ is defined and $\Gamma', \Delta' \vdash E : T, true$, then, by induction on the definition of $[\![E]\!]_\ell$, it is easy to verify that, if $T \neq \mathtt{asset}$, $[\![\cdot]\!]_\ell$ preserves typing. In particular, the evaluation that replaces a name $\mathtt{x}$ with $\ell(\mathtt{x})$, because of the hypothesis $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$. If $T = \mathtt{asset}$, then from $\Gamma', \Delta' \vdash E : \mathtt{asset}, true$ we have that $[\![E]\!]_\ell$ has type $\mathtt{real}$ and, by an application of the subsumption rule we also have $\Gamma', \Delta' \vdash E : \mathtt{real}, true$ The case $[\![E]\!]^{\mathtt{a}}_\ell$ comes by the fact that it corresponds, by definition, to an asset value.

As regards (2) observe that $\Gamma', \Delta' \vdash W\{^{\mathtt{t}}/_{\mathtt{now}}\} : true$ is well-typed (replace every instance of [T-NOW] with an instance of [T-TIME]). Then the proof is similar to the subcase (1). □

Theorem 4 (Subject reduction). *Let* $\Gamma, \Delta \vdash \mathbb{C}\ \mathtt{,}\ \mathbb{t}$ *and* $\mathbb{C}, \mathbb{t} \xrightarrow{\mu} \mathbb{C}', \mathbb{t}'$ *with* $\Gamma, \Delta \vdash \mathtt{C}, \mu$. *Then* $\Gamma, \Delta \vdash \mathbb{C}'\ \mathtt{,}\ \mathbb{t}'$.

Proof The proof comes by a case analysis on the transition rule:

(*Agree*) Then $\mathbb{C} = \mathtt{C}(\_, \varnothing, \_, \_)$ and $\mathbb{C}' = \mathtt{C}(\mathtt{Q}, [\overline{\mathtt{A}} \mapsto \overline{A}, \overline{\mathtt{x}_i} \mapsto \overline{v_i}^{\,i \in 1..n}, \overline{\mathtt{h}} \mapsto \overline{0}], \_, \_)$. The thesis comes from $\Gamma, \Delta \vdash [\overline{\mathtt{A}} \mapsto \overline{A}, \overline{\mathtt{x}_i} \mapsto \overline{v_i}^{\,i \in 1..n}, \overline{\mathtt{h}} \mapsto \overline{0}] : \Gamma', \Delta'$, for some Γ', Δ', which is a consequence of the hypothesis $\Gamma, \Delta \vdash \mathtt{C}, (\overline{A}, \overline{A_i} : \overline{v_i}^{\,i \in 1..n})$.

(*Function*) Then $\mu = A : \mathtt{f}(\overline{u})[\overline{v}]$, $\mathbb{C} = \mathtt{C}(\mathtt{Q}, \ell, \_, \Psi)$ and $\mathbb{C}' = \mathtt{C}(\mathtt{Q}, \ell', S\,W \Rrightarrow \mathtt{@Q}', \Psi)$. Let $\mathtt{@Q\ A} : \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}](E)\,\{\,S\,W\,\} \Rrightarrow \mathtt{@Q}'$ be the code of $\mathtt{f}$ in $\mathtt{C}$.

From the hypothesis $\Gamma, \Delta \vdash \mathbb{C}\ \mathtt{,}\ \mathbb{t}$ and [T-RUNTIME], there are Γ', Δ' such that $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$, and $\Gamma', \Delta' \vdash \Psi$. From the hypothesis $\Gamma, \Delta \vdash \mathtt{C}, A : \mathtt{f}(\overline{u})[\overline{v}]$ we know $\Gamma, \Delta \vdash [\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}] : \Gamma'', \Delta''$ for some Γ'', Δ''. Then, by Proposition 1, we have $\Gamma, \Delta \vdash \ell[\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}] : \Gamma'\Gamma'', \Delta'\Delta''$. Let $\Gamma''' = \Gamma'\Gamma''$ and $\Delta''' = \Delta'\Delta''$.

Next, observe that, by [T-MEMORY], Γ''', Δ''' are obtained from Γ, Δ by means of a substitution σ; in turn, by [T-RUNTIME] applied to $\Gamma, \Delta \vdash \mathbb{C}\ \mathtt{,}\ \mathbb{t}$, the environments Γ, Δ are obtained through the most general unifier σ' for the constraint Υ of the whole program. Therefore $\sigma' \circ \sigma$ is a unifier of every constraint Υ' in the instance of [T-FUNCTION] applied to $\mathtt{f}$. This observation, together with the Substitution Lemma applied to the premises of [T-FUNCTION], allows us to derive $\Gamma''', \Delta''' \vdash S : true$ and $\Gamma', \Delta' \vdash W : true$. By this last judgment, observing that the names in W do not contain name parameters $\overline{\mathtt{y}}, \overline{\mathtt{k}}$, we derive $\Gamma''', \Delta''' \vdash W : true$. Hence, by applying [T-RUN] we obtain $\Gamma''', \Delta''' \vdash S\ W \Rrightarrow @Q'$; we conclude by applying [T-RUNTIME] to this judgment and to $\Gamma''', \Delta''' \vdash \Psi$, which follows by weakening the corresponding premise of $\Gamma, \Delta \vdash \mathbb{C}\ \mathtt{,}\ \mathbb{t}$ (names in Ψ do not contain $\overline{\mathtt{y}}, \overline{\mathtt{k}}$).

(*State Change*) In this case, $\mathbb{C} = \mathtt{C}(\mathtt{Q}, \ell, \_\,W \Rrightarrow \mathtt{@Q}', \Psi)$ and $\mathbb{C}' = \mathtt{C}(\mathtt{Q}', \ell, \_, [\![W\{^{\mathtt{t}}/_{\mathtt{now}}\}]\!]_\ell \mid \Psi)$. The hypothesis implies in particular $\Gamma', \Delta' \vdash W : true$ and $\Gamma', \Delta' \vdash \Psi$, for some Γ', Δ' such that $\Gamma, \Delta \vdash \ell\Gamma', \Delta'$. Since $[\![W\{^{\mathtt{t}}/_{\mathtt{now}}\}]\!]_\ell$ is well-defined, by $\mathbb{C}, \mathbb{t} \xrightarrow{\mu} \mathbb{C}', \mathbb{t}'$, then it is sufficient to show that also $\Gamma', \Delta' \vdash [\![W\{^{\mathtt{t}}/_{\mathtt{now}}\}]\!]_\ell : true$, which follows by Proposition 2(2).

(*Event Match*) $\mathbb{C} = \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,\_\,,\,\Psi)$ and $\mathbb{C}' = \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,S \Rightarrow \mathtt{@Q}'\,,\,\Psi')$ with $\Psi = \mathbb{t} \gg \mathtt{@Q}\ \{\ S\ \} \Rightarrow \mathtt{@Q}' \mid \Psi'$. From the hypothesis there exist Γ', Δ' such that $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$ and $\Gamma', \Delta' \vdash \Psi$, hence $\Gamma', \Delta' \vdash \mathbb{t} \gg \mathtt{@Q}\ \{\ S\ \} \Rightarrow \mathtt{@Q}' : \mathit{true}$ and $\Gamma', \Delta' \vdash \Psi'$. Therefore $\Gamma', \Delta' \vdash S : \mathit{true}$ is also derivable, then $\Gamma'\Delta' \vdash S \Rightarrow \mathtt{@Q}'$ by [T-Run] and we conclude by [T-Runtime].

(*Tick*) This case is trivial.

(*Value Send*) $\mathbb{C} = \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,E \to \mathtt{A}\ \Sigma\,,\,\Psi)$ then from the hypothesis we have that there exist Γ', Δ' such that $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$ and $\Gamma', \Delta' \vdash E \to \mathtt{A}\ \Sigma : \mathit{true}$, which means in particular $\Gamma', \Delta' \vdash \Sigma : \mathit{true}$, that is enough to conclude $\Gamma, \Delta \vdash \mathbb{C}', \mathbb{t}'$.

(*Asset Send*) $\mathbb{C} = \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,E \multimap \mathtt{h}, \mathtt{A}\ \Sigma\,,\,\Psi)$. As before we from the hypothesis we have $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$, $\Gamma', \Delta' \vdash \Sigma : \mathit{true}$ and $\Gamma', \Delta' \vdash E \multimap \mathtt{h}, \mathtt{A} : \mathit{true}$. The last judgement comes form $\Gamma', \Delta' \vdash E : \mathtt{asset}, \mathit{true}$ and $\Delta'(\mathtt{h}) = \mathtt{asset}$. From the hypothesis of [Asset Send] we also have that $[\![E]\!]^{\mathtt{a}}_{\ell} = a$, $[\![\mathtt{h} - a]\!]^{\mathtt{a}}_{\ell} = a'$, then $\Gamma', \Delta' \vdash a' : \mathtt{asset}, \mathit{true}$, then $\Gamma, \Delta \vdash \ell[\mathtt{h} \mapsto a'] : \Gamma', \Delta'$, which is sufficient to conclude.

(*Field Update*) $\mathbb{C} = \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,E \to \mathtt{x}\ \Sigma\,,\,\Psi)$. As before, from the hypothesis, we have $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$, and $\Gamma', \Delta' \vdash \Sigma : \mathit{true}$ and $\Gamma', \Delta' \vdash E \to \mathtt{x} : \mathit{true}$, which comes form $\Gamma', \Delta' \vdash E : \Gamma'(\mathtt{x}), \mathit{true}$, $\Gamma'(\mathtt{x}) \neq \mathtt{asset}$ and $[\![E]\!]_{\ell} = v$. By Proposition 2(1), $\Gamma', \Delta' \vdash v : \Gamma'(\mathtt{x}), \mathit{true}$, Then $\Gamma, \Delta \vdash \ell[\mathtt{x} \mapsto v] : \Gamma', \Delta'$, which is sufficient to conclude.

(*Asset Update*) $\mathbb{C} = \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,E \multimap \mathtt{h}, \mathtt{h}'\ \Sigma\,,\,\Psi)$. By the hypothesis we have $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$ and $\Gamma', \Delta' \vdash \Sigma : \mathit{true}$ and $\Gamma', \Delta' \vdash E \multimap \mathtt{h}, \mathtt{h}' : \mathit{true}$, which comes form $\Gamma', \Delta' \vdash E : \mathtt{asset}, \mathit{true}$ and $\Delta'(\mathtt{h}) = \Delta'(\mathtt{h}') = \mathtt{asset}$. From the hypothesis of [Asset-Update] we also have that $[\![E]\!]^{\mathtt{a}}_{\ell} = a$, $[\![\mathtt{h} - a]\!]^{\mathtt{a}}_{\ell} = a'$ and $[\![\mathtt{h}' + a]\!]^{\mathtt{a}}_{\ell} = a''$. Therefore we have $\Gamma', \Delta' \vdash a' : \mathtt{asset}, \mathit{true}$ and $\Gamma', \Delta' \vdash a'' : \mathtt{asset}, \mathit{true}$. Hence $\Gamma, \Delta \vdash \ell[\mathtt{h} \mapsto a', \mathtt{h}' \mapsto a''] : \Gamma', \Delta'$, which is sufficient to conclude.

(*Cond True*) $\mathbb{C} = \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,(\mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S'\,\}\ \Sigma\,,\,\Psi)$, and from the hypothesis we have in particular $\Gamma', \Delta' \vdash \mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S'\,\} : \mathit{true}$, which comes from $\Gamma', \Delta' \vdash E : \mathtt{bool}, \mathit{true}$, $\Gamma', \Delta' \vdash S : \mathit{true}$ and $\Gamma', \Delta' \vdash S' : \mathit{true}$. Then from $\Gamma', \Delta' \vdash S : \mathit{true}$ and the other judgements that comes from the hypothesis $\Gamma, \Delta \vdash \mathbb{C}, \mathbb{t}$, we have $\Gamma, \Delta \vdash \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,S\ \Sigma\,,\,\Psi)\ \mathtt{,}\ \mathbb{t}$ as desired.

(*Cond False*) This case is analogous to the previous one. □

Theorem 5 (Progress). *Let* $\Gamma, \Delta \vdash \mathbb{C}\ \mathtt{,}\ \mathbb{t}$. *Then*

1. *either there are* $\mathbb{C}', \mathbb{t}', \mu$ *such that* $\Gamma, \Delta \vdash \mathtt{C}, \mu$ *and* $\mathbb{C}\ \mathtt{,}\ \mathbb{t} \xrightarrow{\mu} \mathbb{C}', \mathbb{t}'$,
2. *or* $\mathbb{C} = \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,\Sigma\,,\,\Psi) \nrightarrow$ *because* Σ *has either an unsafe asset operation or an access to an uninitialized field or a division by 0.*

Proof Let $\mathbb{C} = \mathtt{C}(\mathtt{Q}\,,\,\ell\,,\,\Sigma\,,\,\Psi)$. By [T-runtime], $\Gamma, \Delta \vdash \ell : \Gamma', \Delta'$ and $\Gamma', \Delta' \vdash \Sigma$ and $\Gamma', \Delta' \vdash \Psi$. We proceed by a case analysis on $\mathtt{Q}$, ℓ, Σ and Ψ.

($\mathtt{Q} = \_$) In this case there exist $\overline{v_1}, \cdots, \overline{v_n}$ such that $\mu = (\overline{A}, \overline{A_i} : \overline{v_i}^{\,i \in 1..n})$ with $\Gamma, \Delta \vdash \mathtt{C}, \mu$. Then case 1. holds by applying [Agree].

($\Sigma = \_$) There are several possibilities.

- (a) If $\Psi = \mathbb{t} \gg \mathtt{@Q}\ \{\ S\ \} \Rightarrow \mathtt{@Q}' \mid \Psi'$ then case 1. holds by [Event Match].
- (b) $\Psi\ \mathtt{,}\ \mathbb{t} \nrightarrow$ and $\mathtt{@Q}\ \mathtt{A} : \mathtt{f}\,(\overline{\mathtt{y}})\ [\overline{\mathtt{k}}]\ (E)\,\{\,S\,W\,\} \Rightarrow \mathtt{@Q}' \in \mathtt{C}$ and there is $\mu = A : \mathtt{f}(\overline{u})[\overline{v}]$ such that $\Gamma, \Delta \vdash \mathtt{C}, \mu$. There are two subcases. The first one is when $[\![E]\!]_{\ell'}$ is defined and, when $[\![E]\!]_{\ell'} = \mathit{true}$, we have $\ell' = \ell[\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}]$. Therefore case 1. holds by [Function]. The second subcase is when $[\![E]\!]_{\ell'}$ is undefined, either because E has a name that does not belong to $\mathit{dom}(\ell)$ or because $[\![E]\!]_{\ell'}$ contains an undefined/unsafe asset operation or a division by 0. We are in case 2.

(c) Ψ , $\mathbb{t} \nrightarrow$ and no function applies, then case 1. holds by [Tick].

($\Sigma = \_W$) If $[\![W\{^{\mathbb{t}}/_{\mathtt{now}}\}]\!]_\ell$ is well defined, therefore rule [State Change] applies and case 1. holds. Otherwise $[\![W\{^{\mathbb{t}}/_{\mathtt{now}}\}]\!]_\ell$ is not defined. In this case by the hypotheses $\Gamma', \Delta' \vdash \Psi$, we derive $\Gamma', \Delta' \vdash W : \mathit{true}$ and, therefore, $\Gamma, \Delta \vdash W\{^{\mathbb{t}}/_{\mathtt{now}}\} : \mathit{true}$. Then $W\{^{\mathbb{t}}/_{\mathtt{now}}\}$ either contains a name that does not belong to $\mathit{dom}(\ell)$ or a division by 0. Therefore case 2. holds.

($\Sigma = E \to \mathtt{A}\ \Sigma'$) If $[\![E]\!]_\ell$ is defined, rule [Value-Send] applies and case 1. holds, Otherwise, $[\![E]\!]_\ell$ is undefined. From $\Gamma', \Delta' \vdash \Sigma$ we have $\Gamma', \Delta' \vdash E : T, \mathit{true}$. Then if $[\![E]\!]_\ell$ is undefined, it means that E either contains a name that does not belong to $\mathit{dom}(\ell)$ or an undefined operation; henceforth case 2. holds. We notice that $\mathtt{A} \in \mathit{dom}(\ell)$ holds because we constrained *Stipula* syntax such that party names occurring in the code belong to the set of parties included in the agreement.

($\Sigma = E \to \mathtt{x}\ \Sigma'$) The case is similar to the one above, where [Field-Update] is used when $[\![E]\!]_\ell$ is defined.

($\Sigma = E \multimap \mathtt{h}, \mathtt{A}\ \Sigma']$) As before, we have that $\mathtt{A} \in \mathit{dom}(\ell)$ holds because of *Stipula* syntax constraints. From $\Gamma', \Delta' \vdash \Sigma$ we have $\Gamma', \Delta' \vdash E : \mathtt{asset}, \mathit{true}$ and $\Delta'(\mathtt{h}) = \mathtt{asset}$. If $[\![E]\!]^{\mathtt{a}}_\ell$ and $[\![\mathtt{h} - [\![E]\!]^{\mathtt{a}}_\ell]\!]^{\mathtt{a}}_\ell$ are both defined, then [Asset-Send] applies and we are in case 1. Otherwise $[\![E]\!]_{\ell'}$ is undefined and we are in case 2 by arguments similar to the previous cases.

($\Sigma = E \multimap \mathtt{h}, \mathtt{h}'\ \Sigma'$) Similar to the foregoing case.

($\Sigma = \mathtt{if}\ (E)\ \{\ S\ \}\ \mathtt{else}\ \{\ S'\ \}\ \Sigma'$) If $[\![E]\!]_\ell$ is either *true* or *false* then either [Cond-true] or [Cond-false] applies, thus case 1. holds. Otherwise it is an erroneous configuration and we are in case 2. □

An easy consequence of Subject Reduction and Progress is:

Corollary 1 (Safety). *Let* $\vdash \mathtt{stipula\ C\ \{\ assets\ \overline{h}\ fields\ \overline{x}\ G\ F}_1 \cdots \mathtt{F}_n\ \} : [\Gamma, \Delta]$ *and* $\mathtt{C}(\_, \varnothing, \_, \_)$, $\mathbb{t} \stackrel{\overline{\mu}}{\Longrightarrow} \mathtt{C}(\mathtt{Q}, \ell, \Sigma, \Psi)$, $\mathbb{t}'$ *with* $\Gamma, \Delta \vdash \mathtt{C}, \mu$ *for all* $\mu \in \overline{\mu}$. *Then for every* $\mathtt{h} \in \mathit{dom}(\ell)$, $\ell(\mathtt{h}) \geqslant 0$.

7. A *Stipula* prototype

Stipula has been prototyped, therefore one can experiment enforcement of contractual conditions, traceability, and outcome certainty. The prototype is a `Java` application that is available on the github website of the project, together with a number of sample contracts [12]. The development has taken three months and ∼3000 lines of `Java` code. We have committed to the `ANTLR` tool [29]; below we discuss the main issues we found.

Type inference. `ANTLR` takes in input the *Stipula* grammar and returns the syntax tree that may be visited. The type inference is implemented by an ad-hoc extension of the default `ANTLR` visitor. The inference uses a map to associate types to fields, assets and functions. When the contract is entered, the associated types are pairwise different variables. Then, following the rules of Table 4, the type inference tries to instantiate the variable types. For example, if the contract declares `field x`, then `x` is initially bound to the variable type `X`. Then, if a function contains the condition

```
(x>=1 && x<=5)
```

`X` is unified with the type `real`. The same process is used for fields that store events. In particular, when the type inference finds an event $\mathtt{t} \gg @\mathtt{Q}\ \{\ \mathtt{S}\ \} \Rightarrow @\mathtt{Q}'$ then the (variable) type of `t` is unified with `time`. Whenever a unification is not possible, the prototype reports a type error. It is worth noticing that our prototype also instantiates variable types at runtime. If a field (with a variable type) is instantiated at runtime with a value, for example during the agreement, then the type is unified with the type of the value. A type error might occur in the following execution in case a value of different type is assigned to that field.

Contracts and Agreements. A *Stipula* contract is implemented as a `Java` class definition where fields implement *Stipula* parties, fields and assets. Additional fields retain the objects implementing the agreement and the functions. The class contains a `run` method that controls the execution of the contract. The first operation of this method is to execute the agreement. *Stipula* carefully handles digital identities of parties, ensuring that contract's functions are actually invoked by the correct callers. The prototype achieves this by associating a unique serial code to each party's (formal) name, which is randomly generated during the agreement. At runtime, party's invocations must use the corresponding code instead of the name and the program always checks the identity and, if it is correct, the execution continues, otherwise the execution terminates. Once the party's serial codes have been fixed, the agreement begins the consensus process on the initial values of fields. In particular, the parties are asked for the values they must agree on and, when all the values have been inserted, it checks for their consistency. If every party agrees on the values, the `run` method moves the *Stipula* contract to the initial state, otherwise the execution terminates.

Functions and States. Every function is represented by an object of a `Java` class `Function`. The fields store the local variables, the list of parties that can invoke the function, the list of initial states (which are represented as strings) and the final state of the function. A functions has also an `Expression` field that stores a pointer to the node of the syntax tree representing the precondition (if the precondition is not present, then the field is `null`) and a `Statement` field that contains the pointer to the function's body. When a function starts, all the statements are executed. As result of the execution, the state of the system changes and the fields and assets of the contract class are possibly updated.

Assets. Assets are linear resources that cannot be duplicated or leaked. Our prototype features assets as objects of a class `Asset` with a field `rawvalue` retaining the value of the asset and the methods `withdraw`, `move` and `increase`:

```
class AssetException extends RunTimeException {
    AssetException(String s) { super(s); }
}
class Asset {
    float rawvalue ;
    public Asset() {
      rawvalue = 0;
    }
    public void withdraw(float val, Asset h) {
          if (rawvalue >= val) {
            rawvalue -= val ;
            h.increase(val) ;
          } else throw new AssetException("Erroneous withdraw") ;
    }
    public void move(float val, Party p) {
          if (rawvalue >= val) {
            rawvalue -= val ;
            p.receive(val) ;
          } else throw new AssetException("Erroneous withdraw") ;
    }
    public void increase(float val) {
        rawvalue += val ;
    }
}
```

In our implementation `Asset` allows the operations `withdraw` and `move` only if the current value of `rawvalue` is greater or equal than `val`. If one tries to move more assets than owned, an `AssetException` is thrown and the execution terminates. In this way, the operations $\mathtt{E} \multimap \mathtt{h}, \mathtt{h}'$ and $\mathtt{E} \multimap \mathtt{h}, \mathtt{A}$ may be encoded by `h.withdraw(E,h')` and `h.move(E,A)`, respectively.

Events. Events correspond to scheduling a computation for future execution. When $E \gg @\mathtt{Q}\,\{\,S\,\} \Rightarrow @\mathtt{Q}'$ is executed, the time expression E is computed, let it be `t`, and a `TimerTask Java` thread is set to be triggered at time `t`. The prototype admits both absolute and relative times expressed in

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

Figure 3: Correspondence between legal elements and *Stipula* features

hours. In any case, the system computes how many seconds from now the events should start and schedules a `TimerTask` for each event. The main thread synchronises with the event thread and, if the state of the contract agrees with the initial state of the event, then the corresponding body is performed.

Interpretation.. The execution of a *Stipula* program is interpreted by a further extension of the Visitor generated by `ANTLR`. At each step, the method `run` displays the parties/functions that may be invoked and when a function is taken, the interpreter executes the corresponding node of the `ANTLR` syntax tree. At the end of the execution the control returns to `run`. The events are executed in a similar way (see the foregoing discussion).

8. Related works

Dwivedi et al. [17] conduct a systematic literature review about smart contract languages (SCL) and identify properties that are critical for drafting legally binding digital contracts. Some of the distinctive features of *Stipula* appear in existing SCLs, such as the design of contracts as finite state machines (as in Solidity), the constraint that a function is called only by permitted callers and only in given contract's states (as Solidity's modifiers, Flint's caller capability blocks [32] and Obsidian typestates), and assets as primitive datatypes (as in [6, 32], and in languages for financial contracts and bitcoin transfer *e.g.* [1, 5, 22, 13]). However, these SCLs are not specifically designed for legally binding contracts, indeed they do not fully support the contractual aspects that [17] puts forward for legally enforceable SCLs. In *Stipula*, the agreement construct and the specific design patterns enable a precise correspondence between the code and the distinctive elements of legal contracts, such as the meeting of the minds, permissions, prohibitions, obligations and judicial enforcement. Figure 3 summarises the correspondences.

In the literature there are also a number of languages and frameworks that aim at transforming legal semantic rules into smart-contract code, *e.g.* [19, 15, 14, 20]. These languages are actually specification languages, that provide attributes and clauses that naturally encode rights, obligations, prohibitions, which are not easily mapped to high-level programming languages, such as Solidity, `Java`. On the other hand *Stipula*, with its distinctive primitives and legal design patterns, aims to be an intermediate language, between a specification language and an high-level programming language. In this sense, *Stipula* and its formalization inspired by concurrency theory can be considered a *legal calculus* in the sense of [4]. Indeed, the Orlando [3] language for modeling conveyances in property law, and the Catala languages [27] for modeling statutes and regulations clauses, have both a core minimal language under an English-like surface syntax close to what legal professionals would use.

We admit that writing a legal contract directly in *Stipula* requires programming skills, therefore, to achieve the full potential of the language, a user-friendly and easy-to-use programming

interface is needed. We plan topic this for future work, taking inspiration from visual programming interfaces as in [31, 36] or domain specific markup languages as [16].

Several projects have put forward legal markup languages and ontologies to wrap logic and other contextual informations around traditional legal prose and providing templates for common contracts. Among the others, OpenLaw [37] also allows to reference Ethereum-based smart contracts into legal agreements, and automatically triggers them once the agreement is digitally signed by all parties. Signatures by all relevant parties are stored on IPFS (the Inter-Planetary File System) and the Ethereum blockchain. Another relevant project is Accord [34], which provides an open, standardized format for smart legal contracts, consisting of natural language and computable components. These contracts can then be interpreted by machines(that do not necessarily operate on blockchains). These projects come with sets of templates for standard legal 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 of legal contracts (see [11] for a foundational study of legal contracts' main features and their correspondence with *Stipula* design patterns).

Lexon [25] uses context free grammars to define a programming language syntax that is at the same time human readable and automatically translated into, *e.g.* Solidity. Even if the high level Lexon code is very close to natural language, there is no real control over the code that is actually run: the semantics of the high level language is not defined, thus the actual behaviour of the contract is that of the automatically generated Solidity code, which might be much more subtle than that of the (much simpler and more abstract) Lexon source. Compared to the Solidity code of the Lexon examples in [26], the *Stipula* version of the same contracts is much clearer, thanks to primitives like agreement and asset movements. Thus, directly coding in *Stipula* appears to be safer than relying on the Lexon-Solidity pair. Nevertheless, it should be not difficult to design an automatic translation from Lexon to *Stipula* so to not bind Lexon contracts to any specific implementation.

Our work aims at conducing a foundational study of legal contracts, in order to elicit a precisely defined set of building blocks that can be used to describe, analyse and execute (thus enforce) legal agreements. This is similar to what has been done in [21], which puts forward 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 [33, 5], which are (small) domain specific language featuring constructs like participants, tokens, currency and timeouts to wait until a certain condition becomes true (similarly to *Stipula*).

In Marlowe the control logic is fully embedded in the contract, that can always progress (and close) even without the participation of a non collaborative party. In particular, to ensure progress, urgency and default refund mechanism, Marlowe always impose timeouts, even for money commitments and money retrieval authorizations. However, this *pull* mechanism makes the interaction logic indirect and complex, whereas in *Stipula* it is always clear who is the subject of each action and where the assets flow. Setting the correct timeouts in Marlowe may also become difficult, while in *Stipula* timeouts are optional (using events) and default mechanisms can be programmed as escape clauses that can be triggered by the Authority calling corresponding functions. Otherwise, in *Stipula*, the finite lifetime of a contract may be easily programmed by arranging its value during the agreement phase, and issuing a corresponding event in the initial state.

Marlowe semantics is written in Haskell and is executed on the Cardano [24] blockchain, where the lifetime of a contract and timeouts are computed in terms of slot numbers, that are measured by (Cardano's) block number. Then the language interpreter takes as additional input the current slot interval, and the contract will continue as the timeout-continuation as soon as any valid transaction is issued after the timeout's slot number is reached. Thus, the execution of a contract will involve multiple blocks, with multiple steps in each block. Moreover, the time limit T used as timeouts must be intended as $[T - \Delta, T + \Delta]$, where Δ is a parameter of the implementation. Similar ideas can be exploited to implement *Stipula*'s events on top of blockchains.

Overall, we remark that legal contracts are more general and more expressive than financial contracts. Accordingly, DSLs like Marlowe and Findel are built around a fixed set of contract's

combinators, that can be combined according to suitable (algebraic) laws. Then 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, e.g. a Java application or a Ethereum smart contract, and the parties must collaborate by invoking the contract's functions to make the contract progress.

Finally, the class-based programming style of *Stipula* is similar to that of Solidity, but there are many differences between the two. Actually, *Stipula* is much similar to Obsidian [8], which is based on state-oriented programming and explicit management of linear assets, whose usability has been experimentally assessed [7]. Obsidian has a type system that ensures the correct manipulations of objects according to their current states and that linearly typed assets are not accidentally lost. On the contrary, *Stipula* is untyped: the introduction of an expressive type discipline is orthogonal and is postponed to a later stage where we plan to investigate static analysis techniques specifically suitable to the legal setting. As a cons, Obsidian has no agreement nor event primitives, therefore the consensus about the contract's terms and the enforcing of legal obligations must be implemented in a much more indirect way.

9. Conclusions

This paper presents a domain-specific language for defining legal contracts, that has been developed in close connection with lawyers to select few concise and intelligible primitives that have a precise correspondence with the distinctive elements of juridical acts, such as permissions, prohibitions, and obligations. The meaning of *Stipula* primitives is precisely defined in terms of a formal operational semantics, so that the contracts' behaviour is fully specified and amenable to automatic verification. The actual adoption of *Stipula* 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 provide interesting insights on the application of programming languages to the legal field [4, 9].

We studied an equational theory for *Stipula*, that allows one to equate contracts that differ for clauses (events) that can never be triggered or for the order of non-interfering communications. This extensional semantics illustrates that we can think of legal contracts as interaction *protocols* that regulate the relationships between concurrent parties in terms of permissions, obligations, prohibitions, escrows and securities. We have also highlighted the intrinsic open nature of legal contracts, where contractual conditions may depend on contextual situations and may appeal to complex and undetermined social-normative concepts (such as fairness or good faith). We have shown that *Stipula* provides ad-hoc programming patterns to deal with exceptional behaviors and judicial enforcements. Other exceptional behaviours, such as those of *force majeure* and *hardship* can be modelled by extending the language with higher-order features [23]. Overall, it is matter of (our) future research to deeply investigate the interdisciplinary aspects of law and computer science and provide general solutions in *Stipula*.

From the computer science point of view, a number of issues deserve to be investigated in full detail: a detailed study of contracts' errors and execution failures, the precise implementation of the language together with a programming interface well suited to legal practitioners, and further static analysis techniques. As regards this last point, we are currently developing an analyzer that statically verifies *liquidity* of software contracts [10, 2]. This property is held by those contracts that do not freeze any asset forever, *i.e.* that are not redeemable by any party. Liquidity could be imposed by construction by adopting a programming pattern that allows a third party –say, an Authority– to retrieve or redistribute all assets in any contract's (final) state. However legal contracts can be very general, so we do not impose their liquidity by construction, leaving the choice of whether to use the suitable pattern. Similarly, the expressiveness of *Stipula* allows to write erroneous contracts with contradicting or nondeterministic permissions/prohibitions/obligations. However, even real natural legal contracts sometimes contain contradictions and dilemmas, and their execution might lead to nondeterministic situations, especially when one of the parties do

not perform an action that was expected to do. Depending on how the textual contract is written, this may even cause a legally uncertain situation that can only be solved by a court. Therefore, in the design of *Stipula* we privileged the direct formalisation of normative elements as programming patterns, so to increase transparency and help in disambiguating contractual clauses. Contradictions and erroneous contracts behaviours can later be identified by means of static analysis tools developed on top of the formal semantics of the language.

Overall, we are optimistic that future research on *Stipula* can satisfactorily address the above issues because its model is simple and rigorous, which are, in our opinion, fundamental criteria for reasoning about legal contracts and for understanding their basic principles, not just for writing safer contracts. In our mind, *Stipula* is the backbone of a framework where addressing and studying other, more complex features that are drawn from juridical acts.

Acknowledgements

Giovanni Sartor has been supported by the H2020 European Research Council (ERC) Project "CompuLaw" (G.A. 833647). Cosimo Laneve has been partly supported by the H2020-MSCA-RISE project ID 778233 "Behavioural Application Program Interfaces (BEHAPI)". Silvia Crafa dedicates this work to Università di Padova and its 800th academic year.

[1] Bartoletti, M., Zunino, R., 2018. Bitml: A calculus for bitcoin smart contracts. In: Proc. of Computer and Communications Security. CCS '18. ACM, New York, NY, USA, pp. 83–100.

[2] Bartoletti, M., Zunino, R., 2019. Verifying liquidity of bitcoin contracts. In: Nielson, F., Sands, D. (Eds.), Principles of Security and Trust. Springer International Publishing, pp. 222–247.

[3] Basu, S., Foster, N., Grimmelmann, J., 2019. Property conveyances as a programming language. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. Onward! 2019. Association for Computing Machinery, New York, NY, USA, p. 128142.

[4] Basu, S., Mohan, A., Grimmelmann, J., Foster, N., 2022. Legal calculi. Tech. rep., ProLaLa 2022 ProLaLa Programming Languages and the Law, at `https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi`.

[5] Biryukov, A., Khovratovich, D., Tikhomirov, S., 2017. Findel: Secure derivative contracts for ethereum. In: Financial Cryptography and Data Security - FC 2017. Vol. 10323 of Lecture Notes in Computer Science. Springer, pp. 453–467.

[6] Blackshear, S., et al., 2021. Move: A language with programmable resources. `https://developers.diem.com/main/docs/move-paper`.

[7] Coblenz, M. J., Aldrich, J., Myers, B. A., Sunshine, J., 2020. 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.

[8] Coblenz, M. J., Oei, R., Etzel, T., Koronkevich, P., Baker, M., Bloem, Y., Myers, B. A., Sunshine, J., Aldrich, J., 2020. Obsidian: Typestate and assets for safer blockchain programming. ACM Trans. Program. Lang. Syst. 42 (3), 14:1–14:82.

[9] Crafa, S., 2022. From legal contracts to legal calculi: the code-driven normativity. In: Castiglioni, V., Mezzina, C. A. (Eds.), Proceedings Combined 29th International Workshop on Expressiveness in Concurrency and 19th Workshop on Structural Operational Semantics , Warsaw, Poland, 12th September 2022. Vol. 368 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 23–42.

[10] Crafa, S., Laneve, C., 2022. Liquidity analysis in resource-aware programming. In: Formal Aspects of Component Software - 18th International Conference, FACS 2022, Virtual Event, November 10-11, 2022, Proceedings. Vol. 13712 of Lecture Notes in Computer Science. Springer, pp. 205–221.

[11] Crafa, S., Laneve, C., Sartor, G., 10 2021. Pacta sunt servanda: legal contracts in Stipula. Tech. rep., arXiv:2110.11069.

[12] Crafa, S., Laneve, C., Veschetti, A., July 2022. Stipula Prototype. Available on github: `https://github.com/stipula-language`.

[13] Crary, K., Sullivan, M. J., 2015. Peer-to-peer affine commitment using bitcoin. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '15. Association for Computing Machinery, New York, NY, USA, p. 479488.

[14] de Kruijff, J. T., Weigand, H. H., 2018. An introduction to commitment based smart contracts using reactionruleml. In: Proceedings of the 12th international Workshop on Value Modeling and Business Ontologies (VMBO). Vol. 2239. CEUR-WS.org, pp. 149–157.

[15] de Kruijff, J. T., Weigand, H. H., 2019. Introducing commitruleml for smart contracts. In: Proceedings of the 13 th international Workshop on Value Modeling and Business Ontologies (VMBO). Vol. 2383. CEUR-WS.org.

[16] Dwivedi, V., Norta, A., Wulf, A., Leiding, B., Saxena, S., Udokwu, C., 2021. A formal specification smart-contract language for legally binding decentralized autonomous organizations. IEEE Access 9, 76069–76082.

[17] Dwivedi, V., Pattanaik, V., Deval, V., Dixit, A., Norta, A., Draheim, D., jun 2021. Legally enforceable smart-contract languages: A systematic literature review. ACM Comput. Surv. 54 (5).

[18] Fournet, C., Gonthier, G., 2000. The join calculus: A language for distributed mobile programming. In: Applied Semantics, International Summer School, APPSEM 2000. Vol. 2395 of Lecture Notes in Computer Science. Springer, pp. 268–332.

[19] Frantz, C. K., Nowostawski, M., 2016. From institutions to code: Towards automated generation of smart contracts. In: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W). pp. 210–215.

[20] He, X., Qin, B., Zhu, Y., Chen, X., Liu, Y., 2018. Spesc: A specification language for smart contracts. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC). Vol. 01. pp. 132–137.

[21] Jones, S. L. P., Eber, J., Seward, J., 2000. 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. ACM, pp. 280–292.

[22] Lamela Seijas, P., Thompson, S., 2018. Marlowe: Financial contracts on blockchain. In: Margaria, T., Steffen, B. (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. Springer, Cham, pp. 356–375.

[23] Laneve, C., Parenti, A., Sartor, G., 2023. Legal Contracts amending with Stipula, in 2nd Workshop on Programming Languages and the Law, January 15, 2023.

[24] Cardano Developers, 2020. Cardano Documentation. `https://docs.cardano.org/`.

[25] Lexon Foundation, 2019. Lexon Home Page. `http://www.lexon.tech`.

[26] Lexon Foundation, 2020. Lexon Demo Editor. http://demo.lexon.tech/apps/editor/.

[27] Merigoux, D., Chataing, N., Protzenko, J., aug 2021. Catala: A programming language for the law. Proc. ACM Program. Lang. 5 (ICFP).

[28] Milner, R., 1989. Communication and concurrency. PHI Series in computer science. Prentice Hall.

[29] Parr, T., 2013. The Definitive ANTLR 4 Reference, 2nd Edition. Pragmatic Bookshelf.

[30] Pierce, B. C., 2002. Types and programming languages. MIT Press.

[31] Reitwiebner, C., 2018. Babbagea mechanical smart contract language, at https://medium.com/@chriseth/babbage-a-mechanical-smart-contract-language-5c8329ec5a0e.

[32] Schrans, F., Eisenbach, S., Drossopoulou, S., 2018. Writing safe smart contracts in flint. In: Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming. Programming'18 Companion. ACM, New York, NY, USA, p. 218219.

[33] Seijas, P. L., Nemish, A., Smith, D., Thompson, S., 2020. Marlowe: implementing and analysing financial contracts on blockchain. In: Proceedings of the Workshop on Financial Cryptography and Data Security. Springer International Publishing, pp. 496–511.

[34] Source Contributors, O., 2018. The Accord Project. https://accordproject.org.

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

[36] Weingaertner, T., Rao, R., Ettlin, J., Suter, P., Dublanc, P., 2018. Smart contracts using blockly: Representing a purchase agreement using a graphical programming language. In: 2018 Crypto Valley Conference on Blockchain Technology (CVCBT). pp. 55–64.

[37] Wright, A., Roon, D., AG, C., 2019. OpenLaw Web Site. https://www.openlaw.io.