Legal Contracts amending with *Stipula* [⋆]

Cosimo Laneve[1][0000−0002−0052−4061], Alessandro Parenti[2][0000−0002−9855−7792], and Giovanni Sartor[2][0000−0003−2210−0398]

[1] Department of Computer Science and Engineering, University of Bologna, Italy
[2] Department of Legal Studies, University of Bologna, Italy

Abstract. Legal contracts can be amended during their lifetime through the agreement of the parties or in accordance with the doctrines of force majeure and hardship. When legal contracts are defined using a programming language, amendments are made through runtime adjustments to the contract's behavior and must be expressed by means of appropriate language features. In this paper, we examine the extension of *Stipula*, a formal language for legal contracts, with *higher-order functionality* to enable the dynamic updating of contract codes. We discuss the semantics of the language when amendments either extend or override the contract's functionality. Additionally, we study two techniques for constraining amendments, one using annotations within the contract and another that allows for runtime agreements between parties.

1 Introduction

In [7] we presented *Stipula*, a domain specific language that can assist lawyers in drafting executable legal contracts, through specific software patterns. The language is based on a small set of programming primitives that have a precise correspondence with the distinctive elements of legal contracts [6]. By means of these primitives, it is possible to transfer rights (such as rights of property) from one party to another and to take advantage of escrows and securities. The benefits of coding legal contracts are evident: it enables the identification of potential inconsistencies in regulation, reducing the complexity and the ambiguity of legal texts and automatically executing legal rules.

Stipula has been designed with the principle of having an abstraction level as close as possible to traditional legal contracts, which are written in natural languages, thus easing the writing and inspecting of the codes. In this contribution we pursue on our programme addressing the need of removing or amending the effects of a contract after it has been agreed upon.

There may be several reasons for modifying a contract. For example, a contract may be declared totally or partially void by an adjudicator because its content, or the process of its formation, violates the law. More interesting are the situations of *force majeure* and *hardship*, which occur when unforeseen events

[⋆] Supported by the SERICS project (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU – and by the H2020 ERC Project CompuLaw (G.A. 833647).

make performance impossible or impracticable (force majeure) or substantially upset the economic balance of the contract (hardship) [3, 11]. While in the first case the party successfully invoking force majeure may be relieved, at least temporarily, from performance or may terminate the contract, in the second case the party subject to hardship may be entitled to to obtain an adaptation of the contract to the changed circumstances.

The current *Stipula* contracts are immutable. Therefore, in order to model either force majeure or hardship one should anticipate when the contract is traded all the appropriate amendments for each possible circumstance. While this is easy for termination clauses (it is enough to include a transition to a final state), it is clearly impossible for generic amendments [18]. Even an attempt to do that would raise drafting costs and introduce huge complexities in the contract, thus nullifying one of the main objectives of *Stipula*, which is to have a simple and intelligible code.

To address amendments we propose an extension of *Stipula* with a higher-order mechanism. Following [20], we admit that function invocations may carry *codes* that patch the previous ones. In Section 4 we study the formal semantics of the resulting language, called *higher-order Stipula*. In particular, we identify and discuss two paradigmatic scenarios. A scenario where the modification affects the whole body of the contract and its code is completely changed and substituted by a new code. Another scenario is where the amendment only regards some parts of the contract while leaving the other parts still operative. This situation adds a further level of semantic complexity in that it requires to deal with the coexistence of old and new code. We give examples of the use of *higher-order Stipula* in Section 3 that will spot these issues.

According to the semantics defined in Section 4, in *higher-order Stipula* amendments are unconstrained: a party may modify the contract without the consent of all the parties involved. This is clearly at odds with the fundamental principles of contract law (i.e., *consensus ad idem*). We then explore two methods for limiting amendments. In Section 5 we discuss a set of static-time constraints on amendments that the parties agree when the contract is traded. The constraints allow one to implement a predicate that parses the (run-time) amendments and verifies their compliance. In Section 6 we study a technique that requires the agreement of the parties in correspondence of every amendment.

We end our contribution by discussing the related work in Section 7 and delivering our final remarks in Section 8.

2 From *Stipula* to *higher-order Stipula*

Higher-order Stipula is an extension of *Stipula* with higher-order functions. In this contribution, for simplicity, we extend a *lightweight* version of the language in [7] (the full language also has the *agreement clause* and *events*); this allows us to avoid discussions that are out of the scope of this paper. Additionally, since *Stipula* is not popular, we first present the lightweight language and then the extension.

$$
\begin{array}{lcl}
F & ::= & \_ \quad | \quad \texttt{@Q}\ \texttt{A} : \texttt{f}(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rightarrow \texttt{@Q}'\ F \quad | \quad \texttt{@Q}\ \texttt{A} : \texttt{F}(\!|\,\overline{X}\,|\!)\{\,H\,\}\ F \\
P & ::= & E \to x \quad | \quad E \to \texttt{A} \quad | \quad E \times h \multimap h' \quad | \quad E \times h \multimap \texttt{A} \quad | \quad \texttt{if}\,(E)\,\{\,S\,\}\,\texttt{else}\,\{\,S\,\} \\
S & ::= & \_ \quad | \quad P\ S \\
E & ::= & v \quad | \quad V \quad | \quad E\,\texttt{op}\,E \quad | \quad \texttt{uop}\,E \\
H & ::= & (\texttt{remove}\ X)?\ (\texttt{add}\ X)?\ \texttt{run}\ X
\end{array}
$$

Table 1. Syntax of *Stipula* (in black only) and *higher-order Stipula*

We use disjoint sets of names: *contract names*, ranged over by `C`, `C`′, $\cdots$; names referring to digital identities, called *parties*, ranged over by `A`, `A`′, $\cdots$; *function names* ranged over `f`, `g`, $\cdots$ (in general, function names start with a small-case letter); *asset names*, ranged over by h, k, $\cdots$, to be used both as contract's assets and function's asset parameters; *non asset names*, ranged over x, y, $\cdots$, to be used both as contact's fields and function's non asset parameters. Assets and generic contract's fields are syntactically set apart since they have different semantics, similarly for functions' parameters. Names of assets, fields and parameters are generically ranged over by V. Names `@Q`, `@Q`′, $\cdots$ will range over contract states. To simplify the syntax, we often use the vector notation $\overline{x}$ to denote possibly empty *sequences* of elements. With an abuse of notation, in the following sections, $\overline{x}$ will also represent the *set* containing the elements in the sequence.

The code of a *Stipula* contract is

```
stipula C { parties Ā   fields x̄   assets h̄   init @Q   F }
```

where `C` identifies the *contract name*; $\overline{\texttt{A}}$ are the *parties* that can invoke contract's functions, $\overline{x}$ and $\overline{h}$ are the *fields* and the *assets*, respectively, and the *initial state* is set to `@Q`. The contract body also includes the sequence F of functions, whose syntax is defined in Table 1 (the terms in black). It is assumed that there is no clash of names of parties, fields, assets and functions' parameters. In the following, the *declaration part* of a contract, namely the sequence `parties` $\overline{\texttt{A}}$ `fields` $\overline{x}$ `assets` $\overline{h}$ F will be ranged over by the symbols $\mathbb{D}$, $\mathbb{D}'$, $\cdots$.

First-order functions highlight who is the caller party `A`, the state `@Q` when the invocation is admitted and the name of the function. The invocation has two lists of parameters: the *formal parameters* $\overline{y}$ in brackets and the *asset parameters* $\overline{k}$ in square brackets. The *precondition* E constrains the execution of the body; the *body* $\{\ S\ \} \Rightarrow$ `@Q`′ specifies the *statement part* S and the state `@Q`′ of the contract when the function execution terminates.

Statements S include the empty statement _ and different prefixes followed by a continuation. Prefixes P use the two symbols $\to$ and $\multimap$ to differentiate operations on non-asset names and on assets, respectively. The prefix $E \to x$ updates the field or the parameter x with the value of E; $E \to$ `A` sends the value of E to the party `A`; $E \times h \multimap h'$ subtracts the value of $E \times h$ from the asset h and adds it to h', $E \times h \multimap$ `A` subtracts the value of $E \times h$ from the asset h and transfers it to `A`. (The semantics in Section 4 will enforce that assets never have negative values.) In the rest of the paper we will always abbreviate $1 \times h \multimap h'$ and $1 \times h \multimap$ `A` (which are very usual, indeed) into $h \multimap h'$ and $h \multimap$ `A`,

```
stipula Deposit {
    parties Client, Farm
    fields cost_flour
    assets flour
    init @Standard

    @Standard Farm: send()[h]{ h → Client     h ⊸ flour } ⇛ @Standard

    @Standard Client: buy(x)[w] (w == x×cost_flour && x <= flour){
            (x/flour)×flour ⊸ Client     w ⊸ Farm
        } ⇛ @Standard

    @Standard ~ : Hardship(| X, Y, Z |){ remove X add Y run Z }
}
```

Table 2. The `Deposit` contract with a higher-order function

respectively. It is worth to spot the difference between $h \rightarrow$ `A` and $h \multimap$ `A`: in the first case, the real number representing the *value* of h is sent to `A`, but h still retain its value; in the second case, the asset h is sent to `A` and *h is emptied.* We also use " ~ " to address all the parties. For instance, if the parties are `A` and `B`, then `"hello"` $\rightarrow$ ~ means `"hello"` $\rightarrow$ `A`   `"hello"` $\rightarrow$ `B` (the order is not relevant, according to the extensional semantics in [7]). Prefixes also include *conditionals* `if` $(E)\,\{\,S\,\}$ `else` $\{\,S'\,\}$ with the standard semantics.

Expressions E include constant values v, which may be strings, reals, booleans, and asset values, names V, and both binary and unary operations (on reals and booleans). In particular, real numbers n are written as nonempty sequences of digits, possibly followed by "`.`" and by a sequence of digits (*e.g.* `13` stands for `13.0`). The number may be prefixed by the sign `+` or `-`. Reals come with the standard set of binary arithmetic operations (`+`, `-`, $\times$, $/$). Boolean constants are `false` and `true`; the operations on booleans are conjunction $\&\&$, disjunction `||`, and negation `!`. Constant values of type asset represent *fungible* resources (*e.g.* digital currencies). For simplicity, fungible asset constants are assumed to be identical to *nonnegative real numbers* (assets can never assume negative values). Relational operations (`<`, `>`, `<=`, `>=`, `==`) are available between any expression.

To illustrate lightweight *Stipula*, we discuss a simple contract in Table 2 (the part in black). A `Client` contracts with a `Farm` to pay flour at a given cost. The protocol is the following: `Farm` sends the flour (function `send`) and the good is stored in the `flour` asset: no delivery to `Client` is operated till he pays for it. The prefix `h` $\rightarrow$ `Client` communicates to the `Client` that a new amount of flour is available. The function `buy` takes in input a value `x` denoting that the `Client` wants to buy an amount `x` of flour, and an asset `w` representing the money he wants to spend. The function takes `x` kg of flour from the deposit (provided it is in – see the guard), sends the flour to the `Client` and updates the asset `flour` correspondingly – operation `(x/flour)×flour` $\multimap$ `Client` –; the money `w` is transferred to `Farm`.

Contract are invoked by specifying the actual identities of parties and the fields' values (at the beginning all the assets are empty) – *c.f.* the semantics in Section 4. We use italic fonts A, B, *Farm*, *Client*, $\cdots$, to distinguish parties' actual identities from parties formal names `A`, `B`, `Farm`, `Client`, $\cdots$. These parties' actual identities correspond to digital identities and the same identity may be given to different formal names (which are always pairwise different). Indeed, it may happen that the same party may have two roles in a legal contract. The contract will begin in the state that is specified in the `init` clause.

Higher-order Stipula extends the syntax of *Stipula* in Table 1 with *higher-order functions* – the red part. In particular, we use *higher-order function names* ranged over `F`, `G`, $\cdots$ (in general, function names that start with an upper-case letter). We discuss the declaration `@Q A : F`$(\!| X, Y, Z |\!)\{$ `remove` X `add` Y `run` $Z\,\}$ that has a complete set of *(amendment) directives* H. The parameters of `F` are X, Y and Z: X is a sequence of function names (possibly with state and party names) that will be removed from the contract; Y is a possible empty sequence of declarations of new parties with their identities, fields and assets as well as of functions that will amend the contract – it will be instantiated by codes $\mathbb{D}$; Z is the body of `F` and will be instantiated by codes $\{\, S \,\} \Rightarrow$ `@Q`, where `@Q` may also be a new state defined in (the code that instantiates) Y. According to the syntax in Table 1, the remove and add clauses in the directives H are optional, while the run clause is mandatory. For example, the function `Hardship` of the `Deposit` contract in Table 2 represents a clause included by `Client` and `Farm` according to which a party can ask either for the amendment of the contract or for its termination. (This may be subordinated to a third party's decision – a court, an arbitrator or a mediator – assessing the existence of hardship conditions; here, for simplicity, we empower `Client` and `Farm` to perform these updates). In Section 3 we will study possible amendments of the `Deposit` contract.

We notice that our syntax has been inspired by the Delta-Oriented Programming paradigm [15]: the directives "remove" and the "add" are taken from that paradigm. Preliminary investigations show that these directives are already sufficient for specifying hardship clauses. It will be a focus for future works to test *higher-order Stipula* with the representation of more complex, context-specific contracts.

Remark 1. The syntax of (*higher-order*) *Stipula* is type-free: types have been dropped because there are no such annotations in standard legal contracts and therefore they may be initially obscure to unskilled users, such as legal practitioners. The paper [7] defines and the prototype [8] implements 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*.

3 Examples of amendments

Because of the variety of situations, needs and dynamics involved, the contractual practice is, by nature, a very heterogeneous field. This makes it difficult, if not impossible, to create general overarching examples starting from particular cases. Here we discuss three simple examples built on the Deposit contract of Table 2, with the specific purpose of explaining the technical functioning of the *higher-order* to modify *Stipula* contracts.

The initial example is commonly found in practice, *i.e. hardship* cases [11], and builds a simplistic representation of contractual relationship around it. Because of a war outbreak and a sudden rise in production costs, the Farm requests to amend the contract: she requires that the payment is performed *in advance* with respect to the delivery and that half of the amount is sent immediately to her. Therefore she invokes

```
Farm : Hardship⦇ε, 𝔻, {"Pay_in_Advance" →˜  flour ⊸ Farm} ⇒ @Excp⦈
```

where ε indicates that there is no function to remove and $\mathbb{D}$ is

```
assets wallet

@Excp Client: order()[w] {
    w/cost_flour → Farm     0.5×w ⊸ Farm     w ⊸ wallet } ⇒ @Excp2

@Excp2 Farm: send()[h] (h == (2×wallet)/cost_flour){
    h ⊸ Client     wallet ⊸ Farm } ⇒ @Excp

@Excp ˜ : Hardship⦇X, Y⦈{ add X run Y }
```

That is, the code $\mathbb{D}$ is specifying a new asset and three new functions. The function order lets Client pay in advance, sends to Farm the order w/cost_flour and half of the cost 0.5 × w, the other half is stored in the new asset wallet. Once the flour is ready, it is delivered to the Client (function send) and the wallet is delivered to Farm. Notice that the third parameter (the one replacing Z in Table 2) empties the flour asset returning the amount to Farm and lets the contract transit to the *new* state @Excp. Overall, the old behaviour is suppressed in favour of the new one because it is not possible to return to the @Standard state.

After some time, the parties want to return to the old protocol. However, a new law imposes a 20% tax on flour sales. To bear the new taxation, the Farm invokes the hardship clause to increase flour price (also tax payment to the Government gets implemented). Therefore, in the state @Excp, Farm invokes Hardship⦇$\mathbb{D}'$, B'⦈ (notice that the Hardship in @Excp has two arguments only and a different body than the one in @Standard) where

```
𝔻' = parties Government = Govern

    @Standard Client: buy(x)[w] (w == x×cost_flour && x <= flour){
        (x/flour)×flour ⊸ Client   0.2×w ⊸ Government   w ⊸ Farm
        } ⇒ @Standard

B' = { "Back_to_Standard_and_upgrade_flour_price" → ˜
       cost_flour + 0.2×cost_flour → cost_flour } ⇒ @Standard
```

$\mathbb{D}'$ is extending the parties with a new one (Government whose id is *Govern*) and the function buy dispatches the 20% of the cost of every transaction to the Government. The old protocol is restored because the body in the last line is making the transition to the `Standard` state. However, in this case, the new function `@Standard Client:buy` is *overriding* the old one in Table 2, which will be never accessed again because its guard is the same of the new function. We observe that, in *higher-order Stipula*, parties, assets and fields names may be added by the amendment; we only constrain the new names not to clash with old ones.

Later on, `Farm` decides to accept orders only if they are above a certain quantity `lbval`. Therefore, in the state `@Standard`, she invokes `Hardship`$(\!|$ `buy`, $\mathbb{D}''$, B'' $|\!)$ where

```
𝔻″ = fields lower_bound

     @Standard Client: buy(x)[w]
         (w == x×cost_flour && x <= flour && x >= lower_bound){
             (x/flour)×flour ⊸ Client   0.2×w ⊸ Government   w ⊸ Farm
         } ⇒ @Standard

B″ = { "No_order_below_lbval_anymore" → ˜    lbval → lower_bound
     } ⇒ @Standard
```

In this case, the directive to execute is `remove buy add` $\mathbb{D}''$ `run` B'' that removes the function `buy` from $\mathbb{D}$ and adds the new one in $\mathbb{D}''$. We observe that the new field `lower_bound` is initialized in B''. It is also worth to notice that the invocation `Farm:Hardship`$(\!|\,\varepsilon, \mathbb{D}'', B''\,|\!)$ would have displayed a different effect: in this last case, since the `buy` in Table 2 is still in force, the invocations of `buy` with amount lower that `lbval` would have been dispatched to the old `buy` and accepted. This is an issue because the `buy` in Table 2 does not comply with the new law about taxes.

4 Semantics

Following the presentation of Section 2, we first define the operational semantics of lightweight *Stipula* and then we discuss the extension. We use a *transition relation* between *configurations*, *i.e.* $\mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ \Sigma \ \xrightarrow{\mu}\ \mathbb{D}' \Vdash \texttt{@Q}'\,,\ \ell'\,,\ \Sigma'$ where

- $\mathbb{D}$, $\mathbb{D}'$ are the declaration part of a contract (in *Stipula*, it is always $\mathbb{D} = \mathbb{D}'$, in *higher-order Stipula* $\mathbb{D}$ and $\mathbb{D}'$ may be different because of amendments, see below);
- `@Q`, `@Q`$'$ are states of $\mathbb{D}$ or $\mathbb{D}'$;
- ℓ, ℓ' called *memories*, are mappings from names (parties, fields, assets and function's parameters) to values. The values of parties are noted with italic fonts $A, A', \cdots$. These names cannot be passed as function's parameters and cannot be hard-coded into the source contracts, since they do not belong to expressions; they are initialized when the contract is instantiated or, for new parties, in the higher-order step;
- Σ, Σ' are (possibly empty) residuals of function bodies, *i.e.* Σ is either $\_$ (idle) or a term $S \Rightarrow \texttt{@Q}$. We assume that $\_\ S \Rightarrow \texttt{@Q}$ is equal to $S \Rightarrow \texttt{@Q}$;

[VALUE-SEND]
$$\frac{\llbracket E \rrbracket_\ell = v \quad \ell(\texttt{A}) = A}{\mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ E \to \texttt{A}\ \Sigma \xrightarrow{v \to A} \mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ \Sigma}$$

[FIELD-UPDATE]
$$\frac{\llbracket E \rrbracket_\ell = v \quad \ell' = \ell[x \mapsto v]}{\mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ E \to x\ \Sigma \longrightarrow \mathbb{D} \Vdash \texttt{@Q}\,,\ \ell'\,,\ \Sigma}$$

[ASSET-SEND]
$$\frac{\begin{array}{c}\ell(\texttt{A}) = A \quad 0 \le \llbracket E \rrbracket_\ell \le 1 \quad \llbracket E \times \texttt{h} \rrbracket_\ell = u \\ \llbracket \texttt{h} - u \rrbracket_\ell = v \quad \ell' = \ell[\texttt{h} \mapsto v]\end{array}}{\mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ E \times \texttt{h} \multimap \texttt{A}\ \Sigma \xrightarrow{u \multimap A} \mathbb{D} \Vdash \texttt{@Q}\,,\ \ell'\,,\ \Sigma}$$

[ASSET-UPDATE]
$$\frac{\begin{array}{c}0 \le \llbracket E \rrbracket_\ell \le 1 \quad \llbracket E \times \texttt{h} \rrbracket_\ell = u \quad \llbracket \texttt{h} - u \rrbracket_\ell = v \\ \llbracket \texttt{h}' + u \rrbracket_\ell = v' \quad \ell' = \ell[\texttt{h} \mapsto v, \texttt{h}' \mapsto v']\end{array}}{\mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ E \times \texttt{h} \multimap \texttt{h}'\ \Sigma \longrightarrow \mathbb{D} \Vdash \texttt{@Q}\,,\ \ell'\,,\ \Sigma}$$

[COND-TRUE]
$$\frac{\llbracket E \rrbracket_\ell = \texttt{true}}{\begin{array}{r}\mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ \texttt{if}\,(E)\,\{\,S\,\}\,\texttt{else}\,\{\,S'\,\}\ \Sigma \\ \longrightarrow \mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ S\ \Sigma\end{array}}$$

[COND-FALSE]
$$\frac{\llbracket E \rrbracket_\ell = \texttt{false}}{\begin{array}{r}\mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ \texttt{if}\,(E)\,\{\,S\,\}\,\texttt{else}\,\{\,S'\,\}\ \Sigma \\ \longrightarrow \mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ S'\ \Sigma\end{array}}$$

[STATE-CHANGE]
$$\mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ \_ \Rightarrow \texttt{@Q}' \longrightarrow \mathbb{D} \Vdash \texttt{@Q}'\,,\ \ell\,,\ \_$$

[FUNCTION]
$$\frac{\begin{array}{c}\texttt{@Q A : f}(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rightarrow \texttt{@Q}' \ \in\ \mathbb{D}[\texttt{@Q A : f}]_{\ell,\overline{u},\overline{v}} \\ \ell(\texttt{A}) = A \qquad \ell' = \ell[\overline{y} \mapsto \overline{u}, \overline{k} \mapsto \overline{v}]\end{array}}{\mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ \_ \xrightarrow{A:\texttt{f}(\overline{u})[\overline{v}]} \mathbb{D} \Vdash \texttt{@Q}\,,\ \ell'\,,\ S \Rightarrow \texttt{@Q}'}$$

[HO-FUNCTION]
$$\frac{\begin{array}{c}\texttt{@Q A: F}(\!|\, X, Y, Z \,|\!)\{\,\texttt{remove}\ X\ \texttt{add}\ Y\ \texttt{run}\ Z\,\} \ \in\ \mathbb{D}[\texttt{@Q A : F}]_{\ell,\varepsilon,\varepsilon} \\ \mathbb{D}' = \texttt{parties}\ \overline{\texttt{A}'} = \overline{A'}\ \texttt{fields}\ \overline{\texttt{z}}\ \texttt{assets}\ \overline{\texttt{k}}\ F \qquad \ell(\texttt{A}) = A \qquad \ell' = \ell[\overline{\texttt{k}} \mapsto \overline{\texttt{0}}, \overline{\texttt{A}'} \mapsto \overline{A'}]\end{array}}{\mathbb{D} \Vdash \texttt{@Q}\,,\ \ell\,,\ \_ \xrightarrow{A:\texttt{F}(\!|\,\texttt{P},\mathbb{D}',B\,|\!)} \mathbb{D} \setminus \texttt{P} \lhd \mathbb{D}' \Vdash \texttt{@Q}\,,\ \ell'\,,\ B}$$

Table 3. The transition relation of *Stipula* (in black only) and *higher-order Stipula*

– μ is a *label*, which is either empty, or a function call $A : \texttt{f}(\overline{u})[\overline{v}]$, or a value send $v \to A$, or an asset transfer $v \multimap A$. Labels are used to highlight the interactions between the contract and the parties.

We also use the *evaluation function* $\llbracket E \rrbracket_\ell$ that returns the value of E in the memory ℓ. In particular:

– $\llbracket v \rrbracket_\ell = v$ for values, $\llbracket V \rrbracket_\ell = \ell(V)$ for names of assets, fields and parameters.
– let $\underline{uop}$ and $\underline{op}$ be the semantic operations corresponding to uop and op, then $\llbracket uop\, E \rrbracket_\ell = \underline{uop}\, v$, $\llbracket E\, op\, E' \rrbracket_\ell = v\, \underline{op}\, v'$ with $\llbracket E \rrbracket_\ell = v$, $\llbracket E' \rrbracket_\ell = v'$.

Finally, let the *selection operation* be

$$\mathbb{D}[\texttt{@Q A : f}]_{\ell,\overline{u},\overline{v}} \ = \left\{ \texttt{@Q A : f}(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rightarrow \texttt{@Q}' \ \middle|\ \begin{array}{l}\texttt{@Q A :}\ f(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rightarrow \texttt{@Q}'\ \text{in}\ \mathbb{D} \\ \text{and}\ \llbracket E \rrbracket_{\ell[\overline{y} \mapsto \overline{u}, \overline{k} \mapsto \overline{v}]} = \texttt{true}\end{array} \right\}$$

That is, the selection $\mathbb{D}[\texttt{@Q A : f}]_{\ell,\overline{u},\overline{v}}$ returns a *set* of functions in $\mathbb{D}$ such that the corresponding guard E is true.

Table 3 reports the definition of the transition relation for lightweight *Stipula* (the black part); the additional rule for the higher-order functions is discussed afterwards. Among standard rules, [ASSET-SEND] delivers part of an asset $\texttt{h}$ to A.

This part, named u, is removed from the asset, *c.f.* the memory of the right-hand side configuration in the conclusion. In a similar way, [ASSET_UPDATE] moves a part u of an asset $\mathtt{h}$ to an asset $\mathtt{h}'$. For this reason, the final memory becomes $\ell[\mathtt{h} \mapsto v, \mathtt{h}' \mapsto v']$, where $v = \ell(\mathtt{h}) - u$ and $v' = \ell(\mathtt{h}') + u$. Rule [STATE-CHANGE] says that a contract changes state upon termination of the statement in the function body. The relevant rule is [FUNCTION] that defines invocations of (first-order) functions: the label of the transition specifies the party A performing the invocation and the function name $\mathtt{f}$ with the actual parameters. The transition may occur provided (i) the contract is in the state $\mathtt{@Q}$ that admits invocations of $\mathtt{f}$ from A, (ii) it is *idle*, and (iii) the code $\mathbb{D}$ contains a function $\mathtt{@Q\,A} : \mathtt{f}(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rightarrow \mathtt{@Q}'$ such that E is true in the memory ℓ updated with the actual parameters.

A contract `stipula C{ parties` $\overline{\mathtt{A}}$ `fields` $\overline{x}$ `assets` $\overline{\mathtt{h}}$ `init @Q` F } is triggered by executing $\mathtt{C}(\overline{A}, \overline{u})$ that corresponds to the *initial configuration*

$$\mathtt{parties}\,\overline{\mathtt{A}}\quad \mathtt{fields}\,\overline{\mathtt{x}}\quad \mathtt{assets}\,\overline{\mathtt{h}}\quad F \Vdash \mathtt{@Q}\,,\ [\overline{\mathtt{A}} \mapsto \overline{A}, \overline{\mathtt{x}} \mapsto \overline{u}, \overline{\mathtt{h}} \mapsto \overline{0}]\,,\ \_\,.$$

That is, parties' names are instantiated to parties' identities, fields are initialized to values $\overline{u}$ and the initial value of assets is $\mathtt{0}$.

In *higher-order Stipula* the declaration part of a configuration has the form $\mathbb{D} \lhd \mathbb{D}_1 \lhd \cdots \lhd \mathbb{D}_n$, where $\mathbb{D}$ is the declaration of the initial contract and $\mathbb{D}_1, \cdots, \mathbb{D}_n$ is a sequence of amendments. We recall that amendments $\mathbb{D}_i$ have shape

$$\mathtt{parties}\,\overline{\mathtt{A}'} = \overline{A'}\quad \mathtt{fields}\,\overline{\mathtt{z}}\quad \mathtt{assets}\,\overline{\mathtt{k}}\quad F$$

that extends the declaration part of a contract by admitting initializations of parties' names.

Let $\mathbb{D} = \mathbb{D}_0 \lhd \cdots \lhd \mathbb{D}_n$; we let *parties*($\mathbb{D}$), *assets*($\mathbb{D}$) and *fields*($\mathbb{D}$) be the union of party names, asset names and field names defined in every $\mathbb{D}_i$, with $0 \leq i \leq n$, respectively. The sequence $\mathbb{D}_0 \lhd \cdots \lhd \mathbb{D}_n$ is defined provided that, for every $i, j \in 0..n$, $i \neq j$: $\mathit{parties}(\mathbb{D}_i) \cap \mathit{parties}(\mathbb{D}_j) = \varnothing$ and $\mathit{assets}(\mathbb{D}_i) \cap \mathit{assets}(\mathbb{D}_j) = \varnothing$ and $\mathit{fields}(\mathbb{D}_i) \cap \mathit{fields}(\mathbb{D}_j) = \varnothing$. In the following, with an abuse of notation, we will use $\mathbb{D}$, $\mathbb{D}'$, $\cdots$ to range over sequences $\mathbb{D}_0 \lhd \cdots \lhd \mathbb{D}_n$.

We then extend the *selection operation* to declaration parts of *higher-order Stipula* configurations ($\mathbb{D}'$ is a single amendment):

$$(\mathbb{D} \lhd \mathbb{D}')[\mathtt{@Q\,A} : \mathtt{f}]_{\ell,\overline{u},\overline{v}} = \begin{cases} \mathbb{D}'[\mathtt{@Q\,A} : \mathtt{f}]_{\ell,\overline{u},\overline{v}} & \text{if } \mathbb{D}'[\mathtt{@Q\,A} : \mathtt{f}]_{\ell,\overline{u},\overline{v}} \neq \varnothing \\ \mathbb{D}[\mathtt{@Q\,A} : \mathtt{f}]_{\ell,\overline{u},\overline{v}} & \text{otherwise} \end{cases}$$

That is, our selection returns the newest set of functions in the list of amendments whose guard E is `true`. When the function is higher-order, the selection returns $\mathbb{D}[\mathtt{@Q\,A} : \mathtt{F}]_{\ell,\varepsilon,\varepsilon}$ (*i.e.* fields and asset parameters are empty). Finally, let $\mathtt{P}$ range over *sequences* of items $\mathtt{p}$ that are $\mathtt{f}$ or $\mathtt{F}$, $\mathtt{A} : \mathtt{f}$ or $\mathtt{A} : \mathtt{F}$, $\mathtt{@Q\,A} : \mathtt{f}$ or $\mathtt{@Q\,A} : \mathtt{F}$. We define $\mathbb{D} \setminus \mathtt{P}$ by induction on the length of $\mathtt{P}$:

– $\mathbb{D} \setminus \varepsilon = \mathbb{D}$;

– $\mathbb{D} \setminus \mathtt{p} \cdot \mathtt{P} = \mathbb{D}' \setminus \mathtt{P}$, where $\mathbb{D}'$ is obtained from $\mathbb{D}$ by erasing (in every declaration in $\mathbb{D}$)
  - every function `f`, if $\mathtt{p} = \mathtt{f}$;
  - every function `f` that is invoked by `A`, if $\mathtt{p} = \mathtt{A} : \mathtt{f}$;
  - every function `f` that is invoked by `A` in a state `@Q`, if $\mathtt{p} = \mathtt{@Q}\ \mathtt{A} : \mathtt{f}$;
  - similarly for higher-order functions.

Remark 2. The sequence `P` allows the programmer to be more and more selective during the remove operation $\mathbb{D} \setminus \mathtt{P}$. However, the operation $\mathbb{D} \setminus \mathtt{P}$ removes function at every depth in $\mathbb{D}$. We might be less demanding, extending the directives with a “surface remove” that removes the more recent function only.

Every preliminary definition is in place, we therefore comment rule [HO-Function] defining higher-order function invocations. This rule addresses higher-order functions with a complete set of directives – the other type of invocations are sub-cases of it. Once the function `F` has been chosen, the actual arguments `P`, $\mathbb{D}'$ and B' are used as follows: functions in `P` are removed from the declaration part $\mathbb{D}$, which is then amended with the code $\mathbb{D}'$ (provided this operation is well-defined, *c.f.* the foregoing constraint about names in $\mathbb{D}'$) and the memory ℓ is updated with the binding of party names and the initialization of new asset names to `0`. We observe that new fields are not initialized: in case, the initialization must be explicitly performed in the body B (*c.f.* the third example in Section 3).

To illustrate the semantics, consider the `Deposit` contract in Table 2 where `Client` and `Farm` have identities *Client* and *Farm*, respectively, and `cost_flour` is assumed to be 2 (euro per kg). Let $\ell = [\mathtt{Client} \mapsto \mathit{Client},\ \mathtt{Farm} \mapsto \mathit{Farm}, \mathtt{cost\_flour} \mapsto 2, \mathtt{flour} \mapsto 0]$, $\mathbb{D}_{\mathtt{Dep}}$ be the declaration part of the `Deposit` contract and let

$$\begin{aligned} S &= \mathtt{h} \rightarrow \mathtt{Client} \quad \mathtt{h} \multimap \mathtt{flour} \\ S' &= (\mathtt{x/flour}) \times \mathtt{flour} \multimap \mathtt{Client} \quad \mathtt{w} \multimap \mathtt{Farm} \\ B &= \mathtt{"Pay\_in\_Advance"} \rightarrow \tilde{}\quad \mathtt{flour} \multimap \mathtt{Farm} \end{aligned}$$

We have the following transitions (in the rightmost column we write the rule that has been used); memories $\ell_1, \cdots, \ell_5$ are defined afterwards:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbb{D}_{\mathtt{Dep}} \Vdash \mathtt{@Standard}\,,\ \ell\,,\ \_} \\
\xrightarrow{\mathit{Farm}:\mathtt{send()\,[10]}} & \mathbb{D}_{\mathtt{Dep}} \Vdash \mathtt{@Standard}\,,\ \ell_1\,,\ S \Rightarrow \mathtt{@Standard} & [\textsc{Function}] \\
\xrightarrow{10 \rightarrow \mathit{Client}} & \mathbb{D}_{\mathtt{Dep}} \Vdash \mathtt{@Standard}\,,\ \ell_1\,,\ \mathtt{h} \multimap \mathtt{flour} \Rightarrow \mathtt{@Standard} & [\textsc{Value-Send}] \\
\longrightarrow & \mathbb{D}_{\mathtt{Dep}} \Vdash \mathtt{@Standard}\,,\ \ell_2\,,\ \_ \Rightarrow \mathtt{@Standard} & [\textsc{Asset-Update}] \\
\longrightarrow & \mathbb{D}_{\mathtt{Dep}} \Vdash \mathtt{@Standard}\,,\ \ell_2\,,\ \_ & [\textsc{State-Change}] \\
\xrightarrow{\mathit{Client}:\mathtt{buy(4)\,[8]}} & \mathbb{D}_{\mathtt{Dep}} \Vdash \mathtt{@Standard}\,,\ \ell_3\,,\ S' & [\textsc{Function}] \\
\xrightarrow{4 \multimap \mathit{Client}} & \mathbb{D}_{\mathtt{Dep}} \Vdash \mathtt{@Standard}\,,\ \ell_4\,,\ \mathtt{w} \multimap \mathtt{Farm} \Rightarrow \mathtt{@Standard} & [\textsc{Asset-Send}] \\
\xrightarrow{8 \multimap \mathit{Farm}} & \mathbb{D}_{\mathtt{Dep}} \Vdash \mathtt{@Standard}\,,\ \ell_5\,,\ \_ \Rightarrow \mathtt{@Standard} & [\textsc{Asset-Send}] \\
\longrightarrow & \mathbb{D}_{\mathtt{Dep}} \Vdash \mathtt{@Standard}\,,\ \ell_5\,,\ \_ & [\textsc{State-Change}] \\
\xrightarrow{\mathit{Farm}:\mathtt{Hardship}(\!|\,\_, \mathbb{D}, B\,|\!)} & \mathbb{D}_{\mathtt{Dep}} \lhd \mathbb{D} \Vdash \mathtt{@Standard}\,,\ \ell_5\,,\ B & [\textsc{HO-Function}]
\end{array}$$

where $\mathbb{D}$ is the code of Section 3 and

$$\begin{array}{lll} \ell_1 = \ell[\mathtt{h} \mapsto 10] & \ell_2 = \ell_1[\mathtt{h} \mapsto 0, \mathtt{flour} \mapsto 10] & \ell_3 = \ell_2[\mathtt{x} \mapsto 4, \mathtt{w} \mapsto 8] \\ \ell_4 = \ell_3[\mathtt{flour} \mapsto 6] & \ell_5 = \ell_4[\mathtt{w} \mapsto 0] & \end{array}$$

Remark 3. (*Higher-order*) *Stipula* admits a form of *nondeterminism*, called *internal* in the literature, that is problematic in juridical acts: when a party can invoke two homonymous functions. In this case the selection operator returns a set that is not a singleton and, according to [Function] and [HO-Function], the function that is executed is chosen randomly. This corresponds to those real legal contracts that contain contradictions, which are usually solved by a court. 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.

5 Constraining amendments

Up-to now *higher-order Stipula* enables parties to make any kind of amendment, which is considered too liberal by the current legal doctrines. Beside the limit represented by the counterparties' consent to amendings (which will be dealt with in Section 6), parties' freedom is often bound in legal system's mandatory rules (*cf.* the principle in Art. 1418 of the Italian Civil Code, the Art. 1:103 of PECL – the European Principle of Contract Law – and the Art. 1.4 of the international Unidroit Principles). For example, the legislator can impose or set limits to prices for basic commodities, employees' salary or loan interest rates. Additionally, parties themselves can decide to set constraints to their amendment power by declaring them in specific clauses.

In order to implement such possibility, we first discuss restriction that can be added at static-time. That is, when a contract is stipulated, parties agree on the type of amendments they might accept in the future. In particular, by means of a syntactic clause we are going to discuss below, we define a predicate $\mathbb{T}(\cdot)$ that takes amendments and verifies whether they comply or not with the restrictions in the clause. In this context, the rule [HO-Function] becomes (for readability sake, we rewrite the premises of [HO-Function]):

$$\text{[HO-Function-SC]} \quad \frac{\begin{array}{c} \mathtt{@Q\,A\colon F}(\!|\, X, Y, Z \,|\!)\{\, \mathtt{remove}\ X\ \mathtt{add}\ Y\ \mathtt{run}\ Z \,\} \in \mathbb{D}[\mathtt{@Q\ A : F}]_{\ell,\varepsilon,\varepsilon} \\ \mathbb{D}' = \mathtt{parties}\ \overline{\mathtt{A'}} = \overline{A'}\ \mathtt{fields}\ \overline{\mathtt{z}}\ \mathtt{assets}\ \overline{\mathtt{k}}\ F \qquad \ell(\mathtt{A}) = A \qquad \ell' = \ell[\overline{\mathtt{k}} \mapsto \overline{0}, \overline{\mathtt{A'}} \mapsto \overline{A'}] \\ \mathbb{T}(\mathtt{remove\ P\ add}\ \mathbb{D}'\ \mathtt{run}\ B) \end{array}}{\mathbb{D} \Vdash \mathtt{@Q}\,,\, \ell\,,\, \_ \xrightarrow{A:\mathtt{F}(\!|\mathtt{P},\mathbb{D}',B|\!)} \mathbb{D} \setminus \mathtt{P} \lhd \mathbb{D}' \Vdash \mathtt{@Q}\,,\, \ell'\,,\, B}$$

that enables the transition if $\mathbb{T}(\mathtt{remove\ P\ add}\ \mathbb{D}'\ \mathtt{run}\ B)$ is true. The predicate $\mathbb{T}(\cdot)$ is defined by the following clause

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

$T$ ::= constraints [ (parties: fixed;)?  (fields: $\overline{\texttt{z}}$ constant;)?
                      (assets: $\overline{\texttt{k}}$ not-decrease;)?  (reachable states: $\overline{\texttt{@Q}}$)? ]
```

where every constraint in T may be missing (when all the constraints are empty then "`constraints [ ]`" is omitted and we are back to the basic syntax). The constraint "`parties: fixed`" specifies that amendments cannot modify the set of parties. If this constraint was present in the `Deposit` contract of Table 2 then the amendment $\mathbb{D}'$ of Section 3 would have been rejected. The constraint "`fields:` $\overline{\texttt{z}}$ `constant`" disables updates of fields in $\overline{\texttt{z}}$. For example, if the field `rate` contains the interest rate of a loan, the parties may initially decide that the rate can never be changed (loan with fixed rate). In *higher-order Stipula* this may be simply enforced by "`fields: rate constant`". The constraint "`assets:` $\overline{\texttt{k}}$ `not-decrease`" protects private assets to be drained by unauthorised parties. For example, in the code of Table 2, only `Client` can withdraw from the asset `flour`. If this policy must not be changed during the contract lifetime, it is sufficient to insert the constraint "`assets: flour not-decrease`" that disallows amendments draining `flour` (on the contrary, addition of flour is always admitted; we remind that asset values sent during invocation are always nonnegative). Finally, the constraint "`reachable states:` $\overline{\texttt{Q}}$" guarantees that, whatever contract update is performed, the states in $\overline{\texttt{Q}}$ can be reached from the ending state of the amendment. This is because, for example, the corresponding functionalities cannot be disallowed forever.

Below we discuss the implementation of $\mathbb{T}$(`remove P add` $\mathbb{D}$ `run` B) that we are designing for our prototype [8], given a constraint clause in the code of the contract.

Fixed parties. This constraint is easy to implement: it is sufficient to verify that, no term `parties:` $\overline{\texttt{A}}$, with $\overline{\texttt{A}}$ not empty, belongs to $\mathbb{D}$.

Constant fields and not-decreasing assets. The technique for assessing the constraints about fields and assets amounts to parse the amendment and spot the problematic instructions. In particular, if `fields:` $\overline{\texttt{z}}$ `constant` and $\texttt{y} \in \overline{\texttt{z}}$ then both $\mathbb{D}$ and B must not contain the instruction $E \to \texttt{y}$. Similarly, if `assets:` $\overline{\texttt{k}}$ `not-decrease` and $\texttt{k}' \in \overline{\texttt{k}}$ then $\mathbb{D}$ and B must not contain the instructions $E \times \texttt{k}' \multimap \texttt{h}$ and $E \times \texttt{k}' \multimap \texttt{A}$. The predicate $\mathbb{T}(\cdot)$ uses the judgments $\overline{\texttt{f}};\overline{\texttt{h}} \vdash G$, where G ranges over $\mathbb{D}, B, F$, and S, which are formally defined by a type system whose key rules are in Table 4. The rules [T-Update], [T-Send], and [T-Asset-update] are the basic one for guaranteing `fields:` $\overline{\texttt{f}}$ `constant` and `assets:` $\overline{\texttt{h}}$ `not-decrease`; the other rules reduce the analysis to the components of a code. More precisely, according to [T-Amendment], $\mathbb{D}$, S `@Q` is correct provided that the body of every function in $\mathbb{D}$ satisfies $\mathbb{T}(\cdot)$ – premise $\overline{\texttt{f}};\overline{\texttt{h}} \vdash F$ – and the statement S satisfies $\mathbb{T}(\cdot)$ as well – premise $\overline{\texttt{f}};\overline{\texttt{h}} \vdash S$.

State reachability. In general, it is not possible to assess state reachability at static time because the values of guards of functions may depend on memories

$$\text{[T-Update]}\ \frac{\mathtt{g} \notin \overline{\mathtt{f}}}{\overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash E \to \mathtt{g}} \qquad \text{[T-Send]}\ \frac{\mathtt{k} \notin \overline{\mathtt{h}}}{\overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash E \times \mathtt{k} \multimap \mathtt{A}} \qquad \text{[T-Asset-update]}\ \frac{\mathtt{k} \notin \overline{\mathtt{h}}}{\overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash E \times \mathtt{k} \multimap \mathtt{h}'}$$

$$\text{[T-Cond]}\ \frac{\overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash S \quad \overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash S'}{\overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash \mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S'\,\}} \qquad \text{[T-Seq]}\ \frac{\overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash P \quad \overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash S}{\overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash P\ S}$$

$$\text{[T-Function]}\ \frac{\left(\overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash S\right)^{\mathtt{@Q\ A:\ f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}]\ (E)\{\ S\ \}\Rightarrow\mathtt{@Q}'\ \text{in}\ F}}{\overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash F} \qquad \text{[T-Amendment]}\ \frac{\begin{array}{c}\mathbb{D} = \mathtt{parties}\ \overline{\mathtt{A}'} = \overline{A'}\ \mathtt{fields}\ \overline{x}\ \mathtt{assets}\ \overline{h}\ \ F \\ B = \{\,S\,\} \Rightarrow \mathtt{@Q} \qquad \overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash F \qquad \overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash S\end{array}}{\overline{\mathtt{f}}\,;\overline{\mathtt{h}} \vdash \mathbb{D}, B}$$

Table 4. Key rules for verifying constant fields and not-decreasing assets

and actual parameters. That is the following technique may return *false positives* (while it never returns *false negatives*: if a state is unreachable then there is no computation ending in that state). False positives are ruled out only in the restricted case when the functions in the contract code and in the amendments are unguarded.

Following [5], we use the predicate `is_in`: $\mathtt{@Q\ A : f\ @Q}''\ \mathtt{is\_in}\ \mathbb{D}$ holds true if

– $\mathbb{D}$ is a single declaration part and there is $\mathtt{@Q\ A : f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}]\,(\mathtt{E})\{\,\mathtt{S}\,\} \Rightarrow \mathtt{@Q}'$ in $\mathbb{D}$;
– or $\mathbb{D} = \mathbb{D}' \lhd \mathbb{D}''$, where $\mathbb{D}''$ is a single declaration part, and either $\mathtt{@Q\ A} : \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}]\,(\mathtt{E})\{\,\mathtt{S}\,\} \Rightarrow \mathtt{@Q}'$ in $\mathbb{D}'$ or $\mathtt{@Q\ A}: \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}]\,(\mathtt{E})\{\,\mathtt{S}\,\} \Rightarrow \mathtt{@Q}'$ in $\mathbb{D}''$.

The predicate $\mathtt{@Q\ A : f\ @Q}''\ \mathtt{is\_in}\ \mathbb{D}$ is false otherwise. Notice that we are considering first-order functions only.

The set of reachable states in $\mathbb{D}$ from `@Q`, noted $\mathbb{Q}_{\mathtt{@Q}}$, is the least set such that

1. $\mathtt{@Q} \in \mathbb{Q}_{\mathtt{@Q}}$;
2. if $\mathtt{@Q}' \in \mathbb{Q}_{\mathtt{@Q}}$ and $\mathtt{@Q}'\ \mathtt{A : f\ @Q}''\ \mathtt{is\_in}\ \mathbb{D}$ then $\mathtt{@Q}'' \in \mathbb{Q}_{\mathtt{@Q}}$.

We notice that $\mathbb{Q}_{\mathtt{@Q}}$ is always finite and can be easily computed by a standard fixpoint technique that must be run in correspondence of every higher-order function invocation. For example, in Section 3, the invocation

$$\mathtt{Farm\ :\ Hardship}(\!|\,\varepsilon, \mathbb{D}, \{\mathtt{"Pay\_in\_Advance"} \to^{\sim}\ \ \mathtt{flour} \multimap \mathtt{Farm}\} \Rightarrow \mathtt{@Excp}\,|\!)$$

returns the declaration part $\mathbb{D}_{\mathtt{Dep}} \lhd \mathbb{D}$ where $\mathtt{@Standard} \notin \mathbb{Q}_{\mathtt{@Excp}}$, while the second amendment gives a declaration part $\mathbb{D}_{\mathtt{Dep}} \lhd \mathbb{D} \lhd \mathbb{D}'$ where $\mathtt{@Standard} \in \mathbb{Q}_{\mathtt{@Standard}}$.

When `reachable states:` $\overline{\mathtt{@Q}}$ is a constraint and $\mathbb{D}$ is the current declaration part, the predicate $\mathbb{T}(\mathtt{remove\ P\ add}\ \mathbb{D}'\ \mathtt{run}\ \{\,S\,\} \Rightarrow \mathtt{@Q}')$ verifies that $\overline{\mathtt{@Q}} \subseteq \mathbb{Q}_{\mathtt{@Q}'}$ when the declaration part is $\mathbb{D} \setminus \mathtt{P} \lhd \mathbb{D}'$.

We conclude by discussing the presence of false positives in $\mathbb{T}(\cdot)$ with an example. Consider the `Deposit` contract in Table 2 and change the final state of `buy` into `@End` (the `Client` can buy only one time). Then assume the presence of the constraint clause `constraints [ reachable states: @End ]` and verify the predicate $\mathbb{T}(\cdot)$ for the initial declaration part $\mathbb{D}_{\texttt{Dep}}$. It is easy to check that $\texttt{@End} \in \mathbb{Q}_{\texttt{@Standard}}$. However, if `cost_flour` has been initialized with a negative value (because of an error) then no transition `buy` will ever be performed because of its guard that is always false and `@End` will never be reached. Overcoming this issue is out of the scope of this paper. A possible technique could use the definition of $\mathbb{Q}_{\texttt{@Q}}$ to synthesize computations and verify the guards by means an (off-the-shelf) constraint solver technique.

6 Agreement on amendments

The Unidroit Art. 6.2.3 states that a *contract may be supplemented, amended, or modified only by the mutual agreement of the parties.* That is, to deal with this principle, it is necessary to enforce an agreement protocol between parties in correspondence of runtime amendments. Actually, the full *Stipula* language already retains an agreement clause between parties that corresponds to the so-called "meeting of the minds": every one must accept the terms of the contract and the legal effects of the *Stipula* contract are triggered by the achievement of the agreement (see rule [Agree] in [7]; this feature has been omitted in this contribution because we are addressing is a lightweight version of the language).

Below we propose an extension of *higher-order Stipula* with an additional agreement clause that occurs in correspondence of every amendment. To define the rule, let A ACCEPTS H IN ℓ be a predicate that takes a directive H and verifies whether it complies or not with A's policy in the memory ℓ. It is worth to notice that the predicate depends on the memory; therefore the policy of A might change in accordance with the updates. In particular, if ℓ stores a timestamp (the semantics of full *Stipula* has a global clock by which the events are modelled), then ACCEPT may change from time to time. In this context, the rule [HO-Function] becomes (we also rewrite the premises):

$$\text{[HO-Function-AGREE]}\quad \frac{\begin{array}{c}\texttt{@Q A: F}(\!|\, X, Y, Z \,|\!)\{\,\texttt{remove}\ X\ \texttt{add}\ Y\ \texttt{run}\ Z\,\} \in \mathbb{D}[\texttt{@Q A : F}]_{\ell,\varepsilon,\varepsilon} \\ \mathbb{D}' = \texttt{parties}\ \overline{\texttt{A}'} = \overline{A'}\ \texttt{fields}\ \overline{x}\ \texttt{assets}\ \overline{\texttt{h}}\ F \qquad \ell(\texttt{A}) = A \qquad \ell' = \ell[\overline{\texttt{h}} \mapsto \overline{0}, \overline{\texttt{A}'} \mapsto \overline{A'}] \\ \big(\ell(\texttt{A}'')\ \textsc{accepts}\ \texttt{remove P add}\ \mathbb{D}'\ \texttt{run}\ B\ \textsc{in}\ \ell\big)^{\texttt{A}'' \in parties(\mathbb{D})\ \&\&\ \ell(\texttt{A}'') \neq A}\end{array}}{\mathbb{D} \Vdash \texttt{@Q}\,,\,\ell\,,\,\_ \xrightarrow{A:\texttt{F}(\!|\,\texttt{P},\mathbb{D}',B\,|\!)} \mathbb{D} \setminus \texttt{P} \lhd \mathbb{D}' \Vdash \texttt{@Q}\,,\,\ell'\,,\,B}$$

We notice that, according to [HO-Function-AGREE], the acceptance of the directive is restricted to parties in $\mathbb{D}$: the new parties in $\mathbb{D}'$ have nothing to accept. For instance, in the first example of Section 3, we have the invocation

$$\texttt{Farm : Hardship}(\!|\,\varepsilon, \mathbb{D}, \{\texttt{"Pay\_in\_Advance"} \to \tilde{}\ \ \texttt{flour} \multimap \texttt{Farm}\} \Rightarrow \texttt{@Excp}\,|\!)$$

($\mathbb{D}$ refers to the declaration part defined in Section 3). At this point, for the new code becoming operational and enter into force, *Client* must satisfies the predicate

$$\textit{Client}\ \textsc{accepts}\ \texttt{add}\ \mathbb{D}\ \texttt{run}\ \{\texttt{"Pay\_in\_Advance"} \to \tilde{}\ \ \texttt{flour} \multimap \texttt{Farm}\} \Rightarrow \texttt{@Excp}\ \textsc{in}\ \ell'$$

assuming that ℓ and ℓ' are the memories before and after the transition, respectively. (In this case we have omitted the `remove` directive because it is empty.)

7 Related works

Higher-order have been widely used in programming languages to pass functions as arguments to other functions, thus allowing to easily model closures and currying (*cf.* `Haskell`, `JavaScript`, and lambdas in `C++` and `Java`). As regards languages for legal contracts, up-to our knowledge, no-one addresses amendments of contracts. In particular, the literature reports a number of languages and frameworks that aim at transforming legal semantic rules into code, *e.g.* [13, 10, 9, 14]. 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 `Java`. *Stipula*, with its distinctive primitives and legal design patterns, aims to be intermediate between a specification language and a high-level programming language. That is, *Stipula* and its higher-order extension can be considered a *legal calculus* in the terminology of [2], similar to `Orlando` [1] that has been designed for modeling conveyances in property law and `Catala` [17] for modeling statutes and regulations clauses in the fiscal domain.

Recently, there has been increasing interest in smart contract languages because they allow to define programs that can manage and transfer assets. These programs run on distributed networks whose nodes store a common state (that also includes the programs themselves) in the form of a blockchain. Due to the immutability of information stored on a blockchain, several projects have proposed legal frameworks that target smart contracts on Ethereum [4], such as OpenLaw [22] and Lexon [16]. Amending the code of these frameworks is equivalent to upgrading Ethereum smart contracts, which is not straightforward, as once a smart contract is deployed on a blockchain, it is immutable. However, since upgrading may be necessary to fix vulnerabilities or to change smart contract business logic, designers have proposed a number of patterns for safely modifying a contract still preserving the immutability of the blockchain [12]. These pattern rely either (*i*) on decoupling the data storage from the business logic of a contract or (*ii*) on the usage of proxies. In case (*i*), the contract has been defined in such a way that the business logic is accessed by an address stored in the contract (this is similar to our requirement that a contract has an hardship function). This means that updating the business logic amounts to rewrite a new logic, store it in the blockchain at a (new) address `x` and use `x` to update the address stored in the contract. In case (*ii*), the users interact with a proxy contract rather than the original contract, whose data and functionalities are accessed by means of addresses stored in the proxy. Therefore, updating (both the state and the business logic of) the contract amounts to change the addresses stored in the proxy. Proxies are also used for implementing contract versioning: the address of the contract is actually that of a package and ad-hoc policies may direct the invocation to one version or another. When several versions do coexist (*cf.* diamond patterns [19]) and a protocol may dispatch an invocation to one version or another, we get a smart contract concept similar to our operation $\mathbb{D} \lhd \mathbb{D}'$.

Clearly, the foregoing solutions allow neither a control on whom is going to modify the contract nor an agreement between the parties. In fact, *higher-order Stipula* turns out to be at a higher level of abstraction than addresses or proxies, thus allowing reasonings about amendments that integrate well with the other features of the language. Said otherwise, *higher-order Stipula* seems more appropriate and more faithful in representing the structure of a legal contract and the procedure for amending it.

In designing *higher-order Stipula* we have been inspired by operations of Delta-Oriented Programming [15] that has been conceived for implementing software product lines. In this paradigm, deltas are codes that are attached to products and can be combined to obtain complex products starting from a core feature. Compliance and other correctness properties can be verified at static time. On the contrary, in *higher-order Stipula* amendments are not known when the contract is stipulated and every analysis must be postponed at runtime.

8 Conclusions

This paper discusses the amendments of legal contracts in *Stipula* by resorting to higher-order. Our solution handles both amendments where the contract code is completely modified and substituted as well as those where the new code has to coexist with the old one. The latter case, though, may require particular attention, especially to the conditions laid out in the new functions. A wrongly formulated condition could affect the order of codes priorities. This, in turn, could result in an unwanted function overriding or, *vice-versa*, in the persistence uptime of a function that had to be overridden.

We believe that the higher-order extension is crucial for the effective applicability of legal contracts in real-world scenarios. Specifically, it can be used (in the full *Stipula* language which also includes events) to handle new events by passing a function to be executed when an event occurs. This enables more flexible and modular event handling that can account for unforeseen circumstances at the time the contract was initiated. We are already experimenting the higher-order extension of the *Stipula* prototype (that is available on-line at [8]). The higher-order extension admits functions that input codes; these codes are compiled on-the-fly and added to the contract (the compilation also includes a type inference analysis, see [7]). In correspondence of every invocation, a selection function retrieves the right function code as specified by rule [HO-Function].

Future works on the matter shall deal with analyzing a set of more complex use-cases and to implement the policies discussed in Sections 5 and 6. It is worth to remark that our prototype, taking inspiration from visual interfaces as in [21], is integrated with a user-friendly and easy-to-use programming interface. We hope that this additional feature will allow us to collect comments and reports of the proposal by non-expert users.

Acknowledgements. We are grateful to Silvia Crafa for the many insightful discussions about *Stipula* and Adele Veschetti for prototyping both *Stipula* and

higher-order Stipula. We also thank the anonymous Coordination referees for the detailed suggestions that considerably improved the paper.

References

1. Shrutarshi Basu, Nate Foster, and James Grimmelmann. Property conveyances as a programming language. In *Proc. 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pages 128–142, New York, USA, 2019. Association for Computing Machinery.
2. Shrutarshi Basu, Anshuman Mohan, James Grimmelmann, and Nate Foster. Legal calculi. Technical report, ProLaLa 2022 ProLaLa Programming Languages and the Law, 2022. At `https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi`.
3. Fabio Bortolotti. Force Majeure and Hardship Clauses – Introductory note and commentary. Technical report, International Chamber of Commerce, 2020.
4. Vitalik Buterin. Ethereum white paper. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2013.
5. Silvia Crafa and Cosimo Laneve. Liquidity analysis in resource-aware programming. In *In Proc. 18th Int. Conference, FACS 2022*, volume 13712 of *Lecture Notes in Computer Science*, pages 205–221. Springer, 2022.
6. Silvia Crafa, Cosimo Laneve, and Giovanni Sartor. Stipula: a domain specific language for legal contracts. Presented at the Int. Workshop Programming Languages and the Law, January 16, 2022.
7. Silvia Crafa, Cosimo Laneve, Giovanni Sartor, and Adele Veschetti. Pacta sunt servanda: legal contracts in Stipula. *Science of Computer Programming*, 225, January 2023.
8. Silvia Crafa, Cosimo Laneve, and Adele Veschetti. The Higher-order Stipula Prototype, July 2022. Available on github: `https://github.com/stipula-language`.
9. Joost T. de Kruijff and H. Hans Weigand. An introduction to commitment based smart contracts using reactionruleml. In *Proc. 12th Int. Workshop on Value Modeling and Business Ontologies (VMBO)*, volume 2239, pages 149–157. CEUR-WS.org, 2018.
10. Joost T. de Kruijff and H. Hans Weigand. Introducing commitruleml for smart contracts. In *Proc. 13th Int. Workshop on Value Modeling and Business Ontologies (VMBO)*, volume 2383. CEUR-WS.org, 2019.
11. Marcel Fontaine and Filip De Ly. *Drafting International Contracts*. BRILL, 2006.
12. Ethereum Foundation. Upgrading smart contracts. `https://ethereum.org/en/developers/docs/smart-contracts/upgrading`, 2023.
13. Christopher K. Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st Int. Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 210–215, 2016.
14. Xiao He, Bohan Qin, Yan Zhu, Xing Chen, and Yi Liu. Spesc: A specification language for smart contracts. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 132–137, 2018.
15. Roberto E. Lopez-Herrejon, Don Batory, and William Cook. Evaluating support for features in advanced modularization technologies. In Andrew P. Black, editor, *In Proc. ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer, 2005.

16. Lexon Foundation. Lexon Home Page. `http://www.lexon.tech`, 2019.
17. Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: A programming language for the law. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.
18. Eliza Mik. Smart contracts terminology, technical limitations and real world complexity. *Law, Innovation and Technology*, 9:269–300, 2017.
19. Nick Mudge. How diamond upgrades work. `https://dev.to/mudgen/how-diamond-upgrades-work-417j`, 2022.
20. Davide Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In *Proceedings of TAPSOFT'93*, volume 668 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1993.
21. Tim Weingaertner, Rahul Rao, Jasmin Ettlin, Patrick Suter, and Philipp Dublanc. Smart contracts using blockly: Representing a purchase agreement using a graphical programming language. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 55–64, 2018.
22. Aaron Wright, David Roon, and ConsenSys AG. OpenLaw Web Site. `https://www.openlaw.io`, 2019.