Legal Contracts amending with *Stipula*

Cosimo Laneve[1], Alessandro Parenti[2], and Giovanni Sartor[2]

[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 code. 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 [6] 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 [5]. 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 the 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 make performance impossible or impracticable (force majeure) or substantially upset the economic balance of the contract (hardship) [3, 13]. 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 will be entitled to amend the contract to obtain its adaptation 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 [16]. 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 [18], we admit that function invocations may carry *codes* that patch the previous one. 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 at odd with the current legal doctrines. 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 Higher-order *Stipula*

For simplicity, we extend a *lightweight* version of the language in [6] with a higher-order feature (the full language has also the *agreement clause* and *events*). This allows us to avoid discussions that are out of the scope of this paper. Disjoint sets of names are used: *contract names*, ranged over by $\mathtt{C}$, $\mathtt{C}'$, $\cdots$; names referring to digital identities, called *parties*, ranged over by $\mathtt{A}$, $\mathtt{A}'$, $\cdots$; *function names* ranged over $\mathtt{f}$, $\mathtt{g}$, $\cdots$; *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; *code parameters*, ranged over X, Y. Assets and generic contract's fields are syntactically set apart since they have different semantics, similarly for functions'

Functions	F	::=	$\_ \mid \texttt{@Q A}: f(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rightarrow \texttt{@Q}'\,F \mid \texttt{@Q A}: f(\overline{y})[\overline{k}](\!|\,X\,|\!)\,F$
Prefixes	P	::=	$E \to x \mid E \to \texttt{A} \mid E \times h \multimap h' \mid E \times h \multimap \texttt{A}$
Statements	S	::=	$\_ \mid P\,S \mid \texttt{if}\,(E)\,\{\,S\,\}\,\texttt{else}\,\{\,S\,\}\,S$
Expressions	E	::=	$v \mid V \mid E\,\texttt{op}\,E \mid \texttt{uop}\,E$

Table 1. Syntax of *higher-order Stipula*

parameters. Names of assets, fields and (assets and fields) parameters are generically ranged over by V. Names $\texttt{Q}$, $\texttt{Q}'$, $\cdots$ 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 *higher-order Stipula* contract is

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

where $\texttt{C}$ 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 $\texttt{Q}$. The contract body also includes the sequence F of functions, whose syntax is defined in Table 1. It is assumed that there is no clash of names of parties, fields, assets and functions' parameters.

There are two declarations of functions: a *standard* one and a *higher-order* one. They both highlight who is the caller party $\texttt{A}$, the state $\texttt{Q}$ when the invocation is admitted and the name of the function. The standard 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* $\{\ S\ \} \Rightarrow \texttt{@Q}'$ specifies the *statement part* S and the state $\texttt{Q}'$ of the contract when the function execution terminates. The higher-order declaration $\texttt{@Q A}: f(\overline{y})[\overline{k}](\!|\,X\,|\!)$ has an additional parameter X that represents code (it is written in brackets $(\!|\cdot|\!)$). This code is in the scope of the formal parameters $\overline{y}$ and $\overline{k}$. Notice that higher-order declarations have no body: it is intended that the body of f is defined in X.

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 fields and on assets, respectively. The prefix $E \to x$ updates the field or the parameter x with the value of E; $E \to \texttt{A}$ sends the value of E to the party $\texttt{A}$; $E \multimap h, h'$ subtracts the value of E from the asset h and adds it to h', $E \multimap h, \texttt{A}$ subtracts the value of E from the asset h and transfers it to $\texttt{A}$. (The semantics in Section 4 will enforce that assets never have negative values.) Statements also include a *conditionals* $\texttt{if}\,(E)\,\{\,S\,\}\,\texttt{else}\,\{\,S'\,\}$ with the standard semantics. In the rest of the paper we will always abbreviate $h \multimap h, h'$ and $h \multimap h, \texttt{A}$ (which are very usual, indeed) into $h \multimap h'$ and $h \multimap \texttt{A}$, respectively. We also use " ˜ " to address all the parties. For instance, if the parties are $\texttt{A}$ and $\texttt{B}$, then $\texttt{"hello"} \to \tilde{}$ means $\texttt{"hello"} \to \texttt{A}\quad \texttt{"hello"} \to \texttt{B}$.

Expressions E include constant values v, which may be strings, reals, booleans, and asset values, names X of either assets, fields or parameters, 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 *divisible* resources (*e.g.* digital currencies). For simplicity, divisible asset constants are assumed to be identical to positive real numbers and, as discussed above, cannot assume negative values. Relational operations (`<`, `>`, `<=`, `>=`, `==`) are available between any expression.

A contract named `C` is invoked by `C`$(\overline{A}, \overline{u})$ that corresponds to a configuration (see below) where the initial state is the one specified in the `init` clause, the parties' actual identities are $\overline{A}$, the fields' values are $\overline{u}$ and the assets' values are 0. 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 a same party may have two roles in a legal contract.

3 Examples

Because of the molteplicity 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. The specific purpose of our examples was to explain the technical functioning of the *higher-order* to modify Stipula contracts. For this reason, we started from a particlar type of amendment case that is commonly found in practice, i.e. *hardship* cases[13], and built a simplistic representation of contractual relationship around it. It will be a focus for future works to test *Stipula* with the representation of more complex, context-specific contracts.

Table 2 illustrates a first simple scenario of amending a *Stipula* contract: the modification affects the whole body of the contract and its code is completely changed and substituted by the new code. We call this modification *additive*.

A `Client` (restaurant) contracts with a `Farm` to pay flour at a given cost. The protocol is the following: `Farm` sends the flour to the contract (function `send`) and the good is stored in the `flour` asset: no delivery to `Client` is done till he pays for it. The function `buy` takes in input a value `x` denoting that the client wants to buy flour and an asset `w` representing the money he wants to spend. The `Client` takes the corresponding amount of flour (provided it is in the deposit), the asset `flour` is updated correspondingly and the money `w` is transferred to `Farm`. `Client` and `Farm` also decide to include in the contract a hardship clause, 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)

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

    @Standard Farm: send(x)[h]{ x,h → Client     h —∘ flour } ⇒ @Standard

    @Standard Client: buy(x)[w] (w <= flour×cost_flour){
        x,w → Farm     (1/cost_flour)×w —∘ flour, Client     w —∘ Farm
        } ⇒ @Standard

    @Standard ~ : hardship(x)[]⦇X⦈
}
```

Table 2. The Flour Delivery with Hardship

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

$$\texttt{Farm : hardship("Pay\_in\_Advance")[]}\llparenthesis H \rrparenthesis$$

where $H = \mathtt{C}_H\ \{\mathtt{x} \to \tilde{}\quad \mathtt{flour} \multimap \mathtt{Farm}\} \Rightarrow \mathtt{@Excp}$ and $\mathtt{C}_H$ is

```
assets wallet

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

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

@Excp ~ : hardship(x)[]⦇X⦈
```

That is, the code may specify new assets, new functions and a body (which is the body of the `hardship` function). In this case, the code $\mathtt{C}_H$ specifies that the `Client` pays in advance by invoking the function `order`. In particular, `Farm` receives 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 body in H empties the `flour` asset returning the amount to `Farm` and let 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 wants 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, `Farm` invokes

$$\texttt{Farm : hardship("Back\_to\_Standard\_and\_upgrade\_flour\_price")[]}\llparenthesis H' \rrparenthesis$$

where $H' = \mathtt{C}_{H'}\ \{\mathtt{x} \to \tilde{}\quad \mathtt{cost\_flour} + 0.2 \times \mathtt{cost\_flour} \to \mathtt{cost\_flour}\} \Rightarrow \mathtt{@Standard}$ and $\mathtt{C}_{H'}$ is

```
parties Government = Govern

@Standard Client: buy(x)[w] (w <= flour×cost_flour){
    x,w → Farm     (1/cost_flour)×w —∘ flour, Client
    0.2 × w —∘ w, Government     w —∘ Farm
    } ⇛ @Standard

{ x → ~   cost_flour + 0.2×cost_flour → cost_flour} ⇛ @Standard
```

This new code is extending the parties with a new one (`Government` whose id is Govern) that 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. 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.

4 Semantics

The semantics of *higher-order Stipula* is defined operationally by means of a transition relation $\mathbb{C} \xrightarrow{\mu} \mathbb{C}'$, where $\mathbb{C}, \mathbb{C}'$ are *configurations*, *i.e.* tuples $\mathtt{C} \Vdash \mathtt{Q}\,,\, \ell\,,\, \Sigma$ where

- $\mathtt{C}$ is the contract code (in *higher-order Stipula* it is *structured* because of the amendments, see below);
- $\mathtt{Q}$ is a state of $\mathtt{C}$;
- ℓ, called *memory*, is a mapping 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 is the (possibly empty) residual of a function body, *i.e.* Σ is either $\_$ (idle) or a term $S \Rightarrow \mathtt{@Q}$.

We also use the *evaluation function* $[\![E]\!]_\ell$ returns the value of E in the memory ℓ. In particular:

- $[\![v]\!]_\ell = v$ for values, $[\![V]\!]_\ell = \ell(V)$ for names of assets, fields and parameters (V cannot be a code).
- let $\underline{uop}$ and $\underline{op}$ be the semantic operations corresponding to uop and op, then $[\![uop\, E]\!]_\ell = \underline{uop}\, v$, $[\![E\, op\, E']\!]_\ell = v\, \underline{op}\, v'$ with $[\![E]\!]_\ell = v, [\![E']\!]_\ell = v'$.

In *higher-order Stipula* contract codes are $\mathtt{C} \lhd \mathtt{C}_1 \lhd \cdots \lhd \mathtt{C}_n$, where $\mathtt{C}$ is the initial contract and $\mathtt{C}_1, \cdots, \mathtt{C}_n$ are sequences of amendments. With an abuse of notation,

we will range over $\mathtt{C} \lhd \mathtt{C}_1 \lhd \cdots \lhd \mathtt{C}_n$ with $\mathtt{C}$, $\mathtt{C}'$, $\cdots$. The amendments $\mathtt{C}_i$ have the shape

$$\text{parties } \overline{\mathtt{A}} = \overline{A} \quad \text{fields } \overline{x} \quad \text{assets } \overline{h} \quad F$$

(parties $\overline{\mathtt{A}} = \overline{A}$, fields $\overline{x}$ and assets $\overline{h}$ may be missing and F may be empty). We notice that, as regards parties, the codes $\mathtt{C}_i$ also contain their inizialization. Let *parties*($\mathtt{C}$), *assets*($\mathtt{C}$) and *fields*($\mathtt{C}$) be the union of party names, asset names and field names defined in every $\mathtt{C}, \mathtt{C}_1, \cdots, \mathtt{C}_n$, respectively. The operation $\mathtt{C} \lhd \mathtt{C}'$ is defined provided *parties*($\mathtt{C}$) $\cap$ *parties*($\mathtt{C}'$) $= \varnothing$ and *assets*($\mathtt{C}$) $\cap$ *assets*($\mathtt{C}'$) $= \varnothing$ and *fields*($\mathtt{C}$) $\cap$ *fields*($\mathtt{C}'$) $= \varnothing$.

Finally, let

$$\mathtt{C}[\mathtt{@Q\,A} : f]_{\ell,\overline{u},\overline{v}} = \begin{cases} \mathtt{@Q\,A} : f(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rrightarrow \mathtt{@Q}' & \text{if } \mathtt{@Q\,A} : f(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rrightarrow \mathtt{@Q}' \text{ belongs to } \mathtt{C} \\ & \text{and } [\![E]\!]_{\ell[\overline{y} \mapsto \overline{u}, \overline{k} \mapsto \overline{v}]} = \mathtt{true} \\ \_ & \text{otherwise} \end{cases}$$

That is, $\mathtt{C}[\mathtt{@Q\,A} : f]_{\ell,\overline{u},\overline{v}}$ returns a function $\mathtt{@Q\,A} : f(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rrightarrow \mathtt{@Q}'$ if this function belongs to $\mathtt{C}$ *and* the expression E is `true` in the memory $\ell[\overline{y} \mapsto \overline{u}, \overline{k} \mapsto \overline{v}]$. If there are several function in $\mathtt{C}$ satisfying this constraint, the result is non-deterministic. Then we define the *dispatch* operation of a contract $\mathtt{C}$ amended by $\mathtt{C}'$ as follows

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

That is, our dispatch returns the newest function in the list of amendments whose guard E is `true`.

Table 3 reports the definition of $\mathbb{C} \xrightarrow{\mu} \mathbb{C}'$. Among standard rules rules, [State-Change] says that a contract changes state when the execution of the statement in the function's body terminates. Rule [Asset-Send] delivers part of an asset `h` to A. This part, named v, is removed from the asset, *c.f.* the memory of the right-hand side state in the conclusion. In a similar way, [Asset_Update] moves a part v of an asset `h` to an asset `h'`. For this reason, the final memory becomes $\ell[\mathtt{h} \mapsto v', \mathtt{h}' \mapsto v'']$, where $v' = \ell(\mathtt{h}) - v$ and $v'' = \ell(\mathtt{h}') + v$. The relevant rules are [Function] and [HO-Function]. The former 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, (*ii*) it is *idle*, and (*iii*) the code `C` contains a function $\mathtt{@Q\,A} : f(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rrightarrow \mathtt{@Q}'$ such that E is true in the memory ℓ updated with the actual parameters. By definition of dispatch, the chosen function is the one that has not been amended. Rule [HO-Function] defines higher-order. In this case the invocation also carries a code H with a body that will be executed. This features augments the expressivity because the body may address field and asset names defined in H and a state that is undefined in the previous code – see the examples in Section 3.

[State-Change]

$$\mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,\_ \Rightarrow @\mathtt{Q}' \longrightarrow \mathtt{C} \Vdash \mathtt{Q}'\,,\,\ell\,,\,\_$$

[Value-Send]

$$\frac{[\![E]\!]_\ell = v \quad \ell(\mathtt{A}) = A}{\mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,E \to \mathtt{A}\ \Sigma \xrightarrow{v \to A} \mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,\Sigma}$$

[Asset-Send]

$$\frac{\ell(\mathtt{A}) = A \quad 0 \le [\![E]\!]_\ell \le 1 \quad [\![h - E \times h]\!]_\ell = v}{\mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,E \times h \multimap \mathtt{A}\ \Sigma \xrightarrow{v \multimap A} \mathtt{C} \Vdash \mathtt{Q}\,,\,\ell[h \mapsto v]\,,\,\Sigma}$$

[Field-Update]

$$\frac{[\![E]\!]_\ell = v}{\mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,E \to x\ \Sigma \longrightarrow \mathtt{C} \Vdash \mathtt{Q}\,,\,\ell[x \mapsto v]\,,\,\Sigma}$$

[Asset-Update]

$$\frac{0 \le [\![E]\!]_\ell \le 1 \quad [\![h - E \times h]\!]_\ell = v \quad [\![h' + E \times h]\!]_\ell = v' \quad \ell' = \ell[h \mapsto v, h' \mapsto v']}{\mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,E \times h \multimap h'\ \Sigma \longrightarrow \mathtt{C} \Vdash \mathtt{Q}\,,\,\ell'\,,\,\Sigma}$$

[Cond-true]

$$\frac{[\![E]\!]_\ell = \mathtt{true}}{\mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,\mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S'\,\}\ \Sigma \longrightarrow \mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,S\ \Sigma}$$

[Cond-false]

$$\frac{[\![E]\!]_\ell = \mathtt{false}}{\mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,\mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S'\,\}\ \Sigma \longrightarrow \mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,S'\ \Sigma}$$

[Function]

$$\frac{\mathtt{C}[@\mathtt{Q}\,\mathtt{A} : f]_{\ell,\overline{u},\overline{v}} = @\mathtt{Q}\,\mathtt{A} : f(\overline{y})[\overline{k}]\,(E)\{\,S\,\} \Rightarrow @\mathtt{Q}' \quad \ell(\mathtt{A}) = A \quad \ell' = \ell[\overline{y} \mapsto \overline{u}, \overline{k} \mapsto \overline{v}]}{\mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,\_ \xrightarrow{A:f(\overline{u})[\overline{v}]} \mathtt{C} \Vdash \mathtt{Q}\,,\,\ell'\,,\,S \Rightarrow @\mathtt{Q}'}$$

[HO-Function]

$$\frac{\begin{array}{c}\mathtt{C}[@\mathtt{Q}\,\mathtt{A} : f]_{\ell,\overline{u},\overline{v}} = @\mathtt{Q}\,\mathtt{A}\colon f(\overline{y})[\overline{k}](\!|\,X\,|\!) \quad \ell(\mathtt{A}) = A \quad H = \mathtt{C}'\ \{\,S\,\} \Rightarrow @\mathtt{Q}' \\ \mathtt{C}' = \mathit{parties}\ \overline{\mathtt{B}} = \overline{B}\ \ \mathit{fields}\ \overline{x}\ \ \mathit{assets}\ \overline{h}\ \ F \\ \ell' = \ell[\overline{y} \mapsto \overline{u}, \overline{k} \mapsto \overline{v}, \overline{h} \mapsto \overline{0}, \overline{\mathtt{B}} \mapsto \overline{B}]\end{array}}{\mathtt{C} \Vdash \mathtt{Q}\,,\,\ell\,,\,\_ \xrightarrow{A:f(\overline{u})[\overline{v}](\!|\,H\,|\!)} \mathtt{C} \lhd \mathtt{C}' \Vdash \mathtt{Q}\,,\,\ell'\,,\,S \Rightarrow @\mathtt{Q}'}$$

Table 3. The transition relation of *higher-order Stipula* – the rules for functions

As an example of the semantics, consider the `Deposit` contract in Table 2 where `Client` and `Farm` are *Client* and *Farm*, respectively and `cost_flour` is assumed to be 1 (euro per kg). So, let $\ell = [\mathtt{Client} \mapsto \mathit{Client}, \mathtt{Farm} \mapsto \mathit{Farm}, \mathtt{cost\_flour} \mapsto 1, \mathtt{flour} \mapsto 0]$ and let

$$\begin{array}{l} S = \ \mathtt{x}, \mathtt{h} \to \mathtt{Client}\ \ \mathtt{h} \multimap \mathtt{flour} \\ S' = \ \mathtt{x}, \mathtt{w} \to \mathtt{Farm}\ \ (1/\mathtt{cost\_flour}) \times \mathtt{w} \multimap \mathtt{flour}, \mathtt{Client}\ \ \mathtt{w} \multimap \mathtt{Farm} \\ S'' = (1/\mathtt{cost\_flour}) \times \mathtt{w} \multimap \mathtt{flour}, \mathtt{Client}\ \ \mathtt{w} \multimap \mathtt{Farm} \\ S_P = \mathtt{x} \to \tilde{}\ \ \mathtt{flour} \multimap \mathtt{Farm} \end{array}$$

We have the following transitions (in the rightmost column we write the rule that has been used); memories ℓ_1, ℓ_2, ℓ_3 are defined afterwards:

$$\begin{array}{ll}
\mathtt{Deposit} \Vdash \mathtt{Standard}\,,\,\ell\,,\,_ & \\
\quad \xrightarrow{\mathit{Farm}:\mathtt{send("deposit\ kg")[10]}} & \mathtt{Deposit} \Vdash \mathtt{Standard}\,,\,\ell_1\,,\,\mathtt{S} \Rightarrow @\mathtt{Standard} \quad [\textsc{Function}] \\
\quad \xrightarrow{\mathtt{"deposit\ kg"},10 \to \mathit{Client}} & \mathtt{Deposit} \Vdash \mathtt{Standard}\,,\,\ell_1\,,\,\mathtt{h} \multimap \mathtt{flour} \Rightarrow @\mathtt{Standard} \quad [\textsc{Value-Send}] \\
\quad \longrightarrow & \mathtt{Deposit} \Vdash \mathtt{Standard}\,,\,\ell_2\,,\,_ \Rightarrow @\mathtt{Standard} \quad [\textsc{Asset-Update}] \\
\quad \longrightarrow & \mathtt{Deposit} \Vdash \mathtt{Standard}\,,\,\ell_2\,,\,_ \quad [\textsc{State-Change}] \\
\quad \xrightarrow{\mathit{Client}:\mathtt{buy("paid\ euro")[8]}} & \mathtt{Deposit} \Vdash \mathtt{Standard}\,,\,\ell_3\,,\,_ \quad [\textsc{Function}] \\
\quad \xrightarrow{\mathtt{"paid\ \ euro"},8 \to \mathit{Farm}} & \mathtt{Deposit} \Vdash \mathtt{Standard}\,,\,\ell_3\,,\,\mathtt{S}'' \Rightarrow @\mathtt{Standard} \quad [\textsc{Value-Send}] \\
\quad \xrightarrow{8 \multimap \mathit{Client}} & \mathtt{Deposit} \Vdash \mathtt{Standard}\,,\,\ell_4\,,\,\mathtt{w} \multimap \mathtt{Farm} \Rightarrow @\mathtt{Standard} \quad [\textsc{Asset-Send}]
\end{array}$$

$$
\begin{array}{rl}
\xrightarrow{8 \multimap \circ Farm} & \texttt{Deposit} \Vdash \texttt{Standard}\,,\ \ell_5\,,\ \_ \Rightarrow @\texttt{Standard} \quad [\textsc{Asset-Send}] \\
\longrightarrow & \texttt{Deposit} \Vdash \texttt{Standard}\,,\ \ell_5\,,\ \_ \quad [\textsc{State-Change}] \\
\xrightarrow{Farm:\texttt{hardship("Pay in advance")}(\!|\, H \,|\!)} & \texttt{Deposit} \lhd \texttt{C}_H \Vdash \texttt{Standard}\,,\ \ell_6\,,\ \texttt{S}_\texttt{P} \Rightarrow @\texttt{Excp} \quad [\text{HO-}\textsc{Function}]
\end{array}
$$

where $\texttt{C}_H$ is the code of Section 3 and

$$
\begin{array}{ll}
\ell_1 = \ell[\texttt{x} \mapsto \texttt{"deposit kg"}, \texttt{h} \mapsto 10] & \ell_2 = \ell_1[\texttt{h} \mapsto 0, \texttt{flour} \mapsto 10] \\
\ell_3 = \ell_2[\texttt{x} \mapsto \texttt{"paid euro"}, \texttt{w} \mapsto 8] & \ell_4 = \ell_3[\texttt{flour} \mapsto 2] \\
\ell_5 = \ell_4[\texttt{flour} \mapsto 2, \texttt{w} \mapsto 0] & \ell_6 = \ell_5[\texttt{x} \mapsto \texttt{"Pay in advance"}]
\end{array}
$$

5 Constraining amendments

Up-to now *higher-order Stipula* enables parties to make any kind of amendment, which is considered too liberal in the present contractual domains. 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 these in a specific clause.

In order to implement such possibility, a first solution we discuss to constrain amendments relies on static-time restrictions. 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 H and verifies whether they comply or not with the restrictions in the syntactic clause. In this context, the rule [HO-Function] becomes (we rewrite the premises of [HO-Function]):

$$
\frac{\begin{array}{c}
[\text{HO-}\textsc{Function-SC}] \\
\texttt{C}[@\texttt{Q}\ \texttt{A} : f]_{\ell,\overline{u},\overline{v}} = @\texttt{Q}\ \texttt{A}\colon f(\overline{y})[\overline{k}](\!|\, X \,|\!) \\
\ell(\texttt{A}) = A \qquad H = \texttt{C}'\ \ \{\, S \,\} \Rightarrow @\texttt{Q}' \\
\texttt{C}' = \mathit{parties}\ \overline{\texttt{B}} = \overline{B}\ \ \mathit{fields}\ \overline{x}\ \ \mathit{assets}\ \overline{h}\ \ F \\
\ell' = \ell[\overline{y} \mapsto \overline{u}, \overline{k} \mapsto \overline{v}, \overline{h} \mapsto \overline{0}, \overline{\texttt{B}} \mapsto \overline{B}] \\
\mathbb{T}(H)
\end{array}}{\texttt{C} \Vdash \texttt{Q}\,,\ \ell\,,\ \_ \xrightarrow{A:f(\overline{u})[\overline{v}](\!|\, H \,|\!)} \texttt{C} \lhd \texttt{C}' \Vdash \texttt{Q}\,,\ \ell'\,,\ S \Rightarrow @\texttt{Q}'}
$$

that enables the transition if $\mathbb{T}(H)$ is true.

The syntactic clause defining the predicate $\mathbb{T}(\cdot)$ is the following

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

$$
\begin{array}{l}
T ::= \texttt{constraints [parties: fixed; fields: } \overline{\texttt{f}} \texttt{ constant;} \\
\qquad\qquad \texttt{assets: } \overline{\texttt{h}} \texttt{ not-decrease; reachable states: } \overline{\texttt{Q}} \texttt{]}
\end{array}
$$

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 Flour Delivery code of Table 2 then the amendment $\mathtt{C}_{H'}$ of Section 3 would have been rejected. The constraint "`fields:` $\overline{\mathtt{f}}$ `constant`" disable updates of fields in $\overline{\mathtt{f}}$. 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:` $\overline{\mathtt{h}}$ `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 to drain `flour` (on the contrary, addition of flour is always admitted). Finally, the constraint "`reachable states:` $\overline{\mathtt{Q}}$" guarantees that, whatever contract update is performed, the states in $\overline{\mathtt{Q}}$ can be reached from the ending state of the amendment. This because, for example, the corresponding functionalities cannot be disallowed forever.

In the following we discuss the implementation of the predicate $\mathbb{T}(H)$, given a constraint clause in the code.

Fixed parties. This constraint is easy to implement: it is sufficient to verify that, for every amendment H, the term `parties:` $\overline{\mathtt{A}}$ does not belong to H (or $\overline{\mathtt{A}}$ is empty).

Constant fields and not-decreasing assets. The technique for assessing the constraints about fields and assets amounts to parse the H and spot the problematic instructions. In particular, if `fields:` $\overline{\mathtt{f}}$ `constant` and $\mathtt{g} \in \overline{\mathtt{f}}$ then H must not contain the instruction $E \to \mathtt{g}$. Similarly, if `assets:` $\overline{\mathtt{h}}$ `not-decrease` and $\mathtt{k} \in \overline{\mathtt{h}}$ then H must not contain the instructions $E \times \mathtt{k} \multimap \mathtt{h}'$ and $E \times \mathtt{k} \multimap A$. The predicate $\mathbb{T}(\cdot)$ uses the judgments $\overline{\mathtt{f}};\overline{\mathtt{h}} \vdash G$, where G ranges over H, F, and S, which are formally defined by a type system. The key rules of the type system are

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

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

$$\text{[T-Function]}\ \frac{\overline{\mathtt{f}};\overline{\mathtt{h}} \vdash S}{\overline{\mathtt{f}};\overline{\mathtt{h}} \vdash @\mathtt{Q}\ \mathtt{A} :\ \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}]\,(E)\{\,S\,\} \Rrightarrow @\mathtt{Q}'} \qquad \text{[T-Amendment]}\ \frac{H = D\ F \quad \{\,S\,\} \Rrightarrow @\mathtt{Q} \quad \overline{\mathtt{f}};\overline{\mathtt{h}} \vdash F \quad \overline{\mathtt{f}};\overline{\mathtt{h}} \vdash S}{\overline{\mathtt{f}};\overline{\mathtt{h}} \vdash H}$$

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], an amendment $H = D\ F\ \ S\ \ @\texttt{Q}$ is correct provided that every function in F 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 with a static analysis because the values of guards of functions may depend on memories 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.

To define the notion of reachable state of an amendment, we use a predicate `is_in` defined as follows: $@\texttt{Q}\ \texttt{A} : \texttt{f}\ @\texttt{Q}''\ \texttt{is\_in}\ \texttt{C}$ holds true if

- $\texttt{C}$ is a contract without amendments and there is $@\texttt{Q}\ \texttt{A} : \texttt{f}(\overline{\texttt{y}})[\overline{\texttt{k}}]\,(\texttt{E})\{\,\texttt{S}\,\} \Rightarrow @\texttt{Q}'$ in $\texttt{C}$;
- or $\texttt{C} = \texttt{C}' \lhd \texttt{C}''$ and either $@\texttt{Q}\ \texttt{A} : \texttt{f}(\overline{\texttt{y}})[\overline{\texttt{k}}]\,(\texttt{E})\{\,\texttt{S}\,\} \Rightarrow @\texttt{Q}'$ in $\texttt{C}'$ or $@\texttt{Q}\ \texttt{A} : \texttt{f}(\overline{\texttt{y}})[\overline{\texttt{k}}]\,(\texttt{E})\{\,\texttt{S}\,\} \Rightarrow @\texttt{Q}'$ in $\texttt{C}''$.

The predicate $@\texttt{Q}\ \texttt{A} : \texttt{f}\ @\texttt{Q}''\ \texttt{is\_in}\ \texttt{C}$ is false otherwise.

The set of reachable states of a contract $\texttt{C}$ (with amendments) from $\texttt{Q}$, noted $\mathbb{Q}_{\texttt{Q}}$, is the least set such that

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

We notice that $\mathbb{Q}_{\texttt{Q}}$ is always finite and can be easily computed by a standard fixpoint technique. For example, in the first amendment H of Section 3, $\texttt{Standard} \notin \mathbb{Q}_{\texttt{Excp}}$, while $\texttt{Standard} \in \mathbb{Q}_{\texttt{Standard}}$ in $\texttt{Deposit} \lhd \texttt{C}_H \lhd \texttt{C}_{H'}$, where $\texttt{C}_{H'}$ is the second amendment of Section 3. It turns out that $\texttt{Excp} \notin \mathbb{Q}_{\texttt{Standard}}$ in $\texttt{C} \lhd \texttt{C}_H \lhd \texttt{C}_{H'}$.

The predicate $\mathbb{T}(\cdot)$ verifies that `reachable states:` $\overline{\texttt{Q}}$ in the contract $\texttt{C}$ holds for an amendment $H = \texttt{C}'\ \ S\ \ @\texttt{Q}'$ by computing $\mathbb{Q}_{\texttt{Q}'}$ and verifying $\overline{\texttt{Q}} \subseteq \mathbb{Q}_{\texttt{Q}'}$.

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 [6]; 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 $parties(\mathtt{C})$ be defined as follows:

- if $\mathtt{C}$ is a contract without amendments then $parties(\mathtt{C})$ is the set of parties in $\mathtt{C}$;
- if $\mathtt{C} = \mathtt{C}' \lhd \mathtt{C}''$ then $parties(\mathtt{C}) = parties(\mathtt{C}') \cup parties(\mathtt{C}'')$, where $parties(\mathtt{C}'')$ is the set of parties in $\mathtt{C}''$ (that, by definition, is disjoint from $parties(\mathtt{C}')$).

Let also A ACCEPTS H IN ℓ be a predicate that takes an amendment H and verifies whether it complies or not with A's policy in the memory ℓ. It is worth to notice that the predicate also depends on the memory; therefore the policy of A might change in accordance with the updates of the memory. In particular, if ℓ stores a timestamp (the semantics of full *Stipula* has a global clock by which the events are modelled), the predicate ACCEPT may depend on time. In this context, the rule [HO-FUNCTION] becomes (we also rewrite the premises):

$$\text{[HO-Function-AGREE]}\quad \frac{\begin{array}{c}\mathtt{C}[\mathtt{@Q\,A} : f]_{\ell,\overline{u},\overline{v}} = \mathtt{@Q\,A}\colon f(\overline{y})[\overline{k}](\!|\, X \,|\!) \\ \ell(\mathtt{A}) = A \qquad H = \mathtt{C}'\ \{\, S \,\} \Rrightarrow @\mathtt{Q}' \\ \mathtt{C}' = \mathit{parties}\ \overline{\mathtt{B}} = \overline{B}\ \ \mathit{fields}\ \overline{x}\ \ \mathit{assets}\ \overline{h}\ \ F \\ \ell' = \ell[\overline{y} \mapsto \overline{u}, \overline{k} \mapsto \overline{v}, \overline{h} \mapsto \overline{0}, \overline{\mathtt{B}} \mapsto \overline{B}] \\ \big(\ell(\mathtt{A}') \text{ ACCEPTS } H \text{ IN } \ell'\big)^{\mathtt{A}' \in parties(\mathtt{C}) \setminus \mathtt{A}}\end{array}}{\mathtt{C} \Vdash \mathtt{Q}\,,\, \ell\,,\, \_ \xrightarrow{A:f(\overline{u})[\overline{v}](\!| H |\!)} \mathtt{C} \lhd P \Vdash \mathtt{Q}\,,\, \ell'\,,\, S \Rrightarrow @\mathtt{Q}'}$$

We notice that, according to [HO-FUNCTION-AGREE], the acceptance of H is restricted to parties in the contract $\mathtt{C}$: the new parties in H have nothing to accept.

In the context of the example in Table 2, as discussed in Section 3, `Farm` invokes

$$\mathtt{Farm} : \mathtt{hardship("Pay\_in\_Advance")[]}(\!|\, H \,|\!)$$

proposing the amendment H. At this point, for the new code becoming operational and enter into force, `Client` must satisfies the predicate

$$\ell(\mathtt{Client}) \text{ ACCEPTS } P \text{ IN } \ell'$$

assuming that ℓ and ℓ' are the memories before and after the transition, respectively.

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.* [11, 9, 8, 12]. 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 an 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` [15] 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 [20] and Lexon [14]. 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 [10]. 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 logic 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 [17]) and a protocol may dispatch an invocation to one version or another, we get a smart contract concept similar to our operation $\mathtt{C} \lhd P_1 \lhd \cdots \lhd P_n$.

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* is at a higher level of abstraction than addresses or proxies and allows reasoning about amendments smoothly with respect to 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 amend-

ing it. Meanwhile, the above patterns seem useful as lower-level techinques for implementing the language on some (blockchain or distributed or centralized) architecture – see also Section 8.

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 higher-order functionality 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 will be available on-line in the next days at [7]). The higher-order extension admits functions that input codes; these codes are compiled on-the-fly and added to the contract (the compilation also include a type inference analysis, see [6]). In correspondence of every invocation, a dispatch 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 the (basic, first order) prototype, taking inspiration from visual interfaces as in [19], is integrated with a user-friendly and easy-to-use programming interface. We also plan to extend the interface with higher-order features: this will allow us to collect comments and reports of the proposal by non-expert users.

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, 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.
6. Silvia Crafa, Cosimo Laneve, Giovanni Sartor, and Adele Veschetti. Pacta sunt servanda: legal contracts in Stipula. *Science of Computer Programming*, 225, January 2023.
7. Silvia Crafa, Cosimo Laneve, and Adele Veschetti. The Stipula Prototype, July 2022. Available on github: `https://github.com/stipula-language`.
8. 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.
9. 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.
10. Ethereum Foundation. Upgrading smart contracts. `https://ethereum.org/en/developers/docs/smart-contracts/upgrading`, 2023.
11. 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.
12. 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.
13. Filip De Ly and Marcel Fontaine. *Drafting International Contracts*. Brill, 2006.
14. Lexon Foundation. Lexon Home Page. `http://www.lexon.tech`, 2019.
15. Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: A programming language for the law. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.
16. Eliza Mik. Smart contracts terminology, technical limitations and real world complexity. *Law, Innovation and Technology*, 9:269–300, 2017.
17. Nick Mudge. How diamond upgrades work. `https://dev.to/mudgen/how-diamond-upgrades-work-417j`, 2022.
18. 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.
19. 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.
20. Aaron Wright, David Roon, and ConsenSys AG. OpenLaw Web Site. `https://www.openlaw.io`, 2019.