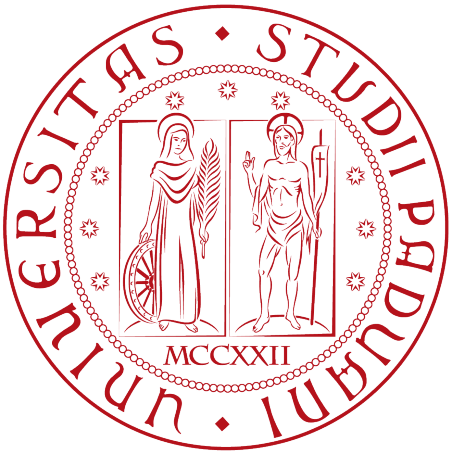
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

MSc course in Computer Science



Advanced typing for asset-aware programming languages

Master Thesis

Supervisor

Prof. Silvia Crafa

Student

Benedetto Cosentino

Academic Year 2022-2023

Advanced typing for asset-aware programming languages
Master Thesis

Abstract

Error detection in programming languages has always been important in reducing bugs and vulnerabilities in programs and tools that help programmers in this task are crucial for the industry. Several kinds of error may occur in programs that can cause unexpected behaviours and lead to crashes. Fortunately, some of them can be detected using *type systems* and *compilers*. However, modern *compilers* may not be enough anymore: in fact, the range of mistakes that programmers may introduce is widely spread with the increasing complexity of today's systems, such as *blockchains*. Languages for these environments should consider new classes of errors related to *assets*: in fact, in this context arbitrary duplication, creation or loss of assets should be avoided. These new kinds of error are strictly related to the concept of the *state* of objects, which are in *blockchains smart contract* instances.

In this thesis, we discuss two *typestated-oriented* programming languages designed for *smart contracts* development, *Stipula* and *Obsidian*, and compare their expressiveness and main properties. In this analysis, we notice that *Stipula* adopts a safer and more flexible approach than *Obsidian* in legal contract writing, due to the primitives available for programmers. These features enable a simpler and more readable implementation of contracts and enforce a safer approach to development. On the other hand, *Stipula* lacks typical functionalities, such as user-defined data and data structures, that other *object-oriented* programming languages have: in particular, it does not support a full-fledged *type system* that ensures safety properties on *asset* operation.

Then, an *Obsidian* implementation for *currencies* and *tokens* used in *Stipula* is provided and discussed in order to find hints to add in this new language *ownership*, which would help in error detection for *asset* references. Proofs that they are well-typed are provided: the tools used in these demonstrations give us some hints, especially on how *ownership* should work in *Stipula* statements.

This thesis focused on the search for features and practises to improve the expressiveness of *typestate-oriented programming* languages. *Obsidian* is a *statically-typed* and *typestate-oriented* with careful attention to safety, security and usability. These features fall within the main goals of Stipula, a newborn *typestate-oriented* language for legal contracts. The insertion of such features in *Stipula* may be a good way to improve its safety and users' experience.

Sommario

La rilevazione di errori nei linguaggi di programmazione è sempre stata importante nel ridurre la presenza di bug e vulnerabilità del software e lo sviluppo di strumenti che aiutino i programmatori in tal senso sono cruciali per le aziende del settore. Nei programmi è possibile riscontrare diverse tipologie di errori che possono causare comportamenti inaspettati da parte del software o addirittura al *crash*. Fortunatamente, alcuni di questi errori possono essere individuati usando *sistemi di tipi* e *compilatori*. Tuttavia, i moderni *compilatori* potrebbero non essere più abbastanza: infatti, lo spettro di possibili errori introdotti dai programmatori si è molto ampliato insieme alla complessità dei sistemi informatici odierni, come ad esempio le *blockchain*. I linguaggi per questi ambienti dovrebbero considerare una nuova categoria di errori correlata agli *asset*: infatti, in questi casi la loro duplicazione, creazione o perdita arbitraria dovrebbe essere evitata. Queste tipologie di errori sono strettamente correlati al concetto di stato di un oggetto, il quale è un'istanza di uno *smart contract*.

In questa tesi, vengono discussi due linguaggi di programmazione *typestate-oriented* progettati per lo sviluppo di *smart contracts*, *Stipula* e *Obsidian*, vengono messi a confronto in termini di espressività e proprietà. In questa analisi, si può notare che *Stipula* adotta un approccio più *safe* e flessibile di *Obsidian* nella scrittura di *legal contracts*, grazie alla disponibilità di determinate primitive di linguaggio. Queste funzionalità permettono una realizzazione più semplice e comprensibile dei contratti e costringe il programmatore a un approccio più *safe* nello sviluppo. D'altro canto, *Stipula* non possiede altre funzionalità, come tipi *user-defined* e *strutture dati*, tipiche dei linguaggi di programmazione a oggetti: in particolare, non supporta un *sistema di tipi* completo e che permetta di assicurare proprietà di *safety* per gli *asset*.

Successivamente, viene fornita e discussa una realizzazione in *Obsidian* delle *currencies* e dei *tokens* usati in *Stipula* allo scopo di trovare un modo di aggiungere in questo nuovo linguaggio un sistema di *ownership*, il quale potrebbe aiutare nella rilevazione di errori per riferimenti ad *asset*. Vengono, dunque, fornite le dimostrazioni che gli *statement* presi in considerazione sono *ben tipati*: gli strumenti teorici usati nelle dimostrazioni forniscono effettivamente delle indicazioni su come l'*ownership* dovrebbe funzionare in *Stipula*.

Questa tesi si concentra sulla ricerca di funzionalità e pratiche per migliorare l'espressività dei linguaggi *typestate-oriented*. *Obsidian* è un linguaggio *staticamente tipato* e *typestate-oriented* con una particolare attenzione alla *safety*, alla *security* e all'usabilità. Queste caratteristiche fanno parte anche dei principali obiettivi di *Stipula*, un nuovo linguagggio *typestate-oriented* per i *legal contracts*. L'inserimento di tali funzionalità in *Stipula* potrebbe essere un buon modo per migliorarne le proprietà di *safety* e l'esperienza d'uso del programmatore.

Acknowledgments

I would like to thank prof. Silvia Crafa, my supervisor, for the meticulousness during the writing of this thesis and for the careful attention payed since the first initial phases of this work.
I also thank my family, my relatives and my friend who always supported and appreciated me in these years.
In general, I want to thank everyone dear to me, longstanding or not, that has always shown affection to me in every moment, especially when I needed the most.

Padova, April 2023

Benedetto Cosentino

Riconoscimenti

Ringrazio la prof.ssa Silvia Crafa, relatrice della mia tesi, per la meticolosità con cui mi ha aiutato nella stesura di questa tesi e per l'attenzione con cui mi ha seguito fin dalle fasi iniziali.
Desidero ringraziare tutta la mia famiglia, i miei parenti e i miei amici che mi hanno sempre sostenuto e apprezzato.
In generale, ringrazio tutte le persone a me care, di vecchia data o incontrate da relativamente poco, che non mi hanno mai fatto mancare il loro affetto in ogni momento, soprattutto quando ne avevo più bisogno.

Padova, Aprile 2023

Benedetto Cosentino

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

Error detection in programming languages has always been important in reducing bugs and vulnerabilities in programs and tools that help programmers in this task are crucial for the industry. Their study and design are valuable areas of research.

Several kinds of error may occur in programs: expressions with incorrect types (e.g. $2 + \mathsf{True}$), invocation of inexistent functions, invoking functions with the wrong number of parameters or with the wrong types, division by zero, nonterminating programs, dereferencing invalid pointers and so on. These errors cause unexpected behaviours and usually lead to crashes. However, some of these errors can be detected using type systems and compilers, which are tools that ensure that well-typed programs are free of certain classes of errors. This detection is done statically, that is without even running the programs.

However, modern compilers may not be enough anymore: in fact, the range of mistakes that programmers may introduce widely spreaded with the increase of complexity of today's systems. Nowadays, we live in a world permeated by distributed systems and programs running on networks in which the state of the objects cannot be known statically. For example, this happens in blockchains that are distributed ledgers mostly used to exchange assets such as virtual currency and Non-Fungible Token (NFT)s. For this reason, languages for blockchains should consider new classes of errors related to assets: in fact, in this context arbitrary duplication, creation or loss of assets should be avoided. These new kinds of error are strictly related to the concept of the state of an object, which is in the blockchain environment an instance of a *smart contract*.

1.2 Problem

Several applications nowadays use blockchains, which provide a tamper-resistant mechanism to manage transactions. However, it happened that security vulnerabilities, such as DAO and Parity bugs, were exploited to steal virtual currency worth millions of dollars [21, 24, 1]. Since contracts in blockchains are immutable, bugs and vulnerabilities cannot be fixed easily. For example, due to the DAO hack, Ethereum was forced to hard fork its network.

Ethereum is implemented in Solidity, which is a statically typed language. Its type system grants safety to its programs. However, this is not enough: for example,

the DAO hack could be avoided with a type system using specific asset types and operations. In particular, the usual typing cannot detect several facts, such as state changes, arbitrary duplication, creation and loss of assets. These features can be realised with new approaches to typing, such as Typestate Oriented Programming (TSOP) and ownership systems.

Typestate Oriented Programming (TSOP) is an extension of the Object Oriented Programming (OOP) paradigm where classes are also associated with a protocol. Their definition allows the implementation of type systems that also check the adherence of the objects to such protocols, which are described in terms of states. For example, a file has three different states: *open*, *read* and *closed*. When the file is *open*, the user can check if the end-of-file is reached or close the file. When it is *read*, the user is authorised to keep reading the file, if the end-of-file is not reached, or close it. From this description, it is easy to understand that the behaviour of a file may define a protocol. The aim of TSOP is to check the adherence to this protocol, usually at compile-time.

Ownership is a set of rules that manages how objects in programs are referenced. Ownership systems can be used for different purposes: for example, Rust uses ownership to improve memory management [25]. Usually, ownership systems respect these rules:

- the objects in the ownership systems have an owner;
- there can be only one owner;
- if the owner goes out of scope, the object must be dropped.

Consider the following Rust code:

```
1 let s1 = String::from("hello");
2 let s2 = s1;
3
4 println!("{}, world!", s1); // ERROR!
```

We have two references `s1` and `s2`. After line 1, the reference `s1` is the owner of the `String` object `"hello"`. Then, after the assignment in line 2, `s2` becomes the owner and the first reference is invalidated. Consequently, when `s1` is used in line 4, the compiler will notify an error that warns the programmer that the value of `s1` is moved.

This mechanism can be used in smart contracts to check statically the ownership of assets in order to avoid duplication of assets or multiple uses of the same asset, for example.

1.3 Typestate Programming

As mentioned above, Typestate Oriented Programming (TSOP) is an extension of the OOP paradigm where classes come along with protocols. The aim is to ensure adherence to protocol and safe object behaviour. A typestate, as the word itself suggests, is a pair composed of a type and a state. Several languages use the syntax `T@S` to say that the object is of type `T` at state `S`. In OOP, for every class, we can define functions or methods, but in TSOP it is also required that each method specifies the state needed for the invocation. For example, if a method `m` is defined for type `T` in state `S`, then it can be called only on objects with typestate `T@S`. The interesting part is that, contrarily to type, the state can change during the execution of the program: in fact, TSOP languages usually also provide statements that trigger objects to change state. Then, each method of an object of a certain type defines a starting state and an ending state, establishing a protocol.

Consider the above example of the `File` class. Typically, we define it as follows:

```
class File {
    File(String filename) { ... }
    boolean open() { ... }
    boolean eof() { ... }
    byte read() { ... }
    void close() { ... }
}
```

However, as already said, the `File` objects in order to be used correctly must follow a certain protocol: in particular, any file when initialised must be opened; then, if the opening does not fail, we have to check if the file has already reached the end-of-file or not. In the first case, we can only close the file; otherwise, we can read the first byte or close the file. If we read a byte, the file is in the same situation as before: we have to check if we reached the end-of-file and consequently choose what to do. Otherwise, we can just close the file. Then, we have five possible states:

① an initial state `Init`;

② a state `Open` that notifies that `File` has been opened succesfully;

③ a state `Read` that specifies that the end-of-file has not been reached and that authorises further readings;

④ a state `EOF` that indicates that the end-of-file has been reached;

⑤ the *end*-state.

We can describe the protocol discussed with an automaton (Figure 1.1), where the states of the protocol are the states of the automaton and the methods are the transitions. Sometimes, the transition from one state to another depends also on the result of the method: for example, the `Init` state can reach `Open` if `open` is successful or `end` otherwise. To describe this automaton, we could write a protocol like the following:

```
FileProtocol {
    constructor :  @Init ;

    open  : @Init ==> (true::@Open | false::@End) ;
    eof   : @Open ==> (false::@Read | true::@EOF) ;
    read  : @Read ==> @Open ;
    close : @Open ==> @End
          | @Read ==> @End
          | @Close ==> @End ;
}
```

The protocol can be used from automated tools to determine the typestates of `File` objects at any point in a program or during its execution. We could, for example, detect if the file is closed before a `File` variable goes out of scope by checking if the state `end` is reached (completion of protocol) or if we are trying to read bytes from a closed `File` (protocol adherence).

In general, TSOP adds to OOP more information on *when* methods should be called, using states as abstraction. This choice fits programming languages: in fact, almost any system can be described as a state machine with transitions that change its behaviour. Consequently, having tools that are aware of states and transitions is very useful in order to detect bugs or vulnerabilities caused by the misuse of objects. The main challenge is to develop type systems that embed concepts of automata theory.

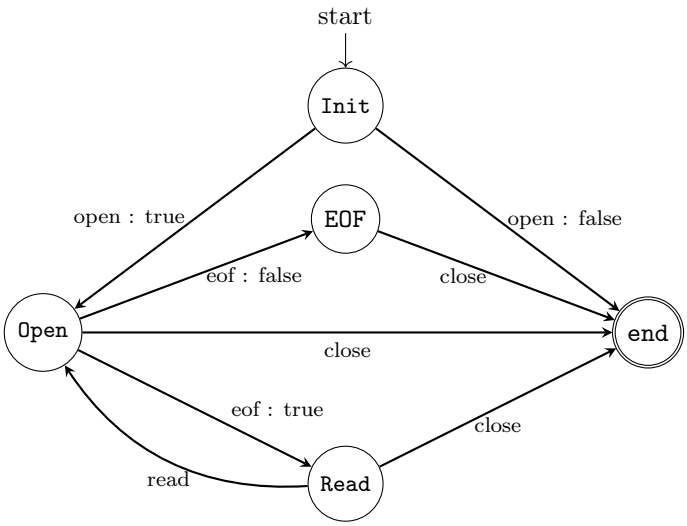


Figure 1.1: Automaton for the protocol of `File`

1.4 Contributions

In this thesis, there are two main contributions:

- we discuss two Typestate Oriented Programming languages, Stipula and Obsidian, designed for smart contracts development and compare their expressiveness and main properties. In this analysis, the advantages of each language are identified: Stipula adopts a safer and more flexible approach due to primitives, such as caller guards or time-triggered events, that model intuitively legal contracts (which is the main purpose of the language). However, we will see that Obsidian has safety guarantees on assets that Stipula lacks.

- the other contribution is a discussion of the typing of asset operations in Obsidian. An Obsidian implementation for currencies and tokens (which are the kind of assets used in Stipula) is provided and discussed in order to find hints to add ownership in Stipula. This may lead to strengthen safety properties in this new language.

1.5 Structure of the document

The second chapter describes Stipula starting from the features that the language offers to the programmers. Then, the type inference system and the properties guaranteed are discussed.

The third chapter describes Obsidian with its features. The presentation also includes a brief description of the blockchain platform that the actual implementation of Obsidian uses (Hyperledger Fabric [12]). Finally, the type system and its core calculus (Silica) are described along with the properties guaranteed by them.

The fourth chapter first introduces a method to translate Stipula contracts into

Obsidian contracts. Then, several examples of this translation are shown and analysed to compare the conciseness of the two languages and other properties.

The fifth chapter shows the typing of the respective Obsidian implementation of the Stipula *send*-statement. For each of them, Silica translation and proofs through derivation trees are provided.

The sixth chapter draws the conclusions of this thesis and reports related works, comparing some of them with Obsidian and Stipula. Hence, some insights on possible future work are provided.

Chapter 2

The Stipula Language

This chapter presents the Stipula language, through an example, that shows the several features provided. Then, the type system and the properties guaranteed by Stipula are discussed.

2.1 Introduction

Digital revolution affected most areas of our reality, one of them is law. However, legal texts are very difficult to translate into a computable representation in a formal language, since they are expressed in a natural language which is very expressive but ambiguous. Due to the *freedom of form* principle, the use of a programming language to draft contracts is a valid option. The benefits are in terms of speed, low ambiguity and automatic and transparent enforcement of contractual clauses.

For this reason, Crafa et al. designed Stipula [9], an *intermediate domain-specific language* for writing legal contracts. It has few selected and intelligible primitives with a precise formalisation and semantics. These primitives preserve the main features of legal contracts:

- the *agreement*: it is the *meeting of the minds* of the parties that must accept the terms of the contract in order to trigger its legal effects;
- the creation of normative positions: *permissions*, *prohibitions*, *obligations* and *powers* of a party are examples of position (that may change as the times goes by);
- the transfer of *assets* and *currency*.

Other desirable features are:

- *Openness* to external conditions, which can trigger contractual effects;
- *Subsequent modification*;
- *Third party adjudication and enforcement*: that is, the possibility of settling disputes over contract terms through some mechanism.

Stipula is designed to possess these features. In particular, the language provides primitives and patterns that match the various parts of legal contracts: *agreement* is directly implemented and corresponds to the constructor of the contract; *permissions*,

powers and *prohibitions* are encoded allowing (or forbidding) parties to invoke certain functions; *obligations* are modelled by scheduling time-triggered events; *assets* transfer and manipulation is managed through specific primitive operations; *openness* to external conditions and *third party adjudication and enforcement* are instead implemented using specific coding patterns. The first one is realised using supplementary parties that have the legal responsibility to fetch data from an external source specified during the agreement. *Third party adjudication and enforcement* are handled including a party that assumes the role of the authority with the respective capabilities.

2.2 The language

In Listing 2.1, we can see a typical Stipula contract: in particular, the contract `BikeRental` regulates a bike rental service. Every contract starts with the keyword `stipula` followed by the name of the contract. Then, the declarations of fields and assets needed during the execution of the contract can be found (lines 2-4). Any term on which the parties agree must appear in the list of fields. At this point, we can recognise the agreement and the list of functions available. To manage changes in normative positions over time, Stipula adopts a *typestate programming* style combined with caller guards. Typestates are preceded by the character `@`. The grammar of the Stipula language can be consulted in Figure 2.1.

2.2.1 Agreement

The `agreement` (lines 6-8) is the constructor of the contract and consists of the list of parties involved, the list of the terms on which they must agree and a body where the agreement is actually executed. In the example, the parties are `Lender`, `Borrower` and `Auhtority` and the terms are `rentingTime` and `cost`. We notice that we can also neglect some field: for example, the field `code` is not involved in the agreement. In fact, this variable will just hold the ID of the bike given to the customer and will not then be part of the terms of the agreement.

In the body, we can find the primitive that actually executes the agreement on `rentingTime` and `cost`. In particular, in the statement on line 7 the parties on the left of the colon must agree on the terms on the right. However, the two lists can also neglect some of the parties: in fact, not all the parties must necessarily agree on all the terms. When executed, the Stipula runtime will ask each party the value of each term involved: for each term, if the values given from each party of the statement are the same, then the agreement succeeds and the contract is constructed; otherwise, an error will be notified to each party.

Finally, the agreement must specify in which state the contract will start: in the example, `BikeRental` starts from the `Inactive` state.

2.2.2 Functions

Every function must declare the state transition and which party can invoke them: for example, the function `offer` (lines 10-13) says that:

- in order to be executed, the contract must be in state `Inactive`;
- after the execution, it will be in state `Payment`;
- it can be called only by `Lender`.

Listing 2.1: BikeRental contract in Stipula

```
stipula BikeRental {
    asset wallet
    field cost, rentingTime, code
    init Inactive

    agreement (Lender,Borrower,Authority)(rentingTime,cost){
        Lender , Borrower: rentingTime , cost
    } ==> @Inactive

    @Inactive Lender : offer(x)[] {
        x -> code;
        _
    } ==> @Payment

    @Payment Borrower : pay()[h] (h == cost) {
        h -o wallet
        code -> Borrower;
        now+rentingTime >>
            @Using {
                "EndReached" -> Borrower
            } ==> @Return
    } ==> @Using

    @Using Borrower : end()[] {
        "EndReached" -> Lender;
        _
    } ==> @Return

    @Return Lender : rentalOk()[] {
        wallet -o Lender;
        _
    } ==> @End

    @Using@Return Lender, Borrower : dispute(x)[] {
        x -> _;
        _
    } ==> @Dispute

    @Dispute Authority : verdict(x,y)[] (y>=0 && y<=1) {
        x -> Lender
        x -> Borrower
        (y*wallet) -o wallet, Lender
        wallet -o Borrower;
        _
    } ==> @End
}
```

A function can be called in different states or from different parties: for example, `dispute` can be called from both `Lender` and `Borrower` and in states `Using` and `Return`. In general, functions have the following form:

$$\texttt{@}\overline{\texttt{S}}\ \overline{\texttt{A}}\ \texttt{: f(}\overline{\texttt{x}}\texttt{)[}\overline{\texttt{k}}\texttt{] (}E\texttt{) \{ ... \} ==> @S'}$$

where $\overline{\texttt{@S}}$ is the list of input states, `@S` is the output state and $\overline{\texttt{A}}$ the list of parties that can invoke the function `f`.

In functions, we have two different lists of parameters: one for the *non-asset* parameters ($\overline{\texttt{x}}$) and one for the *asset* parameters ($\overline{\texttt{k}}$). In particular, this second type of parameter is not simply passed by value, but it is "moved" from the caller: this allows sending currency and assets from one party to another or to the contract itself.

An other feature available in Stipula is the possibility to write a (optional) precondition guard for the functions: its execution is bounded to the condition expressed. For example, if the cost agreed for the rental is 10D ("dollars") and `Borrower` invokes `pay()[8D]`, then the Stipula runtime will respond with an error.

At the end of each function, events can be defined: in particular, events can be triggered only if the function in which they are defined is called.

2.2.3 Statements and Prefixes

Stipula does not provide the same statements as typical programming languages: in fact, the only ones available are conditional statement and *prefixes*, which are used to assign values to or move assets between variables or "send" messages and assets to parties. There are four kinds of *prefix*:

- **assignment:** $E \rightarrow$ `x` - the value of the expression E is assigned to the *non-asset* variable `x`;
- **message sending:** $E \rightarrow$ `A` - the value of the expression E is sent to the party `A`;
- **asset splitting:** $E \multimap$ `h,h'` - the value of the expression E is "moved" from the variable `h` to the variable `h'`;
- **asset sending:** $E \multimap$ `h,A` - the value of the expression E is "moved" from the variable `h` and sent to the party `A`.

Consider the function `verdict` on lines 39-45: its purpose is to emit the verdict of the dispute between `Lender` and `Borrower`. With the first two statements, the reasons for the verdict are sent as a message to the parties involved. Then, the currency contained in `wallet` is split into two parts: since `y` $\in [0, 1]$, then `y*wallet` is necessarily less or equal than the currency in `wallet`. This amount is sent from `wallet` itself to `Lender` as a refund and the remaining part (which is still kept in `wallet`) is returned to `Borrower`.

2.2.4 Events

In Stipula, events can be defined only inside functions: this choice has been made because events are reactions to some party behaviour which emerges from the function called. Consider the event in lines 18-21: the reaction notifies the expiration of the renting time to `Borrower` and changes the state of the contract in order to start the conclusive phases of the contract (the restitution of the bike and the payment completion).

To define an event, we need the time in which it will be triggered, the states that allow its execution, the state reached after the execution and the statements to run. In particular, an event has the following form:

```
time >> @S { statements } S
```

where `time` is the time of trigger activation, `@S` is the list of states that allows the execution of the event and `S` is its output state. Stipula also provides a special keyword `now` that allows us to retrieve the current time of the global system clock.

2.2.5 Expressions

Stipula expressions include values, variables and the keyword `now` combined with the operators (which can be unary or binary). There are different types of values:

- real numbers, usually combined with standard arithmetic operators (`+`,`-`,`*`,`/`);
- boolean values `false` and `true` with boolean operators (`&&`,`||`,`!`);
- strings, that are sequence of characters inside pairs of double quotes (i.e., `"`) possibly concatenated using the operator `^`;
- time values that represent the global system clock, written as `"2023/1/1:00:30"`, eventually combined with `+` and a positive integer number that represents the amount of minutes to add: for example, `now + 2` is an admissible expression and accounts a time 2 minutes after the actual system clock;
- asset values and parties identities, but the use of the relative constants is inhibited since they should be introduced, respectively, only through function parameters and during the agreement.

Asset expressions are combinations of asset values and asset variables with operators `+`, `-` and `*`. However, in this case the semantics is different from the typical meaning of these operators. The semantics changes depending on the kind of asset handled. Consider the *partial function* $[\![\cdot]\!]^{\mathtt{a}}_{\ell} : Expr \hookrightarrow AssetValue$ that, given an expression, returns the respective asset value in the memory ℓ. Then, this function is defined as follows:

- for currencies ($+^{\mathtt{D}}$,$-^{\mathtt{D}}$ are semantic operations that respectively sum and subtract currencies; $*^{\mathtt{D}}$ is the operations that multiplies currencies with a real number):

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

- for tokens:

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

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

Functions	$F ::= \texttt{\_} \mid \texttt{@Q A : f}(\overline{y})[\overline{k}](E)\{S\ W\}\texttt{==> @Q'}\ F$
Prefixes	$P ::= E \to \texttt{x} \mid E \to \texttt{A} \mid E \multimap \texttt{h,h'} \mid E \multimap \texttt{h,A}$
Statements	$S ::= \texttt{\_} \mid P\ S \mid \texttt{if}(E)\{S\}\ \texttt{else}\ \{S\}\ S$
Events	$W ::= \texttt{\_} \mid E \gg \texttt{@Q}\ \{S\}\ \texttt{==> @Q'}\ W$
Expressions	$E ::= v \mid X \mid \texttt{now} \mid E\ \texttt{op}\ E \mid E\ \texttt{uop}\ E$
Values	$v ::= n \mid \texttt{false} \mid \texttt{true} \mid s \mid \Bbbk \mid a$

Figure 2.1: Syntax of Stipula

Any expression can also be combined with relational operators.

2.2.6 Assets

Assets are a particular type of data: the crucial point is that they cannot be created or deleted, but only moved (or *sent*) from a place to another. For this reason, they must be managed differently: we already saw the semantics of the expression. Now, we can focus on the semantics of the *send*-statements for assets: $E \multimap \texttt{h,h'}$ and $E \multimap \texttt{h,A}$. They can be both abbreviated in $\texttt{h} \multimap \texttt{h'}$ and $\texttt{h} \multimap \texttt{A}$ respectively, which are semantically equivalent to $\texttt{h} \multimap \texttt{h,h'}$ and $\texttt{h} \multimap \texttt{h,A}$.

When h is a currency variable, then:

- $E \multimap \texttt{h,h'}$ determines the value a of E, subtracts it from the amount stored in h and adds it to the one stored in h';
- $E \multimap \texttt{h,A}$ determines the value a of E, subtracts it from the amount stored in h and sends it to the party A.

If $a > [\![\texttt{h}]\!]^{\texttt{a}}_{\ell}$, then there is a run-time error.

The token variables must be managed in a different way, since they can be "empty" or "full". In particular, the run-time signals an error:

- if we try to replace a token contained in a "full" variable with another one;
- if we try to send a token from an "empty" variable;
- if we try to update a variable from an "empty" one;
- if we try to clear a "full" variable, sending a 0 value.

Due to these conditions, Stipula avoids losing assets during the execution of *send*-statements and asset expressions. Also, as already mentioned, the only way to introduce assets in a contract is through the function asset parameters. So, the language does not allow asset minting in contracts.

2.3 Type inference system

The syntax of Stipula does not include types in order to make Stipula contracts similar to legal contracts, which do not have (obviously) type annotations, and clearer to unskilled users (like lawyers, for example). However, Stipula has a simple type inference system, which enables derivation of types of assets, fields and function parameters.

The primitive types must obviously match the values mentioned above. In particular, they are defined as follows:

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

The judgements in the Stipula type system are of the following form:

$$\begin{array}{ll} \Gamma, \Delta \vdash E : \alpha, \Upsilon & \text{for expressions} \\ \Gamma, \Delta \vdash S : \Upsilon & \text{for statements} \end{array}$$

where:

- $\alpha, \alpha', ...$ are *type terms* that can be primitive types or *type variables* X, Y, Z;
- Γ is the environment that maps fields and non-asset function parameters to type terms;
- Δ is the environment that maps asset and asset function parameters to type terms;
- Υ is the conjunction of constraints collected during the type inference process.

The inference system works by assigning type variables to the names in the contract and collecting constraints while performing parsing. Then, it finds the correct type values (if exist) through *unification* and substitution techniques.

2.4 Properties

Due to its semantics and type system, Stipula presents several issues, like non-determinism, safety and security. In particular, the management of assets has a central role in all asset-aware languages. The goal should be to avoid the arbitrary creation and loss of assets: these problems may occur in different ways. Stipula guarantees some of the safety and security properties: in particular, the language comes with a tool that verifies *liquidity* of contracts.

2.4.1 Non-Determinism

Stipula is a language designed to run in a distributed context: in fact, the parties are roles for different entities that operate over the same contract. The semantics of the language admits three sources of non-determinism:

① the order of the execution of the ready events;

② the order of the invocations of the permitted functions;

③ the delay of permitted function calls to a later time.

The feature of non-determinism requires a discussion of the topic of equivalence between contracts: along with the language, in the paper is also presented an observational equivalence called *Normative Equivalence*. This relation allows to distinguish contracts checking their behaviour: that is, comparing agreements, permission, prohibitions, obligations and assets received from the contract. Using a formal equivalence is one of the greatest advantages in adopting a programming language to write a contract.

2.4.2 Safety

Stipula manifests mainly six kind of errors: unsafe operations, access to not initialised fields, references to unknown identifiers, drainage of too much value from an asset, forging new assets and accumulation tokens. However, the type system developed addresses some of these problems. In particular, the Safety Theorem proven in the article shows that, if a contract is well-typed, then the only errors that may cause runtime errors are unsafe asset operations, accesses to uninitialised fields, and division by 0. Examples of unsafe asset operations are `123D + 1T` or `123D - 1234D`: in the first case, the problem is the use of different types of assets; in the second case, it is a negative value as a result of an asset expression. The first kind could be faced with the introduction of different types for currencies and tokens. The second is harder to manage statically, especially when variables are involved.

2.4.3 Liquidity

An important security property of smart contracts is *liquidity*, which requires that the contract assets are always transferred to some party. If a contract is non-liquid, then assets can be frozen inside the contract forever and, then, destroyed causing loss of assets.

Silvia Crafa and Cosimo Laneve designed a liquidity analyser for Stipula that combines automata theory, approximation and fixpoint techniques. This tool prevents non-liquidity in Stipula contracts [6].

Chapter 3

The Obsidian language

This chapter introduces the Obsidian language and shows several features of the given Obsidian language. Then, Obsidian type system and guaranteed properties are discussed.

3.1 Introduction

Blockchains nowadays are used for many applications and provide a tamper-resistant mechanism for transactions. However, there were applications that included security vulnerabilities, such as DAO and Parity bugs, that were exploited to steal virtual currency worth millions of dollars [21, 24, 1]. Contracts in blockchains are required to be immutable, so errors cannot be fixed easily.

Obsidian is a programming language that aims to prevent these bugs as soon as possible through strong compile-time features. In particular, Obsidian uses a combination of *typestates* and *linear types* [27]: the goal is to ensure the manipulation of objects according to their states and the safe manipulation of assets to avoid their loss. The guidelines followed for the design of Obsidian are:

- Strong static safety: due to the severity of bugs in smart contracts, the language uses a strong static type system to enforce compile-time error detection with an eye towards *loss of assets*;
- User-centered design: one of the main goals was also to propose a language as usable as possible;
- Blockchain-agnosticism: since the research on blockchain is very lively in these years, a good language should not depend on a single platform, but it should aim to possess properties commonly owned to many platforms.

In order to facilitate usability, in Obsidian TSOP is integrated in an Object Oriented (OO) setting. These two paradigms can be easily merged: in fact, OOP manages data with states that can mutate over time and TSOP allows to specify protocols and to check statically the adherence to such protocols. Then, smart contracts can be seen as objects with a specific protocol.

Smart contracts also manipulate assets. Obsidian uses *linear types* and *ownership* and integrates this approach with typestates. In particular, the language forbids the loss of assets owned by some other entity.

Mode	Other possible aliases	Typestate Mutation
Owned	Unowned	Allowed
Unowned	Unowned, either Owned or Shared	Forbidden
Shared	Unowned, Shared	Allowed
State	Unowned	Allowed

Table 3.1: Asset references

Obsidian is presented together with a *core calculus*, Silica, and an integrated architecture that supports smart contracts and client programs [4]. In particular, the first one is used to prove the key properties of Obsidian, such as *Asset Retention*.

3.2 The language

In Listing 3.1, we can see an example of Obsidian contract [2]. In particular, this simple program models the behaviour of a vending machine that sells candies in return for coins. For this reason, the machine has a coin bin and can be in two different states: `Full` when it has the candy, `Empty` otherwise.

Every contract is declared with the keyword `contract`. Every file written in Obsidian must provide exactly one `main contract`. Then, inside the contract we can define the list of fields, states, constructors and functions available. In Obsidian, functions are called *transactions*.

3.2.1 Assets

In Obsidian, assets are particular types of contracts whose instances are guaranteed to have exactly one owner. Every asset can be referenced with three different modes:

- Owned: the only reference to the object that is allowed to mutate its type state; in this case, there may be many Unowned aliases but no Owned or Shared ones;
- Unowned: a *non-linear* reference to the object; in this case, there may be (or not) an Owned one or many other Shared aliases;
- Shared: one of the references that can mutate the typestate of the object; there are no Owned aliases and there may be many others Shared or Unowned references.

A contract owns an asset when it is referenced with an Owned reference. In this case, the contract must also be declared as an asset. For example, in Listing 3.1 the main contract `TinyVendingMachine` owns the assets `coinBin` and `inventory` (lines 17 and 20). Then, it must also be declared as an asset.

If a contract defines states, then its instances can be referenced using the state: in this case the reference is an Owned reference that also communicates that the object is in a specific state. For example, we could refer to a `TinyVendingMachine` in other contracts as `TinyVendingMachine@Empty` or `TinyVendingMachine@Full`.

Modes and states are separated from the type by the character `@`. For example, an asset of type `Coin` with mode Owned is declared as `Coin@Owned`.

In Obsidian, there are three ways for a reference to lose ownership:

- **assignment:** if an Owned reference is assigned to an Unowned reference `x` of the same contract, it will become Unowned and `x` will obtain the ownership of the object referenced (Listing 3.2).

Listing 3.1: `TinyVendingMachine` contract

```
asset contract Candy {
}

asset contract Coin {
}

asset contract Coins {
    // Currently implemented as a bottomless void for convenience.

    transaction deposit(Coins @ Unowned this, Coin @ Owned >> Unowned c) {
        disown c;
    }
}

// This vending machine sells candy in exchange for candy tokens.
main asset contract TinyVendingMachine {
    Coins @ Owned coinBin;

    state Full {
        Candy @ Owned inventory;
    }
    state Empty; // No candy if the machine is empty.

    TinyVendingMachine@Owned () {
        coinBin = new Coins(); // Start with an empty coin bin.
        ->Empty;
    }

    transaction restock(TinyVendingMachine @ Empty >> Full this,
                        Candy @ Owned >> Unowned c) {
        ->Full(inventory = c);
    }

    transaction buy(TinyVendingMachine @ Full >> Empty this,
                    Coin @ Owned >> Unowned c) returns Candy @ Owned {
        coinBin.deposit(c);
        Candy result = inventory;
        ->Empty;
        return result;
    }

    transaction withdrawCoins() returns Coins @ Owned {
        Coins result = coinBin;
        coinBin = new Coins();
        return result;
    }
}
```

Listing 3.2: Obsidian assignment

```
// Suppose that x and y have types Coin@Unowned
// and Coin@Owned respectively
x = y
// At this point, x and y have types Coin@Owned
// and Coin@Unowned respectively
```

- **reference passed as parameter:** when a reference of contract C is passed as an argument to a transaction and the parameter is declared as C`@Owned » Unowned`, then after the execution of the transaction the reference will have type C`@Unowned`. In general, in Obsidian the type $C@T_{ST} \gg T'_{ST}$ indicates a reference of contract C that will change its mode (or state) from T_{ST} to T'_{ST}.

- `disown` **statement:** the last way for a reference `x` to lose ownership is the statement `disown x`: in this case, the programmer explicitly requires the contract to give up the ownership.

3.2.2 Type Declarations and Static Assertions

In Obsidian, local variables, fields and transaction parameters are required to be declared with their types. In particular, the last two must also be declared with their mode. But, during the execution of some transaction, the actual modes may not be consistent with those declared. This fact may reduce the readability of the code: for this reason, Obsidian also includes *static assertions* (`[e @ mode]`). These are statements that do not have any effect on the dynamic semantics, but are used as checked documentation: that is, the type checker verifies if the given mode is valid for a certain expression. For example, the statement `[coin @ Owned]` is used to check statically if the reference `coin` is owned. For example, consider the transaction `withdrawCoins()` in Listing 3.1. After line 43, `coinBin` will have typestate `Coins@Unowned` and, after line 44, will be again `Coins@Owned`. We could statically check if `coinBin` is still Owned after the first statement:

```
transaction withdrawCoins () returns Coins @ Owned {
    Coins result = coinBin;
    [coinBin@Owned];       // Compile-time error!
    coinBin = new Coins();
    return result;
}
```

In this case, the Obsidian compiler will notify us of an error on the line of the assertion warning us that `coinBin` is not Owned.

3.2.3 State transitions

When defined a state can include a list of fields which are in scope only when the object is in the that state. This feature raises the issue of initialisation and disposal of fields when transitioning from one state to another. Considering a state S:

- when the object is in state S, the fields for S must be initialised;

- when the object is not in state S, the fields for that state cannot be in scope.

To maximise flexibility, Obsidian provides two ways to initialise fields before transitioning from a state to the target state. The first one allows us to write initialisations in the transition:

```
->S(x = a);
```

where the field `x` is initialised to `a`. The second way allows for initialisations before the state transition:

Listing 3.3: Example of private transaction

```
1 contract C {
2     A@S1 a;
3     private (A@S2 >> S1 a) transaction f() {...}
4 }
```

```
S::x = a;
->S;
```

When transitioning to a state, the fields that will not be in scope in the target state must not be owned references to assets. This means that the ownership of these fields must be transferred or disowned prior to the transition.

Consider the `TinyVendingMachine`: in the state `Full` the field `inventory` is declared. When the machine is restocked, Obsidian requires to initialise `inventory` and this is done in line 31. On the other hand, when someone buys the candy from the machine, `inventory` gives up ownership to `result` on line 37 and then the contract can pass to the state `Empty` (line 38).

3.2.4 Transactions

Obsidian transactions can be functions that return values of type T_{out} or procedures, and they have the following form:

```
transaction f(C_f@S_f x_f) returns T_out {...} // function
transaction g(C_g@S_g x_g) { ... }  // procedure
```

To define the scope of transactions or if they determine the transaction to another state, Obsidian requires adding the reference to `this` in the list of parameters ($\overline{\texttt{C}_f\texttt{@S}_f\ \texttt{x}_f}$, $\overline{\texttt{C}_g\texttt{@S}_g\ \texttt{x}_g}$). In particular, the parameter `C@S this` will appear in the list indicating that the transaction can be called only for objects in state `S`. The transition between states is written using the symbol `>>`. For example, the transaction `restock` in Listing 3.1 can be called if the vending machine is `Empty` and at the end of the transaction it will be in state `Full`.

As already mentioned, fields during the execution of transactions may not be consistent with their declarations. Obviously, the only discrepancy allowed involves mode. However, they are required to be coherent to declarations at the end of the transactions. In any case, if the actual types of fields of `this` do not match the ones declared, any transaction invocation is forbidden. The only exception to this rule is represented by private transactions.

Private transactions Obsidian allows the definition of private transactions to simplify refactoring and encourage modular and readable code scripting. To declare private transactions, the keyword `private` is added before the declaration.

As mentioned, before invocation of a private transaction, the fields of the object could not be consistent with those declared. For this reason, private transactions may require a declaration of expected types before and after their invocations (Listing 3.3). In this way, programmers can do refactoring and have the benefits of statically typed transactions.

Listing 3.4: Parametric polymorphic list in Obsidian

```
contract List[T@S] {
    state Empty;
    state HasNext {
        List[T@S]@Owned tail;
        T@S value;
    }
    ...
}
```

3.2.5 Dynamic State checks

Checking states statically may be impossible in some circumstances. For this reason, Obsidian allows programmers to write dynamic state checks. The syntax of these statements is the following:

```
if (x in S) {
    ...
}
```

In order to fit this checks with typestates, Obsidian compiler makes some assumptions on the typestate of the tested variable depending on the mode of the tested variable. For example, if the statements checks `x in Owned`, then in the body of *if*-statement the compiler will assume that `x` is Owned. These assumptions are made to guarantee the most precise and safe type-checking.

3.2.6 Parametric Polymorphism

In Obsidian, class-level inheritance is forbidden. Contracts cannot extend other contracts and can only implement interfaces. This choice was made due to the *fragile base class problem*. However, Obsidian supports parametric polymorphism to guarantee safety for collections and avoid code duplication. Contracts can have two type parameters: one for the contract and one for the mode (Listing 3.4).

3.3 System design and implementation

Obsidian is currently implemented with the authorised blockchain platform Hyperledger Fabric [12] that enables organisations to choose who can access the ledger and which peers should approve each transaction. Smart contracts in Java are written in Java: for this reason, the Obsidian compiler translates the sources into Java code ready to be deployed on Fabric peer nodes.

3.3.1 The ledger

In the ledger the state of smart contracts is stored as key/value pairs. To do so, serialisation must be provided: Obsidian generates automatically serialisation code using *protocol buffers* [17]. To execute a transaction, objects are lazily loaded from the storage, used and eventually modified. After execution, the modified objects are serialised and stored again in the ledger.

Listing 3.5: Client program for `TinyVendingMachine`

```
import "TinyVendingMachine.obs"

main contract TinyVendingMachineClient {
    transaction main(remote TinyVendingMachine@Shared machine) {
        restock(machine);

        if (machine in Full) {
            Coin c = new Coin();
            remote Candy candy = machine.buy(c);
            eat(candy);
        }
    }

    private transaction restock(remote TinyVendingMachine@Shared machine) {
        if (machine in Empty) {
            Candy candy = new Candy();
            machine.restock(candy);
        }
    }

    private transaction eat(remote Candy @ Owned >> Unowned c) {
        disown c;
    }
}
```

3.3.2 Client programs

Usually, blockchain systems use a language for smart contracts and another one for client programs: for example, Solidity is coupled with JavaScript. The interface for a contract is specified in Application Binary Interface (ABI) and, if some incompatibility between JavaScript serialisation code and Solidity deserialisation occurs, then there may be bugs.

For this reason, Obsidian allows users to also write client programs. In this way, client programs can reference the same contract implementations as the ones in the server. This leads to the possibility for the server and clients to use the same serialisation and deserialisation that are also automatically generated in Obsidian.

Client programs in Obsidian have a `main` transaction that has as input a `remote` reference. This keyword specifies that the object referenced is instantiated in the blockchain and indicates to the compiler to implement the relative references with stubs using an RMI-like mechanism. In particular, the `main` transaction input includes the `remote` reference to the smart contract instance. These remote references are transmitted between clients and blockchains using objects' unique IDs. In Listing 3.5, for example the parameter of main contract references an instance of the smart contract `TinyVendingMachine`. The client program will ask to execute the unique ID of an instance of `TinyVendingMachine` in order to reference it.

As is well known, blockchains allow interleaving of transactions, but this is not enough to guarantee safety in Obsidian client programs. In the current implementation, dynamic state checks may not effectively ensure the typestate of remote references. Consider the following statements, where `r` is a remote reference:

```
if (r in S) {
    r.t();
}
```

Since `t()` is executed remotely, an interleaving transaction could change the state of `r` after the check and before the invocation of `t`.

3.4 Type system and Silica

To prove Obsidian's properties, the creators chose to use a *core calculus* called Silica. This language uses concepts and even notation from Featherweight Typestate (FT), a static language for TSOP designed in "Foundations of Typestate-Oriented Programming" [14]. However, they differ significantly from each other since they focus on different features. Like FT, Silica is an expression language: that is, sequencing is allowed only through nested let-bindings. In particular, it uses A-normal form to avoid nested expressions: for example, the statement

```
return f(g(1));
```

is equivalent to the Silica expression:

```
let x : T = g(1)
in f(x)
```

The Silica type system is expressed using several kinds of judgement that employ contexts for typing (Δ,Δ',...) and typing bounds (Γ,Γ',...). The first include local variables and temporary field types; the second represents a set of generic type variables.

Type splitting $\boxed{T_1 \Rrightarrow T_2/T_3}$ An other feature borrowed from FT is *type splitting*, which is a relation used to specify how ownership of objects must be managed among aliases. We say that a type T_1 is split into T_2 and T_3 when $T_1 \Rrightarrow T_2/T_3$. In particular, the relation is designed so that if one of the types on the right side has ownership, then it is held by T_2.

Consider the Obsidian statement `c2 = c1` where `c1 : Candy@Owned` and `c2 : Candy@Unowned`: in Silica this statement is analogous and requires the type of `c1` (T_1) splitting in two types (T_2 and T_3) and using them to update types of the references. In particular, `c1` will have type T_3 and `c2` type T_2. Since T_1 = `Candy@Owned`, then the ownership must be passed to one of the type on the right side of

$$\texttt{Candy@Owned} \Rrightarrow T_2/T_3$$

. As mentioned, the relation is designed to pass the ownership to T_2. Then, `Candy@Owned` $\Rrightarrow$ `Candy@Owned/Candy@Unowned` and after the assignment the types of `c1` and `c2` will be respectively `Candy@Unowned` and `Candy@Owned`: that is, ownership has been transferred from `c1` to `c2`.

Well-typed expressions $\boxed{\Gamma;\Delta \vdash_s e : T \dashv \Delta'}$ Silica expressions are typed using two typing contexts: the one on the left of $\vdash_s$ is the usual *input* typing context; the one on the right of $\dashv$ is the *output* typing context. This is needed because in Silica expressions can change the mode of references. Also, expressions are type-checked considering the invocation object s (i.e. the this reference).

Well-typed transaction $\boxed{M \textbf{ ok in } C}$ For this kind of judgement there are only two rules applicable: one for the public transactions and one for the private transactions. The main difference among them is that private transactions may have at the beginning or at the end fields inconsistent with their declarations.

Well-formed State $\boxed{ST \textbf{ ok}}$ These judgements are used to check if fields have distinct names and if the state is labeled as asset when any field is an asset.

Well-typed Contract, Interface and Program $\boxed{P \textbf{ ok}}$ These judgements check if contracts, interfaces and programs are well-typed (and well-formed) in any part of them: in particular, the judgement for contract (CL **ok**) checks if every transaction is well-typed and if every other part (states, type variables, ...) is well-formed; the judgement for interfaces ($IFACE$ **ok**) controls if the type parameters are well-formed; the judgement for programs (PG **ok**) checks if every contract and interface is well-typed.

Subpermission $\boxed{\Gamma \vdash T_1 <:_* T_2}$**,** $\boxed{\Gamma \vdash T_1 \not<:_* T_2}$ Subpermission specifies when an expression with contract C and mode T_{ST} can be used where an expression with contract C but mode T'_{ST} is expected. According to the definition of $<:_*$, we can determine that:

$$\mathsf{State} <:_* \mathsf{Owned} <:_* \mathsf{Shared} <:_* \mathsf{Unowned}$$

Subtyping $\boxed{\Gamma \vdash T_1 <: T_2}$ Subtyping judgements depends on the subpermission relation $<:_*$. Usually, this relation manages the type parameters and then delegates the check on permission to $<:_*$.

Then, for any contract C,

$$C@\mathsf{State} <: C@\mathsf{Owned} <: C@\mathsf{Shared} <: C@\mathsf{Unowned}$$

3.5 Properties

As Stipula, Obsidian addresses the safety and security issues using static typing. The main efforts in Obsidian were made to guarantee a safe and secure management of assets: in particular, in avoiding asset loss. Concurrency is mainly handled by the blockchain platform that ensures to execute transactions sequentially.

3.5.1 Safety

Obsidian has a full-fledged type system. Then, its type system ensures the typical properties of OO languages. In particular, the Progress Theorem proven for Silica says that programs may stuck only if they end up at a bad state transition, at a reentrant invocation or in a nested dynamic state check.

3.5.2 Asset Retention

The main property guaranteed by the Obsidian type system is *Asset Retention*, which states that a well-typed expression in an appropriate dynamic context drops Owned references to assets if such expression is a `disown` operation. In any other case, assets cannot be lost "accidentally". This is a key feature in Obsidian, since it ensures avoidance of bugs such as the one exploited in the DAO hack.

In particular, this property says that, if the expression e is closed, well-typed ($\Gamma; \Delta \vdash_s e : T \dashv \Delta'$) and not stuck, then its type is non-disposable (that is, holds ownership) or e disowns s at some point. Two other key hypotheses for the *Asset Retention* theorem are:

- in the input context (Δ) exists a reference having non-disposable type, that is, before the evaluation of expression e there is an asset reference in the context;
- in the output context (Δ') the type of every reference is disposable, that is, after the evaluation of e every reference is disposable.

For example, consider the following erroneous lines of code:

```
transaction willNotCompile() return Candy@Unowned {
    Candy c1 = new Candy(); // c1 : Candy@Owned
    Candy c2 = c1;  // c1 : Candy@Unowned,
                    // c2 : Candy@Owned
    return c1;
} // ERROR!
```

This code will cause the compiler to generate a compilation error at the end of the transaction notifying that the reference `c2` still holds ownership when deallocated. This fact is detected by the type system as follows. Consider the translation in Silica of the body of `willNotCompile`:

$$\begin{aligned} &\texttt{let c1 = new Candy()} \\ \text{in } &\texttt{let c2 = c1} \\ \text{in } &\texttt{c1} \end{aligned}$$

After the assignment `c2 = c1`, if we want to type the expression `c1`, we will need to prove the judgment

$$\begin{matrix}\texttt{c1} : \texttt{Candy@Unowned},\\ \texttt{c2} : \texttt{Candy@Owned}\end{matrix} \vdash_s \texttt{c1} : \texttt{Candy@Unowned} \dashv \begin{matrix}\texttt{c1} : \texttt{Candy@Unowned},\\ \texttt{c2} : \texttt{Candy@Owned}\end{matrix}$$

However, in the output context `c2` has type `Candy@Owned`. This means that after the evaluation of the expression `c1` and, consequently, after the whole expression, there are references having non-disposable types. Then, the expression does not respect the conditions for *Asset Retention* and the type system will find this through the rule (T-LET) (shown in chapter 5) which requires *let*-variables to have disposable types after the evaluation of the respective expression after the *in*.

However, the previous example can easily be "fixed" by disowning `c2`:

```
transaction willCompile() return Candy@Owned {
    Candy c1 = new Candy(); // c1 : Candy@Owned
    Candy c2 = c1;  // c1 : Candy@Unowned,
                    // c2 : Candy@Owned
    disown c2;  // c2 : Candy@Unowned
    return c1;
}
```

Then, after the disown expression, the typestate of `c1` will be checked with the judgment:

$$\begin{matrix}\texttt{c1} : \texttt{Candy@Unowned},\\ \texttt{c2} : \texttt{Candy@Owned}\end{matrix} \vdash_s \texttt{c1} : \texttt{Candy@Owned} \dashv \begin{matrix}\texttt{c1} : \texttt{Candy@Unowned},\\ \texttt{c2} : \texttt{Candy@Unowned}\end{matrix}$$

This time, every reference after the evaluation of `c1` is `Unowned` and, then, disposable. In particular, the (T-LET) rule will be completely satisfied since the type of all the local variables is disposable. Since there is a reference in the input context that has non-disposable type, the expression is closed, well-typed and not stuck, the conditions for *Asset Retention* are met: in fact, the transaction contains a `disown` statement.

The example shown is just a simple case where the Obsidian type system shows its ability to check how ownership is transferred among references. The rules of the type system check the disposability of references when ownership is actually transferred or lost: that is, in rules for assignment (T-ASSIGN), introduction/deallocation of local variables (T-LET), field updates (T-FIELDUPDATE) and state changes (T-$\rightarrow_p$).

Chapter 4

Comparison between Stipula and Obsidian

In this chapter, several examples are shown in order to compare the expressiveness of Stipula and Obsidian. The approach adopted was translating a program from one language to another to exploit the differences and the obstacles to implement certain features.

4.1 Introduction

In order to compare the Stipula and Obsidian languages, I wrote several examples using both.

Obsidian examples In Obsidian, the following examples are available:

- **CoinEscrow** This is an example that shows how a simple Stipula contract can be translated in an equivalent Obsidian contract. In particular, the contract manages the sending of currency among parties.
- **NftEscrow** This is analogous to the previous example, but in this case the asset managed is a token.
- **AssetSend** In this example, I implemented a contract that emulates the party of a contract and two different types of assets: `Currency` and `NFT`. The purpose is to understand the types needed for the $\multimap$ operator.
- **BikeRental** This is the implementation of the bike rental contract seen in the Stipula presentation paper [5]. Here, I found that the Stipula states are easily translated in their respective Obsidian states, that Stipula has a safer semantics (due to user identities dynamic checking) and an easy way to perform time-triggered events. Both versions can be read in their entirety in Appendix A.
- **Bet** This is the implementation of the betting contract seen in the presentation paper of Stipula. In particular, the original contract considered two betters bidding over the result of an agreed event. This example shows conclusions similar to those found in **BikeRental**. Available in Appendix A.

- **Auction** Simplified version of the example available in the Obsidian GitHub repository [2]: this is the usual auction in which bidders compete to win an object.

Stipula examples In Stipula, the following examples are available:

- **Auction** Translation of the auction example in Obsidian. This shows that it is easy to write a Stipula program that has the same semantics of an Obsidian program.
- **Parametric Insurance** This is the translation of the example available in the Obsidian repository [2]. This example represents the implementation of a parametric insurance contract involving as parties an insurance service and farmers: this kind of contracts provides a payout to the farmer when specific conditions (certified by a trusted provider) are met. In this specific case, they are related to the moisture of the terrain and the measurements are supplied by a trusted weather service.
  The original example uses user-defined types and dictionaries: neither of them is available in Stipula, yet. Then I adopted two different strategies:

  ① Focusing all the design on the Policy contract, avoiding aggregated data types;

  ② Keeping the focus on the InsuranceService contract, but using arrays as dictionaries.

- **ExampleTokenBank** Translation in Stipula of another example in the main Obsidian paper. This is an implementation of a contract that meets the ERC20 standard. The original code is available in the Obsidian repository [2].
  Again, the dictionaries are widely used and the guards of user identities in Stipula in this case may represent a disadvantage.

Listing 4.1: Stipula contract

```
1  stipula C {
2      asset w
3      field v1, v2, v3
4      init S1
5
6      agreement (A,B,C) (v1,v2) {
7          A,B : v1
8          B,C : v2
9      } ==> @S1
10
11     @S1 A : m(x)[h](cond) {
12         ...
13         h -o w
14         ...
15     } ==> @S2
16
17     ...
18 }
```

4.2 Translation from Stipula to Obsidian

To translate a contract from Stipula, we need at least to define the following elements:

- the contract itself,
- an interface for each party involved,
- their implementations,
- the main procedures that execute the code instantiated in the blockchain.

Consider the Stipula contract in Listing 4.1. We can notice that there are three main parts: the declarations of fields and assets, the agreement and the methods. Each of them corresponds to a part of the respective Obsidian contract (Listing 4.2).

4.2.1 Member and states declarations

The main obstacle in the translation of declarations is the management of asset ownership. The contract in Listing 4.1 declares the asset `w` which is a currency asset. In this case, simply adding the type declaration `Currency@Owned` is enough, because the contract always owns some currency: if `w` is equal to 0, then the contract owns 0 "dollars" (line 6 of Listing 4.2). When the asset is a token, we need to manage when this token is actually owned or not: we can do this by declaring a member of type `NFT@Owned` for each state in which the token must be owned by the contract `C` (this approach will be shown in detail in 4.2.6). For the other members, I simply added the respective type. Then, Obsidian requires us to declare all the states (unlike Stipula). In the translation in Listing 4.2, this part is contained in lines 6-15.

4.2.2 Agreement

To replicate the same semantics of the agreement in Stipula, we need to check that each parameter involved is the same for each party: if some of the parties do not agree, the construction of the contract must be interrupted. For example, the statement at line 7 of Listing 4.1

```
A,B : v1
```

Listing 4.2: Obsidian translation

```
asset interface A { ... }
asset interface B { ... }
asset interface D { ... }

main asset contract C {
    Currency@Owned w;
    T1 v1;
    T2 v2;
    T3 v3;

    state S1;
    state S2;
    ...

    A@Shared partyA;
    B@Shared partyB;
    D@Shared partyD;

    C@S1 (A@Shared pa, B@Shared pb, D@Shared pd) {
        if (pa.v1agreed() != pb.v1agreed()) {
            revert("Agreement failed");
        }

        if (pb.v2agreed() != pd.v2agreed()) {
            revert("Agreement failed");
        }

        v1 = pa.v1agreed();
        v2 = pb.v2agreed();
        v3 = T3();
        w = new Currency(0);

        partyA = pa;
        partyB = pb;
        partyD = pd;

        ->S1;
    }

    transaction m(C@S1 >> S2 this,
                  T1 x,
                  Currency@Owned >> Unowned h,
                  A@Unowned caller) {
        if (caller.id() != partyA.id()) {
            revert("Identity check failed");
        }

        if (!cond) {
            revert("Precondition not fulfilled");
        }

        ...
        w.merge(h);
        ...

        ->S2;
    }

    ...
}
```

becomes

```
if (pa.v1agreed() != pb.v1agreed()) {
    revert("Agreement failed");
}

v1 = pa.v1agreed();
```

To do this check, we need the references to the parties during construction and then the parameters `pa`, `pb` and `pd`. The parties are referenced as `Party@Shared`. The main reason is that objects representing parties are instances of `Party` contracts already deployed on the ledger at the moment of the instantiation of the contract `C`.

When we are sure that everyone agrees on the respective terms, then we can assign the fields with their respective values and apply the transition to the initial state. In Listing 4.2, the lines 21-40 of the Obsidian contract correspond to the agreement.

4.2.3 Methods

The main differences between Stipula methods and Obsidian transactions are the caller and precondition guards. However, the semantics of these two features can be easily replicated: firstly, we need to store in the contract the references to the parties (lines 17 - 19 in Listing 4.2) that can be easily obtained during the construction. Secondly, we have to add the respective checks inside the body of the transaction. Consider the method at line 11 (Listing 4.1): the guard requires the party `A` to be the only caller of `m`. Then, we will add at the beginning of the method the following lines:

```
if (caller.id() != partyA.id()) {
    revert("Identity check failed");
}
```

The same approach is adopted for the precondition guard.

4.2.4 Party interfaces

To apply the translations listed until now, the contract must provide interfaces for each party. Parties need to receive and send assets and share a unique identifier for approval of the invocations. Then, an interface should provide at least the following methods:

```
transaction give(int v) returns Currency@Owned;
transaction receive(Currency@Owned >> Unowned c);
transaction giveToken() returns NFT@Owned;
transaction receiveToken(NFT@Owned >> Unowned t);

transaction id() returns string;
```

Other methods can be added to the interfaces depending on the expected behaviour: for example, if in the method `m` a value `message` is sent to the party `B`, then the respective interface requires this transaction:

```
transaction sendMessage(string m);
```

At last, the getters to check the agreements on the terms of the contract must be added.

4.2.5 Party implementations and main procedures

In order to partecipate in a contract, the parties must implement the respective interfaces. For example, if the person who has the role `A` wants to join `C`, then must provide a `main contract AImpl` that implements the interface and must instantiate

an instance of this contract. After instantiation of every party on the blockchain, the contract `C` can be instantiated.

Now, in the blockchain, the contract `C` and the contracts for each party - `AImpl`, `BImpl` and `CImpl` - are available. At this point, every party has the interest in running a `main contract` with a main procedure to interact with the contract `C`. An example may be Listing 4.3, in which the party A may decide to call the transaction `m` of a given contract `C`. In order to execute the `main` transaction, the party must know its own unique ID and the one associated to the correct instance of contract `C`. Then, once the `main` transaction is called, it checks if the contract is in the correct state, that is `S1` (line 7). If so, an object of type `T1` is created (line 8) and passed as an argument of the `C`'s transaction `m` along with a proper amount of currency (withdrawn from the party itself) and the reference to the party (line 9).

Listing 4.3: Example of main procedure

```
1  import "C.obs"
2  import "A.obs"
3
4  main contract MainA {
5      transaction main(remote A@Shared party,
6                       remote C@Shared contr) {
7          if (contr in S1) {
8              T1 x = T1();
9              contr.m(x,A.give(5),A);
10         }
11     }
12 }
```

4.2.6 Managing NFT variables

In Stipula, a token variable may be “full" or “empty" depending on the value contained. This aspect in Obsidian is handled through ownership: a NFT variable typed with `NFT@Owned` is “full" and, viceversa, one with `NFT@Unowned` as type is “empty". Hence, the ownership of a token must be managed at compile-time, using the appropriate Obsidian tools available (i.e. the typestates). In general, a contract owns an NFT variable if in the current state exists a member of type `NFT@Owned`. Instead, if in the current state the variable does not occur or it does but with type `NFT@Unowned`, then the contract does not own the token.

For example, consider the contract in Listing 4.4 that has only one state `I`. To properly manage the ownership of `nft`, we need two states that correspond respectively to `I` when the token is owned and when it is not. In the first, a `NFT@Owned nft` member must be declared. However, this approach may lead to an explosion of states: in fact, if we had two token variables, four states would be required. We can mitigate this problem avoiding to define states that are surely not reached: for example, if in a certain state `X` a token is always “full", then the Obsidian state `X_NoToken` is surely not needed. The translation showing these two states can be found in Listing 4.5.

Listing 4.4: NFTManage

```
stipula NFTManage {
    asset nft
    init I

    agreement(A)() {} ==> I

    @I A : getToken()[t] {
        t -o nft;
    } ==> @I

    @I A : leaveToken()[] {
        nft -o A;
    } ==> @I
}
```

Listing 4.5: Translation of NFTManage

```
main asset contract NFTManage {
    A@Shared partyA;

    state I_Token {
        NFT@Owned nft;
    }
    state I_NoToken;

    NFTManage@I_NoToken(A@Shared pa) {
        partyA = pa;
        ->I_NoToken;
    }

    transaction getToken(NFTManage@I_NoToken>>I_Token this,
                         NFT@Owned>>Unowned t,
                         A@Unowned caller) {
        if (caller.id() != partyA.id()) {
            revert("Call unauthorized");
        }
        ->I_Token(nft = t);
    }

    transaction leaveToken(NFTManage@I_Token>>I_NoToken this,
                           A@Unowned caller) {
        if (caller.id() != partyA.id()) {
            revert("Call unauthorized");
        }
        A.receiveToken(nft);
        ->NoToken;
    }
}
```

Listing 4.6: CoinEscrow in Stipula

```
1  stipula CoinEscrow {
2      asset w
3      field amount
4
5      agreement(Sender,Receiver)(amount){
6          Sender,Receiver : amount
7      } ==> @S1
8
9      @S1 Sender : put()[h] (h==amount) {
10         h -o w
11     } ==> @S2
12
13     @S2 Receiver : claimCoins()[] {
14         w -o Receiver
15     } ==> @End
16
17     @S2 Receiver : claimPart(v)[] (v>=0 and v<=1) {
18         w*v -o w,Receiver
19         w -o Sender
20     } ==> @End
21 }
```

4.3 Examples

4.3.1 CoinEscrow

This examples shows how a simple Stipula contract is translated using the process discussed in section 4.2. The contract in Listing 4.6 manages the send of currency from a sender to a receiver. In particular, the receiver can claim all the money sent or just a part of them. In the second case, the remaining part of the escrow is returned to the sender. In particular, the contract follows this flow:

① the parties agree on the amount to send or receive;

② the sender puts the currency in the contract;

③ the receiver claims the amount (or a part of it) contained in the contract.

The contract is very simple and can be translated in a straightforward way as shown in Listing 4.7.

If we compare the two versions of the function `put`, we will notice that in Stipula the *send*-statement is not mandatory. In that case, the token given in input will be lost. This chance should be avoided: in fact, in the Obsidian version, the variable `h` must lose ownership and, without the statement `w.merge(h)` (line 56), the contract cannot be compiled. On the other hand, Obsidian cannot analyse the liquidity of a contract: in the function `claimPart`, if the statement `w -o Sender` (line 19) is missing, the liquidity analyser provided in Stipula will notice the error. However, the analogous Obsidian transaction could be written without the last statement and the compiler would not point out any error. A workaround may be declaring the assets for all states except the state `End` and make sure to send properly any asset before reaching the end-state (as shown in Listing 4.8). In detail, the line 16 of Listing 4.7 is replaced with line 11 in Listing 4.8: the difference between them is that, in the first case, the asset `w` is declared for every state of the contract and that, in the second case, the same asset is declared only for states `S1` and `S2`. Follows that, Listing 4.8, `w` cannot be owned by `CoinEscrow` when it reaches the state `End` (i.e. the asset cannot be "frozen" in the contract). However, this expedient is not enough: in fact, we have to make sure that

Listing 4.7: CoinEscrow in Obsidian

```
main contract CoinEscrow {
    Currency@Owned w;
    int amount;

    Sender@Shared sender;
    Receiver@Shared receiver;

    state S1;
    state S2;
    state End;

    CoinEscrow@S1(Sender@Shared s, Receiver@Shared r) {
        if (s.amountAgreed() != r.amountAgreed()) {
            revert("Agreement failed");
        }

        sender = s;
        receiver = r;

        amount = s.amountAgreed();
        w = new Currency@Owned(0);
        ->S1;
    }

    transaction put(CoinEscrow@S1 >> S2 this,
                    Currency@Owned >> Unowned h,
                    Sender@Unowned caller) {
        if (caller.id() != sender.id()) {
            revert("Call unauthorized");
        }

        if (amount != h.getAmount()) {
            revert("put failed: amount incorrect");
        }

        w.merge(h);
        ->S2;
    }

    transaction claimCoins(CoinEscrow@S2 >> @End this,
                           Receiver@Unowned caller) {
        if (caller.id() != receiver.id()) {
            revert("Call unauthorized");
        }

        receiver.receive(w.split(w.getAmount()));
        ->End;
    }

    transaction claimPart(CoinEscrow@S2 >> @End this,
                          int v,
                          Receiver@Unowned caller) {
        if (caller.id() != receiver.id()) {
            revert("Call unauthorized");
        }

        if (v < 0 || v > 100) {
            revert("claimPart failed: v must be a value for the percentage");
        }

        receiver.receive(w.split((w.getAmount()*v)/100));
        sender.receive(w.split(w.getAmount()));
        ->End;
    }
}
```

Listing 4.8: Liquidity in Obsidian

```
1  main contract CoinEscrow {
2      int amount;
3
4      Sender@Shared sender;
5      Receiver@ Shared receiver;
6
7      state S1;
8      state S2;
9      state End;
10
11     Currency@Owned w available in S1,S2;
12
13     CoinEscrow@S1(Sender@Shared s, Receiver@Shared r) { ... }
14
15     transaction put(CoinEscrow@S1 >> S2 this,
16                     Currency@Owned >> Unowned h,
17                     Sender@Unowned caller) { ... }
18
19     transaction claimCoins(CoinEscrow@S2 >> @End this,
20                            Receiver@Unowned caller) {
21         if (caller.id() != receiver.id()) {
22             revert("Call unauthorized");
23         }
24
25         receiver.receive(w);
26         ->End;
27     }
28
29     transaction claimPart(CoinEscrow@S2 >> @End this,
30                           int v,
31                           Receiver@Unowned caller) {
32         if (caller.id() != receiver.id()) {
33             revert("Call unauthorized");
34         }
35
36         if (v < 0 || v > 100) {
37             revert("claimPart failed: v must be a value for the percentage");
38         }
39
40         receiver.receive(w.split((w.getAmount()*v)/100));
41         sender.receive(w);
42         ->End;
43     }
44 }
```

`w` is disowned in the transaction that reaches the state `End` (that is, `claimCoins` and `claimPart`). To do so, we can just replace the expression `w.split(w.getAmount())` in lines 60 and 76 of Listing 4.7 with `w` (lines 25 and 41 of Listing 4.8): recall that `Party::receive` disowns the `Currency` received in input.

Listing 4.9: NftEscrow in Stipula

```
stipula NftEscrow {
    asset nft

    agreement(Sender,Receiver)(){} =>@S1

    @S1 Sender : put()[t] () {
        t -o nft
    } ==> @S2

    @S2 Receiver : claimNft()[] {
        nft -o Receiver
    } ==> @End
}
```

4.3.2 NFTEscrow

This example is analogous to the previous one but with NFT instead of currency. Consider the Stipula contract in Listing 4.9. This contract deals with the transfer of tokens from sender to receiver. The contract has a flow similar to that defined for **CoinEscrow**:

① the parties agree to join the contract;

② the sender puts in the contract the NFT;

③ the receiver claims the token.

This time the sent asset cannot be claimed partially because it is a token.

The contract can also be easily translated following the same process as before in the Obsidian contract in Listing 4.10.

We can also make the same remark as in the previous example: the *send*-statement in the `put` function in Stipula can be omitted, but not in Obsidian. We can notice that the states of the Stipula version are not duplicated in the Obsidian version, since in the original contract in state `S1` the variable `nft` is always “empty" and in state `S2` it is always “full". Then, the states `S1_Token` and `S2_NoToken` in Obsidian are not needed. Consequently, we have a single Obsidian state for each Stipula state and we can assign the same name accordingly.

Listing 4.10: NFTEscrow in Obsidian

```
main contract NftEscrow {

    Sender@Shared sender;
    Receiver@Shared receiver;

    state S1;
    state S2 {
        NFT@Owned nft;
    }
    state End;

    NftEscrow@S1(Sender@Shared s, Receiver@Shared r) {
        sender = s;
        receiver = r;

        ->S1;
    }

    transaction put(NftEscrow@S1 >> S2 this,
                    NFT@Owned >> Unowned t,
                    Sender@Unowned caller) {
        if (caller.id() != sender.id()) {
            revert("Call unauthorized");
        }

        ->S2(nft = t);
    }

    transaction claimNft(NftEscrow@S2 >> @End this,
                         Receiver@Unowned caller) {
        if (caller.id() != receiver.id()) {
            revert("Call unauthorized");
        }

        receiver.receiveToken(nft);
        ->End;
    }
}
```

Listing 4.11: Implementation of Currency

```
main asset contract Currency {
    int value;

    Currency@Owned(int v) {
        value = v;
    }

    transaction getValue(Currency@Unowned this) returns int {
        return value;
    }

    transaction split(Currency@Owned this, int v) returns Currency@Owned {
        if (v > value) {
            revert;
        }
        value = value - v;
        Currency result = new Currency(v);
        return result;
    }

    transaction merge(Currency@Owned this, Currency@Owned >> Unowned other) {
        value = value + other.getValue();
        disown other;
    }
}
```

4.3.3 AssetSend

This example presents a way of implementing a party of a contract using Obsidian. In particular, we are interested in how parties send or receive assets using Obsidian typing and how the ownership relates to these operations. Statements of interest are the following:

- `v` —∘ `h,A` and `v` —∘ `h,h'` where `h` and `h'` are currencies and `A` is a party;
- `h` —∘ `A` and `h` —∘ `h'` where `h` and `h'` are tokens and `A` is a party.

In order to design parties, we also need to realise types for currencies and tokens. `Currency` (Listing 4.11) and `NFT` (Listing 4.12) are standard implementations of assets and are very similar to their counterparts in the main Obsidian article [4]. `Currency` is a contract that stores in an integer field the total value of that currency. This contract provides three transaction:

- `getValue`, which allows to obtain the raw integer value of the currency;
- `split`, which subtracts a given amount of currency and return it as currency;
- `merge`, which adds the value stored from another currency to its own (the currency merged must be disown to avoid asset duplication).

`NFT` is just a representation of a non-fungible token and it does not expose particular transactions: in fact, it provides a method to obtain the string that identifies the token and a method to determine if two tokens are equal or not.

I designed the party as a contract that has a wallet with currency and a list of all owned NFTs (Listing 4.13). Then, each party must provide the necessary transactions to send or receive either currency and tokens. Note that the state of each gift in the “receive" transactions mutates from `Owned` to `Unowned`, as ownership needs to be given up. In general, the statements in the same row of Table 4.1 have the same

Listing 4.12: Implementation of NFT

```
main asset contract NFT {
    string ID;

    transaction getID() returns string {
        return ID;
    }

    transaction equals(NFT@Unowned other) returns bool {
        return ID == other.getID();
    }
}
```

	Stipula	Obsidian
Currency	`v —○ h,A` `v —○ h,h'`	`A.receive(h.split(v))` `h'.merge(h.split(v))`
Token	`h —○ A` `h —○ h'`	`A.receiveToken(h)` `h' = h`

Table 4.1: Asset send statements

semantics: the only exception is the sending of tokens from one variable to another (`h —○ h'` and `h' = h`). In Stipula, if a variable referencing a token is updated without draining the previous NFT value, then a runtime error will occur. In Obsidian, this error is detected by the type system: in particular, in Stipula "empty" and "full" token variables correspond, respectively, to Obsidian variables with types `NFT@Unowned` and `NFT@Owned`. Follows that the state of the variable is known (and guaranteed) at compile-time and the change of state "empty"/"full" (i.e. `NFT@Unowned`/`NFT@Owned`) is completely managed by the ownership system. Since the assignment `h' = h` is well-typed only if `h'` has type `NFT@Unowned` and `h` has type `NFT@Owned`, we cannot assign a new token value to a "full" variable. For this reason, we can avoid checking at runtime the state of the variable.

The consequence is that the Stipula asset-send operator (—○) should apply the same types of respective transaction arguments: that is,

`Numeric —○ Currency@Owned, Party`	for currencies given to a party
`Numeric —○ Currency@Owned, Currency@Owned`	for merging currencies
`NFT@Owned»Unowned —○ Party`	for NFT given to a party
`NFT@Owned»Unowned —○ NFT@Unowned»Owned`	for NFT moved between variables

where `Numeric` is a type that refers to a positive integer.

Listing 4.13: Implementation of Party

```
main asset contract Party {
    Currency@Owned wallet;
    List[NFT@Owned]@Owned tokens;

    Party@Owned (int m) {
        wallet = new Currency(m);
        tokens = new List[NFT@Owned]();
    }

    transaction give(int v) returns Currency@Owned {
        return wallet.split(v);
    }

    transaction receive(Currency@Owned >> Unowned gift) {
        wallet.merge(gift);
    }

    transaction getAmount() returns int {
        return wallet.getValue();
    }

    transaction receiveToken(NFT@Owned >> Unowned gift) {
        tokens.push(gift);
    }

    transaction giveToken(NFT@Unowned t) returns NFT@Owned {
        return tokens.removeElement(t,new NFTComparator());
    }

    transaction getToken(int i) returns NFT@Unowned {
        return tokens.get(i);
    }
}
```

4.3.4 BikeRental

The original example can be consulted in Listing A.11 (Appendix A). The rental contract manages the interactions between a lender and a borrower and includes a central authority to handle legal disputes. Usually, a contract of this kind should follow the following flow:

① the parties agree on the terms of the contracts as the renting time and the cost;

② the lender offers a bicycle to the borrower;

③ the borrower pays for the bike;

④ the borrower gives back the bicycle (and this needs to happen before the expiration of the renting time).

Once the contract is effective (that is, after payment), the lender or the borrower can signal irregularities to the authority, providing their reasons. This entity has the power to decide which party is right and to formulate a verdict.

We can use the BikeRental example to compare the expressiveness of Stipula and Obsidian. For this purpose, it is better to analyse separately the various parts of the contracts: declarations, constructors and methods. Another important topic concerns the contract instantiation: we will see that a time manager is required in order to emulate the time-triggered events system of Stipula. Hence, we need to handle properly the instantiation of this additional element.

Fields and asset declarations As we can see from Listing 4.14 and Listing 4.15, the Stipula list of declarations is more concise than the Obsidian one due to the following Obsidian additional requirements:

- explicit type declaration for each field;
- complete list of states (eventually with the respective fields);
- shared references to parties in order to check the identity in transactions: it is good practise to use interfaces instead of contracts to decouple parties of a contract from their implementations (in the example `Borrower`, `Lender` and `Authority` are interfaces);
- reference to a time manager that emulates the event mechanism of Stipula.

The first two may lead to declarations less clear to the beginner or non-technical reader (such a lawyer), but at the same time may help the programmer to reason efficiently on the contract and find errors faster.
The last two are required because Obsidian has not the corresponding features: in particular, Stipula promotes a safer approach providing primitives to check the caller identity and an additional native feature to manage time-triggered events.

Listing 4.14: Declarations in Stipula

```
1 stipula BikeRental {
2     asset wallet
3     field cost, rentingTime, code
4     init Inactive
```

Listing 4.15: Declarations in Obsidian

```
74 main asset contract BikeRent {
75     // Fields
76     int cost;
77     int rentingTime;
78
79     // Parties
80     Lender@Shared lender;
81     Borrower@Shared borrower;
82     Authority@Shared authority;
83
84     // Time
85     TimeManager@Shared timeManager;
86
87     state Inactive;
88     state Payment;
89     state Using {
90         int expirationTime;
91     };
92     state Return;
93     state End;
94     state Dispute;
95
96     string code available in Payment,Using,Return;
97     Currency@Owned wallet available in Using,Return,Dispute;
```

Construction/Agreement The `agreement` is the Stipula method that allows the contract construction, ensuring that parties agree on the same contract terms. In Obsidian, in order to obtain the same semantics, this agreement requires an explicit implementation. For instance, the statement in line 7 in Listing 4.16 can be implemented in Obsidian checking for each party if the terms agreed (lines 102-107 in Listing 4.17).

Assignments for parties and the time manager in Stipula are not required since they are handled natively.

Listing 4.16: Agreement in Stipula

```
6   agreement (Lender,Borrower,Authority)(rentingTime,cost){
7       Lender , Borrower: rentingTime , cost
8   } ==> @Inactive
```

Listing 4.17: Constructor in Obsidian

```
99  BikeRent@Inactive(
100     Lender@Shared l, Borrower@Shared b, Authority@Shared a,
101     TimeManager@Shared tm){
102     // Fields in the agreement
103     if (l.costAgreed() != b.costAgreed() ||
104             l.rentingTimeAgreed() != b.rentingTimeAgreed()) {
105         revert("Agreement failed");
106     }
107
108     cost = l.costAgreed();
109     rentingTime = l.rentingTimeAgreed();
110
111     // References to parties
112     lender = l;
113     borrower = b;
114     authority = a;
115
116     // Time manager
117     timeManager = tm;
118
119     ->Inactive;
120 }
```

Transactions/Methods If the Stipula methods do not have events defined in their body, they can be easily translated into Obsidian transactions: for example, the method `verdict` in Listing 4.18 has the same semantics as the homonymous transaction in Listing 4.19. In particular, caller guards can be translated into a `revert` statement executed when the given identity does not match the one declared during the construction: in the example, the guard on `Authority` corresponds to the lines 202-204 in Listing 4.19. Note that the transaction needs a `Unowned` reference as an argument to verify the caller's identity.

Listing 4.18: `verdict` method in Stipula

```
39  @Dispute Authority : verdict(x,y)[] (y>=0 && y<=1) {
40      x -> Lender
41      x -> Borrower
42      (y*wallet) -o wallet, Lender
43      wallet -o Borrower;
44      _
45  } ==> @End
```

Listing 4.19: `verdict` transaction in Obsidian

```
205     transaction verdict(BikeRent@Dispute >> End this,
206                         Authority@Unowned a, string x, int y) {
207         // Checks
208         if (a.id() != authority.id()) {
209             revert("Error on authority authentication");
210         }
211 
212         if (y < 0 || y > 100) {
213             revert("Error on bounds over y");
214         }
215 
216         // Body
217         lender.sendMotivations(x);
218         borrower.sendMotivations(x);
219 
220         int total = (wallet.getValue()*y)/100;
221         Currency reimbursement = wallet.split(total);
222 
223         lender.receive(reimbursement);
224         borrower.receive(wallet);
225 
226         ->End;
227     }
```

When the Stipula methods define an event in their body, the translation to their Obsidian counterparts is not trivial, like in the previous example. Consider the method `pay` in Listing 4.20: in order to replicate the same behaviour in Obsidian, we should add the event to the stack of events.

Listing 4.20: `pay` method in Stipula

```
15     @Payment Borrower : pay()[h] (h == cost) {
16         h -o wallet
17         code -> Borrower;
18         now+rentingTime >>
19             @Using {
20                 "EndReached" -> Borrower
21             } ==> @Return
22     } ==> @Using
```

For this purpose, we need a contract that manages the flow of time (the `TimeManager`) and a representation for the events (the `EventInterface`). As can be seen in Listing 4.21, once the TimeManager starts, it cannot be stopped by any party. The execution of the events is delegated to the `tick` transaction, which also manages the passage of time.

Listing 4.21: `TimeManager` in Obsidian

```
1  main asset contract TimeManager {
2      Dict[Integer,List[EventInterface]@Owned]@Owned registry;
3      int clock;
4 
5      state Active;
6      state Inactive;
7 
8      transaction register(int t, EventInterface@Owned >> Unowned event) {...}
9      transaction tick(TimeManager@Active this) {...}
10     transaction start(TimeManager@Inactive this) {...}
11     transaction now() returns int {...}
12 }
```

Listing 4.22: `EventInterface` contract in Obsidian

```
5  interface EventInterface {
6      transaction action();
7  }
```

To correctly register events in the time manager registry, every contract must provide an implementation of `EventInterface`: in the bike rental example, Listing 4.23 shows the event provided for `BikeRent`. During the instatiation of the event, two references are needed:

- a shared `BikeRent` reference to the contract to apply effects on the contract;
- an unowned `Borrower` reference to check the caller identity.

Observing `action` in `Event`, note also that the main contract (i.e. `BikeRent`) must provide the reaction as transaction (Listing 4.24).

Listing 4.23: `Event` contract in Obsidian

```
58 contract Event implements EventInterface {
59     BikeRent@Shared rental;
60     Borrower@Unowned caller;
61
62     Event@Owned(BikeRent@Shared r, Borrower@Unowned c) {
63         rental = r;
64         caller = c;
65     }
66
67     transaction action() {
68         if (rental in Using) {
69             rental.reaction(caller);
70         }
71     }
72 }
```

Listing 4.24: `reaction` transaction in Obsidian

```
153     transaction reaction(BikeRent@Using >> Using | Return this,
154                          Borrower@Unowned b) {
155         if (b.id() != borrower.id()){
156             revert("Not authorized borrower");
157         }
158         borrower.sendMessage("End_Reached");
159         ->Return;
160     }
```

At this point, we can replicate the semantics of the `pay` method by writing the homonymous transaction as shown in Listing 4.25. However, the instruction in line 148 causes a typing error, since `this` is an Owned reference and cannot be shared with other objects. This problem can be solved by calling the `register` transaction every time `pay` is called (as in Listing 4.26).

Listing 4.25: `pay` transaction in Obsidian

```
132 transaction pay(BikeRent@Payment >> Using this,
133                 Borrower@Unowned b,
134                 Currency@Owned >> Unowned h) {
135     // Checks
136     if (b.id() != borrower.id()) {
137         revert("Not authorized borrower");
138     }
139
140     if (h.getValue() != cost) {
141         revert("Currency given don't match with cost");
142     }
143
144     // Body
145     borrower.sendCode(code);
146
147     // REACTION
148     timeManager.register(timeManager.now(),new Event(this,b));
149     ->Using(wallet = h, expirationTime = timeManager.now() + rentingTime);
150 }
```

Listing 4.26: Call of `pay` and `register` in `BorrowerMain`

```
1 if (rental in Payment) {
2     rental.pay(borrower,borrower.give(costAgreed));
3     TimeManager timeManager = rental.getTimeManager();
4     timeManager.register(
5         timeManager.now() + borrower.rentingTimeAgreed(),
6         new Event(rental,borrower));
7 }
```

However, this is not a satisfying way to manage events: in fact, the programmer may forget to register the event or (even worse) deliberately avoid doing it. The reaction triggered may be against the interests of the caller: for instance, in the bike rental example the borrower after the payment could avoid to trigger the expiration and rent bikes without a time limit.

Instantiation In general, a contract translated from Stipula to Obsidian requires:

① Each party to instantiate the respective contract;

② Someone else (ideally a central authority) to instantiate the time manager and the contract itself.

After this phase, the same actor who instantiated the time manager should run the main procedure `TimeManagerMain.obs` (Listing A.24) in order to start the contract clock. Finally, any procedure executed by any party may occur in any order.
In the BikeRental example, the instantiation of the contracts must follow this specific ordering (shown in Figure 4.1):

① parties implementation - `BorrowerImpl`, `LenderImpl`, `AuthorityImpl`,

② `TimeManager`,

③ `BikeRent`.

Then, `TimeManagerMain` must be executed and, finally, the Main contracts of each party - `BorrowerMain`, `LenderMain`, `AuthorityMain` - may run in any order.

This ordering can be easily extrapolated from the dependency between contracts (Figure 4.2). In fact, `TimeManager`, `BorrowerImpl`, `LenderImpl`, `AuthorityImpl` are

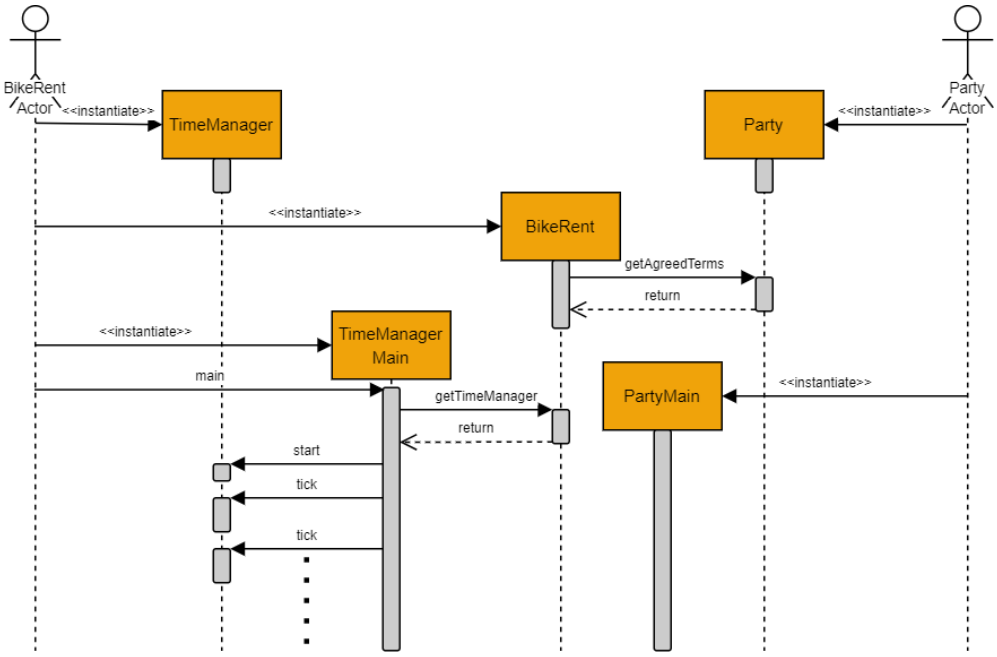


Figure 4.1: Instantiation of bike rental example contracts

independent of any other contract and `BikeRent` depends on the previous four. Ideally, the same actor that instantiates the rental contract should also be responsible for the time management and therefore of the `TimeManager` and `TimeManagerMain` instantiations. In particular, the second must be instantiated just after `BikeRent` does. Finally, several main contracts are instantiated since they all depend on `BikeRent`.

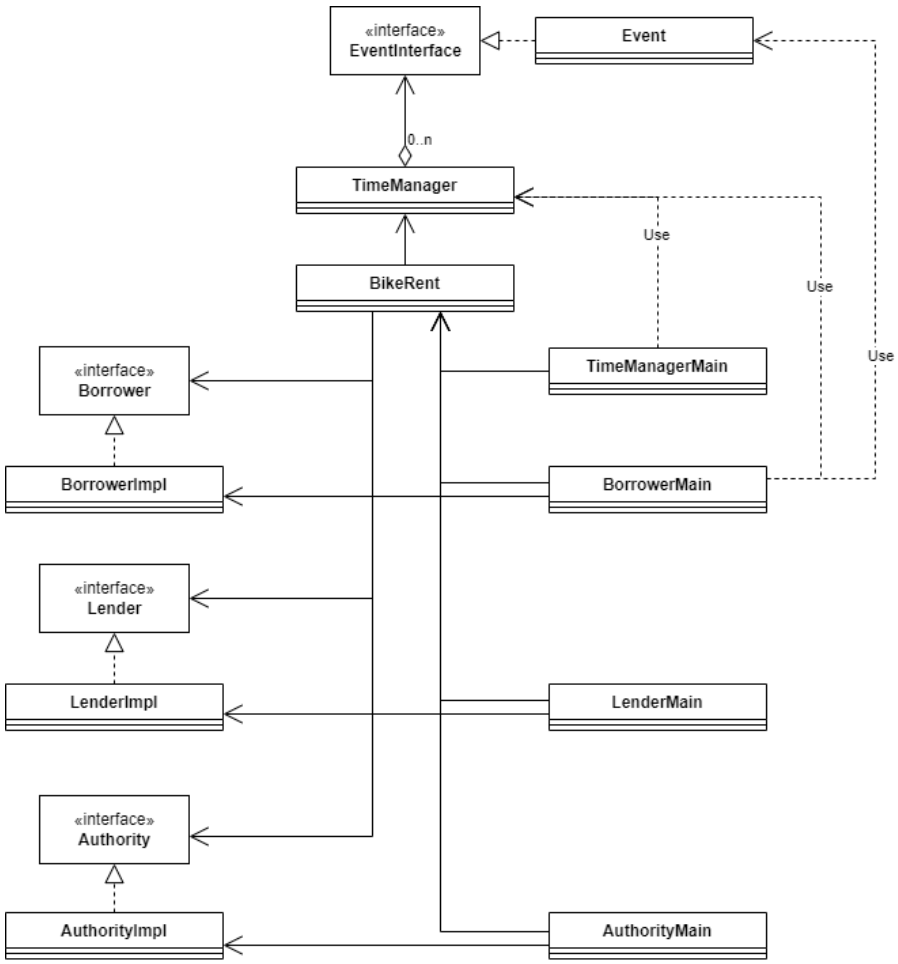


Figure 4.2: Dependencies in BikeRental

4.3.5 Bet

The original example can be consulted in Listing A.25 (Appendix A). The contract manages two betters (`Better1` and `Better2`) that gamble their currency on a given event. The data about this event are provided by a third party (`DataProvider`). For the sake of simplicity, the betters place their bet in turn within an agreed time limit. Then, the contract should follow this flow:

① Betters and `DataProvider` agree on the event, the data source and the its start time;

② Betters agree on the amount of the bet and the time limits;

③ `Better1` places his bet;

④ `Better2` places his bet;

⑤ `DataProvider` provides the result of the event and the winner is rewarded with the amount of currency agreed.

However, if the time runs out (according to the agreed time limits), the currency placed will be returned.

The arguments that we can discuss for this contract are, mostly, the same as the ones for BikeRental. The main differences between these examples are:

- the presence of an overloaded function `place_bet` in the Stipula version;
- the contract defines two different events.

So, the Bet examples allow us to understand better how the time-triggered management may be implemented in Obsidian.

Overloaded function Stipula allows the definition of functions with the same name: in this particular case (Listing 4.27), the two functions are disambiguated by the state in which they can be called (`Init` and `First` respectively).
Since Obsidian doesn't allow overloading, the translation (Listing 4.28, Listing 4.29) provides disambiguation giving different names to different functions. An alternative way to avoid two different names may be to add dynamic checking on typestates like in Listing 4.30. However, this approach introduces a less precise precondition and postcondition on the typestate of `this`.

Listing 4.27: `place_bet` transactions in Stipula

```
12      @Init Better1 : place_bet(x)[h] (h == amount) {
13          h -o wallet1
14          x -> val1;
15          t_before >> @First {
16              wallet1 -o Better1
17          } ==> @Fail
18      } ==> @First
19
20      @First Better2 : place_bet(x)[h] (h == amount) {
21          h -o wallet2
22          x -> val2
23          alea -> DataProvider;
24          t_after >> @Run {
25              wallet1 -o Better1
26              wallet2 -o Better2
27          } ==> @Fail
28      } ==> @Run
```

Listing 4.28: `place_bet1` transaction in Obsidian

```
transaction place_bet1(Bet@Init >> First this,
                       BetterInterface@Shared b1,
                       string x,
                       Currency@Owned >> Unowned h) {
    if (b1.id() != better1.id()) {
        revert("Error on better1 authentication");
    }

    if (h.getValue() != amount) {
        revert("Bet doesn't match the amount agreed");
    }

    // REACTION 1
    ->First(val1 = x, wallet1 = h);
}

transaction reaction1 (Bet@First >> First | Fail this,
                       BetterInterface@Unowned b1) {
    if (b1.id() != better1.id()) {
        revert("Error on better1 authentication");
    }

    better1.receive(wallet1);
    ->Fail;
}
```

Listing 4.29: `place_bet2` transaction in Obsidian

```
transaction place_bet2(Bet@First >> Run this,
                       BetterInterface@Shared b2,
                       string x,
                       Currency@Owned >> Unowned h) {
    if (b2.id() != better2.id()) {
        revert("Error on better2 authentication");
    }

    if (h.getValue() != amount) {
        revert("Bet doesn't match the amount agreed");
    }

    // REACTION 2
    ->Run(val2 = x, wallet2 = h);
}

transaction reaction2 (Bet@Run >> Run | Fail this,
                       BetterInterface@Unowned b2) {
    if (b2.id() != better2.id()) {
        revert("Error on better2 authentication");
    }

    better1.receive(wallet1);
    better2.receive(wallet2);
    ->Fail;
}
```

Listing 4.30: Alternative translation of `place_bet`

```
transaction place_bet(Bet@(Init|First) >> (First|Run) this,
                      BetterInterface@Shared b,
                      string x,
                      Currency@Owned >> Unowned h) {

    if (h.getValue() != amount) {
        revert("Bet doesn't match the amount agreed");
    }

    if (this in Init) {
        if (b.id() != better1.id()) {
           revert("Error on better1 authentication");
        }
        // REACTION 1
        ->First(val1 = x, wallet1 = h);
    } else {
        if (b.id() != better2.id()) {
            revert("Error on better2 authentication");
        }
        // REACTION 2
        ->Run(val2 = x, wallet2 = h);
    }
}
```

Events In order to emulate the Stipula events semantics, I partially took inspiration from the formal description given in the article [5] where the events are managed through a *multiset* of events. In my implementation, `TimeManager` handles a *multimap* of events that have as keys the respective triggering times. For example,

$$\Psi = \begin{Bmatrix} (0, \{W_{0_0}, W_{0_1}, ...\}), \\ (1, \{W_{1_0}, W_{1_1}, ...\}), \\ ... \\ (n, \{W_{n_0}, W_{n_1}, ...\}) \end{Bmatrix}$$

If the clock managed by `TimeManager` reaches the value t, then every event W_{t_l} must be triggered. This is the purpose of the transaction `tick`, which also advances the clock.
In Figure 4.3, the management of the event registered by `Better1` is shown.

Listing 4.31: `tick` transaction of `TimeManager`

```
    transaction tick(TimeManager@Active this) {
        Integer time = new Integer(clock);
        Option[List[EventInterface]] events = registry.remove(time);

        if (events in Some) {
            performActions(events.unpack());
        }

        clock = clock + 1;
    }
```

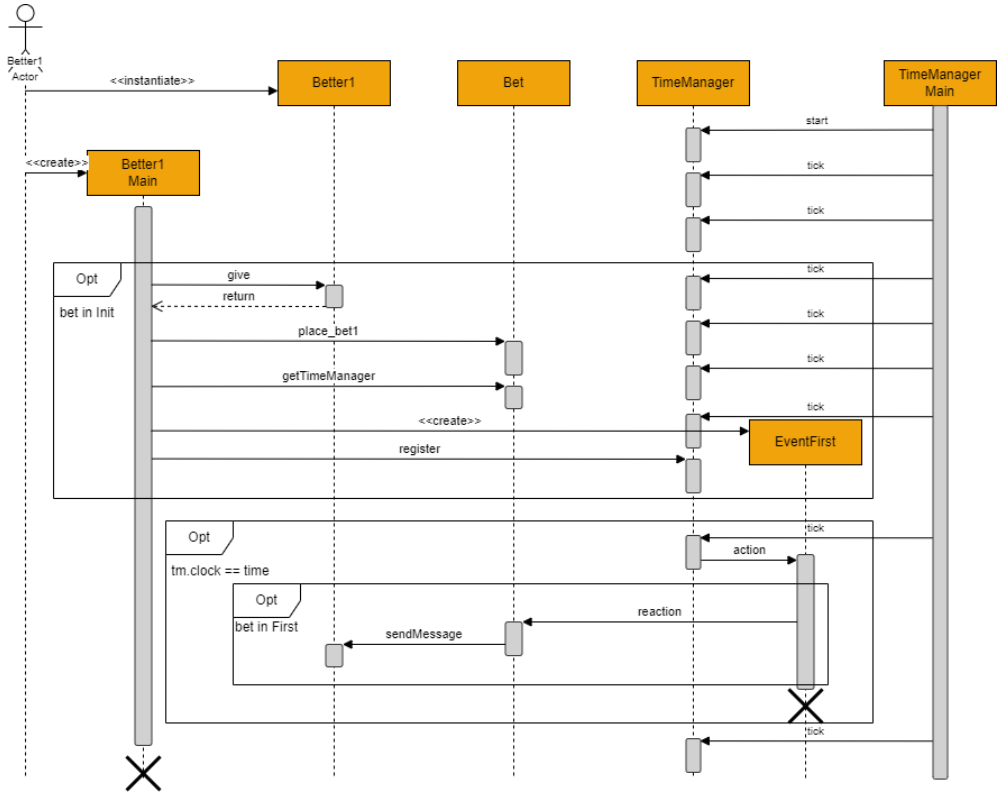


Figure 4.3: `EventFirst` management in Bet example

4.3.6 Auction

The Auction example used is a simplified version of the one available in the official repository [2]. In this contract, several bidders compete to obtain an item given from a seller during an auction. The flow of the contract is the following:

① the seller offers an item and the auction starts;

② the bidders make their bids: the greatest offer is also the best;

③ the bidding phase finish;

④ the auction is closed, the winner is rewarded with the object and the seller receives the currency from the bid.

It is pretty easy to see that the contract in Listing 4.33 is a straightforward implementation of the one in Listing 4.32. The main difference is represented by the necessity to initialise separately the fields `maxBidder` and `token` (lines 10 - 14), since in the agreement *send*-statements are not allowed. Types such as `Seller` or `Bidder` are not required in Stipula, since they only manage the movement of assets between parties and Stipula achieves the same effect natively.
If we compare the translation processes Stipula-to-Obsidian and Obsidian-to-Stipula, we can notice that the first one is harder: this fact occurs because Obsidian does not provide a native mechanism to implement time-triggered events. The choice may be desired since time synchronisation is a non-trivial problem and can introduce vulnerabilities [18]: the main reason is that a timestamp depends on the miner's local system. In Stipula, we usually consider a central authority that manages contracts. Hence, the unique and unambiguous timestamp is the one given by this authority.

Listing 4.32: `Auction` in Obsidian

```
82  main contract Auction {
83
84      state Open {
85          Item@Owned item;
86          Bid@Owned bid;
87      }
88      state BiddingDone {
89          Item@Owned it;
90          Bid@Owned finalBid;
91      }
92      state Closed {
93          Seller@SoldItem sellerSatisfied;
94          Bidder@WonItem winner;
95      }
96
97      Seller@Unsold seller available in Open, BiddingDone;
98      Bidder@Bidding maxBidder available in Open, BiddingDone;
99
100     Auction@Owned(Item@Owned >> Unowned i) {
101         Open::maxBidder = new Bidder("none", 0, 0);
102         Open::seller = new Seller();
103         ->Open(item = i, bid = new Bid(0));
104     }
105
106     transaction makeBid(Auction@Open this, Bidder@Bidding >> Unowned bidder)
            {
107         if (bidder.getBidAmount() > bid.getAmount()) {
108             if (maxBidder.getName() != "none") {
109                 if (bid in Open) {
110                     maxBidder.returnBidMoney(bid);
111                 }
112             }
113
114             bid = bidder.createBid();
115             maxBidder = bidder;
116         }
117     }
118
119     transaction finishBidding(Auction@Open >> BiddingDone this) {
120         -> BiddingDone(it = item, finalBid = bid);
121     }
122
123     transaction giveItem(Auction@BiddingDone >> Closed this) {
124         maxBidder.won(it);
125         seller.receiveBid(finalBid);
126         ->Closed(sellerSatisfied = seller, winner = maxBidder);
127     }
128 }
```

Listing 4.33: `Auction` in Stipula

```
stipula Auction {
    asset wallet, item
    field objectCode, maxBidder
    init Init

    agreement (Seller, Bidder1, Bidder2, Auctioneer)(objectCode) {
        Seller, Bidder1, Bidder2 : objectCode
    } ==> @Init

    @Init Seller : offer(obj)[token] (obj == objectCode) {
        "none" -> maxBidder
        token -o item;
        _
    } ==> @Open

    @Open Bidder1 : makeBid()[bid] (bid > wallet && maxBidder != "one") {
        if (maxBidder == "two") {
            wallet -o Bidder2
        }
        bid -o wallet
        "one" -> maxBidder;
        _
    } ==> @Open

    @Open Bidder2 : makeBid()[bid] (bid > wallet && maxBidder != "two") {
        if (maxBidder == "one") {
            wallet -o Bidder1
        }
        bid -o wallet
        "two" -> maxBidder;
        _
    } ==> @Open

    @Open Auctioneer : stopBidding()[] {
        "End" -> Bidder1
        "End" -> Bidder2;
        _
    } ==> @BiddingDone

    @BiddingDone Auctioneer : giveItem()[] {
        if (maxBidder == "one") {
            item -o Bidder1
            wallet -o Seller
        } else if (maxBidder == "two") {
            item -o Bidder2
            wallet -o Seller
        } else { //maxBidder == "none" and wallet == 0
            item -o Seller
        };
        _
    } ==> @End
}
```

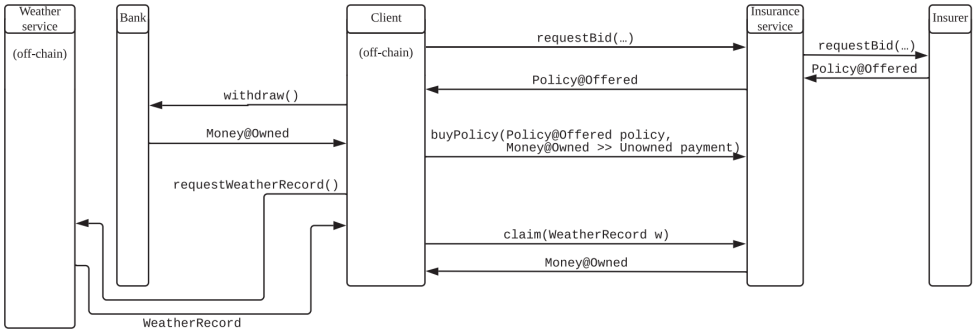


Figure 4.4: Parametric Insurance in Obsidian

4.3.7 Parametric Insurance

The original example is incomplete because it was intended to be used as case study in the usability studies of typestates. For this reason, I decided to replicate the intended behaviour described in the article (Figure 4.4). The parametric insurance contract includes the following parties: an insurer, a farmer and a weather service. If the insurer provides policy through an insurance service, then this one can be considered a party who acts in stead of the insurer and offers several policies or a contract itself that guarantees the access to policies. In general, a policy follows this flow:

① the farmer and the insurance service agree on the contract parameter such as the cost or the parameter involved;

② the farmer buys the policy;

③ the insurance service activates the policy;

④ the weather service signals the achievement of the condition for the agreed parameter;

⑤ the farmer claims the agreed payout from the insurance service.

If the weather does not signal the achievement of the condition, then the policy expires and the escrow is returned to the insurance service.

There are two main possible ways to implement parametric insurance:

- focus the design on the policy itself: in the original case study it is considered as an inert entity, that is, it does not regulate the interactions among insurer and farmer, but it is just an asset with additional information;
- focus the design on the insurance service, following the original design.

The first way allows for a more flexible approach, since every policy may be stipulated with different values from any other policy. At the same time, the agreement is needed every time, even if the insurer agrees only on a (possibly) small set of policy kinds. In this case, the second design solves the problem more properly.

Policy-focused design If we consider a policy as a handle to manage the relationship between the insurer and the farmer, the most reasonable approach is to incorporate

the interactions between the parties in the contract itself (Listing 4.35).
The choice of a particular policy is easily implemented with the agreement (lines 8-15) that leads the policy in the `Offered` state. If the farmer buys the policy (lines 17-20), then the activation from the insurance service is required (lines 22-27): once activated, we need also to provide the time-triggered event that manages the expiration. At this point, there are two chances:

- the policy expires and the escrow is returned to the insurance service;
- the weather service provides a moisture value below the agreed threshold. In this case, the farmer has the right to claim the payout decided during the agreement.

The states `Activating` and `Claimable` in Listing 4.35 in the Obsidian version don't exist: the policy in that case is just an asset with several states and the other parties manage the transactions. In my translation, these states are required in order to estabilish the correct order of asset-send statements and imposed by the caller guards, since both the farmer and the insurance service must send currency and a method like Listing 4.34 is not allowed: in fact, the statement in line 3 is illegal.

Listing 4.34: (Fictional) `buy` method

```
1 @Offered Farmer : buy()[m] (m == cost) {
2     m -o InsuranceService
3     payout -o InsuranceService, escrow;
4     now + expirationTime >> @Active {
5         escrow -o InsuranceService
6     } ==> @Expired
7 } ==> @Active
```

Service-focused design Although the previous approach is a good way to handle the relationship between parties, it does not allow tracking if the farmer stipulated several policies. If we are interested in this aspect, we have to consider an insurance service as a contract where the insurer dictates the terms of a preset number of policy kinds and the farmer can only choose to agree or not. In this case, the insurance service follows the following flow:

① the insurer proposes the conditions for each kind of policy proposed;

② the farmer purchases one or more policies;

③ the weather service signals if some of the policies met the conditions;

④ the farmer claims the payout for each claimable policy;

⑤ go back to ②.

If the weather does not signal the achievement of the condition for any of the policies, then they expire, the escrows are returned to the insurer and the policy is returned to the insurer.

In the example (Listing 4.36), the insurer offers N kinds of policy available in the asset `policies`. This variable is an array of currency assets, where at position k we can find the number of policies of kind k offered by the insurer. The same reasoning is applied for the fields `payouts`, `costs`, `conditions`, `costs` and `expTimes`. The fields `actives`, `expirations` and `claimables` are support variables that help the contract track the state of the policies. Instead, `n` is used to check if the farmer returned the expired policy in its possession.

Listing 4.35: `Policy` contract in Stipula

```
stipula Policy {
    asset escrow
    field cost, expirationTime,
          longitude, latitude, radius,
          moistureContent, payout
    init Offered

    agreement (Farmer, InsuranceService, WeatherService)
              (cost, expirationTime,
               longitude, latitude, radius,
               moistureContent, payout) {
        Farmer, InsuranceService : cost, expirationTime,
                                   longitude, latitude, radius,
                                   moistureContent, payout
    } ==> @Offered

    @Offered Farmer : buy()[m] (m == cost) {
        m -o InsuranceService; // cost -o m, InsuranceService
        _
    } ==> @Activating

    @Activating InsuranceService : activate()[m] (m == payout) {
        m -o escrow;
        now + expirationTime >> @Active {
            escrow -o InsuranceService
        } ==> @Expired
    } ==> @Active

    @Active WeatherService : checkMoist(moist)[] (moist < moistureContent) {
        ;
        _
    } ==> @Claimable

    @Claimable Farmer : claim()[] {
        escrow -o Farmer;
        _
    } ==> @Claimed
}
```

To write this contract, I assumed that Stipula provides arrays and allows us to use the methods' parameters in the events definitions. If the number of kind of contracts is known, then it is possible to write a contract without this additional assumptions.
In any case, we can easily notice that, even if this approach enables the tracking of the number of policies sold, expired or claimed, at the same time it is less natural for Stipula: in general, this contract is less understandable than the one with the policy-focused design.

Listing 4.36: `InsuranceService` contract in Stipula

```
stipula InsuranceService {
    asset policies, escrow
    field payouts,
          costs,
          conditions,
          expTimes,
          actives,
          expirations,
          claimables,
          n
    init Ready

    agreement (Farmer, Insurer, WeatherService)
              (payouts, conditions, expTimes) {
        Insurer : payouts, costs, conditions, expTimes
    } ==> @Ready

    @Ready Insurer : offer(k)[p,e]
    ((k >= 0 || k < payouts.length) && p > 0 && e == payouts[k]*p) {
        p -o policies[k]
        e -o escrow
        conditions[k] -> WeatherService;
        _
    } ==> @Ready

    @Ready Farmer : buy(k)[m]
    ((k >= 0 && k < policies.length) && policies[k] > 0 && costs[k] == m) {
        1 -o policies[k], Farmer
        m -o Insurer
        actives[k] + 1 -> actives[k]

        now + expTimes[k] >> @Ready@PolicyExpired {
            if (k < n) {
                k -> n
            }
            expirations[k] + 1 -> expirations[k]
            actives[k] - 1 -> actives[k]
            payouts[k] -o escrow, Insurer
        } ==> @PolicyExpired
    } ==> @Ready

    @Ready WeatherService : conditionsMet(k)[] {
        if (actives[k] > 0) {
            claimables[k] + 1 -> claimables[k]
            actives[k] - 1 -> actives[k]
        };
        _
    } ==> @Ready

    @Cashing Farmer : claim(k)[p] (p <= claimables[k]) {
        p -o Insurer
        (payouts[k] * p) -o escrow, Farmer
        claimables[k] - 1 -> claimables[k];
        _
    } ==> @Ready

    @PolicyExpired Farmer : returnExpiredPolicy(k)[p] (p <= expirations[k]) {
        p -o Insurer
        expirations[k] - p -> expirations[k];
        _
    } ==> @PolicyExpired

    @PolicyExpired Insurer : checkExpiredReturns()[] (expirations[n] == 0) {
        n + 1 -> n;
        _
    } ==> @PolicyExpired

    @PolicyExpired Insurer : ready()[] (n == policies.length) {
        0 -> n;
        _
    } ==> @Ready
}
```

Listing 4.37: `ERC20` interface for the `ExampleTokenBank`

```
asset interface ERC20 {
    transaction totalSupply() returns int;
    transaction balanceOf(int ownerAddress) returns int;
    transaction transfer(int fromAddress, int toAddress, int value) returns
        bool;

    // Allow ownerAddress to withdraw from your account, multiple times,
    // up to the value amount.
    transaction approve(int ownerAddress, int fromAddress, int value) returns
         bool;

    // Returns the amount of allowance still available.
    transaction allowance(int ownerAddress, int fromAddress) returns int;

    // Transfers tokens from an allowance that has already been granted.
    transaction transferFrom(int senderAddress,
                             int fromAddress, int toAddress,
                             int value) returns bool;
}
```

4.3.8 ExampleTokenBank

The example provided by Obsidian developers is an implementation for a bank that meets the ERC20 standard [26] represented by the interface in Listing 4.37: in general, an ERC20 Token Contract allows one to obtain information about balances (`totalSupply`, `balanceOf`), transfer tokens between balances (`transfer`, `transferFrom`) and manage allowances (`approve`, `allowance`). An allowance is a permission to a user to send from an account a certain amount of currency. For example, consider users A, B and C, with their respective accounts. If there exists an allowance for user A to move at most 10 dollars from the account of C, then A can make transfers from the C account to any other account until the limit of 10 dollars is reached without any limit on the number of transactions.

The original contract in Obsidian is available in Listings A.43(Appendix A). In this implementation, the bank does not show any state, since there is no precise flow. In general, the only requirement for an owner to transfer currency is that he or she has an appropriate allowance.

If we assume again that arrays are provided and that the sequential address starts from 0, then the implementation of the ERC20 contract can be realised as shown in Listing 4.38: the only asset needed is the array of balances. The allowances is a bidimensional array: the first index is an owner address; the second is the address from which the owner is allowed to transfer tokens; the respective value is the amount of tokens that the owner can move from the specified balance.
The “transfer" methods can be easily implemented since Stipula manages assets natively between operations: note that they require just one line of code. The same happens for the `approve` method. Stipula enables also a safer approach thanks to the guards on preconditions. The main problem is represented that the number of owners must be fixed and known in the agreement: a new party cannot join the contract subsequently. Another issue is represented by the presence of multiple `transferFrom` methods (one for each owner): this can be solved trading off safety with conciseness avoiding to check who calls the methods.
At last, `totalSupply`, `balanceOf` and `allowance` cannot be implemented in Stipula since the language does not allow to return values from methods. Hence, the implementation of Stipula cannot meet the ERC20 standard.

Listing 4.38: `ExampleTokenBank` contract in Stipula

```
stipula ExampleTokenBank {
    asset balances
    field allowances
    init Working

    agreement(Owner1, Owner2, Bank)(allowances) {
        Bank : allowances
    } ==> @Working

    @Working Bank : transfer(fromAddress, toAddress, value)[]
    (value <= balance[fromAddress]) {
        value -o balances[fromAddress],balances[toAddress];
        _
    } ==> @Working

    @Working Bank : approve(ownerAddress, fromAddress, value)[]
    ((ownerAddress == 0 || ownerAddress == 1)
     && (fromAddress >= 0 && fromAddress < balances.length)) {
        value -> allowances[ownerAddress][fromAddress];
        _
    } ==> @Working

    @Working Owner1 : transferFrom(fromAddr,toAddr,value)[]
    ((fromAddr >= 0 && fromAddr <= balances.length)
     && (toAddr >= 0 && toAddr <= balances.length)
     && allowances[0][fromAddr] >= value) {
        value -o balances[fromAddr],balances[toAddr];
        _
    } ==> @Working

    @Working Owner2 : transferFrom(fromAddr,toAddr,value)[]
    ((fromAddr >= 0 && fromAddr <= balances.length)
     && (toAddr >= 0 && toAddr <= balances.length)
     && allowances[1][fromAddr] >= value) {
        value -o balances[fromAddr],balances[toAddr];
        _
    } ==> @Working

}
```

4.4 Conclusions

In general, Stipula adopts a safer and more flexible approach in the writing of legal contracts than Obsidian, thanks to the tools available for the programmer: caller and precondition guards, time-triggered events, agreements and dedicated asset operations are necessary elements in legal contracts. Providing these features simplifies implementation, improves readability and forces developers to write safer programs: in particular, asset operations promote safety on currency or tokens transfers and guards encourage the programmer to reason about who calls certain procedures and the conditions in which they are called.
On the other side, Stipula lacks features necessary for more complex design due to the inability to write user-defined data types and data structure. Another disadvantage in Stipula is the absence of a full-fledged type system in order to ensure certain properties (such as *Asset Retention* [4]) and even improve the understanding of contracts. Finally, time-triggered events can introduce vulnerabilities if time is not managed by a centralised and certified authority.
As general consideration, we can notice that Stipula is more effective when used to manage interactions between parties. Obsidian is more "general" in this sense because it allows a larger variety of designs.

4.4.1 Conciseness and readability

Usually a contract in Stipula is more concise than any Obsidian contract with similar semantics. This fact is a consequence of several features of Stipula:

- **Lack of types and states declarations:** The absence of declarations makes the code more concise and readable for the users, in particular if they are not programmers who have been used to statically typed languages. Although there are no declarations, we have to recall that Stipula has types (for fields and assets) and states, but there is not a full list in the "declarative" part of the contract. This helps in conciseness, but at the same time encourages programmers to think about states as they write methods. This approach is the opposite of the one adopted by Obsidian, that forces programmers to write - and design - states before the methods implementation and, then, encourages a less spontaneous approach.

- **Agreement:** A respective procedure in Obsidian would require several code lines, but in Stipula this task is completed by writing just one line.

- **Time-triggered events:** As discussed previously, the implementation of a time-triggered event is a non-trivial task in Obsidian, unlike the way it is provided in Obsidian. Also, the design that I provided presents problems.

- **Dedicated asset-send statement:** Instead of developing contracts that model assets and parties, providing them natively (along with the respective *send*-statements) is the most readable and concise approach.

As a consequence, a Stipula contract seems more readable than an Obsidian contract. It fails only when data structures or user-defined data types are required.

4.4.2 Safety

Stipula introduces some features that allow safety by design, but at the same time it lacks a complete type system. In particular, the language enhances safety through:

- **Caller and precondition guards:** Verifying the caller's identity and preconditions is surely a safer approach, since it gives additional guarantees while programming procedures and encourages the programmer to reason about who calls certain methods and the conditions in which they are called. We can obtain the same behaviour by adding some checks in the first lines of each method: however, this approach is error-prone, since the programmer may forgot to write them.

- **Native asset-send statements:** The *send*-statements, as seen in the AssetSend example, can be obtained by encoding proper contracts for tokens and currency. In Stipula, since they are directly provided by the language, programmers do not need to implement their own version, refraining from accidentally introducing bugs or undesired behaviour. Additionally, this feature allows developers to develop dedicated tools for static analysis that may improve safety.

- **Liquidity analysis:** Stipula is provided with a liquidity analiser that checks if every asset is redeemed by a party involved in the agreement. This property is not guaranteed by Obsidian typing. However, it can be encoded defining an end-state without any asset member (as shown in Listing 4.8).

With respect to safety, Stipula has two main problems:

- **A weak type system:** The language applies type inference and has a weak type system that mainly manages the safety of operators and updates. Obsidian, on the other hand, also manages statically the ownership of assets, which enables further static properties. For example, the Stipula type system cannot detect if a token gets lost in a method.

Listing 4.39: Loss of assets in methods

```
stipula C {
    asset NFT
    ...
    @S A : method()[token] {
        NFT -o B
        token -o h;
        -
    }
    ...
}
```

In Listing 4.39, the token in the input is sent to a local variable `h` that will be lost after execution of the method. This possibility cannot occur in Obsidian due to the static analysis of the ownership.

- **Possible timestamp vulnerabilities:** the introduction of time-triggered events may introduce timestamp dependences [18]. Timestamp dependences may cause vulnerabilities: in fact, the timestamp of the single node is easily modifiable. Consider the `pay` transaction of the bike rental example (Listing 4.20) and the event:

Features	Stipula	Obsidian
Types declarations	No	Yes natively
States declarations	Incomplete	Yes natively
Agreement	Yes natively	Yes by suitable encoding
Time-triggered events	Yes natively	Yes by suitable encoding
Asset-send statements	Yes natively	Yes by suitable encoding
Caller guards	Yes natively	Yes by suitable encoding
Precondition guards	Yes natively	Yes by suitable encoding
Ownership	No	Yes natively
User-defined types	No	Yes natively
Data structures	No	Yes by suitable encoding
Type safety	Operators, updates	Yes natively + Asset retention
Liquidity	Yes	Yes by suitable encoding

Table 4.2: Comparison of Stipula and Obsidian main features

```
@Using {
    "EndReached" -> Borrower
} ==> @Return
```

The borrower may decide to increase the timestamp of its node, then execute `pay`. Follows that `now` evaluates to an inappropriate timestamp: in particular, `now + rentingTime` exceeds the actual expiration time and allows the borrower to rent bikes beyond the agreed renting time. This problem can be solved if there is a central licenced authority that provides the timestamps.

Chapter 5

Typing for asset *send*-statements

In this chapter, we introduce the proofs that show well-typedness for the Obsidian statements that correspond to the send-statements in Stipula. For each statement, the translation in Silica and the respective proofs are provided. The conclusions include some observations on how the Stipula type system could be enhanced.

5.1 Introduction

We are interested in the typing of the *send*-statements in Table 4.1. As mentioned above, the types found in the examples are:

`Numeric` $\multimap$ `Currency@Owned, Party`	for currencies given to a party
`Numeric` $\multimap$ `Currency@Owned, Currency@Owned`	for merging currencies
`NFT@Owned⟩Unowned` $\multimap$ `Party`	for NFT given to a party
`NFT@Owned⟩Unowned` $\multimap$ `NFT@Unowned⟩Owned`	for NFT moved between variables

However, if we want a formal guarantee on the properties proven for Obsidian, proofs must be provided. In particular, in Stipula, it may be interesting to add the *Asset Retention* property. In this way, we will be always sure that assets cannot be lost during the execution of any contract. The first step to achieve this goal is looking at the proofs of the equivalent Silica statements and studying how the type system behaves.

For each statement, we first want to translate the Obsidian statement in Silica and, secondly, to prove its soundness.

5.2 Most used rules in proofs

In the proofs of this chapter, we will use several rules from the Silica type system: in particular, the most important to understand are (PublicTransactionOk), (T-$\rightarrow_p$), (T-let) and (T-disown).

5.2.1 The PublicTransactionOk rule

The (PublicTransactionOk) rule (Figure 5.1) checks if a public transaction is well-typed. To do so, the rule must identify the type parameters through *params* and find the type variables with Var. Then, assert that the type bounds context is formed by the found types and the type parameters of the specific transaction. Finally, the

(PublicTransactionOk)

$$\frac{params(C) = \overline{T_G} \quad \overline{Var(T_G) = T_V} \quad \Gamma = \overline{T_G}, \overline{T_M} \quad \Gamma; \texttt{this} : C\langle\overline{T_V}\rangle@T_{\texttt{this}}, \overline{x : C_x@T_x} \vdash_{\texttt{this}} e : T \dashv \texttt{this} : C\langle\overline{T}\rangle@T'_{\texttt{this}}, \overline{x : C_x@T'_x}}{Tm\langle\overline{T_M}\rangle(\overline{C_x@T_x \gg T'xx})T_{\texttt{this}} \gg T'_{\texttt{this}}\{\texttt{return } e\} \textbf{ ok in } C}$$

Figure 5.1: (PublicTransactionOk) rule

(T-let)

$$\frac{\Gamma; \Delta \vdash_s e_1 : T_1 \dashv \Delta' \quad \Gamma; \Delta', x : T_1 \vdash_s e_2 : T_2 \dashv \Delta'', x : T'_1 \quad \Gamma \vdash disposable(T'_1)}{\Gamma; \Delta \vdash_s \text{let } x : T_1 = e_1 \text{ in } e_2 : T_2 \dashv \Delta''}$$

Figure 5.2: (T-let) rule

rule requires the expression to be well-typed with the proper input and output contexts, which are formed by the `this` object with its input and output types respectively and the transaction parameters (along with their declared input and output types respectively).

5.2.2 The T-let rule

The T-let rule (Figure 5.2) checks typing in *let-in* expressions. A *let-in* expression is well-typed if:

① the expression to assign (e_1) has the same type as the new variable (x);

② the remaining expression (e_2) returns a value with the same type as the whole expression;

③ x can be thrown away after the evaluation of the whole expression.

If we read carefully the rule, we will notice that the output typing context of the first hypothesis (Δ') is the input typing context of the second hypothesis: in fact, since typestate may change after the evaluation of any expression and that statement in Silica are "concatenated" using *let-in* expressions, then the typing contexts must be also concatenated consistently. In particular, the input context of the whole expression must be the input context for the evaluation of e_1. Hence, its output context must be the input for e_2 with the addition of x with the declared type. The output context of the whole expression is the output context of the second hypothesis, but without considering x and its type since it will be disposed after the evaluation of e_2.

5.2.3 The T-disown rule

The (T-disown) rule (Figure 5.3) checks of `disown` expressions. In particular, they are well-typed if the permission for the expression to disown (s') is a state S or Owned. The output type of s' is obtained through type splitting. Since T_{ST} is at most Owned and, as mentioned, the type splitting transfers its ownership to the first type on the right side of the relation, by definition of type splitting T' will be a disposable type. Then, after the evaluation of the disown expression, s' will have T'.

(T-DISOWN)

$$\frac{T_C@T_{ST} \Rrightarrow T/T' \qquad \Gamma \vdash T_{ST} <:_* \mathsf{Owned}}{\Gamma; \Delta, s' : T_C@T_{ST} \vdash_s \texttt{disowns}' : \mathsf{unit} \dashv \Delta', s' : T'}$$

Figure 5.3: (T-DISOWN) rule

(T-$\rightarrow_p$)

$$\frac{\begin{array}{c}\Gamma \vdash T_{ST} <:_* p \qquad p \in \{\mathsf{Owned}, \mathsf{Shared}\} \qquad \Gamma; \Delta \vdash_s \overline{x} : \overline{T} \dashv \Delta' \\ \Gamma; \Delta \vdash T <: stateFields(C\langle\overline{T_A}\rangle, S') \qquad unionFields(C\langle\overline{T_A}\rangle, T_{ST}) = \overline{T_{f_s}\ f_s} \\ fieldTypes_s(\Delta; \overline{T_{f_s}\ f_s}) = \overline{T'_{f_s}} \qquad \Gamma \vdash \overline{disposable(T'_{f_s})}\end{array}}{\Gamma; \Delta, s : C\langle\overline{T_A}\rangle@T_{ST} \vdash_s s \rightarrow_p S'(\overline{x}) : \mathsf{unit} \dashv \Delta', s : C\langle\overline{T_A}\rangle@S'}$$

Figure 5.4: (T-$\rightarrow_p$) rule

5.2.4 The T-$\rightarrow_p$ rule

The (T-$\rightarrow_p$) rule checks of the statements that changes the state of objects are well-typed. The rule requires several conditions to be verified:

- the permission of the object that is changing state (T_{ST}) is compatible with p, which is Owned or Shared by grammar;
- the parameters of the state initialisation ($\overline{x}$) are well-typed with respect to the types of the fields in the contract declaration ($stateFields(C\langle\overline{T_A}\rangle, S')$);
- that types of the fields of the actual state that are inconsistent with their declarations ($\overline{T'_{f_s}}$) are disposable.

5.3 Moving a token between variables

Consider the statement that moves a token from a variable to another: `h` $\multimap$ `h'`. According to its semantics in Stipula, after the statement the memory ℓ will be modified in this way:

$$\ell' = \ell[h \mapsto a', h' \mapsto a'']$$

where

$$\begin{aligned}[\![h]\!]^{\mathsf{a}}_{\ell} &= a \\ [\![h - a]\!]^{\mathsf{a}}_{\ell} &= a' \\ [\![h' + a]\!]^{\mathsf{a}}_{\ell} &= a''\end{aligned}$$

Then, following the semantics shown in subsection 2.2.5,

$$\begin{aligned}[\![h]\!]^{\mathsf{a}}_{\ell} &= \ell(h) = a & \\ [\![h - a]\!]^{\mathsf{a}}_{\ell} &= 0T = a' & \text{if } [\![h]\!]^{\mathsf{a}}_{\ell} = [\![a]\!]^{\mathsf{a}}_{\ell} \\ [\![h' + a]\!]^{\mathsf{a}}_{\ell} &= [\![a]\!]^{\mathsf{a}}_{\ell} = a = a'' & \text{if } [\![h']\!]^{\mathsf{a}}_{\ell} = 0T\end{aligned}$$

Since $[\![h]\!]^{\mathsf{a}}_{\ell} = \ell(h) = a = [\![a]\!]^{\mathsf{a}}_{\ell}$, we have $a' = 0T$ in any case. Then, there are two possible cases:

		h’	
	h —∘ h’	“empty”	“full”
h	“empty”	“empty”,“empty”	Runtime Error
	“full”	“empty”,“full”	Runtime Error

Table 5.1: Values of (h, h’) after h —∘ h’

		h’	
	h’ = h	Unowned	Owned
h	Unowned	Unowned, Unowned	Compile-time Error
	Owned	Unowned, Owned	Compile-time Error

Table 5.2: Types of (h, h’) after the statement h’ = h

- if $\ell(h') = 0T$, then and $a'' = a$;
- if $\ell(h') \neq 0T$, then an error occurs.

This means that if a token value is sent to another token variable, this one must be “empty”. Then, in Obsidian a token must be sent to an Unowned asset variable. In particular, Stipula allows us to assign an “empty” value to a token variable with already an “empty” value. The outcomes of the *send*-statement are briefly summarised in Table 5.1. We can note that they are similar to the results of the type-checking (Table 5.2) for the analogous Obsidian statement: that is, h’ = h which is simply translated into the Silica statement h’ := h. If we embrace the analogy “empty”/Unowned - “full”/Owned, h —∘ h’ and h’ := h have a similar behaviour in absence of errors. When they appear, then Obsidian is able to detect them before the execution thanks to its type system.

5.3.1 Derivation tree

The proof for the assignment is very simple and is based mainly on rule T-Assign that requires the type of h’ to be *disposable*: that is, h’ is not an asset or is not Owned. Then, the type of h is split and the only way allowed by the type system is to give ownership to h’. The proof with Unowned mode for both variables is similar.

$$\dfrac{\dfrac{\texttt{NFT} = \mathit{contract}(\texttt{NFT@Owned})}{\texttt{NFT@Owned} \Rrightarrow \texttt{NFT@Owned}/\texttt{NFT@Unowned}}\ \textsc{Split-Unowned} \qquad \dfrac{\overline{\mathit{notOwned}(\texttt{NFT@Unowned})}}{\Gamma \vdash \mathit{disposable}(\texttt{NFT@Unowned})}}{\begin{array}{c}\Gamma;\Delta,\\ \texttt{h'} : \texttt{NFT@Unowned}, \vdash_s \texttt{h'} := \texttt{h} : \texttt{unit} \dashv \texttt{h'} : \texttt{NFT@Owned},\\ \texttt{h} : \texttt{NFT@Owned} \qquad\qquad \texttt{h} : \texttt{NFT@Unowned}\end{array}}\ \textsc{T-Assign}$$

If we try to prove the same statement considering h’ as Owned, then it will be impossible to proceed due to the *disposable* requirement:

$$\dfrac{\dfrac{\ldots}{\texttt{h} : \texttt{NFT@}P_i \Rrightarrow \texttt{NFT@}P'_o/\texttt{NFT@}P_o} \qquad \dfrac{\ldots}{\Gamma \vdash \mathit{disposable}(\texttt{NFT@Owned})}}{\begin{array}{c}\Gamma;\Delta,\\ \texttt{h'} : \texttt{NFT@Owned}, \vdash_s \texttt{h'} := \texttt{h} : \texttt{unit} \dashv \texttt{h'} : \texttt{NFT@}P'_o,\\ \texttt{h} : \texttt{NFT@}P_i \qquad\qquad \texttt{h} : \texttt{NFT@}P_o\end{array}}\ \textsc{T-Assign}$$

5.4 Sending token to a party

Consider the statement sending a token to a party: `h` $\multimap$ `A`. According to its semantics in Stipula, after the statement the memory ℓ will be modified in this way:

$$\ell' = \ell[h \mapsto a']$$

where

$$\begin{aligned} [\![h]\!]^{\mathtt{a}}_{\ell} &= a \\ [\![h - a]\!]^{\mathtt{a}}_{\ell} &= a' \\ \ell(\mathtt{A}) &= A \end{aligned}$$

Then, following the semantics shown in subsection 2.2.5,

$$\begin{aligned} [\![h]\!]^{\mathtt{a}}_{\ell} &= \ell(h) = a \\ [\![h - a]\!]^{\mathtt{a}}_{\ell} &= 0T = a' \qquad\qquad \text{if } [\![h]\!]^{\mathtt{a}}_{\ell} = [\![a]\!]^{\mathtt{a}}_{\ell} \end{aligned}$$

Since $[\![h]\!]^{\mathtt{a}}_{\ell} = \ell(h) = a = [\![a]\!]^{\mathtt{a}}_{\ell}$, then $a' = 0T$ in any case: that is, if the variable `h` contains a "full" value, it will be drained and the token will be sent to the party A. However, Stipula allows sending "empty" values: that is, no token is actually sent to A. Then, the only case that really makes sense in terms of ownership is the first one.

As shown in Table 4.1, the respective statement is `A.receiveToken(h)` which just adds the token to the list owned by the party. In Obsidian, we can send Unowned references, but sending them is not equivalent to sending $0T$ in Stipula: in fact, an Unowned reference is not a zero value. For this reason, I considered only this case non-viable in Obsidian.

5.4.1 Silica translation

The statement `A.receiveToken(h)` does not need to be translated. However, to provide complete proof, we also have to prove the soundness of the body of `receiveToken`. Due to its grammar, Silica does not allow us to write expressions such as `this.tokens.push(g)`. We could achieve the same effect with the expression

```
    let x : List<NFT@Owned>@Owned = this.tokens
in let em0 : unit = x.push(g)
in let em1 : unit = this.tokens := x
in pack
```

However, in this expression, the invocation of `push` is not allowed, since this transaction is public and the typestate of `this.tokens` is not consistent with the one declared (`List<NFT@Owned>@Owned`) when `push` is called on `x`.

For this reason, I adapted the implementation of `Party` (Listing 4.13) in order to obtain a similar effect (Listing 5.1): in particular, I considered the management for a single NFT. Then, `Party` has two possible states depending on whether it owns the token (`Token`) or not (`NoToken`). These two states are sufficient when considering a single token. In general, if n is the number of tokens that contracts would require, then 2^n states would be needed. In this example, we just want to show how a single token could be sent to a party.

Then, the body of the new version of `receiveToken` is represented by the Obsidian statement `->Token(token = g)`, which can be easily translated in the Silica expression `this` $\rightarrow_P$ `Token(g)` where $P \in \{Owned, Shared\}$.

Listing 5.1: `Party` adaptation for token *send*-statement

```
main asset contract Party {
    state NoToken;
    state Token {
        NFT@Owned token;
    }
    ...

    transaction receiveToken(Party@NoToken >> Token this, NFT@Owned >>
        Unowned g) {
        ->Token(token = g);
    }
}
```

5.4.2 Derivation trees

As mentioned, we have to prove soundness for both the invocation of `receiveToken` and its body. The first uses the rule for invocations (T-INV) which checks if the types of receiver object and the actual parameters of the transaction are subtypes of the ones in the declarations. The second, instead, relies on the rules for the state change (T-$\rightarrow_P$), subtyping for the arguments passed in initialisation, lookup and type splitting.

Statement

The invocation of `receiveToken` on a party `A` (Figure 5.5) requires mainly checking the types of receiver object (π_3) and of the actual parameters (π_4). This check is done by exploiting the subtype and subpermission relations.

$$\frac{\pi_1 \quad \pi_2 \quad \pi_3 \quad \pi_4 \quad \pi_5 \quad \pi_6 \quad \pi_7}{\Gamma;\Delta,\ \texttt{A} : \texttt{Party@NoToken},\ \texttt{h} : \texttt{NFT@Owned} \vdash_s \texttt{A.receiveToken(h)} : \mathit{unit} \dashv \Delta,\ \texttt{A} : \texttt{Party@Token},\ \texttt{h} : \texttt{NFT@Unowned}}\ \text{T-INV}$$

$$\frac{\dfrac{}{\Gamma \vdash \mathsf{NoToken} <:_* \mathsf{NoToken}}\ <:_*\text{-REFL}}{\pi_3} \qquad \frac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{Owned} <:_* \mathsf{Owned}}\ <:_*\text{-REFL}}{\Gamma \vdash \texttt{NFT@Owned} <:_* \texttt{NFT@Owned}}\ <:\text{-MATCHING-DEFS}}{\pi_4}$$

Figure 5.5: Proof for `receiveToken` invocation

Transaction `receiveToken`

The proof for the transaction is more complicated (Figure 5.6): the first rule used is PUBLICTRANSACTIONOK which just requires finding the correct type parameters and proving that the expression `this` $\rightarrow_P$ `Token(g)` is well-typed in the proper context. This fact is proven considering the proper contexts gathered from the transaction definition (π_1): in the input context, the object `this` and the parameter `g` have typestates `Party@NoToken` and `NFT@Owned`; in the output context, the parameter `g` loses ownership (that is, must have type `NFT@Unowned`). To achieve this effect, we must verify that the input type of the actual parameter is consistent with the type of field that must be initialised (π_3). Then, we have to check if the expression `g` is well-typed for the same type (π_6). When evaluating the type of `g`, the lookup is triggered and the type system infers through type splitting that ownership of `g` is transferred to a new owner.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \texttt{Owned} <:_* \texttt{Owned}}\,<:_*\text{-REFL}}{\vdash \texttt{NFT@Owned} <: stateFields(\texttt{Party}, \texttt{Token}) = \texttt{NFT@Owned}}\,<:\text{-MatchingDefs}}{\pi_3}\quad \dfrac{\dfrac{\texttt{NFT} = contract(\texttt{NFT@Owned})}{\texttt{NFT@Owned} \Rrightarrow \texttt{NFT@Owned}/\texttt{NFT@Unowned}}\,\text{Split-Unowned}}{\texttt{this} : \texttt{Party@NoToken},\ \texttt{g} : \texttt{NFT@Owned} \vdash_{\texttt{this}} \texttt{g} : \texttt{NFT@Owned} \dashv \texttt{g} : \texttt{NFT@Unowned}}\,\text{T-Lookup}}{\pi_6}}{\texttt{this} : \texttt{Party@NoToken},\ \texttt{g} : \texttt{NFT@Owned} \vdash_{\texttt{this}} \texttt{this} \to_p \texttt{Token(g)} : \texttt{unit} \dashv \texttt{this} : \texttt{Party@Token},\ \texttt{g} : \texttt{NFT@Unowned}}\,\text{T-}\to_p}{\pi_1}$$

$$\dfrac{\pi_1 \quad params(C) = \varnothing \quad \Gamma = \varnothing}{\texttt{unit receiveToken(NFT@Owned»Unowned g) NoToken»Token } \{\ \texttt{return this} \to_p \texttt{Token(g)}\ \}\ \textbf{ok in Party}}\,\text{PublicTransactionOK}$$

Figure 5.6: Proof for `receiveToken` body

5.5 Sending currency to a variable

The Stipula *send*-statement between currency variables is similar to the one used for tokens. However, in this case, we can send values different from those stored in the input variable. This statement is written as $E \multimap$ `h,h'`. The effects on memory are the same as the ones mentioned for tokens:

$$\ell' = \ell[h \mapsto a', h' \mapsto a'']$$

where

$$\begin{aligned} [\![E]\!]^{\mathsf{a}}_{\ell} &= a \\ [\![h - a]\!]^{\mathsf{a}}_{\ell} &= a' \\ [\![h' + a]\!]^{\mathsf{a}}_{\ell} &= a'' \end{aligned}$$

The main differences are determined by the semantics of the arithmetic operators: for example, as shown in subsection 2.2.5, tokens have no multiplication, in contrast to currencies that can be multiplied by a real number. Also, addition and subtraction have different meanings. Then, following such semantics,

$$\begin{aligned} [\![h - a]\!]^{\mathsf{a}}_{\ell} = [\![h]\!]^{\mathsf{a}}_{\ell} - [\![a]\!]^{\mathsf{a}}_{\ell} = [\![h]\!]^{\mathsf{a}}_{\ell} - [\![E]\!]^{\mathsf{a}}_{\ell} = a' \quad & \text{if } [\![h]\!]^{\mathsf{a}}_{\ell}, [\![E]\!]^{\mathsf{a}}_{\ell} \text{ currencies} \\ & \text{and } [\![h]\!]^{\mathsf{a}}_{\ell} \geq [\![E]\!]^{\mathsf{a}}_{\ell} \\ [\![h' + a]\!]^{\mathsf{a}}_{\ell} = [\![h']\!]^{\mathsf{a}}_{\ell} + [\![a]\!]^{\mathsf{a}}_{\ell} = [\![h']\!]^{\mathsf{a}}_{\ell} + [\![E]\!]^{\mathsf{a}}_{\ell} = a'' \quad & \text{if } [\![h']\!]^{\mathsf{a}}_{\ell}, [\![E]\!]^{\mathsf{a}}_{\ell} \text{ currencies} \end{aligned}$$

When the disequality $[\![h]\!]^{\mathsf{a}}_{\ell} \geq [\![E]\!]^{\mathsf{a}}_{\ell}$ is not satisfied, the program is trying to withdraw from `h` too much currency and a runtime error will occur. Since the error depends on the value of E, it cannot be checked easily using types. Also, the semantics requires `h` and E to be currencies, which should be checked through the type system. However, it cannot prevent errors deriving from expressions with different kinds of assets in them. The respective Obsidian statement for `v` $\multimap$ `h,h'` is `h'.merge(h.split(`E`))`.

However, in Obsidian I required E to be an integer: in fact, in Stipula the value of the currency in E is subtracted from `h` and transferred to `h'`. The type requirements over E are needed in Stipula to correctly manage the semantics of the operations. In Obsidian, this precaution is not needed since there is no specific and formal semantics for asset expressions in this language. Anyway, this approach does not allow expressions with tokens and currencies which leads also in Stipula to unchecked errors. As said, `v` $\multimap$ `h,h'` can be translated to Obsidian as `h'.merge(h.split(`E`))`. To prove the typing for this statement, we also need to provide proofs for `split` and `merge`. For simplicity, we can assume in the proofs that the value of E is represented by an integer variable `v`.

5.5.1 Silica translation

In general, the following translations are straightforward writings in A normal form for the statement and the bodies of `split` and `merge` transactions: the only difference is the presence of `pack` instructions needed to arrange contexts to correctly conclude the proofs.

Statement

Since `h'.merge(h.split(`E`))` is a composite expression, the result of the invocation of `split` must be stored in a proper variable that will be used in the subsequent invocation of `merge`:

$$\begin{aligned} &\texttt{let x : Currency@Owned} = \texttt{h.split(v)} \\ &\texttt{in h'.merge(x)} \end{aligned}$$

Transaction `split`

The `split` transaction applies a reversion if the value given in input is greater than the one stored in `this.value`: that is, the transaction reverts if someone is trying to withdraw too much currency. Since Silica does not provide grammar or typing rules for *if*-statement with boolean conditions and `revert` statements, I decided to prove typing only for the part of the transaction after the *if*-statement.

```
transaction split(Currency@Owned this, int v) returns Currency@Owned {
    if (v > value) {
        revert;
    }
    value = value - v;
    Currency result = new Currency(v);
    return result;
}
```

The statements that actually split the currency contain a subtraction which is not part of the Silica grammar. In this case, I opted to consider it a proper Silica expression. Then, the statement `this.value := this.value - v` is a composite expression and must be stored in a proper variable (`diff`). Hence, the translation considered in the

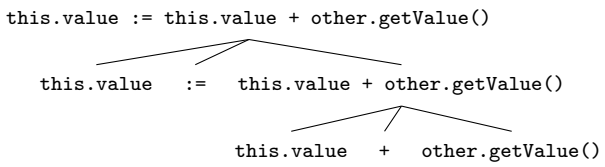


Figure 5.7: Subexpressions of the update in `merge`

proof is the following:

$$
\begin{aligned}
&\text{let } \texttt{diff} : \mathsf{int} = \texttt{this.value - v} \\
&\text{in let } \texttt{em}_0 : \mathsf{unit} = \texttt{this.value := diff} \\
&\text{in let } \texttt{em}_1 : \mathsf{unit} = \texttt{pack} \\
&\text{in let } \texttt{result} : \texttt{Currency@Owned} = \texttt{new Currency@Owned(v)} \\
&\text{in } \texttt{result}
\end{aligned}
$$

Transaction `merge`

The `merge` transaction is far simpler than `split`: there is no reversion and it contains just two lines: in the first one, the method gets the amount of currency stored in `other` and sums it to its own `value` field. Then, the other currency must be disowned to avoid duplication of assets.

```
transaction merge(Currency@Owned this, Currency@Owned >> Unowned other) {
    value = value + other.getValue();
    disown other;
}
```

Even if the transaction is far simpler than `split`, the first line is composed of multiple subexpressions: in particular, we have to consider firstly `other.getValue()`, then we have to sum it to `this.value` and, finally, we can update such field.

Then, following the structure represented in Figure 5.7, we can translate the transaction `merge` starting from `other.getValue()` and storing the amount of the currency contained in `other` in a proper variable `x`. Then, we have to add it to `this.value` and update this field with the obtained result. Then, we can disown the `other` currency. Hence, we can translate the statements in `merge` as follows:

$$
\begin{aligned}
&\text{let } \texttt{x} : \mathsf{int} = \texttt{other.getValue()} \\
&\text{in let } \texttt{sum} : \mathsf{int} = \texttt{this.value + x} \\
&\text{in let } \texttt{em}_0 : \mathsf{unit} = \texttt{this.value := sum} \\
&\text{in let } \texttt{em}_1 : \mathsf{unit} = \texttt{disown other} \\
&\text{in } \texttt{pack}
\end{aligned}
$$

5.5.2 Derivation trees

The proofs of the `h'.merge(h.split(v))`, `split` and `merge` use mainly the (T-LET) rule, which requires type consistency between the expression on the right side and the variable on the left side of the assignment and proceeds to prove the rest of the expression after the keyword "in". The three proofs can be performed in any order. However, I decided to start from the entire statement and, then, proceed with the other two in order of invocation (`split`, `merge`).

Statement

The derivation tree for the Silica expression obtained from `h'.merge(h.split(v))` (Figure 5.8) uses just two kinds of rules: (T-LET), which is used to prove that `x` actually receives from `h.split(v)` a value of type `Currency@Owned`; (T-INV), which is used to check the invocations of `h.split(v)` and `h'.merge(x)`. The proof assumes that `h` and `h'` have type `Currency@Owned` and that `v` is an `int`.

The (T-LET) rule ensures that the type of `h.split(v)` is `Currency@Owned`, that `x` has the same type and that this variable will lose the ownership after the invocation of `h.merge(x)`. In π_1, the type system ensures that `split` is called with the correct input and output types which are computed by $funcArg$. The same checks are done for `merge` in π_2. We can see that, in this case, `x` loses its ownership after the execution of `merge` (as expected).

$$\dfrac{\pi_1 \qquad \pi_2 \qquad \vdash disposable(\texttt{Currency@Unowned})}{\Gamma;\Delta,\ \texttt{h : Currency@Owned},\ \texttt{h' : Currency@Owned},\ \texttt{v : int} \ \vdash_s\ \texttt{let x : Currency@Owned = h.split(v) in h'.merge(x)}\ :\ \mathsf{unit} \dashv \Delta,\ \texttt{h : Currency@Owned},\ \texttt{h' : Currency@Owned},\ \texttt{v : int}}\ \text{T-LET}$$

$$\dfrac{\dfrac{}{\vdash \mathsf{Owned} <:_* \mathsf{Owned}}\ <:_*\text{-REFL} \qquad \dfrac{}{\mathsf{int} <: \mathsf{int}} \qquad \vdash bound(\texttt{Currency@Owned}) = \texttt{Currency@Owned} \qquad \forall f.s.f \notin \Delta_1 \qquad specializeTrans_\Gamma(\texttt{split}, \texttt{Currency}) = \texttt{Currency@Owned split(int v) Owned»Owned \{ ... \}} \qquad \texttt{Currency@Owned} = funcArg(\texttt{Currency@Owned}, \texttt{Currency@Owned}, \texttt{Currency@Owned}) \qquad \mathsf{int} = funcArg(\mathsf{int}, \mathsf{int}, \mathsf{int})}{\Delta_1 = \Gamma;\Delta,\ \texttt{h : Currency@Owned},\ \texttt{h' : Currency@Owned},\ \texttt{v : int} \ \vdash_s\ \texttt{h.split(v)} : \texttt{Currency@Owned} \dashv \Gamma;\Delta,\ \texttt{h : Currency@Owned},\ \texttt{h' : Currency@Owned},\ \texttt{v : int}}\ \text{T-INV}$$

$$\dfrac{\dfrac{}{\vdash \mathsf{Owned} <:_* \mathsf{Owned}}\ <:_*\text{-REFL} \qquad \dfrac{\dfrac{}{\vdash \mathsf{Owned} <:_* \mathsf{Owned}}\ <:_*\text{-REFL}}{\vdash \texttt{Currency@Owned} <: \texttt{Currency@Owned}}\ <:_*\text{-MATCHING-DEFS} \qquad \vdash bound(\texttt{Currency@Owned}) = \texttt{Currency@Owned} \qquad \forall f.s.f \notin \Delta_2 \qquad specializeTrans_\Gamma(\texttt{merge}, \texttt{Currency}) = \texttt{unit merge(Currency@Owned»Unowned other) Owned»Owned \{ ... \}} \qquad \texttt{Currency@Owned} = funcArg(\texttt{Currency@Owned}, \texttt{Currency@Owned}, \texttt{Currency@Owned}) \qquad \texttt{Currency@Unowned} = funcArg(\texttt{Currency@Owned}, \texttt{Currency@Owned}, \texttt{Currency@Unowned})}{\Delta_2 = \Gamma;\Delta,\ \texttt{h : Currency@Owned},\ \texttt{h' : Currency@Owned},\ \texttt{v : int},\ \texttt{x : Currency@Owned} \ \vdash_s\ \texttt{h'.merge(x)} : \mathsf{unit} \dashv \Gamma;\Delta,\ \texttt{h : Currency@Owned},\ \texttt{h' : Currency@Owned},\ \texttt{v : int},\ \texttt{x : Currency@Unowned}}\ \text{T-INV}$$

Figure 5.8: Proof for currency *send*-statement `h'.merge(h.split(v))`

Transaction `split`

The proof for `split` transaction (Figure 5.9) is mainly composed of a relatively long chain of (T-LET) rules that originates from the first one which is (PUBLICTRANSACTIONOK). In particular, this rule set up the contexts for the Silica expression e that corresponds to the translation mentioned above. For the sake of readability, I

abbreviated expressions using names as e, e_0, e_1, and e_2:

e =let `diff` : `int` = `this.value - v` in e_0
e_0 =let $\texttt{em}_0$: `unit` = `this.value := diff` in e_1
e_1 =let $\texttt{em}_1$: `unit` = `pack` in e_2
e_2 =let `result` : `Currency@Owned` = `new Currency@Owned(v)` in `result`

The proof starts considering the transaction

`Currency@Owned split(int v) Owned»Owned {return` e`}`

Then, it firstly checks that the expression `this.value - v` and `diff` are both of type `int` as `this.value` (π_1, π_5). Hence, the packing of `this` is executed: we can easily notice that in π_7 the term `this.value` : `int` in the input context disappears in the output context. Then, the instantiation of a new currency object and its assignment to the `result` is proven in π_{11}. In particular, to prove the instantiation we need the rule (T-NEW), which requires the actual parameter `v` of the constructor to be a subtype of `int`. Finally, the ownership of currency in `result` must be given to the variable that will store the result of the `split` and this fact is proven in π_{13}.

Transaction `merge`

Like `split`, the proof for the transaction `merge` (Figure 5.10) is a chain of (T-LET) rules that ends with the proof of the disowning of the currency merged to `this` and the packing of `this`. As for `split`, I decided to give names to subexpressions to improve the readability of the proof:

e =let `x` : `int` = `other.getValue()` in e_0
e_0 =let `sum` : `int` = `this.value + x` in e_1
e_1 =let $\texttt{em}_0$: `unit` = `this.value := sum` in e_2
e_2 =let $\texttt{em}_1$: `unit` = `disown other` in `pack`

As for `split`, the proof sets up the proper contexts with (PUBLICTRANSACTIONOK) and manages the assignments for the several variables introduced with "let". Then, the proof for the `disown` expression (π_{12}) requires `other` to be `Owned` and apply type splitting to take away from the variable the ownership.

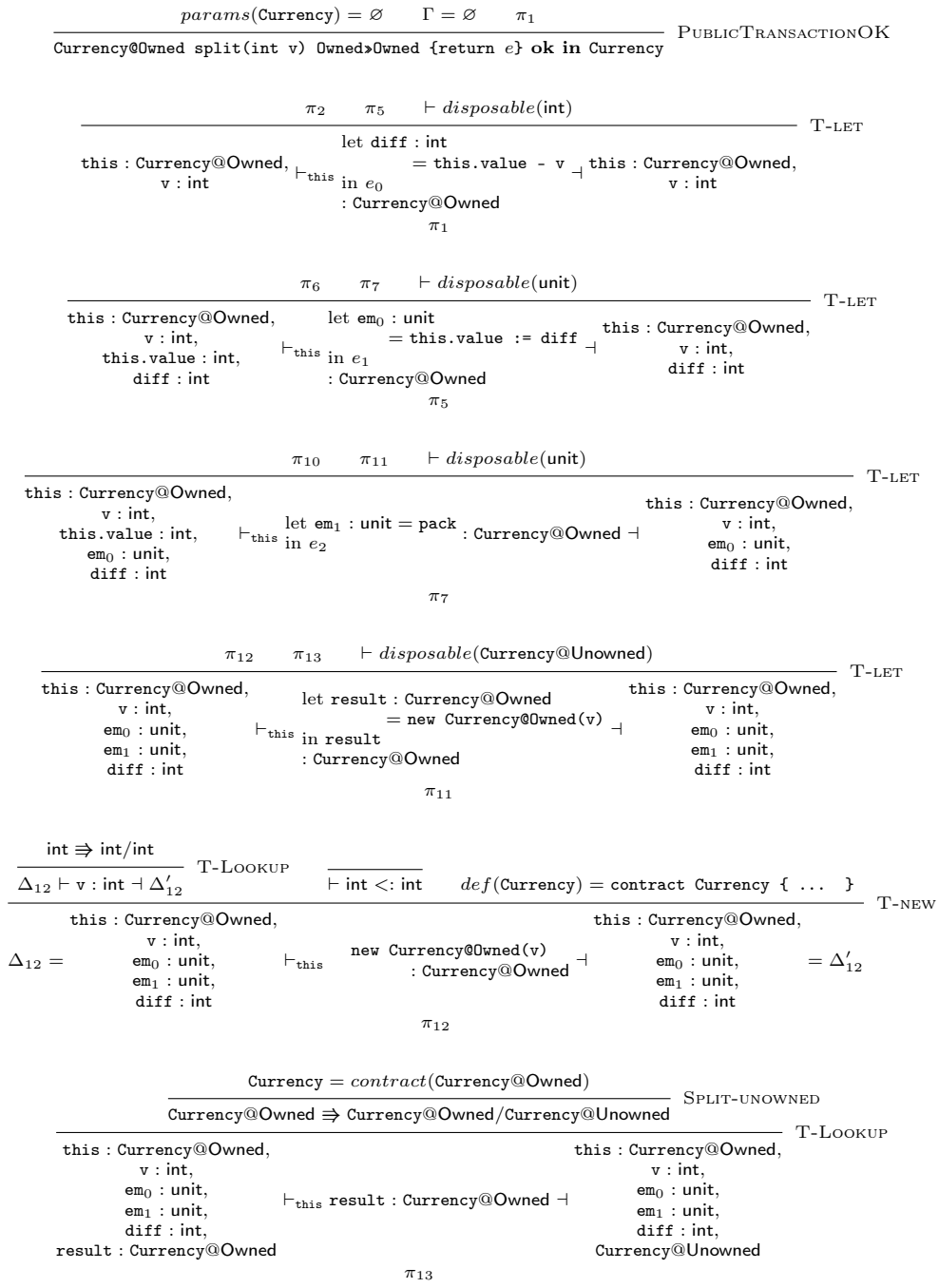


Figure 5.9: Proof for `split` transaction

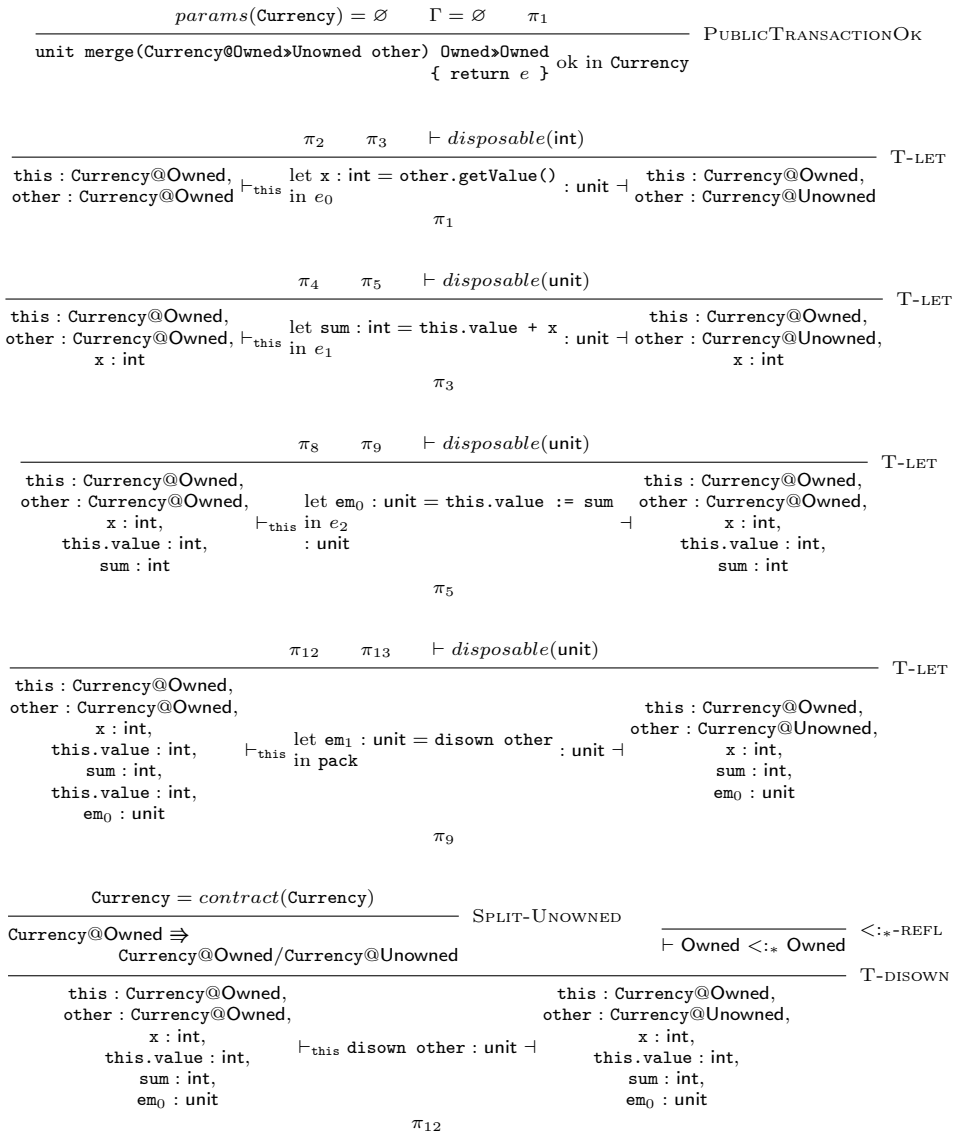


Figure 5.10: Proof for `merge` transaction

5.6 Sending currency to a party

Consider the statement that sends currency from a variable to a party: $E \multimap$ `h,A`. Similarly to what we already noted for `h` $\multimap$ `A`, after the statement the memory ℓ will be modified in this way:

$$\ell' = \ell[h \mapsto a']$$

where

$$\begin{aligned} [\![E]\!]^{\mathtt{a}}_{\ell} &= a \\ [\![h - a]\!]^{\mathtt{a}}_{\ell} &= a' \\ \ell(\mathtt{A}) &= A \end{aligned}$$

Then, following the semantics shown in subsection 2.2.5,

$$[\![h - a]\!]^{\mathtt{a}}_{\ell} = [\![h]\!]^{\mathtt{a}}_{\ell} - [\![a]\!]^{\mathtt{a}}_{\ell} = [\![h]\!]^{\mathtt{a}}_{\ell} - [\![E]\!]^{\mathtt{a}}_{\ell} = a' \quad \begin{array}{l}\text{if } [\![h]\!]^{\mathtt{a}}_{\ell}, [\![E]\!]^{\mathtt{a}}_{\ell} \text{ currencies} \\ \text{and } [\![h]\!]^{\mathtt{a}}_{\ell} \geq [\![E]\!]^{\mathtt{a}}_{\ell}\end{array}$$

Considering the semantics above, we can do the same remarks as for $E \multimap$ `h,h'`:

- $[\![h]\!]^{\mathtt{a}}_{\ell} \geq [\![E]\!]^{\mathtt{a}}_{\ell}$ generates a runtime error that can be checked only runtime;
- E in Obsidian must be an integer, but in Stipula it is a currency: the Obsidian approach avoids expressions with both tokens and assets, which leads to unchecked runtime errors in Stipula.

The statement $E \multimap$ `h,A` corresponds to `A.receive(h.split(`E`))` in Obsidian. Also, in this case, we need the proofs for `split` and `merge` already provided in Figure 5.9 and Figure 5.10. In particular, the second transaction is called in the body of `receive`, which will also be proven.

5.6.1 Silica translation

The translation for the whole statement is straightforward and the process is similar to the one described for `h'.merge(h.split(v))`. Then, the translation for `A.receive(h.split(v))` is

```
    let x : Currency@Owned = h.split(v)
in A.receive(x)
```

The translation for the method body of `receive` is troubling: as discussed in subsection 5.4.1, Silica's grammar does not allow to call transactions having fields as receivers. In particular, the statement `wallet.merge(gift)` in `receive`, which is equivalent to `this.wallet.merge(gift)`, cannot be translated in Silica without causing type inconsistencies at the moment of the invocation of `merge`. Then, I decided to relax the grammar of Silica to consider expressions like `this.wallet` similar to usual variables (e.g. `x`).

```
transaction receive(Currency@Owned >> Unowned gift) {
    wallet.merge(gift);
}
```

Then, we could think that the translation is not needed. However, the mere invocation of `merge` is not allowed in Silica: in fact, we need to unpack `this` to use its

fields. To achieve this requirement, we need to assign `this.wallet` to a new variable `x` and, then, assign the same `x` to `this.wallet`: in this way, we will unpack `this.wallet` and it will have type `Currency@Owned` thanks to the second assignment. This is required since in Obsidian a public transaction as `merge` can be called only if the fields are consistent with the type declared in the contract. Hence, the body of `receive` can be translated as follows:

```
    let x : Currency@Owned = this.wallet
in let em1 : unit = this.wallet := x
in let em2 : unit = this.wallet.merge(g)
in pack
```

5.6.2 Derivation trees

The proofs given are based on (T-LET) and (T-INV) rules. However, due to the grammar relaxations mentioned above, the proof for `receive` body cannot be completed with the (T-INV) rule provided in the Obsidian article [4]. As before, I proved the entire statement and, then, the other proofs. The ones required for `split` and `merge` are omitted since they are the same as seen previously.

Statement

The proof for `A.receive(h.split(v))` (Figure 5.11) is very similar to the one needed for `h'.merge(h.split(v))`: it uses the same rules in the same order with similar intents. The main difference can be found in the initial assumptions for obvious reasons: this time we need to assume that `A` has typestate `Party@Owned` and we don't need the variable `h`.

Transaction `receive`

The proof for the body of the `receive` transaction (Figure 5.12) starts with the rule (PUBLICTRANSACTIONOK) and, then, proceeds with several (T-LET) rules that manage the unpacking of `this.wallet`. Once that `this.wallet : Currency@Owned` is in the input context, then we can apply the T-INV rule to prove well-typedness of the invocation of `merge` on `this.wallet`. However, this rule requires as hypothesis that $\forall f, \texttt{this}.f \notin \Delta_6$: that is, it requires `this` to be packed. But, at the same time, (T-INV) requires `this.wallet` to be in the input context to check the invocation. For this reason, I replaced

$$\forall f, \texttt{this}.f \notin \Delta$$

with the following three hypothesis (the $\overline{\text{overline}}$ indicates sequences):

$$unionFields(D@T_{ST}) = \overline{T_{f,decl}\ f} \tag{5.1}$$

$$fieldStates_s(\Delta; \overline{T_{f,decl}\ f}) = \overline{T_f} \tag{5.2}$$

$$\overline{\Gamma \vdash T_f <: T_{f,decl}} \tag{5.3}$$

where

- s is the name of the receiver object (e.g. `this`);
- D is the name of the contract of the object s (e.g. `Currency`);

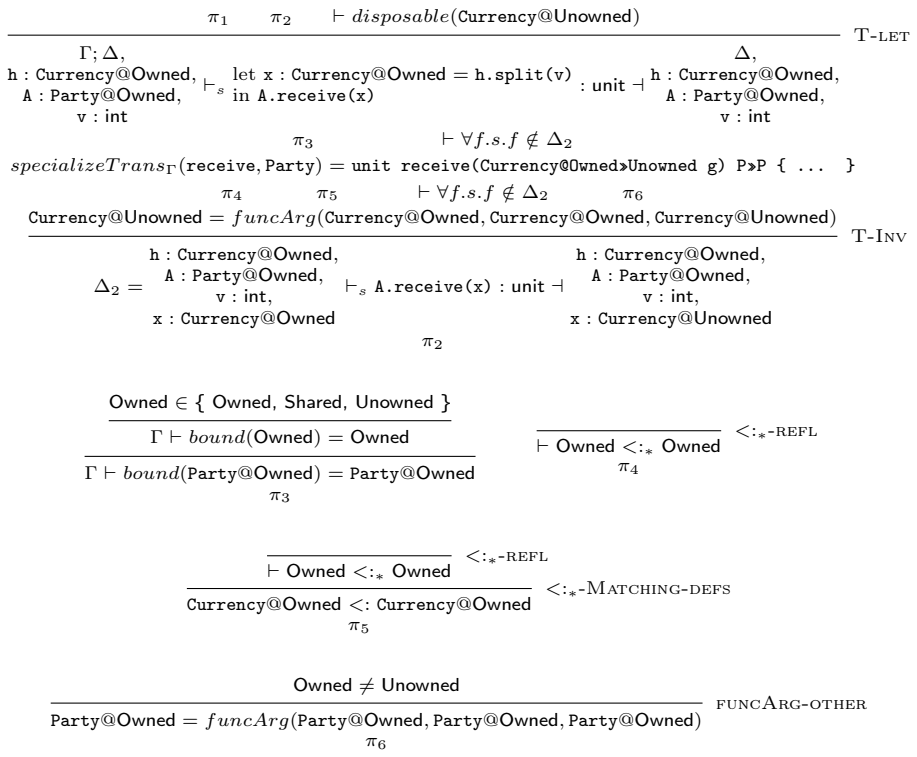


Figure 5.11: Proof for currency *send*-statement `A.receive(h.split(v))`

- $unionFields$ returns the fields f of D with their type declarations $T_{f,decl}$;
- $fieldStates_s$ returns actual typestates of fields in $\overline{T_{f,decl}\ f}$ and in context Δ.

Altogether, these hypotheses ensure that the fields in the input context are consistent with their declarations and, consequently, allow invocations of public transactions as `merge`. In particular, (5.1) and (5.2) provide the types and (5.3) checks the consistency between the actual types of fields and the respective declared types. Then, applying the correct substitutions in these hypotheses, in π_6 of the proof we obtain the hypothesis colored in blue. As for `split` and `merge`, I decided to give names to subexpressions to improve the readability of the proof:

$$
\begin{aligned}
e \ \ &= \text{let } \texttt{x} : \texttt{Currency@Owned} = \texttt{this.wallet} \text{ in } e_0 \\
e_0 &= \text{let } \texttt{em}_1 : \texttt{unit} = \texttt{this.wallet := x} \text{ in } e_1 \\
e_1 &= \text{let } \texttt{em}_2 : \texttt{unit} = \texttt{this.wallet.merge(g)} \text{ in } \qquad\qquad \texttt{pack}
\end{aligned}
$$

5.7 Conclusions

The proofs in this chapter show that the Obsidian implementation of the Stipula *send*-statement is well-typed in the Obsidian type system. This fact (and how the Obsidian type system is designed) give us a good insight into how the insertion of ownership in Stipula could be done and what benefits could bring to the language.

As already discussed, the provided Obsidian implementation avoids expressions with assets: this is usually a source of unchecked errors in Stipula, especially when tokens and currencies both appear in the same expression. The first step to avoid this mixture of kinds of assets is to distinguish them using different types and to forbid operations between those types. If we also consider that the draining of token variables could be managed with ownership and checked statically, expressions between tokens become almost meaningless. About currencies, a good idea may be to forbid operations between them, then to allow casting to `real` and to use only numeric expressions. This approach is similar to the one I proposed with Obsidian: `Currency` cannot be used in arithmetic expressions, but can be "cast" using the transaction `getValue` and `split` allows only `int` values. This approach, as shown, enables a compile-time detection of errors for *send*-statement among token variables and avoids runtime errors in expressions when different kinds of assets are involved.

In addition, this approach could potentially allow the introduction in Stipula of the main Obsidian property, *Asset Retention*. This would enhance the safety of Stipula programs, especially in addition to the liquidity analyser. In this way, Stipula programs would be granted to not freeze assets in the contract itself and to not lose them when introduced or transferred elsewhere.

$$\frac{params(\texttt{Party}) = \varnothing \qquad \Gamma = \varnothing \qquad \pi_1}{\texttt{unit receive(Currency@Owned>Unowned g) Owned>Owned \{ return } e \texttt{ \}}}\ \textsc{PublicTransactionOk}$$

$$\frac{\pi_2 \qquad \pi_3 \qquad \vdash disposable(\texttt{Currency@Unowned})}{\texttt{this : Party@Owned, g : Currency@Owned} \vdash_{\texttt{this}} \texttt{let x : Currency@Owned = this.wallet in } e_0 \texttt{ : unit} \dashv \texttt{this : Party@Owned, g : Currency@Unowned}}\ \textsc{T-let}$$

π_1

$$\frac{\pi_4 \qquad \pi_5 \qquad \vdash disposable(\texttt{unit})}{\texttt{this : Party@Owned, g : Currency@Owned, this.wallet : Currency@Unowned, x : Currency@Owned} \vdash_{\texttt{this}} \texttt{let em}_1 \texttt{ : unit = this.wallet := x in } e_1 \texttt{ : unit} \dashv \texttt{this : Party@Owned, g : Currency@Unowned, x : Currency@Unowned}}\ \textsc{T-let}$$

π_3

$$\frac{\pi_6 \qquad \pi_7 \qquad \vdash disposable(\texttt{unit})}{\texttt{this : Party@Owned, g : Currency@Owned, this.wallet : Currency@Owned, x : Currency@Unowned, em}_1 \texttt{ : unit} \vdash_{\texttt{this}} \texttt{let em}_2 \texttt{ : unit = this.wallet.merge(g) in pack} \dashv \texttt{this : Party@Owned, g : Currency@Unowned, x : Currency@Unowned, em}_1 \texttt{ : unit}}\ \textsc{T-let}$$

π_5

$$\frac{\begin{array}{c}\vdash bound(\texttt{Currency@Owned}) = \texttt{Currency@Owned} \\ specializeTrans_{\Gamma}(\texttt{merge}, \texttt{Currency@Owned}) = \\ = \texttt{unit merge(Currency@Owned>Unowned g) Owned>Owned \{ ... \}} \\ \vdash \texttt{Owned} <:_* \texttt{Owned} \qquad \vdash \texttt{Currency@Owned} <: \texttt{Currency@Owned} \\ unionFields(\texttt{Party@Owned}) = \\ = \texttt{Currency@Owned this.wallet, List<NFT@Owned>@Owned this.tokens} = \\ = \overline{T_{f,decl}\ f} \\ fieldStates_{\texttt{this}}(\Delta_6; \overline{T_{f,decl}\ f}) = \texttt{Currency@Owned} \\ \Gamma \vdash \texttt{Currency@Owned} <: \texttt{Currency@Owned} \\ funcArg(\texttt{Currency@Owned, Currency@Owned, Currency@Owned}) = \texttt{Currency@Owned} \\ funcArg(\texttt{Currency@Owned, Currency@Owned, Currency@Unowned}) = \texttt{Currency@Unowned}\end{array}}{\Delta_6 = \texttt{this : Party@Owned, g : Currency@Owned, this.wallet : Currency@Owned, x : Currency@Unowned, em}_1 \texttt{ : unit} \vdash_{\texttt{this}} \texttt{this.wallet.merge(g) : unit} \dashv \texttt{this : Party@Owned, g : Currency@Unowned, this.wallet : Currency@Owned, x : Currency@Unowned, em}_1 \texttt{ : unit}}\ \textsc{T-Inv}$$

π_6

Figure 5.12: Proof for `receive` transaction

Chapter 6

Conclusions

In conclusion, this thesis argued the differences between two typestate-oriented languages: Stipula and Obsidian. The main concept that they share is the centrality of contracts. Both languages are designed to work in blockchains and provide tools to manage assets. However, Obsidian and Stipula are not the only languages or tools available, but other ways to implement the TSOP approach. Here, we are going to list and briefly compare some of them to Stipula and Obsidian, focusing also on the solutions suggested in this work.

6.1 Related works

In general, TSOP can be enforced in programming languages in two different ways: by using external tools for already existing programming languages or by designing and developing a full-fledged language that supports natively typestate programming.

6.1.1 Mungo

A well-known external tool is **Mungo** [16], which extends Java with typestate definitions. Every typestate defines a protocol in the form of a state machine: that is, for each state, the programmer has to specify a subset of the methods defined in the class and, for each method, which states it reaches. With Mungo, any Java class can be enriched using the annotation `@Typestate("ProtocolName")` (Listing 6.2) where `ProtocolName` names the file that contains the definition of the typestate (Listing 6.1). In the listings provided, we are considering a Java class that models a stack and the respective protocol. As expected, a stack should provide functions to push or pop elements and a way to check if the stack is empty or not. For this operations, it may be reasonable to establish a protocol with three states: `Empty`, `NonEmpty` or `Unknown`. For example, in the first state, the two methods that should be available to be called are `push` and `deallocate`: they will change the state of the stack respectively to `NonEmpty` and `end` (which is a Mungo keyword used for the *end*-state). When the stack is non-empty, we could still push other elements or pop the one in the head. However, after a `pop` we cannot know if the stack is empty or not, that is the state of the stack is `Unknown`. This lack of knowledge can be solved only with the method `isEmpty`, which will take the stack to a known state.

The definition of typestates is used by a type inference algorithm that considers the sequences of methods called on objects and checks if the inferred typestate is a subtype

Listing 6.1: Example of Mungo protocol

```
typestate StackProtocol {
    Empty = { void push(int): NonEmpty,
              void deallocate (): end }
    NonEmpty = { void push(int): NonEmpty,
                 int pop (): Unknown }
    Unknown = { void push(int): NonEmpty ,
                Check isEmpty (): <EMPTY: Empty, NONEMPTY: NonEmpty >}
}
```

Listing 6.2: Example of Java class enriched with Mungo Protocol

```
@Typestate ("StackProtocol")
class Stack {
    private int[] stack;
    private int head;

    Stack() { stack = new int[MAX]; head = 0; }
    void push(int d) { stack[head++] = d; }
    int pop () { return stack[head--]; }
    Check isEmpty () {
        if(head == 0) return Check.EMPTY;
        return Check.NONEMPTY;
    }
    void deallocate () {}
}
```

of the one declared for the object. With Mungo, any typestate is considered a linear type: that is, aliasing is not allowed since it may lead to typestate inconsistencies and any attempt to create aliases is reported as an error at compile-time.

Mungo comes along with **StMungo** (Scribble-to-Mungo) which uses this typestate feature to implement specifications written in Scribble protocol language using Mungo and Java.

6.1.2 Java Typestate Checker

Java Typestate Checker (JATYC) [20, 19] is a new implementation of Mungo that also prevents null pointer errors, enables state transitions that depend on return values, ensures protocol completion (that is, that objects reach the *end*-state), embeds droppable states and permits enriching Java library or third-party libraries with protocols. The tool also uses behavioural types and access permissions to provide the ability to define aliases and share references in a controlled way. Aliases and permission transfers between them are key features that allow sharing of objects among threads with the certainty that their use follows the given specification.

6.1.3 Generational approach

A ***generational* approach** was suggested by Gerbo and Padovani [15]: in this case, Objective Join Calculus [13, 8] and automata theory are used to generate code for concurrent typestate-oriented programs. In particular, the first is a formal model for concurrent TSOP that supports *join patterns*: these entities are used to associate methods with states and synchronise them, which are both considered as messages that objects can receive.

The presented tool focuses on Java, but the approach is easily portable to other mainstream languages. With this *generational* approach, the programmer writes a

Listing 6.3: Generational approach example

```
@Protocol("*get·(EMPTY·put + FULL)")
class Future<A> {
    @State private void EMPTY();
    @State private void FULL(A x);
    @Operation public A get();
    @Operation public void put(A x);
    @Reaction private void when_EMPTY_put(A x) { this.FULL(x); }
    @Reaction private A when_FULL_get(A x) { this.FULL(x); return x; }
    public Future() { this.EMPTY(); }
}
```

Java class and specifies through standard Java annotations the *join patterns* needed to synchronise methods and states. The specified pattern is a representation of the possible traces of the object. Using this pattern, the tool builds a *matching automaton* which is used to generate boilerplate code. This code already contains the synchronisation logic and manages the *automaton* using `switch` statements. However, contrary to other approaches, protocol violations are detected at run-time when an illegal state is reached.

Consider the example in Listing 6.3: in this case, *generational approach* is used to implement a *future* (called also *promise* in other languages). A `Future` can receive four different messages: state messages (`@State`) `EMPTY` and `FULL`; method messages (`@Operation`) `put` and `get`. We also know how a `Future` should react in certain states to certain methods: in fact, not completed (i.e. empty) *futures* react only to `put` messages which complete them; instead, completed (i.e. full) ones react only to `get` messages which request the object stored in the *future* variable. Then, we can write these reactions and mark them with the annotation `@Reaction`: the tool provided by Gerbo and Padovani requires that the reactions are named as `when_`*`STATE_OPERATION`*. Hence, the two reactions discussed are named as shown in Listing 6.3.

The only cases where the protocol is violated are:

- when we try to put something new in an already complete future;
- when we signal multiple times that `Future` is full since, once the future is completed, its state cannot change.

Then, the *join pattern* for `Future` must avoid traces where `FULL` is received before any `put` or multiple `FULL` messages. Follows that the `Future` protocol can be described with the pattern

$$*\texttt{get} \cdot (\texttt{EMPTY} \cdot \texttt{put} + \texttt{FULL})$$

where $*$ indicates any sequences of the messages (possibly empty), $\cdot$ the *shuffle* (or interleaving) of two messages and $+$ the choice among two messages. This pattern must be annotated with the declaration of the class `Future` (`@Protocol`).

At this point, the tool can be used and will directly generate code that checks runtime if the protocol of the object is followed. If an illegal state is reached, the Java code will throw an `IllegalStateException`.

6.1.4 Flint

Unlike the previous approaches, **Flint** [22, 23, 10, 11] is a statically-typed programming language designed to write robust smart contracts. Flint is designed to work on the

Listing 6.4: Flint protection blocks

```
1  contract Auction (Preparing, InProgress, Terminated) {
2      let bidders : [Address]
3      ...
4  }
5
6  Auction :: (any) {
7      // Protection block without any restriction
8      public init(b : [Address], ...) { // Constructor
9          bidders = b;
10         ...
11     }
12 }
13
14 Auction@InProgress :: (bidders) {
15     // Protection block where the only users authorised are
16     // the users with address in variable bidders when
17     // the auction is in progress
18     @payable
19     public func offer(implicit bid : Wei) { ... }
20 }
```

Ethereum Virtual Machine (EVM) and with Ether, the Ethereum cryptocurrency. Flint focuses on the following main aspects:

- **Protection against unauthorised function calls:** in smart contracts, some operations may be sensitive enough to be forbidden to some users. However, most languages do not require programmers to necessarily provide the authorisation information: for this reason, developers may forget to check the identity of the caller when needed. To avoid this possibility, Flint requires its users to explicitly write who is authorised to call each function using *protection blocks*. An invocation may also occur when the state is not consistent with the preconditions of the function called. This is managed in Flint (starting from the second version of the language) using protection blocks also for typestates. For example in Listing 6.4, we can see the definition of a contract `Auction`. After the declarations (lines 1-4), we can see two protection blocks: the first one (lines 6-12) is used for the definition of the constructor and no restriction is placed since it is not specified a state for the block and the caller is marked as `any`, a Flint keyword which matches with any adddress; the second protection block (lines 14-20) applies protections for states and for callers, since the state requested is `InProgress` and the set of user addresses that can call the function `offer` is limited to the `bidders` in the declaration.

- **Safe operations for assets and Ether:** another main feature of Flint is the provision of safe operations to handle assets. Transfer operations are performed atomically and ensure that the state of the contract is always consistent. In particular, assets in Flint cannot be accidentally created, duplicated or destroyed. Wei, the smallest denomination of Ether, in Flint is implemented as an Asset with a dedicated type: which means that it cannot incur accidental conversions, overflows or losses. As shown in Listing 6.5 and in Listing 6.6, Wei can be implemented in Flint as an Asset, which is defined as a trait with specific properties and interface.

- **Interoperability with Solidity** Since Flints works with the EVM, a large collection of smart contracts are already available in Solidity, so interoperability

Listing 6.5: Flint implementation of Wei

```
struct Wei : Asset {
    var rawValue: Int = 0

    // Creates Wei directly from an integer. This is a privileged operation.
    init(unsafeRawValue: Int) { self.rawValue = unsafeRawValue }

    // Creates Wei by transferring a specific quantity of another Wei.
    // Causes a fatalError if the quantity of source is smaller than amount.
    init(source: inout Wei, amount: Int) {
        if source.getRawValue() < amount { fatalError() }
        source.rawValue -= amount
        rawValue = amount
    }

    // Creates Wei by transferring the entire quantity of another Wei.
    init(source: inout Wei) { init(&source, source.getRawValue()) }

    // Returns the quantity of Wei, as an integer.
    func getRawValue() -> Int { return rawValue }
}
```

Listing 6.6: Flint definition and implementation of Asset

```
trait Asset {
    // Create the asset by transferring a given amount of asset's contents.
    init(source: inout Self, amount: Int)

    // Unsafely create the Asset using the given raw value.
    init(unsafeValue: Int)

    // Return the raw value held by the receiver.
    func getRawValue() -> Int

    // Transfer a given amount from source into the receiver.
    mutating func transfer(source: inout Self, amount: Int)
}

// Default implementation of Assets functions
extension Asset {
    // Create the asset by transferring another asset's contents.
    init(from other: inout Self) {
        self.init(from: &other, amount: other.getRawValue())
    }

    // Transfer the value held by another Asset of the same concrete type.
    mutating func transfer(source: inout Self) {
        transfer(from: &source, amount: source.getRawValue())
    }

    // Transfer a subset of another Asset of the same concrete type.
    mutating func transfer(source: inout Self, amount: Int) {
        if amount > source.getRawValue() { fatalError() }
        source.rawValue -= amount
        rawValue += amount
    }
}
```

with this large set of features is surely desirable. For this reason, Flint is provided with an ABI compatible with Solidity. In this way, Solidity contracts can call functions on Flint smart contracts and vice-versa.

Flint is the most similar approach to Stipula and Obsidian, since all of them are languages designed for writing smart contracts that embrace TSOP with the goal to enforce safety and security.

Flint and Stipula share two main features: protection against unauthorised function calls and asset specific operations. The first is implemented through protection blocks in Flint and in Stipula with caller guards. The second is realised in Flint with the `Asset` trait and the specific asset type `Wei` which provide transfer functions between assets and in Stipula with specific *send*-statements.

Consider the function `put` of `CoinEscrow` (Listing 4.6), which is also the only one callable when the contract is in state `S1`

```
@S1 Sender : put()[h] (h==amount) {
    h -o w
} ==> @S2
```

then we could write in Flint the following protection block for a similar `CoinEscrow` smart contract:

```
NftEscrow@S1 :: (sender) {
    @payable
    public func put(implicit h : Wei) {
        if (h == amount) transfer(from: &h)
    }
}
```

The main difference is the place where the precondition is checked: Stipula allows checking the precondition before the execution of the function, contrarily to Flint which must do the same control inside the function. Another difference is that Flint allows using groups of undefined sizes to call functions (consider the protection block in lines 14-20 of Listing 6.4). However, once array types will be inserted into Stipula, this feature could be easy to replicate. Concerning asset-specific operations, we can easily see that the specific operation to send the currency in `h` to `w` can be translated with the help of the `transfer` action which in Flint is granted to be atomic.

Except for checks on states on function calls, the previous Flint features are not present in Obsidian. However, these two languages are similar in other aspects: contrarily to Stipula which provides typing only for primitive types, they are statically typed and do not support inheritance. At the same time, Flint and Obsidian provide mechanisms to define interfaces for structs and contracts, respectively. The first allows users to define traits, similarly to Rust [25], and the second allows to define interfaces. Traits and interfaces can both be used as parameter types. The main difference in typing is that Obsidian always checks states statically, but Flint does not: in fact, Flint performs checks of protection blocks on typestate at compile-time only for internal function calls (that is, in the same contract). parameterThe other strong similarity is the use of *linear* types. However, there is a difference: in Obsidian, misuse of assets would lead to a compilation error, but in Flint the compiler will provide only a warning. In Flint's developers' opinion, due to their inability to detect whether all the local variables are used exactly once, producing errors would worsen the programmer experience.

6.2 Overall Summary

In this work, we compare two programming languages designed for smart contract development, Stipula and Obsidian. In our analysis, we identify the downsides and upsides of each of them. In particular, Stipula adopts a safer and more flexible approach than Obsidian in legal contract writing, due to the primitives available for programmers, such as caller and precondition guards, time-triggered events, agreements and dedicated asset operations. These features enable simpler and more readable implementation of contracts and enforce a safer approach to development, especially thanks to asset operations and guards. On the other hand, Stipula lacks typical functionalities, such as user-defined data and data structures. Also, the language does not support a full-fledged type system that ensures safety properties on asset operations: the Stipula type system ensures typical safety properties for primitive types (e.g. `real`, `bool` and so on). However, Crafa and Laneve designed a *liquidity* analyser for Stipula, which enables a static analysis on assets that detect when they may remain inside the contract indefinitely.

When comparing Stipula and Obsidian with the other approaches discussed in the previous section, these languages definitely diverge from the external tool approach (i.e. Mungo, JATYC and generational) and show several similarities with Flint, especially Stipula: consider the guards on callers and typestate, which are fundamental features for both of these languages. Furthermore, Stipula and Flint chose to manage assets and currencies (Wei in Flint) natively and to provide specific asset operations. However, Flint also allows developers to define their own currencies using the Asset trait in the Standard Library. This approach is more akin to the Obsidian approach, which permits user-defined assets. Flint also uses *linear types* in its code analysis (like Obsidian), but produces only warnings.

The other contribution of this work is the discussion of typing for Stipula *send*-statements. Since we provided an implementation in Obsidian, we could discuss the types involved in these statements. We provided proofs that they are well-typed: the tools used in these demonstrations give us some hints, especially on how ownership should work in Stipula statements. In particular, the key aspects identified are the choice to avoid expressions with different kinds of asset and to disallow arithmetical operations between currencies and between tokens: the idea is that every expression should return an integer value. Then, following the proofs given for Silica, it may be feasible to introduce ownership in this kind of statement and to guarantee a property similar to *Asset Retention*.

This thesis focused on the search for features and practises to improve the expressiveness of TSOP languages. In particular, the choice of Obsidian was justified by its solid formal foundations, which derive from Featherweight Typestate, and the usability remarks reported by Obsidian designers [3]. Obsidian is a statically-typed and typestate-oriented approach with careful attention to safety, security and usability. These features fall within the main goals of Stipula, a newborn typestate-oriented language for legal contracts. Then, their implementation in such language may be a good way to enhance its safety and users' experience.

6.3 Future work

Further studies should investigate the insertion of new features into Stipula. Firstly, a great improvement would be granted by the implementation of array types and,

especially, user-defined data types, which could be challenging to achieve due to typestates. In particular, problems may occur depending on choices about subtyping and inheritance: Featherweight Typestate provides both, Obsidian subtyping but not inheritance, Flint neither of them. In smart contracts, as Coblenz et al. noted [4], inheritance may bring unexpected behaviour due to *fragile base class problem*. In general, subclassing should be avoided in blockchain environments: for example, having a contract C, such that $C <: B <: A <: ...$, that inherits behaviour from at least B and A would lead to unclear and undesired behaviours.

Another work to consider in the future is the implementation of ownership of assets, as mentioned before. To avoid this mixture of kinds of assets, we could differentiate them using diverse types and forbid operations between them. Concerning currencies, we could operate as proposed in our Obsidian implementation: that is, forbidding operations between them, then casting to `real` and using only `real` expressions. As shown, in this way, it may be possible to detect compile-time errors for *send*-statement among token variables and to avoid runtime errors in expressions caused by unsafe asset arithmetic operations.

However, we cannot forget that Stipula does not declare types in contracts and parameter lists. Then, to not change drastically the syntax and the approach thought for Stipula users, we can diversify tokens and currencies in two possible ways:

- **Through specific keywords or syntax:** for example, the keyword `assets` in the declaration could be substituted by two other keywords such as `currencies` and `tokens`. In this case, we should add syntax in the function asset parameter list to differentiate tokens and currencies. We could separate the list of currencies and the list of tokens using "`;`" or using two different pairs of square brackets (`t[]` for tokens, `c[]` for currencies):

```
@S1 A : m(x) [t1,t2 ; c1,c2]
(cond) {
    ...
} == > @S2
```

```
@S1 A : m(x) t[t1,t2] c[c1,c2]
(cond) {
    ...
} == > @S2
```

- **Diversifying *send*-statements:** we could use

`t` $\multimap$ `t'`		(6.1)
`t` $\multimap$ `A`	only for tokens	(6.2)
`v` $\multimap$ `c,c'`		(6.3)
`v` $\multimap$ `c,A`	only for currencies	(6.4)

  and let the type-inference system entail the correct types for the variables: in fact, since (6.1), (6.2) are used only for tokens, we can easily infer that `t` and `t'` are tokens. The same holds for (6.3) and (6.4). In this way, we renounce to some abbreviations, but we gain type-inference for tokens and currencies and, consequently, precious type safety properties (if the ownership is implemented). The main downside, however, is the ambiguity of the $\multimap$ operator, but we could consider choosing different operators.

The second option proposed is more akin to the Stipula user experience design. Since the language is designed also for beginners, the introduction of "strange" and unintuitive syntax is not the best option. Furthermore, the use of $\multimap$ maintains the same general meaning for tokens as for currencies: that is, the sending of assets.

Typestate Oriented Programming Languages are one of the main solutions that the scientific community is exploring to contrast bugs and issues and to enforce safer approaches to smart contract development. In this panorama, Stipula sticks out respect to the other languages and provides features tailored specifically for the main goal it is designed for (i.e. legal contracts). Our work had the intention to study the language by comparing its expressiveness and properties with other approaches. The implementation of the solutions found in this thesis are meant to be completed in future works.

References

[1] Ryan Browne. *'Accidental' bug may have frozen $280 million worth of digital coin ether in a cryptocurrency wallet*. Retrieved March 19, 2023. 2017. URL: https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html (cit. on pp. 1, 15).

[2] Michael Coblenz. *Obsidian : Obsidian language development*. 2017. URL: https://github.com/mcoblenz/Obsidian (cit. on pp. 16, 26, 51, 141, 142).

[3] Michael Coblenz et al. "Can Advanced Type Systems Be Usable? An Empirical Study of Ownership, Assets, and Typestate in Obsidian". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428200. URL: https://doi.org/10.1145/3428200 (cit. on p. 91).

[4] Michael Coblenz et al. "Obsidian: Typestate and Assets for Safer Blockchain Programming". In: *ACM Trans. Program. Lang. Syst.* 42.3 (Nov. 2020). ISSN: 0164-0925. DOI: 10.1145/3417516. URL: https://doi.org/10.1145/3417516 (cit. on pp. 16, 37, 61, 79, 92).

[5] Silvia Crafa. "From Legal Contracts to Legal Calculi: the code-driven normativity". In: *Electronic Proceedings in Theoretical Computer Science* 368 (Sept. 2022), pp. 23–42. DOI: 10.4204/eptcs.368.2. URL: https://doi.org/10.4204%5C%2Feptcs.368.2 (cit. on pp. 25, 49).

[6] Silvia Crafa and Cosimo Laneve. "Liquidity Analysis In Resource-Aware Programming". In: *Formal Aspects of Component Software: 18th International Conference, FACS 2022, Virtual Event, November 10–11, 2022, Proceedings*. Oslo, Norway: Springer-Verlag, 2022, pp. 205–221. ISBN: 978-3-031-20871-3. DOI: 10.1007/978-3-031-20872-0_12. URL: https://doi.org/10.1007/978-3-031-20872-0_12 (cit. on pp. 14, 91).

[7] Silvia Crafa and Cosimo Laneve. "Programming Legal Contracts. – A Beginners Guide to Stipula –". In: *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. Ed. by Wolfgang Ahrendt et al. Cham: Springer International Publishing, 2022, pp. 129–146. ISBN: 978-3-031-08166-8. DOI: 10.1007/978-3-031-08166-8_7. URL: https://doi.org/10.1007/978-3-031-08166-8_7.

[8] Silvia Crafa and Luca Padovani. "The Chemical Approach to Typestate-Oriented Programming". In: *SIGPLAN Not.* 50.10 (Oct. 2015), pp. 917–934. ISSN: 0362-1340. DOI: 10.1145/2858965.2814287. URL: https://doi.org/10.1145/2858965.2814287 (cit. on p. 86).

[9] Silvia Crafa et al. “Pacta sunt servanda: Legal contracts in Stipula”. In: *Science of Computer Programming* 225 (2023), p. 102911. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2022.102911. URL: https://www.sciencedirect.com/science/article/pii/S0167642322001447 (cit. on p. 7).

[10] *Flint Language Guide*. Retrieved March 31, 2023. 2019. URL: https://github.com/flintlang/flint/blob/master/docs/language_guide.md (cit. on p. 87).

[11] *Flint2 Language Guide*. Retrieved March 31, 2023. 2020. URL: https://github.com/flintlang/flint-2/blob/master/docs/guide.md (cit. on p. 87).

[12] The Linux Foundation. *Hyperledger Fabric – Hyperledger Foundation*. Retrieved March 13, 2023. 2023. URL: https://www.hyperledger.org/use/fabric (cit. on pp. 4, 20).

[13] Cédric Fournet et al. “Inheritance in the Join Calculus”. In: *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*. Ed. by Sanjiv Kapoor and Sanjiva Prasad. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 397–408. ISBN: 978-3-540-44450-3. URL: https://doi.org/10.1007/3-540-44450-5_32 (cit. on p. 86).

[14] Ronald Garcia et al. “Foundations of Typestate-Oriented Programming”. In: *ACM Trans. Program. Lang. Syst.* 36.4 (Oct. 2014). ISSN: 0164-0925. DOI: 10.1145/2629609. URL: https://doi.org/10.1145/2629609 (cit. on p. 22).

[15] Rosita Gerbo and Luca Padovani. “Concurrent Typestate-Oriented Programming in Java”. In: *Proceedings Programming Language Approaches to Concurrency and Communication-Centric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*. Ed. by Francisco Martins and Dominic Orchard. Vol. 291. EPTCS. 2019, pp. 24–34. DOI: 10.4204/EPTCS.291.3. URL: https://doi.org/10.4204/EPTCS.291.3 (cit. on pp. 86, 87).

[16] Dimitrios Kouzapas et al. “Typechecking Protocols with Mungo and StMungo”. In: *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*. PPDP ’16. Edinburgh, United Kingdom: Association for Computing Machinery, 2016, pp. 146–159. ISBN: 9781450341486. DOI: 10.1145/2967973.2968595. URL: https://doi.org/10.1145/2967973.2968595 (cit. on p. 85).

[17] Google LLC. *Protocol Buffers Documentation*. Retrieved March 13, 2023. 2023. URL: https://protobuf.dev/ (cit. on p. 20).

[18] Loi Luu et al. “Making Smart Contracts Smarter”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 254–269. ISBN: 9781450341394. DOI: 10.1145/2976749.2978309. URL: https://doi.org/10.1145/2976749.2978309 (cit. on pp. 51, 62).

[19] João Mota. “Coping with the reality: adding crucial features to a typestate-oriented language (MSc thesis)”. Master Thesis. Universidade Nova de Lisboa, 2021. URL: https://github.com/jdmota/java-typestate-checker/blob/master/docs/msc-thesis.pdf (cit. on p. 86).

[20] João Mota, Marco Giunti, and António Ravara. “Java Typestate Checker”. In: *Coordination Models and Languages*. Ed. by Ferruccio Damiani and Ornela

Dardha. Cham: Springer International Publishing, 2021, pp. 121–133. ISBN: 978-3-030-78142-2. URL: https://doi.org/10.1007/978-3-030-78142-2_8 (cit. on p. 86).

[21] Nathaniel Popper. *A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency - New York Times*. Retrieved March 19, 2023. 2016. URL: https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html (cit. on pp. 1, 15).

[22] Franklin Schrans. "Writing Safe Smart Contracts in Flint". In: (June 2018). URL: https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1718-ug-projects/Franklin-Schrans-A-new-programming-language-for-safer-smart-contracts.pdf (cit. on p. 87).

[23] Franklin Schrans et al. "Flint for Safer Smart Contracts". In: *CoRR* abs/1904.06534 (2019). arXiv: 1904.06534. URL: http://arxiv.org/abs/1904.06534 (cit. on p. 87).

[24] Parity Technologies. *A Postmortem on the Parity Multi-Sig Library Self-Destruct*. Retrieved March 19, 2023. 2017. URL: https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/ (cit. on pp. 1, 15).

[25] *The Rust Programming Language*. Retrieved April 3, 2023. 2022. URL: https://doc.rust-lang.org/stable/book/ (cit. on pp. 2, 90).

[26] Fabian Vogelsteller and Vitalik Buterin. *EIP-20: Token Standard*. Ethereum Improvement Proposals, no. 20. Nov. 2015. URL: https://eips.ethereum.org/EIPS/eip-20 (cit. on p. 59).

[27] Philip Wadler. "Linear Types can Change the World!" In: *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*. Ed. by Manfred Broy and Cliff B. Jones. North-Holland, 1990, p. 561 (cit. on p. 15).

Acronyms

ABI *Application Binary Interface*. 21, 90

EVM *Ethereum Virtual Machine*. 88

FT *Featherweight Typestate*. 22, 91, 92

JATYC *Java Typestate Checker*. 86, 91

NFT *Non-Fungible Token*. 1

OO *Object Oriented*. 15, 23

OOP *Object Oriented Programming*. 2, 3, 15

TSOP *Typestate Oriented Programming*. 2–4, 15, 22, 85, 86, 90, 91, 93

Appendix A

Complete Examples

A.1 CoinEscrow

A.1.1 Stipula

Listing A.1: `coinescrow.stipula`

```
stipula CoinEscrow {
    asset w
    field amount

    agreement(Sender,Receiver)(amount){
        Sender,Receiver : amount
    } ==> @S1

    @S1 Sender : put()[h] (h==amount) {
        h -o w
    } ==> @S2

    @S2 Receiver : claimCoins()[] {
        w -o Receiver
    } ==> @End

    @S2 Receiver : claimPart(v)[] (v>=0 and v<=1) {
        w*v -o w,Receiver
        w -o Sender
    } ==> @End
}
```

A.1.2 Obsidian

Listing A.2: `CoinEscrow.obs`

```
asset interface Sender {
    transaction give(int v) returns Currency@Owned;
    transaction receive(Currency@Owned >> Unowned gift);
    transaction id() returns string;
    transaction amountAgreed() returns int;
}

asset interface Receiver {
    transaction give(int v) returns Currency@Owned;
    transaction receive(Currency@Owned >> Unowned gift);
    transaction id() returns string;
    transaction amountAgreed() returns int;
}

```

```
main contract CoinEscrow {
    Currency@Owned w;
    int amount;

    Sender@Shared sender;
    Receiver@Shared receiver;

    state S1;
    state S2;
    state End;

    CoinEscrow@S1(Sender@Shared s, Receiver@Shared r) {
        if (s.amountAgreed() != r.amountAgreed()) {
            revert("Agreement failed");
        }

        sender = s;
        receiver = r;

        amount = s.amountAgreed();
        w = new Currency@Owned(0);
        ->S1;
    }

    transaction put(CoinEscrow@S1 >> S2 this,
                    Currency@Owned >> Unowned h,
                    Sender@Unowned caller) {
        if (caller.id() != sender.id()) {
            revert("Call unauthorized");
        }

        if (amount != h.getAmount()) {
            revert("put failed: amount incorrect");
        }

        w.merge(h);
        ->S2;
    }

    transaction claimCoins(CoinEscrow@S2 >> @End this,
                           Receiver@Unowned caller) {
        if (caller.id() != receiver.id()) {
            revert("Call unauthorized");
        }

        receiver.receive(w.split(w.getAmount()));
        ->End;
    }

    transaction claimPart(CoinEscrow@S2 >> @End this,
                          int v,
                          Receiver@Unowned caller) {
        if (caller.id() != receiver.id()) {
            revert("Call unauthorized");
        }

        if (v < 0 || v > 100) {
            revert("claimPart failed: v must be a value for the percentage");
        }

        receiver.receive(w.split((w.getAmount()*v)/100));
        sender.receive(w.split(w.getAmount()));
        ->End;
    }
}
```

A.2 NftEscrow

A.2.1 Stipula

Listing A.3: `nftescrow.stipula`

```
stipula NftEscrow {
    asset nft

    agreement(Sender,Receiver)(){} =>@S1

    @S1 Sender : put()[t] () {
        t -o nft
    } ==> @S2

    @S2 Receiver : claimNft()[] {
        nft -o Receiver
    } ==> @End
}
```

A.2.2 Obsidian

Listing A.4: `NFTEscrow.obs`

```
asset interface Sender {
    transaction giveToken() returns NFT@Owned;
    transaction receiveToken(NFT@Owned >> Unowned gift);
    transaction id() returns string;
}

asset interface Receiver {
    transaction giveToken() returns NFT@Owned;
    transaction receiveToken(NFT@Owned >> Unowned gift);
    transaction id() returns string;
}

main contract NftEscrow {

    Sender@Shared sender;
    Receiver@Shared receiver;

    state S1;
    state S2 {
        NFT@Owned nft;
    }
    state End;

    NftEscrow@S1(Sender@Shared s, Receiver@Shared r) {
        sender = s;
        receiver = r;

        ->S1;
    }

    transaction put(NftEscrow@S1 >> S2 this,
                    NFT@Owned >> Unowned t,
                    Sender@Unowned caller) {
        if (caller.id() != sender.id()) {
            revert("Call unauthorized");
        }

        ->S2(nft = t);
    }

    transaction claimNft(NftEscrow@S2 >> @End this,
                         Receiver@Unowned caller) {
        if (caller.id() != receiver.id()) {
            revert("Call unauthorized");
        }
```

```
46
47          receiver.receiveToken(nft);
48          ->End;
49      }
50  }
```

A.3 AssetSend

Listing A.5: `Party.obs`

```
import "Currency.obs"
import "NFT.obs"
import "List.obs"

main asset contract Party {
    Currency@Owned wallet;
    List[NFT@Owned]@Owned tokens;

    Party@Owned (int m) {
        wallet = new Currency(m);
        tokens = new List[NFT@Owned]();
    }

    transaction give(int v) returns Currency@Owned {
        return wallet.split(v);
    }

    transaction receive(Currency@Owned >> Unowned gift) {
        wallet.merge(gift);
    }

    transaction getAmount() returns int {
        return wallet.getValue();
    }

    transaction receiveToken(NFT@Owned >> Unowned gift) {
        tokens.push(gift);
    }

    transaction giveToken(NFT@Unowned t) returns NFT@Owned {
        return tokens.removeElement(t,new NFTComparator());
    }

    transaction getToken(int i) returns NFT@Unowned {
        return tokens.get(i);
    }
}
```

Listing A.6: `Currency.obs`

```
main asset contract Currency {
    int value;

    Currency@Owned(int v) {
        value = v;
    }

    transaction getValue(Currency@Unowned this) returns int {
        return value;
    }

    transaction split(Currency@Owned this, int v) returns Currency@Owned {
        if (v > value) {
            revert;
        }
        value = value - v;
        Currency result = new Currency(v);
        return result;
    }

    transaction merge(Currency@Owned this, Currency@Owned >> Unowned other) {
        value = value + other.getValue();
        disown other;
    }
}
```

Listing A.7: `NFT.obs`

```
import "Comparator.obs"

main asset contract NFT {
    string ID;

    transaction getID() returns string {
        return ID;
    }

    transaction equals(NFT@Unowned other) returns bool {
        return ID == other.getID();
    }
}

contract NFTComparator implements Comparator[NFT] {
    transaction equals(NFT@Unowned a, NFT@Unowned b) returns bool {
        return a.equals(b);
    }
}
```

Listing A.8: `Comparator.obs`

```
// Taken from https://github.com/mcoblenz/Obsidian/tree/master/resources/
    demos/ERC20/Comparator.obs

interface Comparator[KeyType] {
    transaction equals(KeyType@Unowned a, KeyType@Unowned b) returns bool;
}
```

Listing A.9: `List.obs`

```
import "Comparator.obs"

// Implementation of a list
main asset contract List[asset ValueType] {
    state Empty;
    state HasNext {
        ValueType@Owned info;
        List@Owned tail;
    };

    List@Empty() {
        ->Empty;
    }

    transaction get(List@Owned this, int index) returns ValueType@Unowned {
        if (this in Empty) {
            revert("Index out of bounds");
        }

        [this@HasNext];

        if (index < 0) {
            revert("Index out of bounds");
        }

        if (index == 0) {
            return info;
        }

        return tail.get(index - 1);
    }

    transaction push(List@Owned this, ValueType@Owned >> Unowned element) {
        if (this in Empty) {
            ->HasNext(info = element, tail = new List[ValueType]());
            return;
        }

        tail.push(element);
    }

```

```
42      transaction remove(List@Owned this, int index) returns ValueType@Owned {
43          if (this in Empty) {
44              revert("List empty");
45          }
46
47          [this@HasNext];
48
49          if (index == 0){
50              ValueType res = info;
51              if (tail in HasNext){
52                  info = tail.remove(0);
53              } else {
54                  [tail@Empty];
55                  disown tail;
56                  ->Empty;
57              }
58              return res;
59          } else {
60              return tail.remove(index - 1);
61          }
62      }
63
64      transaction removeElement(List@Owned this,
65                                ValueType@Unowned element,
66                                Comparator comparator) returns ValueType@Owned
                                      {
67          if (this in Empty) {
68              revert("List empty");
69          }
70
71          [this@HasNext];
72
73          if (comparator.equals(info,element)){
74              ValueType res = info;
75              if (tail in HasNext){
76                  info = tail.removeElement(tail.get(0),comparator);
77              } else {
78                  [tail@Empty];
79                  disown tail;
80                  ->Empty;
81              }
82              return res;
83          } else {
84              return tail.removeElement(element,comparator);
85          }
86      }
87
88  }
```

Listing A.10: `Main.obs`

```
1   import "Party.obs"
2
3   main contract Main {
4       transaction main(remote Party@Shared alice, remote Party@Shared bob,
            remote Party@Shared charles) {
5           int a = alice.getAmount();
6           int b = bob.getAmount();
7
8           // Alice gives to Bob 5 dollars
9           bob.receive(alice.give(5));
10
11          a = alice.getAmount();
12          b = bob.getAmount();
13
14          NFT x = new NFT();
15          NFT y = new NFT();
16          NFT z = y;
17
18          [x@Owned];
19          [y@Unowned];
20
21          y = x;
22
```

```
23          [x@Unowned];
24          [y@Owned];
25          disown y;
26
27          charles.receiveToken(z);    // z -o charles
28      }
29  }
```

A.4 BikeRental

A.4.1 Stipula

Listing A.11: `bikerent.stipula`

```
stipula BikeRental {
    asset wallet
    field cost, rentingTime, code
    init Inactive

    agreement (Lender,Borrower,Authority)(rentingTime,cost){
        Lender , Borrower: rentingTime , cost
    } ==> @Inactive

    @Inactive Lender : offer(x)[] {
        x -> code;
        _
    } ==> @Payment

    @Payment Borrower : pay()[h] (h == cost) {
        h -o wallet
        code -> Borrower;
        now+rentingTime >>
            @Using {
                "EndReached" -> Borrower
            } ==> @Return
    } ==> @Using

    @Using Borrower : end()[] {
        "EndReached" -> Lender;
        _
    } ==> @Return

    @Return Lender : rentalOk()[] {
        wallet -o Lender;
        _
    } ==> @End

    @Using@Return Lender, Borrower : dispute(x)[] {
        x -> _;
        _
    } ==> @Dispute

    @Dispute Authority : verdict(x,y)[] (y>=0 && y<=1) {
        x -> Lender
        x -> Borrower
        (y*wallet) -o wallet, Lender
        wallet -o Borrower;
        _
    } ==> @End
}
```

A.4.2 Obsidian

Listing A.12: `BikeRent.obs`

```
import "Currency.obs"
import "TimeManager.obs"

asset interface Lender {
    transaction give(int v) returns Currency@Owned;

    transaction receive(Currency@Owned >> Unowned gift);

    transaction getAmount() returns int;

    transaction id() returns string;

```

```
13      transaction sendMessage(string m);
14
15      transaction sendReasons(string r);
16
17      transaction sendMotivations(string m);
18
19      transaction rentingTimeAgreed() returns int;
20
21      transaction costAgreed() returns int;
22  }
23
24  asset interface Borrower {
25      transaction give(int v) returns Currency@Owned;
26
27      transaction receive(Currency@Owned >> Unowned gift);
28
29      transaction getAmount() returns int;
30
31      transaction id() returns string;
32
33      transaction sendCode(string c);
34
35      transaction sendMotivations(string m);
36
37      transaction sendMessage(string msg);
38
39      transaction sendReasons(string r);
40
41      transaction rentingTimeAgreed() returns int;
42
43      transaction costAgreed() returns int;
44  }
45
46  asset interface Authority {
47      transaction give(int v) returns Currency@Owned;
48
49      transaction receive(Currency@Owned >> Unowned gift);
50
51      transaction getAmount() returns int;
52
53      transaction id() returns string;
54
55      transaction sendReasons(string r);
56  }
57
58  contract Event implements EventInterface {
59      BikeRent@Shared rental;
60      Borrower@Unowned caller;
61
62      Event@Owned(BikeRent@Shared r, Borrower@Unowned c) {
63          rental = r;
64          caller = c;
65      }
66
67      transaction action() {
68          if (rental in Using) {
69              rental.reaction(caller);
70          }
71      }
72  }
73
74  main asset contract BikeRent {
75      // Fields
76      int cost;
77      int rentingTime;
78
79      // Parties
80      Lender@Shared lender;
81      Borrower@Shared borrower;
82      Authority@Shared authority;
83
84      // Time
85      TimeManager@Shared timeManager;
86
```

```
87      state Inactive;
88      state Payment;
89      state Using {
90          int expirationTime;
91      };
92      state Return;
93      state End;
94      state Dispute;
95
96      string code available in Payment,Using,Return;
97      Currency@Owned wallet available in Using,Return,Dispute;
98
99      BikeRent@Inactive(
100         Lender@Shared l, Borrower@Shared b, Authority@Shared a,
101         TimeManager@Shared tm){
102         // Fields in the agreement
103         if (l.costAgreed() != b.costAgreed() ||
104                 l.rentingTimeAgreed() != b.rentingTimeAgreed()) {
105             revert("Agreement failed");
106         }
107
108         cost = l.costAgreed();
109         rentingTime = l.rentingTimeAgreed();
110
111         // References to parties
112         lender = l;
113         borrower = b;
114         authority = a;
115
116         // Time manager
117         timeManager = tm;
118
119         ->Inactive;
120     }
121
122     transaction offer(BikeRent@Inactive >> Payment this,
123                       Lender@Unowned l, string x) {
124         // Checks
125         if (l.id() != lender.id()) {
126             revert("Not authorized lender");
127         }
128
129         // Body
130         ->Payment(code = x);
131     }
132
133     transaction pay(BikeRent@Payment >> Using this,
134                     Borrower@Unowned b, Currency@Owned >> Unowned h) {
135         // Checks
136         if (b.id() != borrower.id()) {
137             revert("Not authorized borrower");
138         }
139
140         if (h.getValue() != cost) {
141             revert("Currency given don't match with cost");
142         }
143
144         // Body
145         borrower.sendCode(code);
146
147         // REACTION
148         // timeManager.register(timeManager.now(),new Event(this,b));
149         // error caused by ownership
150         ->Using(wallet = h, expirationTime = timeManager.now() + rentingTime)
                ;
151     }
152
153     transaction reaction(BikeRent@Using >> Using | Return this,
154                          Borrower@Unowned b) {
155         if (b.id() != borrower.id()){
156             revert("Not authorized borrower");
157         }
158         borrower.sendMessage("End_Reached");
159         ->Return;
```

```
160     }
161
162     transaction getExpirationTime(BikeRent@Using this) returns int {
163         return expirationTime;
164     }
165
166     transaction end(BikeRent@Using >> Return this, Borrower@Unowned b) {
167         // Checks
168         if (b.id() != borrower.id()) {
169             revert("Not authorized borrower");
170         }
171
172         // Body
173         lender.sendMessage("End_Reached");
174         ->Return;
175     }
176
177     transaction rentalOk(BikeRent@Return >> End this, Lender@Unowned l) {
178         // Checks
179         if (l.id() != lender.id()) {
180             revert("Not authorized lender");
181         }
182
183         // Body
184         lender.receive(wallet);
185         ->End;
186     }
187
188     transaction dispute(BikeRent@Using | Return >> Dispute this,
189                         string callerID, string x) {
190         // Checks
191         if (callerID != lender.id() || callerID != borrower.id()) {
192             revert("Not authorized caller");
193         }
194
195         // Body
196         lender.sendReasons(x);
197         borrower.sendReasons(x);
198         authority.sendReasons(x);
199
200         ->Dispute;
201     }
202
203     // x is the string of motivations, y is the percentage of the wallet
            given
204     // to the lender as reimbursement
205     transaction verdict(BikeRent@Dispute >> End this,
206                         Authority@Unowned a, string x, int y) {
207         // Checks
208         if (a.id() != authority.id()) {
209             revert("Error on authority authentication");
210         }
211
212         if (y < 0 || y > 100) {
213             revert("Error on bounds over y");
214         }
215
216         // Body
217         lender.sendMotivations(x);
218         borrower.sendMotivations(x);
219
220         int total = (wallet.getValue()*y)/100;
221         Currency reimbursement = wallet.split(total);
222
223         lender.receive(reimbursement);
224         borrower.receive(wallet);
225
226         ->End;
227     }
228
229     transaction getTimeManager() returns TimeManager@Shared {
230         return timeManager;
231     }
232 }
```

Listing A.13: LenderImpl.obs

```
import "BikeRent.obs"

main asset contract LenderImpl implements Lender {
    Currency@Owned wallet;
    string id;

    int rentingTime;
    int cost;

    string message;
    string reasons;
    string motivations;


    LenderImpl@Owned (int m, string i, int r, int c) {
        wallet = new Currency(m);
        id = i;
        rentingTime = r;
        cost = c;

        message = "";
        reasons = "";
        motivations = "";
    }

    transaction give(int v) returns Currency@Owned {
        return wallet.split(v); //gift is Owned
    }

    transaction receive(Currency@Owned >> Unowned gift) {
        wallet.merge(gift);
    }

    transaction getAmount() returns int {
        return wallet.getValue();
    }

    transaction id() returns string {
        return id;
    }

    transaction sendMessage(string m) {
        message = m;
    }

    transaction sendReasons(string r) {
        reasons = r;
    }

    transaction sendMotivations(string m) {
        motivations = m;
    }

    transaction rentingTimeAgreed() returns int {
        return rentingTime;
    }

    transaction costAgreed() returns int {
        return cost;
    }

    transaction getMotivations() returns string {
        return motivations;
    }
}
```

Listing A.14: BorrowerImpl.obs

```
import "BikeRent.obs"

main asset contract BorrowerImpl implements Borrower {
    Currency@Owned wallet;
```

```
    string id;

    int rentingTime;
    int cost;

    string code;
    string reasons;
    string motivations;

    string message;

    BorrowerImpl@Owned (int m, string i, int r, int c) {
        wallet = new Currency(m);
        id = i;

        rentingTime = r;
        cost = c;

        code = "";
        reasons = "";
        motivations = "";
        message = "";
    }

    transaction give(int v) returns Currency@Owned {
        return wallet.split(v); //gift is Owned
    }

    transaction receive(Currency@Owned >> Unowned gift) {
        wallet.merge(gift);
    }

    transaction getAmount() returns int {
        return wallet.getValue();
    }

    transaction id() returns string {
        return id;
    }

    transaction sendCode(string c) {
        code = c;
    }

    transaction sendMotivations(string m) {
        motivations = m;
    }

    transaction sendMessage(string msg) {
        message = msg;
    }

    transaction sendReasons(string r) {
        reasons = r;
    }

    transaction rentingTimeAgreed() returns int {
        return rentingTime;
    }

    transaction costAgreed() returns int {
        return cost;
    }

    transaction getCode() returns string {
        return code;
    }
}
```

Listing A.15: `AuthorityImpl.obs`

```
import "BikeRent.obs"

main asset contract AuthorityImpl implements Authority {
```

```
    Currency@Owned wallet;
    string id;

    string reasons;

    AuthorityImpl@Owned (int m, string i) {
        wallet = new Currency(m);
        id = i;
        reasons = "";
    }

    transaction give(int v) returns Currency@Owned {
        return wallet.split(v); //gift is Owned
    }

    transaction receive(Currency@Owned >> Unowned gift) {
        wallet.merge(gift);
    }

    transaction getAmount() returns int {
        return wallet.getValue();
    }

    transaction id() returns string {
        return id;
    }

    transaction sendReasons(string r) {
        reasons = r;
    }

    transaction getReasons() returns string {
        return reasons;
    }

    transaction lenderRightsPercentage() returns int {
        // compute the percentage of rightfulness of the lender
        return 50;
    }

    transaction explainVerdict() returns string {
        return "Motivations based on the reasons received";
    }
}
```

Listing A.16: `TimeManager.obs`

```
import "Dict.obs"
import "List.obs"
import "Integer.obs"

interface EventInterface {
    transaction action();
}

main asset contract TimeManager {
    Dict[Integer,List[EventInterface]@Owned]@Owned registry;
    int clock;

    state Active;
    state Inactive;

    TimeManager@Inactive() {
        clock = 0;
        registry =
            new Dict[Integer,List[EventInterface]@Owned](
                                        new IntegerComparator());
        ->Inactive;
    }

    transaction register(int t, EventInterface@Owned >> Unowned event) {
        Integer time = new Integer(t);
        Option[List[EventInterface]] eventList = registry.remove(time);

```

```
        if (eventList in None) {
            List[EventInterface] l = new List[EventInterface]();
            l.push(event);
            registry.insert(time,l);
        } else {
            List[EventInterface] l = eventList.unpack();
            l.push(event);
            registry.insert(time,l);
        }
    }

    transaction tick(TimeManager@Active this) {
        Integer time = new Integer(clock);
        Option[List[EventInterface]] events = registry.remove(time);

        if (events in Some) {
            performActions(events.unpack());
        }

        clock = clock + 1;
    }

    private transaction performActions(List[EventInterface]@Owned events) {

        if (events in HasNext){
            EventInterface ev = events.pop();
            ev.action();
            performActions(events);
        }
    }

    transaction start(TimeManager@Inactive >> Active this) {
        ->Active;
    }

    transaction now() returns int {
        return clock;
    }
}
```

Listing A.17: `Dict.obs`

```
// Taken from https://github.com/mcoblenz/Obsidian/tree/master/resources/
    demos/ERC20/Dict.obs

import "Comparator.obs"

contract Option[asset T@s] {
    state None;
    asset state Some {
        T@s val;
    }

    Option@None() {
        ->None;
    }

    Option@Some(T@s >> Unowned v) {
        ->Some(val = v);
    }

    transaction unpack(Option[T@s]@Some >> None this) returns T@s {
        T result = val;
        ->None;
        return result;
    }
}

main asset contract Dict[KeyType, asset ValueType@s where s is Owned] {
    DictImpl[KeyType, ValueType@s]@(Empty | HasNext) dictImpl;

    Dict@Owned(Comparator@Unowned _comparator) {
        dictImpl = new DictImpl[KeyType, ValueType@s](_comparator);
    }
```

```
32
33      transaction replace(Dict@Unowned this,
34                          KeyType@Unowned _key,
35                          ValueType@s >> Unowned _value)
36                          returns Option[ValueType@s]@Owned {
37          return dictImpl.replace(_key, _value);
38      }
39
40      transaction remove(Dict@Unowned this, KeyType@Unowned _key)
41      returns Option[ValueType@s]@Owned {
42          return dictImpl.remove(_key);
43      }
44
45      transaction peek(Dict@Unowned this, KeyType@Unowned _key)
46      returns Option[ValueType@Unowned]@Owned {
47          return dictImpl.peek(_key);
48      }
49
50      transaction insert(Dict@Unowned this,
51                         KeyType@Unowned _key,
52                         ValueType@s >> Unowned _value) {
53          Option[ValueType@s] existingValue = replace(_key, _value);
54          if (existingValue in Some) {
55              revert ("insert operation is only permitted when there is no
                    existing value for the given key.");
56          }
57      }
58  }
59
60  contract DictImpl[KeyType, asset ValueType@s where s is Owned] {
61      state Empty;
62      asset state HasNext;
63
64      DictImpl[KeyType, ValueType@s]@(Empty | HasNext) next available in
            HasNext;
65      KeyType@Unowned key available in HasNext, PrivateHasKeyAndValue;
66      ValueType@s value available in HasNext, PrivateHasKeyAndValue,
67                                      PrivateHasValue;
68
69      Comparator[KeyType]@Unowned comparator;
70
71      asset state PrivateHasKeyAndValue;
72      asset state PrivateHasValue;
73
74      DictImpl@Empty(Comparator@Unowned _comparator) {
75          comparator = _comparator;
76          ->Empty;
77      }
78
79      // Puts the given key/value pair into the DictImplionary.
80      // If the key was already in the DictImplionary,
81      // returns an Option containing the old value.
82      // Otherwise, returns None.
83      transaction replace(DictImpl@(Empty | HasNext) this,
84                          KeyType@Unowned _key,
85                          ValueType@s >> Unowned _value)
86                          returns Option[ValueType@s]@Owned {
87          switch this {
88              case HasNext {
89                  if (comparator.equals(key, _key)) {
90                      ValueType oldValue = value;
91                      value = _value;
92                      return new Option[ValueType@s](oldValue);
93                  }
94                  else {
95                      return next.replace(_key, _value);
96                  }
97              }
98              case Empty {
99                  ->HasNext(key = _key, value = _value,
100                            next =
101                                 new DictImpl[KeyType, ValueType@s](comparator
                                         ));
102                 return new Option[ValueType@s]();
```

```
            }
        }
    }


    // Attempts to remove the key/value pair for the given key,
    // returning the value. If the key is not found, returns None.
    transaction remove(DictImpl@(Empty | HasNext) this, KeyType@Unowned _key)
    returns Option[ValueType@s]@Owned {
        switch this {
            case HasNext {
                if (comparator.equals(key, _key)) {

                    ValueType oldValue = value;
                    if (next in HasNext) {
                        DictImpl[KeyType, ValueType@s] newNext =
                                                            next.extractNext
                                                                 ();
                        KeyType newKey = next.extractKey();
                        ValueType newValue = next.extractValue();
                        // next is now Empty, so we can discard it implicitly
                            .

                        ->HasNext(next = newNext,
                                  key = newKey,
                                  value = newValue);
                    }
                    else {
                        // next is already Empty. We're going to be empty too
                            .
                        ->Empty;
                    }

                    return new Option[ValueType@s](oldValue);

                }
                else {
                    return next.remove(_key);
                }
            }
            case Empty {
                return new Option[ValueType@s]();
            }
        }
    }

    private
    transaction extractNext(DictImpl@HasNext >> PrivateHasKeyAndValue this)
    returns DictImpl@(HasNext | Empty) {
        DictImpl[KeyType, ValueType@s] result = next;
        ->PrivateHasKeyAndValue;
        return result;
    }

    private
    transaction extractKey(
                    DictImpl@PrivateHasKeyAndValue >> PrivateHasValue this)
                    returns KeyType@Unowned {
        KeyType result = key;
        ->PrivateHasValue;
        return result;
    }

    private
    transaction extractValue(DictImpl@PrivateHasValue >> Empty this)
    returns ValueType@s {
        ValueType result = value;
        ->Empty;
        return result;
    }

    transaction peek(DictImpl@Unowned this,
                     KeyType@Unowned _key)
                     returns Option[ValueType@Unowned]@Owned {
```

```
174             switch this {
175                 case HasNext {
176                     if (comparator.equals(key, _key)) {
177                         return new Option[ValueType@Unowned](value);
178                     }
179                     else {
180                         return next.peek(_key);
181                     }
182                 }
183                 case Empty {
184                     return new Option[ValueType@Unowned]();
185                 }
186                 // These additional cases are here so that
187                 // peek can be called with unowned references.
188                 case PrivateHasValue {
189                     revert "Do not call peek on inconsistent DictImplionaries.";
190                 }
191                 case PrivateHasKeyAndValue {
192                     revert "Do not call peek on inconsistent DictImplionaries.";
193                 }
194             }
195         }
196 }
```

Listing A.18: `List.obs`

```
1  // Implementation of a FIFO list
2  main asset contract List[ValueType] {
3      state Empty;
4      state HasNext {
5          ValueType@Owned info;
6          List@Owned tail;
7      };
8
9      List@Empty() {
10         ->Empty;
11     }
12
13     transaction get(List@Owned this, int index) returns ValueType@Unowned {
14         if (this in Empty) {
15             revert("Index out of bounds");
16         }
17
18         [this@HasNext];
19
20         if (index < 0) {
21             revert("Index out of bounds");
22         }
23
24         if (index == 0) {
25             return info;
26         }
27
28         return tail.get(index - 1);
29     }
30
31     transaction push(List@Owned this, ValueType@Owned >> Unowned element) {
32         if (this in Empty) {
33             ->HasNext(info = element, tail = new List[ValueType]());
34             return;
35         }
36
37         tail.push(element);
38     }
39
40     transaction pop(List@Owned this) returns ValueType@Owned {
41         if (this in Empty) {
42             revert("List empty");
43         }
44
45         [this@HasNext];
46
47         ValueType res = info;
48         if (tail in HasNext){
```

```
            info = tail.pop();
        } else {
            [tail@Empty];
            disown tail;
            ->Empty;
        }
        return res;
    }
}
```

Listing A.19: `Integer.obs`

```
// Taken from https://github.com/mcoblenz/Obsidian/tree/master/resources/
    demos/ERC20/Integer.obs

import "Comparator.obs"

main contract Integer {
    int value;

    Integer@Owned(int _value) {
        value = _value;
    }

    transaction getValue() returns int {
        return value;
    }
}


contract IntegerComparator implements Comparator[Integer] {
    transaction equals(Integer@Unowned a, Integer@Unowned b) returns bool {
        return a.getValue() == b.getValue();
    }
}
```

Listing A.20: `Comparator.obs`

```
// Taken from https://github.com/mcoblenz/Obsidian/tree/master/resources/
    demos/ERC20/Comparator.obs

interface Comparator[KeyType] {
    transaction equals(KeyType@Unowned a, KeyType@Unowned b) returns bool;
}
```

Listing A.21: `BorrowerMain.obs`

```
import "BikeRent.obs"
import "BorrowerImpl.obs"

main contract BorrowerMain {
    transaction main(remote BikeRent@Shared rental,
                     remote BorrowerImpl@Shared borrower) {

        int costAgreed = borrower.costAgreed();

        if (rental in Payment) {
            rental.pay(borrower,borrower.give(costAgreed));
            TimeManager timeManager = rental.getTimeManager();
            timeManager.register(
                timeManager.now() + borrower.rentingTimeAgreed(),
                new Event(rental,borrower));
        }

        string bikeCode = borrower.getCode();

        waitUntil(borrower.rentingTimeAgreed());

        if (rental in Using) {
            rental.end(borrower);
        }
```

```
25
26          bool someProblem = true;
27
28          if (someProblem){
29              if (rental in Using) {
30                  rental.dispute(borrower.id(),"Some problem occurred");
31                  waitStateEnd(rental);
32              }
33
34              if (rental in Return) {
35                  rental.dispute(borrower.id(),"Some problem occurred");
36                  waitStateEnd(rental);
37              }
38
39              // In borrower.motivations will find the verdict
40              // from the authority
41          }
42
43      }
44
45      private transaction waitStateEnd(BikeRent@Shared br)
46      returns BikeRent@Shared {
47          if (br in End){
48              return br;
49          }
50          return waitStateEnd(br);
51      }
52
53      private transaction waitUntil(int time) {
54          waitUntilRec(0,time);
55      }
56
57      private transaction waitUntilRec(int now, int time) {
58          if (now > time){
59              return;
60          }
61          waitUntilRec(now + 1, time);
62      }
63  }
```

Listing A.22: LenderMain.obs

```
1   import "BikeRent.obs"
2   import "LenderImpl.obs"
3
4   main contract LenderMain {
5       transaction main(remote BikeRent@Shared rental,
6                        remote LenderImpl@Shared lender) {
7           if (rental in Inactive) {
8               rental.offer(lender,"10");
9           }
10
11          waitUntil(lender.rentingTimeAgreed());
12
13          if (rental in Return) {
14              rental.rentalOk(lender);
15          }
16
17          bool someProblem = true;
18
19          if (someProblem){
20              if (rental in Using) {
21                  rental.dispute(lender.id(),"Some problem occurred");
22                  waitStateEnd(rental);
23              }
24
25              if (rental in Return) {
26                  rental.dispute(lender.id(),"Some problem occurred");
27                  waitStateEnd(rental);
28              }
29
30              // In lender.motivations will find the verdict
31              // from the authority
32          }
```

```
33
34      }
35
36      private
37      transaction waitStateReturn(BikeRent@Shared br) returns BikeRent@Shared {
38          if (br in Return){
39              return br;
40          }
41          return waitStateReturn(br);
42      }
43
44      private
45      transaction waitStateEnd(BikeRent@Shared br) returns BikeRent@Shared {
46          if (br in End){
47              return br;
48          }
49          return waitStateReturn(br);
50      }
51
52      private transaction waitUntil(int time) {
53          waitUntilRec(0,time);
54      }
55
56      private transaction waitUntilRec(int now, int time) {
57          if (now > time){
58              return;
59          }
60          waitUntilRec(now + 1, time);
61      }
62  }
```

Listing A.23: `AuthorityMain.obs`

```
1   import "BikeRent.obs"
2   import "AuthorityImpl.obs"
3
4   main contract AuthorityMain {
5       transaction main(remote BikeRent@Shared rental,
6                        remote AuthorityImpl@Shared authority) {
7
8           waitStateDispute(rental);
9
10          int lenderRightfulness = authority.lenderRightsPercentage();
11          string motivations = authority.explainVerdict();
12
13          if (rental in Dispute) {
14              rental.verdict(authority,motivations,lenderRightfulness);
15          }
16      }
17
18      private transaction waitStateDispute(BikeRent@Shared br)
19      returns BikeRent@Shared {
20          if (br in Dispute){
21              return br;
22          }
23          return waitStateDispute(br);
24      }
25
26      private transaction waitUntil(int time) {
27          waitUntilRec(0,time);
28      }
29
30      private transaction waitUntilRec(int now, int time) {
31          if (now > time){
32              return;
33          }
34          waitUntilRec(now + 1, time);
35      }
36  }
```

Listing A.24: `TimeManagerMain.obs`

```
import "BikeRent.obs"

main contract TimeManagerMain {

    transaction main(remote BikeRent@Shared rental) {

        TimeManager tm = rental.getTimeManager();
        if (tm in Inactive) {
            tm.start();
        }

        if (tm in Active){
            keepTicking(tm);
        }
    }

    private transaction keepTicking(TimeManager@Active tm) {
        tm.tick();
        keepTicking(tm);
    }
}
```

A.5 Bet

A.5.1 Stipula

Listing A.25: `bet.stipula`

```
stipula Bet {
    asset wallet1, wallet2
    field val1, val2, source, alea, amount, t_before, t_after
    init Init

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

    @Init Better1 : place_bet(x)[h] (h == amount) {
        h -o wallet1
        x -> val1;
        t_before >> @First {
            wallet1 -o Better1
        } ==> @Fail
    } ==> @First

    @First Better2 : place_bet(x)[h] (h == amount) {
        h -o wallet2
        x -> val2
        alea -> DataProvider;
        t_after >> @Run {
            wallet1 -o Better1
            wallet2 -o Better2
        } ==> @Fail
    } ==> @Run

    @Run DataProvider : data(x,y,z)[] (x==source && x==alea) {
        if (z==val1 && z==val2) {
            wallet1 -o Better1
            wallet2 -o Better2
        } else if (z==val1 && z!=val2) {
            wallet2 -o Better1
            wallet1 -o Better1
        } else if (z!=val1 && z==val2) {
            wallet1 -o Better2
            wallet2 -o Better2
        } else {
            wallet2 -o DataProvider
            wallet1 -o DataProvider
        };
        -
    } ==> @End
}
```

A.5.2 Obsidian

Listing A.26: `Bet.obs`

```
import "TimeManager.obs"
import "Currency.obs"

asset interface BetterInterface {
    transaction give(int v) returns Currency@Owned;

    transaction receive(Currency@Owned >> Unowned gift);

    transaction getAmount() returns int;
    transaction id() returns string;

    transaction getSourceAgreed() returns string;
```

```
13      transaction getAleaAgreed() returns string;
14      transaction getTAfterAgreed() returns int;
15      transaction getTBeforeAgreed() returns int;
16      transaction getAmountAgreed() returns int;
17  }
18
19  asset interface DataProviderInterface {
20      transaction give(int v) returns Currency@Owned;
21
22      transaction receive(Currency@Owned >> Unowned gift);
23
24      transaction getAmount() returns int;
25      transaction id() returns string;
26
27      transaction getSourceAgreed() returns string;
28      transaction getAleaAgreed() returns string;
29      transaction getTAfterAgreed() returns int;
30  }
31
32  contract EventFirst implements EventInterface{
33
34      Bet@Shared bet;
35      BetterInterface@Unowned caller;
36
37      EventFirst@Owned(Bet@Shared b, BetterInterface@Unowned c) {
38          bet = b;
39          caller = c;
40      }
41
42      transaction action() {
43          if (bet in First){
44              bet.reaction1(caller);
45          }
46      }
47  }
48
49  contract EventSecond implements EventInterface{
50
51      Bet@Shared bet;
52      BetterInterface@Unowned caller;
53
54      EventSecond@Owned(Bet@Shared b, BetterInterface@Unowned c) {
55          bet = b;
56          caller = c;
57      }
58
59      transaction action() {
60          if (bet in Run){
61              bet.reaction2(caller);
62          }
63      }
64  }
65
66  main asset contract Bet {
67      // Fields
68      string alea;
69      string source;
70
71      int amount;
72
73      int t_before;
74      int t_after;
75
76      // Parties
77      BetterInterface@Shared better1;
78      BetterInterface@Shared better2;
79      DataProviderInterface@Shared dataProvider;
80
81      // Time Manager
82      TimeManager@Shared timeManager;
83
84      // States
85      state Init;
86      state First;
```

```
87      state Run {
88          Currency@Owned wallet2; // Asset
89          string val2; // Field
90      };
91      state End;
92      state Fail;
93
94      Currency@Owned wallet1 available in First, Run; // Asset
95      string val1 available in First, Run; // Field
96
97      Bet@Init (BetterInterface@Shared b1,
98                BetterInterface@Shared b2,
99                DataProviderInterface@Shared dp,
100               TimeManager@Shared tm,
101               string s, string a, int am, int t_b, int t_a) {
102
103         if (b1.getSourceAgreed() != b2.getSourceAgreed() ||
104                                  b2.getSourceAgreed() != dp.getSourceAgreed())
                                       {
105             revert("Agreement failed on Source");
106         }
107
108         if (b1.getAleaAgreed() != b2.getAleaAgreed() ||
109                                     b2.getAleaAgreed() != dp.getAleaAgreed())
                                          {
110             revert("Agreement failed on Alea");
111         }
112
113         if (b1.getTAfterAgreed() != b2.getTAfterAgreed() ||
114                                  b2.getTAfterAgreed() != dp.getTAfterAgreed())
                                       {
115             revert("Agreement failed on T_After");
116         }
117
118         if (b1.getTBeforeAgreed() != b2.getTBeforeAgreed()) {
119             revert("Agreement failed on T_Before");
120         }
121
122         if (b1.getAmountAgreed() != b2.getAmountAgreed()) {
123             revert("Agreement failed on Amount");
124         }
125
126         better1 = b1;
127         better2 = b2;
128         dataProvider = dp;
129
130         source = b1.getSourceAgreed();
131         alea = b1.getAleaAgreed();
132         amount = b1.getAmountAgreed();
133         t_before = b1.getTBeforeAgreed();
134         t_after = b1.getTAfterAgreed();
135
136         timeManager = tm;
137
138         if (timeManager in Inactive) {
139             timeManager.start();
140         }
141
142         ->Init;
143     }
144
145     transaction place_bet1(Bet@Init >> First this,
146                            BetterInterface@Shared b1,
147                            string x,
148                            Currency@Owned >> Unowned h) {
149         if (b1.id() != better1.id()) {
150             revert("Error on better1 authentication");
151         }
152
153         if (h.getValue() != amount) {
154             revert("Bet doesn't match the amount agreed");
155         }
156
157         // REACTION 1
```

```
158         ->First(val1 = x, wallet1 = h);
159     }
160
161     transaction reaction1 (Bet@First >> First | Fail this,
162                            BetterInterface@Unowned b1) {
163         if (b1.id() != better1.id()) {
164             revert("Error on better1 authentication");
165         }
166
167         better1.receive(wallet1);
168         ->Fail;
169     }
170
171     transaction place_bet2(Bet@First >> Run this,
172                            BetterInterface@Shared b2,
173                            string x,
174                            Currency@Owned >> Unowned h) {
175         if (b2.id() != better2.id()) {
176             revert("Error on better2 authentication");
177         }
178
179         if (h.getValue() != amount) {
180             revert("Bet doesn't match the amount agreed");
181         }
182
183         // REACTION 2
184         ->Run(val2 = x, wallet2 = h);
185     }
186
187     transaction reaction2 (Bet@Run >> Run | Fail this,
188                            BetterInterface@Unowned b2) {
189         if (b2.id() != better2.id()) {
190             revert("Error on better2 authentication");
191         }
192
193         better1.receive(wallet1);
194         better2.receive(wallet2);
195         ->Fail;
196     }
197
198     transaction data(Bet@Run >> End this,
199                      DataProviderInterface@Shared dp ,
200                      string x, string y, string z) {
201         if (dp.id() != dataProvider.id()) {
202             revert("Error on data provider authentication");
203         }
204
205         if (x != source || y != alea) {
206             revert("Source or alea not reliable");
207         }
208
209         // Body
210         if (z == val1 && z == val2) {
211             better1.receive(wallet1);
212             better2.receive(wallet2);
213         } else {
214             if ( z== val1 && z != val2) {
215                 better1.receive(wallet1);
216                 better1.receive(wallet2);
217             } else {
218                 if (z != val1 && z == val2) {
219                     better2.receive(wallet2);
220                     better2.receive(wallet1);
221                 } else {
222                     dataProvider.receive(wallet1);
223                     dataProvider.receive(wallet2);
224                 }
225             }
226         }
227         ->End;
228     }
229
230     transaction getTimeManager() returns TimeManager@Shared {
231         return timeManager;
```

```
    }

    transaction getTBefore() returns int {
        return t_before;
    }

    transaction getTAfter() returns int {
        return t_after;
    }
}
```

Listing A.27: `Better.obs`

```
import "Currency.obs"
import "Bet.obs"

main asset contract Better implements BetterInterface {
    Currency@Owned wallet;
    string id;

    string sourceAgreed;
    string aleaAgreed;
    int t_beforeAgreed;
    int t_afterAgreed;
    int amountAgreed;

    Better@Owned(int m, string i,
                 string s, string al, int tb, int ta, int am) {
        wallet = new Currency(m);
        id = i;

        sourceAgreed = s;
        aleaAgreed = al;
        t_beforeAgreed = tb;
        t_afterAgreed = ta;
        amountAgreed = am;
    }

    transaction give(int v) returns Currency@Owned {
        return wallet.split(v); //gift is Owned
    }

    transaction receive(Currency@Owned >> Unowned gift) {
        wallet.merge(gift);
    }

    transaction getAmount() returns int {
        return wallet.getValue();
    }

    transaction id() returns string {
        return id;
    }

    transaction getSourceAgreed() returns string {
        return sourceAgreed;
    }

    transaction getAleaAgreed() returns string {
        return aleaAgreed;
    }

    transaction getTAfterAgreed() returns int {
        return t_afterAgreed;
    }

    transaction getTBeforeAgreed() returns int {
        return t_beforeAgreed;
    }

    transaction getAmountAgreed() returns int {
        return amountAgreed;
    }
}
```

Listing A.28: DataProvider.obs

```
import "Currency.obs"
import "Bet.obs"

main asset contract DataProvider implements DataProviderInterface {
    Currency@Owned wallet;
    string id;

    string sourceAgreed;
    string aleaAgreed;
    int t_afterAgreed;

    DataProvider@Owned(int m, string i,
                       string s, string a, int t) {
        wallet = new Currency(m);
        id = i;

        sourceAgreed = s;
        aleaAgreed = a;
        t_afterAgreed = t;
    }

    transaction give(int v) returns Currency@Owned {
        return wallet.split(v); //gift is Owned
    }

    transaction receive(Currency@Owned >> Unowned gift) {
        wallet.merge(gift);
    }

    transaction getAmount() returns int {
        return wallet.getValue();
    }

    transaction id() returns string {
        return id;
    }

    transaction getSourceAgreed() returns string {
        return sourceAgreed;
    }

    transaction getAleaAgreed() returns string {
        return aleaAgreed;
    }

    transaction getTAfterAgreed() returns int {
        return t_afterAgreed;
    }
}
```

Listing A.29: TimeManager.obs

```
import "Dict.obs"
import "List.obs"
import "Integer.obs"

interface EventInterface {
    transaction action();
}

main asset contract TimeManager {
    Dict[Integer,List[EventInterface]@Owned]@Owned registry;
    int clock;

    state Active;
    state Inactive;

    TimeManager@Inactive() {
        clock = 0;
```

```
        registry =
            new Dict[Integer,List[EventInterface]@Owned](
                                                new IntegerComparator());
        ->Inactive;
    }

    transaction register(int t, EventInterface@Owned >> Unowned event) {
        Integer time = new Integer(t);
        Option[List[EventInterface]] eventList = registry.remove(time);

        if (eventList in None) {
            List[EventInterface] l = new List[EventInterface]();
            l.push(event);
            registry.insert(time,l);
        } else {
            List[EventInterface] l = eventList.unpack();
            l.push(event);
            registry.insert(time,l);
        }
    }

    transaction tick(TimeManager@Active this) {
        Integer time = new Integer(clock);
        Option[List[EventInterface]] events = registry.remove(time);

        if (events in Some) {
            performActions(events.unpack());
        }

        clock = clock + 1;
    }

    private transaction performActions(List[EventInterface]@Owned events) {

        if (events in HasNext){
            EventInterface ev = events.pop();
            ev.action();
            performActions(events);
        }
    }

    transaction start(TimeManager@Inactive >> Active this) {
        ->Active;
    }

    transaction now() returns int {
        return clock;
    }

}
```

Listing A.30: `Dict.obs`

```
// Taken from https://github.com/mcoblenz/Obsidian/tree/master/resources/
    demos/ERC20/Dict.obs

import "Comparator.obs"

contract Option[asset T@s] {
    state None;
    asset state Some {
        T@s val;
    }

    Option@None() {
        ->None;
    }

    Option@Some(T@s >> Unowned v) {
        ->Some(val = v);
    }

    transaction unpack(Option[T@s]@Some >> None this) returns T@s {
        T result = val;
```

```
        ->None;
        return result;
    }
}

main asset contract Dict[KeyType, asset ValueType@s where s is Owned] {
    DictImpl[KeyType, ValueType@s]@(Empty | HasNext) dictImpl;

    Dict@Owned(Comparator@Unowned _comparator) {
        dictImpl = new DictImpl[KeyType, ValueType@s](_comparator);
    }

    transaction replace(Dict@Unowned this,
                        KeyType@Unowned _key,
                        ValueType@s >> Unowned _value)
                        returns Option[ValueType@s]@Owned {
        return dictImpl.replace(_key, _value);
    }

    transaction remove(Dict@Unowned this, KeyType@Unowned _key)
    returns Option[ValueType@s]@Owned {
        return dictImpl.remove(_key);
    }

    transaction peek(Dict@Unowned this, KeyType@Unowned _key)
    returns Option[ValueType@Unowned]@Owned {
        return dictImpl.peek(_key);
    }

    transaction insert(Dict@Unowned this,
                       KeyType@Unowned _key,
                       ValueType@s >> Unowned _value) {
        Option[ValueType@s] existingValue = replace(_key, _value);
        if (existingValue in Some) {
            revert ("insert operation is only permitted when there is no
                 existing value for the given key.");
        }
    }
}

contract DictImpl[KeyType, asset ValueType@s where s is Owned] {
    state Empty;
    asset state HasNext;

    DictImpl[KeyType, ValueType@s]@(Empty | HasNext) next available in
         HasNext;
    KeyType@Unowned key available in HasNext, PrivateHasKeyAndValue;
    ValueType@s value available in HasNext, PrivateHasKeyAndValue,
                                    PrivateHasValue;

    Comparator[KeyType]@Unowned comparator;

    asset state PrivateHasKeyAndValue;
    asset state PrivateHasValue;

    DictImpl@Empty(Comparator@Unowned _comparator) {
        comparator = _comparator;
        ->Empty;
    }

    // Puts the given key/value pair into the DictImplionary.
    // If the key was already in the DictImplionary,
    // returns an Option containing the old value.
    // Otherwise, returns None.
    transaction replace(DictImpl@(Empty | HasNext) this,
                        KeyType@Unowned _key,
                        ValueType@s >> Unowned _value)
                        returns Option[ValueType@s]@Owned {
        switch this {
            case HasNext {
                if (comparator.equals(key, _key)) {
                    ValueType oldValue = value;
                    value = _value;
                    return new Option[ValueType@s](oldValue);
```

```
93                  }
94                  else {
95                      return next.replace(_key, _value);
96                  }
97              }
98              case Empty {
99                  ->HasNext(key = _key,
100                            value = _value,
101                            next =
102                                  new DictImpl[KeyType, ValueType@s](comparator
                                        ));
103                 return new Option[ValueType@s]();
104             }
105         }
106     }
107
108     // Attempts to remove the key/value pair for the given key,
109     // returning the value. If the key is not found, returns None.
110     transaction remove(DictImpl@(Empty | HasNext) this, KeyType@Unowned _key)
111     returns Option[ValueType@s]@Owned {
112         switch this {
113             case HasNext {
114                 if (comparator.equals(key, _key)) {
115
116                     ValueType oldValue = value;
117                     if (next in HasNext) {
118                         DictImpl[KeyType, ValueType@s] newNext =
119                                                                 next.extractNext
                                                                     ();
120                         KeyType newKey = next.extractKey();
121                         ValueType newValue = next.extractValue();
122                         // next is now Empty, so we can discard it implicitly
                                .
123
124                         ->HasNext(next = newNext,
125                                   key = newKey,
126                                   value = newValue);
127                     } else {
128                         // next is already Empty. We're going to be empty too
                                .
129                         ->Empty;
130                     }
131
132                     return new Option[ValueType@s](oldValue);
133
134                 }
135                 else {
136                     return next.remove(_key);
137                 }
138             }
139             case Empty {
140                 return new Option[ValueType@s]();
141             }
142         }
143     }
144
145     private
146     transaction extractNext(DictImpl@HasNext >> PrivateHasKeyAndValue this)
147     returns DictImpl@(HasNext | Empty) {
148         DictImpl[KeyType, ValueType@s] result = next;
149         ->PrivateHasKeyAndValue;
150         return result;
151     }
152
153     private
154     transaction extractKey(
155                     DictImpl@PrivateHasKeyAndValue >> PrivateHasValue this)
156                     returns KeyType@Unowned {
157         KeyType result = key;
158         ->PrivateHasValue;
159         return result;
160     }
161
162     private
```

```
    transaction extractValue(DictImpl@PrivateHasValue >> Empty this)
    returns ValueType@s {
        ValueType result = value;
        ->Empty;
        return result;
    }

    transaction peek(DictImpl@Unowned this, KeyType@Unowned _key)
    returns Option[ValueType@Unowned]@Owned {
         switch this {
            case HasNext {
                if (comparator.equals(key, _key)) {
                    return new Option[ValueType@Unowned](value);
                } else {
                    return next.peek(_key);
                }
            }
            case Empty {
                return new Option[ValueType@Unowned]();
            }
            // These additional cases are here so that
            // peek can be called with unowned references.
            case PrivateHasValue {
                revert "Do not call peek on inconsistent DictImplionaries.";
            }
            case PrivateHasKeyAndValue {
                revert "Do not call peek on inconsistent DictImplionaries.";
            }
        }
    }
}
```

Listing A.31: `List.obs`

```
// Implementation of a FIFO list
main asset contract List[ValueType] {
    state Empty;
    state HasNext {
        ValueType@Owned info;
        List@Owned tail;
    };

    List@Empty() {
        ->Empty;
    }

    transaction get(List@Owned this, int index) returns ValueType@Unowned {
        if (this in Empty) {
            revert("Index out of bounds");
        }

        [this@HasNext];

        if (index < 0) {
            revert("Index out of bounds");
        }

        if (index == 0) {
            return info;
        }

        return tail.get(index - 1);
    }

    transaction push(List@Owned this, ValueType@Owned >> Unowned element) {
        if (this in Empty) {
            ->HasNext(info = element, tail = new List[ValueType]());
            return;
        }

        tail.push(element);
    }

    transaction pop(List@Owned this) returns ValueType@Owned {
```

```
        if (this in Empty) {
            revert("List empty");
        }

        [this@HasNext];

        ValueType res = info;
        if (tail in HasNext){
            info = tail.pop();
        } else {
            [tail@Empty];
            disown tail;
            ->Empty;
        }
        return res;
    }
}
```

Listing A.32: `Integer.obs`

```
// Taken from https://github.com/mcoblenz/Obsidian/tree/master/resources/
    demos/ERC20/Integer.obs

import "Comparator.obs"

main contract Integer {
    int value;

    Integer@Owned(int _value) {
        value = _value;
    }

    transaction getValue() returns int {
        return value;
    }
}


contract IntegerComparator implements Comparator[Integer] {
    transaction equals(Integer@Unowned a, Integer@Unowned b) returns bool {
        return a.getValue() == b.getValue();
    }
}
```

Listing A.33: `Comparator.obs`

```
// Taken from https://github.com/mcoblenz/Obsidian/tree/master/resources/
    demos/ERC20/Comparator.obs

interface Comparator[KeyType] {
    transaction equals(KeyType@Unowned a, KeyType@Unowned b) returns bool;
}
```

Listing A.34: `Better1Main.obs`

```
import "Better.obs"
import "Bet.obs"
import "IO.obs"
import "TimeManager.obs"

main contract Better1Main {

    transaction main(remote Bet@Shared bet,
                     remote Better@Shared better1) {
        if (bet in Init) {
            bet.place_bet1(better1,"objCode",better1.give(5));
            TimeManager timeManager = bet.getTimeManager();
            timeManager.register(
                bet.getTBefore(),
                new EventFirst(bet,better1));
        }
```

```
17
18          waitStateEnd(bet);
19          // Here better1.wallet will store also the amount
20          // won with the bet
21      }
22
23      private transaction waitStateEnd(Bet@Shared b) returns Bet@Shared {
24          if (b in End){
25              return b;
26          }
27          return waitStateEnd(b);
28      }
29  }
```

Listing A.35: Better2Main.obs

```
1   import "Better.obs"
2   import "Bet.obs"
3   import "IO.obs"
4   import "TimeManager.obs"
5
6   main contract Better2Main {
7
8       transaction main(remote Bet@Shared betContract,
9                        remote Better@Shared better2) {
10
11          waitStateFirst(betContract);
12
13          if (betContract in First) {
14              betContract.place_bet2(better2,"objCode",better2.give(5));
15              TimeManager timeManager = betContract.getTimeManager();
16              timeManager.register(
17                  betContract.getTAfter(),
18                  new EventSecond(betContract,better2));
19          }
20
21          waitStateEnd(betContract);
22          // Here better2.wallet will store also the amount
23          // won with the bet
24      }
25
26      private transaction waitStateFirst(Bet@Shared bet) returns Bet@Shared {
27          if (bet in First){
28              return bet;
29          }
30          return waitStateFirst(bet);
31      }
32
33      private transaction waitStateEnd(Bet@Shared bet) returns Bet@Shared {
34          if (bet in End){
35              return bet;
36          }
37          return waitStateEnd(bet);
38      }
39  }
```

Listing A.36: DataProviderMain.obs

```
1   import "DataProvider.obs"
2   import "Bet.obs"
3   import "IO.obs"
4
5   main contract DataProviderMain {
6
7       transaction main(remote Bet@Shared betContract,
8                        remote DataProvider@Shared dataProvider) {
9
10          waitStateRun(betContract);
11
12          string source = "getSourceSomewhere";
13          string alea = "getAleatoryEventSomewhere";
14          string outcome = "getEventOutcome";
```

```
15
16          if (betContract in Run){
17              betContract.data(dataProvider,source,alea,outcome);
18          }
19      }
20
21      private transaction waitStateRun(Bet@Shared bet) returns Bet@Shared {
22          if (bet in Run){
23              return bet;
24          }
25          return waitStateRun(bet);
26      }
27  }
```

Listing A.37: TimeManagerMain.obs

```
1   import "Bet.obs"
2
3   main contract TimeManagerMain {
4
5       transaction main(remote Bet@Shared betContract) {
6
7           TimeManager tm = betContract.getTimeManager();
8           if (tm in Inactive) {
9               tm.start();
10          }
11
12          if (tm in Active){
13              keepTicking(tm);
14          }
15
16      }
17
18      private transaction keepTicking(TimeManager@Owned tm) {
19          if (tm in Active){
20              tm.tick();
21              keepTicking(tm);
22          }
23      }
24  }
```

A.6 Auction

A.6.1 Stipula

Listing A.38: auction.stipula

```
stipula Auction {
    asset wallet, item
    field objectCode, maxBidder
    init Init

    agreement (Seller, Bidder1, Bidder2, Auctioneer)(objectCode) {
        Seller, Bidder1, Bidder2 : objectCode
    } ==> @Init

    @Init Seller : offer(obj)[token] (obj == objectCode) {
        "none" -> maxBidder
        token -o item;
        _
    } ==> @Open

    @Open Bidder1 : makeBid()[bid] (bid > wallet && maxBidder != "one") {
        if (maxBidder == "two") {
            wallet -o Bidder2
        }
        bid -o wallet
        "one" -> maxBidder;
        _
    } ==> @Open

    @Open Bidder2 : makeBid()[bid] (bid > wallet && maxBidder != "two") {
        if (maxBidder == "one") {
            wallet -o Bidder1
        }
        bid -o wallet
        "two" -> maxBidder;
        _
    } ==> @Open

    @Open Auctioneer : stopBidding()[] {
        "End" -> Bidder1
        "End" -> Bidder2;
        _
    } ==> @BiddingDone

    @BiddingDone Auctioneer : giveItem()[] {
        if (maxBidder == "one") {
            item -o Bidder1
            wallet -o Seller
        } else if (maxBidder == "two") {
            item -o Bidder2
            wallet -o Seller
        } else { //maxBidder == "none" and wallet == 0
            item -o Seller
        };
        _
    } ==> @End
}
```

A.6.2 Obsidian

Listing A.39: Auction.obs

```
contract Bidder {
    string name;
    int balance;

    state WonItem {
        Item@Owned item;
```

```
    }
    state Bidding {
        int bidAmount;
    }

    Bidder@Bidding(string n, int m, int b) {
        name = n;
        balance = m;
        ->Bidding(bidAmount = b);
    }

    transaction createBid(Bidder@Bidding this) returns Bid@Owned {
        Bid bid = new Bid(bidAmount);
        balance = balance - bidAmount;
        return bid;
    }

    transaction getBidAmount(Bidder@Bidding this) returns int {
        return bidAmount;
    }

    transaction getName(Bidder@Owned this) returns string {
        return name;
    }

    transaction won(Bidder@Bidding >> WonItem this, Item@Owned >> Unowned i)
        {
        -> WonItem(item = i);
    }

    transaction returnBidMoney(Bidder@Owned this, Bid@Owned bid) {
        balance = balance + bid.getAmount();
    }

    transaction updateBidAmount(Bidder@Bidding this, int amount) {
        bidAmount = amount;
    }
}

contract Seller {
    state SoldItem {
        Bid@Owned bid;
    }
    state Unsold {
        Item@Owned item;
    }

    Seller@Unsold() {
        ->Unsold(item = new Item());
    }

    transaction receiveBid(Seller@Unsold >> SoldItem this,
                           Bid@Owned >> Unowned b) {
        -> SoldItem(bid = b);
    }

    transaction returnItem(Seller@SoldItem >> Unsold this,
                           Item@Owned >> Unowned i) {
        ->Unsold(item = i);
    }
}

contract Item {}

contract Bid {
    int amount;

    Bid@Owned(int num) {
        amount = num;
    }

    transaction getAmount(Bid@Owned this) returns int {
        return amount;
    }
```

```
}

main contract Auction {

    state Open {
        Item@Owned item;
        Bid@Owned bid;
    }
    state BiddingDone {
        Item@Owned it;
        Bid@Owned finalBid;
    }
    state Closed {
        Seller@SoldItem sellerSatisfied;
        Bidder@WonItem winner;
    }

    Seller@Unsold seller available in Open, BiddingDone;
    Bidder@Bidding maxBidder available in Open, BiddingDone;

    Auction@Owned(Item@Owned >> Unowned i) {
        Open::maxBidder = new Bidder("none", 0, 0);
        Open::seller = new Seller();
        ->Open(item = i, bid = new Bid(0));
    }

    transaction makeBid(Auction@Open this, Bidder@Bidding >> Unowned bidder)
        {
        if (bidder.getBidAmount() > bid.getAmount()) {
            if (maxBidder.getName() != "none") {
                if (bid in Open) {
                    maxBidder.returnBidMoney(bid);
                }
            }

            bid = bidder.createBid();
            maxBidder = bidder;
        }
    }

    transaction finishBidding(Auction@Open >> BiddingDone this) {
        -> BiddingDone(it = item, finalBid = bid);
    }

    transaction giveItem(Auction@BiddingDone >> Closed this) {
        maxBidder.won(it);
        seller.receiveBid(finalBid);
        ->Closed(sellerSatisfied = seller, winner = maxBidder);
    }
}
```

A.7 Parametric Insurance

A.7.1 Stipula

Listing A.40: insuranceservice_arrays.stipula

```
stipula InsuranceService {
    asset policies, escrow
    field payouts,
          costs,
          conditions,
          expTimes,
          actives,
          expirations,
          claimables,
          n
    init Ready

    agreement (Farmer, Insurer, WeatherService)
              (payouts, conditions, expTimes) {
        Insurer : payouts, costs, conditions, expTimes
    } ==> @Ready

    @Ready Insurer : offer(k)[p,e]
    ((k >= 0 || k < payouts.length) && p > 0 && e == payouts[k]*p) {
        p -o policies[k]
        e -o escrow
        conditions[k] -> WeatherService;
        _
    } ==> @Ready

    @Ready Farmer : buy(k)[m]
    ((k >= 0 && k < policies.length) && policies[k] > 0 && costs[k] == m) {
        1 -o policies[k], Farmer
        m -o Insurer
        actives[k] + 1 -> actives[k]

        now + expTimes[k] >> @Ready@PolicyExpired {
            if (k < n) {
                k -> n
            }
            expirations[k] + 1 -> expirations[k]
            actives[k] - 1 -> actives[k]
            payouts[k] -o escrow, Insurer
        } ==> @PolicyExpired
    } ==> @Ready

    @Ready WeatherService : conditionsMet(k)[] {
        if (actives[k] > 0) {
            claimables[k] + 1 -> claimables[k]
            actives[k] - 1 -> actives[k]
        };
        _
    } ==> @Ready

    @Cashing Farmer : claim(k)[p] (p <= claimables[k]) {
        p -o Insurer
        (payouts[k] * p) -o escrow, Farmer
        claimables[k] - 1 -> claimables[k];
        _
    } ==> @Ready

    @PolicyExpired Farmer : returnExpiredPolicy(k)[p] (p <= expirations[k]) {
        p -o Insurer
        expirations[k] - p -> expirations[k];
        _
    } ==> @PolicyExpired

    @PolicyExpired Insurer : checkExpiredReturns()[] (expirations[n] == 0) {
        n + 1 -> n;
        _
    } ==> @PolicyExpired

```

```
    @PolicyExpired Insurer : ready()[] (n == policies.length) {
        0 -> n;
        _
    } ==> @Ready
}
```

Listing A.41: `policy.stipula`

```
stipula Policy {
    asset escrow
    field cost, expirationTime,
          longitude, latitude, radius,
          moistureContent, payout
    init Offered

    agreement (Farmer, InsuranceService, WeatherService)
              (cost, expirationTime,
               longitude, latitude, radius,
               moistureContent, payout) {
        Farmer, InsuranceService : cost, expirationTime,
                                   longitude, latitude, radius,
                                   moistureContent, payout
    } ==> @Offered

    @Offered Farmer : buy()[m] (m == cost) {
        m -o InsuranceService; // cost -o m, InsuranceService
        _
    } ==> @Activating

    @Activating InsuranceService : activate()[m] (m == payout) {
        m -o escrow;
        now + expirationTime >> @Active {
            escrow -o InsuranceService
        } ==> @Expired
    } ==> @Active

    @Active WeatherService : checkMoist(moist)[] (moist < moistureContent) {
        ;
        _
    } ==> @Claimable

    @Claimable Farmer : claim()[] {
        escrow -o Farmer;
        _
    } ==> @Claimed
}
```

A.7.2 Obsidian

Available in the Obsidian official repository [2].

A.8 ExampleTokenBank

A.8.1 Stipula

Listing A.42: `exampletokenbank_arrays.stipula`

```
stipula ExampleTokenBank {
    asset balances
    field allowances
    init Working

    agreement(Owner1, Owner2, Bank)(allowances) {
        Bank : allowances
    } ==> @Working

    @Working Bank : transfer(fromAddress, toAddress, value)[]
    (value <= balance[fromAddress]) {
        value -o balances[fromAddress],balances[toAddress];
        _
    } ==> @Working

    @Working Bank : approve(ownerAddress, fromAddress, value)[]
    ((ownerAddress == 0 || ownerAddress == 1)
     && (fromAddress >= 0 && fromAddress < balances.length)) {
        value -> allowances[ownerAddress][fromAddress];
        _
    } ==> @Working

    @Working Owner1 : transferFrom(fromAddr,toAddr,value)[]
    ((fromAddr >= 0 && fromAddr <= balances.length)
     && (toAddr >= 0 && toAddr <= balances.length)
     && allowances[0][fromAddr] >= value) {
        value -o balances[fromAddr],balances[toAddr];
        _
    } ==> @Working

    @Working Owner2 : transferFrom(fromAddr,toAddr,value)[]
    ((fromAddr >= 0 && fromAddr <= balances.length)
     && (toAddr >= 0 && toAddr <= balances.length)
     && allowances[1][fromAddr] >= value) {
        value -o balances[fromAddr],balances[toAddr];
        _
    } ==> @Working

}
```

A.8.2 Obsidian

Available in the Obsidian official repository [2].

Listing A.43: `ERC20.obs`

```
import "Dict.obs"
import "Integer.obs"

asset interface ObsidianToken {
    transaction getValue() returns int;
    transaction merge(ObsidianToken@Owned >> Unowned other);
    transaction split(int val) returns ObsidianToken@Owned;
}

asset contract ExampleToken implements ObsidianToken {
    int value;

    ExampleToken@Owned(int v) {
        value = v;
    }

    transaction getValue(ExampleToken@Unowned this) returns int {
        return value;
```

```
19      }
20
21      transaction merge(ObsidianToken@Owned >> Unowned other) {
22          value = value + other.getValue();
23          disown other;
24      }
25
26      transaction split(ExampleToken@Owned this, int val) returns ExampleToken@
            Owned {
27          if (val > value) {
28              revert ("Can't split off more than the existing value");
29          }
30          ExampleToken other = new ExampleToken(val);
31          value = value - val;
32          return other;
33      }
34
35  }
36
37  // ERC20 has been slightly adapted for Obsidian, since Obsidian does not have
        a built-in authentication mechanism.
38  asset interface ERC20 {
39      transaction totalSupply() returns int;
40      transaction balanceOf(int ownerAddress) returns int;
41      transaction transfer(int fromAddress, int toAddress, int value) returns
            bool;
42
43      // - allow ownerAddress to withdraw from your account, multiple times, up
            to the value amount.
44      transaction approve(int ownerAddress, int fromAddress, int value) returns
            bool;
45
46      // Returns the amount of allowance still available.
47      transaction allowance(int ownerAddress, int fromAddress) returns int;
48
49      // Transfers tokens from an allowance that has already been granted.
50      transaction transferFrom(int senderAddress, int fromAddress, int
            toAddress, int value) returns bool;
51  }
52
53  main asset contract ExampleTokenBank implements ERC20 {
54      int totalSupply;
55      Dict[Integer, ExampleToken]@Owned balances;
56
57      // map from fromAddress to (map from spender to amount)
58      Dict[Integer, Dict[Integer, Integer]@Owned]@Owned allowed;
59
60      ExampleTokenBank@Owned() {
61          totalSupply = 0;
62          balances = new Dict[Integer, ExampleToken@Owned](new
                IntegerComparator());
63          allowed = new Dict[Integer, Dict[Integer, Integer]@Owned](new
                IntegerComparator());
64      }
65
66      transaction totalSupply() returns int {
67          return totalSupply;
68      }
69
70      transaction balanceOf(int ownerAddress) returns int {
71          Option[ExampleToken@Unowned] balance = balances.peek(new Integer(
                ownerAddress));
72          if (balance in None) {
73              return 0;
74          }
75          else {
76              return balance.unpack().getValue();
77          }
78      }
79
80      transaction transfer(int fromAddress, int toAddress, int value) returns
            bool {
81          Integer fromIntegerAddress = new Integer(fromAddress);
82          Option[ExampleToken@Owned] fromBalance = balances.remove(
```

```
            fromIntegerAddress);
        if (fromBalance in None) {
            return false;
        }
        else {
            ExampleToken fromTokens = fromBalance.unpack();
            if (value <= fromTokens.getValue()) {
                Integer toIntegerAddress = new Integer(toAddress);
                Option[ExampleToken@Owned] toBalance = balances.remove(
                    toIntegerAddress);
                ExampleToken toTokens;
                if (toBalance in Some) {
                    toTokens = toBalance.unpack();
                }
                else {
                    toTokens = new ExampleToken(0); // 0 value
                }

                ExampleToken tokensToMove = fromTokens.split(value);
                toTokens.merge(tokensToMove);
                balances.insert(toIntegerAddress, toTokens); // OK because we
                     just removed it.
                balances.insert(fromIntegerAddress, fromTokens); // OK
                    because we just removed it.

                return true;
            }
            else {
                // Insufficient funds available.
                balances.insert(fromIntegerAddress, fromTokens); // // OK
                    because we just removed it.
                return false;
            }
        }
    }

    // Records a new allowance. Replaces any previous allowance.
    transaction approve(int ownerAddress, int fromAddress, int value) returns
         bool {
        Integer ownerAddressInteger = new Integer(ownerAddress);
        Option[Dict[Integer, Integer]@Owned] ownerAllowancesOption = allowed.
            remove(ownerAddressInteger);

        Dict[Integer, Integer] ownerAllowances;
        if (ownerAllowancesOption in None) {
            ownerAllowances = new Dict[Integer, Integer@Owned](new
                IntegerComparator());
        }
        else {
            ownerAllowances = ownerAllowancesOption.unpack();
        }

        Option[Integer@Owned] oldAllowance = ownerAllowances.replace(new
            Integer(fromAddress), new Integer(value));
        allowed.insert(ownerAddressInteger, ownerAllowances);
        disown oldAllowance; // Options are assets because they CAN hold
            assets, but this one doesn't happen to do so.
        return true;
    }

    transaction allowance(int ownerAddress, int fromAddress) returns int {
        Option[Dict[Integer, Integer]@Unowned] ownerAllowancesOption =
            allowed.peek(new Integer(ownerAddress));
        switch (ownerAllowancesOption) {
            case None {
                return 0;
            }
            case Some {
                Dict[Integer, Integer@Owned] ownerAllowances =
                    ownerAllowancesOption.unpack();
                Option[Integer@Unowned] spenderAllowance = ownerAllowances.
                    peek(new Integer(fromAddress));
                if (spenderAllowance in None) {
                    return 0;
```

```
144                 }
145                 else {
146                     return spenderAllowance.unpack().getValue();
147                 }
148             }
149         }
150     }
151
152     // senderAddress wants to transfer value tokens from fromAddress to
        toAddress.
153     // This requires that an allowance have been set up in advance and that
        fromAddress has enough tokens.
154     transaction transferFrom(int senderAddress, int fromAddress, int
        toAddress, int value) returns bool
155     {
156         int allowance = allowance(senderAddress, fromAddress);
157         if (allowance >= value) {
158             int newAllowance = allowance - value;
159             bool transferSucceeded = transfer(fromAddress, toAddress, value);
160             if (!transferSucceeded) {
161                 // Perhaps not enough tokens were available to transfer.
162                 return false;
163             }
164             approve(senderAddress, fromAddress, newAllowance);
165
166             return true;
167         }
168         else {
169             return false;
170         }
171     }
172 }
```

Appendix B

Proofs for Silica transactions

B.1 Moving a token between variables

The expression to check is `h' := h` where `h` : NFT@Owned and `h'` : NFT@Owned. Then, we have to prove the judgement for any Γ and Δ

$$\begin{array}{c} \Gamma; \Delta, \\ \texttt{h'} : \texttt{NFT@Unowned}, \\ \texttt{h} : \texttt{NFT@Owned} \end{array} \vdash_s \texttt{h':= h} : \mathsf{unit} \dashv \begin{array}{c} \Delta, \\ \texttt{h'} : \texttt{NFT@Owned}, \\ \texttt{h} : \texttt{NFT@Unowned} \end{array}$$

Proof

$$\dfrac{\dfrac{\texttt{NFT} = \mathit{contract}(\texttt{NFT@Owned})}{\texttt{NFT@Owned} \Rrightarrow \texttt{NFT@Owned/NFT@Unowned}}\ \textsc{Split-Unowned} \qquad \dfrac{\dfrac{}{\mathit{notOwned}(\texttt{NFT@Unowned})}}{\Gamma \vdash \mathit{disposable}(\texttt{NFT@Unowned})}}{\begin{array}{c} \Gamma; \Delta, \\ \texttt{h'} : \texttt{NFT@Unowned}, \\ \texttt{h} : \texttt{NFT@Owned} \end{array} \vdash_s \texttt{h':= h} : \mathsf{unit} \dashv \begin{array}{c} \Delta, \\ \texttt{h'} : \texttt{NFT@Owned}, \\ \texttt{h} : \texttt{NFT@Unowned} \end{array}}\ \textsc{T-Assign}$$

B.2 Sending a token to a party

The expression to check is `A.receiveToken(h)` where `A` : Party@NoToken and `h` : NFT@Owned.

To complete the proof we need also the proof for the transaction `receiveToken`.

B.2.1 Statement

The expression to check is `A.receiveToken(h)` where `A` : Party@NoToken and `h` : NFT@Owned. Then, we have to prove the judgement for any Γ and Δ

$$\begin{array}{c} \Gamma; \Delta, \\ \texttt{A} : \texttt{Party@NoToken}, \\ \texttt{h} : \texttt{NFT@Owned} \end{array} \vdash_s \texttt{A.receiveToken(h)} : \mathit{unit} \dashv \begin{array}{c} \Delta, \\ \texttt{A} : \texttt{Party@Token}, \\ \texttt{h} : \texttt{NFT@Unowned} \end{array}$$

Proof

$$\dfrac{\pi_1 \quad \pi_2 \quad \pi_3 \quad \pi_4 \quad \pi_5 \quad \pi_6 \quad \pi_7}{\begin{array}{c} \Gamma; \Delta, \\ \texttt{A} : \texttt{Party@NoToken}, \\ \texttt{h} : \texttt{NFT@Owned} \end{array} \vdash_s \texttt{A.receiveToken(h)} : \mathit{unit} \dashv \begin{array}{c} \Delta, \\ \texttt{A} : \texttt{Party@Token}, \\ \texttt{h} : \texttt{NFT@Unowned} \end{array}}\ \textsc{T-Inv}$$

$$\frac{\texttt{Party@NoToken} \neq \texttt{Unowned}}{\texttt{Party@Token} = funcArg(\texttt{Party@NoToken}, \texttt{Party@NoToken}, \texttt{Party@NoToken})}\ \textsc{funcArg-other}$$
π_1

$$\frac{\texttt{NFT@Owned} \neq \texttt{Unowned}}{\texttt{NFT@Unowned} = funcArg(\texttt{NFT@Owned}, \texttt{NFT@Owned}, \texttt{NFT@Unowned})}\ \textsc{funcArg-other}$$
π_2

$$\frac{}{\Gamma \vdash \mathsf{NoToken} <:_* \mathsf{NoToken}}\ <:_*\textsc{-refl} \qquad \frac{\overline{\Gamma \vdash bound_*(\mathsf{NoToken}) = \mathsf{NoToken}}}{\Gamma \vdash bound(\texttt{Party@NoToken}) = \texttt{Party@NoToken}}$$
π_3 π_7

$$\frac{\overline{\Gamma \vdash \mathsf{Owned} <:_* \mathsf{Owned}}\ <:_*\textsc{-refl}}{\Gamma \vdash \texttt{NFT@Owned} <:_* \texttt{NFT@Owned}}\ <:\textsc{-Matching-defs}$$
π_4

$$\texttt{s.wallet}, \texttt{s.tokens} \in \Delta, \texttt{A} : \texttt{Party@NoToken}, \texttt{h} : \texttt{NFT@Owned}$$
π_5

$$\frac{\begin{array}{c}tdef(\texttt{Party}, \texttt{receiveToken}) = \texttt{unit receiveToken(NFT@Owned»Unowned g) S»S \{return ...\}} \\ params(\texttt{unit receiveToken(NFT@Owned»Unowned g) NoToken»Token \{return ...\}}) = \varnothing \\ params(\texttt{Party}) = \varnothing\end{array}}{specializeTrans_\Gamma(\texttt{receiveToken}, \texttt{Party}) = \texttt{unit receiveToken(NFT@Owned»Unowned g) S»S \{return ...\}}}$$
π_6

B.2.2 Transaction `receiveToken`

The transaction to check is

```
unit receiveToken(NFT@Owned»Unowned g) NoToken»Token
                                        { return this →p Token(g) }
```

Then, we have to prove the judgement

```
unit receiveToken(NFT@Owned»Unowned g) NoToken»Token
                                        { return this →p Token(g) }
                                  ok in Party
```

Proof

$$\frac{\pi_1 \qquad params(C) = \varnothing \qquad \Gamma = \varnothing}{\begin{array}{l}\texttt{unit receiveToken(NFT@Owned»Unowned g) NoToken»Token} \\ \texttt{\{ return this →p Token(g) \}}\ \textbf{ok in}\ \texttt{Party}\end{array}}\ \textsc{PublicTransactionOK}$$

$$\frac{\pi_2 \quad \pi_3 \quad \pi_4 \quad \pi_5 \quad \pi_6 \quad p \in \{Shared, Owned\}}{\begin{array}{l}\texttt{this} : \texttt{Party@NoToken}, \\ \texttt{g} : \texttt{NFT@Owned}\end{array} \vdash_{\texttt{this}} \texttt{this} \rightarrow_p \texttt{Token(g)} : \mathsf{unit} \dashv \begin{array}{l}\texttt{this} : \texttt{Party@Token}, \\ \texttt{g} : \texttt{NFT@Unowned}\end{array}}\ \text{T-}\rightarrow_p$$
π_1

$$\begin{array}{l}fieldTypes_{\texttt{this}}(\texttt{this} : \texttt{Party@NoToken}, \texttt{g} : \texttt{NFT@Owned}; \varnothing) = \\ = fieldTypes_{\texttt{this}}(\texttt{this} : \texttt{Party@NoToken}; \varnothing) = \\ = fieldTypes_{\texttt{this}}(\cdot; \varnothing) = \varnothing\end{array}$$
π_2

$$\frac{\overline{\vdash \mathsf{Owned} <:_* \mathsf{Owned}}\ <:_*\textsc{-refl}}{\vdash \texttt{NFT@Owned} <: stateFields(\texttt{Party}, \texttt{Token}) = \texttt{NFT@Owned}}\ <:\textsc{-MatchingDefs}$$
π_3

$$\frac{stateFields(\texttt{Party@NoToken}) = \varnothing}{unionFields(\texttt{Party@NoToken}) = \varnothing}\ \pi_4$$

$$\frac{\dfrac{}{\vdash \mathsf{NoToken} <:_* \mathsf{Owned}}\ <:_*\text{-S-O} \qquad \dfrac{p \neq \mathsf{NoToken}, \mathsf{Token}}{\vdash \mathsf{Owned} <:_* p}\ <:_*\text{-O-*}}{\vdash \mathsf{NoToken} <:_* p}\ <:_*\text{-Trans}\ \pi_5$$

$$\frac{\dfrac{\texttt{NFT} = contract(\texttt{NFT})}{\texttt{NFT@Owned} \Rrightarrow \texttt{NFT@Owned/NFT@Unowned}}\ \textsc{Split-Unowned}}{\begin{array}{c}\texttt{this} : \texttt{Party@NoToken},\\ \texttt{g} : \texttt{NFT@Owned}\end{array} \vdash_{\texttt{this}} \texttt{g} : \texttt{NFT@Owned} \dashv \texttt{g} : \texttt{NFT@Unowned}}\ \text{T-Lookup}\ \pi_6$$

B.3 Sending currency to a variable

The expression to check is

$$\begin{array}{l}\quad\ \text{let } \texttt{x} : \texttt{Currency@Owned} = \texttt{h.split(v)}\\ \text{in } \texttt{h'.merge(x)}\end{array}$$

where $\texttt{h}, \texttt{h'} : \texttt{Currency@Owned}$ and $\texttt{v} : int$.

To complete the proof, we also need proof for transactions `split` and `merge`.

B.3.1 Statement

The expression to check is

$$\begin{array}{l}\quad\ \text{let } \texttt{x} : \texttt{Currency@Owned} = \texttt{h.split(v)}\\ \text{in } \texttt{h'.merge(x)}\end{array}$$

where $\texttt{h}, \texttt{h'} : \texttt{Currency@Owned}$ and $\texttt{v} : int$. Then, we have to prove, for any Γ and Δ, the judgement

$$\begin{array}{c}\Gamma; \Delta,\\ \texttt{h} : \texttt{Currency@Owned},\\ \texttt{h'} : \texttt{Currency@Owned},\\ \texttt{v} : \mathsf{int}\end{array} \vdash_s \begin{array}{l}\text{let } \texttt{x} : \texttt{Currency@Owned} = \texttt{h.split(v)}\\ \text{in } \texttt{h'.merge(x)}\end{array} : \mathsf{unit} \dashv \begin{array}{c}\Delta,\\ \texttt{h} : \texttt{Currency@Owned},\\ \texttt{h'} : \texttt{Currency@Owned},\\ \texttt{v} : \mathsf{int}\end{array}$$

Proof

$$\frac{\pi_1 \qquad \pi_2 \qquad \vdash disposable(\texttt{Currency@Unowned})}{\begin{array}{c}\Gamma; \Delta,\\ \texttt{h} : \texttt{Currency@Owned},\\ \texttt{h'} : \texttt{Currency@Owned},\\ \texttt{v} : \mathsf{int}\end{array} \vdash_s \begin{array}{l}\text{let } \texttt{x} : \texttt{Currency@Owned} = \texttt{h.split(v)}\\ \text{in } \texttt{h'.merge(x)}\end{array} : \mathsf{unit} \dashv \begin{array}{c}\Delta,\\ \texttt{h} : \texttt{Currency@Owned},\\ \texttt{h'} : \texttt{Currency@Owned},\\ \texttt{v} : \mathsf{int}\end{array}}\ \text{T-let}$$

$$\frac{\begin{array}{c}\dfrac{}{\vdash \mathsf{Owned} <:_* \mathsf{Owned}}\ <:_*\text{-refl} \qquad \dfrac{}{\mathsf{int} <: \mathsf{int}}\\ \vdash bound(\texttt{Currency@Owned}) = \texttt{Currency@Owned} \qquad \forall f.s.f \notin \Delta_1\\ specializeTrans_\Gamma(\texttt{split}, \texttt{Currency}) = \texttt{Currency@Owned split(int v) Owned»Owned \{ ... \}}\\ \texttt{Currency@Owned} = funcArg(\texttt{Currency@Owned}, \texttt{Currency@Owned}, \texttt{Currency@Owned})\\ \mathsf{int} = funcArg(\mathsf{int}, \mathsf{int}, \mathsf{int})\end{array}}{\begin{array}{c}\Gamma; \Delta,\\ \Delta_1 = \texttt{h} : \texttt{Currency@Owned},\\ \texttt{h'} : \texttt{Currency@Owned},\\ \texttt{v} : \mathsf{int}\end{array} \vdash_s \texttt{h.split(v)} : \texttt{Currency@Owned} \dashv \begin{array}{c}\Gamma; \Delta,\\ \texttt{h} : \texttt{Currency@Owned},\\ \texttt{h'} : \texttt{Currency@Owned},\\ \texttt{v} : \mathsf{int}\end{array}}\ \text{T-Inv}\ \pi_1$$

$$\dfrac{\dfrac{}{\vdash \texttt{Owned} <:_* \texttt{Owned}}\ {<:_*\text{-REFL}} \qquad \dfrac{\dfrac{}{\vdash \texttt{Owned} <:_* \texttt{Owned}}\ {<:_*\text{-REFL}}}{\vdash \texttt{Currency@Owned} <: \texttt{Currency@Owned}}\ {<:_*\text{-MATCHING-DEFS}}}{\begin{array}{c}\vdash bound(\texttt{Currency@Owned}) = \texttt{Currency@Owned} \qquad \forall f.s.f \notin \Delta_2 \\ specializeTrans_\Gamma(\texttt{merge}, \texttt{Currency}) = \texttt{unit merge(Currency@Owned»Unowned other) Owned»Owned \{ ... \}} \\ \texttt{Currency@Owned} = funcArg(\texttt{Currency@Owned}, \texttt{Currency@Owned}, \texttt{Currency@Owned}) \\ \texttt{Currency@Unowned} = funcArg(\texttt{Currency@Owned}, \texttt{Currency@Owned}, \texttt{Currency@Unowned})\end{array}}\ \text{T-INV}$$

$$\Gamma;\Delta,\ \texttt{h} : \texttt{Currency@Owned},\ \Delta_2 = \texttt{h'} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int},\ \texttt{x} : \texttt{Currency@Owned} \vdash_s \texttt{h'.merge(x)} : \texttt{unit} \dashv \Gamma;\Delta,\ \texttt{h} : \texttt{Currency@Owned},\ \texttt{h'} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int},\ \texttt{x} : \texttt{Currency@Unowned}$$

π_2

B.3.2 Transaction `split`

The transaction to check is

$$\texttt{Currency@Owned split(int v) Owned»Owned \{return } e\texttt{\}}$$

where

$$\begin{aligned} e &= \text{let } \texttt{diff} : \texttt{int} = \texttt{this.value - v} \text{ in } e_0 \\ e_0 &= \text{let } \texttt{em}_0 : \texttt{unit} = \texttt{this.value := diff} \text{ in } e_1 \\ e_1 &= \text{let } \texttt{em}_1 : \texttt{unit} = \texttt{pack} \text{ in } e_2 \\ e_2 &= \text{let } \texttt{result} : \texttt{Currency@Owned} = \texttt{new Currency@Owned(v)} \text{ in } \texttt{result} \end{aligned}$$

Then, we have to prove the judgement

$$\texttt{Currency@Owned split(int v) Owned»Owned \{return } e\texttt{\}} \textbf{ ok in } \texttt{Currency}$$

Proof

$$\dfrac{params(\texttt{Currency}) = \varnothing \qquad \Gamma = \varnothing \qquad \pi_1}{\texttt{Currency@Owned split(int v) Owned»Owned \{return } e\texttt{\}} \textbf{ ok in } \texttt{Currency}}\ \text{PUBLICTRANSACTIONOK}$$

$$\dfrac{\pi_2 \qquad \pi_5 \qquad \vdash disposable(\texttt{int})}{\texttt{this} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int} \vdash_{\texttt{this}} \text{let } \texttt{diff} : \texttt{int} = \texttt{this.value - v} \text{ in } e_0 : \texttt{Currency@Owned} \dashv \texttt{this} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int}}\ \text{T-LET}$$

π_1

$$\dfrac{\pi_6 \qquad \pi_7 \qquad \vdash disposable(\texttt{unit})}{\texttt{this} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int},\ \texttt{this.value} : \texttt{int},\ \texttt{diff} : \texttt{int} \vdash_{\texttt{this}} \text{let } \texttt{em}_0 : \texttt{unit} = \texttt{this.value := diff} \text{ in } e_1 : \texttt{Currency@Owned} \dashv \texttt{this} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int},\ \texttt{diff} : \texttt{int}}\ \text{T-LET}$$

π_5

$$\dfrac{\pi_{10} \qquad \pi_{11} \qquad \vdash disposable(\texttt{unit})}{\texttt{this} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int},\ \texttt{this.value} : \texttt{int},\ \texttt{em}_0 : \texttt{unit},\ \texttt{diff} : \texttt{int} \vdash_{\texttt{this}} \text{let } \texttt{em}_1 : \texttt{unit} = \texttt{pack} \text{ in } e_2 : \texttt{Currency@Owned} \dashv \texttt{this} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int},\ \texttt{em}_0 : \texttt{unit},\ \texttt{diff} : \texttt{int}}\ \text{T-LET}$$

π_7

$$\frac{\pi_{12} \quad \pi_{13} \quad \vdash \mathit{disposable}(\texttt{Currency@Unowned})}{\texttt{this : Currency@Owned, v : int, em}_0\texttt{ : unit, em}_1\texttt{ : unit, diff : int} \vdash_{\texttt{this}} \texttt{let result : Currency@Owned = new Currency@Owned(v) in result : Currency@Owned} \dashv \texttt{this : Currency@Owned, v : int, em}_0\texttt{ : unit, em}_1\texttt{ : unit, diff : int}}\ \text{T-LET}$$

π_{11}

$$\frac{\frac{\texttt{Currency} = \mathit{contract}(\texttt{Currency@Owned})}{\texttt{Currency@Owned} \Rrightarrow \texttt{Currency@Owned/Currency@Unowned}}\ \text{Split-unowned}}{\texttt{this : Currency@Owned, v : int, em}_0\texttt{ : unit, em}_1\texttt{ : unit, diff : int, result : Currency@Owned} \vdash_{\texttt{this}} \texttt{result : Currency@Owned} \dashv \texttt{this : Currency@Owned, v : int, em}_0\texttt{ : unit, em}_1\texttt{ : unit, diff : int, Currency@Unowned}}\ \text{T-Lookup}$$

π_{13}

$$\frac{\frac{\texttt{int} \Rrightarrow \texttt{int/int}}{\Delta_{12} \vdash \texttt{v : int} \dashv \Delta'_{12}}\ \text{T-Lookup} \quad \frac{}{\vdash \texttt{int} <: \texttt{int}} \quad \mathit{def}(\texttt{Currency}) = \texttt{contract Currency \{ ... \}}}{\Delta_{12} = \texttt{this : Currency@Owned, v : int, em}_0\texttt{ : unit, em}_1\texttt{ : unit, diff : int} \vdash_{\texttt{this}} \texttt{new Currency@Owned(v) : Currency@Owned} \dashv \texttt{this : Currency@Owned, v : int, em}_0\texttt{ : unit, em}_1\texttt{ : unit, diff : int} = \Delta'_{12}}\ \text{T-new}$$

π_{12}

$$\frac{\mathit{contractFields}(\texttt{Currency@Owned}) \triangleq \triangleq \mathit{intersectFields}(\texttt{Currency}) = \{\texttt{int value}\} \quad \vdash \texttt{int} <: \texttt{int} \quad \frac{}{\vdash \texttt{int} \approx \texttt{int}}\ \approx\text{-Refl}}{\texttt{this : Currency@Owned, v : int, this.value : int, em}_0\texttt{ : unit, diff : int} \vdash_{\texttt{this}} \texttt{pack : unit} \dashv \texttt{this : Currency@Owned, v : int, em}_0\texttt{ : unit, diff : int}}\ \text{T-pack}$$

π_{10}

$$\frac{\pi_8 \quad \pi_9 \quad \vdash \mathit{disposable}(\texttt{int})}{\texttt{this : Currency@Owned, v : int, this.value : int, diff : int} \vdash_{\texttt{this}} \texttt{this.value := this.value - v : unit} \dashv \texttt{this : Currency@Owned, v : int, this.value : int, diff : int}}\ \text{T-FieldUpdate}$$

π_6

$$\frac{\texttt{int} \Rrightarrow \texttt{int/int}}{\texttt{this : Currency@Owned, v : int, this.value : int, diff : int} \vdash_{\texttt{this}} \texttt{diff : int} \dashv \texttt{this : Currency@Owned, v : int, this.value : int}}\ \text{T-Lookup}$$

π_9

$$\frac{\frac{}{\texttt{int} \Rrightarrow \texttt{int/int}}}{\texttt{this : Currency@Owned, v : int, this.value : int, diff : int} \vdash_{\texttt{this}} \texttt{this.value : int} \dashv \texttt{this : Currency@Owned, v : int, this.value : int, diff : int}}\ \text{T-this-field-ctxt}$$

π_8

$$\frac{\pi_3 \quad \pi_4}{\texttt{this : Currency@Owned, v : int} \vdash_{\texttt{this}} \texttt{this.value - v : int} \dashv \texttt{this : Currency@Owned, v : int}}$$

π_2

$$\frac{\dfrac{}{\texttt{int} \Rrightarrow \texttt{int}/\texttt{int}}}{\texttt{this} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int},\ \texttt{this.value} : \texttt{int} \vdash_{\texttt{this}} \texttt{v} : \texttt{int} \dashv \texttt{this} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int},\ \texttt{this.value} : \texttt{int}}\ \pi_4 \quad \text{T-Lookup}$$

$$\frac{\texttt{this.value} \in \Delta_3 \qquad \texttt{int value} \in intersectFields(\texttt{Currency}) = \{\texttt{int value}\} \qquad \dfrac{}{\texttt{int} \Rrightarrow \texttt{int}/\texttt{int}}}{\Delta_3 = \texttt{this} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int} \vdash_{\texttt{this}} \texttt{this.value} : \texttt{int} \dashv \texttt{this} : \texttt{Currency@Owned},\ \texttt{v} : \texttt{int},\ \texttt{this.value} : \texttt{int}}\ \pi_3 \quad \text{T-this-field-def}$$

$$\frac{def(\texttt{Currency}) = \texttt{contract Currency \{ } S_0\ \texttt{int value ... \}} \qquad S_0\ \texttt{int value} \in \{S_0\ \texttt{int value}\}}{\begin{array}{c} stateFields(\texttt{Currency}, S_0) = \{\texttt{int value}\} \\ cdef(\texttt{Currency}) = \texttt{contract Currency \{ } S_0\ \texttt{int value ... \}} \\ \texttt{Unowned} \in \{\texttt{Shared}, \texttt{Owned}, \texttt{Unowned}\} \\ \hline intersectFields(\texttt{Currency}) = \{\texttt{int value}\} \end{array}}$$

B.3.3 Transaction `merge`

The transaction to check is

```
unit merge(Currency@Owned»Unowned other) Owned»Owned{ return e }
```

where

$$\begin{aligned} e &= \text{let } \texttt{x} : \texttt{int} = \texttt{other.getValue()} \text{ in } e_0 \\ e_0 &= \text{let } \texttt{sum} : \texttt{int} = \texttt{this.value + x} \text{ in } e_1 \\ e_1 &= \text{let } \texttt{em}_0 : \texttt{unit} = \texttt{this.value := sum} \text{ in } e_2 \\ e_2 &= \text{let } \texttt{em}_1 : \texttt{unit} = \texttt{disown other} \text{ in } \texttt{pack} \end{aligned}$$

Then, we have to prove the judgement

```
unit merge(Currency@Owned»Unowned other) Owned»Owned
                                  { return e } ok in Currency
```

Proof

$$\frac{params(\texttt{Currency}) = \varnothing \qquad \Gamma = \varnothing \qquad \pi_1}{\texttt{unit merge(Currency@Owned»Unowned other) Owned»Owned \{ return } e\ \texttt{\}} \text{ ok in Currency} \quad \text{PublicTransactionOk}$$

$$\frac{\pi_2 \qquad \pi_3 \qquad \vdash disposable(\texttt{int})}{\texttt{this} : \texttt{Currency@Owned},\ \texttt{other} : \texttt{Currency@Owned} \vdash_{\texttt{this}} \text{let } \texttt{x} : \texttt{int} = \texttt{other.getValue()} \text{ in } e_0 : \texttt{unit} \dashv \texttt{this} : \texttt{Currency@Owned},\ \texttt{other} : \texttt{Currency@Unowned}}\ \pi_1 \quad \text{T-let}$$

$$\frac{\pi_4 \qquad \pi_5 \qquad \vdash disposable(\texttt{unit})}{\texttt{this} : \texttt{Currency@Owned},\ \texttt{other} : \texttt{Currency@Owned},\ \texttt{x} : \texttt{int} \vdash_{\texttt{this}} \text{let } \texttt{sum} : \texttt{int} = \texttt{this.value + x} \text{ in } e_1 : \texttt{unit} \dashv \texttt{this} : \texttt{Currency@Owned},\ \texttt{other} : \texttt{Currency@Unowned},\ \texttt{x} : \texttt{int}}\ \pi_3 \quad \text{T-let}$$

$$\frac{\pi_8 \quad \pi_9 \quad \vdash \mathit{disposable}(\mathsf{unit})}{\begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \texttt{sum} : \mathsf{int}\end{array} \vdash_{\texttt{this}} \begin{array}{l}\mathsf{let}\ \mathtt{em}_0 : \mathsf{unit} = \texttt{this.value} := \texttt{sum}\\ \mathsf{in}\ e_2\\ :\ \mathsf{unit}\end{array} \dashv \begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \texttt{sum} : \mathsf{int}\end{array}}\ \text{T-LET}$$

π_5

$$\frac{\pi_{12} \quad \pi_{13} \quad \vdash \mathit{disposable}(\mathsf{unit})}{\begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \texttt{sum} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \mathtt{em}_0 : \mathsf{unit}\end{array} \vdash_{\texttt{this}} \begin{array}{l}\mathsf{let}\ \mathtt{em}_1 : \mathsf{unit} = \mathsf{disown}\ \texttt{other}\\ \mathsf{in}\ \mathsf{pack}\end{array} : \mathsf{unit} \dashv \begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Unowned},\\ \texttt{x} : \mathsf{int},\\ \texttt{sum} : \mathsf{int},\\ \mathtt{em}_0 : \mathsf{unit}\end{array}}\ \text{T-LET}$$

π_9

$$\frac{\mathit{contractFields}(\texttt{Currency}) = \{\texttt{int value}\} \quad \vdash \mathsf{int} <: \mathsf{int} \quad \frac{}{\vdash \mathsf{int} \approx \mathsf{int}}\ \approx\text{-Refl}}{\begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Unowned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \texttt{sum} : \mathsf{int},\\ \mathtt{em}_0 : \mathsf{unit},\\ \mathtt{em}_1 : \mathsf{unit}\end{array} \vdash_{\texttt{this}} \mathsf{pack} : \mathsf{unit} \dashv \begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Unowned},\\ \texttt{x} : \mathsf{int},\\ \texttt{sum} : \mathsf{int},\\ \mathtt{em}_0 : \mathsf{unit},\\ \mathtt{em}_1 : \mathsf{unit}\end{array}}\ \text{T-PACK}$$

π_{13}

$$\frac{\texttt{Currency} = \mathit{contract}(\texttt{Currency})}{\texttt{Currency@Owned} \Rrightarrow \texttt{Currency@Owned}/\texttt{Currency@Unowned}}\ \text{Split-Unowned} \quad \frac{}{\vdash \mathsf{Owned} <:_* \mathsf{Owned}}\ <:_*\text{-refl}}{\begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \texttt{sum} : \mathsf{int},\\ \mathtt{em}_0 : \mathsf{unit}\end{array} \vdash_{\texttt{this}} \mathsf{disown}\ \texttt{other} : \mathsf{unit} \dashv \begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Unowned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \texttt{sum} : \mathsf{int},\\ \mathtt{em}_0 : \mathsf{unit}\end{array}}\ \text{T-DISOWN}$$

π_{12}

$$\frac{\pi_{10} \quad \pi_{11} \quad \vdash \mathit{disposable}(\mathsf{int})}{\begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \texttt{sum} : \mathsf{int}\end{array} \vdash_{\texttt{this}} \texttt{this.value} := \texttt{sum} : \mathsf{unit} \dashv \begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \texttt{sum} : \mathsf{int}\end{array}}\ \text{T-FIELDUPDATE}$$

π_8

$$\frac{\frac{}{\mathsf{int} \Rrightarrow \mathsf{int}/\mathsf{int}}}{\begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \texttt{sum} : \mathsf{int}\end{array} \vdash_{\texttt{this}} \texttt{sum} : \mathsf{int} \dashv \begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int},\\ \texttt{sum} : \mathsf{int}\end{array}}\ \text{T-Lookup}$$

π_{11}

$$\frac{\texttt{this.value} \notin \mathit{Dom}(\Delta_5) \quad \texttt{int value} \in \mathit{intersectFields}(\texttt{Currency}) \quad \mathsf{int} \Rrightarrow \mathsf{int}/\mathsf{int}}{\begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \Delta_5 = \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int}\end{array} \vdash_{\texttt{this}} \texttt{this.value} : \mathsf{int} \dashv \begin{array}{l}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int}\end{array}}\ \text{T-THIS-FIELD-DEF}$$

π_{10}

$$\dfrac{\dfrac{\pi_6 \qquad \pi_7}{\texttt{this} : \texttt{Currency@Owned},\ \texttt{other} : \texttt{Currency@Owned},\ \texttt{x} : \mathsf{int} \vdash_{\texttt{this}} \texttt{this.value + x} : \mathsf{int} \dashv \begin{array}{c}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int}\end{array}}}{\pi_4}$$

$$\dfrac{\dfrac{\overline{\mathsf{int} \Rrightarrow \mathsf{int}/\mathsf{int}}}{\begin{array}{c}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int}\end{array} \vdash_{\texttt{this}} \texttt{x} : \mathsf{int} \dashv \begin{array}{c}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int}\end{array}}\ \text{T-Lookup}}{\pi_7}$$

$$\dfrac{\dfrac{\texttt{this.value} \in \Delta \qquad \texttt{int value} \in intersectFields(\texttt{Currency}) \qquad \mathsf{int} \Rrightarrow \mathsf{int}/\mathsf{int}}{\begin{array}{c}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int}\end{array} \vdash_{\texttt{this}} \texttt{this.value} : \mathsf{int} \dashv \begin{array}{c}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\\ \texttt{x} : \mathsf{int},\\ \texttt{this.value} : \mathsf{int}\end{array}}\ \text{T-this-field-def}}{\pi_6}$$

$$\dfrac{\begin{array}{c}\dfrac{}{\vdash \mathsf{Owned} <:_* \mathsf{Owned}}\ <:_*\text{-refl}\\ specializeTrans_\Gamma(\texttt{getValue}, \texttt{Currency}) = \texttt{int getValue() } S\text{»}S \texttt{ \{ return this.value \}} \forall S\\ \vdash bound(\texttt{Currency@Owned}) = \texttt{Currency@Owned} \qquad \forall f.\texttt{this}.f \notin \Delta_3\\ \texttt{Currency@Owned} = funcArg(\texttt{Currency@Owned}, \texttt{Currency@Owned}, \texttt{Currency@Owned})\end{array}}{\Delta_3 = \begin{array}{c}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned},\end{array} \vdash_{\texttt{this}} \texttt{other.getValue()} : \mathsf{int} \dashv \begin{array}{c}\texttt{this} : \texttt{Currency@Owned},\\ \texttt{other} : \texttt{Currency@Owned}\end{array}}\ \text{T-Inv}$$
$$\pi_2$$

B.4 Sending currency to a party

The expression to check is

$$\begin{array}{l}\quad \texttt{let x} : \texttt{Currency@Owned} = \texttt{h.split(v)}\\ \texttt{in A.receive(x)}\end{array}$$

where `A` : `Party@Owned`, `h` : `Currency@Owned` and `v` : int.

To complete the proof we need also the proof for transactions `split`, `merge` and `receive`. The first two proof are available in the previous section of this appendix.

B.4.1 Statement

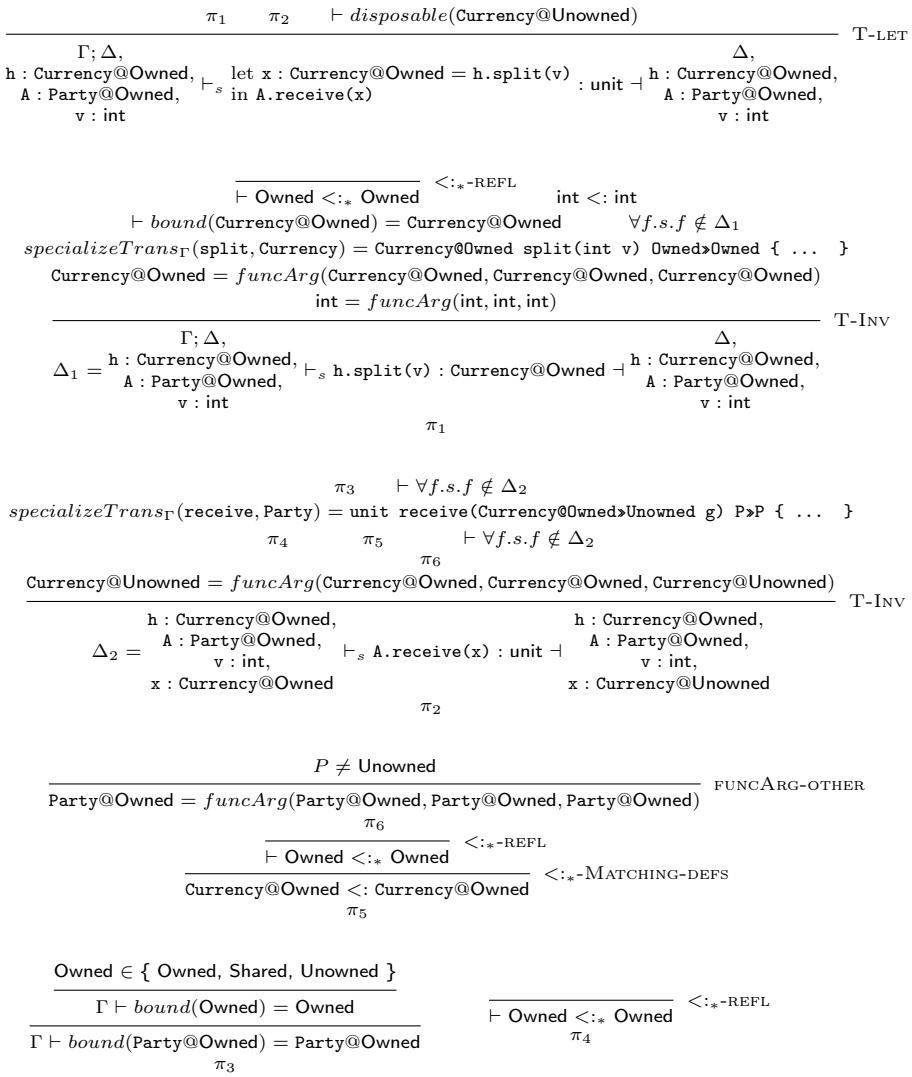
The expression to check is

$$\begin{array}{l}\quad \texttt{let x} : \texttt{Currency@Owned} = \texttt{h.split(v)}\\ \texttt{in A.receive(x)}\end{array}$$

where `A` : `Party@Owned`, `h` : `Currency@Owned` and `v` : int. Then, we have to prove the judgement for any Γ and Δ

$$\begin{array}{c}\Gamma; \Delta,\\ \texttt{h} : \texttt{Currency@Owned},\\ \texttt{A} : \texttt{Party@Owned},\\ \texttt{v} : \mathsf{int}\end{array} \vdash_s \texttt{let x} : \texttt{Currency@Owned} = \texttt{h.split(v)} \texttt{ in A.receive(x)} : \mathsf{unit} \dashv \begin{array}{c}\Delta,\\ \texttt{h} : \texttt{Currency@Owned},\\ \texttt{A} : \texttt{Party@Owned},\\ \texttt{v} : \mathsf{int}\end{array}$$

Proof



B.4.2 Transaction `receive`

The transaction to check is

```
unit receive(Currency@Owned»Unowned g) Owned»Owned { return e }
```

where

$$
\begin{aligned}
e &= \text{let } \texttt{x} : \texttt{Currency@Owned} = \texttt{this.wallet} \text{ in } e_0 \\
e_0 &= \text{let } \texttt{em}_1 : \texttt{unit} = \texttt{this.wallet := x} \text{ in } e_1 \\
e_1 &= \text{let } \texttt{em}_2 : \texttt{unit} = \texttt{this.wallet.merge(g)} \text{ in } \texttt{pack}
\end{aligned}
$$

Then, we have to prove the judgement

```
unit receive(Currency@Owned»Unowned g) Owned»Owned
                                    { return e } ok in Currency
```

Proof

$$\frac{params(\texttt{Party}) = \varnothing \qquad \Gamma = \varnothing \qquad \pi_1}{\texttt{unit receive(Currency@Owned»Unowned g) Owned»Owned \{ return } e \texttt{ \}}\ \textsc{PublicTransactionOk}$$

$$\frac{\pi_2 \qquad \pi_3 \qquad \vdash disposable(\texttt{Currency@Unowned})}{\texttt{this : Party@Owned, g : Currency@Owned} \vdash_{\texttt{this}} \texttt{let x : Currency@Owned = this.wallet in } e_0 \texttt{ : unit} \dashv \texttt{this : Party@Owned, g : Currency@Unowned}}\ \textsc{T-Let}$$

π_1

$$\frac{\pi_4 \qquad \pi_5 \qquad \vdash disposable(\texttt{unit})}{\texttt{this : Party@Owned, g : Currency@Owned, this.wallet : Currency@Unowned, x : Currency@Owned} \vdash_{\texttt{this}} \texttt{let em}_1 \texttt{ : unit = this.wallet := x in } e_1 \texttt{ : unit} \dashv \texttt{this : Party@Owned, g : Currency@Unowned, x : Currency@Unowned}}\ \textsc{T-Let}$$

π_3

$$\frac{\pi_6 \qquad \pi_7 \qquad \vdash disposable(\texttt{unit})}{\texttt{this : Party@Owned, g : Currency@Owned, this.wallet : Currency@Owned, x : Currency@Unowned, em}_1 \texttt{ : unit} \vdash_{\texttt{this}} \texttt{let em}_2 \texttt{ : unit = this.wallet.merge(g) in pack} \dashv \texttt{this : Party@Owned, g : Currency@Unowned, x : Currency@Unowned, em}_1 \texttt{ : unit}}\ \textsc{T-Let}$$

π_5

$$\frac{\begin{array}{c}\texttt{this.wallet} \in Dom(\Delta_7) \\ contractFields(\texttt{Party}) = \texttt{Currency@Owned wallet} \\ \vdash \texttt{Currency@Owned} <: \texttt{Currency@Owned} \\ \vdash \texttt{Currency@Owned} \approx \texttt{Currency@Owned}\end{array}}{\Delta_7 = \texttt{this : Party@Owned, g : Currency@Unowned, this.wallet : Currency@Unowned, x : Currency@Owned, em}_1 \texttt{ : unit, em}_2 \texttt{ : unit} \vdash_{\texttt{this}} \texttt{pack : unit} \dashv \texttt{this : Party@Owned, g : Currency@Unowned, x : Currency@Unowned, em}_1 \texttt{ : unit, em}_2 \texttt{ : unit}}\ \textsc{T-Pack}$$

π_7

$$\frac{\begin{array}{c}\vdash bound(\texttt{Currency@Owned}) = \texttt{Currency@Owned} \\ specializeTrans_{\Gamma}(\texttt{merge}, \texttt{Currency@Owned}) = \\ = \texttt{unit merge(Currency@Owned»Unowned g) Owned»Owned \{ ... \}} \\ \vdash \texttt{Owned} <:_* \texttt{Owned} \qquad \vdash \texttt{Currency@Owned} <: \texttt{Currency@Owned} \\ unionFields(\texttt{Party@Owned}) = \\ = \texttt{Currency@Owned this.wallet, List<NFT@Owned>@Owned this.tokens} = \\ = \overline{T_{f,decl}\ f} \\ fieldStates_{\texttt{this}}(\Delta_6; \overline{T_{f,decl}\ f}) = \texttt{Currency@Owned} \\ \Gamma \vdash \texttt{Currency@Owned} <: \texttt{Currency@Owned} \\ funcArg(\texttt{Currency@Owned, Currency@Owned, Currency@Owned}) = \texttt{Currency@Owned} \\ funcArg(\texttt{Currency@Owned, Currency@Owned, Currency@Unowned}) = \texttt{Currency@Unowned}\end{array}}{\Delta_6 = \texttt{this : Party@Owned, g : Currency@Owned, this.wallet : Currency@Owned, x : Currency@Unowned, em}_1 \texttt{ : unit} \vdash_{\texttt{this}} \texttt{this.wallet.merge(g) : unit} \dashv \texttt{this : Party@Owned, g : Currency@Unowned, this.wallet : Currency@Owned, x : Currency@Unowned, em}_1 \texttt{ : unit}}\ \textsc{T-Inv}$$

π_6

$$\frac{\pi_8 \quad \pi_9 \quad \vdash \mathit{disposable}(\texttt{Currency@Unowned})}{\begin{array}{l}\texttt{this : Party@Owned},\\ \texttt{g : Currency@Owned},\\ \texttt{this.wallet}\\ \quad\texttt{: Currency@Unowned},\\ \texttt{x : Currency@Owned}\end{array} \vdash_{\texttt{this}} \texttt{this.wallet := x} \dashv \begin{array}{l}\texttt{this : Party@Owned},\\ \texttt{g : Currency@Owned},\\ \texttt{this.wallet}\\ \quad\texttt{: Currency@Owned},\\ \texttt{x : Currency@Unowned}\end{array}}\ \textsc{T-fieldUpdate}$$

π_4

$$\frac{\texttt{Currency@Owned} \Rrightarrow \texttt{Currency@Owned/Currency@Unowned}}{\begin{array}{l}\texttt{this : Party@Owned},\\ \texttt{g : Currency@Owned},\\ \texttt{this.wallet}\\ \quad\texttt{: Currency@Unowned},\\ \texttt{x : Currency@Owned}\end{array} \vdash_{\texttt{this}} \texttt{x : Currency@Owned} \dashv \begin{array}{l}\texttt{this : Party@Owned},\\ \texttt{g : Currency@Owned},\\ \texttt{this.wallet}\\ \quad\texttt{: Currency@Unowned},\\ \texttt{x : Currency@Owned}\end{array}}\ \textsc{T-Lookup}$$

π_9

$$\frac{\texttt{Currency@Unowned} \Rrightarrow \texttt{Currency@Unowned/Currency@Unowned}}{\begin{array}{l}\texttt{this : Party@Owned},\\ \texttt{g : Currency@Owned},\\ \texttt{this.wallet}\\ \quad\texttt{: Currency@Unowned},\\ \texttt{x : Currency@Owned}\end{array} \vdash_{\texttt{this}} \texttt{this.wallet : Currency@Unowned} \dashv \begin{array}{l}\texttt{this : Party@Owned},\\ \texttt{g : Currency@Owned},\\ \texttt{this.wallet}\\ \quad\texttt{: Currency@Unowned},\\ \texttt{x : Currency@Owned}\end{array}}\ \textsc{T-this-field-ctxt}$$

π_8

$$\frac{\begin{array}{c}\texttt{this.wallet} \in \mathit{dom}(\Delta_2)\\ \texttt{Currency@Owned wallet} \in \mathit{intersectFields}(\texttt{Party@Owned})\\ \texttt{Currency@Owned} \Rrightarrow \texttt{Currency@Owned/Currency@Unowned}\end{array}}{\Delta_2 = \begin{array}{l}\texttt{this : Party@Owned},\\ \texttt{g : Currency@Owned}\end{array} \vdash_{\texttt{this}} \texttt{this.wallet : Currency@Owned} \dashv \begin{array}{l}\texttt{this : Party@Owned},\\ \texttt{g : Currency@Owned},\\ \texttt{this.wallet :}\\ \quad\texttt{Currency@Unowned}\end{array}}\ \textsc{T-this-field-def}$$

π_2