Faustus: Adding Formally Verified Parameterized Abstractions to the Smart Contract Language Marlowe*

Kegan McIlwaine Stone Olguin James Caldwell

University of Wyoming
19 March 2022

Abstract. In this paper we describe the Faustus Smart Contract Programming Language. Faustus is an extension of Marlowe. Marlowe itself is implemented in Haskell as a deeply embedded Domain Specific Language (DSL). Parameterized contracts in Faustus provide the means to i.) compactly represent a large class of Marlowe contracts, and ii.) make Marlowe contracts more readable by eliminating duplicated code. The programs and theories described in this paper have been formalized in the Isabelle theorem prover. We describe the syntax, typing rules, and formal small-step semantics for Faustus. We have also implemented a Faustus evaluator and proved that, for well-typed programs, the single step evaluator implements the small-step semantics and that the behavior of the small-step semantics coincides with step wise evaluation. Furthermore, we have defined a small-step semantics for Marlowe and have proved that the existing Marlowe evaluator and our small-step Marlowe semantics behave identically. We have also implemented a compiler mapping Faustus programs to Marlowe programs. We have proved the compiler correct, *i.e.* given a Faustus program (say F) taking a step in F using the Faustus semantics and then compiling the result to a Marlowe program, is equivalent to compiling F and then taking (some number) of steps in the Marlowe semantics.

1 Introduction

The Marlowe Smart Contract Programming Language is a deeply embedded domain-specific language (DSL) that was designed by Seijas and Thompson [8]. The language is designed for writing and analyzing financial contracts on blockchains. In the first version of Marlowe, there were seven small *Contract* constructs that could be put together to form larger contracts [8]. As the language strives for simplicity, the number of those constructs has been reduced to five in the most recent version of core Marlowe [6].

A design goal for Marlowe was to simplify the static analysis of contracts expressed in the language. For this reason, Marlowe does not allow recursion or

* The research presented in this paper was supported by a grant from IOG Singapore Pte. Ltd.

even a bounded form of looping. It is clearly not Turing complete, and of course neither is Faustus. With the static analysis goal forefront, every instance of the Haskell datatype representing the abstract syntax tree of a Marlowe program represents all inline executions of the contract. User input is processed when execution reaches a *When* contract. The *When* contract defines branches for different cases of valid user input and another branch for a timeout when no input is received. Before and after When contracts, the evaluation of a program is purely functional. If there isn't a path in the Marlowe abstract syntax tree corresponding to a specific execution trace, it is disallowed by the contract.

The first goal of Faustus[1] is to add language constructs to the Marlowe language that allow programmers to reuse code, decreasing the total amount of code written. The next goal is to preserve the semantic properties of Marlowe to ensure the same useful guarantees of Marlowe program execution made by Seijas et al. can be made for Faustus [6]. To this end, Faustus contains parameterized abstractions of the Contract type that can be used elsewhere in the program. Then, to allow more useful types of parameters in the *Contract* abstractions, Faustus also contains abstractions of the *Observable* and *PubKey* types. Note that these abstractions only entail binding phrases of meaningful syntactic parts of the language to a name as described by Schmidt [11]. For example, the only abstractions available in Marlowe are *Value* abstractions, where names are bound to the result of a Value expression. Our new abstractions allow programmers to better follow the "don't repeat yourself" (DRY) principle suggested by Hunt and Thomas to help programmers write easier to maintain code [4]. A type-checker is also provided for Faustus to prevent undefined behavior that would be found when using an identifier of the incorrect type for any of the types that can be abstracted. We will show through examples that these new language features allow Faustus programs to be smaller than their Marlowe counterparts.

Additionally, the programs and theories for Faustus have been formalized in the Isabelle theorem prover [10]. We have chosen to work in Isabelle, because Marlowe has already been formalized in the Isabelle theorem prover by Seijas and Thompson [8]. The existing formalization of Marlowe provides a base to build the Faustus formalization. We also found the proof search and automation tools in Isabelle to be invaluable in our work.

In this paper we present:

- Small-step semantics for Marlowe.
- Type-checker for Faustus.
- Small-step semantics for Faustus.
- Compiler for compiling Faustus to Marlowe.

2 Overview of Marlowe

In this section we will give an overview of Marlowe for readers that are unfamiliar. As described by Seijas et al. [6] Marlowe is a deeply embedded DSL. The

[1] The full Faustus project is available from `https://gitlab.ssc.dev/wabl/faustus`

DSL itself is intended to be language and transaction model [1] agnostic, but is currently implemented in Haskell to interface with the Cardano blockchain. The Contract datatype defines the main building blocks of Marlowe programs. A fully defined Marlowe Contract is a tree structure constructed at run-time in the language that Marlowe is embedded in. Each branch in the Contract tree structure fully defines a possible execution path for the Marlowe program, with no backtracking. The semantics of these Contracts are defined in [6] in an evaluator function, reduceContractStep, that performs one step of execution through the Contract. The constructors for the mutually recursive Case and Contract datatypes can be seen in the following Isabelle definition:

```
datatype Case = Case Action Contract
and Contract = Close
             | Pay AccountId Payee Token Value Contract
             | If Observation Contract Contract
             | When "Case list" Timeout Contract
             | Let ValueId Value Contract
```

Note that contracts are presented in continuation passing style. The *Close* constructor refunds money to owners of all the accounts that still exists in the contract. Since this is the end of a contract, there is no continuation for this constructor.

The *Pay* constructor sends money from the account with the given *AccountId* to the *Payee* which can be a participant or account in the contract. The currency and amount are given in the *Token* and *Value* parameters respectively. Values are arithmetic operations that evaluate to integers. Additionally, *Values* can use choices from participants made earlier in the execution of the program in their computation. The program will continue with the *Contract* argument.

The *If* constructor allows branching. *Observations* are *Boolean* expressions in Marlowe. Like *Values*, *Observations* can use choices made by participants earlier in the execution of the program in their computation. If the *Observation* is true at the time of execution, then the program continues with the first *Contract*. Otherwise the program continues with the second Contract.

The behavior of a Marlowe contract may depend on input from parties involved in the contract. These parties are identified by their public key identifiers. User input is formalized by Seijas et al. [7] as a list of transactions, each of which can hold a list of inputs. The logic for applying the inputs is defined by the *applyCases* function. When evaluation of a contract reaches a *When* constructor, it is paused unless the next input is a valid choice, deposit, or notification, or the timeout was passed at the time the current transaction was submitted. If inputs from participants match the *Action* of a *Case*, then the program continues with the corresponding *Contract*. If the timeout has been reached then the program continues with the *Contract* argument in the *When* constructor.

Last, the *Let* constructor allows binding a *Value* to a *ValueId* for later use inside the program. The program continues with the *Contract* argument, and will use the integer result of the eagerly evaluated *Value* argument where requested.

3 Formalized Marlowe Semantics

In this section we present the small-step semantics for Marlowe[2] in the style of Nipkow and Klein [9], and a formal verification of the semantics against the Marlowe reduceContractStep function defined by Seijas et al. [6]. We provide these new small-step semantics, because we found the *reduceContractStep* function difficult to work with when following the techniques for formalizing the semantics of programming languages provided by Nipkow and Klein [9].

Our small-step semantics define valid transitions between two Marlowe *Configurations* using the $\rightarrow$ infix operator. A *Configuration* is a 5-tuple type that holds the current *Contract*, *State*, *Environment*, *ReduceWarning* list, and *Payment* list information of a Marlowe program. The *ReduceWarning* list is a list of warnings that have been issued throughout the execution of the contract, up to the current *Configuration*. Similarly, the Payment list is a list of payments made by the execution of the contract, up to the current *Configuration*. Using the *reduceContractStep* function as our guide, we define the inductive rules for Marlowe small-step semantics in Appendix B.

We then verify that our small-step semantics behaves identically with the *reduceContractStep* function. In other words, we show that there is a valid small-step transition if, and only if, the Marlowe evaluator gives a *Reduced* result. Other results from the existing Marlowe evaluator are errors in the contract execution, and do not advance the contract execution [8]. Additionally, we do not define a small-step semantics corresponding to the *applyCases* function that processes inputs on a paused When constructor. Although user interaction is modeled here as a list of inputs, in an actual implementation *applyCases* works with part of the language that interfaces with transaction information from the host language.

The following three lemmas prove that if the Marlowe evaluator gives a *Reduced* result, either with or without a payment output, there is a valid small-step transition in our small-step semantics for Marlowe. Note that the *ws*, *ws1*, and *ws2* variables are of type *Warning* list and the *ps*, *ps1*, and *ps2* variables are of type *Payment* list, and are implicitly universally quantified.

```
lemma smallStepReductionAddsOneWarning:
assumes "(c1, s1, e1, ws1, ps1) → (c2, s2, e2, ws2, ps2)"
shows "∃w . ws2 = ws1 @ [w]"

lemma reduceStepIsSmallStepReductionNoPayment:
assumes "reduceContractStep e s1 c1 = Reduced w ReduceNoPayment s2 c2"
shows "(c1, s1, e, ws, ps) → (c2, s2, e, ws @ [w], ps)"

lemma reduceStepIsSmallStepReductionWithPayment:
assumes
  "reduceContractStep e s1 c1 = Reduced w (ReduceWithPayment p) s2 c2"
shows "(c1, s1, e, ws, ps) → (c2, s2, e, ws @ [w], ps @ [p])"
```

[2] The complete definition of the Marlowe small-step semantics in Isabelle can be found at `https://gitlab.ssc.dev/wabl/faustus/-/blob/master/isabelle/SmallStep.thy`.

The proofs for these lemmas are done by cases of the *c1* variable[3].

The next two lemmas prove that if there is a valid small-step transition in our small-step semantics for Marlowe, the Marlowe evaluator gives a *Reduced* result.

```
lemma smallStepReductionImpReduceStepNoPayment:
assumes "(c1, s1, e1, ws, ps) → (c2, s2, e2, ws @ [w], ps)"
shows "reduceContractStep e1 s1 c1 = Reduced w ReduceNoPayment s2 c2"

lemma smallStepReductionImpReduceStepWithPayment:
assumes "(c1, s1, e1, ws, ps) → (c2, s2, e2, ws @ [w], ps @ [p])"
shows
  "reduceContractStep e1 s1 c1 = Reduced w (ReduceWithPayment p) s2 c2"
```

The proofs for these lemmas are done by cases of the *c1* variable[3].

So we have presented a small-step semantics for Marlowe and proved that there is a valid small-step transition if, and only if, the Marlowe evaluator gives a *Reduced* result.

4 Formalized Faustus Semantics

Now that we have defined a small-step semantics for Marlowe, we extend the language to create Faustus[4]. Faustus adds abstractions of *Observations*, *PubKeys*, and *Contracts* to the language. Additionally, *Contract* abstractions can be parameterized by *Values*, *PubKeys*, and *Observations*.

We follow the techniques of Schmidt [11] to add these new abstractions. An *FConfiguration* is an extension of the Marlowe *Configuration*, that is now a 6-tuple type that holds the current *FContract*, *FContext*, *FState*, *Environment*, *ReduceWarning* list, and *Payment* list information of a Faustus program. We define the new *FContext* and *AbstractionInformation* types as follows:

```
datatype AbstractionInformation = ValueAbstraction Identifier
  | ObservationAbstraction Identifier
  | PubKeyAbstraction Identifier
  | ContractAbstraction "Identifier × FParameter list × FContract"

type_synonym FContext = "AbstractionInformation list"
```

The *FContext* type is a list of *AbstractionInformation*, and corresponds to the environment that holds identifier-meaning information from Schmidt [11]. *AbstractionInformation* holds an identifier-meaning pair. Since identifiers are directly mapped to values in the *FState*, the *FContext* only holds *Identifiers* for bound *Values*, *Observations*, and *PubKeys*. There are no locations in the *FState* to map to the identifiers in the *FContext*. For *Contracts*, the *FContext* holds parameter information and the body of the *Contract* that is bound to an *Identifier*.

[3] The full proof in Isabelle can be found at `https://gitlab.ssc.dev/wabl/faustus/-/blob/master/isabelle/SmallStep.thy`.

[4] The complete definition of Faustus in Isabelle can be found at `https://gitlab.ssc.dev/wabl/faustus/-/blob/master/isabelle/FaustusSemantics.thy`.

In this section, when modifications are made to a language construct that results in a new datatype definition, that new datatype will replace the old datatype in other language constructs as needed. For example, we create the *FPubKey* definition, and replace all Marlowe *PubKey* parameters with *FPubKey* parameters in Marlowe language constructs.

4.1 Abstracting PubKeys

We extend the *PubKeys* definition to include support for *PubKey* identifiers, in addition to *PubKey* constants which are already supported by Marlowe. This gives a new datatype called *FPubKey* with two constructors:

```
datatype FPubKey = ConstantPubKey PubKey  | UsePubKey Identifier
```

Now we add the *boundPubKeys* field to the *State* record to hold the bound *PubKeys*:

```
record FState = ··· boundPubKeys :: (Identifier × PubKey) list
```

Then we add a new constructor to the *Contract* datatype and rename it to *FContract*:

```
datatype FContract = ···  | LetPubKey Identifier FPubKey FContract
```

Finally, we add the corresponding rule to the small-step semantics now denoted with the $\rightarrow_f$ infix operator. This rule eagerly evaluates the *FPubKey*, stores the result in the *FState*, stores the identifier in the *FContext* as being bound to a *PubKey* abstraction using the Isabelle cons operator, #, and advances the contract:

```
LetPubKey:
  "⟦evalFPubKey s pk = Some res⟧ ⟹
  (LetPubKey i pk c, cx, s, e, ws, ps) →f
  (c, PubKeyAbstraction i#cx,
    s(|boundPubKeys := MList.insert i res (boundPubKeys s)|), e,
    ws @ [ReduceNoWarning], ps)"
```

4.2 Abstracting Observations

In Marlowe, *Observations* provide the means of writing Boolean expressions, and they evaluate to Boolean values. *Observations* are modified to take an identifier of a previously bound *Observation*. So we add a new constructor to the *Observation* datatype, and rename it to *FObservation*:

```
datatype FObservation = ··· | UseObservation Identifier
```

Note that we also rename the *Value* datatype to *FValue* since the *Value* and *Observation* datatypes are mutually recursive. Then we add a new constructor to the *FContract* datatype:

```
datatype FContract = ···
  | LetObservation Identifier FObservation FContract
```

Finally, we add the corresponding rule to the small-step semantics. As with *Values* and *PubKeys*, *Observations* are eagerly evaluated. The identifier is added to the *FContext* as bound to an *Observation*. The result is stored in the same

FState field as Values due to the way *LetObservation* contracts are compiled into a Marlowe contract in Section 6. A stored value of 1 corresponds to a true result, and a stored value of 0 corresponds to a false result.

```
LetObservation:
  "⟦evalFObservation e s obs = Some res⟧ ⟹
  (LetObservation i obs cont, cx, s, e, ws, ps) →f
  (cont, ObservationAbstraction i#cx, s(|boundValues := MList.insert i
    (if res then 1 else 0) (boundValues s)|),
    e, ws @ [ReduceNoWarning], ps)"
```

4.3 Abstracting Contracts

Two new constructors are added to the *FContract* datatype. These allow binding of parameterized contract abstractions to an identifier and executing a previously bound identifier with arguments:

```
datatype FContract = ···
  | LetC Identifier "FParameter list" FContract FContract
  | UseC Identifier "FArgument list"
```

An *FParameter* can bind *FPubKeys*, *FValues*, or *FObservations* to an identifier that can be used in the contract body. Then an *FArgument* supplies the actual *FPubKey*, *FValue*, or *FObservation* that will be used:

```
datatype FParameter = ValueParameter Identifier
  | ObservationParameter Identifier
  | PubKeyParameter Identifier

datatype FArgument = ValueArgument FValue
  | ObservationArgument FObservation
  | PubKeyArgument FPubKey
```

Then we have two rules to add to the small-step semantics:

```
LetC:
  "(LetC i params c1 c2, cx, s, e, ws, ps) →f
  (c2, ContractAbstraction (i, params, c1)#cx, s, e,
    ws @ [ReduceNoWarning], ps)"

UseCFound:
  "⟦lookupContractIdAbsInformation cx1 i = Some (params, c, cx2);
  evalFArguments e s1 params args = Some s2⟧ ⟹
  (UseC i args, cx1, s1, e, ws, ps) →f
  (c, (paramsToFContext params)@cx2, s2, e, ws @ [ReduceNoWarning], ps)"
```

The *LetC* rule adds the *c1 FContract* to the context with the *Identifier* and *FParameters*, and then continues the execution of the *c2 FContract*. Then the *UseCFound* rule requires the *Identifier* to exist in the context, and the arguments to evaluate to a new *FState*. The found body is executed with the new *FState* from executing arguments sequentially in *evalFArguments*. The *innerContext* from *lookupContractIdAbsInformation* is the *AbstractionInformation* list that comes after the first *ContractAbstraction* in the list that is bound to the *Identifier* that is being looked up. This removes the ability to use recursion, as the *innerContext* will not contain the *ContractAbstraction* that is found.

4.4 Summary

We have added the language constructs and rules for handling abstractions of *PubKeys*, *Observations*, and *Contracts*. These extended datatypes have been renamed to *FPubKey*, *FObservation*, and *FContract* respectively. With these new rules, we can only continue execution of the *Use*, *UseObservation*, *UsePubKey*, and *UseC* constructs if the corresponding *Identifier* has been bound. This adds new halting conditions to the execution of Faustus smart contracts that do not exist in Marlowe. In the next section we present a type-checker that prevents a Faustus smart contract from halting at a *Use*, *UseObservation*, *UsePubKey*, or *UseC*.

5 Type Checking Faustus

In this section we present a type-checker for Faustus[5]. Marlowe notably does not have a type-checker, as all syntactically correct Marlowe programs will execute as described by Seijas and Thompson [8]. Nipkow and Klein demonstrate how to implement a type-checker as a series of inductive rules in Isabelle [9]. Modifying, but based on their model, we implement our Faustus type-checker as a series of functions that type check the *FContract* and other pieces of the syntax tree recursively along with the *FState* and *FContext*. The types of these functions and the *TypeContext* are:

```
datatype AbstractionTypeInformation = ValueType | ObservationType
 | PubKeyType | ContractType "FParameter list"

type_synonym TypeContext =
  "(Identifier × AbstractionTypeInformation) list"

fun paramsToTypeContext :: "FParameter list ⇒ TypeContext"
fun wellTypedPubKey :: "TypeContext ⇒ FPubKey ⇒ bool"
fun wellTypedPayee :: "TypeContext ⇒ FPayee ⇒ bool"
fun wellTypedChoiceId :: "TypeContext ⇒ FChoiceId ⇒ bool"
fun wellTypedValue :: "TypeContext ⇒ FValue ⇒ bool" and
  wellTypedObservation :: "TypeContext ⇒ FObservation ⇒ bool"
fun wellTypedArgument :: "TypeContext ⇒ FParameter ⇒ FArgument ⇒ bool"
fun wellTypedArguments :: "TypeContext ⇒ FParameter list ⇒
  FArgument list ⇒ bool"
fun wellTypedContract :: "TypeContext ⇒ FContract ⇒ bool" and
  wellTypedCases :: "TypeContext ⇒ FCase list ⇒ bool"
```

The type checking rules for Faustus can be found in Figures 1, 2, and 3, and the Isabelle functions for the type-checker can be found in Appendix C. For readability, in Figure 1, we use $\mathcal{C}$ to denote the *Contract* type. Also note that the type context Γ is represented as a list, and we use : , the list cons operator, to add type information to the type context.

[5] The complete definition and corresponding proofs of the Faustus type-checker in Isabelle can be found at `https://gitlab.ssc.dev/wabl/faustus/-/blob/master/isabelle/FaustusSemantics.thy`.

$$\frac{}{\Gamma \vdash \mathit{Close} :: \mathcal{C}} \qquad \frac{\Gamma \vdash \mathit{obs} :: \mathit{Observation} \quad \Gamma \vdash \mathit{c1} :: \mathcal{C} \quad \Gamma \vdash \mathit{c2} :: \mathcal{C}}{\Gamma \vdash \mathit{If\ obs\ c1\ c2} :: \mathcal{C}}$$

$$\frac{\Gamma \vdash \mathit{pk} :: \mathit{PubKey} \quad \Gamma \vdash \mathit{py} :: \mathit{Payee} \quad \Gamma \vdash \mathit{v} :: \mathit{Value} \quad \Gamma \vdash \mathit{c} :: \mathcal{C}}{\Gamma \vdash \mathit{Pay\ pk\ py\ t\ v\ c} :: \mathcal{C}}$$

$$\frac{\Gamma \vdash \mathit{cs} :: \mathit{Case\ list} \quad \Gamma \vdash \mathit{c} :: \mathcal{C}}{\Gamma \vdash \mathit{When\ cs\ t\ c} :: \mathcal{C}}$$

$$\frac{\Gamma \vdash \mathit{v} :: \mathit{Value} \quad (i, \mathit{Value}) : \Gamma \vdash \mathit{c} :: \mathcal{C}}{\Gamma \vdash \mathit{Let\ i\ v\ c} :: \mathcal{C}}$$

$$\frac{\Gamma \vdash \mathit{o} :: \mathit{Observation} \quad (i, \mathit{Observation}) : \Gamma \vdash \mathit{c} :: \mathcal{C}}{\Gamma \vdash \mathit{LetObservation\ i\ o\ c} :: \mathcal{C}}$$

$$\frac{\Gamma \vdash \mathit{p} :: \mathit{PubKey} \quad (i, \mathit{PubKey}) : \Gamma \vdash \mathit{c} :: \mathcal{C}}{\Gamma \vdash \mathit{LetPubKey\ i\ p\ c} :: \mathcal{C}}$$

$$\frac{(\mathit{paramsToTypeContext\ p}) : \Gamma \vdash \mathit{c1} :: \mathcal{C} \quad (i, C, p) : \Gamma \vdash \mathit{c2} :: \mathcal{C}}{\Gamma \vdash \mathit{LetC\ i\ p\ c1\ c2} :: \mathcal{C}}$$

$$\frac{}{(i, \mathcal{C}, []) : - \vdash \mathit{UseC\ i\ []} :: \mathcal{C}} \qquad \frac{\Gamma \vdash \mathit{UseC\ i\ a} :: C}{(k, \mathcal{C}, p) : \Gamma \vdash \mathit{UseC\ i\ a} :: \mathcal{C}}\, k \neq i$$

$$\frac{(i, \mathcal{C}, p) : \Gamma \vdash \mathit{UseC\ i\ a} :: \mathcal{C}}{(i, \mathcal{C}, \mathit{ValueParameter\ k} : p) : \Gamma \vdash \mathit{UseC\ i\ (ValueArgument\ v{:}a)} :: \mathcal{C}}$$

$$\frac{(i, \mathcal{C}, p) : \Gamma \vdash \mathit{UseC\ i\ a} :: \mathcal{C}}{(i, \mathcal{C}, \mathit{ObservationParameter\ k} : p) : \Gamma \vdash \mathit{UseC\ i\ (ObservationArgument\ o{:}a)} :: \mathcal{C}}$$

$$\frac{(i, \mathcal{C}, p) : \Gamma \vdash \mathit{UseC\ i\ a} :: \mathcal{C}}{(i, \mathcal{C}, \mathit{PubKeyParameter\ k} : p) : \Gamma \vdash \mathit{UseC\ i\ (PubKeyArgument\ pk{:}a)} :: \mathcal{C}}$$

Fig. 1. Faustus Contract type rules. Implemented in Isabelle with the wellTypedContract function.

From our definition of the type-checker, we can prove properties of the execution of well-typed Faustus contracts. We first show that well-typed *PubKeys*, *Values*, and *Observations* will evaluate to a meaningful value in the following lemmas:

lemma `wellTypedPubKeyEvaluates:`

```
"wellTypedPubKey tcx p ⟹  wellTypedState tcx s ⟹
  evalFPubKey s p ≠ None"
```

lemma `wellTypedValueObservationEvaluates:`

```
"wellTypedValue tcx v ⟹ wellTypedState tcx s ⟹
  evalFValue e s v ≠ None"
"wellTypedObservation tcx o ⟹ wellTypedState tcx s ⟹
  evalFObservation e s o ≠ None"
```

The proofs for these lemmas are done by using the induction rules defined by the type-checker. Now we create a definition for *FConfigurations* in a final state. Then the following lemma proves that a Faustus small-step reduction on a well-typed *FContract* only halts on *Close* and *When* contracts.

definition `"final cs ⟷ ¬(∃cs'. cs →f cs')"`

$$\frac{}{[] \vdash [] :: Context} \qquad \frac{\Gamma \vdash C :: Context}{(i, PubKeyType) : \Gamma \vdash PubKeyAbstraction\ i : C :: Context}$$

$$\frac{\Gamma \vdash C :: Context}{(i, ValueType) : \Gamma \vdash ValueAbstraction\ i : C :: Context}$$

$$\frac{\Gamma \vdash C :: Context}{(i, ObservationType) : \Gamma \vdash ObservationAbstraction\ i : C :: Context}$$

$$\frac{\Gamma \vdash C :: Context \quad (paramsToTypeContext\ p) : \Gamma \vdash c :: Contract}{(i, ContractType\ p) : \Gamma \vdash ContractAbstraction\ (i, p, c) : C :: Context}$$

Fig. 2. Faustus context type rules. Implemented in Isabelle as the wellTypedContext function.

$$\frac{}{[] \vdash S :: State} \qquad \frac{\Gamma \vdash S :: State}{(i, ContractType\ p) : \Gamma \vdash S :: State}$$

$$\frac{\Gamma \vdash S :: State \quad member\ i\ (boundValues\ S)}{(i, ValueType) : \Gamma \vdash S :: State}$$

$$\frac{\Gamma \vdash S :: State \quad member\ i\ (boundValues\ S)}{(i, ObservationType) : \Gamma \vdash S :: State}$$

$$\frac{\Gamma \vdash S :: State \quad member\ i\ (boundPubKeys\ S)}{(i, PubKeyType) : \Gamma \vdash S :: State}$$

Fig. 3. Faustus state type rules. Implemented in Isabelle as the wellTypedState function.

lemma `finalD:`

```
"(final (c1, cx, s, e, w, p) ∧ wellTypedContract tcx c1 ∧
  wellTypedContext tcx cx ∧ wellTypedState tcx s) ⟹
  c1 = Close ∨ (∃ cs t c2 . c1 = When cs t c2)"
```

The proof for this lemma is done by cases of the *c1* variable. We are able to show that all contracts except *Close* and *When* will reduce.

The next lemma proves that a well-typed *FContract* will continue to be well-typed after a small-step reduction.

lemma `typePreservation:`

```
assumes "(c1, cx1, s1, e1, w1, p1) →f (c2, cx2, s2, e2, w2, p2) ∧
    wellTypedContract tcx1 c1 ∧ wellTypedContext tcx1 cx1 ∧
    wellTypedState tcx1 s1"
shows "(∃tcx2 . wellTypedContract tcx2 c2 ∧ wellTypedContext tcx2 cx2 ∧
    wellTypedState tcx2 s2)"
```

The proof for this lemma is done by cases of the *c1* variable. By going through small-step reduction rules, we construct a type context so that the *c2*, *cx2*, and *s2* are well-typed.

In this section we have defined the type-checker for *FContracts*. The type-checker allows us to avoid halting on the *UsePubKey*, *Use*, *UseObservation*, and *UseC* constructs when running small-step reductions. Note that *Use* is a Marlowe

Value. Marlowe does not have a type-checker, and so the Marlowe evaluator can encounter an identifier that has not previously been declared. The Faustus type-checker can be used on Marlowe contracts to avoid this error, because Marlowe is a subset of Faustus. In the next section we present the Faustus compiler, and show that well-typed *FContracts* will compile.

6 Formalized Faustus Compiler

In this section we present the compiler for *Faustus*[6]. The compiler translates an *FContract*, *FContext*, and *FState* at any point of execution into a Marlowe Contract. The *FState* is also translated into a Marlowe *State* by dropping the *boundPubKeys* field in the record. As demonstrated by Nipkow and Klein [9], but extended for parameterized abstractions, we define the compiler as a series of functions that translate Faustus program into a Marlowe program in Appendix D, and prove that the semantics of the original Faustus program are preserved in the compiled Marlowe program. The types of these functions are:

```
fun compileFPubKey :: "FPubKey ⇒ FState ⇒ PubKey option"
fun compileFPayee :: "FPayee ⇒ FState ⇒ Payee option"
fun compileFChoiceId :: "FChoiceId ⇒ FState ⇒ ChoiceId option"
fun compileFValue :: "FValue ⇒ FState ⇒ Value option" and
  compileFObservation :: "FObservation ⇒ FState ⇒ Observation option"
fun compileAction :: "FAction ⇒ FState ⇒ Action option"
fun argumentsToCompilerState :: "FParameter list ⇒
  FArgument list ⇒ FState ⇒ FState option"
fun compileArguments :: "FParameter list ⇒ FArgument list ⇒
  FState ⇒ Contract ⇒ Contract option"
function compile :: "FContract ⇒ FContext ⇒ FState ⇒ Contract option"
```

Note that *PubKeys* can be evaluated at compile time. We show this in the following lemma.

```
lemma preservationOfPubKeyEvaluation:
  "(∀tcx. wellTypedPubKey tcx p ⟶ wellTypedState tcx s ⟶
    evalFPubKey s p = compileFPubKey p s)"
```

The proof for this lemma is done by cases on the p variable.

For *Values* and *Observations* we need multiple lemmas to show that the compiler is correct. One lemma shows that well-typed *Values* and *Observations* will compile. The next two lemmas show that the evaluation of *Values* and *Observations* is preserved by the compiler.

```
lemma wellTypedValueObservationCompiles:
"wellTypedValue tcx v1 ⟶ wellTypedState tcx s ⟶
  (∃v2 . compileFValue v2 s = Some v2)"
"wellTypedObservation tcx o1 ⟶ wellTypedState tcx s ⟶
  (∃o2 . compileFObservation o1 s = Some o2)"
```

[6] The complete definition and corresponding proofs of the Faustus compiler in Isabelle can be found at `https://gitlab.ssc.dev/wabl/faustus/-/blob/master/isabelle/FaustusSemantics.thy`.

```
lemma preservationOfValueEvaluation:
"(∀tcx i. wellTypedValue tcx v ⟶ wellTypedState tcx s ⟶
  evalFValue e s v = Some i ⟶
  evalValue e (fStateToMState s) (the (compileFValue v s)) = i)"
"(∀tcx b. wellTypedObservation tcx o ⟶ wellTypedState tcx s ⟶
  evalFObservation e s o = Some b ⟶
  evalObservation e (fStateToMState s)
    (the (compileFObservation o s)) = b)"
lemma preservationOfValueEvaluation_MarloweImpliesFaustus:
"(∀tcx i. wellTypedValue tcx v ⟶ wellTypedState tcx s ⟶
  evalValue e (fStateToMState s) (the (compileFValue v s)) = i ⟶
  evalFValue e s v = Some i)"
"(∀tcx b. wellTypedObservation tcx o ⟶ wellTypedState tcx s ⟶
  evalObservation e (fStateToMState s)
    (the (compileFObservation o s)) = b ⟶
  evalFObservation e s o = Some b)"
```

The proofs for these lemmas are done by using the induction rules defined by the compiler.

Now that we have shown well-typed *PubKeys*, *Values*, and *Observations* compile, we show that well-typed Faustus programs will compile in the following lemma.

```
lemma wellTypedCompiles:
"(∀tcx . (wellTypedContract tcx c ⟶ wellTypedState tcx s ⟶
  wellTypedContext tcx cx ⟶ (∃mc . compile c cx s = Some mc)))"
"(∀tcx .(wellTypedCases tcx cs ⟶ wellTypedState tcx s ⟶
  wellTypedContext tcx cx ⟶ (∃mcs. compileCases cs cx s = Some mcs)))"
```

This lemma is proved by induction using the induction rule defined by the compile function.

Now that we have proved that well-typed Faustus programs compile into Marlowe programs, and we have proved that *PubKeys*, *Values*, and *Observations* evaluations are preserved by the compiler, we show the semantics of the Faustus program are preserved by the compiled Marlowe program in the next lemma. We use the $\rightarrow^*$ operator to denote zero or more steps in the Marlowe semantics.

```
lemma preservationOfSemantics_FaustusStepImpMarloweSteps:
  "(c1, cx1, s1, e1, w1, p1) →f (c2, cx2, s2, e2, w2, p2) ⟹
  wellTypedContract tcx c1 ⟹ wellTypedContext tcx cx1 ⟹
  wellTypedState tcx s1 ⟹ compile c1 cx1 s1 = Some mc1 ⟹
  compile c2 cx2 s2 = Some mc2 ⟹
  (∃mw2 . (mc1, fStateToMState s1, e1, w1, p1) →*
    (mc2, fStateToMState s2, e2, mw2, p2))"
```

The proof for this lemma is done by induction, using the small-step semantic rules previously defined for the $\rightarrow_f$ operator in Section 4.

Since the above lemma proves that a Faustus program's executing semantics are preserved by the compiled Marlowe program, we now show that a halted Faustus programs will compile to a halted Marlowe program in the next lemma.

```
lemma presOfSemantics_HaltedFaustusImpHaltedMarlowe:
  "final (c, cx, s, e, w, p) ⟹ wellTypedContract tcx c ⟹
  wellTypedContext tcx cx ⟹ wellTypedState tcx s ⟹
  finalMarlowe (the (compile c cx s), fStateToMState s, e, w, p)"
```

The proof for this lemma is done by cases on the c variable.

Last, we must show that applying inputs is preserved by the compiler. The following lemmas show that a Faustus contract only advances on user input if, and only if, the Marlowe contract that it compiles to advances on that same user input.

```
lemma preservationOfApplyCases_FaustusErrorImpliesMarloweError:
  "applyCases e s i c1 = ApplyNoMatchError ⟹ wellTypedCases tcx c1 ⟹
  wellTypedState tcx s ⟹ compileCases c1 cx s = Some c2 ⟹
  applyCases e (fStateToMState s) i c2 = ApplyNoMatchError"

lemma preservationOfApplyCases_FaustusAppliedImpliesMarloweApplied:
  "applyCases e s1 i cs1 = Applied w s2 c ⟹ wellTypedCases tcx cs1 ⟹
  wellTypedState tcx s1 ⟹ compileCases cs1 cx s1 = Some cs2 ⟹
  applyCases e (fStateToMState s1) i cs2 =
    Applied w (fStateToMState s2) (the (compile c cx s2))"

lemma preservationOfApplyCases_MarloweErrorImpliesFaustusError:
  "wellTypedCases tcx c ⟹ wellTypedState tcx s ⟹
  wellTypedFaustusContext tcx cx ⟹
  applyCases e (fStateToMState s) i (the (compileCases c cx s)) =
    ApplyNoMatchError ⟹ applyCases e s i c = ApplyNoMatchError"

lemma preservationOfApplyCases_MarloweAppliedImpliesFaustusApplied:
  "(∃s2 c. wellTypedCases tcx cs ⟶ wellTypedState tcx s1 ⟶
  wellTypedFaustusContext tcx cx ⟶
  applyCases e (fStateToMState s1) i (the (compileCases cs cx s1)) =
    Applied w ms mc ⟶
  (applyCases e s1 i cs = Applied w s2 c ∧ fStateToMState s2 = ms ∧
    (the (compile c cx s2)) = mc))"
```

The proofs for these lemmas are done by induction on the Faustus cases that *applyCases* is called on.

Now we have shown that the Marlowe program returned by the compiler fully preserves the execution semantics of the original Faustus program. If the Faustus can continue executing, then the compiled Marlowe program will also continue executing in the same way. Additionally, if the Faustus program has halted, the compiled Marlowe program will also halt. Finally, we note that any property that holds for the execution of all Marlowe programs (*e.g.* guaranteed termination or money preservation proved by Seijas et al. [6]) holds for all well-typed Faustus programs. This follows from the fact that a well-typed Faustus program will compile to a Marlowe program that preserves the semantics of the Faustus program.

7 Conciseness of Faustus Programs

For an example of code size savings, we will examine a simple escrow contract written in Marlowe that can be found in Figure 4 in Appendix A. In this contract we can identify two pieces of similar code when "alice" or "bob" have made a claim to the money after it is deposited. No matter who makes the claim, "carol" will choose to agree or disagree with the claimant, and the money is sent to the claimant or nonclaimant based on carol's choice.

With these observations, we can rewrite this contract in Faustus in Figure 5 in Appendix A. We move the code after "bob" or "alice" make a claim to a Contract abstraction, "processClaim" that takes the claimant and nonclaimant PubKeys as parameters. So when "alice" makes a claim we call "processClaim" with the parameters "alice" and "bob", and vice versa when "bob" makes a claim.

Even in this simple example, the Faustus program is shorter and more maintainable. The sections of code that are meant to have the same behavior in Marlowe are in a single abstraction in Faustus as expected. By better following the DRY principle from Hunt and Thomas programmers will not be required to find all duplicate pieces of code in a Faustus program if the behavior of a program needs to be changed [4].

Experiments with larger multisignature contracts have also shown the power of our simple extension[7]. A multisignature contract that allows 5 people to vote in any order can be written in Faustus in 199 lines of code when pretty printed. In Marlowe, the compiled contract is 4030 lines of code when pretty printed. This savings comes from five *LetC* abstractions, and 15 *UseC* calls in the Faustus program.

8 Related Work

There has been work done by Kondratiuk et al. to make Marlowe more usable for programmers by creating standardized contract generators [5]. By using generators, programmers avoid re-implementing common contracts. Our approach to this problem allows programmers to re-use more specialized logic throughout a contract by using the *LetC* and *UseC FContract* constructors.

There has also been work done by Freeman and Pryce exploring the process of extending a DSL with usability in mind [3]. Our work in creating Faustus differs by providing a formal verification of the extended language, the type-checker, and the compiler. These techniques are common in work with general purpose programming languages, but we have not found an example of applying formal methods to guarantee the preservation of semantic properties in the extension of a DSL.

[7] The multisignature contract example in Marlowe and Faustus can be found at `https://gitlab.ssc.dev/wabl/faustus/-/tree/master/src/Language/Faustus/Examples`.

9 Conclusions and Future Work

In this paper we have described the Faustus Smart Contract Programming Language. We first defined the small-step semantics of the Marlowe Smart Contract Programming Language for easier future work in the Isabelle theorem prover. Then we extended the Marlowe language and semantics with abstractions of *PubKeys*, *Observations*, and *Contracts* to create the Faustus language. In order to work with the new abstractions, we then defined a type-checker for Faustus that prevented halting when using the newly added pieces of the language. Finally, we defined a compiler that maps Faustus programs to Marlowe programs, and then proved that the compiled Marlowe programs preserve the semantics of the original Faustus programs.

We foresee a few ways that work on Faustus can be continued. As discussed in Section 7, Faustus programs are smaller than their corresponding Marlowe programs. An implementation of Faustus on a blockchain will reduce the amount of data transmitted when running a smart contract. Providing a method of converting Marlowe programs to Faustus programs through clone detection and anti-unification will allow programmers to reduce the size of programs, and data transmission, without needing to re-implement and test the program logic themselves. We believe this can be done using the abstract syntax tree clone detection process described by Bulychev and Minea [2].

In addition to a conversion from Marlowe to Faustus, we also believe there may be ways to improve the usability of Faustus through staged computation. Currently, the Faustus program will have to be constructed at run-time in the language that it is embedded in, so a programmer could create and run a Faustus program that is never type-checked if they forgo the type-checking by accident or on purpose. Creating a library that moves the type-checking and compilation closer to the compilation of the language that Faustus is embedded in would help prevent additional programmer errors.

Acknowledgements

The work done on this project was made possible through the support provided by IOG Singapore Pte. Ltd. We are extremely grateful to Simon Thompson, Pablo Lamela Seijas, Alexander Nemish, Brian Bush, and the other members of the Marlowe team at IOG for their discussions and feedback on our implementation of Faustus. We also extend our thanks to Mike Borowczak, Sujan Dhakal, Finley McIlwaine, Hadi Schafei, Philip Schlump, and the rest of the University of Wyoming Advanced Blockchain Laboratory for their feedback and support through this project.

References

1. Brünjes, L., Gabbay, M.J.: UTxO- vs Account-Based Smart Contract Blockchain Programming Paradigms. In: Leveraging Applications of Formal Methods, Verification and Validation: Applications: 9th International Symposium on Leveraging

Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III. pp. 73–88. Springer-Verlag, Berlin, Heidelberg (Oct 2020). https://doi.org/10.1007/978-3-030-61467-6_6, https://doi.org/10.1007/978-3-030-61467-6_6

2. Bulychev, P., Minea, M.: Duplicate Code Detection Using Anti-Unification. In: Spring Young Researchers' Colloquium on Software Engineering. pp. 51–54 (2008). https://doi.org/10.15514/SYRCOSE-2008-2-22, http://syrcose.ispras.ru/2008/files/22_paper.pdf
3. Freeman, S., Pryce, N.: Evolving an embedded domain-specific language in Java. In: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. pp. 855–865. OOPSLA '06, Association for Computing Machinery, New York, NY, USA (Oct 2006). https://doi.org/10.1145/1176617.1176735, http://doi.org/10.1145/1176617.1176735
4. Hunt, A., Thomas, D.: The Pragmatic programmer : from journeyman to master. Addison-Wesley, Boston [etc.] (2000), http://www.amazon.com/The-Pragmatic-Programmer-Journeyman-Master/dp/020161622X
5. Kondratiuk, D., Seijas, P.L., Nemish, A., Thompson, S.: Standardized Crypto-Loans on the Cardano Blockchain. In: Bernhard, M., Bracciali, A., Gudgeon, L., Haines, T., Klages-Mundt, A., Matsuo, S., Perez, D., Sala, M., Werner, S. (eds.) Financial Cryptography and Data Security. FC 2021 International Workshops, vol. 12676, pp. 579–594. Springer Berlin Heidelberg, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-662-63958-0_41, https://link.springer.com/10.1007/978-3-662-63958-0_41, series Title: Lecture Notes in Computer Science
6. Lamela Seijas, P., Nemish, A., Smith, D., Thompson, S.: Marlowe: Implementing and Analysing Financial Contracts on Blockchain. In: Bernhard, M., Bracciali, A., Camp, L.J., Matsuo, S., Maurushat, A., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security. pp. 496–511. Lecture Notes in Computer Science, Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-54455-3_35
7. Lamela Seijas, P., Smith, D., Thompson, S.: Efficient Static Analysis of Marlowe Contracts. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 161–177. Springer International Publishing, Cham (2020)
8. Lamela Seijas, P., Thompson, S.: Marlowe: Financial contracts on blockchain. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. pp. 356–375. Springer International Publishing, Cham (2018)
9. Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer (2014). https://doi.org/10.1007/978-3-319-10542-0, https://doi.org/10.1007/978-3-319-10542-0
10. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9, https://doi.org/10.1007/3-540-45949-9
11. Schmidt, D.A.: The structure of typed programming languages. The Mit Press (1994)

A Example Programs

An example of an escrow contract as a Marlowe program can be found in Figure 4, and an example of the same contract as a Faustus program can be found in Figure 5.

B Marlowe Small-Step Semantics

The full Isabelle definition of the Marlowe small-step semantics can be found in Figure 6 and Figure 7.

C Faustus Type-Checker

The full Isabelle definition of the Faustus type-checker can be found in Figure 8, Figure 9, and Figure 10.

D Faustus Compiler

The full Isabelle definition of the Faustus compiler can be found in Figure 11, Figure 12, Figure 13, Figure 14, and Figure 15.

```
When [
  (Case
     (Deposit "alice" "alice"
        (Constant 450))
     (When [
       (Case
          (Choice
             (ChoiceId "claim" "bob") [
             (Bound 0 0)])
          (When [
                (Case
                   (Choice
                      (ChoiceId "agree" "carol") [
                      (Bound 0 0)])
                   (Pay "alice"
                      (Party "bob")
                      (Constant 450) Close))
                ,
                (Case
                   (Choice
                      (ChoiceId "agree" "carol") [
                      (Bound 1 1)]) Close)] 100 Close))
       ,
       (Case
          (Choice
             (ChoiceId "claim" "alice") [
             (Bound 0 0)])
          (When [
                (Case
                   (Choice
                      (ChoiceId "agree" "carol") [
                      (Bound 0 0)]) Close)
                ,
                (Case
                   (Choice
                      (ChoiceId "agree" "carol") [
                      (Bound 1 1)])
                   (Pay "alice"
                      (Party "bob")
                      (Constant 450) Close))] 100 Close))] 90 Close))]
  10 Close
```

Fig. 4. Marlowe simple escrow contract.

```
LetC "processClaim" [(PubKeyParameter "claimant"),
  (PubKeyParameter "nonclaimant")]
  (When [
    (Case
       (Choice
          (ChoiceId "agree" "carol") [
          (Bound 0 0)])
       (Pay "*nonclaimant"
          (Party "*claimant")
          (Constant 450) Close))
    ,
    (Case
       (Choice
          (ChoiceId "agree" "carol") [
          (Bound 1 1)])
       (Pay "*claimant"
          (Party "*nonclaimant")
          (Constant 450) Close))] 100 Close)
  (When [
     (Case
        (Deposit "alice" "alice"
           (Constant 450))
        (When [
          (Case
             (Choice
                (ChoiceId "claim" "bob") [
                (Bound 0 0)])
             (UseC "processClaim" [(PubKeyArgument "bob"),
                                   (PubKeyArgument "alice")]))
          ,
          (Case
             (Choice
                (ChoiceId "claim" "alice") [
                (Bound 0 0)])
             (UseC "processClaim" [(PubKeyArgument "alice"),
                                   (PubKeyArgument "bob")]))] 90 Close))]
     10 Close)
```

Fig. 5. Faustus simple escrow contract.

```
type_synonym Configuration = "Contract * State * Environment *
(ReduceWarning list) * (Payment list)"
inductive
  small_step_reduce :: "Configuration ⇒ Configuration ⇒ bool" (infix
"→" 55)
where
CloseRefund:  "refundOne (accounts s) = Some ((p, t, m), a) ⟹
  (Close, s, e, ws, ps) → (Close, (s(|accounts := a|)), e, ws @
[ReduceNoWarning], ps @ [Payment p t m])" |
PayNonPositive: "evalValue e s v ≤ 0 ⟹
  (Pay aId r t v c, s, e, ws, ps) → (c, s, e, ws @ [ReduceNonPositivePay
aId r t (evalValue e s v)], ps)" |
PayPositivePartialWithPayment: "⟦evalValue e s v > 0;
  evalValue e s v > moneyInAccount aId t (accounts s);
  updateMoneyInAccount aId t 0 (accounts s) = a;
  moneyInAccount aId t (accounts s) = m;
  giveMoney r t (m) a = (payment, a2);
  payment = ReduceWithPayment p⟧ ⟹
  (Pay aId r t v c, s, e, ws, ps) → (c, s(|accounts := a2|), e, ws @
[ReducePartialPay aId r t m (evalValue e s v)], ps @ [p]))" |
PayPositivePartialWithoutPayment: "⟦evalValue e s v > 0;
  evalValue e s v > moneyInAccount aId t (accounts s);
  updateMoneyInAccount aId t 0 (accounts s) = newAccs;
  moneyInAccount aId t (accounts s) = moneyToPay;
  giveMoney r t (moneyToPay) newAccs = (payment, finalAccs);
  payment = ReduceNoPayment⟧ ⟹
  (Pay aId r t v cont, s, e, warns, payments) →
  ((cont, s(|accounts := finalAccs|), e, warns @ [ReducePartialPay aId r t
moneyToPay (evalValue e s v)], payments))" |
PayPositiveFullWithPayment: "⟦evalValue e s val > 0;
  evalValue e s val ≤ moneyInAccount accId token (accounts s);
  moneyInAccount accId token (accounts s) = moneyInAcc;
  moneyInAcc - (evalValue e s val) = newBalance;
  updateMoneyInAccount accId token newBalance (accounts s) = newAccs;
  giveMoney payee token (evalValue e s val) newAccs = (payment,
finalAccs);
  payment = ReduceWithPayment somePayment⟧ ⟹
  (Pay accId payee token val cont, s, e, warns, payments) →
  (cont, s(|accounts := finalAccs|), e, warns @ [ReduceNoWarning], payments
@ [somePayment])" |
```

Fig. 6. Marlowe small-step semantics part 1.

```
PayPositiveFullWithoutPayment: "⟦evalValue e s val > 0;
  evalValue e s val ≤ moneyInAccount accId token (accounts s);
  moneyInAccount accId token (accounts s) = moneyInAcc;
  moneyInAcc - (evalValue e s val) = newBalance;
  updateMoneyInAccount accId token newBalance (accounts s) = newAccs;
  giveMoney payee token (evalValue e s val) newAccs = (payment,
finalAccs);
  payment = ReduceNoPayment⟧ ⟹
  (Pay accId payee token val cont, s, e, warns, payments) →
  (cont, s(|accounts := finalAccs|), e, warns @ [ReduceNoWarning],
payments)" |
IfTrue: "evalObservation e s obs ⟹
  (If obs cont1 cont2, s, e, warns, payments) →
  (cont1, s, e, warns @ [ReduceNoWarning], payments)" |
IfFalse: "¬evalObservation e s obs ⟹
  (If obs cont1 cont2, s, e, warns, payments) →
  (cont2, s, e, warns @ [ReduceNoWarning], payments)" | WhenTimeout:
"⟦slotInterval e = (startSlot, endSlot);
  endSlot ≥ timeout;
  startSlot ≥ timeout⟧ ⟹
  (When cases timeout cont, s, e, warns, payments) →
  (cont, s, e, warns @ [ReduceNoWarning], payments)" |
LetShadow: "lookup valId (boundValues s) = Some oldVal ⟹
  (Let valId val cont, s, e, warns, payments) →
  (cont, s(| boundValues := MList.insert valId (evalValue e s val)
(boundValues s)|), e, warns @ [ReduceShadowing valId oldVal (evalValue
e s val)], payments)" |
LetNoShadow: "lookup valId (boundValues s) = None ⟹
  (Let valId val cont, s, e, warns, payments) →
  (cont, s(| boundValues := MList.insert valId (evalValue e s val)
(boundValues s)|), e, warns @ [ReduceNoWarning], payments)" |
AssertTrue: "evalObservation e s obs ⟹
  (Assert obs cont, s, e, warns, payments) →
  (cont, s, e, warns @ [ReduceNoWarning], payments)" |
AssertFalse: "¬evalObservation e s obs ⟹
  (Assert obs cont, s, e, warns, payments) →
  (cont, s, e, warns @ [ReduceAssertionFailed], payments)"
```

Fig. 7. Marlowe small-step semantics part 2.

```
fun wellTypedPubKey :: "TypeContext ⇒ FPubKey ⇒ bool" where
"wellTypedPubKey tyCtx (ConstantPubKey pk) = True" |
"wellTypedPubKey [] (UsePubKey pkId) = False" |
"wellTypedPubKey ((boundPkId, ty)#rest) (UsePubKey pkId) = ((boundPkId
= pkId ∧ ty = PubKeyType) ∨ (boundPkId ≠ pkId ∧ wellTypedPubKey rest
(UsePubKey pkId)))"

fun wellTypedPayee :: "TypeContext ⇒ FPayee ⇒ bool" where
"wellTypedPayee tyCtx (Account pk) = wellTypedPubKey tyCtx pk" |
"wellTypedPayee tyCtx (Party pk) = wellTypedPubKey tyCtx pk"

fun wellTypedChoiceId :: "TypeContext ⇒ FChoiceId ⇒ bool" where
"wellTypedChoiceId tyCtx (FChoiceId cname pk) = wellTypedPubKey tyCtx pk"

fun wellTypedValue :: "TypeContext ⇒ FValue ⇒ bool" and
  wellTypedObservation :: "TypeContext ⇒ FObservation ⇒ bool" where
"wellTypedValue tyCtx (AvailableMoney pk token) = wellTypedPubKey tyCtx
pk" |
"wellTypedValue tyCtx (Constant integer) = True" |
"wellTypedValue tyCtx (NegValue val) = wellTypedValue tyCtx val" |
"wellTypedValue tyCtx (AddValue lhs rhs) = ((wellTypedValue tyCtx lhs) ∧
(wellTypedValue tyCtx rhs))" |
"wellTypedValue tyCtx (SubValue lhs rhs) = ((wellTypedValue tyCtx lhs) ∧
(wellTypedValue tyCtx rhs))" |
"wellTypedValue tyCtx (MulValue lhs rhs) = ((wellTypedValue tyCtx lhs) ∧
(wellTypedValue tyCtx rhs))" |
"wellTypedValue tyCtx (Scale n d rhs) = wellTypedValue tyCtx rhs" |
"wellTypedValue tyCtx (ChoiceValue choId) = (wellTypedChoiceId tyCtx
choId)" |
"wellTypedValue tyCtx (SlotIntervalStart) = True" |
"wellTypedValue tyCtx (SlotIntervalEnd) = True" |
"wellTypedValue [] (UseValue valId) = False" |
"wellTypedValue ((boundValId, ty)#rest) (UseValue valId) = ((boundValId
= valId ∧ ty = ValueType) ∨ (boundValId ≠ valId ∧ wellTypedValue rest
(UseValue valId)))" |
"wellTypedValue tyCtx (Cond cond thn els) = (wellTypedObservation tyCtx
cond ∧ wellTypedValue tyCtx thn ∧ wellTypedValue tyCtx els)" |
"wellTypedObservation tyCtx (AndObs lhs rhs) = ((wellTypedObservation
tyCtx lhs) ∧ (wellTypedObservation tyCtx rhs))" |
"wellTypedObservation tyCtx (OrObs lhs rhs) = ((wellTypedObservation
tyCtx lhs) ∧ (wellTypedObservation tyCtx rhs))" |
"wellTypedObservation tyCtx (NotObs subObs) = wellTypedObservation tyCtx
subObs" |
"wellTypedObservation tyCtx (ChoseSomething choId) = (wellTypedChoiceId
tyCtx choId)" |
"wellTypedObservation tyCtx (ValueGE lhs rhs) = ((wellTypedValue tyCtx
lhs) ∧ (wellTypedValue tyCtx rhs))" |
"wellTypedObservation tyCtx (ValueGT lhs rhs) = ((wellTypedValue tyCtx
lhs) ∧ (wellTypedValue tyCtx rhs))" |
"wellTypedObservation tyCtx (ValueLT lhs rhs) = ((wellTypedValue tyCtx
lhs) ∧ (wellTypedValue tyCtx rhs))" |
"wellTypedObservation tyCtx (ValueLE lhs rhs) = ((wellTypedValue tyCtx
lhs) ∧ (wellTypedValue tyCtx rhs))" |
"wellTypedObservation tyCtx (ValueEQ lhs rhs) = ((wellTypedValue tyCtx
lhs) ∧ (wellTypedValue tyCtx rhs))" |
"wellTypedObservation [] (UseObservation obsId) = False" |
"wellTypedObservation ((boundObsId, ty)#rest) (UseObservation obsId) =
((boundObsId = obsId ∧ ty = ObservationType) ∨ (boundObsId ≠ obsId ∧
wellTypedObservation rest (UseObservation obsId)))" |
"wellTypedObservation tyCtx TrueObs = True" |
"wellTypedObservation tyCtx FalseObs = True"
```

Fig. 8. Faustus type-checker part 1.

```
fun wellTypedArgument :: "TypeContext ⇒ FParameter ⇒ FArgument ⇒ bool"
where
"wellTypedArgument tyCtx (ValueParameter vid) (ValueArgument val) =
wellTypedValue tyCtx val" |
"wellTypedArgument tyCtx (ObservationParameter obsId)
(ObservationArgument obs) = wellTypedObservation tyCtx obs" |
"wellTypedArgument tyCtx (PubKeyParameter pkId) (PubKeyArgument pk) =
wellTypedPubKey tyCtx pk" |
"wellTypedArgument tyCtx p a = False"

fun wellTypedArguments :: "TypeContext ⇒ FParameter list ⇒ FArgument
list ⇒ bool" where
"wellTypedArguments tyCtx [] [] = True" |
"wellTypedArguments tyCtx p [] = False" |
"wellTypedArguments tyCtx [] a = False" |
"wellTypedArguments tyCtx (firstParam#restParams) (firstArg#restArgs) =
(wellTypedArgument tyCtx firstParam firstArg ∧ wellTypedArguments tyCtx
restParams restArgs)"

fun lookupContractIdParamTypeInformation :: "TypeContext ⇒ Identifier ⇒
(FParameter list × TypeContext) option" where
"lookupContractIdParamTypeInformation [] cid = None" |
"lookupContractIdParamTypeInformation ((i, ty)#restTy) cid = (if i = cid
  then case ty of
    ContractType params ⇒ Some (params, restTy)
    | _ ⇒ None
  else lookupContractIdParamTypeInformation restTy cid)"

fun wellTypedContract :: "TypeContext ⇒ FContract ⇒ bool" and
  wellTypedCases :: "TypeContext ⇒ FCase list ⇒ bool" where
"wellTypedContract tyCtx Close = True" |
"wellTypedContract tyCtx (Pay accId payee token val cont) =
  (wellTypedPubKey tyCtx accId ∧ wellTypedPayee tyCtx payee ∧
wellTypedValue tyCtx val ∧ wellTypedContract tyCtx cont)" |
"wellTypedContract tyCtx (If obs cont1 cont2) =
  (wellTypedObservation tyCtx obs ∧ wellTypedContract tyCtx cont1 ∧
wellTypedContract tyCtx cont2)" |
"wellTypedContract tyCtx (When cases t cont) = (wellTypedCases tyCtx
cases ∧ wellTypedContract tyCtx cont)" |
"wellTypedContract tyCtx (Assert obs cont) = (wellTypedObservation tyCtx
obs ∧ wellTypedContract tyCtx cont)" |
"wellTypedContract tyCtx (Let valId val cont) = (wellTypedValue tyCtx val
∧ wellTypedContract ((valId, ValueType)#tyCtx) cont)" |
"wellTypedContract tyCtx (LetObservation obsId obs cont) =
(wellTypedObservation tyCtx obs ∧ wellTypedContract ((obsId,
ObservationType)#tyCtx) cont)" |
"wellTypedContract tyCtx (LetPubKey pkId pk cont) = (wellTypedPubKey
tyCtx pk ∧ wellTypedContract ((pkId, PubKeyType)#tyCtx) cont)" |
"wellTypedContract tyCtx (LetC cid params body cont) = (wellTypedContract
((paramsToTypeContext params)@tyCtx) body ∧ wellTypedContract ((cid,
ContractType params)#tyCtx) cont)" |
"wellTypedContract tyCtx (UseC cid args) = (case
lookupContractIdParamTypeInformation tyCtx cid of
  Some (params, innerTyCtx) ⇒ wellTypedArguments tyCtx params args
  | _ ⇒ False)" |
"wellTypedCases tyCtx [] = True" |
"wellTypedCases tyCtx ((Case (Deposit fromPk toPk token value) c)#rest) =
  ((wellTypedPubKey tyCtx fromPk) ∧ (wellTypedPubKey tyCtx toPk)
∧ (wellTypedValue tyCtx value) ∧ (wellTypedContract tyCtx c) ∧
(wellTypedCases tyCtx rest))" |
```

Fig. 9. Faustus type-checker part 2.

```
"wellTypedCases tyCtx ((Case (Choice choiceId bounds) c)#rest) =
((wellTypedChoiceId tyCtx choiceId) ∧ (wellTypedContract tyCtx c) ∧
(wellTypedCases tyCtx rest))" |
"wellTypedCases tyCtx ((Case (Notify observation) c)#rest) =
((wellTypedObservation tyCtx observation) ∧ (wellTypedContract tyCtx
c) ∧ (wellTypedCases tyCtx rest))"

fun wellTypedContext :: "TypeContext ⇒ FContext ⇒ bool" where
"wellTypedContext [] [] = True" |
"wellTypedContext tyCtx [] = False" |
"wellTypedContext [] ctx = False" |
"wellTypedContext ((tyValId, ValueType)#restTypes) ((ValueAbstraction
absValId)#restAbstractions) =
  (tyValId = absValId ∧ wellTypedContext restTypes restAbstractions)" |
"wellTypedContext ((tyPkId, PubKeyType)#restTypes) ((PubKeyAbstraction
absPkId)#restAbstractions) =
  (tyPkId = absPkId ∧ wellTypedContext restTypes restAbstractions)" |
"wellTypedContext ((tyObsId, ObservationType)#restTypes)
((ObservationAbstraction absObsId)#restAbstractions) =
  (tyObsId = absObsId ∧ wellTypedContext restTypes restAbstractions)" |
"wellTypedContext ((tyCid, ContractType tyParams)#restTypes)
((ContractAbstraction (absCid, absParams, absBody))#restAbstractions)
=
  (tyCid = absCid ∧ tyParams = absParams ∧ wellTypedContract
((paramsToTypeContext absParams)@restTypes) absBody ∧ wellTypedContext
restTypes restAbstractions)" |
"wellTypedContext (someType#restTypes) (someAbs#restAbs) = False"

fun wellTypedState :: "TypeContext ⇒ FState ⇒ bool" where
"wellTypedState [] s = True" |
"wellTypedState ((valId, ValueType)#restTypes) s = (member valId
(boundValues s) ∧ wellTypedState restTypes s)" |
"wellTypedState ((obsId, ObservationType)#restTypes) s = (member obsId
(boundValues s) ∧ wellTypedState restTypes s)" |
"wellTypedState ((pkId, PubKeyType)#restTypes) s = (member pkId
(boundPubKeys s) ∧ wellTypedState restTypes s)" |
"wellTypedState ((cid, ContractType params)#restTypes) s = wellTypedState
restTypes s"
```

Fig. 10. Faustus type-checker part 3.

```
fun compileFPubKey :: "FPubKey ⇒ FState ⇒ PubKey option" where
"compileFPubKey (UsePubKey pkId) fs = MList.lookup pkId (boundPubKeys
fs)" |
"compileFPubKey (ConstantPubKey pk) s = Some pk"

fun compileFPayee :: "FPayee ⇒ FState ⇒ Payee option" where
"compileFPayee (Account pk) fs = (case compileFPubKey pk fs of
  Some mPk ⇒ Some (Semantics.Account mPk) | None ⇒ None)" |
"compileFPayee (Party pk) fs = (case compileFPubKey pk fs of
  Some mPk ⇒ Some (Semantics.Party mPk) | None ⇒ None)"

fun compileFChoiceId :: "FChoiceId ⇒ FState ⇒ ChoiceId option" where
"compileFChoiceId (FChoiceId cname pk) fs = (case compileFPubKey pk fs of
  Some mPk ⇒ Some (ChoiceId cname mPk) | None ⇒ None)"

fun compileFValue :: "FValue ⇒ FState ⇒ Value option" and
  compileFObservation :: "FObservation ⇒ FState ⇒ Observation option"
where
"compileFValue (AvailableMoney fPk token) s = (case compileFPubKey fPk s
of
  Some mPk ⇒ Some (Semantics.AvailableMoney mPk token)
  | None ⇒ None)" |
"compileFValue (Constant v) s = Some (Semantics.Constant v)" |
"compileFValue (NegValue v) s = (case compileFValue v s of
  Some mv ⇒ Some (Semantics.NegValue mv)
  | None ⇒ None)" |
"compileFValue (AddValue lhs rhs) s = (case (compileFValue lhs s,
compileFValue rhs s) of
  (Some compLhs, Some compRhs) ⇒ Some (Semantics.AddValue compLhs
compRhs)
  | _ ⇒ None)" |
"compileFValue (SubValue lhs rhs) s = (case (compileFValue lhs s,
compileFValue rhs s) of
  (Some compLhs, Some compRhs) ⇒ Some (Semantics.SubValue compLhs
compRhs)
  | _ ⇒ None)" |
"compileFValue (MulValue lhs rhs) s = (case (compileFValue lhs s,
compileFValue rhs s) of
  (Some compLhs, Some compRhs) ⇒ Some (Semantics.MulValue compLhs
compRhs)
  | _ ⇒ None)" |
"compileFValue (Scale i1 i2 val) s = (case (compileFValue val s) of
  Some compVal ⇒ Some (Semantics.Scale i1 i2 compVal)
  | None ⇒ None)" |
"compileFValue (ChoiceValue choiceId) s = (case compileFChoiceId choiceId
s of
  Some compChoiceId ⇒ Some (Semantics.ChoiceValue compChoiceId)
  | None ⇒ None)" |
"compileFValue SlotIntervalStart s = Some Semantics.SlotIntervalStart" |
"compileFValue SlotIntervalEnd s = Some Semantics.SlotIntervalEnd" |
"compileFValue (UseValue valId) s = Some (Semantics.UseValue valId)" |
"compileFValue (Cond obs trueVal falseVal) s = (case (compileFObservation
obs s, compileFValue trueVal s, compileFValue falseVal s) of
  (Some compObs, Some compTrueVal, Some compFalseVal) ⇒ Some
(Semantics.Cond compObs compTrueVal compFalseVal)
  | _ ⇒ None)" |
```

Fig. 11. Faustus compiler part 1.

```
"compileFObservation (AndObs lhs rhs) s = (case (compileFObservation lhs
s, compileFObservation rhs s) of
  (Some compLhs, Some compRhs) ⇒ Some (Semantics.AndObs compLhs compRhs)
  | _ ⇒ None)" |
"compileFObservation (OrObs lhs rhs) s = (case (compileFObservation lhs
s, compileFObservation rhs s) of
  (Some compLhs, Some compRhs) ⇒ Some (Semantics.OrObs compLhs compRhs)
  | _ ⇒ None)" |
"compileFObservation (NotObs obs) s = (case compileFObservation obs s of
  Some compObs ⇒ Some (Semantics.NotObs compObs) | None ⇒ None)" |
"compileFObservation (ValueGE lhs rhs) s = (case (compileFValue lhs s,
compileFValue rhs s) of
  (Some compLhs, Some compRhs) ⇒ Some (Semantics.ValueGE compLhs
compRhs)
  | _ ⇒ None)" |
"compileFObservation (ValueGT lhs rhs) s = (case (compileFValue lhs s,
compileFValue rhs s) of
  (Some compLhs, Some compRhs) ⇒ Some (Semantics.ValueGT compLhs
compRhs)
  | _ ⇒ None)" |
"compileFObservation (ValueLT lhs rhs) s = (case (compileFValue lhs s,
compileFValue rhs s) of
  (Some compLhs, Some compRhs) ⇒ Some (Semantics.ValueLT compLhs
compRhs)
  | _ ⇒ None)" |
"compileFObservation (ValueLE lhs rhs) s = (case (compileFValue lhs s,
compileFValue rhs s) of
  (Some compLhs, Some compRhs) ⇒ Some (Semantics.ValueLE compLhs
compRhs)
  | _ ⇒ None)" |
"compileFObservation (ValueEQ lhs rhs) s = (case (compileFValue lhs s,
compileFValue rhs s) of
  (Some compLhs, Some compRhs) ⇒ Some (Semantics.ValueEQ compLhs
compRhs)
  | _ ⇒ None)" |
"compileFObservation (UseObservation obsId) s = Some (Semantics.NotObs
(Semantics.ValueEQ (Semantics.UseValue obsId) (Semantics.Constant 0)))" |
"compileFObservation (ChoseSomething cid) s = (case compileFChoiceId cid
s of
  Some compChoiceId ⇒ Some (Semantics.ChoseSomething compChoiceId)
  | None ⇒ None)" |
"compileFObservation TrueObs s = Some Semantics.TrueObs" |
"compileFObservation FalseObs s = Some Semantics.FalseObs"

fun compileAction :: "FAction ⇒ FState ⇒ Semantics.Action option"
where
"compileAction (Deposit fromPk toPk token value) s = (case
(compileFPubKey fromPk s, compileFPubKey toPk s, compileFValue value
s) of
  (Some compiledFromPk, Some compiledToPk, Some compiledValue) ⇒ Some
(Semantics.Deposit compiledFromPk compiledToPk token compiledValue)
  | _ ⇒ None)" |
"compileAction (Choice choiceId bounds) s = (case compileFChoiceId
choiceId s of
  Some compChoiceId ⇒ Some (Semantics.Choice compChoiceId bounds)
  | None ⇒ None)" |
"compileAction (Notify observation) s = (case compileFObservation
observation s of
  Some compiledObservation ⇒ Some (Semantics.Notify compiledObservation)
  | None ⇒ None)"
```

Fig. 12. Faustus compiler part 2.

```
fun argumentsToCompilerState :: "FParameter list ⇒ FArgument list ⇒
FState ⇒ FState option" where
"argumentsToCompilerState [] [] fs = Some fs" |
"argumentsToCompilerState params [] fs = None" |
"argumentsToCompilerState [] args fs = None" |
"argumentsToCompilerState (PubKeyParameter pkId#restParams)
(PubKeyArgument pk#restArgs) fs =
  (case compileFPubKey pk fs of
    Some pkCompiled ⇒ argumentsToCompilerState restParams restArgs
(fs(|boundPubKeys := MList.insert pkId pkCompiled (boundPubKeys fs)|))

    | None ⇒ None)" |
"argumentsToCompilerState (ValueParameter valId#restParams)
(ValueArgument val#restArgs) ctx = argumentsToCompilerState restParams
restArgs (ctx(|boundValues := MList.insert valId 0 (boundValues ctx)|))" |
"argumentsToCompilerState (ObservationParameter obsId#restParams)
(ObservationArgument obs#restArgs) ctx = argumentsToCompilerState
restParams restArgs (ctx(|boundValues := MList.insert obsId 0 (boundValues
ctx)|))" |
"argumentsToCompilerState (firstParam#restParams) (firstArg#restArgs) ctx
= None"

fun compileArguments :: "FParameter list ⇒ FArgument list ⇒ FState ⇒
Contract ⇒ Contract option" where
"compileArguments [] [] fs cont = Some cont" |
"compileArguments params [] fs cont = None" |
"compileArguments [] args fs cont = None" |
"compileArguments (ValueParameter valId#restParams) (ValueArgument
val#restArgs) fs cont =
  (case compileFValue val fs of
  Some valCompiled ⇒ (case compileArguments restParams restArgs
(fs(|boundValues := MList.insert valId 0 (boundValues fs)|)) cont of
    Some compiledRest ⇒ Some (Semantics.Let valId valCompiled
compiledRest)
    | None ⇒ None)
  | None ⇒ None)" |
"compileArguments (ObservationParameter obsId#restParams)
(ObservationArgument obs#restArgs) fs cont =
  (case compileFObservation obs fs of
  Some obsCompiled ⇒ (case compileArguments restParams restArgs
(fs(|boundValues := MList.insert obsId 0 (boundValues fs)|)) cont of
    Some compiledRest ⇒ Some (Semantics.Let obsId (Semantics.Cond
obsCompiled (Semantics.Constant 1) (Semantics.Constant 0)) compiledRest)
    | None ⇒ None)
  | None ⇒ None)" |
"compileArguments (PubKeyParameter pkId#restParams) (PubKeyArgument
pk#restArgs) fs cont =
  (case compileFPubKey pk fs of
  Some pkCompiled ⇒ compileArguments restParams restArgs
(fs(|boundPubKeys := MList.insert pkId pkCompiled (boundPubKeys fs)|))
cont
  | None ⇒ None)" |
"compileArguments (firstParam#restParams) (firstArg#restArgs) ctx cont =
None"
```

Fig. 13. Faustus compiler part 3.

```
function compile :: "FContract ⇒ FContext ⇒ FState ⇒ Contract option"
  and compileCases :: "FCase list ⇒ FContext ⇒ FState ⇒
Semantics.Case list option" where
"compile Close ctx state = Some Semantics.Close"|
"compile (Pay accId payee tok val cont) ctx state = (let innerCompilation
= compile cont ctx state in
  let compiledAccIdOption = compileFPubKey accId state in
  let compiledPayeeOption = compileFPayee payee state in
  let compiledValueOption = compileFValue val state in
  case (innerCompilation, compiledAccIdOption, compiledPayeeOption,
compiledValueOption) of
    (Some innerCompiled, Some compiledAccId, Some compiledPayee, Some
compiledValue) ⇒ Some (Semantics.Pay compiledAccId compiledPayee tok
compiledValue innerCompiled) | _ ⇒ None)"|
"compile (If obs trueCont falseCont) ctx state = (let trueCompilation =
compile trueCont ctx state in
  let falseCompilation = compile falseCont ctx state in
  let obsCompilation = compileFObservation obs state in
  case (trueCompilation, falseCompilation, obsCompilation) of
    (Some trueCompiled, Some falseCompiled, Some obsCompiled) ⇒ Some
(Semantics.If obsCompiled trueCompiled falseCompiled) | _ ⇒ None)"|
"compile (When cases t tCont) ctx state = (
    let compiledT = compile tCont ctx state in
    let newCases = compileCases cases ctx state in
    case (compiledT, newCases) of
      (Some compiledTCont, Some compiledCases) ⇒ Some (Semantics.When
compiledCases t compiledTCont) | _ ⇒ None)"|
"compile (Let valId val cont) ctx state = (let innerCompilation = compile
cont ((ValueAbstraction valId)#ctx) (state(|boundValues := MList.insert
valId 0 (boundValues state)|)) in
  let valCompilation = compileFValue val state in
  case (innerCompilation, valCompilation) of
    (Some innerCompiled, Some valCompiled) ⇒ Some (Semantics.Let valId
valCompiled innerCompiled) | _ ⇒ None)"|
"compile (LetObservation obsId obs cont) ctx state = (let
innerCompilation = compile cont ((ObservationAbstraction obsId)#ctx)
(state(|boundValues := MList.insert obsId 0 (boundValues state)|)) in
  let obsCompilation = compileFObservation obs state in
  case (innerCompilation, obsCompilation) of
    (Some innerCompiled, Some obsCompiled) ⇒ Some (Semantics.Let obsId
(Semantics.Cond obsCompiled (Semantics.Constant 1) (Semantics.Constant
0)) innerCompiled) | _ ⇒ None)"|
"compile (LetPubKey pkId pk cont) ctx state = (case compileFPubKey pk
state of
  Some pkCompiled ⇒ compile cont ((PubKeyAbstraction pkId)#ctx)
(state(|boundPubKeys := MList.insert pkId pkCompiled (boundPubKeys
state)|))
  | None ⇒ None)"|
```

Fig. 14. Faustus compiler part 4.

```
"compile (Assert obs cont) ctx state = (let innerCompilation = compile
cont ctx state in
  let obsCompilation = compileFObservation obs state in
  case (innerCompilation, obsCompilation) of
    (Some innerCompiled, Some obsCompiled) ⇒ Some (Semantics.Assert
obsCompiled innerCompiled)
  | _ ⇒ None)"|
"compile (LetC cid params boundCon cont) ctx state = compile cont
((ContractAbstraction (cid, params, boundCon))#ctx) state" |
"compile (UseC cid args) ctx state = (case lookupContractIdAbsInformation
ctx cid of
  Some (params, bodyCont, innerCtx) ⇒ (case argumentsToCompilerState
params args state of
    Some newState ⇒ (case compile bodyCont ((paramsToFContext
params)@innerCtx) newState of
      Some bodyCompiled ⇒ compileArguments params args state
bodyCompiled
      | None ⇒ None)
    | None ⇒ None)
  | None ⇒ None)" |
"compileCases ((Case a c) # rest) ctx state =
  (case (compile c ctx state, compileCases rest ctx state, compileAction
a state) of
    (Some compiledCaseCont, Some compiledCases, Some compiledAction) ⇒
Some ((Semantics.Case compiledAction compiledCaseCont) # compiledCases)
    | _ ⇒ None)" |
"compileCases [] ctx state = Some []"
```

Fig. 15. Faustus compiler part 5.