Faustus

Communicating Sequential Processes for Compact Representation of Very Large Inline Programs*

Kegan McIlwaine / Ph.D. Student of Computer Science / September 13, 2023
In collaboration with Professor James Caldwell

*This work was funded by a grant from IOG, Wyoming state funds, and the University of Wyoming

Marlowe

- Marlowe is a domain specific language (DSL) for writing financial smart contracts on the blockchain.
- Marlowe lacks traditional control flow operators (other than **When** or **If** branching). Control is explicitly passed using a continuation passing model.
- All Marlowe programs are fully expanded instances of the syntax tree, *i.e.* they are inline programs that always terminate.
- A Marlowe contract executes until closed, or it requires user input that hasn't occurred before timing out.

```
data Contract = Close
              | Pay Party Payee Token Value Contract
              | If Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract
              | Assert Observation Contract
```

A Marlowe Contract

This is a zero coupon bond contract where one user, the “Lender”, sends a loan to another user, the “Borrower”. The “Borrower” later pays the loan back with interest added.

Some simple auction contracts end up being over one gigabyte of text.

```
When [
  (Case
     (Deposit
        (Role "Lender")
        (Role "Lender")
        (Token "" "")
        (ConstantParam "Amount"))
     (Pay
        (Role "Lender")
        (Party
           (Role "Borrower"))
        (Token "" "")
        (ConstantParam "Amount")
        (When [
           (Case
              (Deposit
                 (Role "Borrower")
                 (Role "Borrower")
                 (Token "" "")
                 (AddValue
                    (ConstantParam "Interest")
                    (ConstantParam "Amount")))
              (Pay
                 (Role "Borrower")
                 (Party
                    (Role "Lender"))
                 (Token "" "")
                 (AddValue
                    (ConstantParam "Interest")
                    (ConstantParam "Amount")) Close))]
           (TimeParam "Payback deadline") Close))))]
  (TimeParam "Loan deadline") Close
```

Communicating Sequential Processes (CSP)

- CSP (Hoare, 1978) provided the first formal framework for defining processes and how they evolve and interact concurrently.
- Concurrency is represented by all possible linear interleavings of actions.
- A process can be written in the form (x:B -> F(x)) where B is a set of actions and F(x) is a process.
- CSP has additional operators for *choice* and *interleaving*. Written <+> and ||| respectively.
- Pseudo-code for the zero coupon bond contract could be written:

> *lender* deposits *amount* into *borrower* account ->
> emit payment to *borrower* of *amount* ->
> *borrower* deposits *amount* + *interest* into *lender* account ->
> emit payment to *lender* of *amount* + *interest*

Faustus Features

- A concrete syntax separated from a higher level language for better readability.
- Parameterized abstractions of all meaningful syntactic classes.
  - Contracts, Observations, Parties, Cases, PosixTimes, Actions, Choices, and Tokens.
- PosixTime expressions for easier relative timing within the contract.
  - Reduce manual calculation of exact times for every timeout in the contract.
- Combinatorial operators from CSP for easier sequencing of Actions.
  - Marlowe requires explicitly programming every possible order of user input that is allowed.
  - Choices between two sequences of Actions.
  - All possible orderings of Actions.
  - Strict orderings of Actions.

Example Program - Multisignature Contract with 5 Voters

```
when {
      "alice" deposits 1000 "" into @"carol" -> {
            var votes = 0;
            when {
                        ("voter1" chooses "agree" -> votes := votes + 1 <+> "voter1" chooses not "agree") |||
                        ("voter2" chooses "agree" -> votes := votes + 1 <+> "voter2" chooses not "agree") |||
                        ("voter3" chooses "agree" -> votes := votes + 1 <+> "voter3" chooses not "agree") |||
                        ("voter4" chooses "agree" -> votes := votes + 1 <+> "voter4" chooses not "agree") |||
                        ("voter5" chooses "agree" -> votes := votes + 1 <+> "voter5" chooses not "agree") ->{
                              if votes >= 3
                              then @"carol" pays !"bob" (1000, ""); close
                              else @"carol" pays !"alice" (1000, ""); close
                        }
            } after 180 -> { if votes >= 3
                              then @"carol" pays !"bob" (1000, ""); close
                              else @"carol" pays !"alice" (1000, ""); close
            }
      }
} after 100 -> { close }
```

Expands to 100,000+ lines of Marlowe code using standard formatting.

Questions