Formalizing BitML Calculus in Agda

Orestis Melkonian
Utrecht University, The Netherlands

Motivation

- Blockchain technology has opened a whole array of interesting new applications, mainly due to the sophisticated transaction schemes enabled by **smart contracts** – programs that run on the blockchain.
- Reasoning about their behaviour is:
  - *necessary*, significant funds are involved
  - *difficult*, due to concurrency
- Provide rigid foundations via a language-based, type-driven approach, alongside a mechanized meta-theory.
- Formalization of the *Bitcoin Modelling Language* (BitML).

BitML Contracts

- The type of a contract is indexed by the total monetary value it carries and a set of deposits that guarantee it will not get stuck. A contract can have multiple branches using the binary operator _ ⊕ _.

```
data Contract : Value → Values → Set where
  -- collect deposits and secrets
  put _ reveal _ ⇒ _ ⊣ _ :
        (vs : Values) → Secrets → Contract (v + sum vs) vs′
     → Contract v (vs′ ++ vs)
  -- transfer the remaining balance to a participant
  withdraw : ∀ {v vs} → Participant → Contract v vs
  -- split the balance across different branches
  split : ∀ {vs} → (cs : List (∃[v] Contract v vs))
     → Contract (sum (proj₁ ⟨$⟩ cs)) vs
  -- wait for participant's authorization
  _ : _ : Participant → Contract v vs → Contract v vs
  -- wait until some time passes
  after _ : _ : Time → Contract v vs → Contract v vs
```

- A contract is initially made public through an *Advertisement*, denoted ⟨ G ⟩C, where G are the preconditions that have to be met in order for C to be stipulated.

Small-step Semantics

- BitML's semantics describes transitions between *configurations*, which hold advertisements, deposits, contracts, secrets and action authorizations. For the sake of semantic bug discovery, configurations are indexed by assets of type (*List A* , *List A*), where the first element is produced and the second required:

```
data Configuration′ : Asset ∃Advertisement -- advertised contracts
                    → Asset ∃Contract      -- stipulated contracts
                    → Asset Deposit        -- deposits
                    → Set
```

- The small-step relation is a collection of permitted transitions between our intrinsically-typed states:

```
data _ ⟶ _ : Configuration ads cs ds → Configuration ads′cs′ds′ → Set where
  D-AuthJoin :
    ────────────────────────────────────────────────────────
    ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | Γ ⟶ ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | A [0 ↔ 1] | Γ

  D-Join :
    ────────────────────────────────────────────────────
    ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | A [0 ↔ 1] | Γ ⟶ ⟨ A , v + v′ ⟩ᵈ | Γ

  C-Advertise :
    Any (_∈ Hon) (participants (G ad))
    ──────────────────────────────
    Γ ⟶ ad | Γ

  C-AuthCommit :
    (secrets A (G ad) ≡ a₀ . . . aₙ) × (A ∈ Hon → All (_ ≢ nothing) aᵢ)
    ──────────────────────────────────────────────────────────
    ‘ad | Γ ⟶ ‘ad | Γ | . . . ⟨ A : aᵢ # Nᵢ ⟩ . . . | A [♯▷ ad]
  ⋮
```

Equational Reasoning

- Rules are always presented with the interesting parts of the state as the left operand, implicitly relying on (*Configuration* , _|_) being a *commutative monoid*.

Solution Factor out reordering in the *reflective transitive closure* of _ ⟶ _:

```
data _ ⟶* _ : Configuration ads cs ds → Configuration ads′cs′ds′ → Set where
  _∎ : (M : Configuration ads cs ds) → M ⟶* M
  _ ⟶ ⟨ _ ⟩_ : (L : Configuration ads cs ds) {_ : L ≈ L′ × M ≈ M′}
        → (L′ ⟶ M′) → (M ⟶* N) → (L ⟶* N)
```

Example Derivation

- **Timed-commitment protocol**
  A promises to reveal a secret to B, otherwise loses a deposit of B 1.

```
tc : Advertisement 1 [] [] [1 , 0])
tc = ⟨ A ! 1 ∧ A # a ∧ B ! 0 ⟩reveal [a] ⇒ withdraw A ⊣ . . . ⊕ after t : withdraw B
```

- The following proof exhibits a possible execution, where A reveals the secret:

```
tc-derivation : ⟨ A , 1 ⟩ᵈ ⟶* ⟨ A , 1 ⟩ᵈ | A : a # 6
tc-derivation =       ⟨ A , 1 ⟩ᵈ
  ⟶⟨ C-Advertise ⟩  ‘tc | ⟨ A , 1 ⟩ᵈ
  ⟶⟨ C-AuthInit ⟩   ‘tc | ⟨ A , 1 ⟩ᵈ         | ⟨ A : a # 6 ⟩ | A [♯▷ tc]
  ⟶⟨ C-Init ⟩       ⟨ tc , 1 ⟩ᶜ              | ⟨ A : a # 6 ⟩
  ⟶⟨ C-AuthRev ⟩    ⟨ tc , 1 ⟩ᶜ              | A : a # 6
  ⟶⟨ C-Control ⟩    ⟨ [reveal . . .] , 1 ⟩ᶜ  | A : a # 6
  ⟶⟨ C-PutRev ⟩     ⟨ [withdraw A] , 1 ⟩ᶜ    | A : a # 6
  ⟶⟨ C-Withdraw ⟩   ⟨ A , 1 ⟩ᵈ               | A : a # 6
  ∎
```

Symbolic Model

- What we eventually want is to reason about the behaviour of participants. By observing that our small-step derivations correspond to possible execution *traces*, we can develop a game-theoretic view of our semantics by considering *strategies*, in which participants make moves depending on the current trace.
- **Honest participants** can pick a set of possible next moves, which have to adhere to certain validity conditions (e.g. the move has to be permitted by the semantics).

```
record HonestStrategy (A : Participant) where
  field strategy : Trace → Labels
        valid    : (A ∈ Hon)
                 × (∀ R α → α ∈ strategy R * → ∃[R′] (R ↣⟦ α ⟧ R′))
                 × (∀ R α → α ∈ strategy R * → authorizers α ⊆ [A])
                 ⋮
```

- An **adversary** will choose the final move, out of the honest ones:

```
record AdversaryStrategy (Adv : Participant) where
  field strategy : Trace → (∀ (A : Participant) → HonestStrategy A) → Label
        valid    : (Adv ∉ Hon)
                 × ∀ {R : Trace} {honestMoves} →
                     let α = strategy R * honestMoves in
                     (  ∃[A] (A ∈ Hon × α ∈ honestMoves A)
                     ⊎ . . . )
```

- We can now demonstrate a possible **attack** by proving that a given trace *conforms* to a specific set of strategies, i.e. we can arrive there from an initial configuration using the moves emitted by the strategies.

Towards Certified Compilation

- Originally, the BitML proposal involved a compilation scheme from BitML contracts to Bitcoin transactions, accompanied by a proof that attacks in the compiled contracts can always be observed in the symbolic model.
  However, we aim to give a compiler to a more abstract accounting model for ledgers based on *unspent output transactions* (UTxO) and mechanize a similar *compilation correctness* proof.
- We already have an Agda formalization for dependently-typed UTxO ledgers, which statically enforces the validity of their transactions (e.g. all referenced addresses exist) at **https://github.com/omelkonian/formal-utxo**.

Universiteit Utrecht   INPUT | OUTPUT   https://github.com/omelkonian/formal-bitml