SECTION 1

Formal Model II: BitML Calculus

Now let us shift our focus to our second subject of study, the BitML calculus for modelling smart contracts. In this subsection we sketch the formalized part of BitML we have covered so far, namely the syntax and small-step semantics of BitML contracts, as well as an example execution of a contract under these semantics. All code is publicly available on Github[1].

First, we begin with some basic definitions that will be used throughout this section. Instead of giving a fixed datatype of participants, we parametrise our module with a given *asbtract datatype* of participants that we can check for equality, as well as non-empty list of honest participants:

```
module BitML (Participant : Set) (_≟_ : Decidable {A = Participant} _≡_)
             (Honest : List⁺ Participant)
             where

Time : Set
Time = ℕ

Value : Set
Value = ℕ

Secret : Set
Secret = String

record Deposit : Set where
  constructor _has_
  field participant : Participant
        value : Value
```

Representation of time and monetary values is again simplistic, both modelled as natural numbers. while we model participant secrets as simple strings[2]. A deposit consists of the participant that owns it and the number of bitcoins it carries.

[1] https://github.com/omelkonian/formal-bitml

[2] Of course, one could provide more realistic types (e.g. words of specific length) to be closer to the implementation, as shown for the UTxO model in Section **??**.

We, furthermore, introduce a simplistic language of logical predicates and arithmetic expressions with the usual constructs (e.g. numerical addition, logical conjunction) and give the usual semantics (predicates on booleans and arithmetic on naturals). A more unusual feature of these expressions is the ability to calculate length of secrets (within arithmetic expressions) and, in order to ensure more type safety later on, all expressions are indexed by the secrets they internally use.

```
data Arith : List Secret → Set where
  `_ : ℕ → Arith []
  `len : (s : Secret) → Arith [ s ]
  _`+_ : Arith s₁ → Arith s₂ → Arith (s₁ ++ s₂)
  _`-_ : Arith s₁ → Arith s₂ → Arith (s₁ ++ s₂)

ℕ⟦ _ ⟧ : ∀ {s} → Arith s → ℕ
ℕ⟦ _ ⟧ = ...

data Predicate : List Secret → Set where
  `True : Predicate []
  _`∧_ : Predicate s₁ → Predicate s₂ → Predicate (s₁ ++ s₂)
  `¬_ : ∀ {s} → Predicate s → Predicate s
  _`≡_ : Arith s₁ → Arith s₂ → Predicate (s₁ ++ s₂)
  _`<_ : Arith s₁ → Arith s₂ → Predicate (s₁ ++ s₂)

𝔹⟦ _ ⟧ : ∀ {s} → Predicate s → Bool
𝔹⟦ _ ⟧ = ...
```

1.1 Contracts in BitML

A *contract advertisement* consists of a set of *preconditions*, which require some resources from the involved participants prior to the contract's execution, and a *contract*, which specifies the rules according to which bitcoins are transferred between participants.

Preconditions either require participants to have a deposit of a certain value on their name (volatile or not) or commit to a certain secret. A *persistent* deposit has to be provided before the contract is stipulated, while a *volatile* deposit may be needed dynamically during the execution of the contract. Both volatile and persistent deposits required by a precondition are captured in its two type-level indices, respectively:

```
data Precondition : List Value → List Value → Set where
  -- volatile deposit
  _?_ : Participant → (v : Value) → Precondition [ v ] []
```

```
  -- persistent deposit
  _!_ : Participant → (v : Value) → Precondition [] [ v ]

  -- committed secret
  _#_ : Participant → Secret → Precondition [] []

  -- conjunction
  _∧_ : Precondition vsᵛ vsᵖ → Precondition vsᵛ′ vsᵖ′
        → Precondition (vsᵛ ++ vsᵛ′) (vsᵖ ++ vsᵖ′)
```

Moving on to actual contracts, we define them by means of a collection of five types of commands; *put* injects participant deposits and revealed secrets in the remaining contract, *withdraw* transfers the current funds to a participant, *split* distributes the current funds across different individual contracts, _ : _ requires the authorization from a participant to proceed and *after* _ : _ allows further execution of the contract only after some time has passed.

```
data Contract : Value       -- the monetary value it carries
              → List Value  -- the volatile deposits it presumes
              → Set where

  -- collect deposits and secrets
  put _ reveal _ if _ ⇒ _ ⊣ _ : ∀ {s′ : List Secret} {
    → (vs : List Value) → (s : List Secret) → Predicate s′ → Contract (v + sum vs) vs′
    → s′ ⊆ s
    → 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
```

There is a lot of type-level manipulation across all constructors, since we need to make sure that indices are calculated properly. For instance, the total value in a contract constructed by the *split* command is the sum of the values carried by each branch. The *put* command[3] additionally requires an explicit proof that the predicate of the *if* part only uses secrets revealed by the same command.

[3] *put* comprises of several components and we will omit those that do not contain any helpful information, e.g. write *put* _ ⇒ _ when there are no revealed secrets and the predicate trivially holds.

We also introduce an intuitive syntax for declaring the different branches of a *split* command, emphasizing the *linear* nature of the contract's total monetary value:

```
_ ⊸ _ : ∀ {vs} → (v : Value) → Contract v vs → ∃[ v ] Contract v vs
v ⊸ c = v , c
```

Having defined both preconditions and contracts, we arrive at the definition of a contract advertisement:

```
record Advertisement (v : Value) (vsᶜ vsᵛ vsᵖ : List Value) : Set where
  constructor _⟨ _ ⟩⊣ _
  field G     : Precondition vsᵛ vsᵖ
        C     : Contracts v vsᶜ
        valid : length vsᶜ ⩽ length vsᵛ
              × participantsᵍ G ++ participantsᶜ C ⊆ (participant ⟨$⟩ persistentDeposits G)
```

Notice that in order to construct an advertisement, one has to also provide proof of the contract's validity with respect to the given preconditions, namely that all deposit references in the contract are declared in the precondition and each involved participant is required to have a persistent deposit.

To clarify things so far, let us see a simple example of a contract advertisement. We first open the *BitML* module with a trivial datatype for participants, consisting of *A* and *B*:

```
data Participant : Set where
  A B : Participant

_ ≟ _ : Decidable {A = Participant} _≡_
A ≟ A = yes refl
A ≟ B = no λ ()
B ≟ A = no λ ()
B ≟ B = yes refl

Honest : Σ[ ps ∈ List Participant ] (length ps > 0)
Honest = [ A ] , s≤s z≤n

open BitML Participant _≟_ [A]⁺
```

We then define an advertisement, whose type already says a lot about what is going on; it carries ₿ 5, presumes the existence of at least one deposit of ₿ 200, and requires two deposits of ₿ 200 and ₿ 100.

```
ex-ad : Advertisement 5 [200] [200] [100]
ex-ad = ⟨ B?200 ∧ A ! 100 ⟩
          split ( 2 ⊸ withdraw B
                ⊕ 2 ⊸ after 100 : withdraw A
                ⊕ 1 ⊸ put [200] ⇒ B : withdraw {201} A ⊣ ...
                )
        ⊣ ...
```

Looking at the precondition itself, we see that the required deposits will be provided by B and A, respectively. The contract first splits the bitcoins across three branches: the first one gives ₿ 2 to B, the second one gives ₿ 2 to A after some time period, while the third one retrieves B's deposit of ₿ 200 and allows B to authorize the withdrawal of the remaining funds (currently ₿ 201) from A.

We have omitted the proofs that ascertain the well-formedness of the *put* command and the advertisement, as they are straightforward and do not provide any more intuition[4].

1.2 Small-step Semantics

BitML is a *process calculus*, which is geared specifically towards smart contracts. Contrary to most process calculi that provide primitive operators for inter-process communication via message-passing [Hoare 1978], the BitML calculus does not provide such built-in features.

It, instead, provides domain-specific synchronization mechanisms through its *small-step* reduction semantics. These essentially define a *labelled transition system* between *configurations*, where *action* labels are emitted on every transition and represent the required actions of the participants. This symbolic model consists of two layers; the bottom one transitioning between *untimed* configurations and the top one that works on *timed* configurations.

We start with the datatype of actions, which showcases the principal actions required to satisfy an advertisement's preconditions and an action to pick one branch of a collection of contracts (introduced by the choice operator ⊕). We have omitted uninteresting actions concerning the manipulation of deposits, such as dividing, joining, donating and destroying them. Since we will often need versions of the types of advertisements/contracts with their indices existentially quantified, we first provide aliases for them. For convenience in notation, we will sometimes write $\exists A$ to mean this existential packing of the indices of A:

```
AdvertisedContracts : Set
AdvertisedContracts = List (∃[ v ] ∃[ vsᶜ ] ∃[ vsᵛ ] ∃[ vsᵖ ] Advertisement v vsᶜ vsᵛ vsᵖ)

ActiveContracts : Set
ActiveContracts = List (∃[ v ] ∃[ vs ] List (Contract v vs))

data Action (p : Participant) -- the participant that authorizes this action
```

[4] In fact, we have defined decidable procedures for all such proofs using the *proof-by-reflection* pattern [Van Der Walt and Swierstra 2012]. These automatically discharge all proof obligations, when there are no variables involved.

```
  :  AdvertisedContracts   -- the contract advertisements it requires
  → ActiveContracts        -- the active contracts it requires
  → List Value             -- the deposits it requires from this participant
  → List Deposit           -- the deposits it produces
  → Set where

  -- commit secrets to stipulate an advertisement
  ♯ ▷ _ : (ad : Advertisement v vsᶜ vsᵛ vsᵖ)
        → Action p [v , vsᶜ , vsᵛ , vsᵖ , ad] [] [] []

  -- spend x to stipulate an advertisement
  _ ▷ˢ _ : (ad : Advertisement v vsᶜ vsᵛ vsᵖ)
         → (i : Index vsᵖ)
         → Action p [v , vsᶜ , vsᵛ , vsᵖ , ad] [] [vsᵖ !! i] []

  -- pick a branch
  _ ▷ᵇ _ : (c : List (Contract v vs))
         → (i : Index c)
         → Action p [] [v , vs , c] [] []

  ⋮
```

The action datatype is parametrised[5] over the participant who performs it and includes several indices representing the prerequisites the current configuration has to satisfy, in order for the action to be considered valid (e.g. one cannot spend a deposit to stipulate an advertisement that does not exist).

The first index refers to advertisements that appear in the current configuration, the second to contracts that have already been stipulated, the third to deposits owned by the participant currently performing the action and the fourth declares new deposits that will be created by the action (e.g. dividing a deposit would require a single deposit as the third index and produce two other deposits in its fourth index).

Although our indexing scheme might seem a bit heavyweight now, it makes many little details and assumptions explicit, which would bite us later on when we will need to reason about them.

Continuing from our previous example advertisement, let's see an example action where A spends the required ₿ 100 to stipulate the example contract[6]:

```
ex-spend : Action A [5 , [200] , [200] , [100] , ex-ad] [] [100] []
ex-spend = ex-ad ▷ˢ 0 SF
```

The $0\ SF$ is not a mere natural number, but inhibits $Fin\ (length\ vs^{p})$, which ensures we can only construct actions that spend valid persistent deposits.

[5] In Agda, datatype parameters are similar to indices, but are not allowed to vary across constructors.

[6] Notice that we have to make all indices of the advertisement explicit in the second index in the action's type signature.

Configurations are now built from advertisements, active contracts, deposits, action authorizations and committed/revealed secrets:

```
data Configuration′ :  --        current        ×      required
                          AdvertisedContracts × AdvertisedContracts
                       → ActiveContracts      × ActiveContracts
                       → List Deposit         × List Deposit
                       → Set where

  -- empty configuration
  ∅ : Configuration′ ([] , []) ([] , []) ([] , [])

  -- contract advertisement
  ‘_ : (ad : Advertisement v vsᶜ vsᵛ vsᵖ)
     → Configuration′ ([ v , vsᶜ , vsᵛ , vsᵖ , ad ] , []) ([] , []) ([] , [])

  -- active contract
  ⟨ _ , _ ⟩ᶜ : (c : List (Contract v vs)) → Value
            → Configuration′ ([] , []) ([ v , vs , c ] , []) ([] , [])

  -- deposit redeemable by a participant
  ⟨ _ , _ ⟩ᵈ : (p : Participant) → (v : Value)
             → Configuration′ ([] , []) ([] , []) ([ p has v ] , [])

  -- authorization to perform an action
  _ [ _ ] : (p : Participant) → Action p ads cs vs ds
          → Configuration′ ([] , ads) ([] , cs) (ds , ((p has _) ⟨$⟩ vs))

  -- committed secret
  ⟨ _ : _#_ ⟩ : Participant → Secret → Maybe ℕ
              → Configuration′ ([] , []) ([] , []) ([] , [])

  -- revealed secret
  _ : _#_ : Participant → Secret → ℕ
          → Configuration′ ([] , []) ([] , []) ([] , [])

  -- parallel composition
  _ | _ : Configuration′ (adsˡ , radsˡ) (csˡ , rcsˡ) (dsˡ , rdsˡ)
        → Configuration′ (adsʳ , radsʳ) (csʳ , rcsʳ) (dsʳ , rdsʳ)
        → Configuration′ (adsˡ          ++ adsʳ , radsˡ ++ (radsʳ \ adsˡ))
                         (csˡ           ++ csʳ  , rcsˡ  ++ (rcsʳ  \ csˡ))
                         ((dsˡ \ rdsʳ) ++ dsʳ  , rdsˡ  ++ (rdsʳ  \ dsˡ))
```

The indices are quite involved, since we need to record both the current advertisements, stipulated contracts and deposits and the required ones for the configuration to become valid. The most interesting case is the parallel composition operator, where the resources provided by the left operand might satisfy some requirements of the right operand. Moreover, consumed deposits have

to be eliminated as there can be no double spending, while the number of advertisements and contracts always grows.

By composing configurations together, we will eventually end up in a *closed* configuration, where all required indices are empty (i.e. the configuration is self-contained):

```
Configuration : AdvertisedContracts → ActiveContracts → List Deposit → Set
Configuration ads cs ds = Configuration′ (ads , []) (cs , []) (ds , [])
```

We are now ready to declare the inference rules of the bottom layer of our small-step semantics, by defining an inductive datatype modelling the binary step relation between untimed configurations:

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

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

  C-Advertise : ∀ {Γ ad}
    → ∃[p ∈ participantsᵍ (G ad)] p ∈ Hon
    ──────────────────────────────────────
    → Γ ⟶ ‘ad | Γ

  C-AuthCommit : ∀ {A ad Γ}
    → secrets (G ad) ≡ a₀ ... aₙ
    → (A ∈ Hon → ∀ [i ∈ 0 ... n] aᵢ ≢ ⊥)
    ──────────────────────────────────────
    → ‘ad | Γ ⟶ ‘ad | Γ | ... ⟨ A : aᵢ#Nᵢ ⟩ ... | A [♯▷ ad]

  C-Control : ∀ {Γ C i D}
    → C !! i ≡ A₁ : A₂ : ... : A : D
    ──────────────────────────────────
    → ⟨ C , v ⟩ᶜ | ... Aᵢ [C ▷ᵇ i] ... | Γ ⟶ ⟨ D , v ⟩ᶜ | Γ
  ⋮
```

There is a total of 18 rules we need to define, but we choose to depict only a representative subset of them. The first pair of rules initially appends the authorization to merge two deposits to the current configuration (rule *DEP-AuthJoin*) and then performs the actual join (rule [*DEP-Join*]). This

is a common pattern across all rules, where we first collect authorizations for an action by all involved participants, and then we fire a subsequent rule to perform this action. [*C-Advertise*] advertises a new contract, mandating that at least one of the participants involved in the pre-condition is honest and requiring that all deposits needed for stipulation are available in the surrounding context. [*C-AuthCommit*] allows participants to commit to the secrets required by the contract's pre-condition, but only dishonest ones can commit to the invalid length ⊥. Lastly, [*C-Control*] allows participants to give their authorization required by a particular branch out of the current choices present in the contract, discarding any time constraints along the way.

It is noteworthy to mention that during the transcriptions of the complete set of rules from the paper [Bartoletti and Zunino 2018] to our dependently-typed setting, we discovered some discrepancies or over-complications, which we document extensively in Section 1.6.

The inference rules above have elided any treatment of timely constraints; this is handled by the top layer, whose states are now timed configurations. The only interesting inference rule is the one that handles time decorations of the form *after* _ : _, since all other cases are dispatched to the bottom layer (which just ignores timely aspects).

```
record Configurationᵗ (ads : AdvertisedContracts)
                      (cs : ActiveContracts)
                      (ds : Deposits) : Set where
  constructor _ @ _
  field cfg  : Configuration ads cs ds
        time : Time

data _ ⟶ₜ _ : Configurationᵗ ads cs ds → Configurationᵗ ads′ cs′ ds′ → Set where
  Action : ∀ {Γ Γ′ t}
    → Γ ⟶ Γ′
    ──────────────
    → Γ @ t ⟶ₜ Γ′ @ t
  Delay : ∀ {Γ t δ}
    ──────────────────────
    → Γ @ t ⟶ₜ Γ @ (t + δ)
  Timeout : ∀ {Γ Γ′ t i contract}
    → All (_ ⩽ t) (timeDecorations (contract !! i))  -- all time constraints are satisfied
    → ⟨ [ contract !! i ] , v ⟩ᶜ | Γ ⟶ Γ′           -- resulting state if we pick this branch
    ──────────────────────────────
    → (⟨ contract , v ⟩ᶜ | Γ) @ t ⟶ₜ Γ′ @ t
```

1.3 Reasoning Modulo Permutation

In the definitions above, we have assumed that $(\_ \mid \_ , \varnothing)$ forms a *commutative monoid*, which allowed us to always present the required sub-configuration individually on the far left of a composite configuration. While such definitions enjoy a striking similarity to the ones appearing in the original paper [Bartoletti and Zunino 2018] (and should always be preferred in an informal textual setting), this approach does not suffice for a mechanized account of the model. After all, this explicit treatment of all intuitive assumptions/details is what makes our approach robust and will lead to a deeper understanding of how these systems behave. To overcome this intricacy, we introduce an *equivalence relation* on configurations, which holds when they are just permutations of one another:

```
_≈_ : Configuration ads cs ds → Configuration ads cs ds → Set
c ≈ c′ = cfgToList c ↭ cfgToList c′
  where
    open import Data.List.Permutation using (_↭_)

    cfgToList : Configuration′ p₁ p₂ p₃ → List (∃[p₁] ∃[p₂] ∃[p₃] Configuration′ p₁ p₂ p₃)
    cfgToList ∅                 = []
    cfgToList (l ∣ r)           = cfgToList l ++ cfgToList r
    cfgToList {p₁} {p₂} {p₃} c = [ p₁ , p₂ , p₃ , c ]
```

Given this reordering mechanism, we now need to generalize all our inference rules to implicitly reorder the current and next configuration of the step relation. We achieve this by introducing a new variable for each of the operands of the resulting step relations, replacing the operands with these variables and requiring that they are re-orderings of the previous configurations, as shown in the following generalization of the *DEP-AuthJoin* rule[7]:

```
DEP-AuthJoin :
      Γ′ ≈ ⟨ A , v ⟩ᵈ ∣ ⟨ A , v′ ⟩ᵈ ∣ Γ              ∈ Configuration ads cs (A has v :: A has v′ :: ds)
    → Γ″ ≈ ⟨ A , v ⟩ᵈ ∣ ⟨ A , v′ ⟩ᵈ ∣ A [ 0 ↔ 1 ] ∣ Γ  ∈ Configuration ads cs (A has (v + v′) :: ds)
    ──────────────────────────────────────────────────
    → Γ′ ⟶ Γ″
```

Unfortunately, we now have more proof obligations of the re-ordering relation lying around, which makes reasoning about our semantics rather tedious. We are currently investigating different techniques to model such reasoning up to equivalence:

- *Quotient types* [Altenkirch et al. 2011] allow equipping a type with an equivalence relation. If we assume the axiom that two elements of the underlying type are *propositionally* equal when they are equivalent, we could discharge our current proof burden trivially by reflexivity. Unfortunately, while one can easily define *setoids* in Agda, there is not enough support

[7] In fact, it is not necessary to reorder both ends for the step relation; at least one would be adequate.

from the underlying type system to make reasoning about such an equivalence as easy as with built-in equality.

- Going a step further into more advanced notions of equality, we arrive at *homotopy type theory* [hom 2013], which tries to bridge the gap between reasoning about isomorphic objects in informal pen-paper proofs and the way we achieve this in mechanized formal methods. Again, realizing practical systems with such an enriched theory is a topic of current research [Cohen et al. 2016] and no mature implementation exists yet, so we cannot integrate it with our current development in any pragmatic way.
- The crucial problems we have encountered so far are attributed to the non-deterministic nature of BitML, which is actually inherent in any process calculus. Building upon this idea, we plan to take a step back and investigate different reasoning techniques for a minimal process calculus. Once we have an approach that is more suitable, we will incorporate it in our full-blown BitML calculus. Current efforts are available on Github [8].

For the time being, the complexity that arises from having the permutation proofs in the premises of 20 rules, is intractable. As a quick workaround, we can factor out the permutation relation in the *reflexive transitive closure* of the step relation, which will eventually constitute our equational reasoning device:

```
data _ ↠ _ : Configuration ads cs ds → Configuration ads′ cs′ ds′ → Set where
  _□ : (M : Configuration ads cs ds) → M ↠ M
  _ ⟶⟨ _ ⟩_ : ∀ {M N} (L : Configuration ads cs ds)
    → L′ ⟶ M′
    → M ↠ N
    → {_ : L ≈ L′ × M ≈ M′
      ──────────────────}
    → L ↠ N
begin_ : ∀ {M N} → M ↠ N → M ↠ N
```

The permutation relation is actually decidable, so we can always discharge the implicitly required proof, similarly to the techniques described in Section ??.

1.4 Example: Timed-commitment Protocol

We are finally ready to see a more intuitive example of the *timed-commitment protocol*, where a participant commits to revealing a valid secret a (e.g. "qwerty") to another participant, but loses her deposit of ₿ 1 if she does not meet a certain deadline t:

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

[8] https://github.com/omelkonian/formal-process-calculus

Below is one possible reduction in the bottom layer of our small-step semantics, demonstrating the case where the participant actually meets the deadline:

```
tc-semantics : ⟨ A , 1 ⟩ᵈ ↠ ⟨ A , 1 ⟩ᵈ | A : a # 6
tc-semantics =
  begin
    ⟨ A , 1 ⟩ᵈ
  ⟶⟨ C-Advertise ... ⟩
    ‘tc | ⟨ A , 1 ⟩ᵈ
  ⟶⟨ C-AuthCommit ... ⟩
    ‘tc | ⟨ A , 1 ⟩ᵈ | ⟨ A : a # 6 ⟩ | A [ ♯ ▷ tc ]
  ⟶⟨ C-AuthInit ... ⟩
    ‘tc | ⟨ A , 1 ⟩ᵈ | ⟨ A : a # 6 ⟩ | A [ ♯ ▷ tc ] | A [ tc ▷ˢ 0 ]
  ⟶⟨ C-Init ... ⟩
    ⟨ tc , 1 ⟩ᶜ | ⟨ A : a # inj₁ 6 ⟩
  ⟶⟨ C-AuthRev ... ⟩
    ⟨ tc , 1 ⟩ᶜ | A : a # 6
  ⟶⟨ C-Control ... ⟩
    ⟨ [ reveal [ a ] ⇒ withdraw A ⊣ ... ] , 1 ⟩ᶜ | A : a # 6
  ⟶⟨ C-PutRev ... ⟩
    ⟨ [ withdraw A ] , 1 ⟩ᶜ | A : a # 6
  ⟶⟨ C-Withdraw ... ⟩
    ⟨ A , 1 ⟩ᵈ | A : a # 6
  □
```

At first, A holds a deposit of ₿ 1, as required by the contract's precondition. Then, the contract is advertised and the participants slowly provide the corresponding prerequisites (i.e. A commits to a secret via *C-AuthCommit* and spends the required deposit via *C-AuthInit*, while B does not do anything). After all pre-conditions have been satisfied, the contract is stipulated (rule *C-Init*) and the secret is successfully revealed (rule *C-AuthRev*). Finally, the first branch is picked (rule *C-Control*) and A retrieves her deposit back (rules *C-PutRev* and *C-Withdraw*).

We chose to omit the proofs required at the application of each inference rules (replaced with ... above), since these are tedious and mostly uninteresting. Moreover, we plan to develop decision procedures for these proofs[9] to automate this part of the proof development process.

1.5 Symbolic Model

In order to formalize the BitML's symbolic model, we first notice that a constructed derivation witnesses one of many possible contract executions. In other words, derivations of our small-step semantics model *traces* of the contract execution. Our symbolic model will provide a game-theoretic view over those traces, where each participant has a certain *strategy* that selects moves depending

[9] Most proofs of decidability are in the Agda standard library already, but there is still a lot of "plumbing" to be done.

on the current trace of previous moves. Moves here should be understood just as emissions of a label, i.e. application of a certain inference rule.

1.5.1 Labelled Step Relation

To that end, we associate a label to each inference rule and extend the original step relation to additionally emit labels, hence defining a *labelled transition system*.

We first define the set of labels, which basically distinguish which rule was used, along with all (non-proof) arguments that are required by the rule:

```
data Label : Set where

  auth-join [ _ , _ ↔ _ ] : Participant → DepositIndex → DepositIndex → Label
  join [ _ ↔ _ ] :                        DepositIndex → DepositIndex → Label

  advertise [ _ ] : ∃Advertisement → Label

  auth-commit [ _ , _ , _ ] : Participant → ∃Advertisement → List CommittedSecret → Label
  auth-init [ _ , _ , _ ] : Participant → ∃Advertisement → DepositIndex → Label
  init [ _ ] : ∃Advertisement → Label

  auth-control [ _ , _ ▷ᵇ _ ] : Participant → (c : ∃Contracts) → Index (proj₂ (proj₂ c)) → Label
  control : Label

  ⋮

  delay [ _ ] : Time → Label
```

Notice how we existentially pack indexed types, so that *Label* remains simply-typed. This is essential, as it would be tedious to manipulate indices when there is no need for them. Moreover, some indices are now just $\mathbb{N}$ instead of *Fin*, losing the guarantee to not fall out-of-bounds.

The step relation will now emit the corresponding label for each rule. Below, we give the updated kind signature and an example for the *DEP-AuthJoin* rule:

```
data _ ⟶⟦ _ ⟧ _ : Configuration ads cs ds
                → Label
                → Configuration ads′ cs′ ds′
                → Set where

  ⋮

  DEP-AuthJoin :
    ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | Γ
    ⟶⟦ auth-join [ A , 0 ↔ 1 ] ⟧
```

```
    ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | A [0 ↔ 1] | Γ
  ⋮
```

Naturally, the reflexive transitive closure of the augmented step relation will now hold a sequence of labels as well:

```
data _ →⟦ _ ⟧ _ : Configuration ads cs ds
                → List Label
                → Configuration ads′ cs′ ds′
                → Set where

  _□ : (M : Configuration ads cs ds)
       ──────────────────────────────
     → M →⟦ [] ⟧ M

  _ ⟶⟨ _ ⟩_⊢_ : (L : Configuration ads cs ds) {L′ : Configuration ads cs ds}
                 {M M′ : Configuration ads′ cs′ ds′ } {N : Configuration ads″ cs″ ds″ }
     → L′ ⟶⟦ a ⟧ M′
     → (L ≈ L′ ) × (M ≈ M′ )
     → M →⟦ as ⟧ N
       ──────────────────────────────────────────────
     → L →⟦ a :: as ⟧ N

start : {M : Configuration ads cs ds} {N : Configuration ads′ cs′ ds′ }
  → M →⟦ as ⟧ N
    ──────────
  → M →⟦ as ⟧ N
start M →↠ N = M →↠ N
```

The timed variants of the step relation follow exactly the same procedure, so we do not repeat the definitions here.

1.5.2 Traces

Values of type `_ →⟦ _ ⟧ _` model execution traces. Since the complex type indices of the step-relation datatype is not as useful here, we define a simpler datatype of execution traces that is a list of labelled transitions between (existentially-packed) timed configurations:

```
data Trace : Set where
  _∙          : ∃TimedConfiguration → Trace
```

```
_ ::⟦ _ ⟧ _ : ∃TimedConfiguration → Label → Trace → Trace
```

Stripping. Strategies will make moves based on these traces, so we need a *stripping* operation that traverses a configuration with its emitted labels and removes any sensitive information (i.e. committed secrets):

```
stripCfg : Configuration′ p₁ p₂ p₃ → Configuration′ p₁ p₂ p₃
stripCfg⟨ p : a # _ ⟩ = ⟨ p : a # nothing ⟩
stripCfg (l | r ⊣ p) = stripCfg l | stripCfg r ⊣ p
stripCfg c           = c

stripLabel : Label → Label
stripLabel auth-commit [p , ad , _] = auth-commit [p , ad , []]
stripLabel a = a

_ ∗ :Trace → Trace
(... , Γ @ t) ∗          = (... , stripCfg Γ @ t)
(... , Γ @ t) ::⟦ α ⟧ ts = (... , stripCfg Γ @ t) ::⟦ stripLabel α ⟧ (ts ∗)
```

1.5.3 Strategies

Participant strategies are functions which, given the (stripped) trace so far, pick a set of possible next moves for its participant. These moves cannot be arbitrary; they have to satisfy several validity conditions which we require as proof in the datatype definition itself.

Strategies are expected to be PPTIME algorithms, so as to have a certain computational bound on the processing they can undergo to compute secrets, etc. Since working on a resource-aware logic would make this much more difficult in search of tooling and infrastructure, we ignore this requirement and simply model strategies as regular functions.

Before we define the types of strategies, we give a convenient notation to extend a trace with another (timed) transition:

```
_ ↣⟦ _ ⟧ _ : Trace → Label → ∃TimedConfiguration → Set
R ↣⟦ α ⟧ (_ , _ , _ , tc′ )
  = proj₂ (proj₂ (proj₂ (lastCfg R))) ⟶⟦ α ⟧ tc′
```

Honest strategies. Each honest participant is modelled by a symbolic strategy that outputs a set of possible next moves with respect to the current trace. These moves have to be *valid*, thus we define *honest strategies* as a dependent record:

```
record HonestStrategy (A : Participant) : Set where
  field
```

```
strategy : Trace → List Label

valid    : A ∈ Hon                                                    (1)
         × (∀ R α → α ∈ strategy (R ∗) →                               (2)
             ∃[ R′ ] (R ↣⟦ α ⟧ R′ ))
         × (∀ R α → α ∈ strategy (R ∗) →                               (3)
             All (_ ≡ A) (authDecoration α))
         × (∀ R Δ Δ′ ad →                                             (4)
             auth-commit [ A , ad , Δ ] ∈ strategy (R ∗) →
             auth-commit [ A , ad , Δ′ ] ∈ strategy (R ∗) →
               Δ ≡ Δ′ )
         × (∀ R T′ α → α ∈ strategy (R ∗) →                           (5)
             ∃[ α′ ] (R ↣⟦ α′ ⟧ T′ ) →
             ∃[ R″ ] (T′ ::⟦ α ⟧ R ↣⟦ α ⟧ R″) →
               α ∈ strategy ((T′ ::⟦ α ⟧ R) ∗))
```

Condition (1) restricts our participants to the honest subset[10] and condition (2) requires that chosen moves are in accordance to the small-step semantics of BitML. Condition (3) states that one cannot authorize moves for other participants, condition (4) requires that the lengths of committed secrets are *coherent* (i.e. no different lengths for the same secrets across moves) and condition (5) dictates that decisions are *consistent*, so as moves that are not chosen will still be selected by the strategy in a future run (if they are still valid).

All honest participants should be accompanied by such a strategy, so we pack all honest strategies in one single datatype:

```
HonestStrategies : Set
HonestStrategies = ∀ {A} → A ∈ Hon → ParticipantStrategy A
```

Adversary strategies. All dishonest participant will be modelled by a single adversary *Adv*, whose strategy now additionally takes the moves chosen by the honest participants and makes the final decision.

Naturally, the chosen move is subject to certain conditions and is again a dependent record:

```
record AdversarialStrategy (Adv : Participant) : Set where
  field
    strategy : Trace → List (Participant × List Label) → Label

    valid    : Adv ∉ Hon                                              (1)
             × (∀ {B ad Δ} → B ∉ Hon → α ≡ auth-commit [ B , ad , Δ ] →  (2)
                 α ≡ strategy (R ∗) [ ])
             × ∀ {R : Trace} { moves : List (Participant × List Label) } →  (3)
```

[10] Recall that *Hon* is non-empty, i.e. there is always at least one honest participant.

```
let α = strategy (R ∗) moves in
(   ∃[A]
       ( A ∈ Hon
       × authDecoration α ≡ just A
       × α ∈ concatMap proj₂ moves)
 ⊎ (   authDecoration α ≡ nothing
    × (∀ δ → α ≢ delay [δ])
    × ∃[R′ ] (R ⤖⟦ α ⟧ R′ ))
 ⊎ (∃[B]
       (   (authDecoration α ≡ just B)
        × (B ∉ Hon)
        × (∀ s → α ≢ auth-rev [B , s])
        × ∃[R′ ] (R ⤖⟦ α ⟧ R′ )))
 ⊎ ∃[δ]
       ( (α ≡ delay [δ])
       × All (λ {(_ , ) → ( ≡ [])
                              ⊎ Any (λ {delay [δ′ ] → δ′ ≥δ; _ → ⊥}) }) moves)
 ⊎ ∃[B]∃[s]
       ( α ≡ auth-rev [B , s]
       × B ∉ Hon
       × ⟨ B : s#nothing ⟩ ∈ (R ∗)
       × ∃[R∗′]∃[Δ]∃[ad]
             ( R∗′ ∈ prefixTraces (R ∗)
             × strategy R∗′[] ≡ auth-commit [B , ad , Δ]
             × (s , nothing) ∈ Δ)))
```

The first two conditions state that the adversary is not one of the honest participants and that committing cannot depend on the honest moves, respectively. Condition (3) constraints the move that is chosen by the adversary, such that one of the following conditions hold:

(1) The move was chosen out of the available honest moves.
(2) It is not a *delay*, nor does it require any authorization.
(3) It is authorized by a dishonest participant, but is not a secret-revealing move.
(4) It is a *delay*, but one that does not influence the time constraints of the honest participants.
(5) It reveals a secret from a dishonest participant, in which case there is valid commit (i.e. with non-$\bot$ length) somewhere in the previous trace.

A complete set of strategies includes a strategy for each honest participant and a single adversarial strategy:

```
Strategies : Set
Strategies = AdversarialStrategy × HonestStrategies
```

We can now describe how to proceed execution on the current trace, namely by retrieving possible moves from all honest participants and giving control to the adversary to make the final choice for a label:

```
runAdversary : Strategies → Trace → Label
runAdversary (S† , S) R = strategy S†(R ∗) (runHonestAll (R ∗) S)
  where
    runHonestAll : Trace → List (Participant × List Label) → HonestMoves
    runHonestAll R S = mapWith ∈ Hon (λ {A} A ∈→ A , strategy (S A ∈) (R ∗))
```

Symbolic Conformance. Given a trace, we can formulate a notion of *conformance* of a trace with respect to a set of strategies, namely when we transitioned from an initial configuration to the current trace using only moves obtained by those strategies:

```
data _-conforms-to-_ : Trace → Strategies → Set where

  base : ∀ {Γ : Configuration ads cs ds} {SS : Strategies}
    → Initial Γ
      ──────────────────────────────────
    → (ads , cs , ds , Γ @ 0)· -conforms-to- SS

  step : ∀ {R : Trace} {T′ : ∃TimedConfiguration} {SS : Strategies}
    → R -conforms-to-SS
    → R ↣⟦ runAdversary SS R ⟧ T′
      ──────────────────────────────
    → (T′ ::⟦ runAdversary SS R ⟧ R) -conforms-to- SS
```

1.5.4 Meta-theoretical results

To increase confidence in our symbolic model, we proceed with the mechanization of two meta-theoretical lemmas.

Stripping preserves semantics. The first one concerns the operation of stripping sensitive values out of a trace. If we exclude moves that reveal or commit secrets (i.e. rules *AuthRefv* and *AuthCommit*), we can formally prove that stripping preserves the small-step semantics:

```
∗ − preserves-semantics :
  (∀ A s      → α ≢ auth-rev [A , s]) →
  (∀ A ad Δ → α ≢ auth-commit [A , ad , Δ]) →
  (∀ T′   → R ↣⟦ α ⟧ T′
```

$$\rightarrow R * \rightarrowtail[\![\alpha]\!] T' *)$$

$$\times (\forall T' \rightarrow R * \rightarrowtail[\![\alpha]\!] T'$$

$$\rightarrow \exists[T''] (R \rightarrowtail[\![\alpha]\!] T'') \times (T' * \equiv T'' *)$$

The second part of the conclusion states that if we have a transition from a stripped state, then there is an equivalent target state (modulo additional sensitive information) to which the un-stripped state can transition.

Adversarial moves are always semantic. Lastly, it holds that all moves that can be chosen by the adversary are admitted by the small-step semantics:

$$\begin{aligned}&\mathit{adversarial\text{-}move\text{-}is\text{-}semantic} :\\&\quad \exists[T'] (R \rightarrowtail[\![\mathit{runAdversary}\ (S\dagger, S)\ R]\!] T')\end{aligned}$$

1.6 BitML Paper Fixes

It is expected in any mechanization of a substantial amount of theoretical work to encounter inconsistencies in the pen-and-paper version, ranging from simple typos and omissions to fundamental design problems. This is certainly one of the primary selling points for formal verification; corner cases that are difficult to find by testing or similar methods, can instead be discovered with rigorous formal methods.

Our formal development was no exception, since we encountered several issues with the original presentation, which led to the modifications presented below.

Inference Rules. Rule *DEP-Join* requires two symmetric invocations of the *DEP-AuthJoin* rule, but it is unclear if this gives us anything meaningful. Instead, we choose to simplify the rule by requiring just one authorization.

When rule *C-AuthRev* is presented in the original BitML paper, it seems to act on an atomic configuration $\langle A : \alpha \# \mathbb{N} \rangle$. This renders the rule useless in any practical scenario, so we extend the rule to include a surrounding context:

$$\langle A : s \# \mathit{just}\ n \rangle \,|\,\Gamma \longrightarrow[\![\mathit{auth\text{-}rev}\ [A, s]]\!]\ A : s \# n \,|\,\Gamma$$

Small-step Derivations as Equational Reasoning. In Section **??** , we saw an example derivation of our small-step semantics, given in an equational-reasoning style. This is possible, because the involved rules follow a certain format.

Alas, rule *C-Control* includes another transition in its premises which results in the same state Γ' as the transition in the conclusion, resulting in a tree-like proof structure. which is arguably inconvenient for textual presentation. This is problematic when we try to reason in an equational-reasoning style using our multi-step relation $\_ \twoheadrightarrow \_$, since this branching will break our sequential way of presenting the proof step by step.

To avoid this issue, notice how we can "linearize" the proof structure by removing the premise and replacing the target configuration of the conclusion with the source configuration of the removed premise. Our version of *C-Control* reflects this important refactoring.

Conditions for Adversarial Strategies. Moves chosen by an adversarial strategy come in two forms: labels and pairs (A, j) of an honest participant A with an index into his/her current moves. However, this is unnecessary, since we can both cases uniformly using our *Label* type.

Semantics-preserving Stripping. The meta-theoretical lemma concerning stripping in the original paper (*Lemma 3*) requires that the transition considered is not an application of the *Auth-Rev* rule. It turns out this is not a strong enough guarantee, since the *AuthCommit* rule also contains sensitive information, thus would not be preserved after stripping. We, therefore, fix the statement in Lemma 3 to additionally require that $\alpha \not\equiv A : \langle\, G \,\rangle C, \Delta$.

References

2013. Homotopy Type Theory: Univalent Foundations of Mathematics. *CoRR* abs/1308.0729 (2013). arXiv:1308.0729 http://arxiv.org/abs/1308.0729

Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. 2011. Definable quotients in type theory. *Draft paper* (2011), 48–49.

Massimo Bartoletti and Roberto Zunino. 2018. *BitML: a calculus for Bitcoin smart contracts.* Technical Report. Cryptology ePrint Archive, Report 2018/122.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical Type Theory: a constructive interpretation of the univalence axiom. *CoRR* abs/1611.02108 (2016). arXiv:1611.02108 http://arxiv.org/abs/1611.02108

Charles Antony Richard Hoare. 1978. Communicating sequential processes. In *The origin of concurrent programming*. Springer, 413–443.

Paul Van Der Walt and Wouter Swierstra. 2012. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*. Springer, 157–173.