Smart Contracts as Authorized Production Rules

Ben Lippmeier
UNSW (Australia)
benl@ouroborus.net

Amos Robinson
UNSW (Australia)
amos.robinson@unsw.edu.au

Andrae Muys
Digital Asset
andrae.muys@digitalasset.com

ABSTRACT

Rainfall is a smart contract programming model that allows mutually distrusting parties to manage assets on a distributed ledger. The model consists of a tuple space of authorized facts, and a set of production rules. Rules match on authorized facts, gaining their authority, and produce new facts with a subset of the gained authority. Rainfall allows assets such as crypto currencies to be defined in user code, rather than being baked directly into the ledger framework. Our authorization model also provides a natural privacy model, where not all rules or facts need to be revealed to all parties.

ACM Reference Format:
Ben Lippmeier, Amos Robinson, and Andrae Muys. 2019. Smart Contracts as Authorized Production Rules. In *Proceedings of Principles and Practice of Declarative Programming (PPDP).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Distributed ledgers allow information about financial assets to be recorded and modified by mutually distrusting parties. A prime application is to manage cryptocurrencies such as Bitcoin, Dogecoin, Ethereum and so on. In such systems, some of the rules that specify how assets can be transferred between parties are baked into the ledger framework, while others are defined in a programming language whose runtime is part of the ledger system. Early ledgers such as Bitcoin used a sequence of simple, non-looping bytecodes to specify the rules of coin transfer [11]. Latter systems use more general purpose languages such as the Ethereum Virtual Machine (EVM) [53], EOS [30], Scilla [49], Pact [47], FCL [3], Plutus [34] and DAML [2], which can include looping control flow, structured data and polymorphism. We refer to these as *smart contract* languages, as the intended application is to express the rules, workflows, rights and obligations involved in managing assets [31].

Most smart contract languages are expressive enough that they are used to define *tokens*, which are new currencies separate from the native currency of the system (such as Bitcoin, Dogecoin and so on). The rules and data to define a token is typically installed in "user space" on the ledger, rather using builtin support for the native currency. Awkwardly, although tokens can have similar features to the native currency, when they are defined separately they need special handling by user facing tool chains — to provide wallet interfaces, transaction listings and so on [52].

PPDP, October 7 – 9, 2019, Porto, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Besides user experience, there is often an expressivity gap between the support a distributed ledger system provides for its native currencies, and what can be implemented for tokens. For example, in account based [55] Ethereum, the user code that defines a token typically stores the balances for all token accounts in an array structure that is owned by a single native Ethereum account. As the token balances are stored in a user level structure, it is easy for a contract to perform queries over this data, such as to find the account with the highest balance. On the other hand, Ethereum does not provide a way for contract code to perform the same queries over account balances of its own native currency, as this is not supported by the virtual machine the contract runs on. The Unspent Transaction Output (UTxO) based [54] DAML is more homogeneous in that assets are defined uniformly as user code. However, the current version of DAML, somewhat inversely, does not support general queries over user data, which we discuss further in §5.4.

FCL provides an alternate programming model based on Colored Petri Nets (CPNs) [35]. Petri nets are related to *production rule* frameworks such as OPS5 [23] and CLIPS [48]. A production rule watches a store of facts, waits for a particular set of matching facts to become available, and then produces new facts, possibly triggering other rules, and so on. This matching process is fundamentally a query process, like a relational join. In comparison with the Petri net model, we can view a *place* in the Petri net as an unordered *table* in the database model. Petri nets and production rules are expressive enough to implement real business workflows, with prior work demonstrating how to use Petri nets as a compilation target for the standard Business Process Execution Language (BPEL) [38]. However, although FCL is based on Petri nets, its native asset system is baked into the language framework, rather than being user defined. Our contribution is to unify these concerns:

- We present Rainfall, a programming model based on authorized production rules that allows mutually distrusting parties to manage assets on a distributed ledger. (§2)
- Rules defining how assets can be transferred are specified directly in user code, instead of requiring the programming model to provide primitive support for asset transfers. (§2.4)
- Our production rule framework makes it straightforward to define workflows that perform queries over all facts visible to some party. (§3.6)
- Our authorization system naturally extends to provide *privacy*, so facts and rules do not need to be revealed to all parties using the system. (§3)

Our work focuses on the language semantics and authority mechanism, rather than details of the networking layer. However, we do mention cross cutting concerns such as the intended representation of transactions. In our case, "contracts" are executable programs, rather than constraints on the values produced by a function [22], messages exchanged in a protocol [1, 19], or a way to compute prices of financial products [46], as in similarly named work.

```
fact  Coin   [issuer: Party,  holder:   Party]
fact  Offer  [id:     Symbol, terms:    Text,
              giver:  Party,  receiver: Party]
fact  Accept [id:     Symbol, accepter: Party]

rule  transfer
await Offer  [id = ?i, giver = ?g, receiver = ?r] gain {g}
  and Accept [id = i,  accepter = r]              gain {r}
  and Coin   [issuer = !Isabelle,  holder = g]
      gain {!Isabelle,g}
to
  say Coin   [issuer = !Isabelle,  holder = r]
   by {!Isabelle,r}  use {'transfer}
```

Figure 1: Coin Transfer Workflow

2 FACTS, RULES, AND AUTHORITY

The Rainfall programming model uses a ledger of facts and a set of rules. Parties using the system add facts to the ledger, cryptographically signing them to demonstrate that they authorize their contents. Rules match on subsets of facts and create new facts, possibly consuming matched facts in the process. Rules can gain authority from matched facts, and the newly created facts can then be given a subset of the authority gained from the matched facts. The set of facts visible to each party is controlled by the authority system, so not all facts need to be visible to all parties. In this section we describe facts, rules and the authority system, finishing with the formal definition of the data model.

2.1 Facts

Figure 1 shows the fact and rule definitions for a simple coin transfer workflow. A `fact` declaration gives the *tag* and *payload* types of each sort of fact used in the workflow. In this example, a `Coin` fact represents a virtual coin that has been created by an `issuer` party, and is currently held by a `holder`. An `Offer` fact indicates an offer by the coin holder, the `giver`, to transfer their coin to a `receiver`. The offer includes an `id` value of the abstract `Symbol` type that uniquely identifies the offer, and a text string describing the terms of the offer. An `Accept` fact indicates that the receiver does indeed wish to accept the coin offer with the given terms. For example, suppose we have the following facts:

```
Coin   [issuer = !Isabelle, holder = !Alice]
Offer  [id     = '1234,  terms    = "To purchase a guitar",
        giver  = !Alice, receiver = !Bob]
```

Names prefixed by `!` are literal party identifiers which have type `Party`. Names prefixed by `'` are symbolic identifiers (strings) which have type `Symbol`. The facts reveal that `Alice` wishes to transfer her coin to `Bob` for the purchase of a guitar. If `Bob` wishes to accept the offer he can add the following fact:

```
Accept [id = '1234, accepter = !Bob]
```

Given the `Offer`, `Accept` and `Coin` facts, the `transfer` rule from Figure 1 can then fire, which *consumes* the three input facts and produces a new one:

```
Coin   [issuer = !Isabelle, holder = !Bob]
```

This new coin belongs to Bob, alternatively, we could say that the coin Alice once had has been transferred to Bob. We will properly introduce Isabelle in §2.4.

2.2 Weights

Suppose Bob already had a coin, and then receives another one. We manage this by giving each fact a *weight* that specifies a positive integral number of "copies" of the fact. We indicate the weight of a fact using the `num` keyword, typically eliding it if the weight is one. For example, suppose the ledger already contained:

```
Coin [issuer = !Isabelle, holder = !Bob]  num 5
```

If Alice transfers an additional coin to Bob then the entry on the ledger would become:

```
Coin [issuer = !Isabelle, holder = !Bob]  num 6
```

Rules can also consume an arbitrary weight of a fact, including zero, which non-destructively reads it, which we discuss in §3.6.

2.3 Rules

The `transfer` rule of Figure 1 specifies how existing facts can be combined to create new facts. The rule is written with syntax based on production rule languages such as OPS5 [23] and CLIPS [48]. A rule definition has the form (`rule` *name* `await` *patterns* `to` *body*) where *patterns* specifies which facts must be available for the rule to fire, and *body* is a pure term expression that constructs a new set of facts to add to the ledger. Each pattern can include a `gain` clause that first checks the matched fact is authorized by a set of parties, and causes the rule to gain that same authorization. The new facts created can be authorized by a subset of the gained authorization.

Names prefixed by `?` are binding occurrences of variables, so the `transfer` rule waits for an `Offer` fact with its `id` field set to some value, binds it to `i`, and must then wait for an `Accept` fact whose `id` field is set to the same value. The runtime intuition is that matching of facts proceeds in sequence, so the rule will wait for an `Offer` fact, then a `Accept` fact, then a `Coin` fact. Variables bound in earlier patterns are in scope in latter ones, and also in the body. Implementations of traditional production rule engines based on the RETE [24] algorithm do not require facts to be matched in order, but we fix the sequence here to simplify our operational semantics.

By default, the facts that a rule matches on are all consumed, and any new facts produced by the same rule are added to the ledger in an atomic transaction. We refer to the process of consuming a fact as *spending* that fact, after the original UTxO [54] model of the Bitcoin system.

2.4 Authority

A given currency can only retain value when it is *scarce*. Fiat currencies like Icelandic Króna (ISK) are scarce because a central authority issues a limited number per year. Bitcoins are scarce because they represent the solution of a particular cryptographic problem, which at a particular time, required significant energy to solve. In our coin transfer example we prevent Alice and Bob from creating an arbitrary number of their own coins, by requiring that they are also authorized by an issuing party Isabelle. We assume that all parties using the system trust Isabelle to not add new, signed `Coin` facts to the ledger in an inappropriate way.[1] For a private financial system Isabelle might represent a bank. For a public ledger system, an initial fixed supply of coins might be generated using a secure multiparty protocol to sign the facts, similarly to how the ZCash system [14, 33] was initialized.

[1] meaning Isabelle will not print money, or at least, not too much.

Given a specific fact, the set of parties that fact is authorized by is called the *by-authority* set. A single party can add any fact to the ledger at any time, provided the fact is only authorized by that single party. Likewise, a single party can consume (spend) a fact from the ledger at any time, provided the fact is authorized by that party alone. In our transfer example, when coin facts are created we assume they are authorized by *both* the issuer and the initial holder of the coins. Ensuring the coins are authorized by both parties means that neither can unilaterally create, transfer or consume them. Jointly authorized facts can only be modified by pre-agreed rules that first collect the authority of all relevant parties. Our by-authority sets are similar to the *contract signatories* of DAML [2].

The by-authority set is part of the fact data, so a rule can read the set of parties that have authorized a fact just as they would read the payload data. In a concrete implementation, when a new fact is created directly by a single party it would be cryptographically signed by that party, and its by-authority set would contain just that party. When a new fact is created by rule execution then the transaction log of the system, which lists the rule name along with the facts spent and created by that rule, records that the authorization of the new fact is as intended. We will discuss transactions further in §3.1.

2.5 Observation

When a fact has been authorized by a particular party, then naturally that party should see the details of that fact. Facts that are authorized by a party are visible to that party. For example, the original `Coin` fact from §2.1 is authorized by both Isabelle and Alice, so those parties will be informed of the creation and subsequent consumption of that fact. However, the `Offer` fact that Alice creates is authorized by Alice alone, so we need an additional mechanism to reveal it to Bob. We include an additional set, the *obs-authority* set, in each fact, which lists the extra parties that are permitted to observe a fact but have not authorized its creation. The same party name may be present in both the by-authority set, as well as the obs-authority set, though this provides no additional benefit.

2.6 Usable Rules

Rainfall is an open system in the sense that new parties can join at any time, and add new facts and rules as they see fit. As mentioned in §2.4, any single party can unilaterally create and consume any fact in the system, provided the fact is authorized by that party alone. If we also allow parties to add any *rule* they like, then we must ensure that these rules do not consume facts, or gain authority, in a way that other parties using the system did not intend. We achieve this by including a final set, the *use-set*, in each fact, which lists the rules that can consume or gain authority from that fact.

A concrete implementation would record the cryptographic hash of the rule code, instead of just the literal rule name, but we use the name in our examples for readability. For practical workflows the set of rule names attached to each fact could be quite large. We assume that the underlying implementation represents multiple facts that include the same use-set efficiently. This could be done by recording the hash of each *set* of rule hashes, instead of listing the individual per-rule hashes directly.

The use-set of a fact determines the business-level meaning of the fact. In our coin transfer example the consumed and created `Coin` facts all have a use-set that specifies the single `transfer` rule. Coins are things that can be transferred and nothing else. As we will see in §3.7, including the use-set directly in facts makes it easy to upgrade both the format of facts and the rule code, which is necessary in most practical information systems.

2.7 Data Model

We can now restate the example facts from §2.1 in full, assuming that Alice starts out owning 100 coins.

```
Coin   [issuer = !Isabelle, holder  = !Alice]
   by  {!Isabelle, !Alice}  obs {}
   use {'transfer}          num 100

Offer  [id = '1234, terms = "To purchase a Guitar",
        giver = !Alice, receiver = !Bob]
   by  {!Alice}             obs {!Bob}
   use {'transfer}          num 1

Accept [id = '1234, accepter = !Bob]
   by  {!Bob}               obs {!Alice}
   use {'transfer}          num 1
```

In summary, the word `Coin` in the first fact is its *tag*, and the following record value the *payload*. The `by` keyword marks the *by-authority* set, which is the set of parties the fact has been authorized by. The `obs` keyword marks the *obs-authority* set, which lists extra parties that can observe the fact but do not necessarily authorize it. The `use` keyword marks the *use-set*, which lists the names of rules that can consume a fact or gain authority from it. Finally, the `num` keyword marks the *weight* of the fact, which can be interpreted as the number of active copies of the fact on the ledger.

The full data model is specified below. The current ledger state is a map from *Fact* to its *Weight*, where the *Fact* includes the by-authority, obs-authority and use-set along with the tag and payload. Facts with differing authority or use sets are different facts.

$$
\begin{array}{ll}
\mathit{State} & = \mathit{Map\ Fact\ Weight} \\
\mathit{Fact} & = (\mathit{Name}, \mathit{Payload}, \mathit{By}, \mathit{Obs}, \mathit{Use}) \\
\mathit{Payload} & = \mathit{List}\ (\mathit{Name}, \mathit{Value}) \\
\mathit{By} & = \mathit{Set\ Party} \\
\mathit{Obs} & = \mathit{Set\ Party} \\
\mathit{Use} & = \mathit{Set\ Name} \\
\mathit{Weight} & = \mathit{Nat}
\end{array}
$$

When the ledger state includes a fact with weight zero we treat this as identical to a state without that fact included at all. We could equivalently specify the state as being a *multiset* of facts, but in most cases we find using a map from facts to weights to be more intuitive. Note the *State* here is the current *active state* of the ledger. In related work, the *ledger* itself is often defined as a sequence of transactions that describe the full history of changes. Given the sequence of transactions starting from the empty state, it is always possible to rebuild the active state after any prefix of those transactions has been applied [55].

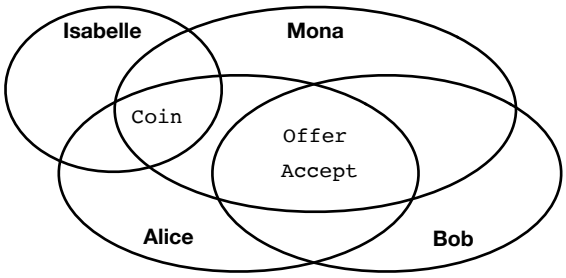

Figure 2: Fact Visibility for Monitored Coin Transfer

3 PRIVACY

In practical multi-party workflows it is often not desirable, or even legal, for all data used in the workflow to be provided to all parties. In the coin transfer example from §2, we could assume that Alice did not want to reveal the total number of coins she holds to Bob, nor the item she wishes to purchase (the Guitar) to Isabelle. Conversely, sometimes the details of a workflow *must* be revealed to third parties that do not otherwise participate in that workflow. Details of transactions may need to be sent to financial regulators that monitor the operation of markets, or to credit agencies that offer loans based on the spending patterns of their clients. For the sake of example, we extend the coin transfer workflow described in the previous section with an extra party, Mona, who monitors all coin transactions. Figure 2 describes fact visibility for the new workflow as a Venn diagram. The extended transfer rule is as per Figure 1, but with Mona listed as an observer of the produced coin fact.

3.1 Transaction and Validation

Assume that Alice, Bob, Mona and Isabelle all have their own computers in their own offices, each containing a *fragment* of the ledger state as per Figure 2. Each party also has a copy of the extended transfer rule. Alice decides that it is time to perform the transfer, and builds the following transaction structure:

```
Transaction
 seq    = ... sequence number ...
 rule   = ... hash of the transfer rule ...
 input  = [ Offer [id = '1234, terms = "To purchase a Guitar",
                   giver = !Alice, receiver = !Bob]
            by  {!Alice}            obs {!Mona, !Bob}
            use {'transfer}         num  1

          , Accept [id = '1234, accepter = !Bob]
            by  {!Bob}              obs {!Mona, !Alice}
            use {'transfer}         num 1

          , Coin   [issuer = !Isabelle, holder = !Alice]
            by  {!Isabelle, !Alice} obs {!Mona}
            use {'transfer}         num 1 ]

 output = [ Coin   [issuer = !Isabelle, holder = !Bob]
            by  {!Isabelle, !Bob}   obs {!Mona}
            use {'transfer}         num 1 ]
```

The transaction includes a fresh sequence number, the hash of the transfer rule definition, the list of facts being consumed (input) by the transaction, and the list of new fact created (output). Alice would like the other parties to agree that this is a valid execution of the transfer rule, and update their own local fragments of the ledger state. However, as per Figure 2, not all parties are entitled to see all facts listed in the transaction.

Recall from §2.5 that a party P can see a fact F when it is listed in either its *by-authority* set or its *obs-authority* set. We express this as a predicate, where the functions auth-by and auth-obs retrieve the corresponding sets from the fact value.

$$\text{sees } P\ F = (P \in \text{auth-by } F) \vee (P \in \text{auth-obs } F)$$

Applying this predicate to the facts in the transaction, Alice computes that 1) the `Offer` and `Accept` should be visible to Alice, Bob and Mona; 2) the input `Coin` fact should be visible to Isabelle, Alice and Mona, and 3) the output `Coin` fact should be visible to Isabelle, Bob and Mona. Importantly, the details of the transaction do not reveal how many coins Alice happens to have when she builds it. Both Isabelle and Mona will already know how many coins Alice has, as they have seen previous coin transfers, but there is no reason for this information to appear in the transaction structure itself.

Alice cannot send the complete transaction to all parties as they are not all entitled to see all the facts. Instead, Alice computes a *restricted view* of the transaction for each of the other parties, blinding the facts that a particular party is not entitled to see from their view before sending it.

3.2 Transaction Views

We abbreviate the four weighted facts in the transaction structure as d_1, d_2, d_3, d_4. The letter d is a mnemonic for *factoid* — being an "unreliable" fact, because the weight might be zero. We express the complete transaction as the following tuple, using $h(X)$ to mean the hash of value X, and tx to denote the name of the transfer rule:

$$(seq,\ h(tx), [d_1, d_2, d_3], [d_4])$$

This tuple contains the transaction sequence number, the hash of the rule definition, the list of input factoids, and the list of output factoids. The views for each party are computed by replacing some of the factoids by their *blinded hashes*. A blinded hash is a cryptographic hash which has been combined with a random salt value, so that the plaintext data cannot feasibly be recovered by brute force guessing. We write $s_1..s_4$ as the salts for each factoid.

Before computing the view for each party, Alice first computes an overall transaction identifier by replacing all factoids in the transaction with their blinded hashes, then hashing the result:

$$h((seq,\ h(tx), [h(d_1, s_1), h(d_2, s_2), h(d_3, s_3)], [h(d_4, s_4)]))$$

This is a unique(ish) identifier for the transaction, provided the hash values are long enough that we will not see a collision in practice.

Now, as Isabelle is entitled to see the `Coin` facts but not the `Offer` or `Accept` facts, she receives a view containing the data and salt values for the `Coin` facts, but only the blinded hashes of the `Offer` and `Accept` facts:

$$\text{for Isabelle: } (seq,\ h(tx), [h(d_1, s_1), h(d_2, s_2), (d_3, s_3)], [(d_4, s_4)])$$

Similarly, Bob is entitled to see the `Offer`, `Accept` and produced `Coin` fact, but not the consumed `Coin` fact, so receives a corresponding view.

$$\text{for Bob: } (seq,\ h(tx), [(d_1, s_1), (d_2, s_2), h(d_3, s_3)], [(d_4, s_4)])$$

Finally, Mona is entitled to see all the facts, so she gets the full unblinded transaction.

$$\text{for Mona: } (seq,\ h(tx), [(d_1, s_1), (d_2, s_2), (d_3, s_3)], [(d_4, s_4)])$$

All four parties, including Alice, can now use their own view to compute the same transaction identifier. Isabelle was not given the data for the `Offer` or `Accept` facts, but as she knows their blinded hashes she can still compute the hash of the overall transaction. Isabelle *can* infer that the giver and receiver fields of the `Offer` must have contained values `!Alice` and `!Bob` respectively. Isabelle knows which rule the transaction has been generated from, and that rule says the giver and receiver of an `Offer` must match the corresponding fields of the `Coin` facts that she does see. However, Isabelle cannot see that the coin is being transferred `"To purchase a Guitar"`, because this information was only present in the `Offer`.

3.3 Consensus

Once each party has received their own transaction view, they can compare it against their own fragment of the ledger state and confirm with each other whether their views are valid. The fragment of ledger state visible to Isabelle includes the total weight of `Coin` facts currently held by Alice. When Bob confirms with Isabelle that her view is valid, this tells Bob whether Alice actually has a coin to transfer to him. Similarly, when Isabelle confirms with Bob that his view is valid, this tells Isabelle that Bob really did agree to the transfer. In a practical workflow Isabelle might represent a commercial Bank, and in this case Bob would likely trust Isabelle to answer truthfully when asked if enough coins are available for a transfer, even though he does not want her to know that he is adding to his collection of guitars.

Consensus for the Rainfall model could be implemented in several different ways. For a small number of parties, such as to manage commercial workflows between banks, it would be sufficient for each party to confirm the transaction directly with all others. This would require $O(n^2)$ confirmations, but in the happy case the only information that needs to be exchanged is that the confirming party agrees with the transaction view identified by its hash code. For a greater number of parties, cryptographically signed confirmation messages could be propagated with a peer-to-peer protocol [21], or Byzantine Fault Tolerant (BFT) consensus protocol [29, 37, 43]. Both Corda [32] and Hyperledger Fabric [9] are existing networking frameworks that can be configured with custom transaction validation routines.

Alternatively, `Mona` could represent a monitoring company that simply receives transaction views and archives them. In applications where the parties know each other and are expected to be honest, they may not need to synchronously confirm every transaction. If there are any disputes between Alice, Bob or Isabelle, then they could retrieve the complete views given to Mona, and validate that the transaction hashes of those views match their own.

3.4 Nested Transactions

For the transaction in §3.1, Isabelle's view is constructed by blinding the `Offer` and `Accept` facts, as Isabelle is not entitled to see them. Blinding these facts means that Isabelle cannot use the input list to directly execute the rule and check its output. In a concrete implementation with a trusted monitor, such as Mona, this would not matter as the other parties would expect Mona to answer truthfully when asked if the transaction is valid. If we instead wish Isabelle to be able to validate the output directly, then we can split the transfer rule into two parts: one that combines Alice's offer with Bob's agreement, and another to perform the actual transfer.

```
rule  agreeOffer
await Offer  [id = ?i, giver = ?g, receiver = ?r] gain {g}
  and Accept [id = i,  accepter = r]              gain {r}
to
  say Agreed [giver = g, receiver = r]
   by {g,r}  obs {!Mona,!Isabelle} use {'performTransfer}

rule  performTransfer
await Agreed [giver = ?g, receiver = ?r]        gain {g,r}
 and  Coin   [issuer = !Isabelle, holder = g]
      gain {!Isabelle,g}
to
  say Coin   [issuer = !Isabelle, holder = r]
   by {!Isabelle,r} obs {!Mona} use ...
```

With these two rule definitions Alice can build a *nested transaction* [42], which for our purposes is a list containing the subtransaction for each of the rule firings. For the above rules, the first subtransaction will consume the `Offer` and `Accept` facts to produce an intermediate `Agreed` fact that is authorized by both Alice and Bob. The second subtransaction will then immediately consume this `Agreed` fact along with the input `Coin` fact, to produce the output `Coin` fact. Both the `Agreed` fact, as well as the input `Coin` fact are guaranteed to be visible to Isabelle, so she will be able to re-execute the second subtransaction and validate the output herself. Isabelle will not be able to see the terms of the offer, but she will be able to confirm with Bob that he agreed to it.

In related work, the DAML [2] ledger model combines facts and rules into a *contract instance*, similar to an object in an OO model — where our facts would be the object's fields, and our rules its methods. Parties using the system authorize whole objects, instead of using separate systems for the authorization of facts (our by-authority set) and rules (our use-set). DAML is based on UTxO [54], so calling a method on an object typically causes it to allocate some new objects and consume the called object.

A design decision of DAML is to ensure that each party who authorizes an object is able to re-execute the subtransaction that consumes that object. Supporting this requires the system to sometimes *divulge* additional objects to an authorizing party that were not visible until the object they authorized was consumed. However, the object oriented code structure of DAML causes the transactions produced by method invocation to have a nested form similar to the above. In DAML, the details of the equivalent `performTransfer` subtransaction can be sent to `Isabelle`, leaving the details of `agreeOffer` private between Alice and Bob.

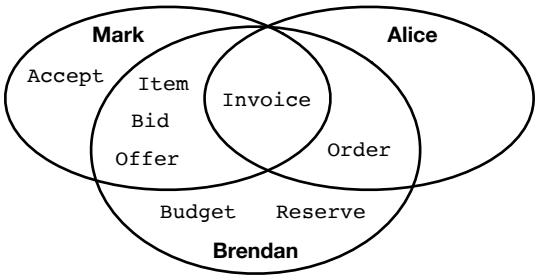

Figure 3: Fact Visibility for Market Example

3.5 Incidental Observers

For the transaction from §3.1, although Alice was the one that formed this transaction, she herself is not listed as an observer of the output `Coin` fact. When Alice computes the restricted views for Bob, Isabelle and Mona she knows that those parties will add this output fact to their own stores, but she should not add it to her own. As Alice is not an observer of the output fact, any other party that builds a transaction that consumes this fact will not inform her that this has happened. In this case we say that Alice is an *incidental observer* of the output `Coin` fact.

3.6 Fact Selection and Checking

In the coin transfer rules discussed so far, each pattern matching clause only gathered a single fact at a time. Consider instead a market workflow where rules must perform more complex queries, such as selecting the cheapest item that matches some criteria. Figure 3 shows the fact visibility for an example workflow. Mark runs the market, Brendan is a broker, and Alice is a client who wishes to buy some items. Alice must interact with Mark through Brendan, instead of communicating directly, like so:

(1) Mark maintains `Item` facts that describe items available for sale, along with their asking price. Brendan can pay the asking price to buy an item immediately, or bid below the asking price, which Mark may or may not accept.
(2) Alice creates `Order` facts describing the sort of items she wishes to buy, her price limit per item, and her total budget for items of this sort. Brendan can see Alice's `Order` facts, but Mark naturally cannot, as Alice does not want Mark to know the maximum price she is willing to pay.
(3) Brendan attempts to fulfill Alice's orders by bidding on multiple items concurrently. Brendan maintains a local `Budget` fact recording how much of Alice's total budget has not yet been committed to bids. This ensures he does not accidentally bid on items that Alice is not prepared to pay for.
(4) When Brendan wishes to enter a new bid he creates a local `Reserve` fact, which indicates that some of Alice's budget needs to be reserved for this bid. This causes one of Brendan's business rules to fire, which first checks that enough `Budget` is available, and if so, subtracts from the current `Budget`, then sends an active `Bid` to Mark.
(5) On Mark's side, one of Mark's business rules finds the cheapest available `Item` that matches the description in Brendan's `Bid`, and if the bid price is lower than the ask price, converts the bid to an outstanding `Offer`.
(6) Later, if Mark decides to accept Brendan's lower offer, then Mark creates a local `Accept` fact. One of Mark's business rules matches the `Accept` with the corresponding `Offer` and `Item`, consumes all three, and creates an `Invoice` which is visible to all three parties.

An interesting aspect of this workflow is that different parties in the system will naturally want to maintain their own local business rules, and corresponding fact declarations. The way Brendan accounts for bids he has placed on Alice's behalf is of no concern to Mark, but expressing all rules in the same framework means they can communicate directly. Here is the rule Brendan uses to check the `Budget` and produce a `Bid`.

```
fact Item    [lot:  Nat,  desc:   Text, ask: Nat]
fact Bid     [lot:  Nat,  offer:  Nat]
fact Order   [desc: Text, limit:  Nat,  budget: Nat]
fact Budget  [desc: Text, total:  Nat,  remain: Nat]
fact Reserve [lot:  Nat,  bid:    Nat]

rule  reserve
await Order   [desc  = ?d, limit  = ?l]
      consume none                       gain  {!Alice}
  and Item    [lot   = ?o, desc   = d,   ask = ?a]
      select first a   consume none      check {!Mark}
  and Budget  [total = ?t, remain = ?m] gain  {!Brendan}
  and Reserve [lot   = o,  bid    = ?b]
      where b <= l && b <= a && b <= m   gain  {!Brendan}
 to union
  (say Budget  [desc  = d,  total = t,  remain = m - a]
      by {!Brendan}  use ...)
  (say Bid     [lot   = o,  price  = a]
      by {!Brendan, !Alice}  obs {!Mark}  use ... })
```

The `Order` pattern has a `consume none` clause to indicate that we only want to read the fact data, rather then consume any weight of it, as the complete order might not be fulfilled yet. The `Order` pattern does not mention the `budget` field as this particular rule does not use it. The `Item` pattern has a `select first a` clause to indicate that all items that match the description in the order should be sorted by the asking price `a`, and the first one selected. Brendan should only bid on the cheapest matching item available. The `Reserve` pattern has a `where` clause to check the bid being placed is no more than Alice's price limit, no more than the market asking price, and no more than the remaining budget for Alice.

When the rule matches on its facts it will `gain` the authority of the client and broker. The output `Budget` only needs to be authorized by Brendan as this is his internal accounting. The output `Bid` is authorized jointly by Brendan and Alice, as Brendan is entering the bid on behalf of Alice. Finally, in the pattern for `Item`, the rule `check`s that this fact has been authorized by Mark rather than trying to `gain` his authority. The `check` clause allows patterns to match on observable facts without the rule name being mentioned in the use-set of that fact. Mark should not need to concern himself with Brendan's accounting, so the `Item` facts that Mark creates do not need to mention Brendan's rules. Brendan's `reserve` rule can still execute because it is not consuming any facts authorized by Mark, or producing any that are authorized in his name.

3.7 Upgrade

As a final example, in practical workflows it is necessary to upgrade data formats and business rules as requirements change. In Rainfall, upgrading workflows is easy as the use-sets attached to each fact can be manipulated directly:

```
rule   upgrade
await  Coin [issuer = ?s, holder = ?h]      gain {s,h}
   and LetsUpgrade [party = s, rules = ?rs] gain {s}
   and LetsUpgrade [party = h, rules =  rs] gain {h}
to say Coin [issuer = s, holder = h]
    by {s,h} use rs
```

This rule allows the issuer and holder of a `Coin` to jointly agree to change the fact's use-set, and the new use-set can mention new versions of existing rules. There is no need for a privileged operator party to orchestrate the upgrade. The meaning of facts is controlled by the parties that authorize them, and not the people that operate the ledger system.

4 SEMANTICS

The Rainfall semantics is defined in terms of a core language where pattern matching is expressed as set comprehension style generators. For expositional purposes we express rule bodies using a version of Simply Typed Lambda Calculus (STLC) with records and sets, but a production implementation could also use a more expressive language like System-F, or a well defined byte code. The key ideas of our system are embodied in the rule structure and authority mechanism, while the language of rule bodies is arbitrary.

4.1 Grammar

The grammar for the core language is in Figure 4. A matching desugared version of the coin transfer rule from Figure 1, using `!Mona` as an observer, is in Figure 5. A *Rule* has a name, pattern matching clauses, and a term for the body to produce a set of new factoids. We use EBNF, so *Match*+ in the production for *Rule* requires at least one match clause. Each *Match* clause has form:

$$X \texttt{ from } N \texttt{ where } M \texttt{ select } C \texttt{ consume } U \texttt{ gain } I$$

This says we should scan through all facts in the store with name N, binding each in turn to the variable X which is in scope for the sub-terms in the clause. We then *gather* all such facts that satisfy predicate M into a set, *select* the single fact specified by C, *consume* the weight specified by U, and *gain* the authority specified by I. In the select clause C, the `any` keyword indicates that any gathered fact that satisfies the `where` predicate can be selected. For `first` M and `last` M we sort facts by the key M and take the first or last one. Our gather/select/consume/gain process is a regularized database query, reminiscent of the FLWOR blocks of XQuery [13]. Join style queries are expressed using multiple fact matching patterns. We default previously elided `select`, `consume` and `gain` clauses to `select any`, `consume 1` and `gain {}` respectively.

In the term language we require a few primitive operators to split out the components of a fact value — `fact'payload` and so on. In Figure 5, however, we have left applications of `fact'payload` implicit for readability, and retained some standard infix operators. The `check` clauses used in §3.6 can be desugared into applications of `fact'by` that check the authority of a fact in a `where` clause.

$$
\begin{array}{lll}
N, \textit{Name} & ::= & \ldots \\
L, \textit{Label} & ::= & \ldots \\
X, \textit{Var} & ::= & \ldots \\
R, \textit{Rule} & ::= & \texttt{rule } \textit{Name} \texttt{ await } \textit{Match}+ \texttt{ to } \textit{Term} \\
H, \textit{Match} & ::= & \textit{Var} \texttt{ from } \textit{Name} \texttt{ where } \textit{Term} \\
 & & \quad \texttt{select } \textit{Select} \texttt{ consume } \textit{Consume} \texttt{ gain } \textit{Gain} \\
C, \textit{Select} & ::= & \texttt{any} \mid \texttt{first } \textit{Term} \mid \texttt{last } \textit{Term} \\
U, \textit{Consume} & ::= & \texttt{none} \mid \textit{Term} \\
I, \textit{Gain} & ::= & \texttt{none} \mid \textit{Term} \\
T, \textit{Type} & ::= & \texttt{Unit} \mid \texttt{Bool} \mid \texttt{Nat} \mid \texttt{Text} \mid \texttt{Symbol} \mid \texttt{Party} \\
 & \mid & \texttt{Set } \textit{Type} \mid \texttt{Sets } \textit{Type} \mid \texttt{Fact } \textit{Type} \mid \texttt{FACT} \\
 & \mid & [(\textit{Label} : \textit{Type})*] \mid \textit{Type} \rightarrow \textit{Type} \\
M, \textit{Term} & ::= & \textit{Literal} \mid \textit{Var} \mid \textit{Term}\ \textit{Term} \mid \lambda \textit{Var} : \textit{Type}.\ \textit{Term} \\
 & \mid & [(\textit{Label} = \textit{Term})*] \mid \textit{Term}\ .\ \textit{Label} \mid \{\mid \textit{Term}* \mid\} \\
 & \mid & \texttt{say } \textit{Name}\ \textit{Term} \\
 & & \quad \texttt{by } \textit{Term} \texttt{ obs } \textit{Term} \texttt{ use } \textit{Term} \texttt{ num } \textit{Term} \\
V, \textit{Value} & ::= & \textit{Literal} \mid \lambda \textit{Var} : \textit{Type}.\ \textit{Term} \\
 & \mid & [(\textit{Label} = \textit{Value})*] \mid \{\textit{Value}*\} \\
 & \mid & \textit{Fact} \mid \{\textit{Fact} \mapsto \textit{Weight}\} \\
L, \textit{Literal} & ::= & \texttt{unit} \mid \textit{Bool} \mid \textit{Nat} \mid \textit{Text} \mid \textit{Symbol} \mid \textit{Party}
\end{array}
$$

(primitive operators)

$$
\begin{array}{ll}
\texttt{fact'payload}_T & :: \texttt{Fact } T \rightarrow T \\
\texttt{fact'by}_T & :: \texttt{Fact } T \rightarrow \texttt{Set Party} \\
\texttt{fact'obs}_T & :: \texttt{Fact } T \rightarrow \texttt{Set Party} \\
\texttt{fact'use}_T & :: \texttt{Fact } T \rightarrow \texttt{Set Symbol} \\
\texttt{sets'union}_T & :: \texttt{Sets } T \rightarrow \texttt{Sets } T \rightarrow \texttt{Sets } T
\end{array}
$$

(environments)

$$
\begin{array}{lll}
\Gamma, \textit{Env} & ::= & \cdot \mid \textit{Env},\ \textit{Var} : \textit{Type} \\
\Sigma, \textit{Decls} & ::= & \cdot \mid \textit{Decls},\ \textit{Name} : [(\textit{Label} : \textit{Type})*]
\end{array}
$$

Figure 4: Core Language Grammar

```
rule  transfer
await offer  from Offer
       where true
       select any  consume 1  gain {offer.giver}
 and  accept from Accept
       where accept.id       == offer.id &&
             accept.accepter == offer.receiver
       select any  consume 1  gain {offer.receiver}
 and  coin   from Coin
       where coin.issuer     == !Isabelle &&
             coin.holder     == offer.receiver
       select any  consume 1  gain {!Isabelle, offer.giver}
to
     say Coin [ issuer = !Isabelle
              , holder = offer.receiver ]
      by  {!Isabelle,offer.receiver}  obs {!Mona}
      use {'transfer}                 num 1
```

Figure 5: Desugared Coin Transfer Rule

4.2 Static Semantics

Our type language is standard. In Figure 4 type (`Sets` *Type*) classifies multisets. The type (`Fact` *Type*) classifies fact values whose payload has type *Type*. The `FACT` type is used as the supertype of all such fact types, omitting a parameter for the payload type so that rule bodies can produce sets of factoids of differing sorts.

The typing rules are in Figure 6. Most judgment forms use two environments: *Decls* (Σ) that maps fact names to their payload types, and *Env* (Γ) that maps variable names to their types. The grammars for the environments are back in Figure 4. In the source language example in Figure 1 we specified the payload type of each fact using the `fact` keyword. In the static semantics here we assume all such types are added to the initial *Decls* environment.

The judgment ($\Sigma \vdash R$ **ok**) checks that rule R is well typed. In the premises we check the sequence of pattern matches, producing a type environment Γ, that lists the types of variables that are in scope in the body of the rule. The body M produces a multiset of new factoids. The judgment ($\Sigma \mid \Gamma \vdash X$ `from` N `where` $M \ldots \Rightarrow \Gamma'$) checks a single pattern match, where X is bound to to each fact of name N in turn, and we keep the facts that match the boolean predicate M. The premise $(N : T) \in \Sigma$ retrieves the payload type T of the fact, which is used to construct the type of X which is in scope in the rule body M. Checking of *Select*, *Consume*, *Gain* and *Term* expressions is straightforward.

4.3 Dynamic Semantics

The evaluation rules are in Figure 7. We use the abbreviation *Auth* to mean a set of party values that authorize some fact, *Facts* to mean a set of facts and *Factoids* a map of facts to their weights. We use *Store* to also map facts to their weights, but name it differently to hint that this is the current ledger state used for rule evaluation. We use *Env* to map variable names to their values. In the notation we indicate that a variable stands for a collection by including a superscript that indicates the size of that collection, so F^n would stand for a set of facts with size n.

This semantics can be used to both execute rules to produce a transaction as per §3.1, and also to validate that a transaction is well formed. The semantic rules are non-deterministic. Given a particular store and production rule definition, it may be possible to execute that production rule by matching several different subsets of facts, and the semantic rules do not specify which particular subset to use. When a particular party builds a transaction and submits the views to others, it is up to the submitter to resolve any non-determinism as they see fit. Rainfall is a *contract system* also in the sense of specifying a range of valid behavior, rather than an abstract machine that fixes a single order for rule evaluation.

In Figure 7, starting with the top-level EvFire rule, the judgment: ($A_{sub} \mid S \vdash R \Rightarrow F^r_{read} \mid D^s_{spend} \mid D^n_{new} \mid S'$ **fire**) says that a submitting party with authority A_{sub} and initial store S can execute rule R, which reads facts F^r_{read}, spends factoids D^s_{spend}, creates new factoids D^n_{new}, producing a new store S'. The result sets can be used to produce (or check) a transaction structure, where the input list in the transaction is formed from both F^r_{read} and D^s_{spend}, using zero valued weights for facts listed in F^r_{read} which are read but not consumed.

$$\boxed{\mathit{Decls} \vdash \mathit{Rule}\ \textbf{ok}}$$

$$\frac{\Sigma \mid \cdot \vdash H^n \Rightarrow \Gamma \qquad \Sigma \mid \Gamma \vdash M :: \texttt{Sets FACT}}{\Sigma \vdash \texttt{rule}\ N\ \texttt{await}\ H^n\ \texttt{to}\ M\ \textbf{ok}}$$

$$\boxed{\mathit{Decls} \mid \mathit{Env} \vdash \overline{\mathit{Match}} \Rightarrow \mathit{Env}}$$

$$\Sigma \mid \Gamma \vdash \cdot \Rightarrow \Gamma$$

$$\frac{\Sigma \mid \Gamma \vdash H \Rightarrow \Gamma' \qquad \Sigma \mid \Gamma' \vdash H^n \Rightarrow \Gamma''}{\Sigma \mid \Gamma \vdash H\,H^n \Rightarrow \Gamma''}$$

$$\boxed{\mathit{Decls} \mid \mathit{Env} \vdash \mathit{Match} \Rightarrow \mathit{Env}}$$

$$\frac{\begin{array}{l}(N : T) \in \Sigma \qquad \Gamma' = \Gamma,\ X : \texttt{Fact}\ T \\ \cdot \mid \Gamma' \vdash M :: \texttt{Bool} \quad \Gamma' \vdash C\ \textbf{ok} \quad \Gamma' \vdash U\ \textbf{ok} \quad \Gamma' \vdash I\ \textbf{ok}\end{array}}{\Sigma \mid \Gamma \vdash X\ \texttt{from}\ N\ \texttt{where}\ M\ \texttt{select}\ C\ \texttt{consume}\ U\ \texttt{gain}\ I \Rightarrow \Gamma'}$$

$$\boxed{\mathit{Env} \vdash \mathit{Select}\ \textbf{ok}}$$

$$\Gamma \vdash \texttt{any}\ \textbf{ok} \qquad \frac{\cdot \mid \Gamma \vdash M :: \texttt{Nat}}{\Gamma \vdash \texttt{first}\ M\ \textbf{ok}} \qquad \frac{\cdot \mid \Gamma \vdash M :: \texttt{Nat}}{\Gamma \vdash \texttt{last}\ M\ \textbf{ok}}$$

$$\boxed{\mathit{Env} \vdash \mathit{Consume}\ \textbf{ok}}$$

$$\Gamma \vdash \texttt{none}\ \textbf{ok} \qquad \frac{\cdot \mid \Gamma \vdash M :: \texttt{Nat}}{\Gamma \vdash M\ \textbf{ok}}$$

$$\boxed{\mathit{Env} \vdash \mathit{Gain}\ \textbf{ok}}$$

$$\Gamma \vdash \texttt{none}\ \textbf{ok} \qquad \frac{\cdot \mid \Gamma \vdash M :: \texttt{Set Party}}{\Gamma \vdash M\ \textbf{ok}}$$

$$\boxed{\mathit{Decls} \mid \mathit{Env} \vdash \mathit{Term} :: \mathit{Type}}$$

$$\frac{(X : T) \in \Gamma}{\Sigma \mid \Gamma \vdash X :: T} \qquad \frac{\Sigma \mid \Gamma,\ X : T \vdash M :: T'}{\Sigma \mid \Gamma \vdash \lambda X : T.\ M :: T \to T'}$$

$$\frac{\Sigma \mid \Gamma \vdash M :: T \to T' \qquad \Sigma \mid \Gamma \vdash M' :: T}{\Sigma \mid \Gamma \vdash M\ M' :: T'}$$

$$\frac{\{\Sigma \mid \Gamma \vdash M_i :: T_i\}^{i \leftarrow 1..n}}{\Sigma \mid \Gamma \vdash [l_1 : M_1 \ldots l_n : M_n] :: [l_1 : T_1 \ldots l_n : T_n]}$$

$$\frac{\Sigma \mid \Gamma \vdash M :: [l_1 : M_1 \ldots l_n : M_n] \qquad l : T \in \{l_1 : T_1 \ldots l_n : T_n\}}{\Sigma \mid \Gamma \vdash M\,.\,l :: T}$$

$$\frac{\{\Sigma \mid \Gamma \vdash M_i :: T\}^{i \leftarrow 1 \ldots n}}{\Sigma \mid \Gamma \vdash \{\mid M_1 \ldots M_n \mid\}^{i \leftarrow 1 \ldots n} :: \texttt{Set}\ T}$$

$$\frac{\begin{array}{ll}(N_{tag} : T) \in \Sigma & \\ \Sigma \mid \Gamma \vdash M_{payload} :: T & T = [l_1 : T_1 \ldots l_n : T_n] \\ \Sigma \mid \Gamma \vdash M_{by} :: \texttt{Set Party} & \Sigma \mid \Gamma \vdash M_{obs} :: \texttt{Set Party} \\ \Sigma \mid \Gamma \vdash M_{use} :: \texttt{Set Symbol} & \Sigma \mid \Gamma \vdash M_{num} :: \texttt{Nat}\end{array}}{\begin{array}{l}\Gamma \mid \Sigma \vdash \texttt{say}\ N_{tag}\ M_{payload} \\ \qquad \texttt{by}\ M_{by}\ \texttt{obs}\ M_{obs}\ \texttt{use}\ M_{use}\ \texttt{num}\ M_{num} :: \texttt{Sets FACT}\end{array}}$$

Figure 6: Static Semantics (selected rules)

The semantics is specified from a global point of view, where the *Store* includes the complete set of factoids visible to all parties. The particular subset of facts visible to a submitting party is controlled by A_{sub}, which is the authority of that party. In the premises of EvFire, we apply the pattern matches to produce a set of read and spent facts, along with the gained authority A_{gain}, which must cover the by-authority of all new factoids D^n_{new} produced by the body of the rule. The premise $S \downarrow D^s_{spend} \Rightarrow S'$ checks that factoids needing to be spent are available in the store S with sufficient weight, and then removes them from the store, producing a new store S'. Similarly $S' \uparrow D^n_{new} \Rightarrow S''$ adds the new facts produced by the rule body. The corresponding rules are in Figure 8.

Rules EvMatchNil/Cons apply the pattern matches to the store, gathering the set of facts read factoids spent, authority gained and the environment containing the matched facts. We use $\uplus$ operator to mean multiset union, where the weights of identical facts are summed. Rule EvMatchOne performs a single pattern match, performing the gather/select/consume/gain stages to produce the fact selected, a factoid also mentioning the weight consumed, and the authority gained from that fact. If a fact is to be read but not consumed then the weight W_{spend} will be zero. The $(F,\ 0)$ factoid produced by EvMatch will be eliminated by the use of $\uplus$ in EvMatchCons, but the fact F will be retained in the set of facts read.

Rule EvGather collects the facts that match the gather predicate. The premise is written as a set comprehension, where the clause "sees $A_{sub}\ F$" ensures we only include facts visible to the submitting party. The "sees" predicate was defined back in §3.1.

Importantly, we bind the *fact* value instead of the whole *factoid* to avoid confusion about what weight a pattern match should observe when a pattern before it consumes the same fact. Binding the whole factoid would allow rules to check the weights of factoids more directly than using `consume` clauses, but then interleaving consumption with matching would mean patterns could not be reordered freely. Similar issues arise in related *active database systems* [45]. A programmer would ultimately want to specify further behavior, but we leave this to future work.

Rules EvAny/First specify how a single fact should be selected from the set of gathered facts. With EvAny any fact can be selected. In EvFirst we compute a set of pairs D^m of sort keys and values, and select the value with the smallest key. Handling `last` is similar.

Rule EvConsumeSome evaluates the term specifying the fact weight to be consumed, and checks the current rule is in the use set of the fact. The EvConsumeNone version does not need the check, as the fact itself is not consumed. Rules EvGainNone/Some are similar, with EvGainSome checking that the fact supply the desired authority before returning it.

A direct implementation of the semantic pattern matching rules would essentially compute a relational join using naive cartesian product and filtering. A production implementation could instead use the RETE algorithm [20, 24], a parallel extension of it [10], or by conversion onto relational algebra for execution on a back-end relational database that maintains indexing structures.

Rule EvSay evaluates its arguments and produces the corresponding factoid. The remaining execution rules for the STLC term language are standard and have been omitted to save space.

4.4 Properties

We have mechanized the Rainfall semantics using the Isabelle/HOL interactive theorem prover, and proved several useful isolation and authorization properties. Using this theory as a basis, we have also mechanized the market example from §3.6 to demonstrate how to prove safety properties of business logic encoded in our system. The proof scripts and full statements of invariants are available at http://github.com/rainfall-lang.

4.4.1 *Properties of the Semantics.*

Theorem 4.1. *Frame Constriction: Given an arbitrary store, if a rule can fire using facts from that store, producing a set of facts read, factoids spent, and new factoids created, then the same rule can fire in a store containing only information from those produced sets.*

If $\quad A \mid S \vdash R \Rightarrow F^r_{read} \mid D^n_{spent} \mid D^m_{new} \mid S'$ **fire**
then $\ A \mid F^r_{read} \Cup D^n_{spent} \vdash R \Rightarrow F^r_{read} \mid D^n_{spent} \mid D^m_{new} \mid S''$ **fire**
where $S'' = D^m_{new} \uplus (F^r_{read} - D^n_{spent})$

Frame Constriction guarantees that the (unblinded) transaction structures described in §3.1 can always be validated in isolation, without needing the entire state of the ledger that existed at the time the transaction was formed.

In our formalization we use multisets for both the set of facts read F^r_{read} and factoids spent D^n_{spent}. When forming the store used for the second rule firing we need to ensure that facts listed in both F^r_{read} and D^n_{spent} are not counted twice. To achieve this we use the $\Cup$ operator which performs a version of multiset union where elements that exist in both of the argument sets are given their maximum weight in the result set, rather than their sum.

Theorem 4.2. *Focused Firing: Facts that are not visible to a submitting party do not influence rule firing:*

If $\quad A_{sub} \mid S \vdash R \Rightarrow F^r_{read} \mid D^n_{spent} \mid D^m_{new} \mid S'$ **fire**
and $\ \forall f \in S_{others}.\ \neg(\text{sees}\ A_{sub}\ f)$
then $\ A_{sub} \mid S \uplus S_{others} \vdash R \Rightarrow F^r_{read} \mid D^n_{spent} \mid D^m_{new} \mid S'$ **fire**

Theorem 4.3. *Visible Spending: A rule firing cannot influence facts that are not visible to the submitting party.*

If $\quad A_{sub} \mid S \vdash R \Rightarrow F^r_{read} \mid D^n_{spent} \mid D^m_{new} \mid S'$ **fire**
then $\ \forall f \in F^r_{read} \Cup D^n_{spent}.\ \text{sees}\ A_{sub}\ f$

Focused Firing ensures that the validity of transaction views sent by the submitter to a receiving party will not be influenced by extra facts that are visible to the receiver but not the submitter. Dually, Visible Spending ensures that any extra facts that are visible to the receiver but not the submitter cannot be influenced by those views.

Theorem 4.4. *Authority Flow: If a fact created by a rule firing is authorized by some party, then the same party authorized a fact that was read or spent by that rule firing.*

If $\quad A_{sub} \mid S \vdash R \Rightarrow F^r_{read} \mid D^n_{spent} \mid D^m_{new} \mid S'$ **fire**
then $\ \forall f \in D^m_{new}.\ \forall a \in \text{auth-by}\ f.\ \exists d \in F^r_{read} \Cup D^n_{spent}.$
$\quad a \in \text{auth-by}\ d$

This key property of our authorization system implies that the submitter of a transaction cannot attach their own authority to any of the created facts. The authorization of facts is controlled by the production rules expressed in the system, rather than the particular parties that form transactions.

$$\boxed{\mathit{Auth} \mid \mathit{Store} \vdash \mathit{Rule} \Rightarrow \mathit{Facts} \mid \mathit{Factoids} \mid \mathit{Factoids} \mid \mathit{Store}\ \textbf{fire}}$$

$$\frac{\begin{array}{c} N \mid A_{sub} \mid S \mid \cdot \vdash H^m \Rightarrow F^r_{read} \mid D^s_{spend} \mid A_{gain} \mid E\ \textbf{matches} \quad S \downarrow D^s_{spend} \Rightarrow S' \\ E \vdash M \Downarrow D^n_{new}\ \textbf{eval} \quad \bigwedge\{A_{gain} \supseteq \text{auth-by}\ D \mid D \in D^n_{new}\} \quad S' \uparrow D^n_{new} \Rightarrow S'' \end{array}}{A_{sub} \mid S \vdash \texttt{rule}\ N\ \texttt{await}\ H^m\ \texttt{to}\ M \Rightarrow F^r_{read} \mid D^s_{spend} \mid D^n_{new} \mid S''\ \textbf{fire}} \quad \text{(EvFire)}$$

$$\boxed{\mathit{Name} \mid \mathit{Auth} \mid \mathit{Store} \mid \mathit{Env} \vdash \mathit{Matches} \Rightarrow \mathit{Facts} \mid \mathit{Factoids} \mid \mathit{Auth} \mid \mathit{Env}\ \textbf{matches}}$$

$$N \mid A \mid S \mid E \vdash \cdot \Rightarrow \emptyset \mid \emptyset \mid \emptyset \mid E\ \textbf{matches} \quad \text{(EvMatchNil)}$$

$$\frac{\begin{array}{c} N_{rule} \mid A_{sub} \mid S \mid E \vdash H \Rightarrow F \mid D \mid A_{gain} \mid E'\ \textbf{match} \\ N_{rule} \mid A_{sub} \mid S \mid E' \vdash H^n \Rightarrow F^e \mid D^s \mid A'_{gain} \mid E''\ \textbf{matches} \end{array}}{N_{rule} \mid A_{sub} \mid S \mid E \vdash H\ H^n \Rightarrow \{F\} \cup F^e \mid \{D\} \uplus D^s \mid A_{gain} \cup A'_{gain} \mid E''\ \textbf{matches}} \quad \text{(EvMatchCons)}$$

$$\boxed{\mathit{Name} \mid \mathit{Auth} \mid \mathit{Store} \mid \mathit{Env} \vdash \mathit{Match} \Rightarrow \mathit{Fact} \mid \mathit{Factoid} \mid \mathit{Auth} \mid \mathit{Env}\ \textbf{match}}$$

$$\frac{\begin{array}{rl} & E' = E, X \mapsto F \\ F^n \mid E \vdash X\ ;\ C \Rightarrow F\ \textbf{select} & N_{rule} \mid F \mid E' \vdash U \Rightarrow W_{spend}\ \textbf{consume} \\ A_{sub} \mid S \mid E \vdash X\ ;\ N_{fact}\ ;\ M \Rightarrow F^n\ \textbf{gather} & N_{rule} \mid F \mid E' \vdash I \Rightarrow A_{gain}\ \textbf{gain} \end{array}}{\begin{array}{c} N_{rule} \mid A_{sub} \mid S \mid E \vdash X\ \texttt{from}\ N_{fact}\ \texttt{where}\ M\ \texttt{select}\ C\ \texttt{consume}\ U\ \texttt{gain}\ I \\ \Rightarrow F \mid (F,\ W_{spend}) \mid A_{gain} \mid E' \end{array}} \quad \text{(EvMatchOne)}$$

$$\boxed{\mathit{Auth} \mid \mathit{Store} \mid \mathit{Env} \vdash \mathit{Var}\ ;\ \mathit{Name}\ ;\ \mathit{Term} \Rightarrow \mathit{Facts}\ \textbf{gather}}$$

$$\frac{F^n = \left\{ \begin{array}{ll} F & \mid F \in \text{dom}\ S \\ & , \text{name}\ F = N_{fact},\ \text{sees}\ A_{sub}\ F \\ & , (E,\ X \mapsto F \vdash M \Downarrow \texttt{true}\ \textbf{eval}) \end{array} \right\}}{A_{sub} \mid S \mid E \vdash X\ ;\ N_{fact}\ ;\ M \Rightarrow F^n\ \textbf{gather}} \quad \text{(EvGather)}$$

$$\boxed{\mathit{Facts} \mid \mathit{Env} \vdash \mathit{Var}\ ;\ \mathit{Select} \Rightarrow \mathit{Fact}\ \textbf{select}}$$

$$\frac{F \in F^n}{F^n \mid E \vdash X\ ;\ \text{any} \Rightarrow F\ \textbf{select}} \qquad \frac{\begin{array}{c} D^m = \{(V, F) \mid F \in F^n, (E, X \mapsto F \vdash M \Downarrow V\ \textbf{eval})\} \\ V' = \text{minimum}\ \{V \mid (V, \_) \in D^m\} \quad (F', V') \in D^m \end{array}}{F^n \mid E \vdash X\ ;\ \texttt{first}\ M \Rightarrow F'\ \textbf{select}} \quad \text{(EvAny/First)}$$

$$\boxed{\mathit{Name} \mid \mathit{Fact} \mid \mathit{Env} \vdash \mathit{Consume} \Rightarrow \mathit{Weight}\ \textbf{consume}}$$

$$N_{rule} \mid F \mid E \vdash \text{none} \Rightarrow 0\ \textbf{consume} \qquad \frac{N_{rule} \in \text{rules}\ F \quad E \vdash M \Downarrow W\ \textbf{eval}}{N_{rule} \mid F \mid E \vdash M \Rightarrow W\ \textbf{consume}} \quad \text{(EvConsumeNone/Some)}$$

$$\boxed{\mathit{Name} \mid \mathit{Fact} \mid \mathit{Env} \vdash \mathit{Gain} \Rightarrow \mathit{Auth}\ \textbf{gain}}$$

$$N_{rule} \mid F \mid E \vdash \text{none} \Rightarrow \emptyset\ \textbf{gain} \qquad \frac{N_{rule} \in \text{rules}\ F \quad E \vdash M \Downarrow A\ \textbf{eval} \quad A \subseteq \text{auth-by}\ F}{N_{rule} \mid F \mid E \vdash M \Rightarrow A\ \textbf{gain}} \quad \text{(EvGainNone/Some)}$$

$$\boxed{\mathit{Env} \vdash \mathit{Term} \Downarrow \mathit{Value}\ \textbf{eval}}$$

$$\frac{F = (N_{fact}, V_{payload}, A_{by}, A_{obs}, V_{use}) \quad E \vdash M_{payload} \Downarrow V_{payload}\ \textbf{eval} \quad \ldots}{E \vdash \text{say}\ N_{fact}\ M_{payload}\ M_{by}\ M_{obs}\ M_{use}\ M_{num} \Downarrow \{F \mapsto V_{num}\}\ \textbf{eval}} \quad \text{(EvSay)}$$

Figure 7: Dynamic Semantics

$$\boxed{\mathit{Store} \downarrow \mathit{Factoids} \Rightarrow \mathit{Store}} \qquad \boxed{\mathit{Store} \uparrow \mathit{Factoids} \Rightarrow \mathit{Store}}$$

$$\frac{D^n \subseteq S \quad S' = S - D^n}{S \downarrow D^n \Rightarrow S'} \qquad \frac{S' = S \uplus D^n}{S \uparrow D^n \Rightarrow S'}$$

Figure 8: Store Modification

4.4.2 Properties of the market example. We have also mechanized the market example in §3.6 in Isabelle/HOL and proven the following business level theorem:

Theorem 4.5. *Budget Adherence: The total value of invoices a broker receives for items bought on behalf of a client never exceeds the budget specified in the client's original order.*

We prove this by first specifying an invariant over all facts in the store, and then proving that the possible firings of each rule in the workflow all preserve the invariant. For example, the top-level statement for the `reserve` production rule is:

If $A_{sub} \mid S \vdash \texttt{reserve} \Rightarrow F^r_{read} \mid D^n_{spent} \mid D^m_{new} \mid S'$ **fire**
and store-ok S then store-ok S'

The statement of the invariant is available in the appendix. We also prove that the invariant is established for newly created budgets with no initial associated bids, invoices or offers.

Rainfall's production rule based programming model makes it easy to approach proofs of business level properties like this. Provided one can write down a suitable invariant, the task then decomposes automatically into a separate subgoal for each rule.

4.5 Extensions

The semantics described in this section has a few natural extensions that we have elected to omit to avoid obscuring the presentation.

4.5.1 Extended Weight Types. There is no particular reason why the weight of each fact must be restricted to being a natural number as per §2.2. Our dynamic semantics only depends on the representation of weights in three places: to combine the weights of factoids in EvMatchCons and EvStoreUp, and in the subset test and set difference operators in EvStoreDown.

In general it would sufficient to use an Abelian group which also has a partial order. An Abelian group provides a weight combining operator that is associative, commutative and invertible. Requiring weight combining operator to be associative and commutative allows the facts needed by a rule to be matched in arbitrary order. We need the partial order to determine if a fact with the required weight is present in the store, and the weight combining operator needs to be invertible so we can reduce the weight of facts in the store when they are spent.

An example extended weight type is a simple `Present` / `Absent` indicator. When multiple facts are added with a weight of `Present` then the resulting fact is still `Present`. If this `Present` fact is then consumed it becomes `Absent`. Such a weight type is used for facts like "someone needs to water the plants". As the plants should only be watered once, it does not make sense to allow a numeric weight with value greater than one. Later, when a real-world party actually gets around to watering the plants, then the whole fact can be removed from the store.

4.5.2 Minimum Weight Thresholds. In EvFire from Figure 7, when a production rule reads a fact but does not consume any weight of it, or gain any authority from it, then the fact appears in the set of facts read F^r_{read}, but not the map of factoids spent D^s_{spend}. This means the rule can fire if the required fact is present in the store with any non-zero weight. An extension is to allow a rule to wait for a particular weight of a fact to be available before firing, without also needing to consume it. To achieve this we would change the representation of F^r_{read} to also be a map of factoids, and add a (`require` M) form to *Consume* specifications, where M is a term that evaluates to the weight the rule must have before firing.

5 RELATED WORK

5.1 Linda-style Tuple Spaces

Linda [28] is a coordination model where processes communicate by adding, removing and non-destructively reading tuples from a globally shared tuple space. The basic Linda model is open, meaning that any party using it is free to add and remove tuples at will. This lack of access control or provenance information makes it unusable for as a communication medium for mutually distrusting parties.

Several extensions to the basic Linda model add metadata to the tuples that are similar to the 'by' and 'obs' authority sets of our own Rainfall model. SecSpaces [16] signs tuples with the private keys of parties that create them, and adds metadata that specifies the identities of those that can see and consume them. Merrick [39] describes a scoping/visibility system for tuple spaces where new scopes can be created at will and combined using a set of scope combinators. Oriol [44] describes a model of tagged sets where the tuples are identified by a formula in propositional logic that allows authority and visibility information to be encoded uniformly. Udzir [51] describes a model where collections of tuples have unique identifiers, and the client program must provide a matching runtime capability when accessing them. These systems refine the Linda data model, but do not provide a mechanism to allow parties using the system to combine authorized tuples to produce new ones that are authorized by any other party except themselves. The ability to do this is the main contribution of our own system.

5.2 Law Governed Linda

Law Governed Linda (LGL) [40, 41] takes the basic Linda data model and inserts a *controller* between the tuple space and each communicating process. Each controller has a copy of a communication law, written in a fragment of Prolog, that specifies the allowable interactions with the tuple space. For example, the law could state that a process may only create a tuple that includes a `from` field, when the value in that field is its own process identifier.

The codified law specifies the allowable *interaction* a process may have with the communication medium. The controllers are assumed to run on a trusted computing base, either as part of the physical server that provides the tuple space, or on a secure co-processor [41]. In contrast, the production rules in our Rainfall model do not limit the form of data added to the system. Instead, they specify how authorized facts that are already in the store may be combined to produce new authorized facts. The Authority Flow theorem of §4.4.1 ensures that we do not need to rely on a trusted computing base to enforce the rules of the system.

5.3 Extended Shared Prolog

Like LGL, Extended Shared Prolog (ESP) [17, 18] combines the Linda coordination model with rules written in a restricted subset of Prolog. In this case the rules are stored as special *program tuples* in the main tuple space, and the rules describe how existing tuples can be combined, rather than controlling the interaction between the tuple space and its clients. The format of each rule is similar to a Rainfall production rule, including a section to gather matching tuples, a section to decide which should be consumed, and a section to compute new tuples. Program tuples behave like triggers in an active database [45], where a rule is activated when all tuples it was waiting for become available. However, as with the basic Linda coordination model, there is no mechanism to enforce transitive authority, or track the provenance of created tuples.

5.4 Permissioned Distributed Ledgers

As mentioned in §3.4, DAML combines facts and rules into a *contract instance*, which is similar to an object in an Object-Oriented (OO) model. Objects are referred to by *contract identifiers*, which are equivalent to typed references in the OO model. The DAML coordination model is based on UTxO [54], so invoking a method on an object typically causes it to create some new objects, then consume/delete that object. Deleting an object causes any existing references to it to become dangling, and following a dangling reference at runtime causes an exception. In contrast, our Rainfall model identifies facts by their content, rather than using a physical reference or pointer value. If a particular fact is not available with sufficient weight then this inhibits rule firing, rather than being treated as an execution failure.

A DAML method can invoke methods on other objects that it already has a reference to, but cannot query the ledger state directly. This restriction is standard in the OO coordination model, where method code cannot directly query the runtime heap to discover other objects based on their field data. Instead, objects typically communicate using shared references to mutable data. However, as the DAML programming model purposefully does not include shared mutable data, the usual OO programming patterns are unavailable. In practice, ledger actions are coordinated by "nanobots", which are driver routines written in an external language. The query performed by a nanobot yields a set of contract identifiers, which are then passed to DAML code as arguments to method invocations. The Rainfall model was specifically developed to avoid the need for nanobots, while providing an authority system similar the one available in DAML.

Corda [32], and Hyperledger Fabric [9] are related permissioned distributed ledgers. Instead of defining a specific contract language, both systems allow custom procedures to be installed that accept transactions directly and report whether they are valid. In Corda the validation procedures are expressed in a version of JVM bytecode that has been modified to ensure execution is deterministic. In Hyperledger Fabric the validation procedures can be arbitrary native code encapsulated in a Docker [4] container. These systems both provide the networking layer for a distributed ledger system, but purposefully do not specify a programming model in sufficient detail to prove safety properties such as those in §4.4, leaving this as a separate implementation choice.

5.5 Actors, Process Algebras, and Constraints

Production rule systems like Rainfall have a passing similarity to the Actor [8] model, but the computation framework is quite different. Production rules do not maintain their own private state, or have instance identity in the sense that they are addressable by mailbox or channel names. However, one could compile an actor program *into* Rainfall, by building facts that represent the local state of each actor, and defining production rules to handle the messages. A proposed extension to Erlang provides the multi-headed pattern matching needed by production rules [50], though matching is performed on ordered streams of incoming messages, rather selecting from an unordered soup of tuples. ActorSpaces [7] is a related model that uses message passing for communication while also allowing messages to be broadcast to all actors in a group.

Existing process languages such as the Join Calculus [25] and the Chemical Abstract Machine (CHAM) [12] allow processes to wait for multiple related facts (messages) to become available before activating. Similar functionality is available in systems of Constraint Handling Rules (CHR) [26]. However, as with Extended Shared Prolog (§5.3) these systems do not have a built in authority or provenance mechanism that could be used to guide data privacy as described in §3.

5.6 Authorization Logics

The Dependency Core Calculus (DCC) [6] extends Moggi's computational lambda calculus with an extra judgment form that indicates the value produced by a computation is protected at a given security level. Abadi [5] studies DCC applied to access control and tracking in a distributed system. This work uses a proposition (P says A), where P is some principle/party that affirms statement A. The 'says' former abstracts away from the details of what exactly is being authenticated or authorized. The statement (P says A) can variously be interpreted as "P has caused A to be said", "A has been said on P's behalf" or "P supports A". Garg [27] gives a sequent style presentation with two judgement forms (A true) and (P affirms A). The 'affirms' form is internalized as a proposition (P says A). Garg's system comes with meta theory of Affirmation Flow, meaning that unless a principle P affirms a particular statement, no affirmations of the form (P affirms A) can be derived from it. Bowers [15] gives a natural deduction style presentation that also has a (P signed A) form to model a message being cryptograpically signed.

DCC and related systems are logics rather than programming languages that have a direct operational interpretation. The Aura language [36] then specifies a functional operational semantics, as well as a proof term assignment for a version of DCC. Proofs of authority can be passed to functions as pure proof terms. Rainfall is directly inspired by the DCC family of logics and languages. Instead of building functional proof terms to demonstrate authority, we gather it in stages, incrementally writing authorized facts back to the ledger. Our ledger then can be viewed as a distributed proof of authority, where versions of DCC style logical properties still apply. For example, our Authority Flow theorem (§4.4) is the operational version of Garg's [27] Affirmation Flow.

Acknowledgements Many thanks to Fil Mackay, Lance Arlaus, Raphael Speyer, Erwin Ramirez and Ben Sinclair for helpful discussions and pointers to related work.

REFERENCES

[1] CSL language guide documentation, release v0.30.0. https://deondigital.com/docs/v0.30.0/cslguide.pdf.

[2] DAML SDK documentation. https://docs.daml.com/index.html. Accessed: 2019-04-27.

[3] Financial Core Language (FCL). https://www.adjoint.io/docs/workflows.html. Accessed: 2019-04-27.

[4] The Docker website. https://www.docker.com/. Accessed: 2019-05-03.

[5] Martín Abadi. Access control in a core calculus of dependency. *Electronic Notes in Theoretical Computer Science*, 172, 2007.

[6] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G Riecke. A core calculus of dependency. In *Principles of Programming Languages (POPL)*, 1999.

[7] Gul Agha and Christian J Callsen. ActorSpace: an open distributed programming paradigm. In *Principles and Practice of Parallel Programming (PPoPP)*. ACM, 1993.

[8] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. Towards a theory of actor computation. In *International Conference on Concurrency Theory (CONCUR)*, 1992.

[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018.

[10] Mostafa M Aref and Mohammed A Tayyib. The Lana–Match algorithm: a parallel version of the Rete–Match algorithm. *Parallel Computing*, 24(5-6), 1998.

[11] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of Bitcoin transactions. *IACR Cryptology ePrint Archive*, 2017.

[12] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1), 1992.

[13] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: An XML query language. 2002.

[14] Sean Bowe, Ariel Gabizon, and Matthew D Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *International Conference on Financial Cryptography and Data Security*, 2018.

[15] Kevin D Bowers, Lujo Bauer, Deepak Garg, Frank Pfenning, and Michael K Reiter. Consumable credentials in logic-based access-control systems. In *Annual Network and Distributed System Security Symposium*, 2007.

[16] Nadia Busi, Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. SecSpaces: a data-driven coordination model for environments open to untrusted agents. *Electronic Notes in Theoretical Computer Science*, 68(3), 2003.

[17] Paolo Ciancarini. Coordinating rule-based software processes with ESP. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3), 1993.

[18] Paolo Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12(3), 1994.

[19] Ankush Das, Stephanie Balzer, Jan Hoffmann, and Frank Pfenning. Resource-aware session types for digital contracts. *arXiv preprint arXiv:1902.06056*, 2019.

[20] Robert B Doorenbos. Production matching for large learning systems. Technical report, Carnegie-Mellon University, 1995.

[21] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient broadcast in structured P2P networks. In *International workshop on Peer-to-Peer systems*, 2003.

[22] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN Notices*, volume 37. ACM, 2002.

[23] Charles L Forgy. OPS5 user's manual. Technical report, Carnegie-Mellon University, 1981.

[24] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Readings in Artificial Intelligence and Databases*. Elsevier, 1989.

[25] Cedric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Principles of Programming Languages (POPL)*, 1996.

[26] Thom Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37(1-3), 1998.

[27] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *IEEE Computer Security Foundations Workshop*, 2006.

[28] David Gelernter. Generative communication in Linda. *Transactions on Programming Languages and Systems*, 7(1), 1985.

[29] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Symposium on Operating Systems Principles*. ACM, 2017.

[30] Ian Grigg. EOS - An Introduction, 2017.

[31] Dominik Harz and William J. Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods. *CoRR*, abs/1809.09805, 2018.

[32] Mike Hearn. Corda: A distributed ledger. *Corda Technical White Paper*, 2016.

[33] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. ZCash protocol specification. *Tech. rep. 2016–1.10. Zerocoin Electric Coin Company*, 2016.

[34] IOHK. Formal specification of the Plutus core language (version 2.0). https://hydra.iohk.io/build/798158/download/1/plutus-core-specification.pdf. Accessed: 2019-04-27.

[35] Kurt Jensen. Coloured Petri nets and the invariant-method. *Theoretical computer science*, 14(3), 1981.

[36] Limin Jia, Jeffrey A Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: A programming language for authorization and audit. In *ACM SIGPLAN Notices*, volume 43, 2008.

[37] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3), 1982.

[38] Niels Lohmann, Eric Verbeek, Chun Ouyang, Christian Stahl, and Wil MP van der Aalst. Comparing and evaluating Petri net semantics for BPEL. *International Journal of Business Process Integration and Management*, 4(1), 2009.

[39] Iain Merrick and Alan Wood. Coordination with scopes. In *Symposium on Applied Computing*, 2000.

[40] Naftaly H Minsky and Jerrold Leichter. Law-governed Linda as a coordination model. In *European Conference on Object-Oriented Programming*, 1994.

[41] Naftaly H Minsky, Yaron M Minsky, and Victoria Ungureanu. Safe tuplespace-based coordination in multiagent systems. *Applied Artificial Intelligence*, 15(1), 2001.

[42] John Eliot Blakeslee Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Massachusetts Institute of Technology, 1981.

[43] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, 2014.

[44] Manuel Oriol and Michael Hicks. Tagged sets: a secure and transparent coordination medium. In *International Conference on Coordination Languages and Models*, 2005.

[45] Norman W Paton and Oscar Díaz. Active database systems. *ACM Computing Surveys (CSUR)*, 31(1), 1999.

[46] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ACM SIGPLAN Notices*, volume 35. ACM, 2000.

[47] Stuart Popejoy. The Pact smart contract language, 2016.

[48] Gary Riley. CLIPS basic programming guide, version 6.40 beta, 2017.

[49] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *CoRR*, abs/1801.00687, 2018.

[50] Martin Sulzmann, Edmund SL Lam, and Peter Van Weert. Actors with multi-headed message receive patterns. In *International Conference on Coordination Languages and Models*, 2008.

[51] Nur Izura Udzir, Alan M Wood, and Jeremy L Jacob. Coordination with multicapabilities. *Science of Computer Programming*, 64(2), 2007.

[52] Fabian Vogelsteller and Vitalik Buterin. ERC-20: A standard interface for tokens. https://github.com/ethereum/EIPs/blob/e7dac5b8287106143d361b1de3704ce0bba31983/EIPS/eip-20.md.

[53] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

[54] Joachim Zahnentferner. An abstract model of UTxO-based cryptocurrencies with scripts. *IACR Cryptology ePrint Archive*, 2018.

[55] Joachim Zahnentferner. Chimeric ledgers: translating and unifying UTxO-based and account-based cryptocurrencies. *IACR Cryptology ePrint Archive*, 2018.

A MARKET RULES

This section includes the full rule definitions of the market example that appears in §3.6. An executable version is available online.

```
fact Order   [desc: Text, limit: Nat,  budget: Nat]
fact Item    [lot:  Nat,  desc:  Text, ask: Nat]
fact Accept  [lot:  Nat,  price: Nat]
fact Offer   [lot:  Nat,  price: Nat]
fact Bid     [lot:  Nat,  offer: Nat]
fact Budget  [desc: Text, total: Nat, remain: Nat]
fact Reserve [lot:  Nat,  bid:   Nat]
fact Invoice [seller: Party, buyer: Party,
              desc: Text,    amount: Nat]


-- Brendan reserves a portion of the budget allotted to
-- Alice, and forwards a bid to Mark.
rule  reserve
await Order     [desc  = ?d, limit = ?l]
      consume none                       gain  {!Alice}
 and  Item      [lot   = ?o, desc = d, ask = ?a]
      select first a  consume none       check {!Mark}
 and  Budget    [desc  = ?d, total = ?t, remain = ?m]
      gain  {!Brendan}
 and  Reserve   [lot   = o,  bid    = ?b]
      where b <= l && b <= a && b <= m   gain {!Brendan}
 to union
      (say Budget [desc = d, total = t, remain = m - a]
       by {!Brendan} use {'reserve})
      (say Bid    [lot   = o, offer = b]
       by {!Brendan, !Alice} obs {!Mark} use {'bid})


-- Mark converts bids that Brendan has placed that are below
-- the asking price of the item into resting offers.
rule  bid
await Bid    [lot = ?o, offer = ?b]   gain {!Brendan}
  and Item   [lot = o,  ask   = ?a]
      where b < a  consume none       gain {!Mark}
 to
      say Offer [lot = o, price = b]
      by {!Brendan, !Mark} use {'accept}


-- Mark accepts a resting offer, removes the item listing
-- and produces an invoice for the sale price.
rule  accept
await Accept [lot = ?o, price = ?p]  gain {!Mark}
  and Offer  [lot = o,  price = p]   gain {!Brendan, !Mark}
  and Item   [lot = o,  desc  = ?d]
      consume 1                      check {!Mark}
 to
      say Invoice [ seller = !Mark, buyer = !Brendan
                  , desc = d, amount = p]
      by  {!Mark, !Brendan}
```

B MARKET INVARIANTS

The market example introduced in §3.6 used the "store-ok" invariant to ensure that, if the budgets are adhered to in a particular store, then after execution of any of the market rules, any updated budgets in the new store are also adhered to. A budget is adhered to when the total value of invoices issued to a broker for a particular client do not exceed the client's specified budget limit.

For all rules $r \in \{\texttt{accept}, \texttt{bid}, \texttt{reserve}\}$,
if $A_{sub} \mid S \vdash r \Rightarrow F^r_{read} \mid D^n_{spent} \mid D^m_{new} \mid S'$ **fire**
and store-ok S then store-ok S'

The definition of the invariant for stores finds all orders in the store, and ensures that for each order the order invariants are satisfied:

store-ok S =
  For all orders `Order` $o \in S$,
  require order-ok S o

The order invariant ensures that an order has at most one budget, and that the budget invariants are satisfied. An order with no associated budget is valid:

order-ok S o =
  For all budgets `Budget` $b \in$ budgets-for-order S o,
  require unique S b
  and require budget-ok S b

While our desired property is to show that the total value of invoices does not exceed the budget limit, our invariant must show a stronger property, which is that the total value of bids, invoices and offers must not exceed the budget limit. This stronger property is required as bids and offers can eventually result in invoices, as bids are transformed to offers, and offers are accepted. The budget invariant finds all associated bids, invoices and offers, and sums their prices to compute the total amount reserved by the budget. This total reserved amount plus the remaining budget must equal the budget limit:

budget-ok S b =
  Require budget-total b = *total* + budget-remain b
  where *total* = *bids* + *invoices* + *offers*,
  and *bids* = $\sum_{d \in \text{bids-for-budget } S\ b}$ bid-price d
  and *invoices* = $\sum_{i \in \text{invoices-for-budget } S\ b}$ invoice-price i
  and *offers* = $\sum_{o \in \text{offers-for-budget } S\ b}$ offer-price o

This invariant ensures that the reserved amount is less than or equal to the total budget, as all numbers are non-negative natural numbers. The functions bids-for-budget, invoices-for-budget and offers-for-budget compute the multiset of associated bids, invoices or offers, for a particular budget. The functions budget-total, budget-remain, bid-price, and so on, are accessor functions to retrieve components of the fact values.