Deon Digital CSL Language Guide Documentation

Release v0.60.0

Deon Digital

Apr 15, 2020

Contents

This is an introduction to and overview of Deon Digital's Contract Specification Language (CSL). As the name implies, CSL is used to specify contracts – details such as the particular execution method or how participating parties are authenticated/validated are deliberately not included in the CSL specification.

This guide is also available as a single PDF file here.

If you encounter any problems, don't hesitate to contact us at support@deondigital.com.

CHAPTER 1

Introduction

For an introduction to the CSL language with a simple walk-through of your first contract and explanation of the basic concepts, start here:

1.1 Contracts 101

In a nutshell, a CSL contract is a specification of a set of sequences of events. Each such event sequence is a complete way to conclude the contract successfully. Events are represented by *event values* which are inserted into the runtime system in a platform-dependent way. In the following, we shall use the terms "event" and "event value" interchangeably except where noted otherwise.

Execution of CSL contracts can be *monitored* as they unfold in time under the arrival of events. Specifically, given an event a contract can be reduced to a contract that represents the remaining obligations after the event has occurred.

Contracts are expressed using *composition*: Smaller contracts are combined into larger ones. A larger contract specifies a behaviour that is a combination of its constituent smaller contracts. The smallest contracts do not do anything other than being "finished" – either signaling a success or failure ("contract breach") state. New events are described by using a *prefix combinator*, which uses a predicate on incoming events to decide whether it is allowed or not. Many other contract combinators exist; for a comprehensive list see the *contract reference*.

1.2 Your first CSL contract

In this section we will guide you through the formulation of your first contract specification. Our goal is to describe a scenario in which the participant "Bob" sells one bicycle to "Alice" for 100 euros.

1.2.1 Specifying events

The first step in specifying our contract is to understand exactly what the scenario we want to describe actually contains:

- Is the bicycle delivered to Alice or must she pick it up?
- If it is delivered, is there a time limit for how long Alice wants to wait?
- If a third party, "Eve", comes along and offers 200 USD for the bicycle, will she get it instead of Alice?

- Is there one single bicycle, or will Bob be able to sell more bicycles?
- When does payment occur?

One could come up with many more considerations than the above for a real-world scenario. For didactic purposes we shall limit ourselves to a simplified bike shop:

- Alice orders – and pays for – a bicycle, and then it is delivered to her with no time constraints;
- No other parties than Alice and Bob exist;
- Bob has one bicycle and when he sells it he closes down his shop.

Our idealized transaction thus boils down to the following:

1. Alice orders a bicycle by transferring 100 euros to Bob;
2. Bob delivers one bicycle to Alice.

1.2.2 Encoding events

In the preceding section we have established that our contract considers two separate types of events: a `BikeOrder` event and a `BikeDelivery` event. The `BikeOrder` event encodes the order that Alice places for a bicycle, and it should therefore include the price she is willing to pay and whom she orders a bike from. We specify this in CSL as follows:

```
type BikeOrder : Event {
  amount : Int, // The amount of euros that Alice will pay for a bicycle.
  recipient : Agent // The recipient of the order.
}
```

This CSL code specifies that a `BikeOrder` is a type of event which has the two fields `amount` and `recipient` in addition to the standard fields of events (which we shall see in a bit). Note that everything on a line that comes after `//` is treated as a comment and is ignored by CSL. CSL is not whitespace-sensitive, so we are free to layout the code as we please.

The `BikeDelivery` event represents the delivery of an item to a receiver. Hence, we include as extra fields in this event the `recipient` of the bicycle:

```
type BikeDelivery : Event {
  recipient : Agent // The recipient of the bicycle
}
```

Both `BikeOrder` and `BikeDelivery` are specified as a *subtype* of an event type called `Event` in CSL. This means that our newly defined types are events that we can use in contracts. Any event type we want to use in contracts has to be declared as a subtype of `Event` like this. In CSL we refer to this relationship as *inheritance*: our `BikeOrder`/`BikeDelivery` event *inherits* the properties of `Event`. An event is a subtype of another event if there is a so-called "is-a relationship" between them: A `BikeOrder` *is* an `Event` and a `BikeDelivery` *is* an `Event`.

Being an `Event` means that it has two fields `agent` and `timestamp` next to whatever else you specify. The `Event` type is defined as follows:

```
type Event {
  agent : Agent,
  timestamp : DateTime
}
```

Notice that `Event` does not inherit from any other events. `Event` is the *root* of the *event hierarchy*.

1.2.3 Specifying the desired behaviour

To specify the basic structure of our sales contract, we define a contract template named `BikeSale` with the `template` keyword. We also mark the template as an `entrypoint` to allow us to instantiate a contract from this template – see *the next chapter*. For now, we just encode the order in which the events must occur: One `BikeOrder` event followed by one `BikeDelivery` event.

```
template entrypoint BikeSale0() =
  <*> BikeOrder // Someone sends a BikeOrder
  then
  <*> BikeDelivery // Someone sends a BikeDelivery
  // The contract is done
```

The syntax used here needs a bit of elaboration. A *prefix contract* is a contract that requires some given event to occur. Prefixes are written using the syntax `<*> Event`. The initial `<*>` means that the event may come from anyone, and the following `Event` is the actual type of the expected event. Combining a prefix contract with another contract with `then` results in a contract that requires the specified event to occur, followed by whatever the other contract requires.

Clearly, this contract is too generic. The only thing we have specified so far is the *order* of events and the fact that the contract is finished once the order and delivery has taken place. It therefore allows many sequences of events that should not be accepted: it is for instance quite possible for Bob to send a `BikeOrder` event and for Alice (or even Bob himself) to send the `BikeDelivery` event. To address this, we write the agents next to the events. In principle, we want the `BikeOrder` to come from Alice, and `BikeDelivery` to come from Bob – but this can be naturally generalised to any buyer and seller, respectively. In fact, in CSL we do not hardcode names of agents in contract templates. Instead, we make the buyer and the seller abstract in the contract, which is quite easily done by adding two parameters, `buyer` and `seller` to the contract template:

```
template entrypoint BikeSale1(buyer, seller) =
  <buyer> BikeOrder // Buyer places a bike order
  then
  <seller> BikeDelivery // Seller delivers a bicycle
  // The contract is done
```

These parameters are bound to concrete values once the contract template is *instantiated*.

We have now used a different format for the prefix construct. If one writes `<buyer>` instead of `<*>` it means that the event must originate from an agent given by the template argument `buyer`. Hence, our contract now specifies that it is the buyer that must issue the `BikeOrder` event which must be followed by a `BikeDelivery` event from the seller. This is still too general: Our informal contract specification states that Alice places an order to Bob worth 100 euros and that Bob delivers a bicycle to Alice. Since we declared the `BikeOrder` and `BikeDelivery` events to include information about price, receiver, etc., we may use these properties to place restrictions on allowed events in our contract:

```
template entrypoint BikeSale2(buyer, seller) =
  // Buyer sends an order worth 100 euros to seller
  <buyer> order: BikeOrder where
      order.amount = 100 &&
      order.recipient = seller
  then
  // Seller delivers a bicycle to buyer
  <seller> delivery: BikeDelivery where
      delivery.recipient = buyer
```

The prefix format is extended here to associate a name for a particular received event, making it possible to formulate a predicate on it. This predicate follows the keyword `where`, and the event is only accepted if it evaluates to `True`. The `BikeOrder` event is bound to a variable named `order`, and we use the `where` clause to check that the fields of `order` have the desired values, using the connective `&&` which is the logical "and" operator. In this case, the event `order` is accepted only if the field `order.price` is set to `100` and the field `order.recipient` is set to `seller`. Likewise, the `BikeDelivery` event is bound to the variable `delivery` and is only accepted if `delivery.recipient` is `buyer`.

Generalizing the contract

Our contract template can now be instantiated to a specification of what we described at the beginning of this section: The sale of a bicycle for 100 euros from Bob to Alice – by setting the `buyer` parameter to `alice` and the `seller` parameter to `bob`. The contract is still quite limited, though, as it only models this particular idealized sale of one single bicycle for a fixed price. In this section we will demonstrate how to generalize the contract to handle a wider range of scenarios.

We currently have a contract template that specifies the sale of one bicycle from some `seller` to some `buyer`. Now, what if it was not a bicycle but a hammer that was the transaction's object? Clearly, there is no substantial difference between a sales transaction that involves one object or the other, safe for the price. However, our way of modeling a transaction with `BikeOrder` and `BikeDelivery` events fixates the objects to be a bicycle. We must therefore extend our events a bit to be able to account for orders and deliveries of other items:

```
type Order : Event {
  amount : Int, // The amount of euros is offered for the item.
  recipient : Agent, // The recipient of the order.
  item : String // The item that is ordered
}
type Delivery : Event {
  recipient : Agent, // The recipient of the item
  item : String // The item that is delivered
}
```

Now we can formulate a more general sales contract template (call it `Sale`), by abstracting the item and the price out as parameters of the contract template:

```
template entrypoint Sale0(buyer, seller, amount, item) =
  // Some buyer orders an item for some price from a seller
  <buyer> order: Order where
      order.amount = amount &&
      order.recipient = seller &&
      order.item = item
  then
  // The seller delivers that item
  <seller> delivery: Delivery where
      delivery.item = item &&
      delivery.recipient = buyer
```

Exercise: Pay on delivery

Write a variant of the `Sale0` contract where the payment comes after the delivery *(solution)*.

There is glaring problem with our new contract specification: Nowhere is it specified that certain items have certain prices! According to the contract in its current formulation, a buyer just has to offer *any* amount of euros for any item, including even a negative amount or zero, and she will receive it! This is not a desirable state of affairs: we must model some basic notion of an inventory that associates prices to items.

```
// Accept offer if the item and offered price fit.
val acceptOffer =
  \ "Bike" -> (\p -> p >= 100)
  | "Hammer" -> (\p -> p >= 30)
  | "Saw" -> (\p -> p >= 40)
  | _ -> \p -> False // Seller does not have this item; reject.

template entrypoint Sale1(buyer, seller, amount, item) =
  // Some buyer orders an item for some price from a seller
  <buyer> order: Order where
      order.amount = amount &&
      order.recipient = seller &&
```

(continues on next page)

(continued from previous page)

```
      order.item = item
  then
  // The seller delivers that item
  <seller> delivery: Delivery where
      acceptOffer order.item order.amount &&
      delivery.item = order.item &&
      delivery.recipient = buyer
```

This version of the contract template introduces a new concept of CSL: functions. The basic idea with this addition to our contract is to specify an "inventory function" `acceptOffer` that takes an item and a price and returns either `True` or `False` depending on whether the offered price for the item is acceptable. The notation used here deserves some elaboration: A function has the form `\arg -> body` where `arg` is a name that can be used inside of `body` as the value that was given when the function was called. In CSL, functions always take one single argument. How do we handle the case where we need more than one argument to a function then, like adding two numbers? We use a technique called *currying*, where we write our function such that it return *another function* that takes the remaining argument: `\arg1 -> \arg2 -> body`. Functions may have separate bodies for separate kinds of input. This is written in CSL as `\form1 -> body1 | form2 -> body2`. We call this *pattern matching*, and this is what we have used in the `acceptOffer` above where each body is a new function that takes the price `p` and checks that is above some threshold. In fact, the simple form `\p -> body` also uses pattern matching: the name `p` is a pattern that matches any value and binds it to the name `p` inside of `body` where it can then be used. Note that when pattern matching, the symbol _ is a special pattern that behaves like a name but does not result in a new binding in the function's body, so `\_ -> body` is like above except that `body` cannot refer to the value that the function was called with. Functions are an important part of the CSL language, and they are discussed in much more detail in the *detailed description* in the *value language reference*.

Our new version of the contract allows three items to be sold at three different minimum prices (the buyer is free to offer more), and it protects the seller from accidentally selling items at an unacceptable discount. However, the contract is once again restricted to a certain set of items! What we *really* should do here is to abstract away the shop's inventory.

First, how will an arbitrary shop's inventory be represented? We can either create a separate checking function for every shop, or use some data structure to represent any shop's inventory, and provide a function which, given an inventory, a name of an item, and the price that the `buyer` is offering, either accepts or rejects the offer. Here we show the latter of the two approaches.

What data structure should we use to represent the shop's inventory? It can simply be a *dictionary*: a list of tuples containing the name of the item and a minimal acceptable price. For example, the bike shop's inventory can be represented as:

```
val bikeShopInventory =
  [("Bike", 100),
   ("Brakes", 20),
   ("Helmet", 30)]
```

Next, we need to write a function which decides if an offer can be accepted. It should search the inventory for the given item and check that the price is acceptable. This is one possibility of an implementation:

```
val checkOffer = \inventory -> \(item  : String) -> \(price : Int) ->
  // If the item is listed in the inventory and the price is right,
  // then accept the offer
  // 'List::any' returns True if there is an element of the list
  // that satisfies the predicate
  List::any
    (\(name, acceptPrice) -> name = item  && price >= acceptPrice)
    inventory
```

We use a standard library function *List::any*, which traverses a list, looking for an element which satisfies the given predicate. It returns `True` if it finds one and `False` otherwise. In our case, the list in question is the inventory, and the predicate states that the item is in the inventory, and the proposed price is accepted. Notice that we need to explicitly say that the `item` argument is of `String` type and that `price` is an `Int`. This is because not all

types in CSL are comparable using = or <= *operators*.

Now we can finish writing the `Sale` template:

```
// The 'inventory' parameter is a list of items available in
// the store, together with their prices.
template entrypoint Sale2(buyer, seller, amount, item, inventory) =
  // Some buyer orders an item for some price from a seller
  <buyer> order: Order where
      order.amount = amount &&
      order.recipient = seller &&
      order.item = item
  then
  // The seller delivers that item
  <seller> delivery: Delivery where
      // Use the provided inventory to determine whether
      // the seller has the item.
      checkOffer inventory order.item order.amount &&
      delivery.item = order.item &&
      delivery.recipient = buyer
```

With this formulation of the contract template, we could instantiate it to be bikeshop-specific, bookshop-specific, etc., by defining different inventories for each shop:

```
// Items in a bike shop are bicycles or other gear.
val bikeShopInventory =
  [ ("Bike", 100),
    ("Brakes", 20),
    ("Helmet",30)] // It's a small bike shop.

// Items in our book shop are just the authors.
val bookShopInventory =
  [ ("Joyce", 10),
    ("Proust", 2),
    ("Hugo", 13)] // We just sell three books

// Items in this shop are the allowed ingredients in brunch.
val brunchShopInventory =
  [ ("Pancake", 1),
    ("Bacon", 2),
    ("Egg", 1)] // Nothing else goes in a brunch.
```

Providing fail-safe mechanisms

In CSL, sending an unexpected event to a contract does not cause this contract to *fail*. Instead, the contract simply waits for another event to arrive. This means that if the seller sends a `Delivery` event containing a different item than the one ordered, such an event is simply ignored by the contract, and the seller can try again.

Sometimes we want to react to a situation where something wrong happened, and have the contract *fail*, as it has been breached. For instance, if we ordered a bike and paid enough for it, we expect the seller to not try sending us a hammer instead. To account for the possibility of a contract failing, we use the contract combinator `or` and the primitive contract `failure`:

```
template entrypoint Sale3(buyer, seller, amount, item, inventory) =
  // Some buyer orders an item for some price from a seller
  <buyer> order: Order where
      order.amount = amount &&
      order.recipient = seller &&
      order.item = item
  then (
    // The seller delivers that item
```

(continues on next page)

(continued from previous page)

```
    <seller> delivery: Delivery where
        checkOffer inventory order.item order.amount &&
        delivery.item = order.item &&
        delivery.recipient = buyer
  or
    // The seller tries to cheat
    <seller> delivery: Delivery where
        // The order was accepted
        checkOffer inventory order.item order.amount &&
        // But something isn't right with the delivery
        (not (delivery.item = order.item) ||
         not (delivery.recipient = buyer))
    then failure
  )
```

We only want the contract to fail if the seller tries to deliver the wrong item, or deliver it to the wrong person. To express those two possibilities in the `where` clause, we use the connective `||`, which is the logical "or" operator, and the negation operator `not`. Combining two different events with an `or` combinator gives us two possibilities of how the contract can evolve. In both cases we wait for a `Delivery` event, we also expect that the order was accepted (`inventoryFunction` returning `true`). Whether the contract results in `failure` or not, depends then on the correctness of the delivery details.

Exercise: Pay or return

Continue the *pay on delivery* exercise by allowing a return of unwanted items.

Create an `inventoryPrice` function, which, for each item, returns its price. For simplification, you can assume that for items which are not explicitly listed, the price is some arbitrary large number. The seller should now include the price of the item in the `Delivery` event, based on what the `inventoryPrice` function returns. The buyer then has two options: either pay what is asked of them, or return the item to the seller *(solution)*.

Exercise: You need a bike *and* a helmet!

Write a contract in which the buyer orders both `"Bike"` and `"Helmet"`, in any order. You can write it using only `or` and `then`, but it is easier if you use the *and* combinator instead *(solution)*.

Adding time constraints

By now we have reached a fairly advanced contract, describing in an accurate way a number of sales-related scenarios. There is one serious omission in the contract formulation which we must address: The concept of *time* does not occur in the contract at all and therefore does not influence which events are allowed or not. Currently, it is perfectly legal behaviour for, e.g., Alice to place an order and then for Bob to wait ten years before delivering the ordered item.

What needs to be done is clear in light of the above sections. All events already have a `timestamp` field, simply by virtue of being an `Event`. We will use this timestamp to specify a maximum time period between when an order is placed and when the delivery takes place, i.e., the difference in the `timestamp` fields on the `Order` and `Delivery` events must not exceed some amount of days:

```
template entrypoint Sale(buyer, seller, amount, item, inventory, maxDays) =
  // Some buyer orders an item for some price from a seller
  <buyer> order: Order where
    order.amount = amount &&
    order.recipient = seller &&
```

(continues on next page)

(continued from previous page)

```
6     order.item = item
7   then (
8     // The seller delivers that item
9     <seller> delivery: Delivery where
10      checkOffer inventory order.item order.amount &&
11      delivery.item = order.item &&
12      delivery.recipient = buyer &&
13      // 'DateTime::addDays t d' creates a new timestamp that
14      // is 'd' days after timestamp 't'.
15      delivery.timestamp <= DateTime::addDays order.timestamp maxDays
16    or
17    // The seller tries to cheat
18    <seller> delivery: Delivery where
19        checkOffer inventory order.item order.amount &&
20        (not (delivery.item = order.item) ||
21         not (delivery.recipient = buyer))
22    then failure
23  )
```

The new clause that has been added on line 15 states that the `Delivery` must occur before `maxDays` has passed since the `Order` event occurred. We achieve this with a simple comparison of timestamps: The timestamp on the received `Delivery` event must be no later than a timestamp that is exactly `maxDays` in the future from the time of the `Order` event. The function *DateTime::addDays* takes a timestamp and an integer denoting a number of days and creates a new timestamp that is exactly the specified number of days removed from the input timestamp. This is another example of usage of a function from the *standard library*.

Exercise: Late payment

Extend the *pay on delivery* exercise again, this time by giving a time limit of 3 days to pay for the delivered item. In case the payment is not received in time, the price increases by a `fine`, which is as a parameter to the contract template *(solution)*.

1.3 Contract life cycle

After having formulated a contract template, it is time to "instantiate" it. *Instantiating* a contract template corresponds to *running* a computer program, where the contract specification corresponds to the program executable (or the source code). It is making operational the relation between events that is *accepted* by a contract and the contract specification.

A *residual contract* denotes what is remaining of an instantiated contract, where "remaining" means which events the contract still accepts. If a contract specifies that some event is currently expected and that event is applied, the contract is *evolved* to the residual contract that represents the remaining expected events after the occurrence of the just given event. If a contract no longer accepts any events, i.e. it is reduced to `success` or `failure`, it is said to be *complete*.

Instantiated contracts are stored in a *contract store*, which for now can be considered a container for keeping instantiated contracts and the events that have been applied to them. The contract store provides an interface for submitting events to the contract and for querying its state and all past events that have been accepted by it so far.

The life cycle of a contract can roughly be summarized as follows:

1. Instantiate the contract template with required arguments. The template must be marked with the modifier `entrypoint` as discussed *here*.
2. Submit an event to the instantiated contract.
3. If the event applies to the contract, compute the residual contract after applying the submitted event.
4. Repeat from *(2)* until the residual contract is *complete*.

These steps are described in more detail below.

1.3.1 Instantiating contracts

When you *instantiate* a contract template, an artifact is put into the contract store that represents the contract after zero events have been applied. The precise nature of this "artifact" is not important (and is also dependent on the underlying contract store technology), it is just a piece of data that represents a running contract, much like a process in an operating system represents a running computer program. After a contract has been instantiated, a new entity exists in the contract store. We may instantiate contract templates as often as we want -– a desirable property as we can describe different contracts using the agreed common underlying structure, like reusing a general sales contract for different sales.

Deon Digital maintains a prototype GUI that allows users to interact with a contract storage platform by writing and instantiating contract specifications as well as submitting events to instantiated contracts.

Templates that can be instantiated are marked with `entrypoint` in the CSL code. This modifier roughly means that the template is "public" and that some additional restrictions on its input types must be fulfilled. This ensures that it's always possible to construct values of the requested types from the client code that attempts to instantiate the template.

1.3.2 Submitting events to contracts

Two things happen when you submit an event to a contract:

- First, the system tests whether the event applies to the contract *in its current state*. This means that the issuing agent and the event fields must match the predicates given in the contract. That is, if a contract expects an event from agent `alice`, any event from agent `bob` will be rejected.
- Secondly, if the event can be accepted by the contract in its current state, the event is applied to the contract, *evolving* it to a *residual* version.

We illustrate this using the example contract developed in the *preceding section*, whose definitions we repeat here:

```
type Order : Event {
  amount : Int, // The amount of euros is offered for the item.
  recipient : Agent, // The recipient of the order.
  item : String // The item that is ordered
}
type Delivery : Event {
  recipient : Agent, // The recipient of the item
  item : String // The item that is delivered
}

val bikeShopInventory =
  [("Bike", 100),
   ("Brakes", 20),
   ("Helmet", 30)] // It's a small bike shop.

val acceptOffer = \inventory -> \(item  : String) -> \(price : Int) ->
  // If the item is listed in the inventory and the price is right,
  // then accept the offer
  List::any
    (\(name, acceptPrice) -> name = item  && price >= acceptPrice)
    inventory

template Sale(buyer, seller, amount, item, inventory, maxDays) =
  // Some buyer orders an item for some price from a seller
  <buyer> order: Order where
    order.amount = amount &&
    order.recipient = seller &&
    order.item = item
```

(continues on next page)

(continued from previous page)

```
  then (
    // The seller delivers that item
    <seller> delivery: Delivery where
      acceptOffer inventory order.item order.amount &&
      delivery.item = order.item &&
      delivery.recipient = buyer &&
      // 'DateTime::addDays t d' creates a new timestamp that
      // is 'd' days after timestamp 't'.
      delivery.timestamp <= DateTime::addDays order.timestamp maxDays
    or
    // The seller tries to cheat
    <seller> delivery: Delivery where
      acceptOffer inventory order.item order.amount &&
      (not (delivery.item = order.item) ||
      not (delivery.recipient = buyer))
    then failure
  )

template entrypoint BikeSale(buyer, seller) =
  Sale(buyer, seller, 100, "Bike", bikeShopInventory, 3)
```

In the last line, we instantiated our `Sale` contract to specify the sale of one bicycle for 100 euros from the seller to the buyer:

```
Sale(buyer, seller, 100, "Bike", bikeShopInventory, 3)
```

Notice that we have passed the `bikeShopInventory` list as the inventory and that we set the maximum delay between the delivery and the order to three days. The `buyer` and `seller` parameters remain abstract in the new `BikeSale` template.

The instantiated contract now corresponds to the following snippet:

```
<buyer> order: Order where
  order.amount = 100 && // 'amount' is set to 100
  order.recipient = seller &&
  order.item = "Bike" // 'item' is set to "Bike"
then (
  <seller> delivery: Delivery where
    // 'inventory' is set to 'bikeShopInventory'
    acceptOffer bikeShopInventory order.item order.amount &&
    delivery.item = order.item &&
    delivery.recipient = buyer &&
    // 'maxDays' is set to 3
    delivery.timestamp <= DateTime::addDays order.timestamp 3
  or
  <seller> delivery: Delivery where
     acceptOffer bikeShopInventory order.item order.amount &&
     (not (delivery.item = order.item) ||
      not (delivery.recipient = buyer))
  then failure
)
```

The following is an example of an event that would be accepted:

```
Order {
  agent = alice,                        // The 'buyer' is now set to 'alice'
  // This is arbitrary, as there are no constraints on the 'timestamp'
  //  of an 'Order' event in the contract.
  timestamp = #2017-12-24T16:00:00Z#,
  amount = 100,
  recipient = bob,                      // The 'seller' is now set to 'bob'
```

(continues on next page)

(continued from previous page)

```
  item = "Bike"
}
```

Note that the `timestamp` field is not specified in the first part of the contract, so any value would be accepted. The value of the `timestamp` field in the `Order` event does influence the acceptable values of the `timestamp` in the `Delivery`, however, as we have instantiated the contract such that it requires that no more than three days pass between an `Order` and a `Delivery`.

When this event is applied to the contract, it evolves into something equivalent to the following:

```
<bob> delivery: Delivery where // 'seller' is set to Bob
  acceptOffer bikeShopInventory "Bike" 100 &&
  delivery.item = "Bike" &&
  delivery.recipient = alice && // 'buyer' is set to Alice
  // 'maxDays' is set to 3
  delivery.timestamp <= DateTime::addDays #2017-12-24T16:00:00Z# 3
or
<bob> delivery: Delivery where
  acceptOffer bikeShopInventory "Bike" 100 &&
  (not (delivery.item = "Bike") ||
  not (delivery.recipient = alice)
then failure
```

The following is an accepted event for the contract in this state:

```
Delivery {
  agent = bob,
  // <= 3 days after the 'timestamp' in the 'Order' event
  timestamp = #2017-12-25T17:00:00Z#,
  item = "Bike",
  recipient = alice
}
```

After this event has been applied, the contract is evolved to a finishing, successful state:

```
success
```

The contract store now holds the residual contract, the original specification, and the events that were applied to evolve the original instantiated contract to its current state. As this contract no longer accepts any events, it is *complete*.

Exercise: Selling a bike

In this and the next exercise, we assume you are using the provided *GUI* to interact with the system.

First, instantiate the `BikeSale` contract in the `Composer` tab of the web application. Next, check that the `Viewer` tab shows the correct instantiation of the contract. In the `Actions` tab, try applying an event that should not be accepted, for instance offer too little money for the bike. Verify that the contract hasn't changed in the `View` tab. Now apply an event that is expected. Verify that this time the contract has evolved. Continue interacting with the contract to get it to be a `success`.

Can you get stuck in any way?

Exercise: Make a contract stuck

Wrong instantiation parameters may make it impossible for the contract to ever evolve to a `success`. Try changing the `BikeSale` contract in such a way that no chain of events will let it progress. For instance, see what happens if the price offered for the chosen item is too low in the contract instantiation.

1.4 Guideline for using Deon Digital's GUI

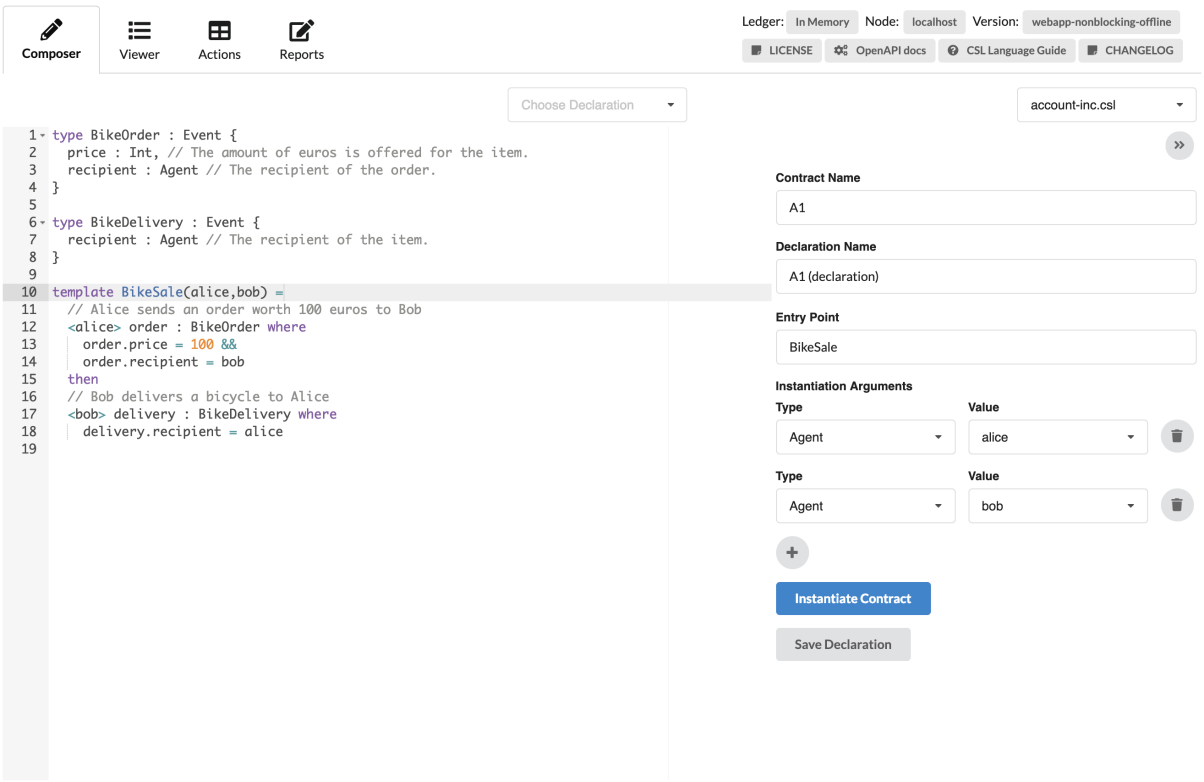
Deon Digital's GUI allows users to write and evaluate contracts by using the contract specification language. A contract written in Deon Digital's CSL is a textual specification of allowed sequences of actions (events) which multiple parties are expected to perform in order to fulfill a contract. The GUI displays the node name and ledger type as well as links to relevant documentation such as our guidelines about the API, the language and the changelog. The GUI exposes three interactions with the Deon application platform:

- Editing and instantiation: *Composer*
- Contract interaction: *Viewer*, *Actions*
- Reporting: *Reports*

In the following sections, we will discuss each of these in detail:

1.4.1 Composer

A contract in Deon Digital's CSL can be specified by using the composer.

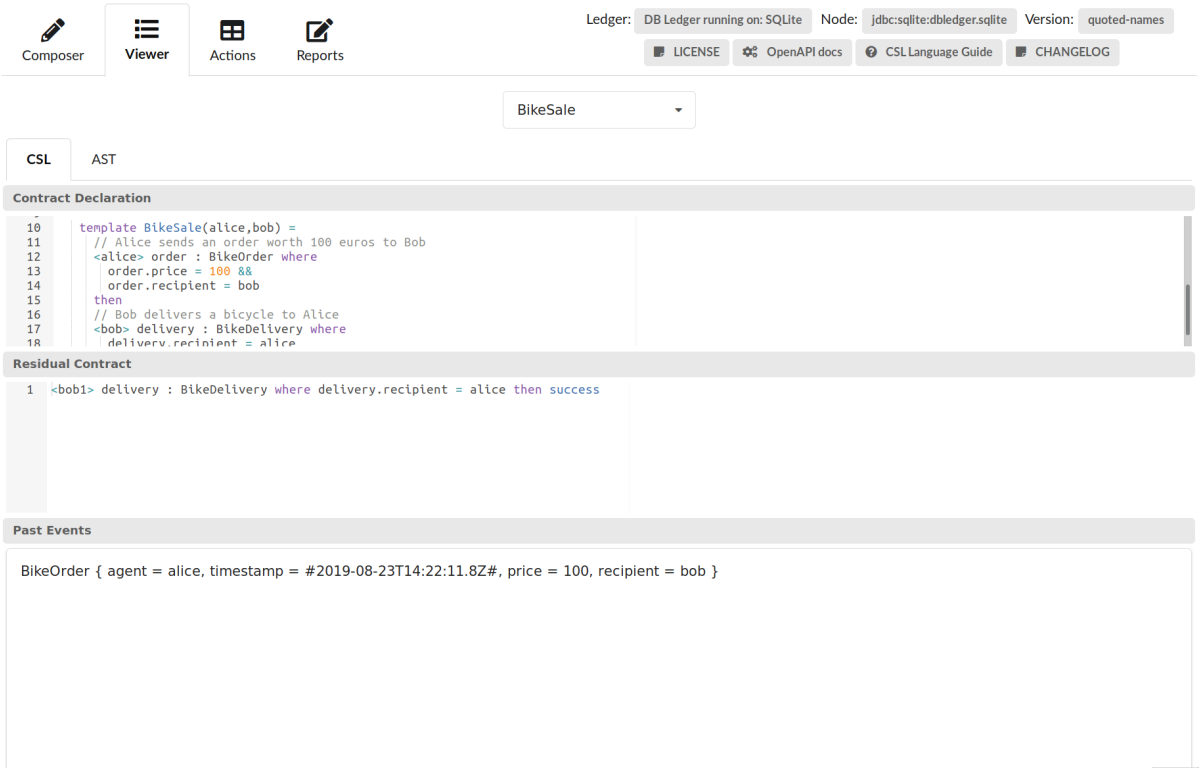


Within the composer, the user has the opportunity to press *"Save Declaration"* or *"Instantiate Contract"*. *"Save Declaration"* will use the "Declaration Name" field. The user can reuse saved declarations later by using the *"Choose Declaration"* dropdown at the top of the page.

"Instantiate Contract" both saves the declaration and instantiates a contract on the ledger using the information from the remaining fields. The "Contract Name" field specifies the name of the contract instance, and can be chosen freely by the user. The peers that are part of the contract network have to be defined as well. The "peers" dropdown is populated by the available peers on the network on which data can be stored and where contracts can be executed. The entry point defines the name of the starting contract. This name must refer to a contract in the CSL and specifies which contract is executed first. If a contract accepts input parameters, they must be specified in the correct order with type and value. Once the contract is instantiated, the data is stored on the ledger which is distributed. The distribution on the network happens automatically.

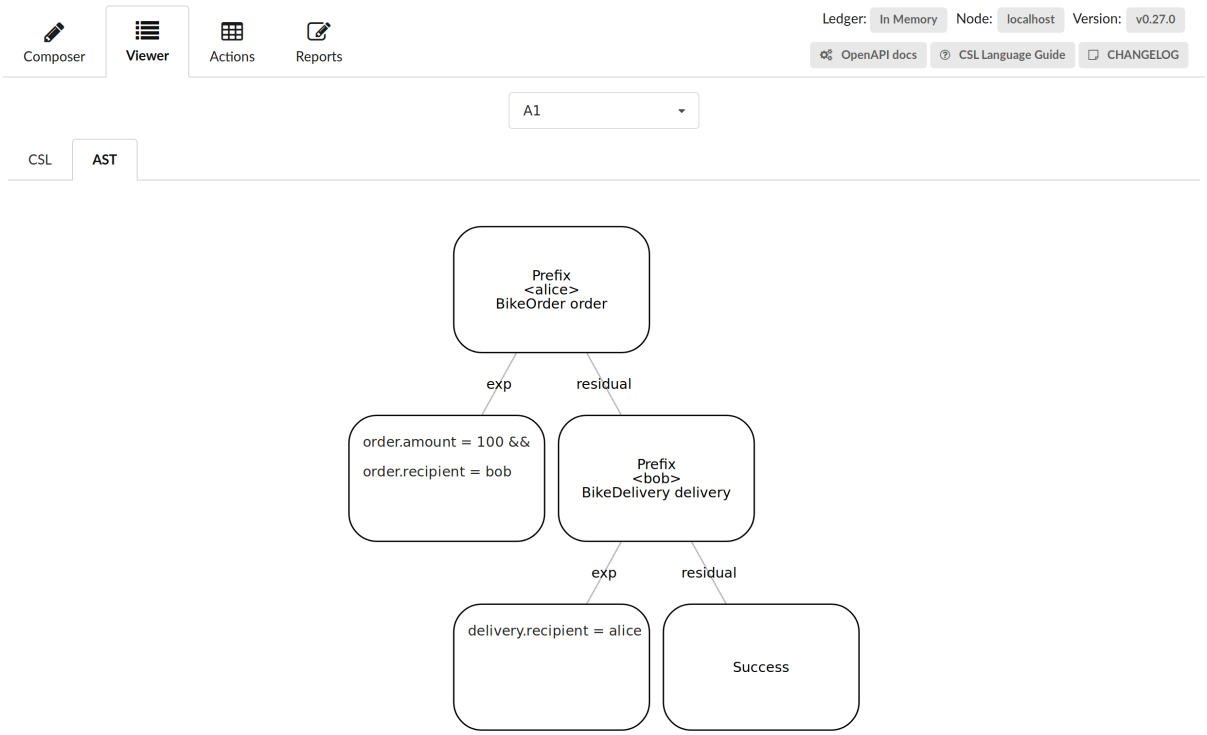
1.4.2 Viewer

The viewer lets the user see the contract specification language (CSL) and the abstract syntax tree (AST) for each instantiated contract. In this example, the contract "Bike Sale" has been choosen:



By selecting the CSL tab, the user can see the individual contract declaration, the *residual contract* which specifies the still remaining events, as well as the list of past events that have successfully been submitted to the contract so far. The user can create events by pressing *"Create Event"* at the bottom of the viewer page. Events can also be applied in the *Actions* tab.

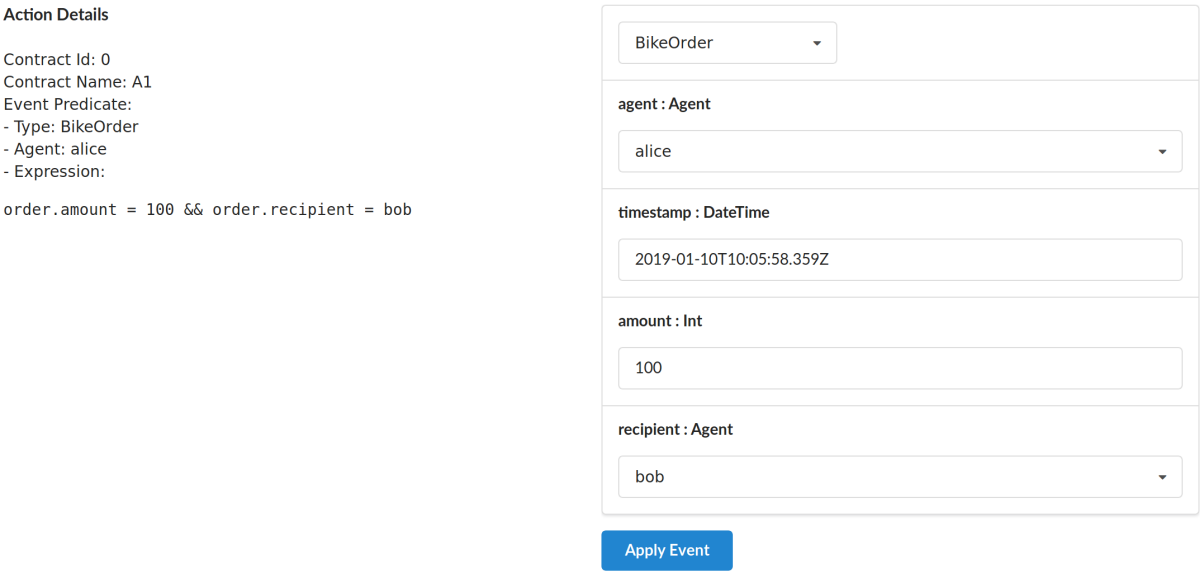
By selecting the AST tab, a graphic visualisation of the structure of the residual contract is provided.



1.4.3 Actions

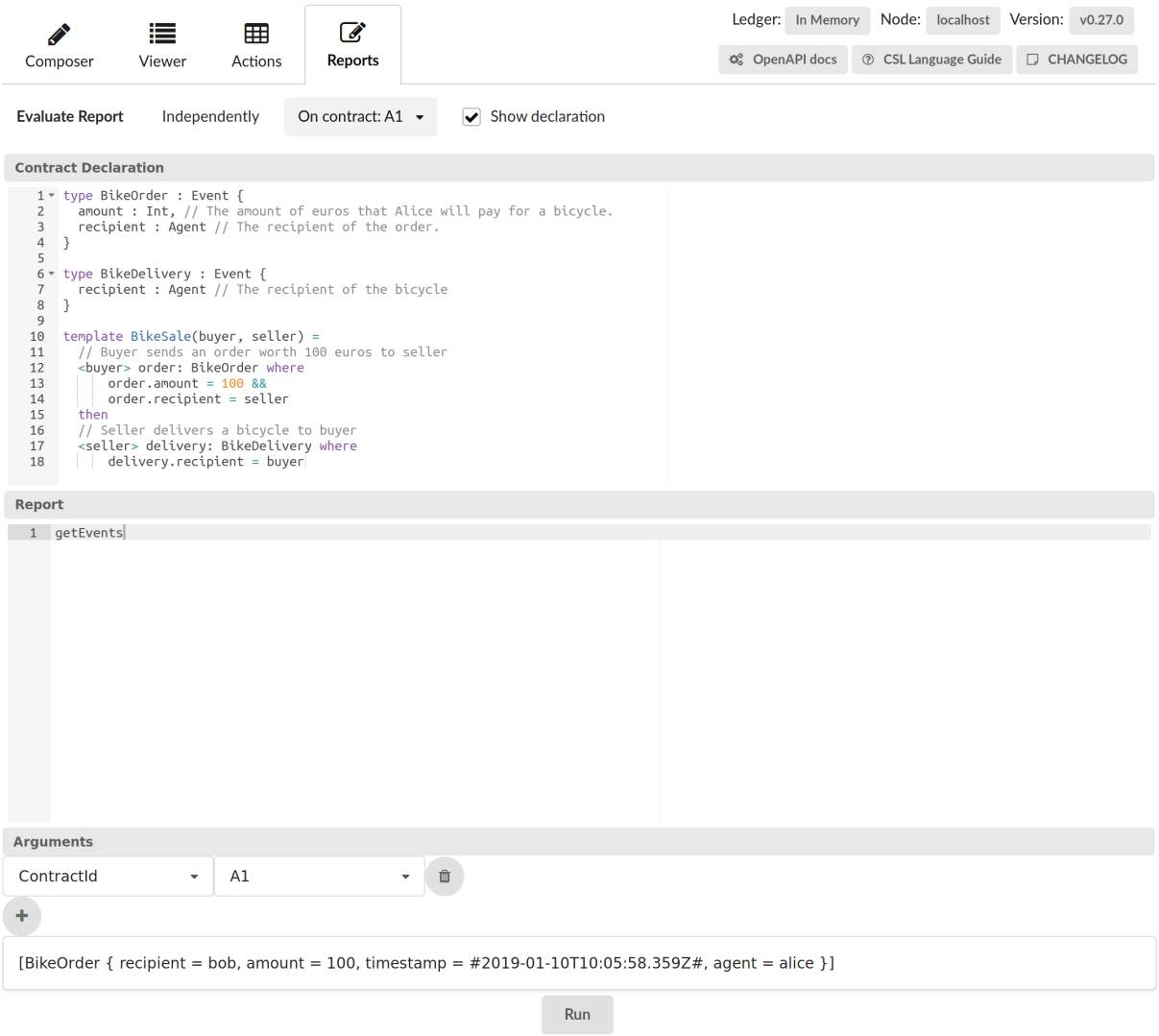
Based on the actual state of the contract, event types can be executed within the actions tab at the given time. The never changing ID (instantiating a contract creates a new ID for every instantiation), the never changing declaration ID (forever attached to an instantiated contract), the name, the event, the agent and the expression are visible for each expected event. Based on the specification of the contract, specific events can be added to each event type.

Id	Declaration ID	Name	Event	Agent	Expression
0	0db28922-30d5-4d0d-a482-77356c130acc	A1	BikeOrder	alice	order.amount = 100 && order.recipient = bob



1.4.4 Reports

The examination of the contracts can be done by using the *value expression language*. Report expressions are either evaluated within the context of a specific contract, or without context. The latter can be done by selecting *"Independently"* in the drop down menu. Reports can be given value arguments with the dropdown menus in the bottom. Pressing *"Run"* causes the system to execute the operation and display the result at the bottom of the page.



More practical examples about how contracts can be specified are mentioned *here*.

CHAPTER 2

Reference

The following chapters are useful as reference once you know the basic concepts:

2.1 Contract language reference

This section presents an overview of the CSL language constructions for the composition of contracts.

The *atomic* contracts of CSL are the smallest building blocks:

- *the successfully completed contract* and
- *the breached contract*.

The non-atomic contracts all specify behaviour relative to smaller sub-contracts. Herein lies the composability of CSL: We formulate general structures that we may re-use across several contracts. There are five ways to compose contracts:

- *expecting the occurrence of an event before another contract*;
- *putting two contracts in sequence*;
- *composing two contracts concurrently*;
- *making a choice between two contracts*; and
- *wrapping contracts in templates* and *recursive templates*.

Note that the code examples in the following sections will sometimes consist of *snippets* of CSL code, not entire CSL specifications that can be copy-pasted and run. We have attempted to add comments in the code snippets to make it clear which parts have been left out, so it should not be too much work to copy and reuse the example snippets.

2.1.1 The successfully completed contract

A successfully completed contract is written as:

```
success
```

Because it is completed, this contract accepts no events.

2.1.2 The breached contract

The failed, or breached, contract is written as:

```
failure
```

A breached contract is in a state of failure and therefore it accepts no events.

2.1.3 The prefixed-contract

A "prefix-contract" is a contract that expects an event satisfying some predicate and after receiving a matching event it becomes a new contract. The general format is:

```
<AGENT> x:EVENTTYPE where PREDICATE
```

The above general form should be read as:

> The agent identified by the value `AGENT` must issue an event of the type `EVENTTYPE`. This event is called `x`, and the predicate `PREDICATE` must evaluate to `True`. If it does, consume the event and evolve to `success`. Otherwise, do nothing.

If the originator of the event is unimportant, we may write `*`, matching any agent. Likewise, we may completely omit the where-clause with the predicate. Doing so corresponds to matching any event with the given type.

Examples

The following example illustrates how one could match certain `Foo` events from `Alice`:

```
<alice> n: Foo where n.timestamp < #2017-08-31T23:00:00Z#
```

This is a specification of a contract that expects an event of type `Foo` from `Alice` where the `timestamp` field is before August 31st at eleven o'clock in the evening. After receipt of such an event, the reduced contract is `success`.

If we instead wanted to match just any event of type `Foo`, with no constraints on either the fields of the event or on the originating agent, we could write the following:

```
<*> Foo
```

This snippet is equivalent to writing

```
<*> Foo where True
```

which is again equivalent to

```
<*> f: Foo where True
```

We may chain together several prefix contracts using the `then` *sequence combinator*. For example, the following contract snippet specifies that first a payment event from `Alice` and then a delivery event from `Bob` must occur, after which no more events may occur:

```
type Payment : Event { receiver: Agent, amount: Int }
type Delivery : Event { receiver: Agent }

// ... definition of Payment and Delivery events ...
template C1(alice,bob) =
  <alice> p: Payment where
    p.receiver = bob &&
```

(continues on next page)

(continued from previous page)

```
    p.timestamp = #2017-08-24T12:00:00Z#
  then
  <bob> d: Delivery where
    d.receiver = alice &&
    d.timestamp < #2017-08-26T12:00:00Z#
```

Now, assume that the agent `Alice` has input the following event:

```
Payment {
  agent = alice,
  timestamp = #2017-08-24T12:00:00Z#,
  receiver = bob
}
```

This event satisfies the first predicate, so our contract accepts it and evolves into its residual:

```
template C2(alice,bob) =
  <bob> d: Delivery where
    d.receiver = alice &&
    d.timestamp < #2017-08-26T12:00:00Z#
```

`Bob` now issues an event:

```
Delivery {
  agent = bob
  timestamp = #2017-08-25T12:00:00Z#,
  receiver = alice
}
```

The second predicate of the original contract (or the first predicate of the residual contract) is satisfied with this event, so the contract evolves into:

```
template C3() = success
```

No more events may be applied; the sequence of events that our contract specified was observed.

2.1.4 Sequentially composed contracts

Given two contracts `c1` and `c2`, their *sequential composition* is the contract:

```
c1 then c2
```

A contract that is a sequential composition of two other contracts specifies that the left-hand contract must be completed before the events of the right-hand contract can be considered.

Precedence of binary operators

The precedence of the three binary contract operators is as follows:

1. `then` binds tighter than
2. `and`, which binds tighter than
3. `or`.

This means that the following

```
a then b or c and d then e
```

is equivalent to

```
(a then b) or (c and (d then e))
```

When you write a contract and need a different structure than the default imposed by the operators' precedence levels you can use parentheses to change it.

Examples

Consider the following example where we define the two contracts `AlicePays` and `BobDelivers`:

```
template AlicePays(alice,bob) =
  <alice> p: Payment where p.receiver = bob

template BobDelivers(alice,bob) =
  <bob> d: Delivery where d.receiver = alice
```

Given these definitions, the following two contracts are equivalent:

```
// Combine two smaller contracts into one.
template AlicePaysThenBobDelivers(alice,bob) =
  AlicePays(alice,bob) then BobDelivers(alice,bob)
// Equivalent formulation:
template AlicePaysThenBobDelivers2(alice,bob) =
  <alice> p: Payment where p.receiver = bob
  then
  <bob> d: Delivery where d.receiver = alice
```

2.1.5 Concurrently composed contracts

The concurrent composition of two contracts `c1` and `c2` is the contract:

```
c1 and c2
```

Composing contracts concurrently yields a contract that accepts all event sequences that are accepted by either the two constituent contracts, where the sequences might be interleaved. For any contract `c`, the contracts `success and c` and `c and success` are equivalent.

The *precedence* level is below `then` and above `or`.

Examples

Say we wanted to express a sales contract where the buyer was free to pay the seller before or after delivery of the goods. The following contract is a reformulation of our sales contract between `Alice` and `Bob` that allows for just that:

```
template SaleWithoutTemporalRestrictions(alice,bob) =
  <alice> p: Payment where p.receiver = bob
  and
  <bob> d: Delivery where d.receiver = alice
```

This contract specifies that `Alice` and `Bob` must issue `Payment` and `Delivery` events, but not that they must occur in any specific order. Both the event sequence "`Payment`-then-`Delivery`" and "`Delivery`-then-`Payment`" are valid according to this contract. After applying the event

```
Payment {
  agent = alice,
  timestamp = #2017-08-24T13:37:00Z#
  receiver = bob
}
```

the residual contract will be

```
success
and
<bob> d: Delivery where d.receiver = alice
```

This contract accepts just the `Delivery` event from `Bob`.

If we had instead applied a `Delivery` event first, the residual contract would have been

```
<alice> p: Payment where p.receiver = bob
and
success
```

Now the contract expects a `Payment` event from `Alice` next.

2.1.6 Choice between contracts

The *choice* between two contracts `c1` and `c2` is the contract:

```
c1 or c2
```

The choice-contract allows composition of two independent contracts into a new contract that expects *exactly* one of the two contracts to be completed. Depending on the shape of the two sub-contracts and the incoming event, one or the other will be picked as the residual contract.

The *precedence* level is below both `then` and `and`.

Examples

Consider a scenario in which we would like `Alice` to pay a sum of money and the sum depends on whether she pays before or after Christmas eve 2018. Each of the sub-scenarios can be modelled with a simple *prefix-contract*, and we can use `or` to combine them in the appropriate way:

```
val deadline0 = #2018-12-24T13:37:00Z#

template PayBeforeOrAfterChristmas0(alice) =
  <alice> p: Payment where p.timestamp <= deadline0 && p.amount = 100
  or
  <alice> p: Payment where p.timestamp > deadline0 && p.amount = 200
```

The contract can evolve in two different ways. Either `Alice` performs a `Payment` before the deadline of `2018-12-24T13:37:00` with an amount of `100`, or `Alice` performs a `Payment` after the deadline and then the amount has to be `200`. Exactly one of these two things has to occur in order for the entire contract to be satisfied. Applying a `Payment` event either before or after the deadline yields the residual contract

```
success
```

This is because the residual contract of the two individual prefix-contracts are both `success`.

Consider instead the following variation on the contract, where either `Santa Claus` or `Bob` sends a `Delivery` event depending on whether the `Payment` came before or after Christmas:

```
val deadline1 = #2018-12-24T13:37:00Z#

template PayBeforeOrAfterChristmas1(alice,bob,santaClaus) =
    <alice> p: Payment where p.timestamp <= deadline1
    then
    <santaClaus> Delivery
  or
```

(continues on next page)

(continued from previous page)

```
    <alice> p: Payment where p.timestamp > deadline1
    then
    <bob> Delivery
```

Applying a `Payment` event before the deadline will now yield the residual contract

```
<santaClaus> Delivery
```

If the `Payment` came after the deadline, the residual would instead be

```
<bob> Delivery
```

The `success` contract denotes a contract that is considered to have been completed in a successful manner. If we combine that with some other contract `c` using `or`, we construct a contract where the events of `c` are *permitted* but not *obligated*. Suppose we construct a contract as follows:

```
template C(alice) = <alice> Payment or success
```

This is a contract where `Alice` *may* perform a `Payment` event if she chooses.

We can extend the permission contract to a contract where there is an optional choice to be made:

```
type Option1 : Event {}
type Option2 : Event {}
type Action : Event {}

template OptionalChoice(alice,bob,charlie) =
    <alice> Option1
    then
    <bob> Action
  or
    <alice> Option2
    then
    <charlie> Action then success
```

In this contract, `Alice` may choose to perform either of the events `Option1` or `Option2`, resulting in either the residual contract where `Bob` can perform an `Action` or the contract where `Charlie` can perform an `Action`. She may also choose not to send any events; the final composition with the `success` contract allows that.

2.1.7 Contract templates

A *contract template* with the name `TemplateName` has the general form:

```
template [c0, ..., cn] [entrypoint] TemplateName (p0, ..., pm) = BODY
```

where `c0`, ..., `cn` are *contract parameters*, `p0`, ..., `pm` are *value parameters*, and the keyword `entrypoint` is optional. In the special case where there are zero contract parameters, we may also use the equivalent shorthand:

```
template [entrypoint] TemplateName (p0, ..., pm) = BODY
```

A contract template, either with or without contract parameters, is a name for a contract that can be re-used. *Using* a contract template means instantiating the contract and value parameters, if any, with contract arguments and value arguments, respectively. This means that we can capture general contract design patterns with contract templates and apply them whenever they are needed.

CSL allows the use of *local* contract templates: Templates that are only usable inside some other contract. This can be handy if we need some contract template that is too specific to be at the top-level but where we would still like the benefits from encapsulating a pattern. The syntax for a local contract template is:

```
// ... inside some contract definition
let template [c0, ..., cn] TemplateName (p0, ..., pm) = TEMPLATEBODY
in
    BODY
```

Inside of the contract body `BODY` we may now use the template `TemplateName`, but it is not visible from the outside nor is it visible inside `TEMPLATEBODY`. Earlier definitions are visible in later ones but not vice versa

Template entrypoints

Templates that you intend to use as "entry points" by instantiating them from the outside using, e.g., the *sic interfaces* must be marked with the `entrypoint` keyword. The adds the following constraints to the template's types:

1. The template's value parameter types must be *monomorphic*– that is, each of the parameters *p0* to *pm* must have a type that does not contain any type variables.
2. The template's value parameter types must not contain any occurence of a function type (e.g., `Int -> Int`).
3. The template cannot take any contract parameters.

Together, these restrictions ensure that it's possible to instantiate the template externally through, e.g., the REST api. A CSL contract may contain multiple entry points. The `entrypoint` keyword is only available on *top-level* templates; local templates cannot be entry points.

Examples

In its most basic form we can write

```
template entrypoint JustSucceed() = success
```

which effectively serves as an alias for the `success` contract. This contract template takes no value parameters. Say we want to express that a `Payment` event from `Alice` for some not-yet-specified amount of money has to occur. There is just one (value) parameter to this contract template, namely the sum of money. We express it as follows:

```
// ... Payment event defined here
template AlicePaysAmount(amount,alice) =
<alice> p: Payment where p.amount = amount
```

If now want to express some sale where `Alice` pays 42 euros and then, for the sake of simplicity, nothing more happens, we write

```
template entrypoint SomeSale(alice) = AlicePaysAmount(42,alice)
```

Note that this is itself a contract template (still with zero contract parameters). In the section about *instantiating* contracts it is explained how to get from "static" descriptions of contracts to something that actually lives and runs somewhere.

We might often need to express that some contract be dependent on a signature from an agent. This is a quite general pattern, so we formulate it as a contract template:

```
type Signature : Event {
 // ... Signature event defined here with appropriate fields
}
template [c] SignAndContinue(agent) =
  <agent> Signature then c
```

This contract template takes one contract parameter, `c`, and one value parameter, `agent`, and it constructs a *prefix-contract* requiring a `Signature` event from `agent` before it evolves into the contract `c`. We might want to refine this pattern by allowing some default contract to happen if the agent can't issue a `Signature`. Our contract template just needs two contract parameters instead of one to achieve this:

```
// ... Signature event defined as above
template [cSigned, cUnsigned] SignOrDefault(agent) =
  (<agent> Signature then cSigned) or cUnsigned
```

Here we use the `or` combinator to *choose* between the contract where the agent signs which continues as `cSigned`, and the contract called `cUnsigned` where the agent does not sign. It could of course be generalized to handle different kinds of signatures, default cases, etc.

Templates themselves can be nested using the `let ... in` construct:

```
type SayHello : Event {}
type SayGoodbye : Event {}
template HelloHelloGoodbyeLocal(agent) =
  let
    // 'agent' says "hello" and the template evolves
    // into the contract given by 't'
    template [t] Hello() = <agent> SayHello then t
    // 'agent' says either "goodbye" and the contract
    // evolves into 't', or she says "hello" and the
    // contract evolves into 't'.
    template [t] Goodbye() =
        (<agent> SayGoodbye then t) or Hello[t]()
  in
    // 'agent' says "hello" twice before saying
    // either "goodbye" or "hello" once and then
    // the contract ends successfully.
    Hello[Hello[Goodbye[success]()]()]()
```

In this example we exploit the fact that the template `Goodbye` can use the template `Hello`. We could write the above without the use of local templates as follows:

```
// ... SayHello and SayGoodbye defined here ...
template HelloHelloGoodbye(agent) =
  <agent> SayHello then
  <agent> SayHello then
  (
    <agent> SayGoodbye
    or
    <agent> SayHello
  )
```

2.1.8 Recursive contract templates

A *recursive contract template* with the name `TemplateName` has the form

```
template rec [entrypoint] TemplateName(p0, p1, ..., pm) = TEMPLATEBODY
```

where `p0`, . . . , `pm` are *value parameters* and the keyword `entrypoint` is optional. Unlike non-recursive contract templates, recursive contract templates are not allowed to have any contract parameters.

Recursive contract templates work like normal contract templates, except that `TEMPLATEBODY` may call itself by calling `TemplateName`. All recursive calls must be *guarded*, which means that at least one event *must* be accepted by `TEMPLATEBODY` before a call to `TemplateName` has to be unfolded to determine what the possible next events of the residual contract is.

If multiple recursive contract templates will need to call each other, the following general form can be used:

```
template rec [entrypoint] TemplateName-0( ... ) = TEMPLATEBODY-0
        with [entrypoint] TemplateName-1( ... ) = TEMPLATEBODY-1
        with ...
        with [entrypoint] TemplateName-n( ... ) = TEMPLATEBODY-n
```

All template names `TemplateName-0`, ..., `TemplateName-n` can be used recursively in all bodies `TEMPLATEBODY-0` ... `TEMPLATEBODY-n`. The same restrictions regarding guardedness apply. Any number of the templates `TemplateName-0` to `TemplateName-n` may be marked as `entrypoint`, subjecting it to the restrictions described *above*.

Recursive contract template examples

Consider the following recursive contract template `PingPong`:

```
type Ping : Event {}
type Pong : Event {}
template rec entrypoint PingPong() =
  <*> ping:Ping then <ping.agent> Pong then PingPong()
  or
  success
```

Instantiating this template yields a contract which accepts the event `Ping` from any agent and then requires the same agent to generate an event `Pong`, after which it goes back to accepting another `Ping` event from another agent by calling `PingPong()`. The contract can be terminated as long as all `Ping` events have been matched by a corresponding `Pong`.

For example, if `Alice` applies `Ping` to `PingPong()`, then the residual contract is

```
<alice> Pong then PingPong()
```

The following example uses mutual recursion to model a house where a specified agent can move inside and outside and, depending on where the agent is, may perform tasks such as watching TV or mowing the lawn.

```
type WatchTv : Event {}
type GoOutside : Event {}
type GoInside : Event {}
type MowLawn : Event {}
template entrypoint House(agent) =
  let template rec Inside() =
        <agent> WatchTv then Inside()
        or <agent> GoOutside then Outside()

      with Outside() =
        <agent> MowLawn then Outside()
        or <agent> GoInside then Inside()
        or success
  in
    Outside()
```

We have defined the mutually recursive templates as local templates inside a non-recursive contract template to ensure that a `House` contract can only be instantiated starting in the `Outside` state. In each of the states `Inside` and `Outside`, the agent may perform any number of `WatchTv` and `MowLawn` events, respectively, but the agent can only go outside if already inside, and vice versa. Note that the contract may only successfully terminate when the agent is outside.

Guardedness

One has to take care that a (mutually) recursive contract template definition is *well-guarded*, as unguarded contracts do not have a well-defined semantics. The CSL system will automatically check if a contract is well-guarded

and reject it with a descriptive error message otherwise.

A recursive template definition is unguarded whenever it is possible for it to call itself without first consuming an event. For example, the following recursive template does nothing but call itself, and so is clearly unguarded:

```
template rec Unguarded() = Unguarded()
```

Unguarded recursive templates are problematic because they can be used to express contracts which has to be unfolded indefinitely before we can determine whether a given event can be applied or not. For example, consider the following unguarded contract template:

```
type Count : Event { n: Int }
```

```
template rec Unguarded(n) =
  <*> p:Count where p.n = n or Unguarded(n+1)
```

The instantiation `Unguarded(0)` can, without applying any events, be unfolded to any of the following:

```
<*> p:Count where p.n = 0 or Unguarded(1)
<*> p:Count where p.n = 0 or p:Count where p.n = 1 or Unguarded(2)
<*> p:Count where p.n = 0 or p:Count where p.n = 1 or
    p:Count where p.n = 2 or Unguarded(3)
...
```

and so on. In this case, if we are given the event `Count { n = 42 }`, then the system will have to figure out that it needs to do 42 unfoldings before it reveals a prefix matching the event. Given the event `Count { n = -42 }`, the system has to analyze the contract to determine that no number of unfoldings can reveal a matching prefix. In general, an unguarded unfolding can generate much more complicated mathematical sequences, and it can be shown that it is theoretically impossible to determine, in general, if there exists a number of unfoldings that reveals a matching prefix or not.

The above example becomes guarded if we change `or` to `then` (and add `success` to allow the contract to terminate):

```
template rec Guarded(n) =
  <*> p:Count where p.n = 1 then Guarded(n + 1)
  or success
```

While we can still unfold the recursive calls indefinitely, we do not *have* to do that in order to determine whether a given event can be accepted: any event must be matched by the prefix that is visible now, so unfolding `Guarded(n+1)` will not reveal any new information.

It is generally easy to spot unguarded contracts simply by checking that all recursive calls occur below a prefix. However, there is one common pitfall that may lead to accidental unguardedness. Consider the following example:

```
template Nullable() = <*> Count or success
template rec Unguarded() = Nullable() then Unguarded()
```

The call to `Unguarded()` is apparently below a `then`, making it appear guarded on cursory inspection. However, since it is possible to terminate `Nullable()` via the right alternative without consuming any events, the call to `Unguarded()` is exposed.

2.1.9 Contract abbreviations

A *contract abbreviation* with the name `abbreviationName` has the general form:

```
contract abbreviationName = ABBREVIATIONBODY
```

This defines an abbreviation for the contract `ABBREVIATIONBODY` and binds it to a contract variable with the name `abbreviationName`. Abbreviations are defined using the `contract` keyword with a lowercase

name. They are distinguished from templates in that it is not possible to specify any contract or value parameters, and furthermore contract abbreviations cannot refer to themselves, which for the above example means that `abbreviationName` may not occur as a contract variable within `ABBREVIATIONBODY`.

CSL allows the definition of *local* abbreviations that are only visible within contracts. The syntax for a local contract abbreviation is

```
// ... inside some contract definition
let
    contract abbreviationName1 = ABBREVIATIONBODY1
    contract abbreviationName2 = ABBREVIATIONBODY2
    ...
    contract abbreviationNameN = ABBREVIATIONBODYN
in
    BODY
```

Inside `BODY` we may use `abbreviationName1`, `abbreviationName2`, ..., `abbreviationNameN` as a contract variables, but they are not visible from the outside. Furthermore, each `ABBREVIATIONBODYi` may use the previously defined abbreviations `abbreviationName1, abbreviationName2, ..., abbreviationName(i-1)`.

Abbreviations can be used to reformat a complex contract to make it more readable, but otherwise they add no expressive power to the language. It is generally preferable to use abbreviations when the full power of contract templates are not needed, since this communicates to the system that the abbreviations are intended to be used as such, and that any self-reference is accidental and should be flagged as an error.

Examples

In the following example, we express a contract where some agent may either borrow a car or buy it, after which the agent is allowed to drive it. If the car was only borrowed and not bought, then it must be returned again when the agent has finished driving it. To avoid repeating the specification of what it means to drive a car, we define that part of the contract using an abbreviation:

```
type BorrowCar : Event {}
type ReturnCar : Event {}
type BuyCar : Event {}

type DriveCar : Event {}
type TurnOnCar : Event {}
type TurnOffCar : Event{}

template entrypoint Drive(agent) =
  let
    contract driveCar =
      <agent> TurnOnCar then
      <agent> DriveCar then
      <agent> TurnOffCar
  in
    (<agent> BorrowCar then
     driveCar then
     <agent> ReturnCar
    )
    or
    (<agent> BuyCar then
     driveCar
     )
```

We could also have defined a local template `DriveCar()` instead, but that would have been unnecessary since the `driveCar` contract never needs to call itself. In this way, readers of our contract can determine just by looking at the definition head of `driveCar` that it is truly just an abbreviation, and that its body cannot refer back to itself.

2.2 Value expression language

The *value expression language* is the part of the CSL language that lets us specify conditional expressions for event predicates and for defining values using the keyword `val`. It is also used by the reporting engine to define the values returned by reports.

Values may be defined at the top-most level, or as local values inside expressions with the *let construct*. It takes the form `val p = e` where `e` is an expression and `p` is a *pattern*. For top-level values the pattern may be exactly one of `n` or `n : T` where `n` is an identifier beginning with a lower-case letter and `T` is a type. Values declared inside a let construct may take one of these forms as well as the more complex patterns described in the sectino *below*. Note that in both cases the type annotation `T` must be *closed*, i.e., it cannot contain any type variables.

2.2.1 Primitive values and value literals

The value expression language has the following built-in primitive types, all of which can be written with *literals*:

- `Int`, e.g. `0`, `1`, `123`.
- `Float`, e.g. `0.0`, `1.23`, `12.46`, `1e6`, `1E10`, `1E-3`, `1.234E3`.
- `String`, e.g. `""`, `"abc"`, `"\""`.
- `DateTime`, e.g. `#1969-07-20T20:18:04Z#`, `#2018-02-02T11:06:08Z#`.
- `Duration`, e.g. `#P1DT2H3M4S#`, `#-PT42.314S#`.

The value expression language has support for tuples of at least 2 elements where each value in the tuple can be of a different type:

- `Tuple`, e.g. `(1, 2)`, `("a", "b", 42)`.

Furthermore, there is a type of *symbols* which are references to ledger entities. The type of a symbol is one of the following *entity types*:

- `Agent` which represents agents that can send events. Agent symbols can be compared for equality, and they compare equal if they refer to the same agent.
- `Contract` which represents an instantiated contract on the ledger. Contract symbols can be compared for equality similarly to agents.
- `PublicKey` which represents a cryptographic public key used for asymmetric encryption. Public keys can be compared for equality and be used for checking signatures on signed data.
- `Signed` which represents a value (whose serialized representation is) signed by some private key. Signed value symbols can be compared for equality, and their signatures can be checked respective to a public key.

Number literals

Integer numbers (`Int`) use signed Two's complement 32-bit integers as their underlying implementation. They are in the range from `-2147483648` to `2147483647`. Calculations with integers wrap around at their bounds: in CSL `2000000000 + 2000000000 = -294967296` is `True`.

Floating-point numbers (`Float`) use decimal128 as their underlying implementation. This representation allows for exact financial calculations. With decimal representation `0.1 + 0.2 = 0.3` as you would expect, not `0.30000000000000004` as one would see with binary64. With decimal128 you get up to 34 digits of precision. `1.0 / 3.0 = 0.3333333333333333333333333333333333`. Arithmetic operations on floating-point numbers round half to even also called Bankers rounding. We have `1.0 + 2.5e-33 = 1.0 + 2e-33` but because we round to the nearest even digit we also have `1.0 + 1.5e-33 = 1.0 + 2e-33`.

`DateTime` literals

`DateTime` literals, however, deserve a bit of elaboration. They are fenced by the `#` symbol, and between these a subset of the ISO 8601 standard for date and time formatting is expected. Specifically, a `DateTime` in CSL takes one of the following forms from the ISO standard:

1. A date and a time in UTC format:

```
// 28th of February 2018, at 13:37 o'clock UTC
val a = #2018-02-28T13:37:00Z#
// Christmas eve 2017, half past six in the evening UTC
val b = #2017-12-24T18:30:00Z#
// Christmas eve 2017, half past six in the evening UTC with high precision
val c = #2017-12-24T18:30:00.000Z# // Up to 3 decimals on the seconds.
```

2. A date and time with an explicit time zone offset:

```
// 28th of February 2018, at 13:37 o'clock (in UTC+1)
val a = #2018-02-28T13:37:00+01:00#
// Christmas eve 2017, half past six in the evening (in timezone UTC-9)
val b = #2017-12-24T18:30:00-09:00#
// Same, with high precision
val c = #2017-12-24T18:30:00.000-09:00#
```

`DateTime`s specified in different time zones can be compared:

```
// 13:37 in UTC+1 is 12:37 in UTC
val a = #2018-02-28T13:37:00+01:00# = #2018-02-28T12:37:00Z#
// = True
```

To allow for more readable contracts, CSL supports a shorthand syntax for writing `DateTime` literals where parts may be omitted. The omitted parts are treated as either `01` for months and days or `00` for hours, minutes, and seconds. The time zone designator may be appended to any literal in one of the shorthand forms. When it is not present, the time zone is UTC. Therefore, the following `DateTime` literals all denote the same point in time:

```
// We may forgo writing the time zone, in which case the time is in UTC.
val datetime0 = #2018#
val datetime1 = #2018Z#
val datetime2 = #2018+00:00#
val datetime3 = #2018+0000#
// The following could also be written with a UTC time zone designator.
val datetime4 = #2018-01#
val datetime5 = #2018-01-01#
val datetime6 = #2018-01-01T00#
val datetime7 = #2018-01-01T00:00#
val datetime8 = #2018-01-01T00:00:00#
// Between 0 and 3 decimals after the decimal point.
val datetime9 = #2018-01-01T00:00:00.#
val datetime10 = #2018-01-01T00:00:00.0#   // Etc.
val datetime11 = #2018-01-01T00:00:00.000#
```

`Duration` literals

Like `DateTime`, the syntax for durations follows the ISO 8601 standard. A Duration is written between two `#` symbols and contains an optional sign, a `P`, a date component and/or a time component.

- The date component designates days and consists of a number with up to three fractional decimals followed by a `D`.
- The time component is prefixed with a `T` and is comprised of three sub-components. Each component is optional but the order must be adhered to and at least one must be present.

- The hour component is a number with up to three fractional decimals followed by an `H`
- The minute component is a number with up to three fractional decimals followed by an `M`
- The seconds component is a number with up to three fractional decimals followed by an `S`

These are all valid duration literals:

```
val duration0 = #P1D#                // 1 day
val duration1 = #PT1.2S#             // 1.2 seconds
val duration2 = #P1DT2.5H3.001S#     // 1 day, 2 hours, 30 minutes, and 3.001␣
↪seconds
val duration3 = #P1DT2.5H2M3.001S#   // 1 day, 2 hours, 32 minutes, and 3.001␣
↪seconds
val duration4 = #-P1DT2.5H3.001S#    // negative (1 day, 2 hours, 30 minutes, and 3.
↪001 seconds)
val duration5 = #PT62M0S#            // 62 minutes
```

Note that at least one component must be present, so `#P#` is not a valid duration. It's easy to forget the prefixed `T` of the time component, e.g. #P1S" is not valid since the leading `T` of the time component is missing. A trailing `T` is also not allowed as in `P1DT`.

Note that durations are compared down to the millisecond and thus `#PT62M# = #PT1H2M#`.

`Signed` data

A `Signed a` value refers to a message of type `a` that was signed by some private key. You can check if a `Signed a` value was signed by a private key corresponding to a specific public key, and you can also extract the message from the `Signed a` value.

```
type Answer : Event { answer: Signed String }
template WaitForAnswer(deepThought: PublicKey) =
    <*> a:Answer where
        Signed::checkSignature deepThought a.answer
        && Signed::message a.answer = "42"
```

2.2.2 Operators

There are a number of arithmetic, comparison, and logical operators defined:

- *Arithmetic operators* (+, −, ⋆, and /):

  Work on `Int` and `Float`. Note that we don't have implicit conversion of numeric types. That means `2.0 * 4` will be rejected by the type checker.

- *Equate* (=):

  This operator compares values for equality. See the section *below* for a more detailed description

- *Comparison operators* (<, >, <=, and >=):

  Works on `Int`, `Float`, and `DateTime`.

- *Logical operators* ("and" `&&` and "or" `||`):

  Combine `Bool` values.

- *Unary minus* (−):

  Negates an `Int` or `Float` value.

Note that the only valid operation on `Agent` is equality.

The usual precedence rules between the operators are in effect. Thus, for the arithmetic operators the multiplication/division operators have higher precedence than the addition/subtraction operators, and for the logical

operators the "and" operator (`&&`) binds tighter than the "or" (`||`) operator. We may use parentheses to enforce a different order of operators.

```
val a = (1 + 2) * 4 // = 12
val b = 1 + 2 * 4 // = 9
val c = 1.5 * 3.0 // = 4.5
// Wrong!  Int and Float mixed up!
// val d = 1 - 1.0
val e = 4 <= 4 // True
val f = #1969-07-20T20:18:04Z# < #2018-02-02T11:06:08Z# // True
val g = 4 = 3 // = False
val h = "stringA" = "stringA" // = True
val i = #1969-07-20T20:18:04Z# = #2018-02-02T11:06:08Z# // False
val j = True && True // = True
val k = True && False // = False
val l = (True || False) && (False || False) // = False
val m = - 1 * 3 // = - 3
```

2.2.3 Constructors and sum types

CSL lets you define *sum types*. Values of sum types can be constructed in only those ways that are specified in the declaration of the sum type. This is very handy when you have to model something that can only take values of certain forms.

We declare sum types with the `type` keyword followed by the name of the new type and a list of *constructors*:

```
type YesOrNo
  | Yes
  | No
```

The sum type `YesOrNo` has two constructors, `Yes` and `No`. Both constructors take zero parameters, but in general constructors of sum types can take parameters. For example, a type called `IntFloatOrNothing` with three constructors that takes one `Int`, one `Float`, or nothing as its parameter, respectively, is declared as:

```
type IntFloatOrNothing
  | AnInt Int
  | AFloat Float
  | NoneOfTheAbove
```

Values of our new type `IntFloatOrNothing` can be constructed in three ways since there are three constructors:

```
val anInt = AnInt 43 // 'anInt' has the type IntFloatOrNothing
val aFloat = AFloat 42.4 // 'aFloat' has the type IntFloatOrNothing
val none = NoneOfTheAbove // 'none' has the type IntFloatOrNothing
```

Sum types can be declared with *type parameters* by writing them after the type name. These act as placeholders for concrete types, and they can be used in the constructor declarations as types for their parameters. The following type declares values that are either of one type, `a`, or of some other type, `b`:

```
type OneOrTheOther a b
  | One a
  | TheOther b
```

This type takes two type parameters `a` and `b`. We can therefore make many different kinds of `OneOrTheOthers` depending on how we instantiate the `a` and `b`:

```
val oneFortyTwo = One 42 // This is a 'OneOrTheOther Int b'
val y = TheOther "Hello" // This is a 'OneOrTheOther a String'
val z = One #2018-02-02T11:06:08Z# // This is a 'OneOrTheOther DateTime b'
val h = TheOther 42.42 // This is a 'OneOrTheOther a Float'
```

Notice that `x`, `y`, `z`, and `h` above do not have complete types yet: The constructors only restrict one of the two parameters to be `Int`, `String`, `DateTime`, or `Float`. Because they do not place conflicting restrictions on the parameters, we may use both `x` and `y` anywhere that expects a `OneOrTheOther Int String`. Likewise we can use both `z` and `h` anywhere that expects a `OneOrTheOther DateTime Float`, and we can use `x` and `h` anywhere we need a `OneOrTheOther Int Float` and so on.

Examples

`Bool` is a sum type with two constructors that each take zero parameters. The `Bool`-type could be defined as follows:

```
type Bool
  | True
  | False
```

That is, `Bool` has zero type parameters and two constructors called `True` and `False`, each taking zero parameters. We can therefore construct an instance of a `Bool` in two ways, just as one would expect from a boolean data type:

```
val trueValue = True
val falseValue = False
```

Similarly, the type of lists with elements of some type `a`, `List a`, can be defined as a sum type as follows:

```
type List a
  | Nil
  | Cons a (List a)
```

A list is either the empty list `Nil` or the list `Cons x l` with one element `x` followed by the the list `l`. The actual type of elements in the list is not important to the list structure itself, as long as all elements in the list are of the same type. This is enforced by the `Cons` constructor: To make a `Cons` value one must provide a value of type `a` as the first parameter and a value of type `List a` as the second. For example, creating a list with numbers or strings is done by writing:

```
val oneTwoThree = Cons 1 (Cons 2 (Cons 3 Nil))
val strings = Cons "First string" (Cons "Second string" Nil)
```

Note that CSL also supports a special *shorthand syntax for lists* that allows us to write lists in a more readable way than the above.

Neither of the `Bool` and `List a` types are defined in the standard library. Both types are builtin types in the expression language.

A number of functions that work with lists are defined in the *standard library*.

The *standard library* defines a type `Maybe a` that represents values that can be undefined:

```
type Maybe a
  | None
  | Some a
```

We may construct a `Maybe a` in one of two ways: By using the constructor `None` with zero arguments, or by using the constructor `Some x` where `x` is an object of type `a`. This is useful when a value is only present in some cases.

2.2.4 List shorthand

Although the `List` type is an ordinary sum type with *two constructors*, it is useful to have a more concise notation than writing the constructor names `Cons` and `Nil` explicitly. Therefore we may write `[]` for the empty

list, `[1]` for the singleton list with one integer element, `["a", "b", "c"]` for a list with three strings etc., as exemplified by the following table:

Shorthand	Equivalent expression
`[]`	`Nil`
`[[]]`	`Cons Nil Nil`
`[1]`	`Cons 1 Nil`
`[1.0, 2.0]`	`Cons 1.0 (Cons 2.0 Nil)`
`[[1], [2,3]]`	`Cons (Cons 1 Nil) (Cons (Cons 2 (Cons 3 Nil)) Nil)`

Using the shorthand syntax makes for more readable contract specification and is therefore encouraged. Because there is no difference between writing a list in the short or long form, the type of a list is not affected by the choice of notation. In particular, the following all specify the same list:

```
val a = Cons 1 (Cons 2 (Cons 3 Nil))
val b = Cons 1 (Cons 2 [3])
val c = Cons 1 [2, 3]
val d = [1, 2, 3]
```

Pattern matching on lists can use this shorthand notation as well.

2.2.5 Records

A *record* is a data type that contains named fields with possibly different types. To instantiate a value of a record type one must assign a value to each field that is declared as part of the record. Record types are declared with the `type` keyword followed by the (upper-case) name and a list of (lower-case) field names with their associated types.

The following is a declaration of a record type with the name `AnIntAndAFloat`. It has two fields called `theInt` and `theFloat` with the types `Int` and `Float`, respectively:

```
type AnIntAndAFloat {
  theInt : Int,
  theFloat : Float
}
```

To make values of this type we must assign values to both fields:

```
val intAndFloat1 = AnIntAndAFloat { theInt = 42, theFloat = 42.42 }
val intAndFloat2 = AnIntAndAFloat { theInt = 1, theFloat = 0.0 }
// Wrong: we forgot to specify 'theFloat'!
// val intAndFloat3 = AnIntAndAFloat { theInt = 0 }
```

Individual fields in a value of a record type can be accessed with the `.` projection syntax:

```
val theInt = intAndFloat1.theInt     // = 42
val theFloat = intAndFloat1.theFloat // = 42.42
```

A record can *inherit* fields from another record if it is declared as a *subtype*. All records are implicitly subtypes of the built-in type `Record` with zero fields. To make a record a subtype of another record—the "parent record"—we annotate our record definition with `: ParentName` after the name of the new type. For example, if we wanted to make a new record type that not only contains an `Int` and `Float`, but also a `String`, we could write the following:

```
type IntFloatAndString : AnIntAndAFloat {
  theString : String
}
```

A value of type `IntFloatAndString` is now *also* of the type `AnIntAndAFloat`. This means that to make a new value, we must fill out all the fields, including inherited ones:

```
val intFloatAndString = IntFloatAndString {
  theString = "some string value",
  theInt = 17,      // These two fields are inherited and
  theFloat = 117.4 // therefore a part of the new type.
}
```

Records can inherit from records that themselves inherit from other records. The record type `IntFloatStringAndListOfInt` below inherits from `IntFloatAndString`, which itself inherits from `AnIntAndAFloat`. This means that an `IntFloatStringAndListOfInt` record has all of the fields `theInt`, `theFloat` `theString`, and `theList`:

```
type IntFloatStringAndListOfInt : IntFloatAndString {
  theList : List Int
}
val intFloatList = IntFloatStringAndListOfInt {
  theList = [1, 2, 3],
  theString = "I have four fields", // From IntFloatAndString
  theInt = 42, // From AnIntAndAFloat
  theFloat = 2.0 * 21.0 // From AnIntAndAFloat
}
```

Record types cannot be recursive, that is, we cannot have a field in a record of the record's own type. Unlike sum types, they also cannot take type parameters.

Record extension

When you want to create a new record that contains mostly the same fields as another you can use the *record extension* syntax. Assuming a record expression `r` of type `R` is in scope, whenever you create a new record of type `R` or a subtype thereof you can add `use r with` in the beginning of list of field assigments. This means that all fields of the record denoted by `r` that are not explicitly set in the list of assignments are carried over unchanged:

```
type R { a : Int, b : Float }
type S : R { c : String }

val r = R { a = 1, b = 2.0 }
val r2 = R { use r with a = 3 }
//     = R { a = 3, b = 2.0 }
val s = S { use r with c = "123" }
//    = S { a = 1, b = 2.0, c = "123" }

// The record expression can be any expression that is of a supertype of S
val s2 = S { use S { use s with a = 15 } with b = 5.0 }
//     = S { a = 15, b = 5.0, c = "123" }
```

Examples

Even though a record type cannot contain fields of its own type, it can contain fields of other record types or sum types:

```
type Address {
  streetName : String,
  houseNumber : Int
}
type Person {
  name : String,
  idNumber : Int
}
type Direction
```

(continues on next page)

(continued from previous page)

```
  | Left
  | Right
type DirectionsTo {
  person : Person,
  from : Address,
  to : Address,
  how : List Direction
}
```

We create instances of `DirectionsTo` by nesting the record creation:

```
val dirs = DirectionsTo {
  person = Person { name = "Bob", idNumber = 1 },
  from = Address { streetName = "Main st.", houseNumber = 2 },
  to = Address { streetName = "Side av.", houseNumber = 1334 },
  how = [Left, Left, Right, Right]
}
// We can use the projection syntax '.' to access nested fields:
val fromStreet = dirs.from.streetName // "Main st."
val toStreet = dirs.to.streetName // "Side av."
```

Events in CSL contracts are records that inherit from the `Event` record type of the *standard library*. For instance, in a contract that uses delivery events encoded the type `DeliveryEvent` we must define the record `DeliveryEvent` as a subtype of `Event`:

```
type DeliveryEvent : Event {
  product : String
}
```

This means that `DeliveryEvent` will have the fields `timestamp` and `agent` from `Event` defined in addition to the field `product`.

2.2.6 Type aliases

It is possible to declare *type aliases* for types. This can sometimes help in readability, as for example type parameters that are guaranteed to be of a certain type can be expressed using a type alias. The general form of a type alias is `typealias <NAME> <type parameters> = <TYPE>` where `<TYPE>` may contain type parameters from the given list. For instance, to construct a pair-like type where the first component may be missing we could write

```
typealias PartialPair a b = Tuple (Maybe a) b
```

This alias takes two type parameters, `a` and `b`. Say we now have a use case for a `PartialPair` but we know that the component that may be missing is always an integer. This can be handled by refining the `PartialPair` we just defined:

```
typealias IntPartialPair a = PartialPair Int a
```

Furthermore, if we at some other point know that both the first and second component of the tuple is an integer we can omit the type parameters entirely:

```
typealias T = IntPartialPair Int
```

2.2.7 Functions

We can define *functions* in CSL by using *lambda notation*. A function that takes an `Int` and increments it is written:

```
val f = \x -> x + 1
```

The function `f` takes an integer as argument, binds it to the parameter `x` in the *function body* `x + 1`, and returns the result of that computation. We *apply* the function to an argument by writing the argument immediately after the function:

```
val two = f 1 // Apply 'f' to '1'
val three = (\x -> x + 1) 2 // Apply an anonymous/unnamed function to '2'
```

We have used the `val` keyword because functions themselves are just values. This means that we can pass them around just like we do with other values:

```
val addOne = \x -> x + 1
val callFuncWithValue = \func -> \x -> func x
val onePlusOne = callFuncWithValue addOne 1 // = 2
```

`addOne` is the function that increments an integer and `callFuncWithValue` is a function that takes first another function as argument, binds it to the name `f`, and returns a new function. This new function takes a single argument, calls it `x` and invokes the function bound as `f` with that argument. It is instructive to see how we can derive that the `onePlusOne` is `2`:

```
val onePlusOneDerived = callFuncWithValue addOne 1
//                    = callFuncWithValue (\x -> x + 1) 1
//                    = (\f -> \x -> f x) (\x -> x + 1) 1
//                    = (\x -> (\x -> x + 1) x) 1
//                    = (\x -> x + 1) 1
//                    = 1 + 1
//                    = 2
```

Function application is *left associative*, meaning that the following values are all equivalent:

```
// 'f', 'g', and 'h' are functions
val x = f g h 0
val y = (f g) h 0
val z = ((f g) h) 0
```

Reports

Functions are called *reports* when they are intended to be called from the outside, for example by using the interfaces generated by *sic*. Such functions must be marked with the `report` keyword, which ensures:

1. that they are *monomorphic*, that is, their argument and result types do not contain any type variables, and
2. neither their argument nor result types contain any occurrence of a function type, e.g. `Int -> Int`.

This ensures that the function can be called via the external API – it also serves a bit like the `public` of Java.

```
val report f = \(agent : Agent) -> addUpTotalsFor agent
```

As with *entry points*, reports may only occur as top-level definitions; *local* `let`-bound values cannot be reports

Multi-case functions and patterns

Functions can consist of several function bodies, each associated with a *pattern*. The particular function that is used for a certain input is chosen as the first one with a *matching* pattern. The general form of a function is:

```
\ pattern0 -> body0
| pattern1 -> body1
| ...
| patternN -> bodyN
```

Patterns are used to inspect the arguments to functions and assign names to them. A pattern is a restriction of the "shape" that the argument value has to be in for the associated function branch to be taken.

The least restrictive pattern for input arguments is the one that does not restrict at all – the *wildcard*. A wildcard pattern is written with an underscore, and it matches anything:

```
val fortyTwo = \_ -> 42
```

The above is this a function that will return `42` no matter the input. If the value of a matched wildcard is needed in a function, we may give it a name:

```
val addThree = \x -> x + 3
```

Here, `x` is a wildcard pattern that matches anything and binds it to the name `x` in the function body.

Writing patterns that restrict the allowed arguments is done by specifying how the value would have had to be constructed for it to be accepted for that particular branch. For sum types, this usually means that there is a pattern for each constructor:

```
val isEmpty =
  \ Nil -> True
  | Cons _ _ -> False
```

Here we have declared a function `isEmpty` that takes a list and returns `True` if and only if it is empty. This is done by inspecting the way that list was constructed: Either it was constructed with the `Nil` constructor, in which case it is an empty list by definition, or it was constructed with the `Cons` constructor with some arguments (which we ignore with nested wildcard patterns), which means it is a non-empty list by definition.

We can combine named wildcard with restrictions:

```
val duplicateHead =
  \ Nil -> Nil // equivalently: []
  | Cons x l -> Cons x (Cons x l)
```

In the function `duplicateHead`, different branches are selected based on the shape of the input list, just like in `isEmpty` above. However, if the input list is a `Cons` element, the element and the remaining list are given the names `x` and `l`, respectively. Sometimes it is useful to be able to name both substructures and the enclosing structure. We may attach an extra name with the `as` keyword:

```
val duplicateHead =
  \ Nil -> Nil // equivalently: []
  | (Cons x _) as l -> Cons x l
```

The result of `duplicateHead` is the same as for `duplicateHead`, but we have used the `as` keyword to give the name `l` to the *entire* non-empty input list, while still giving the head element the name `x`.

Branches in a function can be selected by looking deeper into the arguments:

```
val isFirstTwoElementsFive =
  // The list contains at least two elements and both of them are 5
  \ Cons 5 (Cons 5 _) -> True
  // The list does not start with two 5s
  | _ -> False
```

The order of the patterns is important because the first match is picked. Therefore, had we defined `isFirstTwoElementsFive` as below, it would always return `False`:

```
val isFirstTwoElementsFive =
  \ _ -> False // Wrong!  Any input will be caught by this pattern
  | Cons 5 (Cons 5 _) -> True // This will never be reached
```

Pattern-matching is also supported for tuples. A function that extracts the third element of a value of type `Tuple a b c` can be defined as:

```
val third = \(_, _, c) -> c
```

The type of the `third` function is `Tuple a b c -> c` for any types `a`, `b`, and `c`.

We can restrict the pattern of `third` so that it requires the third value to be of type `Int`:

```
val third1 = \(_, _, c : Int) -> c
```

Record pattern matching

We can do pattern matching on records and their fields by writing the record name prefixed with a `?`. A record pattern can contain sub-patterns for each field. The pattern match is successful if

1. the specified record name in the pattern designates the record's type or a supertype of the record, and
2. all field patterns match their associated field value.

The following code snippet illustrates this:

```
// Address as defined above
type BusinessAddress : Address {
  company : String
}
val isMainStreet =
  // Bind the local name "sn" to contents of "streetName"
  \ ?Address { streetName = sn } ->
    sn = "Main street"
  | _ -> False
val isMainStreet12 =
  // Match the field values with literal values
  \ ?Address {
       streetName = "Main street",
       houseNumber = 12
    } -> True
  | _ -> False
```

One important caveat to remember when using pattern matching on records is that they cause the type inferer to derive the most general record type – `Record` – as the input type for the function. Therefore, if you construct an instance of a record type and want to pattern match on it you must *upcast* it to `Record` first:

```
// isMainStreet : Record -> True
val mainStreeAddresses = List::map isMainStreet [
  BusinessAddress {
    streetName = "Main street",
    houseNumber = 12,
    company = "Deon"
  } :> Record,
  Address {
    streetName = "Wall st.",
    houseNumber = 10
  } :> Record,
  BusinessAddress {
    streetName = "Main street",
    houseNumber = 17,
    company = "Acme"
  } :> Record
]
// [True, False, True]
```

2.2.8 Conditional expressions (if-then-else)

It is possible to define conditional expressions using the syntax:

```
val a = if (True) 1.0 else 2.0 // = 1.0
val b = if (False) 1.0 else 2.0 // = 2.0
```

The condition must be some a value of the `Bool` type. We can also use more complex expressions:

```
val isGreaterThanTwo = \x -> x > 2
val g = \x -> if (isGreaterThanTwo x)
                "greater than two"
              else
                "smaller than two"
val a = g 1 // = "smaller than two"
val b = g 3 // = "greater than two"
```

2.2.9 Type case expressions

Type casing is used to split the control flow based on the type of a record. This is useful when we have a hierarchy of record types and want to perform different computations depending on which type is encountered. Given the two record definitions:

```
type Delivery : Event { deliveryMultiplier : Int }
type DroneDelivery : Delivery { droneDeliveryAddition : Int }
```

We can formulate a function that computes the prices for a `DroneDelivery` and a `Delivery` like so:

```
val deliveryCost = \(delivery : Delivery) ->
  type d = delivery of {
    // Here 'd' is a 'DroneDelivery', so we can access both fields.
    DroneDelivery -> d.deliveryMultiplier * 100 + d.droneDeliveryAddition ;
    // Here, 'd' is just a 'Delivery', so there is no droneDeliveryAddition field.
    Delivery -> d.deliveryMultiplier * 100 ;
    // Here we cannot access fields of 'd' because we don't know its type.
    _ -> 100
  }
```

A type case expression consists of several branches separated by `;`. The type case is evaluated to the first branch where the record type (left of the `->`) is a super type of the input record. There must always be a final *default case*, written with the `_` wildcard syntax. The record being cased on is bound to a name (`d` in this example), and in each branch the name refers to a value with the appropriate record type. Thus, in line 4 the `d` is a `DroneDelivery` record and therefore we may read both the field `deliveryMultiplier` and `droneDeliveryAddition`. In line 6 the `d` is a `Delivery` record, so we can only read the field `deliveryMultiplier`. Finally, in line 8 `d` is not accessible because we do not know the type of `d`.

2.2.10 Upcast annotation

Upcasts (`:>`) are used to lift record expressions to a super type. For instance we can write:

```
val delivery = \(amazon : Agent) -> DroneDelivery {
  timestamp = #2018-07-20T20:18:04Z#, agent = amazon,
  deliveryMultiplier = 2, droneDeliveryAddition = 50
} :> Delivery
```

Here `delivery` will be of type `Delivery` instead of `DroneDelivery` to the type checker. We can now use the `deliveryCost` function above to calculate delivery costs: `deliveryCost delivery`. Note that

`deliveryCost` uses a type case expression to discern between differently instantiated records. Calling the function with `delivery` causes the `DroneDelivery` case to be selected, as the value `delivery` was instantiated as a `DroneDelivery`, so a cost of `2 * 100 + 50` is returned.

The upcast annotation also allows us to specify a list of events (of type `List Event`):

```
type BikeDelivery : Event { recipient: Agent }
type BikeOrder : Event { amount: Int, recipient: Agent }

val bikeEvents = \(bob : Agent) -> \(alice : Agent) -> [
  BikeDelivery {
    timestamp = #2018-07-20T20:18:04Z#, agent = bob,
    recipient = alice
  } :> Event,
  BikeOrder {
    timestamp = #2018-07-20T20:18:04Z#, agent = bob,
    recipient = alice, amount = 100
  } :> Event
]
```

These upcasts are necessary, as `BikeDelivery` and `BikeOrder` are of different types and otherwise couldn't be in the same list.

2.2.11 Let-expressions

We use *let-expressions* to name sub-expressions within an expression:

```
val twelve =
  let
    val x = 5
    val y = 7
  in
    x + y
```

In this example, `twelve` is an expression in which `x` and `y` are *locally* bound to the values `5` and `7`. Locally bound names are visible only in the expression after the associated `in` keyword. Thus, here we may use `x` and `y` in the expression `x + y` to produce the value `12`.

After the `let` keyword we can use any *pattern* to deconstruct values:

```
val twelve =
  let
    val (x, y) = (5, 7)
  in
    x + y
```

In this example, a value of the type *Tuple Int Int* is created and immediately used in a pattern that assigns the left and right component of the pair to the names `x` and `y`, respectively. In the body after the `in` we can therefore use `x` and `y` in the same way as in the previous example.

One should be careful that the pattern always matches. The following example fails at run time as the empty list does not match a pattern for a non-empty list.

```
val fails =
    let val Cons x xs = Nil in "should fail" // fails at run-time
```

The order the definitions of values in the let-expressions matter: early definitions are visible by later ones but not the other way around:

```
val twelve =
  let
```

(continues on next page)

(continued from previous page)

```
    // val addTo = \n -> x + n // Wrong! 'x' is not in scope here!
    val x = 5
    val addTo = \n -> x + n // OK, addTo has 'x' in scope.
    val y = 7
  in
    addTo y
```

2.2.12 Type annotations

When working with records we are sometimes required to insert type-annotations in order to tell the type checker what kind of record we are projecting on. The following is a function that takes a value of the type `Event` and returns the `agent` field:

```
val getAgent = \(e: Event) -> e.agent
```

Without the annotation `: Event` on the function's parameter, the type checker cannot know which type the function expects. By inserting the annotation we make it possible for the type checker to continue with checking the function body. Because the `Event` type defines an `agent` field, the type checker can infer that this function has the type `Event -> String`.

2.2.13 Equality checking

CSL allows equality checking using the = operator for values of any type satisfying the following:

- It does not contain any occurrences of type variables.
- It does not contain any occurrence of a function type `a -> b`.
- It is *non-recursive* other than the built-in `List` type.
- It does not contain any occurrences of the signed type `Signed a`.

Note that we can only test equality for elements of the same type, i.e. typing `5 = 5.0` will result in a type error.

```
val a = [[1]] = [[1]]
//    = True

val b = [[1], []] = [[1]]
//    = False

val c = [[1]] = [[1], [1]]
//    = False

type IntFloatOrNothing
  | AnInt Int
  | AFloat Float
  | NoneOfTheAbove

val eqIntFloatOrNothing = \(x : IntFloatOrNothing) -> \y -> x = y

type AnIntAndAFloat {
  theInt : Int,
  theFloat : Float
}

val eqAnIntAndAFloat = \(x : AnIntAndAFloat) -> \y -> x = y
```

2.3 Modules

Modules provide a way to structure contract specifications into blocks. An example of a module is to group related constants such as VAT rate, payment grace period, etc. to make the contract easier to read. Modules are specified using the `module` keyword:

```
module Constants {
  val vat = 0.25
  val paymentGrace = 10 // days
}
```

To use declarations in a module, the `::` syntax is used:

```
val a = Constants::paymentGrace
// a is 10
```

Modules can be nested into other modules:

```
type BaseShape {}
module Shape {
  type Circle : BaseShape {
    radius : Float
  }
  module Circle {
    val pi = 3.14159
    val area = \(c : Shape::Circle) -> c.radius * c.radius * Shape::Circle::pi
  }

  type Rectangle : BaseShape {
    length : Float,
    width : Float
  }
  module Rectangle {
    val area = \(r : Shape::Rectangle) -> r.length * r.width
  }
}

// Compute area for any Shape
val area = \(s : BaseShape) ->
  type x = s of {
    Shape::Circle -> Shape::Circle::area x;
    Shape::Rectangle -> Shape::Rectangle::area x;
    _ -> 0.0
  }
```

Notice how values are accessed in modules using the `ModuleName::value` notation.

It is also possible to define contracts in modules:

```
module Sale {
  type Sale: Event {}
  val inventory = ...
  template Sale(item, price, buyer) = ...

  val income = ...
}

module Purchase {
  type Purchase: Event {}
  val stock = ...
  template Purchase(item, price, seller) = ...
```

(continues on next page)

(continued from previous page)

```
  val expenses = ...
}
```

Some things to consider when working with modules:

- You must always use the full module path to refer to declarations, also when referring to other declarations inside the same module. In the example above, we wrote `Shape::Rectangle` and `Shape::Circle` inside the module `Shape` to refer to the circle and rectangle types, for example.
- Modules do not provide any isolation or restrict access: Any value/type/contract defined in a module is accessible with the full module path.

2.4 Scoping

When an entity is bound to name a *binding* is created. A binding has a *scope* in which it is effective (or "visible"). An entity can be a value, defined with `val`; a contract template, defined with `template`; or a contract shorthand, defined with `contract`.

In this section we will use the convention that a comment of the form `// { a = 1, b : Int }` means that there are two bindings in scope on the line of the comment: one binding of the name `a` to the value `1` and one binding of the name `b` to some value of type `Int`. We use this to illustrate how the scoping of names behaves.

We illustrate most of the scoping rules with values (`val`), but the same rules apply for contract abbreviations (`contract`) and contract templates (`template`) unless otherwise specified. For the latter, the additional rules for recursive templates also apply – see *below*.

2.4.1 Sequential bindings

In the basic case, the scope of a binding is the lexical substructure of the code that follows immediately *after* the binding itself. For example, consider the following code snippet:

```
val a = 1
val b = a + 1 // { a = 1 }
val c = b + 1 // { a = 1, b = 2}
// { a = 1, b = 2, c = 3 }
```

In the binding of `b` to a value, `a` is in scope and in the binding of `c` both `a` and `b` are in scope. Recursive values are disallowed in CSL, so the binding of `a` to `1` is not visible to itself and likewise for `b` and `c`.

It is not allowed to redeclare definitions, so we cannot shadow existing top-level definitions like so:

```
val someValue = 10
val someValue = 2000   // not allowed
```

2.4.2 Simultaneous bindings

Sometimes it is not desirable to bind names sequentially. If we need to bind, e.g., three values to names and they are not dependent on one another, the order in which they are written in the source code should not have any effect on them. A block of simultaneous bindings may contain several bindings:

```
val severalWiths = let
  val  a = 1
  with b = 2
  with c = "hello"
  with d = False
  // with e = a
in d
```

Had we not commented out the last line in this example it would not have typechecked: we would have attempted to bind `e` to `a` before `a` had been bound. Note that each individual binding in a block can be marked as a `report`, making it accessible by *sic* interfaces and subjecting it to the additional requirements of *reports*:

```
val report a = 1
with b = 2 // not a report
with report c = 3
```

2.4.3 Local scopes

It is also possible to explicitly mark where a scope should end: such scopes we call *local scopes*. A binding can be introduced in a local scope by using the `let ... in` construction (see also *here* and *here*). The general form is as follows:

```
let
  <bindings>
in
  <body>
```

The `<bindings>` can be one or more bindings like described above; `<body>` is the scope in which these bindings are visible. In the code following the `<body>` the definitions from `<bindings>` are out of scope.

The scoping rules for locally bound names in a `let` and on the top-level are the same. In particular, sequential bindings can be "unrolled" as a sequence of nested `let` blocks:

```
val myVal0 =
 let
   val a = 1
   val b = 2
   val c = a + b
 in
   c
// is equivalent to
val myVal1 =
 let val a = 1
 in
   let
     val b = 2
     val c = a + b
   in c
// is equivalent to
val myVal2 =
let val a = 1
in
  let val b = 2
  in
    let val c = a + b
    in
      c
```

When we are working in the value language, a `let`-expression is itself also a *value*. This means that we may use it anywhere a value can be used:

```
val x = 2
val a = (let val x1 = 42 in x1 + 1) + (let val y = 15 in x + y)
//  a = (42 + 1) + (2 + 15)
// { a = 60, x = 2 }
```

One must take care that the body of a `let`-expression does not accidentally span over more than what was intended. For example, if we had forgotten the parentheses in the above code snippet the meaning would have

been different because the inner `x1` with a value of `42` would have been in scope in the second subexpression `x1 + y` as well:

```
val x = 2
val a = let val x2 = 42 in x2 + 1 + let val y = 15 in x2 + y
//  a = 42 + 1 + 42 + 15
// { a = 100, x = 2 }
```

Locally scoped bindings do not differ from bindings at the top level apart from their scope. We may for example also use simultaneous bindings in `let`-blocks:

```
val c = let
  val  a = 1
  with b = 2
  in
  a + b          // { a = 1, b = 2 }
```

Note that it is not allowed to shadow names locally in `let` blocks:

```
val x = 10
val f = let x = 5 in x    // not allowed
```

Local scopes without `let`

Attention: This only applies to values, not to contract abbreviations or templates.

Aside from `let`-expressions, bindings with a local scope can be introduced in three ways:

- with a *function*;
- as a named event in the *prefix contract* construction;
- with the *type case* construct; or
- as value parameters for *templates*.

Function scope

Binding a name in a function is similar to binding a name in a `let`-expression. The binding has a scope that covers the body of the function, and for everything outside of the function body the binding does not exist:

```
val f = \ x ->
  x + 1        // { x : Int }
val x = f 1    // { f : Int -> Int }
val y = x + 1 // { f : Int -> Int, x = 2 }
```

Recall that parameters to functions are *patterns*; every variable in the pattern is introduced as a binding in the scope of the function body:

```
val g =
  \ (x : Int) ->
  \ (a, b) ->
   (a + x, b + x)     // { a : Int, b : Int, x : Int }
val p = g 1 (10, 11) // { g : Int -> Tuple Int Int -> Tuple Int Int }
// { g : Int -> Tuple Int Int -> Tuple Int Int, p = (11, 12) }
```

Bindings of names introduced by patterns in multi-case functions only have scope in the body of the their own case:

```
type T | Left Int | Right Int
val h =
  \ (x : Int) ->
  \ Left l -> Left (l + x)    // { x : Int, l : Int }
  | Right r -> Right (r + x) // { x : Int, r : Int }
val l = h 1 (Left 10)         // { h : Int -> T -> T }
val r = h 2 (Right 10)        // { h : Int -> T -> T, l : T }
// { h : Int -> T -> T, l : T, r : T }
```

Note that it is not allowed to shadow names in parameter patterns:

```
val x = 10
val f = \x -> x   // not allowed
```

Prefix-contract scope

If a prefix contract binds a name to the matched event, that binding has a scope that covers the predicate and the residual of the prefix contract:

```
type M0 : Event { x : Int }
template C0() =
  <*> e0 : M0
  where e0.x > 0    // { e0 : M0 }
  then
  <*> e1 : M0
  where e1.x > e0.x // { e0 : M0, e1 : M0 }
  then
  <*> e2 : M0
  where e2.x > e1.x // { e0 : M0, e1 : M0, e2 : M0 }
```

The scope only spans the residual contract of the prefix in which the binding is made; hence, the scoping can be altered with parentheses:

```
type M1 : Event { x : Int }
template C1() =
  (<*> e0 : M1
   where e0.x > 0    // { e0 : M1 }
   then
   <*> e1 : M1
   where e1.x > e0.x // { e0 : M1, e1 : M1 }
  )
  then
  <*> e2 : M1
  where e2.x > 0     // { e2 : M1 }
```

Bindings introduced this way cannot shadow one another:

```
template C2() =
  <*> e : Event
  then
  <*> e : Event // fails check because we bind e again
```

Type case scope

The type case construct introduces a binding of the value that is being matched against in the scope of all cases. The binding in each case will have the same name but a value of a different type, as it will bind the name to a value of the specific type of the case. The mandatory default case is the exception: here, the type case construct does not introduce any bindings, so the only ones in scope those that are in the scope of the entire construct:

```
type Rec { def : Int }
type A : Rec { aField : Int }
type B : Rec { bField : Int }
val fn = \(r : Rec) ->
  type x = r of {                          // { r : Rec }
    A -> x.aField + x.def;                 // { r : Rec, x : A }
    B -> x.bField + x.def;                 // { r : Rec, x : B }
    _ -> r.def                             // { r : Rec }
  }
val a = fn (A { aField = 1, def = 1 } :> Rec) // { fn : Rec -> Int }
val b = fn (B { bField = a, def = 2 } :> Rec) // { fn : Rec -> Int, a = 2 }
val c = fn (Rec { def = 42 })                 // { fn : Rec -> Int, a = 2, b = 4 }
// { fn : Rec -> Int, a = 2, b = 4, c = 42 }
```

Note that in the default case above only `{r : Rec}` is in scope. This binding is in scope because it was introduced by the function in which the type case expression resides.

Value parameters in templates

Bindings for value parameters in contract templates behave like bindings introduced in functions. A binding is made for each variable name in the patterns, and the scope of those bindings is the entire body of the contract template:

```
type E : Event { x : Int }
template A(x:Int, (a : Int, b : Int)) =
  <*> e1: E      // { a : Int, b : Int, x : Int }
  where e1.x = x // { a : Int, b : Int, x : Int, e1 : E }
  then
  <*> e2: E      // { a : Int, b : Int, x : Int, e1 : E }
  where e2.x = a // { a : Int, b : Int, x : Int, e1 : E, e2 : E }
```

2.4.4 Recursive bindings

So far we have only discussed scoping that disallows recursion. The scoping rules for contract templates are the same as for values and contract abbreviations regarding the difference between sequentially and simultaneously bound templates, but they also handle the case of recursive templates. A recursive template is written by adding the `rec` modifier after `template` when binding it. Adding this modifier has the effect that the template can be referred to in its own definition – the binding is in scope in its definition:

```
// Error: Template Bad not in scope
// template Bad() = <*> Event then Bad()

// OK: Template Good is in scope
template rec Good() = <*> Event then Good()
```

The addition of recursion is orthogonal to the distinction between sequential and simultaneous bindings. That is, in the following code snippet we sequentially bind three recursive contracts, each one able to see both itself (due to `rec`) and the preceding definitions:

```
template rec RecA() = <*> Event then RecA()
template rec RecB() = RecA() or <*> Event then RecB()
template rec RecC() = RecA() or RecB() or <*> Event then RecC()
```

It is also possible to simultaneously bind recursive contracts. This has the effect that all bindings from the simultaneous block are visible in all of the right-hand-sides, thereby allowing us to write *mutually recursive templates*:

```
template rec RecA() = <*> Event then RecB()
with         RecB() = <*> Event then RecA()
```

(continues on next page)

(continued from previous page)

```
// This would fail as 'RecA' and 'RecB' are not scope due to the missing 'rec'
// template RecA() = <*> Event then RecB()
// with     RecB() = <*> Event then RecA()
```

2.4.5 Overview

The table below is a summary of the different ways values, contract abbreviations, and contract templates can be assigned scope:

Kind	Sequential	`with`	`rec`	`let`	Function	Prefix contract	Type case
`val`	X	X		X	X	X	X
`contract`	X	X		X			
`template`	X	X	X	X			

2.5 Report query language

The *report query language* (RQL) is an embedded language in CSL that allows defining relations on data from past events. It is possible to use relations in the expression language to compute aggregate information based on the data in the relations.

2.5.1 Overview

RQL is a variant of Datalog which is a declarative logic-based programming language. RQL introduces two new constructions:

- *relation definitions* for describing relations between data of past events; and
- *for expressions* for computing aggregate information from past events.

A relation definition merely *declares* a relationship between data - by itself it does not "do" anything. This decoupling of the logical specification from the actual query execution allows the CSL runtime system to apply various optimization techniques. The `for` expressions are used to query the relations and apply a custom aggregation function to the result tuples. Relationships are expressed by merely *stating* them, and the computation of the final query result must take care of finding a set of tuples that satisfies the constraints set out in the definitions of the relations through a process called *unification*. Informally, each tuple in the result set will satisfy the property that if it is substituted into the relation being queried then it will be satisfied. Moreover, the result set will contain all tuples with this property that can be deduced from the system in its current state given the events seen so far.

2.5.2 Relations

A `relation` specifies a relation between values such that they satisfy one or more of the *clauses* given in the definition. These relations can be thought of as sets of *n*-ary tuples, and can be conveniently visualized as a table in this text.

To define a relation, we use the top-level definition `relation`:

```
relation relationName(<params>) <clauses>
```

Here, `<params>` (the "head") and `<clauses>` (the "body") stand for a list of parameters `x_0, ..., x_n` and a list of clauses `clause_0, ..., clause_m`, respectively. The name `relationName` is the name of a relation between values of the parameters such that at least one of the clauses `clause_0` - `clause_m` can be *satisfied*. A clause is satisfied of there exists a value substitution of all parameters that occur in the clause such that the clause evaluates to `True`.

Parameters are formed using the normal syntax for constants and variables we use *elsewhere* in the language. In particular, type annotations may occur anywhere to either aid in documentation or to help the type checker, or both. The parameters in the head are special insofar that only variables, possibly type-annotated, may occur there. Relations themselves are also typed: the inferred type of a relation is a tuple consisting of each of the types of the parameters in the relation's head.

There are three basic clause constructs and one combinator for forming bigger clauses from smaller ones:

equality clauses `x = y`, stipulating the equivalence of two parameters. This clause is satisfied when the values being substituted for `x` and `y` are equal;

event-matching clauses `Event { field0 = param0 , ..., fieldN = paramN } @ cid`, denoting that an event of the given type and with the given fields has been applied on a contract. This clause is satisfied when there exists a value substitution of `param0` to `paramN` such that an event with the corresponding fields set to those values has happened on contract with contract id `cid`;

relation-invocation clauses `relationName(param0, ..., paramN)`, which is satisfied only if the specified relation `relationName` exists and can be satisfied with the given parameters; and

conjunctive clauses the conjunction `and` is used to form more advanced clauses from basic clauses: `clause0 and ... and clauseN`..

The parameters for a relation must be a *subset* of the variables that occur in the rule-body. For example, the following relation definition is *not* allowed, as none of `a`, `b`, or `c` occur in the clause:

```
relation gibberish(a, b, c)
  | Sale {} @ cid
```

Equality clauses

To stipulate that two terms must be equal, we use the *equality clause*: `x = y`. Either side of the equality symbol may be a constant.

Example

```
relation theNumberTwo(x)
  | x = 2
```

The relation `theNumberTwo` is unary, so it just consists of a set of values. This relation is satisfied for those values that can be substituted for `x` and make one of the clauses satisfied. In this example, there is only one clause, `x = 2`, stipulating that `x` must be equal to `2` for it to be satisfied. Clearly, there is exactly one satisfying assignment for `x` here, namely the value `2`. As it is the only clause in the relation, it contains only the value `2`.

Example

We can expand the simple relation from the previous example with some more clauses to make the unary relation bigger:

```
relation fourFirstPrimes(x)
  | x = 2
  | x = 3
  | x = 5
  | x = 7
```

This relation contains the values `2`, `3`, `5`, and `7`.

Example

In the previous two examples, constants were involved in all the equality clauses. We can also use parameters on both sides of the equality sign:

```
relation same(x : Int, y : Int)
  | x = y
```

The binary relation denoted by `same` contains all `Int` tuples `(x, y)` such that `x` is equal to `y`. This relation is of course exactly that denoted by the equality clause itself, so the rule `same` is superfluous. However, in more advanced settings, where there are multiple constraints on the parameters, requiring two parameters to be equivalent becomes useful.

Note: Note that we have used type annotations in the relation definition above. Relations must be well-typed and the contract writer may use type annotations to guide the type inferrer.

Event-matching clauses

A clause may contain a condition on past events of contracts:

```
Sale { saleId = x } @ someId
```

The clause above expresses a condition that is true if the contract with id equal to that assigned to the variable `someId` was matched with an event of type `Sale` where the field `saleId` was assigned a value equal to the value of the variable `x`.

Example

```
type Item | Book | Movie | Coffee

type Sale : Event {
    saleId : Int,
    price: Int,
    item: Item
}

relation saleIds(x, y)
    | Sale { saleId = x }@y
```

The relation `saleIds` contains pairs of values `(x, y)` of the type `Tuple Int Contract` for all events of type `Sale` applied to contract with id `y` and where the value of `saleId` on the event was `x`.

Suppose the following events have been applied: (reusing the notation `ev @ c` to denote that event `ev` was applied to contract with contract-id `c`)

```
...
Sale { saleId = 1, price = 10, item = Book } @ c1
Sale { saleId = 2, price = 42, item = Coffee } @ c1
Sale { saleId = 1, price = 10, item = Book } @ c2
Sale { saleId = 2, price = 10, item = Book } @ c2
Sale { saleId = 3, price = 117, item = Movie } @ c1
...
```

The values related by the `saleIds` relation can be visualized in the following table:

x	y
1	c1
2	c1
1	c2
2	c2
3	c1

Example

A slightly extended relation that includes more information about the event-data is given below:

```
relation sales(saleId, price, item, cid)
  | Sale { saleId = saleId, price = price, item = item }@cid
```

In the context of the events given above, the following table visualizes the data as related by `sales`.

saleId	price	item	cid
1	10	Book	c1
2	42	Coffee	c1
1	10	Book	c2
2	10	Book	c2
3	117	Movie	c1

Relation-invocation clauses

A clause may consist of an invocation of a relation with a list of parameters: `relationName(param0, ..., paramN)`. These parameters can be either constants or variables themselves. If a relation is invoked with only constants it corresponds to a check of whether the tuple containing those constants is in the relation. Invoking a relation with *n* separate variables as parameters denotes the same relation as the rule itself, but restrictions can be placed by giving the same variable multiple times.

Example

The relation `lessThanFive` denotes the set containing the integers `0` to `4`:

```
relation lessThanFive(x) | x = 0 | x = 1 | x = 2 | x = 3 | x = 4
```

We can invoke a relation as follows, essentially creating an alias:

```
relation alsoLessThanFive(x) | lessThanFive(x)
```

This relation definition expresses that whatever integer is in the unary relation `lessThanFive` is also in the unary relation `alsoLessThanFive`.

Conjunctive clauses

A clause in a relation may contain multiple conditions using the conjunctive combinator `and`. The format of a conjunctive clause is:

```
clause0 and clause1 and ... and clauseN
```

where each of `clause0` to `clauseN` are basic (non-conjunctive) clauses. Combining two or more clauses with `and` yields a new clauses that is satisfied when all of the constituent subclauses are satisfied.

Example

```
relation bookSales(s, p, cid)
  | Sale { saleId = s, price = p, item = i} @ cid and i = Book
```

The satisfying assignments of variables for the `bookSales` relation in the context of the previously shown events are:

s	p	cid
1	10	c1
1	10	c2
2	10	c2

Example

In this example, we assume that an event `Payment` can also be applied to the contracts to which `Sale` can be applied. A relation that collects information from all `Payment`-events is given by the `payments` rule.

```
type Payment: Event {
  receiver: Agent,
  amount: Int,
  id: Int
}

relation payments(r, a, i, cid)
  | Payment { receiver = r, amount = a, id = i } @ cid
```

Assume that the following events are added.

```
Payment { receiver = a1, amount = 10, id = 1} @ c1
Payment { receiver = a2, amount = 10, id = 1} @ c2
Payment { receiver = a3, amount = 117, id = 3} @ c1
```

The relation `payments` is visualized in the following table:

r	a	i	cid
a1	10	1	c1
a2	10	1	c2
a3	117	3	c1

Joining relations

By combining relation invocations with conjunctions we can formulate *joins*. Joining relations is central to the usage of RQL, so it does not have any dedicated syntax. By repeating variable names in multiple places in a clause we introduce constrains corresponding to joins: because the system finds satisfying assignments of all variables occurring in the clause, it is enough to repeat a variable multiple places to express that the same value must be found in that subpart.

Example

For example, the relations `sales` and `payments` can be combined into a joined relation that collects information for sales that have received payment:

```
relation paidSales(id, amount, item, cid)
  | sales(id, amount, item, cid) and payments(r, amount, id, cid)
```

The occurrence of the same variable in `sales` and `payments` ensures that only the assignments to variables that are part of *both* relations are included as part of the `paidSales` relation.

id	amount	item	cid
1	10	Book	c1
1	10	Book	c2
3	117	Movie	c1

Example

One can further constrain the `paidSales` relation and specialize it into a `paidBookSales` rule:

```
relation paidBookSales(id, amount, cid)
  | sales(id, amount, x, cid)
    and payments(r, amount, id, cid)
    and x = Book
```

In this case, we can inline the equality-condition on `x = Book` and get:

```
relation paidBookSales(id, amount, cid)
  | sales(id, amount, Book, cid) and payments(r, amount, id, cid)
```

id	amount	cid
1	10	c1
1	10	c2

Multiple clause-bodies

A relation may be defined by the union (disjunction) of multiple clauses:

```
relation rName(params)
  | clause_1
  | clause_2
  | ...
  | clause_n
```

An assignment of values to parameters is satisfying if at least *one* of `clause_1` ... `clause_2` can be evaluated to `True`.

Example

A relation that includes sales of *either* `Coffee` or `Movie` can be given as:

```
relation bookOrMovieSales(id, price, cid)
  | sales(id, price, Coffee, cid)
  | sales(id, price, Movie, cid)
```

The visualized version is below:

id	price	cid
2	42	c1
3	117	c1

Example

The clauses in disjunctions may be completely different:

```
relation saleOrPayment(id, cid)
  | sales(id, amount1, item, cid)
  | payments(r, amount2, id, cid)
```

Recursive relations

Relations can be recursive if the `rec` keyword is used. This allows us to express complex relationships in a succinct manner.

A recursive relation is declared in the same way as a normal one, except that we add the keyword `rec`:

```
relation rec relationName(<params>) | <clauses>
```

Clauses in the list `<clauses>` may now include relation invocations of the relation `relationName` itself. Relations may also be mutually recursive: this is achieved with the combination of `rec` and `with` also used for mutually recursive templates:

```
relation rec relation0(<params0>) | <clauses0>
with relation1(<params1>) | <clauses1>
...
with relationN(<paramsN>) | <clausesN>
```

The relations `relation0` to `relationN` may freely refer to one another recursively through relation invocations.

Example

The following CSL code demonstrates how to express a *transitive closure* of another relation. Customers are related via their id strings through `CustomerEvent` events. In line 6 the relation `directlyRelated` is defined, containing pairs of customer ids such that an event has happened (on any contract id, since `cid` is free) that relates the two. Next, line 9 defines the relation `relatedCustomers` as the transitive closure of `directlyRelated`. Customer id pairs belong to this relation if either 1) they are directly related, as per the `directlyRelated` relation, or 2) there exists some third customer id `x` that `ancestor` is related to and which itself is directly related to `descendant`. Hence, customer id pairs in the `relatedCustomers` may be either directly related or indirectly related through a chain of ids.

```
1  type CustomerEvent : Event {
2    customerId: String,
3    referrerId: String
4  }
5
6  relation directlyRelated(parent, child, cid)
7    | CustomerEvent { customerId = parent, referrerId = child } @ cid
8
9  relation rec relatedCustomers(ancestor, descendant, cid)
10   | directlyRelated(ancestor, descendant, cid)
11   | relatedCustomers(ancestor, x, cid)
12     and directlyRelated(x, descendant, cid)
```

Let us assume that the following events have occurred:

```
CustomerEvent { customerId = 2, referrerId = 1 } @ c1
CustomerEvent { customerId = 3, referrerId = 1 } @ c1
CustomerEvent { customerId = 4, referrerId = 3 } @ c1
CustomerEvent { customerId = 5, referrerId = 3 } @ c1
CustomerEvent { customerId = 6, referrerId = 2 } @ c1
```

The relation `directlyRelated` now contains:

parent	child	cid
1	2	c1
1	3	c1
3	4	c1
3	5	c1
2	6	c1

If we draw this relation as a tree we get something akin to a "referral hierarchy":

```
    1
   / \
  2   3
 /   / \
6   4   5
```

The relation `relatedCustomers` expresses that a customer id occurs below another one in this tree:

ancestor	descendant	cid
1	2	c1
1	6	c1
1	3	c1
1	4	c1
1	5	c1
2	6	c1
3	4	c1
3	5	c1

If we want to take the *reflexive* transitive closure, i.e., express that a customer id is always related to one self, we simply add an extra clause:

```
relation relatedCustomersRefl(ancestor, descendant, cid)
  | CustomerEvent { customerId = ancestor } @ cid
    and ancestor = descendant
  | relatedCustomers(ancestor, descendant, cid)
```

2.5.3 `for` expressions

Relation definitions do not by themselves compute the tuples that are members of the relation, they merely state the constraints such tuples must satisfy. In order to get actual values that can be used in the expression language as part of reports we must execute a *query* over a relation. For this purpose, the expression language contains the `for` expression, which performs a computation over the result tuples of a query over a relation:

```
for pattern in relationName do reducer
```

A `for` expression provides a way to perform a computation over all the satisfying assignments of values to parametes for the given relation. It contains three parts:

- the `pattern` is a pattern that is used to assign names to the values;

- the `relationName` is some previously-defined relation; and
- the `reducer` is an expression of the type `t -> t`, where `t` is the type of the result value that is being built by the query.

Any names bound in the pattern will be in scope in the `reducer`, and the `reducer` expression is called once for each value in the result set, carrying along the partially built result of type `t` and returning a "more complete" result of the same type. The type of the entire `for` expression is itself the type of the given reducer, i.e., `t -> t`. Thus, once we have built an expression with the `for` construct, we need to supply it some initial state in order to compute the complete result. An example of a an initial value would be `0` if the state type `t` is an `Int` and the reducer computes a sum.

Example

The following simple example demonstrates how the `for` expression is used to perform a query computing a sum based on past events. We first define a sum type `BookOrMovie` that we use as an example of data which can be used to discern between events of the type `BookOrMovieEvent` through the `style` field. Next we define the relation `bookStyleEventTimeStamps` between time stamps and contract ids - i.e., a set of tuples of the type `Tuple DateTime Contract`. Each tuple in the set has the property that the timestamp is from a `BookOrMovieEvent` where the `style` field is set to the value `Book`. This is ensured by the constraint put on the `style` field in line 7.

The relation is used in a query in lines 11 and 12, where we deconstruct each tuple in the result set with the pattern `(_, c)`. This pattern ignores the actual timestamp values by using the _ wildcard pattern and it binds the contract ids to the name `c`. Our report `countFrom` takes a contract id `cid` and yields a function that, given an initial integer value, returns the sum of the initial value and the number of `Book`-style `BookOrMovieEvent`'s that have happened on the contract `cid`. The report `sum` uses `countFrom` and sets the initial value to `0`.

```
1  type BookOrMovie | Book | Movie
2  type BookOrMovieEvent : Event {
3    style : BookOrMovie
4  }
5
6  relation bookStyleEventTimeStamps(t, cid)
7    | BookOrMovieEvent { timestamp = t, style = Book }@cid
8
9  /// countFrom : Contract -> Int -> Int
10 val countFrom = \cid ->
11   for (_, c) in bookStyleEventTimeStamps do
12   if (c = cid) \x -> 1 + x else \x -> x
13
14 /// sum : Contract -> Int
15 val sum = \cid -> countFrom cid 0
```

Caveat

The use of `for`-expressions isn't supported in `where`-clauses of contracts. Rules and `for`-expressions are supported only when running reports (functions) externally.

Example

A consequence of disallowing use of `for`-expressions is that the following definition is not allowed:

```
1  type BookOrMovie | Book | Movie
2  type BookOrMovieEvent : Event {
3    style : BookOrMovie
4  }
5
6  relation bookStyleEventTimeStamps(t, cid)
```

(continues on next page)

(continued from previous page)

```
  | BookOrMovieEvent { timestamp = t, style = Book }@cid

/// countFrom : Int -> Int
val countFrom =
  for (_, _) in bookStyleEventTimeStamps do
    \x -> 1 + x

/// sum : Contract -> Int
val sum = countFrom 0
```

2.6 The CSL Standard Library

This section gives an overview of the types and functions which are currently available from the CSL standard library. A number of types are built-in and therefore not defined in the standard library. The complete source code of the standard library can be seen *here*.

2.6.1 Functions operating on numbers

The following functions supplement the built-in operators on numbers, like +, ⋆, −, /.

`Int::toFloat : Int -> Float`

Converts an `Int` to a `Float`.

Examples

```
val a = Int::toFloat 4
//    = 4.0
```

`Int::toString : Int -> String`

Converts an `Int` to a `String`.

Examples

```
val a = Int::toString 4
//    = "4"
```

`Math::abs : Int -> Int`

Get the absolute value of an `Int`.

Examples

```
val a = Math::abs (10 - 15)
//    = 5
val b = Math::abs 8
//    = 8
```

Math::fabs : Float -> Float

Get the absolute value of a `Float`.

Examples

```
val a = Math::fabs (10.0 - 15.0)
//    = 5.0
val b = Math::fabs 8.0
//    = 8.0
```

Math::pow : Float -> Float -> Float

The power function: a^b.

Examples

```
val a = Math::pow 4.0 3.0
//    = 64.0
```

Math::sqrt : Float -> Float

The square root function: $\sqrt{a}$.

Examples

```
val a = Math::sqrt 9.0
      = 3.0
```

2.6.2 String functions

String::append : String -> String -> String

Append two strings.

Examples

```
val a = String::append "Hello, " "World!"
//    = "Hello, World!"
```

2.6.3 Boolean functions

The type `Bool` is a built-in type defined as:

```
type Bool
  | True
  | False
```

There are a number of built-in operators defined on values of type `Bool`, e.g., `&&`, `||`, and the `if` expression.

`not : Bool -> Bool`

The `not` function returns the negated value of its input.

Examples

```
val a = not True
//    = False
val b = not False
//    = True
```

2.6.4 Time and date functions

The following functions operate on the built-in type `DateTime` for timestamps.

`DateTime::addSeconds : DateTime -> Int -> DateTime`

The function `DateTime::addSeconds` adds seconds to a timestamp.

Examples

```
val a = DateTime::addSeconds #2017-12-24T13:37:00Z# 60
//    = #2017-12-24T13:38:00Z#
```

`DateTime::addDays : DateTime -> Int -> DateTime`

Add days to a timestamp.

Examples

```
val now = #2018-02-08T10:00:00Z#
val tomorrow  = DateTime::addDays now 1
//            = #2018-02-09T10:00:00Z#
val yesterday = DateTime::addDays now (0 - 1)
//            = #2018-02-07T10:00:00Z#
```

DateTime::Components

The type `DateTime::Components` represents the components of a `DateTime` value, e.g., year, month, etc. The `DateTime::Components` is a built-in type.

The definition is

```
module DateTime {
  type Components {
    year : Int,
    month : Int,
    day : Int,
    hour : Int,
    minute : Int,
    second : Int
  }
}
```

DateTime::components : DateTime -> DateTime::Components

Extract a `DateTime::Components` representation from a `DateTime` value.

Examples

```
val a = DateTime::components #2018-02-08T10:15:30Z#
//    = DateTime::Components {
//        year = 2018,
//        month = 2,
//        day = 8,
//        hour = 10,
//        minute = 15,
//        second = 30
//      }
```

DateTime::DayOfWeek

The built-in type `DateTime::DayOfWeek` represents the days of the week.

Its definition is

```
module DateTime {
  type DayOfWeek
    | Monday
    | Tuesday
    | Wednesday
    | Thursday
    | Friday
    | Saturday
    | Sunday
}
```

DateTime::dayOfWeek : DateTime -> DateTime::DayOfWeek

Get the day of week for a `DateTime` value.

Examples

```
val monday = DateTime::dayOfWeek #2007-01-01#
//         = DateTime::Monday
```

DateTime::dayCountFraction : DateTime::DayCountFraction -> DateTime -> DateTime -> Float

Get the Day Count Fraction according to some convention, where conventions are defined as:

```
module DateTime {
  type PaymentFrequency
    | Annual
    | SemiAnnual
    | Quarterly

  type DayCountFraction
    | ActActISDA DateTime DateTime
    | ActActICMA DateTime DateTime DateTime DateTime DateTime::PaymentFrequency
    | Act365F DateTime DateTime
    | Act360 DateTime DateTime
    | Act365A DateTime DateTime
    | Thirty360ISDA DateTime DateTime
    | Thirty360ICMA DateTime DateTime
}
```

Where the first two parameters are *from* and *to* dates for the calculation. The additional parameters for *ActActICMA* are *firstPayment*, *maturity*, and *frequency* respectively.

Examples

```
val dcf = DateTime::dayCountFraction (DateTime::ActActISDA #2007-12-28T00:00:00Z#
↪#2008-02-28T00:00:00Z#)
//      = 4/365 + 58/366 ~ 0.16942884946478
```

2.6.5 Duration functions

The following functions operate on the built-in type `Duration` for durations.

Duration::fromSeconds : Float -> Duration

The function `Duration::fromSeconds` creates a new duration from seconds. You can specify milliseconds using the fractional part of the `Float`.

Examples

```
val a = Duration::fromSeconds 62.123
//    = #PT1M2.123S#
```

Duration::fromMinutes : Float -> Duration

The function `Duration::fromMinutes` creates a `Duration` from a `Float` representing minutes.

Examples

```
val a = Duration::fromMinutes 62.5
//    = #PT1H2M30S#
```

Duration::fromHours : Float -> Duration

The function `Duration::fromHours` creates a `Duration` from a `Float` representing hours.

Examples

```
val a = Duration::fromHours 4.25
//    = #PT4H15M#
```

Duration::fromDays : Float -> Duration

The function `Duration::fromDays` creates a `Duration` from a `Float` representing days.

Examples

```
val a = Duration::fromDays 2.5
//    = #P2DT12H#
```

Duration::toSeconds : Duration -> Float

The function `Duration::toSeconds` converts a duration to a `Float` the value of which is the total seconds of the duration.

Examples

```
val a = Duration::toSeconds #PT1M2.123S#
//    = 62.123
```

Duration::addSeconds : DateTime -> Float -> DateTime

The function `Duration::addSeconds` adds the given number of seconds to a duration. In contrast with *DateTime::addSeconds* this function takes a `Float` which therefore allows one to add milliseconds to a timestamp.

Examples

```
val a = Duration::addSeconds #2018-01-01T00:00:00# 62.123
//    = #2018-01-01T00:01:02.123#
```

Duration::between : DateTime -> DateTime -> Duration

The function `Duration::between` returns a `Duration` which is the number of days, hours, minutes and seconds between the arguments.

Examples

```
val a = Duration::between #2018-02-01# #2018-03-01T00:00:00.123#
//    = #P28DT0.123S#
val b = Duration::between #2018-02-01# #2018-03-01#
//    = #P28D#
```

`Duration::diffDateTimes : DateTime -> DateTime -> Duration`

The function `Duration::diffDateTimes` returns a `Duration` which is the number of days, hours, minutes and seconds that passes from the second argument to the first.

Examples

```
val a = Duration::diffDateTimes #2018-03-01T00:00:00.123# #2018-02-01#
//    = #P28DT0.123S#
val b = Duration::diffDateTimes #2018-02-01# #2018-03-01#
//    = #-P28D#
```

`Duration::addDurations : Duration -> Duration -> Duration`

The function `Duration::addDurations` adds two durations together.

Examples

```
val a = Duration::addDurations #PT1H2M3S# #PT60S#
//    = #PT1H3M3S#
```

`Duration::subDurations : Duration -> Duration -> Duration`

The function `Duration::subDurations` subtracts the second duration from the first.

Examples

```
val a = Duration::subDurations #PT1H2M3S# #PT60S#
//    = #PT1H1M3S#
```

`Duration::addToDateTime : DateTime -> Duration -> DateTime`

The function `Duration::addToDateTime` adds a `Duration` to a `DateTime`.

Examples

```
val a = Duration::addToDateTime #2018-01-01# #P1D#
//    = #2018-01-02#
```

`Duration::negate : Duration -> Duration -> Duration`

The function `Duration::negate` negates a `Duration`.

Examples

```
val a = Duration::negate #P1DT2S#
//    = #-P1DT2S#
```

`Duration::components : Duration -> Duration::Components`

The function `Duration::components` returns a value of the type `Duration::Components` which represents the individual components of time of the argument.

Examples

```
val a = Duration::components #P1DT2H62M3.333S#
//    = Duration::Components {
//        day = 1,
//        hour = 3,
//        minute = 2,
//        second = 3.333
//      }
```

2.6.6 Signed functions

The following functions operate on symbolic references to the built-in entity type `Signed` for signed messages.

`Signed::checkSignature : forall a. PublicKey -> Signed a -> Bool`

The function `Signed::checkSignature` checks whether the private key corresponding to the given `PublicKey` was used to signed the mssage.

Examples

```
type Answer : Event { answer: Signed String }
template WaitForAnswer(deepThought: PublicKey) =
    <*> a:Answer where
        Signed::checkSignature deepThought a.answer
```

`Signed::message : forall a. Signed a -> a`

The function `Signed::message` extracts the message from a `Signed`.

Examples

```
type Answer : Event { answer: Signed String }
template WaitForAnswer(deepThought: PublicKey) =
    <*> a:Answer where
        Signed::checkSignature deepThought a.answer
        && Signed::message a.answer = "42"
```

2.6.7 General combinators

These are general functions, which do not pertain to any domain in particular. They are typically used in combination with other functions.

`id : a -> a`

The identity function, which simply returns whatever input it is given.

```
val a = id 14
//    = 14
val b = id "some text"
//    = "some text"
```

`const : a -> b -> a`

`const x` is a function which returns `x` no matter what input it is given, i.e., the constant `x` function.

```
val x = const 30 100
//    = 30
val f = const 0
//    = \_ -> 0
```

`flip : (a -> b -> c) -> b -> a -> c`

Flips the order of the first two arguments of a function.

Examples

```
val a = flip (\x -> \y -> x) 10 20
//    = 20
```

2.6.8 `Tuple`

Tuples of at least 2 values can be represented by the n-ary type constructor `Tuple`. As an example, the type of a pair of values is `Tuple a b`, where `a` and `b` are the types of the first and the second value, respectively. The type of each component of a tuple can be distinct.

We can extract the components of a tuple using pattern matching:

```
val sumIntPair  = \((m, n) : Tuple Int Int) -> m + n
val a = sumIntPair (3, 5)
//    = 8
```

The standard library contains two functions that work with pairs (tuples of two elements): `fst` and `snd`:

`fst : Tuple a b -> a`

The first projection.

Examples

```
val a = fst (0, "s")
//    = 0
```

`snd : Tuple a b -> b`

The second projection.

Examples

```
val a = snd (0, "s")
//    = "s"
```

Triples

```
val sumTriple = \((a, b, c) : Tuple Int Int Int) -> a + b + c
val x = sumTriple (1,2,3)
//    = 1 + 2 + 3 = 6
```

```
val tripleToPair = \((a, b, c) : Tuple Int Int String) -> (a + b, c)
val x = tripleToPair (1, 2, "Hello")
//    = (3, "Hello")
```

Tuples are a convenient way to group pieces of data in a longer computation without a need for explicitly naming the type grouping the data.

```
val f = \(x : Int) -> x + x
val g = \(y : Int) -> y * y
val xs = [1, 2, 3]
val ys = [4, 5, 6]
val combined = List::map (\(x,y) -> (f x, g y)) (List::zip xs ys)
//           = [(2, 16), (4, 25), (6, 36)]
```

2.6.9 Maybe

It is often necessary to model values which can be undefined. This can be done using the type `Maybe a`, where `a` is the type of the value which might be undefined. For example, `Maybe String` can be used to represent the values which are either text strings, or undefined.

The type is defined with two constructors:

```
type Maybe a
  | Some a
  | None
```

So a value of type `Maybe String` will either be of the form `Some "a text string"` (representing the actual string values), or `None` (representing the undefined value).

`maybe : b -> (a -> b) -> Maybe a -> b`

The `maybe` function takes a default value, a function, and a `Maybe` value. The default value is returned if the `Maybe` value is `None`, otherwise the result of applying the function to the value inside the `Some` is returned.

Examples

```
val a = maybe 0 (\x -> x + 5) (Some 2)
//    = 7
val b = maybe 0 (\x -> x + 5) None
//    = 0
```

`fromMaybe : a -> Maybe a -> a`

The `fromMaybe` function extracts the value from a `Maybe`, using the default value for the `None` case.

Examples

```
val a = fromMaybe 0 (Some 5)
//    = 5
val b = fromMaybe 0 None
//    = 0
```

`Maybe::map : (a -> b) -> Maybe a -> Maybe b`

Lift any function to a function in `Maybe`.

Examples

```
val a = Maybe::map (\m -> m * 2) (Some 5)
//    = Some 10
val b = Maybe::map (\m -> m * 2) None
//    = None
```

`Maybe::isSome : Maybe a -> Bool`

Returns `True` if the input is a `Some`, returns `False` otherwise.

Examples

```
val a = Maybe::isSome (Some 2)
//    = True
val b = Maybe::isSome (Some "a text string")
//    = True
val c = Maybe::isSome None
//    = False
```

`Maybe::any : (a -> Bool) -> Maybe a -> Bool`

Returns `True` if, and only if, the given `Maybe` has a value, and this value satisfies the given predicate.

Examples

```
val a = Maybe::any (\n -> n > 4) (Some 5)
//    = True
val b = Maybe::any (\n -> n > 4) (Some 2)
//    = False
val c = Maybe::any (\n -> n > 4) None
//    = False
```

`Maybe::all : (a -> Bool) -> Maybe a -> Bool`

Returns `True` if the given `Maybe` has no value, or if it has a value which satisfies the given predicate.

Examples

```
val a = Maybe::all (\x -> x >= 1) None
//    = True
val b = Maybe::all (\x -> x >= 1) (Some 2)
//    = True
val c = Maybe::all (\x -> x >= 1) (Some 0)
//    = False
```

`Maybe::bind : (a -> Maybe b) -> Maybe a -> Maybe b`

Applies a function that returns a `Maybe` to a `Maybe` value if that value is `Some`. None otherwise

Examples

```
val a = Maybe::bind (\x -> Some (x + 1)) None
//    = None
val b = Maybe::bind (\x -> None) (Some 1)
//    = None
val c = Maybe::bind (\x -> Some (x + 1)) (Some 1)
//    = Some 2
```

2.6.10 `Ordering`

A value of the built-in type `Ordering` is a value denoting that something is "less than", "equal to", or "greater than" something else:

```
type Ordering
  | Less
  | Equal
  | Greater
```

`compareInt : Int -> Int -> Ordering`

`compareFloat : Float -> Float -> Ordering`

`compareDateTime : DateTime -> DateTime -> Ordering`

Examples

```
val a = compareInt 3 6
//    = Less
val b = compareFloat 4.5 4.5
//    = Equal
val c = compareDateTime #2017-12-24T12:00:00Z# #2017-10-11T10:00:00Z#
//    = Greater
```

2.6.11 `List`

A list in CSL is represented by the type `List a`, where `a` is the type of the values in the list. For example, a list of integers, of type `Int`, has the type `List Int`.

A list is either empty or it consists of an element followed by the rest of the list. The empty list is called `Nil`, and the list that contains one element `x` followed by a list `l` is called `Cons x l`. The type of lists is thus:

```
type List a
  | Nil
  | Cons a (List a)
```

Some examples of lists:

```
val oneTwoThree = Cons 1 (Cons 2 (Cons 3 Nil))
val abc = Cons "a" (Cons "b" (Cons "c" Nil))
```

There is support for a *syntactic shorthand* form for writing lists making the following equivalent to the above:

```
val oneTwoThree = [1, 2, 3]
val abc = ["a", "b", "c"]
```

Folding lists with `foldl` and `foldr`

Lists can be *folded* to construct a new value based on the items in the list: Given a function that can combine an accumulating value with the next value in the list and yield a new accumulating value, we may reduce the entire list. An example of this is the sum function. The accumulating value is the sum and the action performed on each element is simply adding it to the running total and returning it. Folding with this function over the entire list then corresponds to adding together all elements. It is written as follows:

```
val sum = \list -> foldl (\(acc:Int) -> \x -> acc + x) 0 list
```

The product of all the integers in a list is written like:

```
val product = \list -> foldl (\(acc : Int) -> \x -> acc * x) 1 list
```

The functions `foldl` and its sibling `foldr` both fold over a list (from the left and right, respectively), combining the elements of the list with a supplied function. Their types are

```
foldl : (b -> a -> b) -> b -> List a -> b
foldr : (a -> b -> b) -> b -> List a -> b
```

The first argument to `foldl` is a function that combines the accumulator value of type `b` with an element of the list of type `a`, producing a new accumulator value of type `b`. The second argument is the starting value for the accumulator – `0` in the example with `sum`; `1` in the `prod` example. The third argument is the list of `a`-values to be folded, and the return value is the final `b` value. For the right-handed sibling `foldr`, the arguments are the same except that the accumulator function takes its argument in the reverse order.

One can think of `foldl` and `foldr` as functions that replace the list constructors `Cons` and `Nil` with, respectively, calls to the supplied accumulator function `f` and the supplied base value (`0` or `1` in the examples above.) If we define the helper function

```
val plus = \(x : Int) -> \(y : Int) -> x + y
```

then folding this over a list as the `sum` function replaces all `Cons` constructors with calls to `sum` and the `Nil` constructor with `0`:

```
val suml = \list -> foldl plus 0 list
val sumr = \list -> foldr plus 0 list
val ns = Cons 1 (Cons 2 (Cons 3 Nil)) // or [1, 2, 3]
val sixl        = suml ns
// {with foldl} = plus (plus (plus 0 1) 2) 3 = 6
val sixr        = sumr ns
// {with foldr} = plus 1 (plus 2 (plus 3 0)) = 6
```

The order of the elements are not changed, but the direction that the `plus` function is applied changes between folding with `foldl` and `foldr`.

Choosing between `foldr` and `foldl`

When we need to solve a problem with a fold, then we need to decide whether to use `foldr` or `foldl`. The reason that we do not always use `foldr` is that `foldl` often has better performance than `foldr`. In most cases we can use the following rule of thumb to choose between them:

1. If the structure of the result of the fold is independent of the structure of the input list, then use `foldl`.
2. If the structure of the result of the fold depends on the structure of the input list, then use `foldr`.

The first situation is typical for cases where the result is an aggregated value. The following example calculates the length and average of a list of `Float`s.

```
val lengthAndAverage = foldl
  (\(l, avg) -> \n -> (l+1, ((Int::toFloat l) * avg + n) / (Int::toFloat (l+1))))
  (0, 0.0)

val a = lengthAndAverage [60.0, 30.0, 120.0]
//    = (3, 70.0)
```

Here the result is independent of the structure of the input list, in the sense that the result type is always `Tuple Float Float`, and the result will be the same no matter how the input list is ordered. Therefore `foldl` is a safe way to get good performance in this example.

The second situation is typical for cases where the output is also a list. The following example filters away small integers from a list.

```
val removeSmallNumbers =
  foldr (\n -> \acc -> if (n >= 10) Cons n acc else acc) Nil

val a = removeSmallNumbers [12, 3, 21, 4]
//    = [12, 21]
```

If we had used `foldl` instead of `foldr` here, then the order of the result would have been reversed.

`List::head : List a -> Maybe a`

Returns the head (i.e. the first element) of the list, if any. Otherwise returns `None`.

Examples

```
val a = List::head [0]
//    = Some 0
val b = List::head []
//    = None
```

List::headOrDefault : a -> List a -> a

Returns the head (i.e. the first element) of a list, if any. Otherwise returns the given default value.

Examples

```
val a = List::headOrDefault 42 [0]
//    = 0
val b = List::headOrDefault 42 []
//    = 42
```

List::tail : List a -> Maybe (List a)

Returns the tail of a list (i.e. what remains when stripping away the first element), if any. Otherwise, if the list is empty, return `None`.

Examples

```
val a = List::tail [0, 1]
//    = Some [1]
val b = List::tail [0]
//    = Some []
val c = List::tail []
//    = None
```

List::sort : (a -> a -> Ordering) -> List a -> List a

We can sort a list of elements if we know how to impose an order on them. The function `List::sort` accepts a function that orders elements and a list of elements and produces a sorted list:

Examples

```
val sortAscending = List::sort compareInt
val sortDescending = List::sort (flip compareInt)
val xs = [2, 3, 1]
val xsAscending = sortAscending xs
//              = [1, 2, 3]
val xsDescending = sortDescending xs
//               = [3, 2, 1]
```

List::length : List a -> Int

The `List::length` function returns the number of elements in a list.

Examples

```
val a = List::length ["a", "b", "c"]
//    = 3
val b = List::length []
//    = 0
```

List::isEmpty : List a -> Boolean

Returns `True` if the list is empty. `False` otherwise.

Examples

```
val a = List::isEmpty ["a", "b", "c"]
//    = False
val b = List::isEmpty [ [] ]
//    = False
val c = List::isEmpty []
//    = True
```

List::map : (a -> b) -> List a -> List b

To transform all elements in a list and produce a list with the transformed elements, we use the function `List::map` to apply some function `f` on all elements in a list.

`List::map` takes a function that can convert elements of type `a` into elements of type `b`, a list of `a` elements, and it produces a list of `b` elements by applying the supplied function to each element in the list.

Examples

```
// type of addOne is Int -> Int
val addOne = \x -> x + 1
// type of incrementList is List Int -> List Int
val incrementList = List::map addOne
val ms = [1, 2, 3]
val msInc = incrementList ms
//        = [addOne 1, addOne 2, addOne 3]
//        = [2, 3, 4]
```

List::mapMaybe : (a -> Maybe b) -> List a -> List b

The `List::mapMaybe` function is a version of `List::map` which takes a function into `Maybe b` (also known as a *partial function*), and throws away the undefined values (i.e. the `None`s).

Examples

```
val a = List::mapMaybe (\n -> if (n > 100) (Some n) else None)
                       [140, 40, 103]
//    = [140, 103]

val b = List::mapMaybe id [Some "a", None, Some "b"]
//    =  ["a", "b"]
```

`List::filter : (a -> Bool) -> List a -> List a`

The `List::filter` function takes a predicate and a list, and returns a list consisting of the elements of the input list which satisfies the predicate.

Examples

```
val a = List::filter (\n -> n < 10) [10, 1, 2, 100]
//    = [1, 2]
```

`List::zipWith : (a -> b -> c) -> List a -> List b -> List c`

The `List::zipWith` function generalises `List::map` to binary functions. It takes a binary function and two lists as arguments, and returns a list resulting from applying the function pairwise on the elements of the lists. The resulting list always has the same length as the shortest input list, i.e., the last elements of the longest list are ignored.

Examples

```
val a = List::zipWith (\(m: Int) -> \(n : Int) -> m + n)
                      [4, 5]
                      [10, 20]
//    = [14, 25]

val b = List::zipWith (\(m : Int) -> \(n : Int) -> m + n)
                      [4, 5]
                      [10]
//    = [14]

val c = List::zipWith (\(m : Int) -> \(n : Int) -> m + n)
                      [4]
                      [10, 20]
//    = [14]
```

`List::any : (a -> Bool) -> List a -> Bool`

Given a predicate and a list, `List::any` returns `True` if, and only if, there exists an element in the list which satisfies the predicate.

Examples

```
val a = List::any (\n -> n > 4) [2, 10]
//    = True
val b = List::any (\n -> n > 4) [2, 0]
//    = False
val c = List::any (\n -> n > 4) []
//    = False
```

`List::all : (a -> Bool) -> List a -> Bool`

Given a predicate and a list, `List::all` returns `True` if, and only if, all elements in the list satisfy the predicate.

Examples

```
val a = List::all (\n -> n > 4) [5, 6]
//    = True
val b = List::all (\n -> n > 4) [5, 3]
//    = False
val c = List::all (\n -> n > 4) []
//    = True
```

List::first : (a -> Bool) -> List a -> Maybe a

Returns the first element in the list which satisfies the predicate, if any.

Examples

```
val a = List::first (\n -> n > 4) [3, 42, 100]
//    = Some 42

val b = List::first (\n -> n > 4) [3, 2, 1]
//    = None
```

List::last : (a -> Bool) -> List a -> Maybe a

Returns the last element in the list which satisfies the predicate, if any.

Examples

```
val a = List::last (\n -> n > 4) [3, 42, 100]
//    = Some 100

val b = List::last (\n -> n > 4) [3, 2, 1]
//    = None
```

List::append : List a -> List a -> List a

Appends two lists.

Examples

```
val a = List::append ["a"] ["b"]
//    = ["a", "b"]

val b = List::append [] a
//    = a

val c = List::append a []
//    = a
```

List::concat : List (List a) -> List a

Flattens a list of lists into one list, by appending them to each other.

Examples

```
val a = List::concat [ [1, 2], [3], [4] ]
//    = [1, 2, 3, 4]
```

List::concatMap : (a -> List b) -> List a -> List b

Maps a list-returning function over a list and concatenates the results.

Examples

```
val a = List::concatMap (\n -> [n, n+1, n+2]) [1, 2, 3]
//    = [1, 2, 3, 2, 3, 4, 3, 4, 5]
```

List::reverse : List a -> List a

Reverses a list.

Examples

```
val a = List::reverse [1, 2, 3]
//    = [3, 2, 1]
```

List::take : Int -> List a -> List a

Given an integer, m, and a list, `List::take` returns the first m elements of the list. If the list has fewer than m elements, the whole list is returned.

Examples

```
val a = List::take 2 ["a", "b", "c"]
//    = ["a", b"]

val b = List::take 2 ["a"]
//    = ["a"]
```

List::drop : Int -> List a -> List a

Given an integer, m, and a list, `List::drop` throws away the first m elements of the list, and returns the rest. If the list has fewer than m elements, the empty list is returned.

Examples

```
val a = List::drop 2 ["a", "b", "c"]
//    = ["c"]

val b = List::drop 1 ["a"]
//    = []
```

(continues on next page)

(continued from previous page)

```
val c = List::drop 1 []
//    = []
```

2.6.12 Map

CSL has the built-in type `Map k v` which is a mapping between keys of type `k` and values of type `v`. For example, a map of type `Map Int MySumType` is a mapping between integers and some custom type `MySumType`.

Maps can only be constructed using the functions in the standard library – there are no constructors for `Map`. This also means that it is not possible to do pattern-matching on `Map`-values like it is with, e.g., `List`.

The simplest way to create a `Map` is by using the `Map::fromList` function from the standard library:

```
val myMap = Map::fromList [(1, "One"), (2, "Two"), (3, "Three")]
val oneString = Map::lookup 1 myMap
//            = Some "One"
val fourString = Map::lookup 4 myMap
//             = None
```

Maps are **immutable**: when you insert an element a new map is created and returned:

```
val myNewMap = Map::insert 4 "Four" myMap
val fourString2 = Map::lookup 4 myNewMap
//              = Some 4
val fourString3 = Map::lookup 4 myMap
//              = None (Unchanged from above)
```

Note: A map is **unordered**: when you iterate over the elements in a map using `Map::fold` you should not make any assumptions about the order the elements are returned. Elements will be returned in *some* order but this order should not be assigned any special significance.

`Map::empty : Map k v`

Returns a new empty map.

`Map::insert : k -> v -> Map k v -> Map k v`

Returns a new map that contains, in addition to the keys already present in the argument map, the given key/value.

Examples

```
val noElms = Map::empty
val oneElm = Map::insert 1 "One" noElms
val twoElms = Map::insert 2 "Two" oneElm
```

`Map::remove : k -> Map k v -> Map k v`

Returns a new map with the given key removed.

Examples

```
val myMap = Map::insert 1 "One" Map::empty
val myEmptyMap = Map::remove 1 myMap
//             = Map::empty
val myMap2 = Map::remove 2 myMap
//         = myMap
```

`Map::lookup : k -> Map k v -> Maybe v`

Returns the value at the given key, if the key is in the map. Otherwise it returns `None`.

Examples

```
val myMap = Map::insert 1 "One" Map::empty
val oneVal = Map::lookup 1 myMap
//         = Just "One"
val twoVal = Map::lookup 2 myMap
//         = None
```

`Map::fold : (k -> v -> a -> a) -> a -> Map k v -> a`

Fold over the key/value pairs in a map. As mentioned above, maps are *unordered*.

Examples

```
val sumKeysAndVals = Map::fold (\k -> \v -> \counter -> k + v + counter) 0
val x = sumKeysAndVals (Map::fromList [(1, 10), (2, 20)])
//    = 1 + 10 + 2 + 20
```

`Map::fromList : List (Tuple k v) -> Map k v`

Constructs a map that contains the elements in the given list.

Examples

```
val prices = Map::fromList [("hammer", 10), ("saw", 12), ("axe", 15)]
```

2.6.13 `Event`

The `Event` type represents events that can be applied to contracts. The `Event` type is built-in. The definition is given below.

```
type Event {
  agent : Agent,
  timestamp : DateTime
}
```

Contract

The built-in abstract type `Contract` represents a contract's id. Values of this type cannot be constructed directly in CSL syntax, so whenever such a value is needed it must be supplied to the runtime system directly – either as an instantiation argument to a template or as an argument to a report.

2.6.14 Built-in functions and types

The following types are built-in and not defined in the standard library:

- `Int`
- `String`
- `Float`
- `DateTime`
- `Duration`
- `Record`
- `Tuple`
- `Bool`
- `List`
- `Ordering`
- `Map`
- `Maybe`
- `DateTime::DayOfWeek`
- `DateTime::Components`
- `DateTime::PaymentFrequency`
- `DateTime::DayCountFraction`
- `Duration::Components`
- `Event`
- `Agent`
- `Contract`
- `Signed`
- `PublicKey`

2.6.15 CSL standard library source code

```
// CSL standard library
//////////////////////////

////////
// Errors
////////

/// error : RuntimeError -> a
val error = prim__error

//////////////////////
```

(continues on next page)

(continued from previous page)

```
// General combinators
//////////////////////

/// The identity function.  'id' simply returns the input.
///
/// Examples:
/// id 4 = 4
/// id "a" = "a"
///
/// id : a -> a
val id = \x -> x

/// The constant function.  'const x' is a function which returns 'x', no matter
↪what input it is given.
///
/// Examples:
/// const "a" "b" = "a"
///
/// const : a -> b -> a
val const = \x -> \_ -> x

/// Flips the order of the arguments for a binary function.
///
/// Examples:
/// flip (\x -> \y -> x) "a" "b" = "b"
///
/// flip : (a -> b -> c) -> b -> a -> c
val flip = \f -> \y -> \x -> f x y


//////////////////////
// Bool
//////////////////////

/// The 'not' function returns the opposite value of its input.
///
/// Examples:
/// not True = False
/// not False = True
///
/// not : Bool -> Bool
val not =
  \ True -> False
  | False -> True


//////////////////////
// Pair
//////////////////////

/// The first projection.
///
/// Examples:
/// fst (0, "a") = 0
///
/// fst : Tuple a b -> a
val fst = \(x, _) -> x

/// The second projection.
///
/// Examples:
```

(continues on next page)

(continued from previous page)

```
/// snd (0, "a") = "a"
///
/// snd : Tuple a b -> b
val snd = \(_, y) -> y


//////////////////////
// Maybe
//////////////////////

/// The 'maybe' function takes a default value, a function, and a 'Maybe' value.
/// The default value is returned if the 'Maybe' value is 'None', otherwise the
/// result of applying the function to the value inside the 'Some' is returned.
///
/// Examples:
/// maybe 0 (\x -> x + 5) (Some 2) = 7
/// maybe 0 (\x -> x + 5) None = 0
///
/// maybe : b -> (a -> b) -> Maybe a -> b
val maybe = \default -> \f ->
  \ None   -> default
  | Some x -> f x

/// The 'fromMaybe' function extracts the value from a 'Maybe', using the
/// default value for the 'None' case.
///
/// Examples:
/// fromMaybe 0 (Some 5) = 5
/// fromMaybe 0 None = 0
///
/// fromMaybe : a -> Maybe a -> a
val fromMaybe = flip maybe id

module Maybe {
  /// Lift any function to a function in 'Maybe'.
  ///
  /// Examples:
  /// map (\m -> m * 2) (Some 5) = Some 10
  /// map (\m -> m * 2) None = None
  ///
  /// map : (a -> b) -> Maybe a -> Maybe b
  val map = \f ->
    \ None -> None
    | Some x -> Some (f x)

  /// Returns 'True' if the input is a 'Some', returns 'False' otherwise.
  ///
  /// Examples:
  /// isSome (Some 2) = True
  /// isSome None = False
  ///
  /// isSome : Maybe a -> Bool
  val isSome =
    \ None -> False
    | Some _ -> True

  /// Returns 'True' if, and only if, the given 'Maybe' has a value, and this␣
↪value satisfies
  /// the given predicate.
  ///
  /// Examples:
```

(continues on next page)

(continued from previous page)

```
  /// any (\n -> n > 4) (Some 5) = True
  /// any (\n -> n > 4) (Some 2) = False
  /// any (\n -> n > 4) None = False
  ///
  /// any : (a -> Bool) -> Maybe a -> Bool
  val any = \pred -> maybe False pred

  /// Returns 'True' if the given 'Maybe' has no value, or it has a value which
  /// satisfies the given predicate.
  ///
  /// Examples:
  /// all (\x -> x >= 1) None = True
  /// all (\x -> x >= 1) (Some 2) = True
  /// all (\x -> x >= 1) (Some 0) = False
  ///
  /// all : (a -> Bool) -> Maybe a -> Bool
  val all = \pred -> maybe True pred

  /// Apply the function 'f' to the value 'a' if
  /// the Maybe is 'Some a', 'None' otherwise.
  /// 'f' must return a 'Maybe' type.
  ///
  /// Examples:
  /// bind (\x -> Some (x + 1)) None = None
  /// bind (\x -> None) (Some 1) = None
  /// bind (\x -> Some (x + 1)) (Some 1) = Some 2
  ///
  /// bind : (a -> Maybe b) -> Maybe a -> Maybe b
  val bind = \f ->
    \ None -> None
    | Some x -> f x

}


//////////////////////
// Ordering
//////////////////////

/// compareInt : Int -> Int -> Ordering
val compareInt = \(x : Int) -> \y ->
  if (x < y) Less else if (x = y) Equal else Greater

/// compareFloat : Int -> Int -> Ordering
val compareFloat = \(x : Float) -> \y ->
  if (x < y) Less else if (x = y) Equal else Greater

/// compareDateTime : DateTime -> DateTime -> Ordering
val compareDateTime = \(x : DateTime) -> \y ->
  if (x < y) Less else if (x = y) Equal else Greater


//////////////////////
// List
//////////////////////

// forall a b . (b -> a -> b) -> b -> List a -> b
val foldl = prim__foldl
// forall a b . (a -> b -> b) -> b -> List a -> b
val foldr = prim__foldr
```

(continues on next page)

(continued from previous page)

```
module List {

  /// Returns the head (i.e. the first element) of the list, if any.  Otherwise
  /// returns 'None'.
  ///
  /// Examples:
  /// head [0] = Some 0
  /// head [] = None
  ///
  /// head : List a -> Maybe a
  val head =
    \ Nil -> None
    | Cons x _ -> Some x

  /// Returns the head (i.e. the first element) of a list, if any.  Otherwise
  /// returns the given default value.
  ///
  /// Examples:
  /// headOrDefault 42 [0] = 0
  /// headOrDefault 42 [] = 42
  ///
  /// headOrDefault : a -> List a -> a
  val headOrDefault = \default -> \xs -> fromMaybe default (List::head xs)

  /// Returns the tail of a list (i.e. what remains when stripping away the
  /// first element), if any.  Otherwise, if the list is empty, return 'None'.
  ///
  /// Examples:
  /// tail [0, 1] = Some [1]
  /// tail [0] = Some []
  /// tail [] = None
  ///
  /// tail : List a -> Maybe a
  val tail =
    \ Nil -> None
    | Cons _ xs -> Some xs

  /// The 'sort' function takes an ordering function and a list,
  /// and returns an ordered list.
  ///
  /// Examples:
  /// sort compareInt [4, 2, 3]
  ///   = [2, 3, 4]
  ///
  /// forall a . (a -> a -> Ordering) -> List a -> List a
  val sort = prim__List_sort

  /// The 'length' function returns the number of elements in a list.
  /// Examples:
  /// length ["a", "b", "c"] = 3
  /// length [] = 0
  ///
  /// length : List a -> Int
  val length = foldl (\ n -> \ x -> n + 1) 0

  /// 'isEmpty' returns True if a list is empty. False otherwise.
  ///
  /// Examples
  /// isEmpty [1, 2, 3]
  ///    = False
  ///
```

(continues on next page)

(continued from previous page)

```
/// List a -> Boolean
val isEmpty =
  \ Nil -> True
  | _ -> False

/// 'map f xs' is the list obtained by applying the function f on each of the
/// elements in the list 'xs'.
///
/// Examples:
/// map (\n -> n * 2) [1, 2, 3]
///   = [2, 4, 6]
///
/// map : (a -> b) -> List a -> List b
val map = \f -> foldr (\x -> \ys -> Cons (f x) ys) Nil

/// The 'mapMaybe' function is a version of 'map' which takes a partial function,
/// and throws away the undefined value (i.e. the 'None's).
///
/// Examples:
/// mapMaybe (\n -> if (n > 100) (Some n) else None) [140, 40, 103]
///   = [140, 103]
///
/// mapMaybe id [Some "a"] [None, Some "b"]
///   = ["a", "b"]
///
/// mapMaybe : (a -> Maybe b) -> List a -> List b
val mapMaybe = \f -> foldr (\x -> \acc -> maybe acc (\y -> Cons y acc) (f x)) Nil

/// The 'filter' function takes a predicate and a list, and returns a list
/// consisting of the elements of the input list which satisfies the predicate.
///
/// Examples:
/// filter (\n -> n < 10) [10, 1, 2, 100]
///   = [1, 2]
/// filter : (a -> Bool) -> List a -> List a
val filter = \f -> foldr (\y -> \ys -> if (f y) Cons y ys else ys) Nil

/// The 'zipWith' function generalises 'map' to binary functions.  It takes
/// a binary function and two lists as arguments, and returns a list
/// resulting from applying the function pairwise on the elements of the
/// lists.
/// The resulting list always has the same length as the shortest input list.
///
/// Examples:
/// zipWith (\m -> \n -> m + n) [4, 5] [10, 20]
///   = [14, 25]
///
/// zipWith (\m -> \n -> m + n) [4, 5] [10]
///   = [14]
///
/// zipWith (\m -> \n -> m + n) [4] [10, 20]
///   = [14]
///
/// zipWith : (a -> b -> c) -> List a -> List b -> List c
val zipWith = \f ->
  let val step = \a -> \g ->
    \ Nil -> Nil
    | Cons b bs -> Cons (f a b) (g bs)
  in foldr step (\_ -> Nil)

/// The 'zip' function takes two lists as arguments, and returns a
```

(continues on next page)

(continued from previous page)

```
/// list of the elements pairwise together.
/// The resulting list always has the same length as the shortest input list.
///
/// Examples:
/// zip [1,2] ["a", "b"]
///   = [(1, "a"), (2, "b")]
///
/// zip [] ["a"]
///   = []
///
/// zip [(1,"a"), (2,"b")] [True, False]
///   = [((1,"a"),True), ((2,"b"), False)]
///
/// zip : List a -> List b -> List (Tuple a b)
val zip = List::zipWith (\a -> \b -> (a, b))

/// Given a predicate and a list, 'any' returns 'True' if, and only if, there
/// exists an element in the list which satisfies the predicate.
///
/// Examples:
/// any (\n -> n > 4) [2, 10] = True
/// any (\n -> n > 4) [2, 0] = False
/// any (\n -> n > 4) [] = False
///
/// any : (a -> Bool) -> List a -> Bool
val any = \pred -> foldl (\b -> \x -> pred x || b) False

/// Given a predicate and a list, 'all' returns 'True' if, and only if, all
/// elements in the list satisfy the predicate.
///
/// Examples:
/// all (\n -> n > 4) [5, 6] = True
/// all (\n -> n > 4) [5, 3] = False
/// all (\n -> n > 4) [] = True
/// all : (a -> Bool) -> List a -> Bool
val all = \pred -> foldl (\b -> \x -> pred x && b) True

/// Returns the first element in the list which satisfies the predicate,
/// if any.
///
/// Examples:
/// first (\n -> n > 4) [3, 42, 100]
///   = Some 42
///
/// first (\n -> n > 4) [3, 2, 1]
///   = None
///
/// first : (a -> Bool) -> List a -> Maybe a
val first = \pred -> foldr (\x -> \acc -> if (pred x) (Some x) else acc) None

/// Returns the last element in the list which satisfies the predicate,
/// if any.
///
/// Examples:
/// last (\n -> n > 4) [3, 42, 100]
///   = Some 100
///
/// last (\n -> n > 4) [3, 2, 1]
///   = None
///
/// last : (a -> Bool) -> List a -> Maybe a
```

(continues on next page)

(continued from previous page)

```
val last = \pred -> foldl (\acc -> \x -> if (pred x) (Some x) else acc) None

/// Appends two lists.
///
/// Examples:
/// append ["a"] ["b"]
///   = ["a", "b"]
///
/// append [] ys = ys
///
/// append xs [] = xs
///
/// append : List a -> List a -> List a
val append = \xs -> \ys -> foldr (\x -> \acc -> Cons x acc) ys xs

/// Flattens a list of lists into one list, by appending them to each other.
///
/// Examples:
/// concat [[1, 2], [3], [4]]
///   = [1, 2, 3, 4]
///
/// concat : List (List a) -> List a
val concat = foldr List::append Nil

/// Maps a list-returning function over a list and concatenates the results.
///
/// Examples:
/// concatMap (\n -> [n, n+1, n+2]) [1, 2, 3]
///   = [1, 2, 3, 2, 3, 4, 3, 4, 5]
///
/// concatMap : (a -> List b) -> List a -> List b
val concatMap = \f -> foldr (\x -> \acc -> List::append (f x) acc) Nil

/// Reverses a list.
///
/// Examples:
/// reverse [1, 2, 3]
///   = [3, 2, 1]
///
/// reverse : List a -> List a
val reverse = foldl (\xs -> \x -> Cons x xs) Nil

/// Given an integer, m, and a list, 'take' returns the first m elements of
/// the list.  If the list has fewer than m elements, the whole list is
/// returned.
///
/// Examples:
/// take 2 ["a", "b", "c"]
///   = ["a", "b"]
///
/// take 2 ["a"]
///   = ["a"]
///
/// take : Int -> List a -> List a
val take = \(m : Int) -> \xs ->
  let val f = \x -> \rest -> \n ->
    if (n <= 0) Nil
    else Cons x (rest (n - 1))
  in foldr f (const Nil) xs m

/// Given an integer, m, and a list, 'drop' throws away the first m elements
```

(continues on next page)

(continued from previous page)

```
  /// of the list, and returns the rest.  If the list has fewer than m elements,
  /// the empty list is returned.
  ///
  /// Examples:
  /// drop 2 ["a", "b", "c"]
  ///   = ["c"]
  ///
  /// drop 1 ["a"] = []
  ///
  /// drop 1 [] = []
  ///
  /// drop : Int -> List a -> List a
  val drop = \(m : Int) -> \xs ->
    let val f = \x -> \rest -> \n -> \xs1 ->
      if (n <= 0) xs1
      else rest (n - 1) (fromMaybe Nil (List::tail xs1))
    in foldr f (const (const Nil)) xs m xs
}

//////////////////////
// Map
//////////////////////

module Map {
    /// Returns a new empty map.
    ///
    /// empty : Map k v
    val empty = prim__Map_empty

    /// Returns a new map that contains, in addition to the keys
    /// already present in the argument map, the given key/value.
    ///
    /// Examples:
    /// val noElms = Map::empty
    /// val oneElm = Map::insert 1 "One" noElms
    /// val twoElms = Map::insert 2 "Two" oneElm
    ///
    /// insert : k -> v -> Map k v -> Map k v
    val insert = prim__Map_insert

    /// Fold over the key/value pairs in a map.
    ///
    /// Examples:
    /// val myMap = Map::insert 1 "One" Map::empty
    /// val oneVal = Map::lookup 1 myMap
    /// //         = Just "One"
    /// val twoVal = Map::lookup 2 myMap
    /// //         = None
    ///
    /// fold : (k -> v -> a -> a) -> a -> Map k v -> a
    val fold = prim__Map_fold

    /// Returns a new map with the given key removed.
    ///
    /// Examples:
    /// val myMap = Map::insert 1 "One" Map::empty
    /// val myEmptyMap = Map::remove 1 myMap
    /// //             = Map::empty
    /// val myMap2 = Map::remove 2 myMap
    /// //         = myMap
    ///
```

(continues on next page)

(continued from previous page)

```
    /// remove : k -> Map k v -> Map k v
    val remove = prim__Map_remove

    /// Returns the value at the given key, if the key is in the map.
    /// Otherwise it returns ``None``.
    ///
    /// Examples:
    /// val myMap = Map::insert 1 "One" Map::empty
    /// val oneVal = Map::lookup 1 myMap
    /// //         = Just "One"
    /// val twoVal = Map::lookup 2 myMap
    /// //         = None
    ///
    /// lookup : k -> Map k v -> Maybe v
    val lookup = prim__Map_lookup

    /// Constructs a map that contains the elements in the given list.
    ///
    /// Examples:
    /// val prices = Map::fromList [("hammer", 10), ("saw", 12), ("axe", 15)]
    ///
    /// fromList : List (Tuple k v) -> Map k v
    val fromList = \theList ->
      let
        val auxFun = \(key, value) -> \accMap ->
          Map::insert key value accMap
      in
        foldr auxFun Map::empty theList
}

//////////////////////
// DateTime
//////////////////////

module DateTime {
  /// addSeconds : DateTime -> Int -> DateTime
  val addSeconds = prim__DateTime_addSeconds

  /// addDays : DateTime -> Int -> DateTime
  val addDays = \(d : DateTime) -> \(days : Int) ->
    DateTime::addSeconds d (days * 24 * 60 * 60)

  /// components : DateTime -> DateTime::Components
  val components = prim__DateTime_components

  /// dayOfWeek : DateTime -> DateTime::DayOfWeek
  val dayOfWeek = prim__DateTime_dayOfWeek

  /// dayCountFraction : DateTime::DayCountFraction -> DateTime -> DateTime ->␣
↪Float
  val dayCountFraction = prim__dayCountFraction
}

//////////////////////
// Duration
//////////////////////

module Duration {

  /// fromSeconds : Float -> Duration
  val fromSeconds = prim__Duration_fromSeconds
```

(continues on next page)

(continued from previous page)

```
  /// toSeconds : Duration -> Float
  val toSeconds = prim__Duration_toSeconds

  /// addSeconds : DateTime -> Float -> DateTime
  val addSeconds = prim__Duration_addSeconds

  /// diffDateTimes : DateTime -> DateTime -> Duration
  val diffDateTimes = prim__Duration_diffDateTimes

  /// between : DateTime -> DateTime -> Duration
  val between = flip Duration::diffDateTimes

  /// fromMinutes : Float -> Duration
  val fromMinutes = \(x : Float) -> Duration::fromSeconds (x * 60.0)

  /// fromHours : Float -> Duration
  val fromHours = \(x : Float) -> Duration::fromMinutes (x * 60.0)

  /// fromDays : Float -> Duration
  val fromDays = \(x : Float) -> Duration::fromHours (x * 24.0)

  /// addDurations : Duration -> Duration -> Duration
  val addDurations = prim__Duration_addDurations

  /// subDurations : Duration -> Duration -> Duration
  val subDurations = \(x : Duration) -> \(y : Duration) ->
    Duration::fromSeconds (Duration::toSeconds x - Duration::toSeconds y)

  /// addToDateTime : DateTime -> Duration -> DateTime
  val addToDateTime = \(dt : DateTime) -> \(dur : Duration) ->
    Duration::addSeconds dt (Duration::toSeconds dur)

  /// negate : Duration -> Duration -> Duration
  val negate = \(d : Duration) -> Duration::fromSeconds (-1.0 *␣
↪(Duration::toSeconds d))

  /// components : Duration -> Duration::Components
  val components = prim__Duration_components

}


//////////////////////
// Int
//////////////////////

module Int {
  /// Converts an `Int` to a `Float`.
  ///
  /// Examples:
  ///
  /// Int::toFloat 4 = 4.0
  ///
  /// Int -> Float
  val toFloat = prim__Int_toFloat

  /// Converts an `Int` to a `String`.
  ///
  /// Examples:
  ///
```

(continues on next page)

(continued from previous page)

```
  /// Int::toString 4 = "4"
  ///
  /// Int -> String
  val toString = prim__Int_toString
}


//////////////////////
// String
//////////////////////

module String {
  /// Appends two string.
  ///
  /// Examples:
  ///
  /// String::append "Hello, " "World!" = "Hello, World!"
  ///
  /// String -> String -> String
  val append = prim__String_append
}


//////////////////////
// Mathematical functions
//////////////////////

module Math {
  /// Get the absolute value of an 'Int'.
  /// abs : Int -> Int
  val abs  = \x -> if (x < 0) 0 - x else x

  /// Get the absolute value of a 'Float'.
  /// fabs : Float -> Float
  val fabs = \x -> if (x < 0.0) 0.0 - x else x

  /// The power function.
  ///
  /// Examples:
  ///
  /// Math::pow 4.0 3.0 = 64.0
  ///
  /// pow : Float -> Float -> Float
  val pow = prim__Math_pow

  /// Square root.
  ///
  /// Examples:
  ///
  /// Math::sqrt 9.0 = 3.0
  ///
  /// sqrt : Float -> Float
  val sqrt = \x -> Math::pow x 0.5
}


//////////////////////
// Signed data
//////////////////////

module Signed {
```

(continues on next page)

(continued from previous page)

```
  /// Check if a 'Signed' is signed by a given 'PublicKey'
  /// checkSignature : forall a. PublicKey -> Signed a -> Bool
  val checkSignature = prim__Signed_checkSignature

  /// Extract the message contained within a 'Signed'
  /// message : forall a. Signed a -> a
  val message = prim__Signed_message
}
```

2.7 CSL grammar

The following grammar describes the CSL language.

CSL is *whitespace insensitive*. This means that the syntax allows an arbitrary amount of whitespace and comment lines around tokens (written as `"token"`). Anything on a line between `//` and the newline character is treated as a comment and ignored.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; The top-level entry point to the grammar

csl-unit = declaration*

declaration = type-declaration
            | value-declaration
            | contract-declaration
            | template-declaration
            | module-declaration



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; modules

module-declaration = "module" upper-case-identifier "{" declaration*  "}"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; types

type-declaration = "type" (sum-type-declaration | record-type-declaration)

sum-type-declaration = upper-case-identifier
                       lower-case-identifier*
                       ("|" upper-case-identifier type-atom*)+

record-type-declaration = upper-case-identifier [ ":" type ]
                          "{" [ record-field-declaration ( "," record-field-
↪declaration)* ] "}"
record-field-declaration = lower-case-identifier ":" type

type = type-atom
     | type-application
     | type "->" type ; function type

type-application = type-name
                 | type-application type

type-name = qualifier upper-case-identifier ; type name

type-atom = lower-case-identifier ; type variable
```

(continues on next page)

(continued from previous page)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; expressions

value-declaration = "val" val-bindings

val-bindings =
  (val-binding "with")* val-binding

val-binding =
  [ "report" ]
  ; Note: patterns other than 'x : T' are only allowed
  ; if the binding occurs within an expression.
  pattern
  "="
  expression

expression = expression expression
           | expression BINARY_OPERATOR expression
           | "(" (expression ",")* expression ")"
           | if-expression
           | type-case-expression
           | function-expression
           | let-expression
           | record-expression
           | qualifier lower-case-identifier ; variable
           | literal
           | qualifier upper-case-identifier ; constructor
           | "[" [ (expression ",")* expression ] "]" ; list
           | expression ":>" type ; upcast operator

if-expression =
  "if" "(" expression ")" expression "else" expression

type-case-expression =
  "type" lower-case-identifier "=" expression "of"
  "{"
  (type-atom "->" expression ";")*
  "_" "->" expression
  "}"

function-expression =
  "\" pattern "->" expression
  [ "|" pattern "->" expression
    ("|" pattern "->" expression)* ]

let-expression =
  "let" ("val" pattern "=" expression)+
  "in" expression

record-expression =
  type "{" [ "use" expression "with" ] (lower-case-identifier "=" expression ",")*
↪"}"


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; patterns

pattern =
  (pattern-atom | pattern-apply) ("as" lower-case-identifier)* (":" type)*
```

(continues on next page)

(continued from previous page)

```
pattern-apply = qualifier upper-case-identifier pattern-atom*

pattern-atom = "_"
             | "(" (pattern ",")* pattern ")"
             | record-pattern
             | qualifier upper-case-identifier ; constructor
             | lower-case-identifier ; variable
             | literal

record-pattern =
  "?" type
  "{"
  [ (lower-case-identifier "=" pattern ",")*
    lower-case-identifier "=" pattern ]
  "}"


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; contracts

template-declaration = rec-template-declaration
                     | meta-template-declaration

rec-template-declaration = "template" ["rec"] rec-template-bindings
rec-template-bindings =
  (rec-template-binding "with")* rec-template-binding
rec-template-binding =
  [ "entrypoint" ]
  [ "[" "]" ] ; Optional empty list of contract parameters
  upper-case-identifier
  "("
  [ (pattern ",")* pattern ]
  ")" "=" contract

meta-template-declaration = "template" meta-template-bindings
meta-template-bindings =
  (meta-template-binding "with")* meta-template-binding
meta-template-binding =
  "[" (lower-case-identifier ",")* lower-case-identifier "]" ; List of contract␣
↪parameters
  upper-case-identifier
  "("
  [ (pattern ",")* pattern ]
  ")" "=" contract

contract-declaration = "contract" contract-bindings
contract-bindings =
  (contract-binding "with")* contract-binding
contract-binding =
  lower-case-identifier
  "="
  contract

local-declaration = template-declaration
                  | contract-declaration
                  | value-declaration

contract = "success"
         | "failure"
         | contract "and" contract
```

(continues on next page)

(continued from previous page)

```
          | contract "or" contract
          | contract "then" contract
          | "(" contract ")"
          | application-contract
          | variable-contract
          | prefix-contract
          | let-contract

application-contract =
  qualifier upper-case-identifier
  [ "[" [ (contract ",")* contract ] "]" ]
  "(" [ (expression ",")* expression ] ")"

variable-contract = lower-case-identifier

prefix-contract =
  "<" ( "*" | expression) ">"
  [ lower-case-identifier ":" ] type
  [ "where" expression ]

let-contract =
  "let" (local-declaration)+
  "in" contract


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Basic grammar components

lower-case-identifier = /[a-z][a-zA-Z0-9_]*/
upper-case-identifier = /[A-Z][a-zA-Z0-9_]*/

literal = float-literal
        | integer-literal
        | duration-literal
        | date-time-literal
        | string-literal

float-literal = /[0-9]+\.[0-9]+([eE][+-]?[0-9]+)?/
integer-literal = /[0-9]+/

fixed3 = /[0-9]+(\.[0-9]{1,3})?/
duration-literal =
  "#"
  /[+-]?P/ (duration-time | duration-date duration-time?)
  "#"

duration-date = fixed3 "D"
duration-time = "T" (duration-hours | duration-minutes | duration-seconds)
duration-hours = fixed3 "H" (duration-minutes | duration-seconds)?
duration-minutes = fixed3 "M" duration-seconds?
duration-seconds = fixed3 "S"

; These two are defined elsewhere.
date-time-literal =
  "#"
  ( ( /[0-9]{4}/
    | /[0-9]{4}-[0-9]{2}/
    | /[0-9]{4}-[0-9]{2}-[0-9]{2}/
    | /[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}/
    | /[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}/
    | /[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}:[0-9]{2}/
```

(continues on next page)

(continued from previous page)

```
    | /[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}:[0-9]{2}\.[0-9]{0,3}/
    )
    [ /[+-][0-9]{4}|[+-][0-9]{2}:[0-9]{2}|Z/ ]
  ) "#"
string-literal = haskell-style-string ; see Haskell 98 report

; A sequence of zero or more qualifer names interspersed with '::'.
qualifier = (upper-case-identifier "::")*

BINARY_OPERATOR = "+" | "-" | "*" | "/"
                | "<=" | ">=" | "<" | ">"
                | "=" | "&&" | "||"
```

CHAPTER 3

Tooling

In this chapter we introduce the code generation tool `sic` and describe how to use it:

3.1 The `sic` boilerplate generator

`sic` is a tool for generating the boilerplate code that makes it possible to seamlessly interact with a CSL contract from another language. Currently it supports two target languages: Kotlin and TypeScript. Because Kotlin is a JVM-based language `sic` indirectly supports other such languages like Java. Likewise, TypeScript is a language that is compiled to JavaScript and is designed to be interoperable with JavaScript code, so `sic` indirectly supports JavaScript as well.

3.1.1 Overview

When you have written a CSL contract and want to integrate it in a larger application you need some way of communicating with the system responsible for running the CSL contract. This can be done using the public API and one of the API clients, however, this means that you would have to take care of sending the right JSON-encoded data yourself, with no possibility of help from your language's type checker.

`sic` generates mappings from the CSL data types to native data types in the target language, meaning that you can get help from the target language's type checker and IDE support when building values of these types. Moreover, reports, events and entry points (templates) in the contract are mapped to suitable constructs in the target language. Hence, instead of communicating directly using the Deon API, you can instead use specialized functions that use the generated native data types as input and output types, and which takes care of (de)serializing to the JSON format expected by Deon's API. This makes it easier to work with the contract as you get the support that you would otherwise get when working in the target language. On top of that, it makes it dramatically less time-consuming to make changes to the CSL contract, as you will get the updated mappings for free by re-running `sic`, and any inconsistencies in the way you use the generated functions will be immediately caught by the target language's typechecker.

Given a CSL contract, `sic` will generate the following components in the target language:

Data type definitions Each data type in the contract will be mapped to a data type in the target language.

Report functions For each report in the contract a corresponding function for invoking that report is created.

Event application functions For each `Event` type in the contract a function for applying that event to a contract is created.

Contract instantiation functions For each top-level template in the contract a function for instantiating a contract from that template is created.

Usage

`sic` is a command-line tool that works on Windows, MacOS, and Linux:

```
$ sic --help
sic <VERSION>

Usage: sic [-V|--version] [-n|--namespace NAMESPACE] [-t|--target ARG]
           [--stdlib PATH] [-w|--write] [-d|--destination PATH] FILES

Available options:
  -V,--version             Print version information
  -h,--help                Show this help text
  -n,--namespace NAMESPACE Namespace to put generated code in
  -t,--target ARG          Target language (default: Kotlin)
  --stdlib PATH            Use alternative CSL standard library
  -w,--write               Write generated source files to disk instead of just
                           printing them to stdout
  -d,--destination PATH    Root directory for generated source
                           files (default: "generated")
```

We shall use the contract "`sic1.csl`" for demonstration of how to use `sic` and for illustrating key points about the structure of the generated code:

```
type CustomerType
    | Regular Int
    | OneTime

type Address {
    street: String,
    number: Int,
    floor: Int
}

type Customer {
    name: String,
    age: Int,
    address: Address,
    customerType: CustomerType
}

type AddCustomer : Event {
    id : Int,
    customer : Customer
}

//  sum : List Int -> Int
val sum = \ints -> foldl (\(x : Int) -> \y -> x + y) 0 ints

//  sumCustomerAge : List Customer -> Int
val sumCustomerAge =
  \(customers : List Customer) ->
    sum (List::map (\(c : Customer) -> c.age) customers)

template rec Shop(total) = <*> a: AddCustomer
  where a.id = total then Shop(total + 1)
```

3.1.2 Using Kotlin as the target language

The default target language of `sic` is Kotlin, and the default behaviour is to write the generated code to standard output. Thus, when we run the command

```
$ sic sic1.csl
```

it will print a bunch of Kotlin code to the terminal. If we pass the flag `--write` to `sic` it will write the code to disk:

```
$ sic --write contract.csl
Generating interface for sic1.csl
Wrote file generated/com/deondigital/api/contract/sic1/sic1.kt
Wrote file generated/com/deondigital/api/contract/sic1/sic1.csl.kt
Wrote file generated/com/deondigital/api/contract/sic1/preamble.kt
Wrote file generated/com/deondigital/api/contract/sic1/builtins.kt
Wrote file generated/com/deondigital/api/contract/sic1/RestContractOperations.kt
Wrote file generated/com/deondigital/api/contract/sic1/Entities.kt
Wrote file generated/com/deondigital/api/contract/sic1/fromValue.kt
Wrote file generated/com/deondigital/api/contract/sic1/InstanceDispatcher.kt
Wrote file generated/com/deondigital/api/contract/sic1/InstanceDispatcher.kt
Wrote file generated/com/deondigital/api/contract/sic1/ReportService.kt
Wrote file generated/com/deondigital/api/contract/sic1/ContractService.kt
```

We see here that one CSL contract is represented as two Kotlin source files in the package `com.deondigital.api.contract.sic1`. The generated code will be put into the root directory `generated/`. Both the package name and the root directory can be changed with the flags `--namespace` (`-n`) and `--destination` (`-d`), respectively.

The first file, `sic1.kt`, contains all the interface and data definitions that enable us to interact with the CSL contract from Kotlin in a convenient manner. The second file, `sic1.csl.kt`, is an embedding of the contract source in Kotlin.

Data types

The file `sic1.kt` will contain, amongst many other things, the definitions of the following data types:

```
sealed class CustomerType : Convertible {
  /* ... */
  data class Regular(val field0: Int) : CustomerType() {
    /* .. */
  }
  object OneTime : CustomerType() {
    /* .. */
  }
}
open class Address(
             val street: String,
             val number: Int,
             val floor: Int) : Record() {
  /* ... */
}
open class Customer(
             val name: String,
             val age: Int,
             val address: Address,
             val customerType: CustomerType) : Record() {
  /* ... */
}
class Shop(/* ..., */
           val total: Int) : ContractInstance {
```

(continues on next page)

(continued from previous page)

```
  companion object {
    fun instantiate(/*...,*/ total: Int) =
      /* ... */
    fun getInstance(ops: ContractOperations, contractId: ContractId) =
      /* ... */
  }
}
```

We have left out a lot of details here, but the snippet demonstrates how a sum type in CSL is converted to a `sealed class` in CSL with a subclass for each constructor while a CSL record type is converted to an `open class`. The names of parameters of an `open class` match the names in the CSL record. Base types such as `Int` and `String` are represented by their native counterparts in Kotlin: `kotlin.String` and `kotlin.Int`. Furthermore, the `Shop` template entrypoint is the basis of a class with the same name that holds a static method `instantiate`. When used, the `instantiate` method gives a `Shop` that can be used to query the state of the running contract and to apply events. It also provides access to instantiation arguments for the contract.

Reports

The CSL reports are converted to functions in the target language with appropriate types. That is, the input and output types are mappings from the CSL type to the target language type as described in the above section.

In our generated Kotlin code, we find the following class:

```
open class ReportService(
  private val reportCaller : (
      String,
      String,
      List<com.deondigital.api.Value>
  ) -> java.util.concurrent.CompletableFuture<com.deondigital.api.Value>,
  private val declarationId : String
) : /* ... */ {

  fun sum(ints: List<Int>) : Int = /* implementation */
  fun sumCustomerAge(customers: List<Customer>) : Int = /* implementation */
}
```

The class `com.deondigital.api.contract.sic1.ReportService` declares two functions, one for each of the CSL reports in `sic1.csl`. Input and output types of the functions are mapped from the corresponding CSL types; note that the CSL `List a` type is mapped to Kotlin/Java's `List<T>` type. To instantiate the class one must provide two arguments:

1. A `reportCaller` which is the function used to issue the actual report call to the Deon API. The parameters of this function match the ones taken by the `DeonApiClient::postReport` function, so in the usual case you can just supply the `postReport` method from a `DeonApiClient` that is set up with the right URL.
2. A `declarationId` – this is the id of the declaration on the ledger with the CSL sources, as returned by the wrapping *RESTContractOperation*.

The following snippet illustrates how one would typically run a report with this interface:

```
val apiClient = DeonAPIClient(API_URL) // Connect to the ledger
val declarationId = RestContractOperations("myIdentity", apiClient).declarationId /
↪/ Wrap the client
val r = ReportService(apiClient::postReport, declarationId)
val s = r.sumCustomerAge(listOf(
    Customer("bob", 42, Address("Main st.", 1, 5), CustomerType.OneTime),
    Customer("alice", 30, Address("Main st.", 10, 2), CustomerType.Regular(1)))
) // == 72
```

The astute reader will have noticed that the class `ReportService` is an `open class` and that it itself is a subclass of another class. When `sic` generates code for *multi-file projects* the report service classes inherit from one another, reflecting the project structure in the `deon-project`. For standalone CSL files the report service just inherits from an empty abstract base class.

Contract instantiation

Every top-level template declaration in a CSL contract represents a possible instantiation point of a contract in the system. The generated Kotlin code for `sic1.csl` contains the following class:

```
class Shop(/* ..., */
           val total: Int) : ContractInstance {
  companion object {
    fun instantiate(`$ops`: ContractOperations, /*...,*/ total: Int) =
      /* ... */
    fun getInstance(ops: ContractOperations, contractId: ContractId) =
      /* ... */
  }
}
```

In order to make it possible to use any backend, the `instantiate` function takes a `ContractOperations` object that describes how contract state is being managed. Moreover, all instantiation functions take the same parameters as the CSL templates (mapped to the Kotlin type), plus two additional optional parameters:

1. `MetaArgs` that can be used to supply additional information about the contract that will be instantiated, such as which peers should be used in the Corda backend and which name will be given to the contract instance
2. `timeProvider` to manage how event timestamps are set.

To create a `ContractOperations` object in the following example, we use a REST backend.

```
val apiClient = DeonAPIClient(API_URL)
// Construct a ContractOperations for a REST backend
val ops = RestContractOperations("myIdentity", apiClient)
// Instantiate the template 'Shop'
val contract1 = Shop.instantiate(ops, 42)
// Instantiate a new contract from the template 'Shop'
val contract2 = Shop.instantiate(ops, 11)
// Instantiate yet another contract from the 'Shop'
// template, but give the instance a custom name
val namedContract = Shop.instantiate(ops, 47, MetaArgs(name = "shopContract47"))
```

Event application

Every subtype of `Event` in the contract gets mapped to a function that applies an event of that type to a running contract. Any fields that the event record might have is represented as a parameter to the event application function.

The classes created with `.instantiate(...)` exposes a field `applyEvent` that allows application of events as functions, i.e. `AddCustomer()`. Because the `AddCustomer` event record contains two fields in addition to the fields in `Event`, `id : Int` and `customer : Customer`, the Kotlin function accepts two parameters corresponding to the fields. The return type is parameterized like it was the case for contract instantiation from Kotlin.

The example snippet below uses the contract `contract1` instantiated above:

```
// Now we can apply two 'AddCustomer' events on the contract:
contract1.applyEvent.AddCustomer(0, Customer("bob",
                                 42,
                                 Address("Main st.", 1, 5),
```

(continues on next page)

(continued from previous page)

```
                                    CustomerType.OneTime));
contract1.applyEvent.AddCustomer(1, Customer("alice",
                                    30,
                                    Address("Main st.", 10, 2),
                                    CustomerType.Regular(1)));
```

Targeting R3 Corda

`sic` provides an additional Kotlin target named `Corda`. Using this target, `sic` will, in addition to creating the usual Kotlin files, also create a `contractOps.kt` file. This file contains the class `ContractOps` that provides an interface to contracts on the R3 Corda Distributed Ledger (https://www.corda.net/). To construct an instance of the class, a `CordaRPCOps` and a `ContractHandler` is needed. The former is obtained from creating a connection to a Corda Node; and the latter is available from the `csl-cordapp` package:

```
dependencies {
  compile group: 'com.deondigital', name: 'csl-cordapp', version: 'v0.58.0-SNAPSHOT
↪' // Insert appropriate CSL version here
}
```

The plugin `csl-cordapp` provides a default anonymized implementation of the `ContractHandler` interface, but it is possible to implement a custom implementation.

For the complete documentation of CorDapps, we refer to https://docs.corda.net/building-a-cordapp-index.html.

Gradle plugin

The `sic` boilerplate generator comes with a plugin that supports both `Kotlin` and `Corda` gradle projects. To use it, the root gradle project will need the following additions:

- `settings.gradle` must, in addition to any other subprojects, include:

```
include 'flows'
include 'states'
```

- `build.gradle` must include:

```
buildscript {
  dependencies {
    classpath "com.deondigital:gradle-sic-plugin:v0.58.0-SNAPSHOT"
  }
}
apply plugin: 'com.deondigital.gradle.sic-plugin'

csl {
  destinationDir = 'generated/sic/'
  cslDir = 'src/main/csl/'
  relocate = true
  cslVersion        = 'v0.58.0-SNAPSHOT'
  cordapp {
    cordappVersion     = '0.1.0'
    cordappName        = 'example'
    cordappPackage     = 'org.foocorp'
    cordappVendor      = 'FooCorp'
    resourceApiVersion = '0.1.0'
  }
}
```

In the `csl {}` configuration block `cslVersion = 'v0.58.0-SNAPSHOT'` refers to the version of the `sic` Gradle plugin in use. In `csl.cordapp {}` configuration block, `resourceApiVersion = '0.1.0'` specifies which version of the Deon Digital resource API to use.

The following properties are needed to download the relevant dependencies from Deon Digital's Nexus repository: `NEXUS_USER` and `NEXUS_PASSWORD`. It is recommended to supply these via the command line as build arguments or have them in a Gradle settings file that is not checked into version control.

The `relocate` parameter is used to do relocation of the `com.deondigital.cordapp` packages which contain Corda contracts. This is to support multiple deployments of `sic` generated CorDapps in the same Corda network. When `relocate` is `true`, package `com.digital.cordapp` is relocated to `<cordappPackage>-<cordappName>`.

The plugin provides the following tasks (use `gradle flows:tasks` and `gradle states:tasks` for an overview):

Task	Description
generateKotlin	Generate Kotlin code with `sic`
generateAST	Generate the abstract syntax tree for the input CSL file and the CSL standard library
assembleCorda	Assemble Corda projects for both 'states' and 'flows'
flows:generateDeonCorDappFlows	Generate `sic` 'flows' output, compile, relocate, and package jar file
states:generateDeonCorDappStates	Generate `sic` 'states' output and package jar file

There are a number of additional internal tasks exposed by the plugin, but they should generally not be used in most applications.

For CorDapp projects, the following dependencies are needed to wire the output from `flows` and `states` into any root- or sub project requiring the generated code:

```
dependencies {
  compile project (':flows')
  compile project (':states')
}
```

If you need IDE support for working with generated classes, add the following to the relevant project. This will make it possible for, e.g., IntelliJ IDEA to show the generated and relocated code in the JAR files.

```
repositories {
  flatDir {
      dirs "$rootDir/flows/build/libs/"
  }
}

dependencies {
  // A runtime dependency to exclude it from the shadowed jar
  if (rootProject.file("$rootDir/flows/build/libs/flows-0.1.0.jar").exists()) {
      compileOnly name: "flows-0.1.0"
  }
}
```

The version part of the dependency must match the version specified in `csl.cordapp.cordappVersion`.

For a more thorough example of building a CorDapp, we refer to https://github.com/corda/samples where the sub projects `flows` and `states` correspond to the `workflows-kotlin` and `contracts-kotlin` respectively. These examples also contain documentation of how to use `CordFormation` to build and deploy CorDapps.

For a complete example of using the plugin to build CorDapps, see https://gitlab.deondigital.com/open/csl-cordapp-examples.

3.1.3 Using TypeScript as the target language

To instruct `sic` to generate TypeScript mappings for the CSL contract, run it with the flag `--target TypeScript`:

```
$ sic --write --target TypeScript sic1.csl
Generating interface for sic1.csl
Wrote file generated/./sic1.ts
Wrote file generated/./sic1.csl.ts
Wrote file generated/./preamble.ts
Wrote file generated/./builtins.ts
```

We get three files: `sic1.ts` contains the data definitions and interfaces for our CSL contract, `sic1.csl.ts` contains the CSL source, and `builtins.ts` and `preamble.ts` contains definitions that are not tied to the particular contract.

Data types

Amongst the definitions in the file `sic1.ts` are the mappings of the CSL types:

```
export type CustomerType
 = Regular
 | OneTime
export class Regular {
 constructor(field0 : number) { /* ... */ }
 /* ... */
}
export class OneTime {
 constructor() { /* ... */ }
}
export class Address extends Record {
 street : string
 _number : number
 floor : number
 constructor (street : string, _number : number, floor : number) { /* ... */ }
 /* ... */
}

export class Customer extends Record {
 name : string
 age : number
 address : Address
 customerType : CustomerType
 constructor (name : string, age : number, address : Address, customerType :␣
↪CustomerType) {
   /* ... */
 }
 /* ... */
}
```

These definitions allow us to work in TypeScript directly with, e.g., a `Customer` class with the field `name` as a native `string`. Note that, because "`number`" is a reserved word in TypeScript, the field `number` in the CSL contract has been renamed to `_number` in the TypeScript embedding.

Reports

The generated TypeScript code contains the class:

```
// In sic1.ts
export class Reports extends builtins.Reports {
    constructor(client : p.DeclarationsApi) {
        super(client);
        this.declarationId = addDeclaration(client);
    }
    sum(ints : number[]) : Promise<number> {
        /* implementation */
    }
    sumCustomerAge(customers : Customer[]) : Promise<number> {
        /* implementation */
    }
}
```

`Reports` requires one parameter in its constructor, an implementation of the `DeclarationsApi` (from `@deondigital/api-client`), used for issuing the actual call to the API.

Prior to calling reports, a check is made to ensure the declaration is stored on the server. Otherwise it is added.

We can call a report with native TypeScript types analogously to the way we did it for Kotlin:

```
// 'deonApiClient' and 'contractId' defined elsewhere
const r = new Reports(deonApiClient.declarations)
const s = await r.sumCustomerAge([new Customer (
    "bob",
    42,
    new Address (
        "Main st.",
        1,
        5
      ),
    new OneTime()
  ), new Customer (
    "alice",
    30,
    new Address (
      "Main st.",
      10,
      5
    ),
    new Regular(1)
  )])
```

Contract instantiation

The generated TypeScript code for `sic1.csl` contains the following class for contract instantiation:

```
export class Shop {
    private constructor(
        readonly contractId : string,
        private readonly client : p.DeonApi,
        readonly total : number
    ) {}

    static async instantiate(
        client: p.DeonApi,
        total: number,
        $args? : { peers? : p.ExternalObject[], name? : string }
    ) : Promise<Shop> {
        const declarationId = await addDeclaration(client.declarations);
        return await new Instantiate(client.contracts, declarationId).Shop(total,
↪$args).then(cid => {
```

(continues on next page)

(continued from previous page)

```
            return {
                contractId: cid,
                total: total,
                commands: (agent: string, timeProvider : () => Date = () => new␣
↪Date()) => new Commands(client.contracts, cid, agent, timeProvider)
            }
        })
    }

    commands(agent: string, timeProvider? : () => Date){
        return new Commands(this.client.contracts, this.contractId, agent,␣
↪timeProvider)
    }
}
```

This is similar to the Kotlin code: `sic` generates one class per template. The generated class is a representation of the instantiated contract, with instantiation parameters, contract id, and a `commands` function to apply events. Furthermore, it contains a static method `instantiate` that takes as parameters:

1. A `DeonApi` client (from `@deondigital/api-client`) to do the declaration and instantiation call.
2. Any parameters the contract template expects (in this example, `total`)
3. An optional `$args` for specifying peers and name of the contract instance.

```
// 'deonApiClient' defined elsewhere
// Instantiate 'Shop'
const contract1 = await Shop.instantiate(deonApiClient, 42);
// Instantiate a new contract from 'Shop'
const contract2 = await Shop.instantiate(deonApiClient, 11);
// Instantiate yet another contract from the 'Shop'
// template, but give the instance a custom name
const namedContract = await Shop.instantiate(deonApiClient, 47, { name :
↪"shopContract47" });
```

Event application

`sic` has generated the following TypeScript class:

```
// In sic1.ts
export class Commands extends builtins.Commands {
    async AddCustomer(
        id : number,
        customer : Customer,
        $tag? : p.Tag) : Promise<p.Tag | void> {
        /* implementation */
    }
}
```

Again, the pattern is quite like it was for Kotlin: one function per event type and the functions take as parameters the fields of the event. Note that the function for applying the basic event type `Event` is contained in the super class `builtins.Commands`.

Access to the `Commands` object is through the `.commands(...)` field on the template interface (here: `Shop`).

Supply the following parameters to the `commands` function:

1. The originating agent for the event.
2. An optional "time provider" that returns the timestamp to be used in the event. It can be left out, in which case the current time is used.

In this snippet we apply two `AddCustomer` events on a contract:

```
// 'deonApiClient', 'contract', and 'agent' defined elsewhere
const c = contract.commands(agent);
// Now we can apply two 'AddCustomer' events on the contract:
await c.AddCustomer(0, new Customer (
   "bob",
   42,
   new Address ("Main st.", 1, 5),
   new OneTime()
));
await c.AddCustomer(1, new Customer (
   "alice",
   30,
   new Address ("Main st.", 10, 5),
   new Regular(1)
));
```

The `@deondigital/sic` NPM package

The `sic` tool is distributed in the NPM package `@deondigital/sic`. It provides a handy way to install `sic`:

```
$ npx @deondigital/sic
```

This will download the latest version of `sic` and run it. If you want `sic` code generation as part of your build process, add `@deondigital/sic` as a project dependency and add a `"generate"` entry to the `"scripts"` section of `package.json`:

```
/* ... */
"scripts": {
  /* ... */
  "generate": "sic --target TypeScript --write *.csl"
},
"dependencies": {
  /* ... */
  "@deondigital/sic": "0.58.0"
}
```

Now you can use the `generate` script in your project:

```
$ npm run generate
```

Note: The version number of the `@deondigital/sic` package follows that of the rest of Deon Digital CSL. Version `0.58.0` of `@deondigital/sic` will download a version of `sic` that is compatible with version `0.58.0` of the Deon Digital CSL runtime.

3.1.4 Emit JSON representation of core AST

It is possible to use `sic` to emit a structured core representation of a CSL project in JSON format by using the `CoreAST` target.

```
$ sic --write --target CoreAST sic1.csl
Generating interface for sic1.csl
No project file found
Wrote file generated/./sic1.coreast.json
```

The file `sic1.coreast.json` contains a JSON representation of the core AST of `sic1.csl` along with the standard library. Note that this core representation is the result of several steps of internal processing that among other things strips away type information.

3.1.5 Emit JSON representation of type definitions

It is possible to get a JSON-serialized representation of all types used in a contract.

To use, simply run

```
$ sic --write --target Ontology sic1.csl
Generating interface for sic1.csl
No project file found
Wrote file generated/./sic1.ontology.json
```

This will write the ontology for the whole project to `sic1.ontology.json`, including the ontology for the the preamble and built-ins. The output is an array of ontology elements. Each element has the following form:

```
{
  kind: { /** type descriptor */ },
  name: {
    name: /** type name (string) */,
    qualifier: [ /** type qualifier */ ]
  }
}
```

Type descriptors have one of the following forms:

```
{
  tag: "OntologyRecord",
  parent: { /** qualified name */ },
  fields: {
    "field1": { /** type identifier */ },
    /** ... */
  }
}

{
  tag: "OntologyUnion",
  parameters: [ /** type parameters */ ],
  constructors: [ /** constructors */ ]
}
```

For example, the ontology element for the `Maybe a` union type looks as follows:

```
{
  "kind": {
    "tag": "OntologyUnion",
    "constructors": [
      {
        "arguments": [],
        "name": "None"
      },
      {
        "arguments": [
          {
            "tag": "Var",
            "identifier": "a"
          }
        ],
        "name": "Some"
      }
    ],
    "parameters": [
      {
        "parameter": "a"
```

(continues on next page)

(continued from previous page)

```
      }
    ]
  },
  "name": {
    "name": "Maybe",
    "qualifier": []
  }
}
```

The ontology element for the `Event` record type looks as follows:

```
{
  "kind": {
    "parent": {
      "name": "Record",
      "qualifier": []
    },
    "tag": "OntologyRecord",
    "fields": {
      "agent": {
        "tag": "Apply",
        "params": [],
        "name": {
          "name": "Agent",
          "qualifier": []
        }
      },
      "timestamp": {
        "tag": "Apply",
        "params": [],
        "name": {
          "name": "DateTime",
          "qualifier": []
        }
      }
    }
  },
  "name": {
    "name": "Event",
    "qualifier": []
  }
}
```

3.1.6 Emit declaration signature

It is possible to get a JSON representation of the types of all top-level declarations in a CSL file.

To use, simply run

```
$ sic --write --target Signature sic1.csl
Generating interface for sic1.csl
No project file found
Wrote file generated/./sic1.signature.json
```

This will write the signature for the whole project to `sic1.signature.json`, including the signature for the the preamble and built-ins. The output is an array of type declarations. Each element has one of the following forms:

```
// Value declaration
{
```

(continues on next page)

(continued from previous page)

```
  "tag": "DeclVal",
  "name": { /* ... a QualifiedName object ... */ },
  "type": { /* ... an OntologyTypeIdentifier object ... */ }
}

// Contract declaration
{
  "tag": "DeclContract",
  "name": { /* ... a QualifiedName object ... */ }
}

// Template declaration
{
  "tag": "DeclTemplate",
  "parameterTypes": [
    /* ... list of OntologyTypeIdentifier objects... */
  ],
  "name": { /* ... a QualifiedName object ... */ },
  "contractArity": 0
}
```

For example, given the following CSL file:

```
val x = 42
contract c = success
template [x] T(a) = <*> Event where a = 42 or x
```

If we strip out the signature for the preamble and built-ins (which is the same for all input files), then the following declarations are generated:

```
[
  {
    "tag": "DeclVal",
    "name": {
      "name": "x",
      "qualifier": []
    },
    "type": {
      "tag": "Apply",
      "params": [],
      "name": {
        "name": "Int",
        "qualifier": []
      }
    }
  },
  {
    "tag": "DeclContract",
    "name": {
      "name": "c",
      "qualifier": []
    }
  },
  {
    "tag": "DeclTemplate",
    "parameterTypes": [
      {
        "tag": "Apply",
        "params": [],
        "name": {
          "name": "Int",
```

(continues on next page)

(continued from previous page)

```
          "qualifier": []
        }
      }
    ],
    "name": {
      "name": "T",
      "qualifier": []
    },
    "contractArity": 1
  }
]
```

3.1.7 Projects with multiple CSL files

CSL contracts can be grouped into projects by defining a file called `deon-project` in a folder. This file contains a newline-separated list of relative or absolute paths to CSL files, its presence folder `foo` means that the folder is a "project", and that the CSL files should be loaded in the order specified in the `deon-project` file. For example, the following `deon-project` file specifies a project that contains the files `sic1.csl`, `sic2.csl`, and `sic3.csl`:

```
sic1.csl
subfolder/sic2.csl
/absolute/folder/sic3.csl
```

Contracts that are part of a `deon-project` are typechecked in the context of all contracts that come before them in the project specification. Thus, `sic1.csl` may only refer to names declared in the same file or in the standard library, whereas anything declared in `sic1.csl` is in scope in `sic2.csl`, and anything in `sic1.csl` and `sic2.csl` is in scope in `sic3.csl`.

Using projects in `sic`

Given the project file `myproject/deon-project`:

```
sic1.csl
sic2.csl
```

Running the command:

```
$ sic --write --target Kotlin --namespace org.foo myproject/*.csl
```

Will output the following:

```
Wrote file generated/org/foo/myproject/ContractOperations.kt
Wrote file generated/org/foo/myproject/RESTContractOperations.kt
Wrote file generated/org/foo/myproject/Entities.kt
Wrote file generated/org/foo/myproject/preamble.kt
Wrote file generated/org/foo/myproject/builtins.kt
Wrote file generated/org/foo/myproject/sic1.kt
Wrote file generated/org/foo/myproject/sic1.csl.kt
Wrote file generated/org/foo/myproject/sic2.kt
Wrote file generated/org/foo/myproject/sic2.csl.kt
Wrote file generated/org/foo/myproject/fromValue.kt
Wrote file generated/org/foo/myproject/InstanceDispatcher.kt
Wrote file generated/org/foo/myproject/ReportService.kt
Wrote file generated/org/foo/myproject/ContractService.kt
```

CHAPTER 4

Examples

For various examples look in these chapters:

4.1 Examples of CSL contracts

In this section you can see a selection of example CSL contracts. Some of these will be directly copy-pasteable as-is while others assume that they occur in some context with the proper event types defined. It will be clear from the examples whether you need to insert the appropriate missing bits or not.

Many contracts use the recurring event types `Payment` and `Delivery`, defined as follows:

```
type Payment : Event {
  amount : Int, // Amount of money in euros
  receiver : Agent
}
type Delivery : Event {
  goods : String
}
```

Recall that the base `Event` type is defined as:

```
type Event {
  agent : Agent,
  timestamp : DateTime
}
```

4.1.1 Delivery deadline relative to payment

In this example, a payment from `Alice` is required before a delivery from `Bob`. The delivery has a deadline of "two days after receiving payment":

```
template entrypoint DeliveryDeadline(alice,bob) =
  <alice> p: Payment
  then
  <bob> d: Delivery where
    d.timestamp < DateTime::addDays p.timestamp 2
```

The expression uses the strictly-less-than comparison operator < and as a consequence, the delivery cannot be *exactly* two days later than the payment but only until, but not including, two days later. Had we instead used the <= operator, the delivery would be allowed to occur two days after payment on the second.

4.1.2 Late payment

The following is a specification of a contract in which `Alice` must pay 100 euros before some deadline and once this has occurred, `Bob` has 2 days to perform a delivery of `"SomeBike"`. If `Alice` pays after the deadline, she has to pay 110 euros in order to get `Bob` to deliver the `"SomeBike"`. `Bob` still has to deliver `"SomeBike"` before two days after payment when `Alice` pays too late.

```
template entrypoint LatePayment
  (paymentDeadline: DateTime, alice: Agent, bob: Agent) =
  <alice> p: Payment where
    p.amount = 100 &&
    p.timestamp <= paymentDeadline
  then
  <bob> d: Delivery where
    d.timestamp < DateTime::addDays p.timestamp 2 &&
    d.goods = "SomeBike"
  or
  <alice> p: Payment where
    p.amount = 110 &&
    p.timestamp >= paymentDeadline
  then
  <bob> d: Delivery where
    d.timestamp < DateTime::addDays p.timestamp 2 &&
    d.goods = "SomeBike"
```

The contract uses the `or` combinator to combine the contracts for payment on time and late payment.

4.1.3 Partial payments

In the previous contracts, the full amount had to be paid in one go. It is possible to write a contract that allows payments to occur over several events:

```
template entrypoint PartialPayment(deadline,alice,bob) =
  let
    template rec PayPartially(remainingAmount) =
      <alice> p: Payment where
        p.timestamp < deadline &&
        p.amount < remainingAmount
      then
        // She paid only part of the amount, so await the remaining sum.
        PayPartially(remainingAmount - p.amount)
      or
      <alice> p: Payment where
        p.timestamp < deadline &&
        p.amount = remainingAmount

    val deliveryDeadline = DateTime::addDays deadline 2
  in
    PayPartially(100) // Alice must pay 100 euros in total
    then
    <bob> d: Delivery where
      d.timestamp < deliveryDeadline &&
      d.goods = "SomeBike"
```

In this example we make essential use of recursive contracts. We define a local contract template called `PayPartially` which takes as its single value parameter the remaining amount of money that has to be paid.

This template gets instantiated to a contract that is a combination (with `or`) of two contracts for the following scenarios:

1. `Alice` pays an amount that is smaller than what is owed. The contract template is used to create a new contract that handles the remaining amount.
2. `Alice` pays the entire amount and the contract completes successfully.

When the `PayPartially`-instantiated contract finishes it can only be because `Alice` has sent one or more events with payments of 100 euros. We omit the concept of *change* here, so the contract only supports exact payments. At that point, the "top-level" contract is simply:

```
// .. deliveryDeadline defined earlier ...
(
  success
)
then
<bob> d: Delivery where
  d.timestamp < deliveryDeadline &&
  d.goods = "SomeBike"
```

One can always simplify a contract `success then c` to just `c`, so the residual contract is:

```
// ... deliveryDeadline defined earlier ...
<bob> d: Delivery where
  d.timestamp < deliveryDeadline &&
  d.goods = "SomeBike"
```

Only a delivery is expected now.

4.1.4 Escrow

Below is a version of the standard sale-of-bike contract repeated for the sake of example:

```
template entrypoint BikeSale(seller, buyer) =
  <buyer> p: Payment where
    p.amount = 100 &&
    p.receiver = seller
  then
  <seller> d: Delivery where
    d.goods = "SomeBike" &&
    p.receiver = buyer
```

One could consider it problematic that the `buyer` is obligated to perform a payment before receiving the bike as maybe the bike was not in good condition upon arrival. However, requiring delivery before payment is also unreasonable because the seller would not like to deliver a bike without knowing that it will be paid for. Instead we introduce a trusted third-party agent to the contract:

```
template entrypoint BikeSaleEscrow(trustedParty, seller, buyer, deliveryDeadline) =
  <buyer> p: Payment where
    p.receiver = trustedParty &&
    p.amount = 100
  then (
    <seller> d: Delivery where
      p.receiver = buyer &&
      d.goods = "SomeBike" &&
      p.timestamp < deliveryDeadline
    then
    <trustedParty> tp: Payment where
      tp.receiver = seller &&
      tp.amount = 100
    or
```

(continues on next page)

(continued from previous page)

```
    <trustedParty> bp: Payment where
      bp.receiver = buyer &&
      bp.amount = 100 &&
      bp.timestamp > deliveryDeadline
  )
```

The idea is that `trustedParty` is a third-party agent that both `seller` and `buyer` trusts enough to hold the payment in escrow. Initially, the `buyer` submits a `Payment` to the `trustedParty` of the expected amount. After that, either the `seller` will perform a `Delivery` after which the `trustedParty` is obliged to perform a `Payment` to the `seller`. If the seller does not perform a `Delivery` within the deadline, the `trustedParty` must to pay back the money to `buyer`.

An important thing to consider when entering into the escrow contract is that the resource-flows are as expected. After successful completion of the contract, either the `seller` has one `SomeBike` fewer than before and `buyer` has 100 fewer euros *or* no party in the contract have any change in resource holdings.

4.1.5 Transferable contract

The following is an example of a car-ownership/rental contract where the car can be rented out to other parties and where the owner may transfer ownership of the car completely:

```
type Open : Event {} // Open the car doors
type Close : Event {} // Close the car doors
type Return : Event {} // Return car to owner
type TransferOwnership : Event {
  newOwner : Agent
}
type ReserveCar : Event {
  renter : Agent,
  rentalStart : DateTime,
  rentalEnd : DateTime
}
// When does user stop having the rights to a car
type EndUsage
  | Ends DateTime // At this point in time.
  | Never // The user owns the car.

val mayStillUseCar = \time ->
  \ Ends endTime -> time <= endTime
  | Never        -> True

template CarUsage(user, start, endUsage) =
  <user> o: Open where
    o.timestamp > start &&
    mayStillUseCar o.timestamp endUsage
  then
  <user> Close

template rec CarRental(renter, rentalStart, rentalEnd) =
  CarUsage(renter, rentalStart, Ends rentalEnd)
  then CarRental(renter, rentalStart, rentalEnd)
  or
  <renter> Return

template rec entrypoint CarOwnership(owner, start) =
  <owner> t: TransferOwnership then CarOwnership(t.newOwner, t.timestamp)
  or
  CarUsage(owner, start, Never) then CarOwnership(owner, start)
  or
  <owner> r: ReserveCar
```

(continues on next page)

(continued from previous page)

```
40    then
41    CarRental(r.renter, r.rentalStart, r.rentalEnd)
42    then
43    CarOwnership(owner, start)
```

Ownership of the car is modeled by the contract `CarOwnership`. An owner can do one of the following:

1. Give the car to someone else by issuing a `TransferOwnership`. This causes the next valid sequence of events to be modeled by a new instance of the `CarOwnership` contract with a new owner.
2. Use the car. Car usage is modeled in a simplified way where a user can just open and close the car doors if she is allowed to at that point in time. The `CarOwnership` contract makes sure that the owner of the car is always allowed to use the car by setting the `endUsage` to `Never`.
3. Rent out the car to someone else, temporarily giving the renter the rights to use the car but requiring that she returns it back again. The renter may only use the car until a certain point in time. This is achieved by setting the `endUsage` to `Ends rentalEnd` on line 29.

This contract also demonstrates the usefulness of *sum types*. `EndUsage` expresses that either there is an expiration date for the car or the car can be used as long as the user likes, reflecting the situations where the car is either used be the renter or the owner. We also introduced a convenience function `mayStillUseCar` that compares a timestamp to a value of type `EndUsage` on line 18. Either the `EndUsage` is an `Ends d` where `d` is a `DateTime`, in which case we just compare the timestamps to see if the user may use the car, or the `EndUsage` is a `Never`, in which case the user may always use the car and the function returns `True`.

4.1.6 Signing contracts

In the car ownership/rental example above, the car owner basically gets to decide whether a new owner should have the car or not. This is unlikely to model a realistic transfer of ownership, as the recipient might want the chance to either accept or reject the offer. We can extend our contract from above with a notion of *signing*:

```
1   // ... events 'Close', 'Return', etc. defined as before
2   // ... templates 'CarUsage', 'CarRental', defined as before
3   type Sign : Event {}
4   type Reject : Event {}
5
6   template [theContract, default] SignedContract(party, deadline) =
7   <party> s: Sign where
8     s.timestamp < deadline
9   then theContract
10  or
11  <party> Reject then default
12
13  template rec entrypoint CarOwnershipSigned(owner, start) =
14    <owner> t: TransferOwnership
15    then
16    SignedContract[
17      // Transfer ownership if the new owner accepts the transfer and signs.
18      CarOwnershipSigned(t.newOwner, t.timestamp),
19      // Keep ownership if the new owner rejects the transfer.
20      CarOwnershipSigned(owner, start)
21    ](t.newOwner, DateTime::addDays t.timestamp 2)
22    // New owner must sign before two days
23    or
24    CarUsage(owner, start, Never) then CarOwnershipSigned(owner, start)
25    or
26    <owner> r: ReserveCar
27    then
28    CarRental(r.renter, r.rentalStart, r.rentalEnd)
29    then
30    CarOwnershipSigned(owner, start)
```

After being offered a car with the `TransferOwnership` event, the recipient can choose to either accept or reject the offer with a `Sign` or `Reject` event, respectively. We use the contract template `SignedContract` on line 6 with two template parameters to model the general pattern where two different things might happen depending on whether the user signs or not. In the `CarOwnership` template on line 16 we then simply instantiate the `SignedContract` with contracts reflecting the cases where the recipient accepts or rejects the car. For the purposes of this example, the only requirements that are checked on the signature is that it happened before a certain deadline.

4.2 Examples of CSL reports

In this section we show some example CSL reports. We will focus primarily on reports using the *Report Query Language*. For some of the examples, we will re-use the contract declarations defined in the *examples* section.

4.2.1 Late payment

In the *late payment* contract, the price of a bike depends on whether it was paid for on time or late. By observing the amount paid, we can decide if the payment was late, without knowing the agreed deadline.

```
relation paymentRel(amount: Int, cid)
| Payment {amount = amount} @ cid

val when = \bexp -> \e -> if (bexp) e else id

val report wasPaymentLate = \(cid: Contract) ->
  let val payment =
    (for (amount, c) in paymentRel do
      when (c=cid) (\_ -> Some amount)) None
  in Maybe::map (\x->x=110) payment
```

First we define a relation `paymentRel`, giving us access to the `amount` field of the `Payment` event, for a concrete contract instance denoted by `cid`. As each contract expects exactly one `Payment` event to occur, if we have found the right contract instance we can ignore the accumulator in the *for expression* and simply return the `amount` field. If no payment occurred, this code will return `None`, if a payment occurred it returns `Some True` when it was late, and `Some False` when it was on time.

4.2.2 Partial payments

When taking part in a recursive contract where a specific type of event can occur multiple times, such as the *partial payments* contract, it is often of interest to inspect the data load of all applied events. For instance in partial payments contract we may want to calculate how much money has already been paid. We can use the same relation as before, but this time we expect multiple payments, and we want to calculate the sum of all the `amount` fields in `Payment` events.

```
relation paymentRel(amount: Int, cid)
| Payment {amount = amount} @ cid

val when = \bexp -> \e -> if (bexp) e else id

val report paidTotal = \cid ->
  (for (amount, c) in paymentRel do
      when (c=cid) (\acc -> acc + amount)
  ) 0
```

We again focus on elements of `paymentRel` coming from a specific contract we have picked when calling the report, and add the `amount` field of each of them to an accumulator. We begin with the accumulator value of 0, as this is the correct total in a contract where no payments were yet made. Note that without the `when (c=cid)`

conditional, the code above would instead calculate the sum of all payments observed by a contract manager, regardless of which contract they originated from.

If, instead of calculating the sum of payments, we simply want to list them in the order in which they were submitted, we have to change the `paymentRel` definition to also include timestamps.

```
relation paymentRel(timestamp : DateTime, amount: Int, cid)
| Payment {timestamp = timestamp, amount = amount} @ cid

val when = \bexp -> \e -> if (bexp) e else id

val paidDates = \cid ->
  (for (dt, am, c) in paymentRel do when (c=cid) (Cons (dt, am))) Nil

val report sortedPaidDates = \cid ->
  List::sort (\x -> \y -> compareDateTime (fst x) (fst y)) (paidDates cid)
```

`paidDates` report allows us to construct a list of tuples containing the payment date and the paid amount. We can then sort this list based on the value of the first component, as seen in `sortedPaidDates` report.

4.2.3 Periodic payments

Another typical scenario is a contract where some payments should occur periodically. Consider for instance a simple monthly payment scheme, where `alice` pays `bob` a pre-specified amount of money every 30 days. The first date of payment is given by `bob` in a `TermsAndConditions` event, the next ones follow every 30 days.

A typical query to such a contract may be: "When is the next payment due?". To answer this question we need to know when was the last payment made, and add 30 days to it. If no payment has yet been made, the next payment is due on the date specified in terms and conditions, if that was issued.

```
type Payment : Event {
  amount : Int,
  receiver : Agent
}

type TermsAndConditions : Event {
  firstPayment : DateTime
}

template entrypoint Main(alice, bob, amount) =
  let template rec RecurringPayment(nextPaymentDate) =
    <alice> p : Payment where
      p.amount = amount &&
      p.timestamp = nextPaymentDate &&
      p.receiver = bob
    then
    RecurringPayment(DateTime::addDays nextPaymentDate 30)
  in
  <bob> tac : TermsAndConditions then RecurringPayment(tac.firstPayment)

relation tocRel (timestamp : DateTime, cid)
| TermsAndConditions {firstPayment = timestamp} @ cid

relation paymentRel(timestamp : DateTime, amount: Int, cid)
| Payment {timestamp = timestamp, amount = amount} @ cid

val when = \bexp -> \e -> if (bexp) e else id

val headTail =
  \ Nil -> None
  | Cons x xs -> Some (x, xs)
```

(continues on next page)

(continued from previous page)

```
32
33 val maximum = \lst -> \cmp ->
34   let val minFnd =
35     foldl (\x -> \y -> (
36       \ Less -> y
37       | Equal -> y
38       | Greater -> x) (cmp x y))
39   in Maybe::map (\(hd, tl) -> minFnd hd tl) (headTail lst)
40
41 val report paidDates = \cid ->
42   (for (dt, _, c) in paymentRel do when (c=cid) (Cons dt)) Nil
43
44 val report firstPayment = \cid ->
45   (for (dt, c) in tocRel do when (c=cid) (\_ -> Some dt)) None
46
47 val report nextPayment = \cid ->
48   let val lastPaid = maximum (paidDates cid) compareDateTime
49       val nextToPay =
50         Maybe::map (\dt -> DateTime::addDays dt 30) lastPaid
51   in
52   if (Maybe::isSome lastPaid) nextToPay else (firstPayment cid)
```

`paidDates` and `firstPayment` reports follow what we have already seen in previous examples. `nextPayment` is an example of combining reports – it tries to find the date of last payment (i.e. latest date within `paidDates`) and adds 30 days to it. If no payments have yet been made, both `lastPaid` and `nextToPay` will have a value of `None`. In this case, we resort to the `firstPayment` value – which again may be a `None` if the `TermsAndConditions` event has not been submitted yet. The helper `maximum` report defined in line 33 finds a maximum of a given list using the comparison function provided, whereas the `headTail` report simply combines the functionality of *List::head* and *List::tail* reports from the standard library.

CHAPTER 5

Exercises

Some more advanced exercises and solutions to most exercises presented in this guide:

5.1 Exercises

This chapter collects somewhat more advanced exercises for CSL. Most of them require the use of the CSL *standard library* and/or some functionality that is documented in the *contract language reference* or *value expression reference* and not presented in detail in the introductory chapters.

5.1.1 Writing and running contracts

Exercise: Age-restricted purchase

Write a contract for operating a shop where some items are age-restricted. This means that the `buyer` will have to provide a valid identification in order to purchase these items.

Allow for multiple items to be sold. The valid id only has to be presented once for the entire purchase.

Test the contract with both restricted and unrestricted items (*solution*).

Exercise: Shopping basket

Write a contract which allows you to add multiple items from the shop to your basket and paying for all of them together.

Try adding an event for removing an item from the basket. Remember to check that the item was indeed first added to it.

Try running the contract to see if `Checkout` behaves as you expect. Test that if you added an item once, you can remove it also only once (*solution*).

Exercise: Two-buyer protocol

Write a protocol in which two buyers interact with one seller. Buyers first negotiate which item to purchase. They then make enquiries about its price. Seller informs both of them about the price of the item they agreed on. Each

of the buyers declares how much of the price they are willing to pay. The buyer who suggested the item to be purchased declares his share first. Otherwise, if the sum of the two offers is at least as high as the price, the seller delivers the item to the buyers.

Try writing this contract using subcontracts for the different stages: item negotiations, price enquiry, two buyers declaring shares etc.

Run the contract and verify that at each stage, correct events are expected (*solution*).

5.1.2 Expressions and reports

Exercise: Bank account balance

`AccountOperations` is a very simplistic way of managing a single bank account.

```
type Deposit : Event { amount : Int }

type Withdraw : Event { amount : Int }

template rec AccountOperations() =
  (<*> d : Deposit where
     d.amount > 0
  or
  <*> w : Withdraw where
    w.amount > 0)
  then AccountOperations()
```

Write a reporting function which calculates the balance of a bank account. Your function should be of type `Contract -> Int`.

Next, change the contract in such a way, that it prevents the balance to drop below 0 (*solution*).

Exercise: Reimbursement for business travel

The following declaration specifies a protocol allowing an employee to inform their employer about a planned business travel. The employer can either accept or reject the planned trip.

```
type Transportation
  | Bike
  | Bus
  | Car
  | Plane
  | Other Int

val calculateTransportationPrice =
  \ Bike -> 0
  | Bus -> 10
  | Car -> 50
  | Plane -> 500
  | Other price -> price

type TravelNotice : Event {
  employee : Agent,
  transport : Transportation,
  startDate : DateTime,
  endDate : DateTime
}

type TravelAcknowledgement : Event {
  employee : Agent,
```

(continues on next page)

(continued from previous page)

```
  startDate: DateTime,
  endDate: DateTime
}

type TravelCancellation : Event {
  employee : Agent,
  startDate: DateTime,
  endDate: DateTime
}

template BusinessTravelAgreement (employer, employee) =
  <employee> tn : TravelNotice where
    tn.employee = employee
  then
  (<employer> ta : TravelAcknowledgement where
    tn.employee = ta.employee &&
    tn.startDate = ta.startDate &&
    tn.endDate = ta.endDate
  or
  <employer> tc : TravelCancellation where
    tn.employee = tc.employee &&
    tn.startDate = tc.startDate &&
    tn.endDate = tc.endDate)
```

Write a reporting function which calculates how much money should the `employee` be reimbursed for, based on the mode of transportation and the duration of travel. Assume that for trips shorter than 24 hours there is no *per diem*, but for trips of 1 day or longer, the employee gets 50 Euros for each full day.

Next, write a report which calculates how much should each employee be reimbursed. The function that calculates this report should have the following type: `List Contract -> List Agent -> List (Tuple Agent Int)`. Remember that employees might take multiple business trips, and that only approved trips are reimbursed (*solution*).

Hint

When using the provided *GUI* to solve this exercise, make sure that the source code is the same for all of the contracts to be included in the report. This ensures that contracts can *see* each other, allowing you to run `generateTravelReport` with all the contract identifiers of different contracts you run.

For instance, if you have 3 instances of `BusinessTravelAgreement` and 2 employees, you can run:

```
\c1 -> \c2 -> \c3 -> \a1 -> \a2 -> generateTravelReport [c1,c2,c3] [a1,
a2]
```

in the `Reports` tab, and select appropriate arguments.

5.2 Solutions

Here we present solutions to the exercises that can be found throughout this language guide – both to the simpler ones from the introductory chapters, and to the more advanced ones presented in the previous section.

The solutions presented here are merely examples.

5.2.1 Solution: Pay on delivery

See the exercise *statement*.

```
type Delivery : Event {
  recipient : Agent,
  item : String
}

type Payment : Event {
  amount : Int,
  recipient : Agent,
  item : String
}

template entrypoint Sale(buyer, seller, amount, item) =
  <seller> delivery: Delivery where
    delivery.item = item &&
    delivery.recipient = buyer
  then
  <buyer> payment: Payment where
    payment.amount = amount &&
    payment.recipient = seller &&
    payment.item = item
```

5.2.2 Solution: Pay or return

See the exercise *statement*.

```
type Delivery : Event {
  recipient : Agent,
  item : String,
  price : Int
}

type Payment : Event {
  amount : Int,
  recipient : Agent,
  item : String
}

type Return : Event {
  item : String,
  recipient : Agent
}

val inventoryPrice =
\ "Bike" -> 100
| "Hammer" -> 30
| "Saw" -> 40
| _ -> 20000 // Arbitrary large number

template entrypoint Sale(buyer, seller, item) =
  <seller> delivery: Delivery where
    delivery.item = item &&
    delivery.recipient = buyer &&
    delivery.price = inventoryPrice item
  then
  (<buyer> payment: Payment where
    payment.amount = delivery.price &&
    payment.recipient = seller &&
    payment.item = item
  or
  <buyer> return: Return where
```

(continues on next page)

(continued from previous page)

```
    return.item = delivery.item &&
    return.recipient = seller
  )
```

5.2.3 Solution: Bike and helmet

See the exercise *statement*.

```
type Order : Event {
  price : Int,
  recipient : Agent,
  item : String
}

val inventoryPrice =
\ "Bike" -> 100
| "Helmet" -> 50
| "Saw" -> 40
| _ -> 20000 // Arbitrary large number

template entrypoint DoubleOrder(buyer, seller) =
  <buyer> helmetOrder : Order where
    helmetOrder.item = "Helmet" &&
    helmetOrder.price = inventoryPrice "Helmet" &&
    helmetOrder.recipient = seller
  and
  <buyer> bikeOrder : Order where
    bikeOrder.item = "Bike" &&
    bikeOrder.price = inventoryPrice "Bike" &&
    bikeOrder.recipient = seller
```

5.2.4 Solution: Late payment

See the exercise *statement*.

```
type Delivery : Event {
  recipient : Agent,
  item : String
}

type Payment : Event {
  amount : Int,
  recipient : Agent,
  item : String
}

template entrypoint Sale(buyer, seller, amount : Int, item, fine) =
  let val amountFromDate = \delivery -> \timestamp ->
    if (timestamp <= DateTime::addDays delivery 3) amount
    else amount + fine
  in
  <seller> delivery: Delivery where
    delivery.item = item &&
    delivery.recipient = buyer
  then
  <buyer> payment: Payment where
    payment.amount = amountFromDate delivery.timestamp payment.timestamp &&
```

(continues on next page)

(continued from previous page)

```
    payment.recipient = seller &&
    payment.item = item
```

5.2.5 Solution: Age-restricted purchase

See the exercise *statement*.

```
type ProvideId : Event {}

type Purchase : Event {
  ageRestriction : Bool,
  item : String
}

template rec entrypoint Loop (buyer, shownId) =
  <buyer> ProvideId then Loop (buyer, True)
  or
  <buyer> p : Purchase where
    not (p.ageRestriction) || shownId
    then Loop(buyer, shownId)

template RestrictedSale(buyer) = Loop(buyer, False)
```

5.2.6 Solution: Shopping basket

See the exercise *statement*.

```
type AddItem : Event{
  item : String
}

type RemoveItem : Event{
  item : String
}

type Checkout : Event {
  total : Int,
  basket : List String
}

val priceList =
 \ "Cabbage" -> 10
 | "Potato" -> 5
 | "Milk" -> 7
 | "Apple" -> 9
 | _ -> 0

val removeFirst = \rm -> \l ->
  let val inner = \el -> \(acc : Tuple (List String) Bool) ->
    if (snd acc) (Cons el (fst acc), True)
    else if (el=rm) (fst acc, True) else (Cons el (fst acc), False) in
  fst (foldr inner ([], False) l)

template rec entrypoint MultiSale(buyer, basket, total) =
  <buyer> order : AddItem where
    priceList order.item > 0
  then MultiSale (buyer, Cons order.item basket, total + priceList order.item)
  or
```

(continues on next page)

(continued from previous page)

```
  <buyer> order : RemoveItem where
    Maybe::isSome (List::first (\(item : String) -> item = order.item) basket)
  then MultiSale(buyer,
                 removeFirst order.item basket,
                 total - priceList order.item)
  or
  <buyer> checkout : Checkout where
    checkout.basket = basket &&
    checkout.total = total

template Sale (buyer) = MultiSale(buyer, [], 0)
```

5.2.7 Solution: Two-buyer protocol

See the exercise *statement*.

```
type ItemSuggesion : Event {
  item : String
}

type SuggestionAccept : Event {}

type PriceEnquiry : Event {
  item : String,
  buyer1 : Agent,
  buyer2 : Agent,
  seller : Agent
}

type ItemPrice : Event {
  item : String,
  price : Int,
  recipient1 : Agent,
  recipient2 : Agent
}

type DeclareShare : Event {
  amount : Int
}

type Delivery : Event {
  item : String,
  recipient1 : Agent,
  recipient2 : Agent
}

type InsufficientFunds : Event {}

template AttemptPurchase (seller, buyer1, buyer2, item, price) =
  <buyer1> d1 : DeclareShare then <buyer2> d2 : DeclareShare
  then
  (<seller> d : Delivery where
    d1.amount + d2.amount >= price &&
    d.recipient1 = buyer1 &&
    d.recipient2 = buyer2 &&
    d.item = item
  or
  <seller> InsufficientFunds where
    d1.amount + d2.amount < price)
```

(continues on next page)

(continued from previous page)

```
template CheckPrice (seller, buyer1, buyer2, item) =
  <buyer1> p : PriceEnquiry where
    p.item = item &&
    p.buyer1 = buyer1 &&
    p.buyer2 = buyer2 &&
    p.seller = seller
  then
  <seller> ip : ItemPrice  where
    ip.item = item &&
    ip.recipient1 = buyer1 &&
    ip.recipient2 = buyer2
  then
  AttemptPurchase (seller, buyer1, buyer2, item, ip.price)

template rec AgreeItem(seller, buyer1, buyer2) =
  <buyer1> i : ItemSuggesion
  then
  ((<buyer2> SuggestionAccept then CheckPrice(seller, buyer1, buyer2, i.item))
  or
  AgreeItem(seller, buyer2, buyer1))

template entrypoint Sale (seller, buyer1, buyer2) = AgreeItem(seller, buyer1,␣
↪buyer2)
```

5.2.8 Solution: Bank account balance

See the exercise *statement*.

```
type Deposit : Event { amount : Int }

type Withdraw : Event { amount : Int }

template rec entrypoint AccountOperations(accountBalance) =
  (<*> d : Deposit where
      d.amount > 0 then AccountOperations(accountBalance + d.amount))
  or
  (<*> w : Withdraw where
      w.amount <= accountBalance &&
      w.amount > 0
      then AccountOperations(accountBalance - w.amount))
```

5.2.9 Solution: Reimbursement for business travel

See the exercise *statement*.

```
val reimburseForTravel =
  \(transport : Transportation) ->
  \(startDate: DateTime) ->
  \(endDate : DateTime) ->
    let val duration =
      Duration::components (Duration::between startDate endDate) in
    let val perDiem = if (duration.day > 1) duration.day*50 else 0 in
    calculateTransportationPrice transport + perDiem

val travelReportForEmployee = \(travelNotices:List TravelNotice) -> \employee ->
  let val relevantNotices =
    List::filter (\(tn:TravelNotice) -> tn.employee = employee) travelNotices in
```

(continues on next page)

(continued from previous page)

```
  (employee,
   foldl
     (\acc -> \(e:TravelNotice) ->
       acc + reimburseForTravel e.transport e.startDate e.endDate)
     0
     relevantNotices)

val isAcceptedNotice = \(notice : TravelNotice) ->
  List::any
    (\(ack:TravelAcknowledgement) ->
      ack.employee = notice.employee &&
      ack.startDate = notice.startDate &&
      ack.endDate = notice.endDate)

val report generateTravelReport =
  \(eventsFromCids : List (List Event)) ->
  \(employees: List Agent) ->
    let val travelAcknowledgements =
      List::mapMaybe
        (\(e:Event) ->
          type ta = e of {TravelAcknowledgement -> Some ta; _ -> None})
        eventsFromCids in
    let val acceptedTravelNotices =
      List::mapMaybe
        (\(e:Event) ->
          type ta = e of {
            TravelNotice ->
              if (isAcceptedNotice ta travelAcknowledgements) Some ta else None;
            _ -> None})
        eventsFromCids in
    List::map (travelReportForEmployee acceptedTravelNotices) employees
```

5.3 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

5.3.1 0.60.0 - 2020-04-15

Changed

- Upgraded Corda target platform from 4.0 to 4.4.
- TypeScript code generated by `sic` no longer adds `equals`/`hashCode` methods on `Object.prototype`.
- Significant performance improvements to the evaluator and contract executor. Tail-recursive contracts no longer get slower over time.

Fixed

- Fixed a bug related to Kryo deserialization where a Corda node would crash if resuming a flow from a persisted snapshot.
- Fixed a bug in the derivation of derived equality functions that caused runtime exceptions when one derived an equality function for a constructor with fixed type parameters.

5.3.2 0.59.0 - 2020-03-27

Changed

- The TypeScript code generated by `sic` now has better support for the `immutable-js` data structures, in particular the `immutable.Map` that CSL maps are represented as.
- `sic` generated contract operations for DbLedger now returns tags when applying events.
- Event agents are now validated against contract participants on DbLedger.

Fixed

- The automatically generated equality functions do not depend on the standard library anymore.

5.3.3 0.58.0 - 2020-03-09

Added

- `sic` now generates TypeScript classes for template entrypoints, including a static `instantiate` method.

Changed

- CSL reports that are called externally (with, e.g., `sic`) must be marked with the new `report` keyword.
- Contract templates that are used as entrypoints (from, e.g., `sic`) must be marked with the new `entrypoint` keyword.
- Event application on the DbLedger through the API now always return a tag. Previously it was only when a tag was included in the input.

Fixed

- Corrected a substitution bug the in report query language that caused wrong results whenever a variable unified with a term which itself contained variables.
- Corrected an issue in DbLedger that caused wrong external symbols to be used in event application.

5.3.4 0.57.0 - 2020-02-21

Added

- The `deondigital/vitznauerstock` Docker image now includes drivers for PostgreSQL.
- `sic` now exposes a field `instantiationTime` on `ContractInstance` objects.
- The TypeScript code generated by `sic` now contains copy methods with named parameters for classes which represent CSL records.
- Built-in `Map` data type with accompanying standard library functions.
- `sic` now generates Kotlin classes for template entrypoints, including a static `instantiate` method.

Changed

- The performance and disk usage of the DbLedger backend has been improved.

5.3.5 0.56.0 - 2020-02-10

Added

- The Deon webservice now has the endpoint `/declaration/{id}/namedReport` for invoking a report in a declaration by its qualified name and a list of value arguments.

Changed

- `sic` now reports a `ContractInstanceNotFound` instead of a general exception if a non-existing contract id is used in `getContractInstance`.
- the `sic` type `ContractInstance` now includes a function for retrieving `nextEvents` for a contract.
- `nextEvents` now include the event predicate and agent as an `Exp` type.

Fixed

- Corrected a bug in the type checker that caused it to fail to reject = on records that contained fields for which we cannot use =.
- Corrected a bug that caused lists that are returned from queries to be wrongly encoded, thereby causing runtime errors.

5.3.6 0.55.0 - 2020-01-28

Added

- Signature backend to `sic` that emits as JSON the types of all top-level declarations.

Changed

- The API endpoints for running reports have been moved from `/contracts/{id}/report` (`/contracts/report`) to `/declarations/{id}/report` (`/declarations/report`), reflecting the fact that running a report requires just the declaration, not an instantiated contract.
- The object that represents an instantiated contract now includes the peers that were specified on instantiation.

Fixed

- Fixed a bug that caused the surface-to-core translation to crash if one used = on a type alias for a built-in type.

5.3.7 0.54.0 - 2020-01-15

Added

- `sic` and the `gradle-sic-plugin` now support `deon-project` for Corda and Kotlin.

Changed

- The equality operator = now works on any closed type that does not contain function or `Signed` types.

Removed

- The standard library functions `Int::equals`, `Float::equals`, `String::equals`, and `DateTime::equals` have all been removed.

5.3.8 0.53.0 - 2020-01-09

Added

- New asynchronous API client.

Changed

- Update the `RESTContractOperations` generated by `sic` to use the asynchronous API client.
- Reports and contract entrypoints that can be called from the outside must have monomorphic types.

Fixed

- It is now possible to pass record values that are subtypes of the expected argument value to reports and template instantiations.

5.3.9 0.52.0 - 2019-12-03

Added

- CSL now supports type annotations on top-level `val` declarations: `val x : Int = 42`.

Changed

- The `sic` emitters for Kotlin and Corda now generate wrapper classes for the ledger-specific representations of `Agent` and `ContractId` to make it possible to use the same code on top of both DBLedger and Corda.
- The API type `Contract` now contains the fields `entryPoint` and `instantiationArguments`.
- The `sic` emitters for Kotlin and Corda generate the `ContractOperations` interface and `RESTContractOperations`/`CordaContractOperations` implementations which can be used to interact with the `sic`-generated contract API in a ledger-agnostic way.

5.3.10 0.51.0 - 2019-11-22

Added

- The TypeScript code generated by `sic` now contain factory methods for creating objects with named parameters.
- The Kotlin code generated by `sic` now contains copy methods with named parameters for classes which represent CSL records.
- CSL now supports type aliases: `typealias MyType a = Tuple a Int`.

Changed

- Ontology backend to `sic` was changed to output only type definitions instead of the internal contract descriptors of `sic`.
- Code generated by `sic` now inspects API values for their actual types when deserializing, instead of statically deserializing values to their indicated type. This means that a record `MyEvent` can be correctly deserialized from a value that is statically typed as `Event` but dynamically tagged as `MyEvent`.
- The `gradle-sic-plugin` has been updated to better handle multiproject builds.

5.3.11 0.50.0 - 2019-11-08

Changed

- The `csl-plugin` has been removed. It is superseded by the new gradle plugin `gradle-sic-plugin`.
- `sic` output is now available when using `--info` with the gradle plugin.

Fixed

- `sic` no longer generates optional parameters for the TypeScript target which caused problems when mixing optional and required parameters.

5.3.12 0.49.0 - 2019-10-29

Added

- New record extension syntax for carrying over some of the fields of an existing record into a new one: `R { use x with a = 1, b = 2 }`.

Fixed

- Corrected a bug that prevented instantiation of contracts with `sic` generated code on Corda nodes.

5.3.13 0.48.0 - 2019-10-24

Added

- `DateTime::dayCountFraction` can now handle the `Act/Act ICMA` convention.

Fixed

- Corrected a bug in `toContractAST` that caused a crash when working with nullable `or` branches.

Changed

- The concept of "pseudo value" has been removed in favor of "external objects". Bound names are no longer to be specified for agents, contracts, etc. represented as CSL values.
- The `DateTime::dayCountFraction` function now only takes one parameter. The usual `from` and `to` parameters should now be applied to the convention constructor. I.e. `DateTime::dayCountFraction (DateTime::ActActISDA #2007-12-28# #2008-12-28#)`.

5.3.14 0.47.0 - 2019-10-10

Fixed

- Corrected a bug in `DayCountFraction` calculations that occurred when start and end dates are in the same year.
- Fixed a bug in `sic` that caused it to not generate code for reports and entry points that use one of the built-in types as input or output types:
  - `Ordering`,
  - `DateTime::DayOfWeek`, `DateTime::Components`, `DateTime::DayCountFraction`,
  - `Event`,
  - `Duration::Components`,
  - `RuntimeError`.

5.3.15 0.46.0 - 2019-09-27

Removed

- The `sic-maven-plugin` has been removed. It is superseded by the gradle plugin `csl-plugin`.
- The standard library function `getEvents` has been removed.

5.3.16 0.45.0 - 2019-09-03

Fixed

- Corrected a bug in `DateTime::dayCountFraction`.

Changed

- Renamed `DateTime::dayCountFraction` convention `Thirty360ISMA` to `Thirty360ICMA`.

Added

- A list of past events was added to the contract view in the webapp.

5.3.17 0.44.0 - 2019-09-03

Added

- Added the standard library function `DateTime::dayCountFraction` that calculates Day Count Fraction for a number of conventions.

Changed

- The `sic` "JSON" emitter has been renamed to "CoreAST".

Fixed

- Corrected a bug in the publication settings for one of the JAR dependencies which caused the generated CorDapps to break.

5.3.18 0.43.0 - 2019-08-19

Added

- Ontology backend to `sic` that emits as JSON the intermediate contract descriptor used by the `sic` emitters.

Changed

- The `sic`-generated Kotlin code does not generate an `interface` with `*Impl` class implementations, but just the class implementations directly without the `*Impl` suffix.
- The structure in multi-file `deon-projects` is now more closely reflected in `sic`-generated TypeScript/Kotlin code.
- Both the TypeScript and Kotlin code generated by `sic` introduces an `addDeclaration` function for putting a CSL declaration on the ledger. The instantiation classes now require a declaration id in their constructors.
- When `sic` outputs the JSON-representation of an AST to standard out it does not include the separator strings that are invalid JSON.

Fixed

- Corrected a bug in the build setup that rendered the Maven-published JAR of `deon-api-client` unusable.

5.3.19 0.42.0 - 2019-08-05

Added

- The LSP server now supports go to definition.
- Added an endpoint `contracts/<contractID>/events` to get the event that have been applied to a contract.

Changed

- `sic` now writes multiple files instead of merging everything into one file. Dependencies on `sic` generated files will have to be updated accordingly: For TypeScript, the files are renamed; and for Kotlin projects, the generated namespace has changed.
- The helper functions for serialisation/deserialisation generated by TypeScript backend in `sic` have been refactored, so `sic` users will need to make a few changes to their TypeScript code. Instead of `mkXValue` and `getX` there are now methods `X.toValue` and `X.fromValue`. Also `getAgent` has been replaced by `Agent.fromValue`.

Fixed

- Corrected the code that converts `StateTree` based residual contracts to their abstract syntax form.

5.3.20 0.41.0 - 2019-07-10

Added

- The Visual Studio Code plugin now shows type information on mouse hover.

Fixed

- Made sure the Gradle plugin for generating CorDapps includes everything needed to generate them correctly.
- Fixed a bug in the TypeScript emitter of `sic` where records using TypeScript keywords as field names would not function properly.

5.3.21 0.40.0 - 2019-06-28

Added

- Initial version of the new Datalog-inspired query language.
- Gradle plugin to generate self-contained CorDapps directly from CSL code.
- A new backend to `sic` that emits parsed and compiled CSL syntax as a JSON-file.

Changed

- Upgraded Corda to 4.0.
- Upgraded Kotlin runtime to 1.2.

Removed

- The generic Corda ledger backend has been removed.
- The Hyperledger Fabric ledger backend has been removed.
- The special treatment of `Evidence` records has been removed from the Corda backend.
- The example oracle implementations have been deleted.
- The `InMemory` backend has been removed. It's superseded by the `DbLedger` backend.

Fixed

- Fixed an issue where the service would fail in an unexpected way when fed with certain invalid `DateTimes`.
- Fixed an issue where in some cases the type checker would not report error messages in expressions involving records.

5.3.22 0.39.0 - 2019-04-26

Changed

- Added support for simple simplifications of contracts. The simplifications are used for the contracts presented in the webapp in the "Actions" and "Viewer" tab.

Fixed

- Fixed an issue where lists in events (records) constructed from the webapp would not be handled correctly.
- Fixed an issue where the type checker would allow complex patterns at top-level. This is no longer allowed as it was in the old type checker.

5.3.23 0.38.0 - 2019-04-11

Fixed

- Fixed an issue with floating point equality where numbers were not normalized.
- Functions can now be serialized which makes it possible to call templates with functions as arguments.

Changed

- Contracts are now represented more efficiently on the ledger.
- The evaluator uses a new approach with a global environment instead of closures.

5.3.24 0.37.0 - 2019-04-10

Fixed

- Fixed an issue with floating point arithmetic operations where they would not use decimal128 rounding.

5.3.25 0.36.0 - 2019-04-09

Fixed

- Fixed an issue where the type checker would crash instead of reporting an error when a constructor pattern missed an argument.
- Fixed an issue where the language server would report errors in the wrong file.

Changed

- The underlying representation of floating point numbers was changed to decimal128 from binary64, allowing more exact financial calculations.

5.3.26 0.35.0 - 2019-04-05

Fixed

- Fixed an issue where a unary minus was not pretty printed correctly.
- Fixed an issue with applying events to contracts with inner lets

Changed

- `sic` for TypeScript now generates classes instead of type aliases for records.

Removed

- Removed the `corda-light` ledger backend. It was not being used and was obstructing further developments and randomly failing CI.

5.3.27 0.34.0 - 2019-03-21

Fixed

- Fixed an issue in the new type checker where record projection, upcast and type case did not work as expected in certain cases.

5.3.28 0.33.0 - 2019-03-21

Changed

- The type checker was replaced. It reports multiple error messages and returns an AST annotated with types.

5.3.29 0.32.0 - 2019-03-15

Changed

- The `sic` tool now shows warnings on the console when it cannot generate entrypoints.
- We now disallow shadowing of names in CSL. Shadowing is only disallowed within each name kind. That means value expression names cannot shadow other value expression names but contracts and values can still use the same name.

5.3.30 0.31.0 - 2019-03-12

Fixed

- Fixed a bug that caused a significant slowdown when evaluating some reports.

Changed

- Record types in `sic`-generated code implements the subtyping hierarchy of the CSL source with native target language constructs.

5.3.31 0.30.0 - 2019-03-07

Fixed

- Fixed an issue that caused the project structure of `deon-project` files to not be respected in some scenarios.
- Fixed a bug in the webapp where it would crash when adding a boolean argument to a report.
- Fixed an issue in the webapp where the wrong declarations were used for instantiating contracts in certain cases.

Changed

- The contract composer in the webapp has a new layout.

5.3.32 0.29.0 - 2019-02-26

Added

- Added `sic`-support for `deon-project` files.

Fixed

- Fixed a bug that caused type errors in `sic`-generated TypeScript code with tuples.
- Fixed a bug that caused `/simplify` to crash on certain combinations of contract-parameters and pseudo-syntactic values.
- Fixed a bug in the agent matching algorithm that made it problematic to have agents with numbers in their name, such as `agent1` and `agent2`.
- Fixed a bug in the webapp that prevented instantiation of contracts with multiple contract ID parameters.
- Fixed a bug in the webapp that made it crash when attempting to provide a `DateTime` argument to a report.
- Fixed a bug in the webapp where the contents of AST nodes could overflow if it was too large - now the full content will only be visible when the mouse is hovering over the node.

Changed

- Declarations are identified by content hashes (SHA-256) instead of UUIDs.
- Upgraded Fabric to version 1.4.
- The data model for check errors (type errors, parse errors, etc.) has changed. See OpenAPI specification for the new model.

5.3.33 0.28.0 - 2019-01-15

Added

- Add support for heterogeneous tuple values in CSL.
- Added support for event tags in `sic`. Kotlin `sic` users will need small adjustments to their code.

Fixed

- Fixed an issue with serialization of websocket values.
- Fixed a performance issue in the `/contracts` endpoint that caused it to compute a lot of unnecessary data.

Changed

- Updated the cache policy for static resources such as documentation and examples to prevent browsers from showing outdated versions.
- Updated webapp to react 16.7.0
- Changed default database backend from H2 to SQLite for the DBLedger.

5.3.34 0.27.0 - 2018-12-13

Fixed

- Fixed duplications of agents on agent list on Fabric.
- Fixed too liberal type checking of signed data.

Changed

- Upgraded Corda to version 3.3.
- Removed dependency on Jackson and improved performance of JSON serialization.

5.3.35 0.26.0 - 2018-12-03

Added

- Added support for signing of structured data.

Fixed

- Fixed an issue with instantiation with pseudo-syntactic values in records or lists.

Changed

- Updated API specification to OpenAPI 3 format. The specification endpoint is changed from `/v2/api-docs` to `/openapi.json`.

5.3.36 0.25.1 - 2018-11-23

Fixed

- Fixed a rendering issue with contract ids in the instantiation UI.

5.3.37 0.25.0 - 2018-11-22

Added

- Added the power and square root functions; `Math::pow` and `Math::sqrt` to the standard library.

Fixed

- `sic`: Fixed a bug in the conversion of values in the TypeScript backend.
- Fixed broken link to PDF language guide in the HTML language guide.
- Corrected an error in the UI Actions tab.

5.3.38 0.24.0 - 2018-11-19

Changed

- `sic`: It is no longer necessary to wrap contract ids in the Kotlin interface for reports and event application.
- Changed layout of instantiation screen to handle templates with many arguments in the UI.

Fixed

- Fixed a bug that caused the Kotlin API client to break when it was initialized with a request timeout.
- Fixed a bug when serializing durations with jackson.

5.3.39 0.23.0 - 2018-11-07

Changed

- `sic`: Event application functions in Kotlin and TypeScript are no longer lower-cased but follow the capitalization of their CSL types.

Fixed

- Fixed a bug in the TypeScript backend to `sic` that caused reports that return `Pair` or `Maybe` values to fail at runtime.
- Fixed a bug with event application that would sometimes cause a "variable not found in environment" error.
- Fixed a bug in the Kotlin backend to `sic` where the result would not compile when agents were compared.

Removed

- The "Apply Event as Text" button has been removed from the webapp because it is no longer possible to construct an event value within the syntax of the language.

5.3.40 0.22.0 - 2018-10-10

Added

- A new instantiation argument which resolves to the id of the instantiated contract has been added.
- The Visual Studio Code extension now supports projects spanning multiple CSL files (see the plugin's README for more information).
- Tutorial section about the tool `sic` to the language guide.

Fixed

- Fixed a bug that caused Corda nodes to fail when using lists in events.

Changed

- Agents in contracts are no longer strings, but are abstract values issued by the ledger. Use the new `/agents` API to retrieve the available agent values.
- It is no longer possible to use the special variable `events` to refer to the events of the contract instance; use `getEvents` instead.
- `Contracts::getEvents` has been renamed to `getEvents`.
- It is no longer possible to redeclare top-level value bindings.
- Contract ids are no longer strings but instead abstract values of the type `ContractId` that cannot be constructed in syntax.

5.3.41 0.21.0 - 2018-09-21

Added

- Web UI: Contract ID selector in "Reports" tab.
- It is now possible to specify a separate `peers` list for each contract instantiation in the code generated by `sic`.

Fixed

- Fixed a bug where the webapp could not apply events containing data with qualified sum types.
- Fixed a bug in `sic` that caused empty CSL records to be handled wrong.

5.3.42 0.20.0 - 2018-09-14

Added

- Lightweight implementation of Corda adapter where the notary is only used to ensure consensus on the ordering of events for contracts and not validity of the sequence of events.
- Added ISO 8601 compliant syntax for duration literals.
- Added built-ins and standard library functions for working with durations.

Fixed

- Cucumber: ContractIDs are replaced in the expected results of Report tests.
- Fixed a bug in the scoping of event bindings in prefix expressions.

5.3.43 0.19.0 - 2018-08-31

Added

- TypeScript backend to `sic`.
- Upgraded Corda to version 3.2.
- Upgraded Fabric to version 1.2.
- Added support for Fabric in deployment-tool.
- Support for batching in Kotlin-client.

Changed

- It is now only possible to query `Contracts::getEvents` on contracts that are instantiated from the same declaration ID as the caller.
- Updated the formatting and structure of the CSL language guide and reference.

Fixed

- Fixed a bug where agents represented by variables could be shadowed by bound event names.
- Fixed a bug in the serialization of contracts.
- Fixed a bug in `sic` that caused Kotlin code generated from CSL that used `Floats` to break.
- Fixed a bug in `sic` that caused it to generate invalid code due to keyword name clashes.
- Fixed a bug in the corda-adapter that could break the client-to-node communication under high load

5.3.44 0.18.0 - 2018-08-13

Added

- Deployment tool.
- The code generation tool `sic` for automatically generating a Kotlin interface for a CSL contract.

Changed

- Upgraded Corda to version 3.1.

Fixed

- Fixed a bug that crashed the UI if the residual contract could not be computed.
- Fixed a bug where lists of events could not be converted from the internal to the external representation in some cases (like `List::append events events`).
- Fixed a bug in the web UI where events could not be applied as raw text.

5.3.45 0.17.0 - 2018-06-25

Added

- The API methods `/contracts` and `/contracts/{id}` now return the instantiation time of the contract.
- The standard library now has a `Maybe::bind` function. See the documentation for details.
- A built-in function `error : RuntimeError -> a` has been added.

Changed

- The API methods `/contracts/report` and `/contracts/{id}/report` now accept an optional list of `Values` that can be used as arguments for reports that take parameters.

Removed

- The REST API clients have been moved to GitHub.

Fixed

- A bug which caused the type checker to refuse references to previously declared templates from locally defined templates has been fixed.
- A bug where the simplifier would cause run-time errors to disappear has been fixed.
- Corrected a bug where server side errors could break the UI.

5.3.46 0.16.0 - 2018-06-05

Changed

- The syntax for declaring contracts, templates and values has been changed. Recursive contract templates now have to be explicitly marked as so, and mutually recursive templates must be declared together. See the language guide for more information.
- The `Add event` button has been removed from the `Viewer` menu as it duplicates the functionality on `Actions`. Events can be added as raw text from `Actions` now.
- Removed the API method `/parse-value` that was equivalent to `/report` in taking an expression as string and returning a `Value`.
- Renamed the `/contract/{id}/simplified` endpoint to `/contract/{id}/src` endpoint and added an optional parameter `simplified` such that `/contract/{id}/src?simplified=true` returns the simplified residual contract and `/contract/{id}/src` returns the non-simplified residual contract.

Fixed

- Fixed an issue in the (Kotlin) implementation of `apply` that could result in a crash from a stack overflow.

5.3.47 0.15.0 - 2018-05-28

Added

- The UI now displays the current version number.

Changed

- Performance of the CSL parser has been improved.
- Performance of report and expression evaluation has been improved.
- The endpoints `/check`, `/check-expression`, and `/parse-value` will now run type check on successfully parsed declaration parts even if there is a parse error later.

Fixed

- Fixed a bug where the Visual Studio Code plugin was not in contact with the parser and type checker.

5.3.48 0.14.0 - 2018-05-17

Added

- The standard library now has `List::isEmpty` function.

Fixed

- The in-memory ledger and database-backed ledger backend now correctly handle tagged event application.

Changed

- The Node.JS and web REST clients now support optional tags for `applyEvent`.
- Performance of the `/checkEvent` endpoint has been improved.

5.3.49 0.13.0 - 2018-05-14

Added

- Events are now type checked before being applied to contracts.
- The web editor now supports comment toggling with `Cmd-/` or `Ctrl-/`.
- Standard library function `DateTime::dayOfWeek` that gives the day of week (`DateTime::DayOfWeek`) for a `DateTime`.

5.3.50 0.12.0 - 2018-05-07

Added

- The ontology for a CSL program now includes sum types in addition to record types.
- New event input form in "Actions" tab.
- New database-backed ledger backend with support for H2 and SQLite.
- Node.js REST client.
- Added new endpoints to submit contracts and events in batches for faster execution
- Convenience function to JavaScript SDK for handling `Value` objects as JSON data.

Fixed

- Fixed a bug where applying many events to a contract could cause an exception.
- Fixed some bugs where a couple of endpoints in the API client NPM package did not work.
- Fixed bug where contract instantiation would result in an illegal contract due to a bug in the implementation of scoping.

Changed

- Moved endpoint to simulate events, and added the possibility to simulate events across multiple contracts.
- When running reports, the optional parameter for declaration id is now an optional path parameter. The client SDK has been updated to reflect this.

5.3.51 0.11.0 - 2018-04-24

Added

- Shorthand syntax for `DateTime` literals in CSL. The following all denote the same point in time: `#2018#`, `#2018-01#`, `#2018-01-01#`, `#2018-01-01T00#`, `#2018-01-01T00:00#`, `#2018-01-01T00:00:00#`, and `#2018-01-01T00:00:00Z#`.
- Added "guardedness check" to ensure that contracts do not contain infinite recursion.
- Support for unary minus.

Fixed

- Fixed a bug where the UI's pretty printing of `DateTime` values were not syntactically valid.
- Fixed bug where contract instances in the web app AST view were not simplified.
- Fixed bug where the order of arguments to constructors were reversed when converting from value to expression.

5.3.52 0.10.0 - 2018-04-18

Added

- Contract abbreviations can now be defined in `let` blocks and top-level definitions.
- Contract templates with no contract parameters can now also be defined using the `template` keyword.
- CSL extension to Visual Studio Code.
- Node.js package with an API client.
- Contract examples are now available in the UI.
- Report examples in the language guide.
- Added type checking to contract instantiation, preventing you from passing illegal arguments to contracts.
- Added the possibility to run a simulation of event application without modifying a contract on the ledger.

Fixed

- Fixed an issue where contract names and module names could be in conflict, thus causing bugs when applying events to those contracts.
- Changed the parser such that ranges of tokens don't include trailing whitespace.

Changed

- It is no longer possible to do comparison of strings like `"a" > "b"`.
- Locations in error messages have been improved to be more specific.

5.3.53 0.9.0 - 2018-03-21

Added

- Standard library extended with `Int::toString` function.

Changed

- The `/contracts/{id}/next-events` and `/contracts/{id}/tree` endpoints will no longer surround agent matcher with < and >.

Fixed

- Fixed an issue where result values from report evaluation were not rendered correctly.
- Fixed an issue where cross origin resource sharing (CORS) headers were not included in the webservice HTTP responses.
- Local contract and template declarations would not overwrite global ones in the type checker.

5.3.54 0.8.0 - 2018-03-13

Added

- The UI now has a link to the OpenAPI (Swagger) documentation for the webservice API.
- Added GET `/declarations/{id}` endpoint to the webservice API.
- Reports and values are now type checked.
- Evaluation of reports independently of specific contract instances:
  - Added POST `/contracts/report` endpoint to the webservice API for evaluating reports.
  - "Reports" tab in UI has been updated with option to evaluate reports independently.
- The standard library now has `String::append` function.

Changed

- The webservice API has been refactored and documented:
  - Input/output models have new names, e.g. `Declaration` and `Contract` instead of `NamedCSL` and `NamedContract`.
  - Contracts can only be instantiated with a declaration ID - not with a CSL string.
  - Contract AST are now retrieved through `/contracts/{id}/tree`.
  - Next events for contracts are now retrieved through `/contracts/{id}/next-events`.
  - Endpoints now return HTTP codes 4XX when the input is invalid, e.g., 404 for invalid ID's, 400 for failed type checking, etc.
  - Input bodies are always objects with named fields.
  - `/templates` endpoints are now `/declarations` endpoints.
  - `/peers` and `/ledgerInfo` have been merged to `/node-info`.
  - `/csl` is now `/csl/check`.
  - `/parseReport` is now `/csl/check-expression`.
  - `/value` is now `/csl/parse-value`.
  - `/{id}/ontology` is now `/declarations/{id}/ontology`.
  - `/contract/{id}/report` is now `/contracts/{id}/report`.
  - OpenAPI (Swagger) documentation has been updated with examples and data models.

  - Qualified named for `Records`, `Lists`, `types` are now sent as structured objects instead of strings, e.g. `A::a` is now `{qualifier: ["A"], name: "a"}`.

- The UI now refers to "declarations" instead of "templates".

Removed

- The DELETE endpoints `/contracts/{id}` and `/templates/{id}` have been removed from the webservice API.

Fixed

- Fixed an issue where a contract could not be instantiated if a type with the same name is present.
- Fixed an encoding issue where some characters could not be used in CSL.
- Fixed a bug in the type checker where sum type constructors inside modules where globally scoped.

5.3.55 0.7.1 - 2018-03-12

Fixed

- Fixed concurrency issue in InMemory-ledger.

5.3.56 0.7.0 - 2018-03-06

Added

- Structural equality relations for lists of base types added to standard library.
- List shorthand: `[1,2,3]` is shorthand for `Cons 1 (Cons 2 (Cons 3 Nil))`.
- Upcast annotation (`:>`) for records, telling the type checker to consider a record expression as a super record, e.g. `DeliveryEvent {...} :> Event`.

Fixed

- Fixed issues in relation to expansion of templates. `template[c] Foo() = (<*> E then success) then c` would not work as expected.
- Fixed issue where `((<*> E then success) or success) then C` would accept `C` and become `((<*> E then success) or success) then success` but it should have become simply `success`.

5.3.57 0.6.0 - 2018-02-23

Added

- The UI now has a link to the language guide.
- The UI now has a link to the change log (this document).
- The language guide now contains a grammar for CSL.

Changed

- Upgraded Corda to v2.0.0.
- Changed keyword `fail` to `failure` for consistency with `success`. Both `success` and `failure` are nouns.
- Prefix contracts do not include the `then` keyword. Therefore, the following two are now equivalent `<*> E` and `<*> E then success`.

Fixed

- Fix a serialization issue between `FabricLedgerService` and `chaincode`.
- Removed argument field names at instantiation.

5.3.58 0.5.0 - 2018-02-13

Added

- Support for the submission of event subtypes from the web UI if the contract's ontology specifies them.
- Added new ledger service `FabricLedgerService` that supports CSL runtime on Hyperledger Fabric.
- Serve language-guide with webservice (http://webservice:8080/docs/guide.html).
- Parse and type errors are now highlighted in the web UI.

Changed

- The `Order` type in the standard library has been renamed to `Ordering`.

Fixed

- Significantly improve the speed with which contract source code is printed (roughly ~100x). This should improve the performance of cucumber tests.
- Fix a bug that made it impossible to instantiate contracts with `DateTime` parameters in the web UI.

5.3.59 0.4.0 - 2018-02-06

Changed

- Updated the CSL standard library to version 0.2, with the following changes:
  - Renamed module `Time` to `DateTime`, and `Time::TimeComponents` to `DateTime::Components`.
  - Added `DateTime::addDays`.
  - Added general combinators: `id`, `const`, `flip`.
  - Added `Pair`, and projection functions `fst`, `snd`.
  - Added `Maybe` with functions `maybe`, `fromMaybe`, `Maybe::map`, `Maybe::isDefined`, `Maybe::any`, `Maybe::all`.
  - Added `compareInt`, `compareFloat`, and `compareDateTime`.

  - Added the following functions to `List` module: `head`, `headOrDefault`, `tail`, `length`, `map`, `mapMaybe`, `filter`, `zipWith`, `any`, `all`, `first`, `last`, `append`, `concat`, `reverse`, `take`, `drop`.
  - Added documentation.
  - Indentation and formatting.
- The value parser now uses CSL2 syntax. This means events should be entered in CSL2 syntax with uppercase constructors e.g. `True` `False` `Nil` and `Cons`.

Removed

- The CSL1 language level.

Fixed

- Fix a bug that caused the ontology viewer to not be shown when a record type explicitly specified `Record` as its parent.

5.3.60 0.3.0 - 2018-02-05

Added

- In `com.deondigital.api` a `ConstructorValue` is now available to create and read CSL-constructors applied to argument values.
- Record subtyping, so that a record subtype can match the type pattern of its supertype, and an event subtype can be applied to a contract accepting its supertype.

Fixed

- The reports tab would report an error when the result of a report contained any constructor which was not `True`, `False`, `Nil` or `Cons`.