Cover page

Exam information
NDAA93062E - Computer Science Thesis 30 ECTS, Department of Computer Science - Kontrakt:125444 (Jens Kanstrup Larsen)

Handed in by
Jens Kanstrup Larsen
bgs509@alumni.ku.dk

Exam administrators
DIKU Eksamen
uddannelse@diku.dk

Assessors
Fritz Henglein
Examiner
henglein@di.ku.dk
+4535335692

Patrick Bahr
Co-examiner
paba@itu.dk

Hand-in information

Titel: Prototypeimplementering og anvendelse af Contract Specification Language CSL2
Titel, engelsk: Prototype Implementation and Application of Contract Specification Language CSL2
Vejleder / eksaminator: Fritz Henglein
Tro og love-erklæring: Yes
Indeholder besvarelsen fortroligt materiale: Yes
Må besvarelsen gøres til genstand for udlån: No
Må besvarelsen bruges til undervisning: No

UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Computer Science Thesis, 30 ECTS

Jens Kanstrup Larsen `<bgs509@ku.dk>`

MSc in Computer Science, DIKU

Prototype Implementation and Application of Contract Specification Language CSL2

Advisor: Fritz Henglein and Ulrik Terp Rasmussen

May 31, 2021, Block 3+4

Contents

1 Abstract

With the increasing popularity of distributed ledger technology, financial contracts may now be specified using virtual contracts maintained over such a distributed ledger; these virtual contracts are also known as smart contracts. However, a communication gap presents itself between the domain experts that specify the equivalent ”paper” contracts, and the programmers who must implement them. Domain specific languages for writing smart contracts may help bridge this gap. Based on CSL (Contract Specification Language), a domain specific language developed by Deon Digital, we develop CSL2, which extends CSL with output states, a useful property for contract specification. We first define the operational semantics and type system for the language, and then implement the language as a deeply-embedded DSL in Haskell. We also introduce a novel algebraic data type for expressing dates. Finally, we present a variety of financial contracts specified using CSL2, and argue for their readability.

2 Acknowledgements

First, I would like to thank my supervisor, Fritz Henglein, for supporting and guiding me throughout the project. I also give my thanks to Deon Digital, who provided me with this unique opportunity for venturing into real-world DSL design, and the team of Ulrik, Juan and Agata who gave invaluable help during my time working with them. Finally, I would like to thank my family of Steen, Mette, Michael, Peter and Ellen, who have all been by my side and supported me throughout my work on this thesis.

3 Introduction

The increasing popularity of distributed ledger technology [1] has led to focus on *smart financial contracts*, which describe sequences and rules of transactions between multiple parties. The languages used for writing such smart contracts are usually general purpose programming languages built on top of the relevant blockchain system; as such, the contracts are often syntactically similar to ”ordinary” computer programs. However, in the case of financial contracts, the lawyers and accountants who write and verify the ”paper” contracts will not be able to read or verify themselves that the computer programs are actually equivalent. Likewise, the programmers who write the smart contracts may have some broad idea of or even an algorithm that describes how the contract should behave, but they do not have the necessary financial background to verify its correctness. This leaves us with a communication gap. To bridge this gap, a domain-specific language for expressing financial contracts, which is readable by both programmers and financial experts, is necessary.

CSL (Contract Specification Language) is a domain-specific language developed by Deon Digital, designed for writing legal smart contracts. The design goal of the language is to have a both human- and machine-readable language, while still maintaining a formal mathematical semantics such that program properties can be statically analyzed[1].

Nonetheless, CSL still has some limitations due to its design. This thesis describes the specification and implementation of **CSL2**, an evolution of CSL. Specifically, CSL2 introduces the following feature of **output states**, allowing contracts to evaluate to any arbitrary value, rather than just success or failure. This may seem inconsequential at first, but it allows for expressing a much greater variety of contracts, as well as improving the readability of existing contracts.

In this thesis, we will use established knowledge on DSL design to develop CSL2, an evolution of CSL. We will present both mathematical semantics and a fully implemented Haskell semantics, and demonstrate the expressiveness of the language by modelling different types of financial contracts.

First, we give a background on the literature on DSL design. We focus specifically on cases of embedded DSLs, going over multiple different approaches for language syntax. We also establish a semantic model, which will work as the foundation of our language design. From here, we consider the difference between deep and shallow embedding, choosing the one most suitable for our purpose.

We then give a background on the existing CSL, focusing on its syntax and mathematical semantics. We also give an overview of its shortcomings, and how CSL2 may be designed to help overcome these.

Afterwards, we present the grammar and operational semantics of CSL2, this being both big-step and small-step semantics, as well as a typing system. From these, we prove that the CSL2 contract type is a monad by showing that it obeys the monad laws. We then present the Haskell implementation of the semantics, discuss implementation strategies and possible alternative implementations. Lastly, we present the intended Haskell syntax for CSL2 and argue for its readability.

Following this, we describe the different algebraic datatypes used in the CSL2 implementation, focusing on how dates and durations are represented. We also discuss how both resource transfers and exact decimal arithmetic are or may be implemented.

Finally, we showcase a variety of different financial contracts such as bonds and warrants

[1] https://www.deondigital.com/technology/

implemented in CSL2. We consider these implementations from several standpoints: ”Are they readable?”, and ”Can they express what we want?”.

4 Background

4.1 Developing a domain-specific language

Starting development of a domain-specific language should only be done after careful consideration. Depending on the type and scope of the DSL, developing an implementation may require relatively many working hours, which could have been used on implementing a solution in a general purpose language instead. For this reason, it is important to always consider *why* a DSL should be developed, and what benefits can be gained from such a DSL.

The common reason for developing a DSL is that it would be able to accomplish a certain task more efficiently than a general-purpose language. The ambiguity of the task and what constitutes ”efficiency” is not unintended; this may be very different for each DSL case. Unlike a general-purpose language, which may aim to provide many different types of functionality, a domain-specific language is purposed around one specific domain or *semantic model*, which allows it to specialize. A semantic model is, as the name suggests, the underlying phenomenon that we want the DSL to model. For example, if we want to describe a certain type of financial bond with our DSL, the semantic model is the concept of the bond itself, which can in turn be described by an appropriate legal contract or another abstract representation. For example, a financial option would have the semantic model as seen in figure 1.

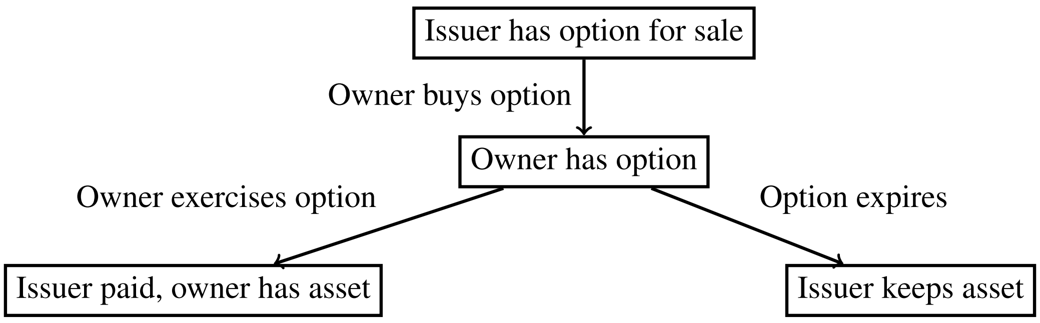

Figure 1: A representation of a semantic model based on a financial option.

For CSL, the semantic model would be not just the construction, but also the the evaluation of contracts, i.e. how the state of the contracts change with respect to incoming events. A DSL that can accurately describe its underlying semantic model is said to be *domain adequate*.

A design goal for DSLs may also be to develop a syntax which allows domain experts, in this case lawyers or economists, to understand and preferably write their own contracts in the language. However, just as COBOL did not eliminate the need for programmers in the business sector, it is unlikely that CSL2 will eliminate the need for programmers in the legal sector. Instead, the aim of CSL2 should be to bridge the gap between contract programmers and lawyers. Lawyers, while not able to write their own programs, should be able to read CSL2 code and understand its meaning. Programmers, while not able to write their own legal contracts, should be able to receive a specification and translate it into correct CSL2 code. This is precisely the reason why the syntax of the DSL should be given heavy consideration. Fowler and Parsons [2] use the term *fluent interface* to describe DSL code that has a ”sentence-like” structure, instead of being a sequence of unconnected function calls. Having a fluent interface also provides a much better, at least clearer, method for writing programs, compared to using an existing markup language such as XML. Even though you may be able to express the same kind of programs in either language, XML can be considered cumbersome to read, while a DSL can be given a syntax with more ”flow”.

Limiting the expressive power of a DSL is also important, for multiple reasons. Since one of the main reasons behind developing a DSL is to create a relevant interface for the semantic model, this interface should be simple, yet targeted. Having a well-designed syntax makes the DSL easier to read at a glance compared to a general-purpose language. Not only does it make the DSL code easier to write and debug for regular programmers, but it also allows the domain experts to partake in the development, and in some cases contribute directly to the software.

The second reason is from an academic perspective. With a limited language, it is easier to reason about the semantic behavior of programs written in it, compared to if the same program was written in a general-purpose language. Having limited semantics opens up for greater reasoning about program behavior; in particular, it may become possible to *prove* some behavior of a DSL program. This provability is especially desirable in the field of contract specification, as it can provide a guarantee that the contract is correct and does not produce any undesireable outcome for either party.

Since the semantic model may vary greatly depending on the domain, it stands to reason that there are also many possible DSL design choices. Spinellis [3] identifies a set of design patterns for DSL building; of these, three are *creational* patterns for constructing DSLs:

- **Lexical processing**:
  DSLs implemented using lexical processing define their own unique syntax, which is then parsed with a string processor in another base language. Often, the base language is a language that has access to string-processing or parsing libraries, such as Python. DSLs using this pattern are often very syntactically lightweight; ”programs” in these languages are often only single-line and deliimited by a set of token characters, rather than being multi-line programs.

- **Language extension**:
  DSLs using the language extension pattern are implemented in an existing language, rather than as a separate language. These are also referred to as *embedded* languages. This removes the need for implementing a separate parser, but it also means that the DSLs must conform to the syntax of the base language. The pattern is for this reason often used with base languages where new data types and ”syntactic sugar” are easily defined.

- **Language specialization**:
  The language specialization pattern can be considered the opposite of the language extension pattern: rather than adding features, it removes features. These can be features such as dynamic memory allocation that make the base language ”unsafe”, and are unnecessary within the domain of the DSL. Also contrary to language extension is that execution can be done with the same compiler or interpreter as the base language, but a parser or some other ”verifier” must be implemented to ensure that DSL programs only use the correct base language subset.

Since financial contracts are significantly larger than what can be expressed on one line (at least without making the DSL unreadable), the option must be between a language extension or a language specialization. Given that our goal is to make the contracts readable by domain experts, we need to be able to define custom types and functions that can construct such readable contracts. For this reason, we choose to use the **language extension** pattern and develop an embedded DSL with **Haskell** as the base language.

Why choose Haskell? Let us consider the goal of the project: to *prototype* an implementation of CSL2, meaning that the end result may be subject to change. We want to avoid spending too much time on developing an implementation that is changed later on, so we must take an approach that allows for quick development, yet does not compromise too much on possible language features. A study done by Kosar et al. [4] has participants develop a DSL for the same semantic model, but with very different approaches. Of these, the approach which uses an embedded DSL in Haskell has the lowest code size by a large margin, indicating that it is suitable for a project where a quick implementation is desirable.

Overall, there is a significant difference between developing external and embedded DSLs. For external DSLs, the developer has full control over the syntax and method of evaluation. However, an external DSL also comes with the cost of developing a language parser, as well as a compiler or interpreter.

A benefit of using an embedded DSL is that the DSL parsing will be done together with the base language. This constrains the options for the DSL syntax, as it has to be parseable by the parser of the base language. However, this does not mean that a fluent interface is not achievable. We present some methods for achieving a fluent interface with an embedded DSL:

Consider the following natural language description of a simple contract:

Bob is paid 100 DKK, after which a bike is delivered to Alice.

Using three different strategies described by Fowler and Parsons [2] for implementing embedded DSLs with fluent interfaces, we present three ways of writing the above contract:

1. **Method Chaining:**
   Using this strategy, method calls provide intermediary objects that can provide new method calls, allowing the user to structure a syntax which looks similar to an external DSL. Using the bike-contract example, an example is given below:

```
payment()
  .suchthat()
    .receiver("bob")
    .amount(100)
.then()
.delivery()
  .suchthat()
    .receiver("alice")
    .item("bike")
.end();
```

   This exact strategy is not feasible to use for CSL2, since it relies on the language having object-oriented features, which our base language, Haskell, does not.

2. **Function Sequence** This strategy uses independent function calls to construct the DSL expression:

   ```
   payment();
     suchthat();
       receiver("bob");
       amount(100);
   then();
   delivery();
     suchthat();
       receiver("alice");
       item("bike");
   ```

   This exact strategy is not usable either, due to the purely functional nature of Haskell, which does not allow stateful global functions.

3. **Nested Function** With this strategy, function calls are nested *within* other function calls, such that the results of inner calls are passed as arguments to outer calls:

   ```
   payment(
     suchthat(
       receiver("bob"),
       amount(100)
     ),
     then(
       delivery(
         suchthat(
           receiver("alice"),
           item("bike")
         )
       )
     )
   );
   ```

   This strategy, unlike the previous ones, is possible to implement in Haskell. While the use of tuples as function arguments is somewhat unconventional for Haskell standards, this is not cause for concern, since the CSL2 DSL may be designed intentionally to be distinguishable from ”base” Haskell.

Overall, while the strategies provide some good ideas for language design, most of them are too object-oriented in nature to be properly translated to a purely functional language such as Haskell. However, the syntax and semantics of Haskell also open up for alterative embedded DSL design strategies. We present two strategies uncovered during development:

- **Infix type constructors**

  ```
  Payment
  `suchthat`
  [receiver "bob", amount 100]
  `then`
  Delivery
  ```

```
`suchthat`
[receiver "alice", item "bike"]
```

- **Do-notation**

```
do
payment
  suchthat
    receiver "bob"
    amount 100
then
delivery
  suchthat
    receiver "alice"
    item "bike"
```

Both approaches have different advantages and disadvantages. For infix-type constructors (and functions), there is a litle more grammatical clutter with the accent ` around the calls, as well as the angle brackets; it is not much, but the `do`-notation style has almost nothing in comparison. Although not shown above, the `do`-notation also has the benefit of having arrow notation (<−), which provides a syntax more familiar to programmers with imperative language experience.

4.1.1 Shallow and deep embeddings

After settling on an embedded DSL, it is a question of whether to use a *shallow* or *deep* embedding. Gibbons [5] identifies two important concepts related to deep and shallow embedding: the *language constructs* that are used to ”build” expressions in the language, and *observers*; functions that operate on the language expressions, generally by transforming them from DSL constructs into basic data types such as `Integer` or `Bool`. We present two examples of deep and shallow embedding adapted from the ones presented by Gibbons:

For deep embedding, the constructs are expressed with an abstract syntax tree, usually constructed from an algebraic datatype. For example, consider this simple contract language, which only has the atomic `Event` constructor and the sequential `Then` combinator:

```
data Contract =
    Event String
  | Then String Contract
```

For simplicity, we will assume that all event specifications are represented by strings. The bike transfer from previous examples is then, using infix notation, written as

```
("100kr -> Bob") `Then` (Event "Bike -> Alice")
```

The contract evaluation (whether or not the contract succeeds) would then be written as

```
eval :: [String] -> Contract -> Bool
eval [e] Event e'      = e == e'
eval e:es (Then e' c) = if e == e' then eval es c else False
eval _ _              = False
```

where the first argument the list of events received

. For the shallow embedding, language expressions are not defined using an algebraic datatype, but are instead written directly ”as their semantics”.

```
type Contract = [String] -> Bool

cEvent :: String -> Contract
cEvent s =
  (\events ->
    case events of
      [e] -> s == e
      _    -> False)
cThen :: String -> Contract -> Contract
cThen s c =
  (\events ->
    case events of
      e:es -> if s == e then c es else False
      _    -> False)
```

Evaluating the contract would then be the result of applying the list of events to the contract itself, without the use of an `eval` function or similar. The contract from above would use an almost identical syntax:

```
("100kr -> Bob") `cThen` (cEvent "Bike -> Alice")
```

The main difference between deep and shallow embedding is essentially where the ”workload” lies when extending the language. For deep embedding, it is straightforward to add more functions such as `eval` that operate on contract structures, but extending the syntax itself is troublesome, as all existing functions that use contracts must be extended to cover new contract constructors. Conversely, for shallow embedding, adding new constructs is easy, as you can ”just” write a new function, but adding new functionality is difficult, since this must be built into every construct. For example, consider a function `analyze` which takes a contract and returns a list of all events (strings, in this case). For shallow embedding, this requires changing the type of `Contract` to something else (for example `[String] -> (Bool, [String])`), and updating each of the constructs to match the new output.

Which approach should we choose for our DSL? After some consideration, deep binding would seem to be most appropriate. We do not expect our contract constructors to change often, especially since the atomic constructs rely on an *internal value language* (a separate language from the contract specification language), which can have as much expressiveness as necessary. Additionally, we have not fully determined what functions would be good for function analysis, making it important that such functions can be added and changed without difficulty. Lastly, one of the core concepts of CSL is that contracts can be residualized, meaning that their structure can change as events are applied. Consider an `apply :: String -> Contract -> Contract` function for a deep embedding; if given a string, it essentially acts as a contract transformer, transitioning a contract from one state to the next. We could achieve the same transformation with a shallow embedding, but the contracts we would give and receive would be ”opaque”: we only know that they take a string and return a contract, but the structure of the contract itself would be a mystery.

5 CSL

This section provides an overview of the CSL syntax and semantics, based on the paper by Andersen et al. [6].

CSL is a domain-specific language used for modeling compositional contracts. The language is designed such that it separates *atomic commitments* (constructs that define an atomic event, e.g. a single money transfer) and *contract composition* (operations that combine subcontracts into larger contracts).

The atomic commitment is in essence an *event specification*, which matches received events. Each specification matches against a ”real” event using four identifiers: the sending and receiving *agent*, the *resource* sent/received, and the *time* the event occurred. Furthermore, the event is also matched against a predicate (a boolean expression) where the identifiers from before, as well as identifiers from previously matched events, may be part of it. Note that the syntax and semantics of the boolean expression is not defined by CSL itself, in fact, any language that supports the types *Boolean*, *Agent* ($\mathcal{A}$), *Resource* ($\mathcal{R}$) and *Time* ($\mathcal{T}$) may be used as the expression language. Thus, CSL is *parametric* in its choice of expression language.

The contract language itself, CSL, has the grammar as shown in figure 2.

$$\begin{aligned}
c &::= \text{Success} \mid \text{Failure} \mid f(\mathbf{a}) \mid \\
&\quad\;\; \text{transmit}(A_1, A_2, R, T \mid P).c \mid \\
&\quad\;\; c_1 + c_2 \mid c_1 || c_2 \mid c_1 ; c_2 \\
D &::= \{f_i[\mathbf{X}_i] = c_i\}_i \\
r &::= \text{letrec } D \text{ in } c
\end{aligned}$$

Figure 2: CSL grammar.

Success denotes the trivially completed contract, meaning that it requires no further commitments to be fulfilled. Likewise, Failure denotes the failed or breached contract, which cannot be fulfilled. $f(\mathbf{a})$ instantiates a *contract template* f with the vector of arguments **a**. transmit($A_1, A_2, R, T \mid P$).c specifies that a transmission of resource R should occur from agent A_1 to agent A_2 at time T, while also satisfying the boolean predicate P. ’transmit’ binds the values of the event to the variables A_1, A_2, R and T, such that they can occur in both P and c. The *contract combinators* $c_1 + c_2, c_1||c_2, c_1; c_2$ combine two subcontracts. $c_1 + c_2$ denotes a contract of two alternative subcontracts, $c_1||c_2$ two parallel subcontracts, and $c_1; c_2$ two sequential subcontracts. D is the environment that collects contract templates. Contract templates are specified using an identifier f_i, a vector of formal parameters $\mathbf{X}_i$ and a body c_i. A complete contract c along with a collection of templates D is defined as ’letrec D in c’.

The *execution* of all contracts besides trivial ones depend on receiving events that match event specifications in said contracts. Sequences of events are defined as *event traces* or simply *traces*. In this variant of CSL, where we only have one event specification: transmit$(A_1, A_2, R, T \mid P).c$, we also have only one type of event: transmit(a_1, a_2, r, t) (where $a_1, a_2 \in \mathcal{A}$, $r \in \mathcal{R}$ and $t \in \mathcal{T}$), but the language can be extended to support any number of user-defined events and pertaining specifications.

Modelling real-world contracts such as financial contracts requires the language to have some notion of when a contract is "completed" or "satisfied", i.e. the contract *can* be successfully terminated without receiving further events (although the contract must not *necessarily* be terminated). In CSL, such a contract is said to be *nullable*. The system for inferring the nullability of a contract can be seen in figure 3. From the semantics, the formal definition of nullability is that a contract c is nullable if $D \vdash c$: nullable, where D is the contract template environment.

$$\frac{}{D \vdash \text{Success} : \text{nullable}} \qquad \frac{D \vdash c : \text{nullable} \quad (f(\mathbf{X}) = c) \in D}{D \vdash f(\mathbf{a}) : \text{nullable}}$$

$$\frac{D \vdash c : \text{nullable}}{D \vdash c + c' : \text{nullable}} \qquad \frac{D \vdash c' : \text{nullable}}{D \vdash c + c' : \text{nullable}}$$

$$\frac{D \vdash c : \text{nullable} \quad D \vdash c' : \text{nullable}}{D \vdash c || c' : \text{nullable}} \qquad \frac{D \vdash c : \text{nullable} \quad D \vdash c' : \text{nullable}}{D \vdash c; c' : \text{nullable}}$$

Figure 3: System for deriving CSL contract nullability.

The 'Success' construct is trivially nullable. Contract instantiations are nullable if the instantiated contracts themselves are. Alternative contracts are nullable if *either* of the subcontracts are nullable; parallel and sequential contracts are nullable if *both* of the subcontracts are nullable. Note the absence of a rule for the 'transmit$(A_1, A_2, R, T \mid P).c$' construct, since it implies that the contract needs to receive more events and thus is not complete yet.

With the definition of a nullable contract, we can now move on to the semantics for *contract residuation*, which describe how contracts are altered after trace sequence application. After applying a trace sequence to a contract, there is always a remaining (though perhaps trivial) contract. The resulting contract is defined as a *residual contract*. The operational semantics for contract residuation can be seen in figure 4, where c is reduced to c' after applying the event sequence e, if $D, \delta \vdash_D c \xrightarrow{e} c'$, where D is the contract template environment and δ is the variable environment.

Receiving more events after a 'Success' state implies that the "real world agreement" covered by the contract has not yet been satisfied, thus, it immediately transitions to a 'Failure' state. A 'Failure' contract cannot return from the failed state, so it remains in 'Failure' after receiving any event. Matching a 'transmit' with an event binds the values from the event in the enviroment, and then results in the remaining contract, with identifiers substituted with the matched values. Failing to match a transmit immediately results in a failure. Instantiating a contract template first maps the expressions in the vector **a** to basic elements in their domain ($\mathcal{A}$, $\mathcal{R}$ or $\mathcal{T}$) using the denotation function $\mathcal{Q}[\![\mathbf{a}]\!]^{\delta}$, such that we obtain a vector of values **v**. We then pass on the events e to the contract $c[\mathbf{v}/\mathbf{X}]$, which transitions to c' as the resulting contract.

For alternative contracts, the event may be consumed by either contract, so the contract residuates to the alternative between the residuation of the two subcontracts. For parallel

$$\frac{}{D, \delta \vdash_D \text{Success} \xrightarrow{e} \text{Failure}} \qquad \frac{}{D, \delta \vdash_D \text{Failure} \xrightarrow{e} \text{Failure}}$$

$$\frac{\delta \oplus \{\mathbf{X} \mapsto \mathbf{v}\} \models P}{D, \delta \vdash_D \text{transmit}(\mathbf{X}|P).c \xrightarrow{\text{transmit}(\mathbf{v})} c[\mathbf{v}/\mathbf{X}]} \qquad \frac{\delta \oplus \{\mathbf{X} \mapsto \mathbf{v}\} \not\models P}{D, \delta \vdash_D \text{transmit}(\mathbf{X}|P).c \xrightarrow{\text{transmit}(\mathbf{v})} \text{Failure}}$$

$$\frac{D, \delta \vdash_D c[\mathbf{v}/\mathbf{X}] \xrightarrow{e} c' \quad (f(\mathbf{X}) = c) \in D, \mathbf{v} = \mathcal{Q}[\![\mathbf{a}]\!]^{\delta}}{D, \delta \vdash_D f(\mathbf{a}) \xrightarrow{e} c'}$$

$$\frac{D, \delta \vdash_D c \xrightarrow{e} d \quad D, \delta \vdash_D c' \xrightarrow{e} d'}{D, \delta \vdash_D c + c' \xrightarrow{e} d + d'} \qquad \frac{D, \delta \vdash_D c \xrightarrow{e} d \quad D, \delta \vdash_D c' \xrightarrow{e} d'}{D, \delta \vdash_D c||c' \xrightarrow{e} (c||d') + (d||c')}$$

$$\frac{D \vdash c \text{ nullable} \quad D\delta \vdash_D c \xrightarrow{e} d \quad D, \delta \vdash_D c' \xrightarrow{e} d'}{D, \delta \vdash_D c; c' \xrightarrow{e} (d; c') + d'} \qquad \frac{D \not\vdash c \text{ nullable} \quad D\delta \vdash_D c \xrightarrow{e} d}{D, \delta \vdash_D c; c' \xrightarrow{e} d; c'}$$

Figure 4: CSL reduction semantics.

contracts, exactly one of the two subcontracts must consume the event. We thus have two alternatives: either the left contract consumes it, or the right contract does. For sequential contracts, we can only continue with the second contract once the first contract is nullable. However, the first contract being nullable does not necessarily mean that the event is accepted by the second contract; the first contract may be an alternative contract consiting of Success $+ c'$, with c' containing a 'transmit' specification. The result is then an alternative between the first contract consuming the event, or the first contract being nulled and the second contract consuming the event.

5.1 Limitations of CSL

CSL's current design has certain limitations[7]:

1. Conditionals can only be tied to event specifications, not contracts; trying to apply a predicate to a contract instead would be syntactically invalid.
2. For the same reason, an instantiated contract cannot branch immediately based on its input parameters; it needs to receive an event first.
3. Contract templates cannot return any values, meaning that all event specifications that depend on bound values from previous specifications must be in the same contract as those.

It is clear that the distinction between event specifications and contracts is restrictive. While the limited expressiveness of a DSL is often one of its advantages, in this case, it could be considered too restrictive, as many financial contracts may be either disproportionally difficult or impossible to express in CSL. For example, one might consider describing a *barrier* *option*[2] as a contract where the asset observation and the payment are done in two separate contract templates. However, since the asset-observation-contract cannot "return" any values to the payment-contract, a different structure must be used altogether.

The current version of CSL as presented by Deon Digital increases expressiveness by allowing contract templates to accept other contract identifiers as arguments[3], essentially allowing callbacks. This makes it possible to simulate returning values by using multi-layered currying. Nonetheless, contracts which make heavy use of such currying may be hard to read, much less for a domain-expert with little to no coding experience.

6 CSL2

The mathematical syntax and semantics presented in this section are based on notes from the Deon Digital Core Technology team [7].

The limitations of CSL, as described in the previous section, can be narrowed down to a single design decision, which is that event specifications are fundamentally different from contracts. The Core Technology team presents the following solution:

Redefine event specifications as opaque contracts.

This means removing any notion of predicates or sequential composition from event specifications ('transmit'), instead, the only thing done by event specifications is the matching of real-world events and the binding of values to identifiers. Essentially, 'transmit$(A_1, A_2, R, T \mid P).c$' is reduced to just 'transmit'.

Removing predicates would make it impossible to accept any event conditionally, but it would also make it impossible to bind any type of data from the event at all. This necessitates another design change: Contracts should have an *input state* and an *output state*, allowing data to be passed from one subcontract to the next.

Based on these design choices, we end up with the language syntax presented in figure 5.

$$\begin{aligned} C ::= {} & 1(r) \\ & \mid \bot \\ & \mid X \\ & \mid C_1; C_2 \\ & \mid C_1 || C_2 \\ & \mid C_1 + C_2 \end{aligned}$$

Figure 5: CSL2 mathematical syntax.

Predicates are still expressed through an internal expression language, but are now "their own" construct, and are instead represented by a *relation* $r \subseteq S \times T$, which relates an input state to an output state. Using relations for the definition instead of functions corresponds

[2]See section 8.1.

[3]https://docs.deondigital.com/latest/src/guidechapters/contractreference.html

better to the partial nature of contract evaluations: they may fail, or they may succeed in multiple ways. To further support the notion of a failure, the construct $\perp$ represents the always failing contract, which accepts no input states and thus gives no output states (technically, it can be expressed as a relation with the same description, but we add it as a separate construct to simplify some rules).

The *contract identifier* X replaces the event specification from CSL. It is bound to a collection of events E, where each $e \in E$ uniquely determines an input state s and an output state t. If it receives such an event e, and the contract is in state s, then the contract returns the state t.

We introduce the $\mathcal{E}$ denotation for defining the, possibly nonexistent, output state as a result of matching event e with identifier X and input state s:

$$\mathcal{E}[\![X]\!]^s(e) = \begin{cases} \lfloor t \rfloor & \text{if } e \text{ is bound to } X \text{ with input state } s \\ \perp & \text{otherwise} \end{cases}$$

where $\lfloor t \rfloor$ is part of the *lifted set* $T_\perp$ of output states, which is defined as $T_\perp = \{\lfloor t \rfloor \mid t \in T\} \cup \perp$. This makes event matching comparable to implementing a partial function using the `Maybe` monad. Also, at the risk of mixing notation, note that the $\perp \in T_\perp$ is distinct from the contract construct $\perp$.

The contract combinators are similar to their CSL counterparts, although with some added behaviour for input and output states.

- $C_1; C_2$ defines sequential contract composition. It consumes the set of events $E = E_1 \cup E_2$, and transitions from the input state s to the output state u, if there exists a state t such that C_1 consumes E_1 in state s returning t, and C_2 consumes E_2 in state t returning u.

- $C_1 || C_2$ defines parallel contract composition. It consumes the set of events $E = E_1 \cup E_2$, and transitions from the input state (s_1, s_2) to the output state (t_1, t_2), if C_1 consumes E_1 in state s_1 returning t_1, and C_2 consumes E_2 in state s_2 returning t_2.

- $C_1 + C_2$ defines alternative contract composition. If C_1 consumes E in state s and returns t_1, $C_1 + C_2$ does as well. The same applies for C_2: if it consumes E in state s and returns t_2, $C_1 + C_2$ does as well. In this sense, the result of applying events to an alternative contract is *nondeterministic*.

The explanations above give a notion of consuming events, which was also the case for how the CSL operational semantics were described. We give a similar set of operational semantics for CSL2 in figure 6. The judgement $\Gamma \vdash e : s \xrightarrow{C} t$ means that the contract C in the initial state s consumes the set of events Γ and results in output state t. e is a *proof term*, which shows how the events in Γ are combined to give proof of C being completed successfully.

Note that $\perp$ has no associated rule, as an output state can never be obtained from a failing contract. Also, the '1' event indicates the unit type, i.e. the "empty" event sequence.

The big-step semantics are good for reasoning about contract output states when the event collection is known in advance, but it does not represent residual contracts, which have not finished evaluation and still need to receive events. This would require a semantics that, instead of defining transitions from *state to state* based on a contract and an event sequence, defines transitions from *contract to contract* based on an input state and a single event. The semantics in question would be considered *small-step semantics*, which are defined in figure

$$\text{E-ID}\frac{}{x : s \xrightarrow{X} t \vdash x : s \xrightarrow{X} t}$$

$$\text{E-REL}\frac{}{\vdash 1 : s \xrightarrow{1(r)} t} \qquad \text{if } (s,t) \in r$$

$$\text{E-ALT1}\frac{\Gamma \vdash e_1 : s_1 \xrightarrow{C_1} t_1}{\Gamma \vdash e_1 : s_1 \xrightarrow{C_1+C_2} t_1} \qquad \text{E-ALT2}\frac{\Gamma \vdash e_2 : s_2 \xrightarrow{C_2} t_2}{\Gamma \vdash e_2 : s_2 \xrightarrow{C_1+C_2} t_2}$$

$$\text{E-SEQ}\frac{\Gamma \vdash e_1 : s \xrightarrow{C_1} t \qquad \Delta \vdash e_2 : t \xrightarrow{C_2} u}{\Gamma, \Delta \vdash (e_1; e_2) : s \xrightarrow{C_1;C_2} u}$$

$$\text{E-PAR}\frac{\Gamma \vdash e_1 : s_1 \xrightarrow{C_1} t_1 \qquad \Delta \vdash e_2 : s_2 \xrightarrow{C_2} t_2}{\Gamma, \Delta \vdash (e_1||e_2) : (s_1, s_2) \xrightarrow{C_1||C_2} (t_1, t_2)}$$

Figure 6: CSL2 big-step reduction semantics.

7. The judgement $\vdash (c, s) \xrightarrow{e} c'$ states that the contract c consumes event e in input state s and residuates to c'.

The failing contract $\perp$ residuates to itself, regardless of the received event or starting state. The relation $1(r)$ also always fails on an event, since it cannot consume events, nor can it "discard" events that do not match. The residual contract of the identifier X depends on whether or not it can consume the applied event. If it matches the event e in state s, it residuates to $1(unit_t)$, where $unit_t$ is a relation that returns the state t on any input. Otherwise, it residuates to $\perp$. For sequential composition ; there are two options: either the first subcontract should receive the event, or the first subcontract has finished receiving events, meaning that the second subcontract should receive it instead. We cannot determine in advance which one is the "right" branch, so the only option is to evaluate both options and make the residual contract an alternative between the two. We define the application of *no* events to contract C with input state s as $\vdash 1 : s \xrightarrow{C} t$. Note that, since the big-step semantics are nondeterministic, this makes the small-step semantics nondeterministic as well. For the alternative composition $+$, we do not know which of the contracts should consume the event, so we apply it to both. The result is then the alternative of the two residual subcontracts. The parallel composition $||$ also has two subcontracts, but unlike the alternative where both contracts may consume the event, exactly one of the subcontracts in the parallel composition must do it. Since we still do not know which contract will consume the event, the residual contract is an alternative between two parallel contracts: one where the left-hand contract consumes the event, and one where the right-hand contract consumes it.

As is, the semantics have no notion of types. This is not an issue per se, since any incompatible input state will simply fail to residuate, which is what we would expect anyways. Nonetheless, we introduce a typing semantics for the contract type. We define a contract with an output state of type b as the type CON b.

$$\text{SE-FAIL}\,\frac{}{\vdash (\bot, s) \xrightarrow{e} \bot}$$

$$\text{SE-ID}\,\frac{}{\vdash (1(r), s) \xrightarrow{e} \bot}$$

$$\text{SE-RELS}\,\frac{\mathcal{E}[\![X]\!]^s(e) = \lfloor t \rfloor}{\vdash (X, s) \xrightarrow{e} 1(unit_t)} \qquad \text{SE-RELF}\,\frac{\mathcal{E}[\![X]\!]^s(e) = \bot}{\vdash (X, s) \xrightarrow{e} \bot}$$

$$\text{SE-SEQ}\,\frac{\vdash (C_1, s) \xrightarrow{e} C_1' \qquad \vdash 1 : s \xrightarrow{C_1} t \qquad \vdash (C_2, t) \xrightarrow{e} C_2'}{\vdash (C_1; C_2, s) \xrightarrow{e} (C_1'; C_2) + C_2'}$$

$$\text{SE-ALT}\,\frac{\vdash (C_1, s) \xrightarrow{e} C_1' \qquad \vdash (C_2, s) \xrightarrow{e} C_2'}{\vdash C_1 + C_2 \xrightarrow{e} C_1' + C_2'}$$

$$\text{SE-PAR}\,\frac{\vdash (C_1, s_1) \xrightarrow{e} C_1' \qquad \vdash (C_2, s_2) \xrightarrow{e} C_2'}{\vdash (C_1 || C_2, (s_1, s_2)) \xrightarrow{e} (C_1' || C_2) + (C_1 || C_2')}$$

Figure 7: CSL2 small-step semantics.

$$\text{T-FAIL}\,\frac{}{\vdash \bot : \tau \to \text{CON}\ \tau'} \qquad \text{where } \tau \text{ and } \tau' \text{ are free type variables}$$

$$\text{T-ID}\,\frac{}{X : a \to \text{CON}\ b \vdash X : a \to \text{CON}\ b}$$

$$\text{T-REL}\,\frac{}{\vdash 1(r) : a \to \text{CON}\ b} \qquad \text{if } (s, t) \in r \Rightarrow s : a \wedge t : b$$

$$\text{T-ALT}\,\frac{\vdash C_1 : a \to \text{CON}\ b \qquad \vdash C_2 : a \to \text{CON}\ b}{\vdash C_1 + C_2 : a \to \text{CON}\ b}$$

$$\text{T-SEQ}\,\frac{\vdash C_1 : a \to \text{CON}\ b \qquad \vdash C_2 : b \to \text{CON}\ c}{\vdash C_1; C_2 : a \to \text{CON}\ c}$$

$$\text{T-PAR}\,\frac{\vdash C_1 : a \to \text{CON}\ b \qquad \vdash C_2 : a' \to \text{CON}\ b'}{\vdash C_1 || C_2 : (a, b) \to \text{CON}\ (a', b')}$$

Figure 8: CSL2 typing rules.

We can then define the typing rules in figure 8, where $\vdash c : a \to \text{CON}\ b$ means that contract c

takes an input state of type a and transitions to a contract with an output state b.

We also introduce the shorthand notation

$$() \rightarrow \textsc{Con}\ b \equiv \textsc{Con}\ b$$

The main reason for defining the type constructor CON is that it allows us to define the contract type as a *monad*, which will make it easier to implement the semantics using Haskell's built-in monad support.

6.1 Contracts as monads

Proposition: $(\textsc{Con}, 1(id), \mathbin{;})$, where $id = \{(s, s) \mid s \in S\}$, is a monad.

Proof: In order to show that the contract type is a monad, we must show that it obeys the three monad laws[4]:

$$\begin{array}{rlcl} \textbf{Left identity}: & (\texttt{return}\ a)\ \texttt{>>=}\ f & \Leftrightarrow & f\ a \\ \textbf{Right identity}: & m\ \texttt{>>=}\ \texttt{return} & \Leftrightarrow & m \\ \textbf{Associativity}: & (m\ \texttt{>>=}\ g)\ \texttt{>>=}\ h & \Leftrightarrow & m\ \texttt{>>=}\ (\lambda x.\, g(x)\ \texttt{>>=}\ h) \end{array}$$

where
  $\texttt{return} : a \rightarrow \textsc{Con}\ a$
is the unit-operator and
  $\texttt{>>=} : \textsc{Con}\ a \rightarrow (a \rightarrow \textsc{Con}\ b) \rightarrow \textsc{Con}\ b$
is the bind-operator.

Left identity:
We rewrite the rule in the style of the big-step operational semantics:

$$\begin{aligned} (\texttt{return}\ a)\ \texttt{>>=}\ f &\Leftrightarrow \Gamma \vdash e : a \xrightarrow{1(id);f} t \\ f\ a &\Leftrightarrow \Gamma \vdash e : a \xrightarrow{f} t \end{aligned}$$

for some event sequence e and environment Γ.

Now, we must show that

$$\Gamma \vdash e : a \xrightarrow{1(id);f} t \quad \Leftrightarrow \quad \Gamma \vdash e : a \xrightarrow{f} t$$

We construct the derivation tree for the left-hand side of the equivalence, using the operational semantics. There is only one rule that could be used for sequential composition, E-SEQ, as well as one rule for relations, E-REL:

$$\dfrac{\dfrac{}{\vdash 1 : a \xrightarrow{1(id)} a} \qquad \dfrac{\mathcal{E}_2}{\Gamma \vdash e_2 : a \xrightarrow{f} t}}{\Gamma \vdash e : a \xrightarrow{1(id);f} t}$$

[4] https://wiki.haskell.org/Monad_laws

where $e = 1; e_2$.
From the sole subderivation $\mathcal{E}_2$, we can construct the right-hand side derivation:

$$\begin{array}{c}\mathcal{E}_2\\ \Gamma \vdash e_2 : a \xrightarrow{f} t'\end{array}$$

since $e_2 = (1; e_2) = e$, as 1 denotes no event received.
It should be immediately clear that the equivalence holds in the right direction as well, since the left-hand side can only be constructed with rules E-SEQ and E-REL.

Right identity:
Rewriting in the style of the operational semantics:

$$\begin{aligned} m \texttt{ >>= return} &\Leftrightarrow \Gamma \vdash e : () \xrightarrow{m;1(id)} t \\ m &\Leftrightarrow \Gamma \vdash e : () \xrightarrow{m} t \end{aligned}$$

We must show that

$$\Gamma \vdash e : () \xrightarrow{m;1(id)} t \quad \Leftrightarrow \quad \Gamma \vdash e : () \xrightarrow{m} t$$

Again, the derivation must have used the E-SEQ:

$$\dfrac{\begin{array}{c}\mathcal{E}_1\\ \Gamma \vdash e_1 : () \xrightarrow{m} t\end{array} \qquad \dfrac{}{\vdash 1 : t \xrightarrow{1(id)} t}}{\Gamma \vdash e : () \xrightarrow{m;1(id)} t}$$

where $e = e_1; 1$.
We construct the right-hand side from the $\mathcal{E}_1$ subderivation:

$$\begin{array}{c}\mathcal{E}_1\\ \Gamma \vdash e_1 : () \xrightarrow{m} t\end{array}$$

where $e_1 = (e_1; 1) = e$. Again, the right-to-left equivalence should be clear for the same reason: that only rules E-REL and E-SEQ could have been used to construct the derivation in the same manner.

Associativity:
Rewriting using the operational semantics gives us

$$\begin{aligned} (m \texttt{ >>= } g) \texttt{ >>= } h &\Leftrightarrow \Gamma \vdash e : () \xrightarrow{(m;g);h} c \\ m \texttt{ >>= } (\lambda x.\, g(x) \texttt{ >>= } h) &\Leftrightarrow \Gamma \vdash e : () \xrightarrow{m;(g;h)} c \end{aligned}$$

We must then show that

$$\Gamma \vdash e : () \xrightarrow{(m;g);h} c \quad \Leftrightarrow \quad \Gamma \vdash e : () \xrightarrow{m;(g;h)} c$$

Since the bind operation in Haskell is left-associative, we consider `;` to be as well. This gives the only possible derivation of

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{E}_1\\ \Gamma \vdash e_1 : () \xrightarrow{m} a\end{array} \qquad \begin{array}{c}\mathcal{E}_2\\ \Delta \vdash e_2 : a \xrightarrow{g} b\end{array}}{\Gamma, \Delta \vdash e_1; e_2 : () \xrightarrow{m;g} b} \qquad \begin{array}{c}\mathcal{E}_3\\ \Phi \vdash e_3 : b \xrightarrow{h} c\end{array}}{\Gamma, \Delta, \Phi \vdash e : () \xrightarrow{(m;g);h} c}$$

where $e = e_1; e_2; e_3$.
We construct the right-hand side using the $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}_3$ subderivations:

$$\dfrac{\dfrac{\mathcal{E}_1}{\Gamma \vdash e_1 : () \xrightarrow{m} a} \quad \dfrac{\dfrac{\mathcal{E}_2}{\Delta \vdash e_2 : a \xrightarrow{g} b} \quad \dfrac{\mathcal{E}_3}{\Phi \vdash e_3 : b \xrightarrow{h} c}}{\Gamma, \Delta \vdash e_2; e_3 : a \xrightarrow{g;h} c}}{\Gamma, \Delta, \Phi \vdash e_1; e_2; e_3 : () \xrightarrow{m;(g;h)} c}$$

Again, since E-SEQ is the only valid rule for constructing the derivations, it should be clear that the right-to-left equivalence also holds.

$\square$

With the syntax and operational semantics in place, we are ready to present a concrete implementation for CSL2. Contracts are defined using generalized algebraic data types, as seen in the code below.

```
data Contract b where
   Fail     :: Contract b
   NoEvent  :: b -> Contract b
   Event    :: Event b -> Contract b
   Then     :: Contract a -> (a -> Contract b) -> Contract b
   Par      :: Contract a -> Contract b -> Contract (a, b)
   Alt      :: Contract b -> Contract b -> Contract b
```

It may seem odd that most of the contract constructors (in fact, all of them except `Then`) have no description of an input state; they all generate contracts of the type $() \to \textsc{Con}\ b$. This is however very much intended, as it simplifies contract composition and evaluation while still adhering to the type system specified in figure 8.

Something we overlooked when defining the operational big-step semantics was how the final output state is determined. The small-step semantics residuate contracts to contracts, but we need a semantics for obtaining output states from contracts; specifically in the case where there are no remaining events, but only an input state. Furthermore, since the big-step semantics are nondeterministic, and the result state is based on those semantics, we need a *collecting semantics*, which can collect all possible output states, shown in figure 9. Here, $\mathcal{N}[\![C]\!](s)$ returns the multiset of output states from the contract c given input state s. The nondeterminism is thus represented as multiset of possible output states.

$$\begin{aligned}
\mathcal{N}[\![1(r)]\!](s) &= \{t \mid (s,t) \in r\} \\
\mathcal{N}[\![X]\!](s) &= \{\} \\
\mathcal{N}[\![C_1; C_2]\!](s) &= \{u \mid t \in \mathcal{N}[\![C_1]\!](s), u \in \mathcal{N}[\![C_2]\!](t)\} \\
\mathcal{N}[\![C_1 + C_2]\!](s) &= \mathcal{N}[\![C_1]\!](s) \cup \mathcal{N}[\![C_2]\!](s) \\
\mathcal{N}[\![C_1 || C_2]\!](s_1, s_2) &= \{(t_1, t_2) \mid t_1 \in \mathcal{N}[\![C_1]\!](s_1), t_2 \in \mathcal{N}[\![C_2]\!](s_2)\}
\end{aligned}$$

Figure 9: CSL2 output state collecting semantics

The Haskell implementation uses the list monad for representing the state collection. Recall that in CSL, a residual contract that can succeed as-is (i.e. without receiving more events) has the *nullable* property. Since the collecting semantics essentially collects states that can be obtained without further event application, the implementation function is named `nullify`:

```haskell
nullify :: Contract b -> [b]
nullify contract =
  case contract of
    Fail ->
      mzero
    NoEvent b ->
      return b
    Event _ ->
      mzero
    Then c c' -> do
      b <- nullify c
      nullify (c' b)
    Par c c' -> do
      b <- nullify c
      b' <- nullify c'
      return (b, b')
    Alt c c' ->
      (nullify c) ++ (nullify c')
```

Nullifying the `Fail` contract gives the empty list (`mzero`), since it has no output states. `NoEvent b` gives the singleton list containing the output state `b`. `Event b` also gives an empty list, since it indicates that the contract is still incomplete and needs to consume further events. `Then` finds all output states of the first subcontract, then returns the output states of all the former output states applied to the second subcontract. `Par` simply finds the output states of both subcontracts, then returns the tuple of both. Note that this gives a multiset (list) of tuples, which is the product of the multisets (lists) from the left and right subcontract. `Alt` simply finds the multisets derived from both subcontracts, then unions them.

We move on to the implementation of the small-step semantics, which describe the residuation of a contract after the application of a single event. The function `apply` implements the semantics.

```haskell
apply :: EventInfo -> Contract b -> Contract b
apply eventinfo contract =
  case contract of
    Fail ->
      failure
    Event event ->
      case matchevent event eventinfo of
        Just info -> return info
        Nothing -> failure
    NoEvent _ ->
      failure
    Then c c' -> do
      (apply eventinfo c) `Then` c'
      <|>
      (foldr (\e a -> apply eventinfo (c' e) <|> a) (failure) (nullify c))
    Par c c' -> do
      (apply eventinfo c) `Par` c'
      <|>
      c `Par` (apply eventinfo c')
    Alt c c' -> do
      apply eventinfo c
      <|>
      apply eventinfo c'
```

Fail and NoEvent are identical to their small-step semantic counterpart. Event matches the event specification with the given event using a separate matchevent function. Just like the $\mathcal{E}$ denotational function, this function also returns a Maybe monad.

For the contract combinators, Par and Alt clearly match the small-step semantics. Then is a bit more involved; the left side of the alternative is similar to the semantics, but what about the right side? First, note the result type of the function apply: even though the small-step semantics are non-deterministic, apply does not return a collection of possible contracts, but a single contract. After obtaining the output states from the first contract by nullify c, we apply each output state (and the received event) to the second contract individually, and then make the second contract into an alternative of all the residual contracts. This essentially converts the nondeterministic semantics into a deterministic implementation.

6.2 CSL2 embedded syntax

We now go over some of the possible ways of constructing CSL2 contracts using the embedded DSL. Suppose that we want to define a contract transferByDate, which accepts a single transfer event, and then checks that the transfer was of the correct amount, and occurred before a given date. Using only the existing GADT constructors, the contract could be defined as:

```
transferByDate (amount, deadline) =
  Then (Event Transfer)
       (\t -> if date t <= deadline && resource t == amount
              then t
              else Failure)
```

We can increase readability by using infix notation:

```
transferByDate (amount, deadline) =
  Event Transfer
  `Then`
  \t -> if date t <= deadline && resource t == amount
        then NoEvent t
        else Failure
```

To improve the read- and writeability further, we define a small set of additional contract constructors from the existing ones:

```
suchthat :: Contract a -> (a -> Bool) -> Contract a
suchthat c p = c `Then` \x -> if p x then return x else failure

andthen :: Contract a -> Contract b -> Contract (a, b)
andthen c1 c2 = c1 `Then` \x -> (NoEvent x) `Par` c2

orelse :: Contract a -> Contract a -> Contract a
orelse = Alt

andalso :: Contract a -> Contract b -> Contract (a,b)
andalso = Par
```

Now, `transferByDate` can be rewritten to:

```
transferByDate (amount, deadline) =
  Event Transfer
  `suchthat`
  \t -> date t <= deadline && resource t == amount
```

Since contracts are represented as monads, they can be defined using do-notation, so as an alternative, the contract can be rewritten as:

```
transferByDate (amount, deadline) = do
  t <- Event Transfer
  if date t <= deadline && resource t == amount
  then return t
  else failure
```

While using the arrow notation to ”extract” values from events is only marginally useful, the strength of the do-notation comes from the fact that it can be used to obtain return values from much more elaborate contract instantiations. For example, consider a *knock-out option* in which the option only trades for full value if a *barrier price* has not been breached over a set of observations. If we have a subcontract `collectObservations` which listens for observation events, the contract for the knock-out option may be written as:

```
knockOutOption (barrier, value) = do
  observations <- collectObservations
  if observations `hit` barrier
  then failure
  else Event Transfer
       `suchthat`
       \t -> resource t == value
```

It is not immediately apparent, but due to how the `Contract` monad is defined, using an arrow in `do`-notation is the same as inserting a `Then` statement.

7 Internal expression language

7.1 Time modelling

This section is based on the novel mathematical model of calendar time developed by Henglein [8], as well as some of the functions presented in chapter 2 of Reingold and Dershowitz [9].

We previously mentioned that CSL is parametric in its expression language. This means, by extension, that the internal representation of agents, resources and time depends not on the contract language, but the expression language. Time in particular is interesting, as it is not immediately given how time should be modelled. One may use ”objective” timestamps such as UNIX Epoch time, but there are two reasons against it. First, the contracts that we focus on in this paper, which are mainly financial contracts, rarely need time that is ”finer grained” than dates. Second, the CSL2 contracts are meant to be human readable, and while UNIX timestamps may be readable (at least they are visually *comparable*), it is not clear what the *intent* of the timestamp is, unlike a date. For this reason, we choose to implement a more meaningful model for representing calendar time.

What options exist for modelling calendar time mathematically? There is surprisingly little work done in this field. The closest well-known concept would be vector clocks, which is used for ordering events in distributed systems, but has little use for representing actual dates. The aforementioned paper by Henglein presents a novel way of modelling calendar dates, which is the choice of date representation for this project. We briefly present the model before then showing the concrete implementation.

The model is based on three primitive, incomparable *durations*: Y (year), M (month) and D (day). The durations can be added together with the binary operator $+$, which has the neutral element 0. With the further specification that the durations have inverses and commute with each other, we can define $durations$ as a *free abelian group*, generated by $\{Y, M, D\}$. All durations have the normal form:

$$y \cdot Y + m \cdot M + d \cdot D \qquad y, m, d \in \mathbb{Z}$$

A $duration$ can be interpreted as a function from dates to dates, with $+$ denoting functional composition and 0 the identity.

With $durations$ defined, we can now define a concrete date, or *Free date*, which is represented as a duration applied to some fictious *Date Zero*. Like the reference article, we will from now use the notation y-m-d to denote the duration $y \cdot Y + m \cdot M + d \cdot D$.

There are no ”invalid” free dates, dates such as 2021-02-29 and 2020-13-01 are also free dates, though they are not part of the subset defined by the Julian or Gregorian calendar. Free dates that are valid in a calendar are said to be *standard dates* of that calendar; conversely, dates not in said calendar are *nonstandard dates*.

Calendars are defined as a free abelian group generated by a single element N. The effect of this is that all elements (dates) in the calendar map to a number $n \in \mathbb{Z}$ and vice versa. We can thus define a $dayNumber$ function for each calendar:

$$dayNumber : FreeDate \to \mathbb{Z}$$

$dayNumber$ is defined for the Gregorian (and Julian) calendar to map standard dates monotonically to $\mathbb{Z}$, based on their lexicographic ordering. The date 0000-03-01 maps to 0 for practical reasons, as it makes conversions easier.

7.1.1 Modelling the Gregorian Calendar

To model the Gregorian Calendar, we first need to determine its set of standard dates. They are defined as

$$GregorianDate = \{\text{y-m-d} \mid valid(\text{y-m-d})\}$$

where

$$\begin{aligned}
valid(\text{y-m-d}) &= y, m, d \in \mathbb{Z} \wedge 0 < m \leq 12 \wedge 0 < d \leq d_{m,y} \\
d_{m,y} &= 31 \text{ if } m \in \{1, 3, 5, 7, 8, 10, 12\} \\
d_{m,y} &= 30 \text{ if } m \in \{4, 6, 9, 11\} \\
d_{2,y} &= 29 \text{ if } y \equiv 0 \text{ mod } 400 \vee (y \equiv 0 \text{ mod } 4 \wedge y \not\equiv 0 \text{ mod } 100) \\
d_{2,y} &= 28 \text{ if } y \not\equiv 0 \text{ mod } 4 \vee (y \equiv 0 \text{ mod } 100 \wedge y \not\equiv 0 \text{ mod } 400)
\end{aligned}$$

With $valid$ as a base for the calendar, we can define the $dayNumber$ function in a simple manner by first defining a function $next : GregorianDate \rightarrow GregorianDate$, which gives the successor of any Gregorian date:

$$\begin{aligned}
next(\text{y-m-d}) &= \text{y-m-(d+1) if } d < d_{m,y} \\
next(\text{y-m-d}) &= \text{y-(m+1)-01 if } d = d_{m,y} \wedge m < 12 \\
next(\text{y-m-d}) &= \text{(y+1)-01-01 if } d = d_{m,y} \wedge m = 12
\end{aligned}$$

We define $next^k$ as the application of $next$ k times to the same date, where $k \in \mathbb{Z}$ We can then define $dayNumber : GregorianDate \rightarrow \mathbb{Z}$ as:

$$dayNumber(d) = k \Leftrightarrow next^k(Day0) = d$$

where $Day0 =$ 0000-03-01, since it ensures that *epochs* of 4-, 100- and 400-years have a leap day at the very end, which simplifies calculations.

Using $next^k$ for computing the day number gives a $dayNumber(d)$ an upper bound of $O(k)$, but the number can be computed in $O(1)$ time by using a different algorithm:

1. Given the date y-m-d, find its duration $y'Y + m'M + d'D$ from $Day0$.
2. Calculate the number of leap days between $Day0$ and y'-03-01:

   $$y' \div 4 - y' \div 100 + y' \div 400$$

   The total number of days between $Day0$ and y'-03-01 is then

   $$365 \cdot y' + y' \div 4 - y' \div 100 + y' \div 400$$

   which is our initial number.
3. Find the offset from y'-03-01 to y-m-01, and add it to the number.
4. Finally, add $d' = d - 1$ to the number to get the total number of dates from $Day0$ to y-m-d.

This is expressed in the code as shown below.

```haskell
dayNumberGregorian :: FreeDate -> Int
dayNumberGregorian date =
  year_days + month_days + d
  where
    Date y m d = day0Difference date
    year_days = 365*y + (y `div` 4) - (y `div` 100) + (y `div` 400)
    month_days = month_offset !! m
```

We also show how to compute the inverse $dayNumber^{-1} : \mathbb{Z} \to GregorianDate$, which is slightly more involved:

1. We first compute the number of epochs of 400-, 100- and 4-year intervals from $Day0$ to t days forward:

$$\begin{aligned} epochs_{400} &= t \div 146097 \\ rem_{400} &= t \text{ mod } 146097 \\ epochs_{100} &= epochs_{400} \div 36524 \\ rem_{100} &= rem_{400} \text{ mod } 36524 \\ epochs_{4} &= epochs_{100} \div 1461 \\ rem_{4} &= epochs_{100} \text{ mod } 1461 \end{aligned}$$

   The resulting date is then rem_4 days after the date

$$(400 \cdot epochs_{400} + 100 \cdot epochs_{100} + 4 \cdot epochs_4)\text{-m-d}$$

2. If $rem_4 = 1460$, then the date is the leap day, which is exactly

$$(400 \cdot epochs_{400} + 100 \cdot epochs_{100} + 4 \cdot epochs_4 + 4)\text{-02-29}$$

3. Else, if $rem_{400} = 146096$, the day is also a leap day, which is

$$(400 \cdot epochs_{400} + 100 \cdot epochs_{100})\text{-02-29}$$

4. Otherwise, we know that the amount of years y is

$$y = 400 \cdot epochs_{400} + 100 \cdot epochs_{100} + 4 \cdot epochs_4 + rem_4 \div 365$$

   Now, we find the amount of days t' between y-m-d (the final date) and y-03-01 which is $t' = rem_4$ mod 365, and convert it to a number of months m' and days d' from March 1st.

5. Finally, we have $m = m' + 3$ and $d = d' + 1$ to offset the March 1st $Day0$, which yields the date of y-m-d.

In financial contracts, the interest rate for a loan is commonly given as a yearly rate, but the payments themselves are often made more frequently, such as once a month. Computing the time between two payment dates is then necessary in order to pay the correct fraction of the rate, known as the *coupon factor*. It would appear that the coupon factor could be calculated by simply counting the number of days and then dividing by either 365 or 366, but this is not a given. There are many *day count conventions* that are used when computing interest rates for payments. Each of them generally use the following formula to determine the coupon factor f for the duration $y \cdot Y + m \cdot M + d \cdot D$:

$$f = 1 \cdot y + \frac{1}{12} \cdot m + \frac{1}{d^y} \cdot d$$

where d^y is the number of days in a year, according to the convention. Furthermore, day count conventions use a different number of days per month between them; it may be the actual value, or it may be a constant such as 30. We give a few examples of day count conventions:

- **30/360**:
  This convention treats all months as 30 days and all years as 360 days ($d^y = 360$).
- **30/360 German**:
  The German convention may also adjust the start date (y1-m1-d1) and end date (y2-m2-d2) if either is the last day of the month, specifically
  - If y1-m1-d1 is the last day of the month, set it to y1-m1-30.
  - If y2-m2-d2 is the last day of the month, set it to y2-m2-30.
- **30/360 US**:
  The US convention may also adjust the start and end date, but under different conditions:
  - If y1-m1-d1 is the last day of the month, set it to y1-m1-30 (same as 30/360 German).
  - If y2-m2-d2 is the last day of the month *and* the start date is y1-m1-30, set it to y2-m2-30.

  For example, under 30/360 German, y1-04-29 and y2-05-31 would be changed to y1-04-29 and y2-05-30, while they would remain the same under 30/360 US.

7.2 Resource modelling

So far, we have shown how time can be implemented from the basis of an algebraic model. But what about other types of resources? Namely, resources such as money that cannot be produced or consumed, but can only be transferred between owners in a system. We use an algebraic resource model introduced by Torres[10] which presents a set of invariants to ensure that resource ownership within the system remains consistent after event application.

We first provide definitions for essential parts of the model. We omit the more technical details of linear algebra and present the concepts in an informal manner:

Resources: *Let X be the set of user-defined resource types. Then the set of resources R is the set of finite maps from resource types to $\mathbb{R}$.*

Ownership states: *Let A be the set of agents in the system. Then the set of ownership states S is the set of finite maps from agents to resources.*

Transfers: *Let sum : $S \rightarrow_1 R$ be a homomorphism that sums each resource from all agents, resulting in a single resource map that maps each resource to its total amount in that ownership state. Then, a transfer t is an ownership state where each resource in sum(t) maps to 0.* For reference, the following ownership state is a valid transfer:

$$\{\text{Alice} \mapsto 42 \cdot \text{DKK}, \text{Bob} \mapsto -42 \cdot \text{DKK}\}$$

which describes a payment of 42 DKK from Bob to Alice.

A resource system is modelled as a single ownership state, which is then changed by applying (adding) transfers to it. By the definition of a transfer, the total amount of resources in such an ownership state will remain constant, as resources cannot be created ”out of the blue”, which mirrors real world financial trades.

Even if neither the ownership state or the transfer are inconsistent (e.g. someone owns a negative or non-integer amount of currency), the ownership state may be inconsistent after application; for example, a user with a 5 DKK balance transfers 100 DKK to another user. States post-transfer may be verified using *ownership state predicates*:

Ownership state predicate: *A boolean function $p : S \rightarrow \{0, 1\}$ over the set of ownership states S.*

Although not decisively implemented in this project, the intention is to have the complete CSL2 solution combine a *contract manager*, which receives and applies events to contracts, with a *resource manager*, which maintains the ownership state for the relevant actors.

```haskell
contractManager :: IORef OwnershipState -> EventInfo -> {- ... -} =
contractManager refRMState eventInfo {- ... -} =
  {- ... -}
  case eventInfo of
    TransferInfo (fromAgent, toAgent, (resource, amount), date) -> do
      resourceManager (fromAgent, toAgent, resource, amount) refRMState
    _ ->
      return ()
  {- ... -}

resourceManager (fromAgent, toAgent, resource, amount) refRMState = do
  rmState <- readIORef refRMState
  let t = makeOwnershipState' [
    (fromAgent, [(resource, -amount)]) ,(toAgent, [(resource, amount)])
  ]
  case commitTransfer (\_ -> True) t rmState of
    Just rmState' ->
      writeIORef refRMState rmState'
    Nothing ->
      putStrLn "Failed to transfer"
```

The code example shows how the contract manager and resource manager may be combined. An `IORef` monad is used to make the ownership state `refRMState` mutable in a purely functional environment. The `makeOwnershipState'` function is part of the resource manager libary developed by Torres, which uses a list of tupled agents and resources to generate an opaque `OwnershipState` type. The `commitTransfer` is also part of the library; it takes an ownership state predicate, a transfer, and an ownership state, then returns either `Nothing` if the predicate fails or `Just` the updated state if the predicate succeeds.

7.3 Exact integer arithmetic

On the topic of modelling payments, specifically amounts of a given currency, there is the issue of how to perform arithmetic on integer amounts with non-integer factors. To the author's knowledge, there is no currency which does not use integer values for their respective smallest monetary unit; for example, 0.6 cents in USD is not a valid amount. Yet, such an amount might be the result of multiplying with a non-integer interest rate.

In practice, the problem is handled by rounding the result. However, this leaves plenty of room for ambiguity when considering rounding conventions. Futhermore, even what should be straight forward calculations may be affected by the data types used: consider the common example of $0.1 + 0.2$ resulting in a value slightly larger than 0.3 when using floating points.

We present a solution by using the `Rational` type provided in Haskell, which represents rational numbers as the ratio of two integers, giving an exact representation[5]. This allows us to perform computations with non-integer numbers without loss of precision. However, at some point we still need to round from non-integer to integer, specifically when verifying transfer events, where only integer amounts can be transferred. For all of our contracts, we consistently use the following function for rounding:

```haskell
roundMoney :: Rational -> Integer
roundMoney = round
```

Using this rounding method globally is sufficent for our proof-of-concept contract implementations, but since the preferred rounding method may differ from party to party, the rounding method should be parametrized as part of the contract template, rather than be a standardized function.

8 Representing contracts

In this section, we present a number of financial contracts that can be modelled with CSL2, and are based on existing Deon Digital CSL contracts. We choose contracts that are difficult to represent in CSL due to the lack of return values, and thus enjoy the "statefulness" of CSL2.

8.1 Terminology

A *bond*[6] is a financial instrument similar to a loan, where a *holder* buys the bond from an *issuer* at a *principal* (initial) price. The holder may then receive regular *coupon* (interest)

[5]https://wiki.haskell.org/Rational

[6]https://www.investopedia.com/terms/b/bond.asp

payments from the issuer based on a *notional* amount, as well as being paid back the principal amount when the *maturity* (expiration) date of the bond is reached.

An *option*[7] is a contract that allows the holder (owner) to buy a specific asset at a pre-determined price (called the *strike price*). A *barrier option*[8] is an option tied to an asset that becomes either active (*knock-in*) or inactive (*knock-out*) if the price of the relevant asset reaches a certain *barrier price* during the lifetime of the option.

For the financial contracts we have implemented, the initial purchase of the bond or option is not part of the contract specification. This is because the contracts describe the *expectations* of the owner and the issuer, but there is no prerequisite expectation of an owner purchasing a bond or option. The contracts are instead instantiated with values determined at the initial purchase.

8.2 Zero-coupon bond

As the name suggests, the issuer of a *zero-coupon bond* makes no coupon payments, but only pays a notional amount at the maturity date. Using the CSL2 mathematical syntax, it can be expressed as

$$1(bondInfo); transfer; 1(verifyMaturityPayment)$$

where $bondInfo$ accepts the contract template input parameters, and $checkMaturityPayment$ checks the validity of the transfer events, causing the contract to fail if the transfer event was invalid.

With the embedded CSL2 language, the contract is written in the following way:

```
zeroCouponBond issuer currency notional maturityDate bdc =
  Event Transfer
  `suchthat`
  (\t -> sender t == issuer
      && (currency.resource) == currency
      && (money.resource) == notional
      && date t == bcd maturityDate)
```

Here `bdc` : $FreeDate \to FreeDate$ is a *business day convention*, which is a convention for how to translate a non-business day (i.e. weekends, national holidays) into a business day. It may be as simple as selecting the following business day, e.g. a saturday would be converted to next week's monday.

8.3 Fixed-rate bond

A *fixed-rate bond* or *bullet bond* is a loan in which the issuer pays interest on the notional amount at regular intervals, and then pays back the entire notional amount on the maturity date. The interest is usually paid in intervals of one or more months, but since the interest rate is given as an annual rate and not a monthly rate, one of the previously mentioned day count conventions must be used to compute the payment amount for each coupon.

[7] https://www.investopedia.com/terms/o/option.asp
[8] https://www.investopedia.com/terms/b/barrieroption.asp

```
payCoupon issuer bondCurrency startDate endDate
          notional interestRate dcc bdc =
  let couponFactor = dcc startDate endDate in
  let paymentAmount = (toRational notional)
                      * interestRate
                      * couponFactor in
    Event Transfer
    `suchthat`
    (\t -> sender t == issuer
        && (currency.resource) t == bondCurrency
        && (money.resource) t == roundMoney paymentAmount
        && date t == bdc endDate)
```

There is an interesting ambiguity regarding the semantics of this bond which is not immediately obvious. Suppose that the coupons are paid in intervals of a month, and that the first day of the initial month is a business day, but the first day of the second month (the first coupon payment date) is a weekend. It is clear that the coupon payment itself must then be rescheduled to a weekday, but should the payment amount change as a result of this? Furthermore, should the remaining coupon payments also be recalculated as a result of the offset?

```
payCoupons issuer bondCurrency startDate maturityDate
           frequency notional interestRate dcc bdc =
  if maturityDate <= startDate
  then return ()
  else do
    payCoupon issuer bondCurrency
              startDate (startDate `after` frequency)
              notional interestRate dcc bdc
    payCoupons issuer bondCurrency
               (startDate `after` frequency) maturityDate frequency
               notional interestRate dcc bdc
```

Our implementation calculates all coupon payment amounts based on the ”intended” date, even though the actual payments themselves may be rescheduled to another date.

Another interesting property is that the CSL2 option of returning values supports the alternative method quite nicely: the coupon payment subcontract can return the actual payment date as a value, allowing the remaining coupon payment dates and values to be calculated ”on the fly”.

Since the payment intervals may not completely line up with the start and end date, there is the possibility of having a *stub* in either end, where the payment interval is shortened. The fixed-rate template takes a boolean indicating whether the stub should be placed at the start or not.

```
bulletBond issuer bondCurrency notional interestRate dcc bdc
  (startDate, maturityDate, frequency, stubAtStart, stubTolerance) = do
  let stub = getStub startDate maturityDate frequency
  if stub == 0
  then payCoupons issuer bondCurrency
                  startDate maturityDate frequency
                  notional interestRate dcc bdc
  else if stubAtStart
       then do
            payCoupon issuer bondCurrency
                      startDate (startDate `afterDays` stub)
                      notional interestRate dcc bdc
            payCoupons issuer bondCurrency
                       (startDate `afterDays` stub) maturityDate frequency
                       notional interestRate dcc bdc
            return ()
       else do
            payCoupons issuer bondCurrency
                       startDate (maturityDate `beforeDays` stub)
                        ↪  frequency
                       notional interestRate dcc bdc
            payCoupon issuer bondCurrency
                      (maturityDate `beforeDays` stub) maturityDate
                      notional interestRate dcc bdc
            return ()
 `followedBy`
 Event Transfer
 `suchthat`
 (\t -> sender t == issuer
     && (currency.resource) t == bondCurrency
     && (money.resource) t == notional
     && date t == maturityDate)
```

Using do-notation allows implicit use of the >> monad operator; notice that `payCoupon` is followed by `payCoupons` without an assignment from the former. This may require some extra parentheses when also using infix notation, so an alternative is to use `followedBy`, which is equivalent to >>.

8.4 Floating-rate bond

The *floating-rate bond*, like the fixed-rate bond, has multiple coupons, however instead of having a fixed interest rate, the floating-rate bond has a varying interest over the course of the entire bond. Changes in interest rates are signaled through events, meaning that the contract should now listen for two different types of events. Although there may be some conventions regarding the intervals in which the interest rate changes, the contract itself has no predetermined knowledge of when the interest rate is updated; notably, it is (at least, it may be) independent of the coupon payment dates. This means that two scenarios can occur:

1. A coupon may have multiple interest updates during it, meaning that the coupon payment must be computed not only based on the different interest rates, but also for *how long* those rates were in effect.

2. The duration of an interest rate may span multiple coupons, requiring that long-term interest rates are *stored* between coupon payments.

For this contract, we move the interest-event collection into a separate contract. The contract collects all interest events until the next coupon deadline (using recursion), and then returns the collection of interest intervals as an output state.

```
getInterestRates interestRateRef mostRecentRateEnd deadline = do
  Event Interest
  `leadsto`
  (\(ref, val, (start, end)) ->
     if ref == interestRateRef && mostRecentRateEnd == start
     then
       if end < deadline
       then do
         interestRates <- getInterestRates interestRateRef end deadline
         return ((val, start, end):interestRates)
       else
         return [(val, start, end)]
     else
       failure)
```

This means that `payCoupons` must now perform both the coupon payments and interest observations. It may seem that two parallel contracts would be useful here, but parallel contracts cannot communicate with each other, so using sequential composition becomes necessary. Notice also the way that interest rates are passed through templates: the `lastInterestRate` is given as a parameter, and is also returned as the base case of the recursion. Since `getInterestRate` is not idempotent (nor is any event collecting contract), data based on collected events must be passed through templates.

```
payCoupons issuer bondCurrency couponStartDate maturityDate frequency
            notional lastInterestRate interestRateRef spread
            dcc bdc =
  let (_,_,interestExpiry) = lastInterestRate in
  let couponPayDate = couponStartDate `after` frequency in
  if maturityDate <= couponStartDate
  then
    return lastInterestRate
  else
    if maturityDate <= interestExpiry
    then do
      payCoupon issuer bondCurrency couponStartDate couponPayDate
                notional spread
                dcc bdc [lastInterestRate]
      payCoupons issuer bondCurrency couponPayDate maturityDate frequency
                 notional lastInterestRate interestRateRef spread
                 dcc bdc
    else do
      interestRates <-
        getInterestRates interestRateRef interestExpiry couponPayDate
      payCoupon issuer bondCurrency couponStartDate couponPayDate
                notional spread
                dcc bdc (lastInterestRate:interestRates)
      payCoupons issuer bondCurrency couponPayDate maturityDate frequency
                 notional (last interestRates) interestRateRef spread
                 dcc bdc
```

The contract template `floatingBond` is not too different from `bulletBond`, only that it has extra cases depending on whether or not the stub is covered by one or multiple interest periods.

8.5 Turbo warrant

The *turbo warrant* is a ”knock-out” barrier option, where the barrier price is the same as the strike price. Furthermore, the turbo warrant's final value depends on the asset prices that have been observed over the lifetime of the warrant, not only because of the possibility of the knockout, but because the notional amount paid back may depend on the closing (final) value of the asset. This necessitates that the values observed during the lifetime of the option are ”stored”, such that they can be accessed later, when the notional amount should be calculated. For our specific turbo warrant implementation, stocks are the asset to be monitored.

The observations are done in the `observe` and `collectObservations` contract templates, as shown:

```
observe (oType, oDate, oPrice) =
  Event Observation
  `suchthat`
  (\o -> obsType o == oType
      && obsDate o == oDate
      && obsPrice o == oPrice)
  `yielding`
  obsValue

collectObservations observationSpecs =
  case observationSpecs of
    [] -> NoEvent (Right [])
    o:os -> do
      obsResult <- withTerminateAndReplace (observe o)
      obsMap <- collectObservations os
      return $ (do val <- obsResult
                   obsMap' <- obsMap
                   return ((o,val):obsMap'))
```

The contract template is instantiated with a list of *observation specifications*, which describe a sequence of stock observations that must be mapped to the price of the stock at that time.

Specifically, each specification describes what type of observation it is (e.g. an underlying asset or a currency exchange rate), the date of the observation, and what value selection should be used (e.g the minimum or maximum value). The contract instantiates a subcontract *observe* with the first observation specification, which then returns the value associated with the observation event conforming to that specification. The contract then recursively calls itself with the remaining list, generating a map from ”specs” to values, which it then returns.

The above code is also an example of contract templates using callbacks: The function `withTerminateAndReplace` takes a contract as argument, and ”wraps” it in an alternative contract, where it may either return its original value or a tuple which specifies a new contract ID and a reason for termination. Since the `Either` monad works as an error monad, we can use do-notation to avoid including several `case of` statements.

```
terminateAndReplace :: Contract (String, String)
terminateAndReplace =
  Event Disruption
  `orelse`
  Event EconomicConditions
  `orelse`
  Event LegalReason

withTerminateAndReplace :: Contract a
                        -> Contract (Either (String, String) a)
withTerminateAndReplace contract =
   (terminateAndReplace `yielding` Left)
    `orelse`
   (contract `yielding` Right)
```

The main control flow of the turbo warrant itself is defined in the `turboWarrantRec` function which repeatedly checks for knock outs, paying a small early redemption if there is one, or paying a larger final redemption amount if there is none at the end.

```
turboWarrantRec ([], barrierSpecFunc, paymentSpec, twData) =
  finalRedemption ( warrantAgent twData
                  , warrantSetC twData
                  , (optDateSettlement . expiryAndSettlement) twData
                  , paymentSpec)

turboWarrantRec (d:ds, barrierSpecFunc, paymentSpec, twData) = do
  isKnockedOut <- knockOutCheck ( warrantAgent twData
                                , warrantSetC twData
                                , knockOutSettlement d
                                , barrierSpecFunc (knockOutValuation d))
  case isKnockedOut of
    Left err -> return $ Left err
    Right isKnockedOut' -> do
      if isKnockedOut'
        then do
```

```
          payment paymentArgs
          return $ Right ()
        else
          turboWarrantRec (ds, barrierSpecFunc, paymentSpec, twData)
  where
    paymentArgs = ( warrantAgent twData
                  , knockOutSettlement d
                  , toRational ((earlyRedemptionAmount twData)
                                * (warrantUnits twData))
                  , warrantSetC twData)
```

8.6 Barrier reverse convertible

The *barrier reverse convertible* shares similarities with both the fixed-rate bond and the turbo warrant:

1. Observe the initial value of an underlying asset (or the average of a *basket* of assets), and assign this value as the strike price.
2. Begin paying coupons at predetermined dates. Each coupon payment is set at a fixed percentage of the notional.
3. While paying coupons, also record the value of the underlying asset at fixed dates.
   - If the ratio between the current underlying value and the strike price is *higher* than a fixed *Automatic Early Settlement level* (i.e. the underlying value has increased by a certain factor), repay the notional amount and terminate the contract immediately, cancelling all future coupon payments.
   - If the ratio is lower, do nothing and continue the contract.
4. At the end of the contract, if no early settlement has occurred, the last value of the underlying asset is determined at a *settlement valuation date*.
   - If the ratio between this value and the strike price is *lower* than a predetermined *knock-in level*, gain a potentially increased payout using the following formula:

     $$notional * max(const_{\%} + gearing * put\_option, floor_{\%})$$

     where $const_{\%}$, $gearing$ and $floor_{\%}$ are preset factors, and put_option is the ratio between the underlying and the strike price, minus a preset *strike percentage*.
   - If the ratio is higher, only the notional is paid.

One of the more challenging issues with modelling this contract is in the beginning of the contract, where the underlying value must be determined. If the underlying consists of only a single asset, it is just a matter of consuming the event and returning the observed value.

```
getUnderlyingValue (SingleUnderlying ref) date =
  Event Observation
  `suchthat`
  (\o -> obsType o == Underlying ref
      && obsDate o == date)
  `yielding`
  obsValue
```

However, if there are multiple underlyings (a basket), the contract does not specify the order in which the underlyings are observed. Thus, if we have underlyings with indexes ”1A”, ”2B” and ”3C”, they might arrive in the order ”2B”→”3C”→”1A”. This means that we must construct a *parallel* contract, where each branch gathers exactly one observation for each underlying. We construct the parallel contract using a list of observations and `foldr`. Since a parallel contract returns a tuple, but a list of values is needed for the processing function, we use the `yielding` function to convert the tuple into a list. Finally, the list is given to the processing function, which returns a final value (generally the average of all the values).

```
getUnderlyingValue (MultiUnderlying (ref:refs) processFunc) date = do
  let firstObs =
        Event Observation
        `suchthat`
        (\o -> obsType o == Underlying ref
            && obsDate o == date)
        `yielding`
        (\o -> [o])
  let makeParallel =
        (\r contract ->
            (Event Observation
             `suchthat`
             (\o -> obsType o == Underlying r
                 && obsDate o == date)
             `andalso`
             contract)
            `yielding`
            (\(o,os) -> o:os))
  foldr makeParallel firstObs refs
  `yielding`
  processFunc
```

Most of the contract is executed in the `payCouponsWithAES` function, which is both responsible for paying the coupons while also comparing the underlying to the AES level. The function returns a boolean stating if the AES level has been reached or not.

```
payCouponsWithAES issuer setCurrency notional
                  couponRate underlying []
                  aesCheckAndPayDates aesLevel strikePrice = do
  checkAES issuer setCurrency notional underlying
           strikePrice aesCheckAndPayDates aesLevel

payCouponsWithAES issuer setCurrency notional
```

```
                  couponRate underlying
                  couponDates [] aesLevel strikePrice = do
  payCoupons issuer setCurrency couponDates notional couponRate
  return False

payCouponsWithAES issuer setCurrency notional
                  couponRate underlying couponDates
                  aesCheckAndPayDates aesLevel strikePrice = do
  let nextCouponDate:couponDates' = couponDates
  let (aesCheckDate,aesPayDate):aesCheckAndPayDates' = aesCheckAndPayDates
  if nextCouponDate < aesCheckDate
  then do
    payCoupon issuer setCurrency nextCouponDate notional couponRate
    payCouponsWithAES issuer setCurrency notional
                      couponRate underlying couponDates'
                      aesCheckAndPayDates aesLevel strikePrice
  else do
    aesBarrierHit <-
      checkAES issuer setCurrency notional underlying
               strikePrice [(aesCheckDate,aesPayDate)] aesLevel
    if aesBarrierHit
    then return True
    else payCouponsWithAES issuer setCurrency notional
                           couponRate underlying couponDates
                           aesCheckAndPayDates' aesLevel strikePrice
```

Even though it is usually the case, there is no predetermined relation between the aes-checking dates and the coupon-payment dates, so the function simply takes the most recent date of both, performs the relevant action (check for a barrier hit or pay the coupon), and continues recursively with the other date still remaining.

The `barrierReverse` contract template is straightforward:

```
barrierReverse issuer setCurrency refCurrency underlying notional
               aesCheckAndPayDates couponDates strikeDates settlementDates
               couponRate aesLevel cc
               (knockInLevel, constPercent, strikePercent,
                gearing, floorPercent) = do
  strikePrice <-
    getStrikePrice underlying strikeDates
  aesActivated <-
    payCouponsWithAES issuer setCurrency notional
                      couponRate underlying couponDates
                      aesCheckAndPayDates aesLevel strikePrice
  if aesActivated
  then return ()
  else do
       payMaturity issuer setCurrency refCurrency notional strikePrice
                   underlying cc settlementDates
```

```
                  (knockInLevel, constPercent, strikePercent,
                   gearing, floorPercent)
        return ()
```

It obtains the strike price, then pays the coupons while checking the AES level, and finally, if the AES level has not been reached, pays the maturity.

8.7 Evaluation

It is perhaps time to step back a little bit and consider: would a domain expert, who is not proficient in computer programming, be able to understand the code presented in this section? Some of the simpler contracts such as the zero coupon bond or the `observe` contract template may be fully readable, but the more extensive contracts are certainly not without some assistance from a programmer. Even the contracts that do not use any functional-style programming and rely solely on do-notation may prove to be a substantial challenge to read.

However, as previously mentioned, the goal of this project was not to design a language that would make contract-programmers obsolete, but to design a language that could help *bridge the communication gap* between contract programmers and domain experts. The importance is not that all parts of the contract can be read by domain experts, but that the *general structure* of the contract is decipherable *with* the aid of a programmer. This is certainly the case; for example, even if the entirety of the barrier reverse convertible program is not readable, the `barrierReverse` contract template makes the control flow clear.

We believe that overall, CSL2 makes a significant contribution towards bridging this communication gap.

9 Discussion

9.1 Related work

Modelling resource systems and transactions using notions of resources, agents and events was first presented in 1982 by McCarthy [11], but this system did not introduce any way of of defining ”obligations”, which would be necessary for modelling contracts.

One of the first compositional contract languages is developed by Jones et. al [12]. The language is designed towards use in the field of financial engineering, and it shares similar compositional constructs with CSL such as `and`, `then` and `or`. However, in this language, a contract is not residualized to a ”success” or ”failure” as with CSL, but it is instead modelled as *partial function* from *time* to a *value*, where the result of the contract depends on the time of evaluation (and it may not evaluate at all, hence the partial function).

CSL was first defined by Andersen et. al [6], where it combines the concepts found in the previous cases: contracts are defined using ”atomic” contracts and contract combinators, but contract evaluation itself is event driven through *transmit* events that specify resource transfers between agents.

It would be difficult to discuss related work without remarking on the concept of *smart contracts*. The history of the term is itself interesting: the term ”smart contracts” originates from the 1990s, where one of its first usages is in the description of the rights and obligations layer *FIRM* of the *Stanford InfoBus* [13], which aimed to introduce internet protocols for information management. The smart contracts presented in the paper (also called *commpacts*) are concrete objects instantiated from contract templates, and are represented as a combination of ”informal textual descriptions” and code. These contracts are described as similar to state machines, and are also event-driven like CSL contracts, but the contract description and implementation are two distinct objects, which is fundamentally different from CSL contracts, where the contract *is* the implementation. The meaning of the term has gradually changed with the introduction of distributed ledger technology. The Ethereum foundation is somewhat vague [9] regarding the nature of smart contracts, and in some cases [10] directly state that ”*A 'smart contract' is simply a program that runs on the Ethereum Blockchain*”. This is also further supported by the case that smart contracts in Ethereum are often written in Turing-complete languages such as Solidity, which is opposite to what CSL tries to accomplish with its limited features.

As an example, this is what a zero-coupon bond may look like when written in Solidity:

```
struct AccountCurrency {
  address accountOwner;
  uint currencyID;
}

contract ZeroCoupon {
  address issuer; address owner; uint notional; uint currencyID;
  mapping (AccountCurrency => uint) public accountBalance;

  event bondPaid(address issuer, address owner, uint currencyID,
                 uint amount, uint dateNumber);

  constructor (address _owner, uint _notional, uint _currencyID) {
    issuer = msg.sender; owner = _owner;
    notional = _notional; currencyID = _currencyID;
    accountBalance[AccountCurrency(issuer, currencyID)] += notional;
    accountBalance[AccountCurrency(owner, currencyID)]  -= notional;
  }

  function payBond(uint paymentDate) public {
    require(msg.sender == issuer);
    accountBalance[AccountCurrency(issuer, currencyID)] -= notional;
    accountBalance[AccountCurrency(owner, currencyID)]  += notional;
    emit bondPaid(issuer, owner, currency, notional, paymentDate);
  }
}
```

Even disregarding the lack of a parameterized business-day convention, there are large differences between this and ”equivalent” CSL/CSL2 programs. In Solidity, contracts are more similar to classes in object-oriented programming, and even though the concept of events also exists in this language, it is not as ubiquitous as in CSL/CSL2; in fact, event listening does not even occur in Solidity, but as part of a related Javascript framework.

The *Plutus Platform*, which is built on the *Cardano* blockchain, provides a grammar much closer to CSL. The Plutus Platform is provided as a set of Haskell libraries which, com-

[9]ethereum.org/en/whitepaper/

[10]ethereum.org/en/developers/docs/smart-contracts/

bined with the syntax it uses, effectively makes it a type of embedded DSL, similar to CSL2. However, there are some drawbacks: its use of boilerplate code is not insignificant, and the language is mostly oriented towards transfers across distributed ledgers, which may make it a less suitable medium for ”traditional” financial contracts.

An even more specialized language for smart contract design would be *Marlowe*, which is built on top of the Plutus Platform. It is implemented as both an embedded DSL in Haskell, and a separate external DSL. Marlowe is based on combinatory, event-driven contracts. An example of a 100 DKK payment followed by a bike transfer might be expressed in ”external” Marlowe with

```
(Pay
  (Role "Alice")
  (Role "Bob")
  (Token "DKK" "")
  (Constant 100)
  (Assert
    (Role "Bob")
    (Role "Alice")
    (Token "" "Bike")
    (Constant 1)
    Close))
```

In this case, the transfer of a bike is represented by the `Pay` construct, but since payments are usually tied to financial currency and not real-world objects, an improved alternative might use the `Assert` construct to verify that the actual transfer of the bike has occurred.

An interesting property that Marlow exhibits, due to its specialized nature, is that it allows some degree of static analysis, akin to CSL. Examples of properties that can be statically determined are partial payments; payments where the payer does not have the necessary resources in their account, and non-positive payments; where a payment of 0 or a negative amount is attempted.

One of the main differences between Marlowe and CSL is that for CSL, the internal value expression language is independent of the contract composition language, the ”main” CSL. This guarantees that CSL internal expressions are never limited by language constraints.

9.2 Future work

Despite having presented both operational and denotational semantics for CSL2, we have defined no significant theorems and made few claims about the properties of the language. For example, we have presented both big-step or small-step semantics of CSL2, but we have not made any attempt to show that these two semantics are equivalent.

The CSL2 language, as-is, is able to model event-driven financial contracts, but the circumstances for how each contract should receive such events are unspecified. The examples so far have all consisted of ”made up” events for testing instantiated versions of the different types of contracts. We previously mentioned that the next step would be to create a contract manager, which handles communication between the instantiated contracts and the relevant actors. The contract manager would maintain a database of active contracts, and would convert incoming notifications of real events to CSL2-style Haskell types, which are then applied to the instantiated contracts. Other managers in the system are not limited to a resource manager, but may also include an an identity manager, which authorizes transfers, and an observation manager, which handles continous observations of asset values. If such system of managers was to be implemented, it would likely be done using a web-framework and accessed by users though HTTP requests (or some other type of wrapping API).

10 Conclusion

We have developed a prototype version of CSL2, a successor to the current Deon Digital CSL. Development strategy was based on established DSL design principles, electing us to choose a deep embedding for the DSL. We have defined the mathematical semantics for the language, and shown that the contract type exhibits monadic properties. The implementation has been done in the host language Haskell, where novel algebraic datatypes for time have been implemented as well. Various financial contracts such as bonds and warrants have been implemented in CSL2 and tested, showing satisfactory results with regards to expressiveness and readability. Future work would primarily involve scaling the current contract evaluator to a contract manager and connecting it to resource managers and identity/authorization managers, in order to make CSL2 contracts integrable with external financial systems.

References

[1] Ali Sunyaev. *Distributed Ledger Technology*, pages 265–299. Springer International Publishing, Cham, 2020.

[2] Martin Fowler and Rebecca Parsons. *Domain-specific languages.* Addison-Wesley, 2011.

[3] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of systems and software*, 56(1):91–99, 2001.

[4] Tomaž Kosar, Pablo E Martı, Pablo A Barrientos, Marjan Mernik, et al. A preliminary study on various implementation approaches of domain-specific language. *Information and software technology*, 50(5):390–405, 2008.

[5] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 339–347, 2014.

[6] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer*, 8(6):485–516, 2006.

[7] Deon Digital Core Technology team. CSL2 design: Semantic considerations. *Unpublished notes*, 2020.

[8] Fritz Henglein. It's about time (draft). *Unpublished notes*, 2020.

[9] Edward M. Reingold and Nachum Dershowitz. *Calendrical Calculations: The Ultimate Edition.* Cambridge University Press, 4 edition, 2018.

[10] Juan-Manuel Torres. Algebraic resource accounting for transfers and transformations. *MSc Thesis in Computer Science, DIKU*, 2020.

[11] William E McCarthy. The rea accounting model: A generalized framework for accounting systems in a shared data environment. *Accounting review*, pages 554–578, 1982.

[12] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering. *ACM SIG-PLAN Notices*, 35(9):280–292, 2000.

[13] Martin Röscheisen, Michelle Baldonado, Kevin Chang, Luis Gravano, Steven Ketchpel, and Andreas Paepcke. The stanford infobus and its service layers. *Augmenting the Internet with Higher-Level Information Management Protocols http://www-diglib. stanford. edu/cgi-bin/WP/get/SIDL-WP-1997-0065 [eingesehen: 26.05. 98]*, 1997.

11 Appendix

11.1 Contract.hs

```haskell
{-# OPTIONS -XExistentialQuantification -XGADTs -fwarn-incomplete-patterns
↪  #-}

{- CSL2 contracts modeled monadically -}

module Contract where

import Control.Monad
import Control.Applicative
import FreeTime

data CallOptionData = CallOptionData
    { optionStockTicker :: String
    , optionStrike :: Integer
    , optionExpiry :: FreeDate
    , optionUnits :: Integer
    , optionSettlementDate :: FreeDate
    } deriving (Show, Eq)

type Agent = String
type Currency = String

type Resource = (String, Integer)

type Information = String
type Date = FreeDate
type Time = Integer
data Underlying =
    Stock String
  | Index String
  deriving (Show, Eq)
data ObservationType =
    Underlying Underlying
  | FXRate Currency Currency
  deriving (Show, Eq)
data PriceSelection =
    Opening
  | Closing
  | Minimum
  | Maximum
  | AtTimePoint Date
  deriving (Show, Eq)


```

```
-- ////////// Turbo Warrant Types //////////
-- // Auxiliary type for knock-out valuation and corresponding settlement
 ↪  date
data KnockOutCheckDates = KnockOutCheckDates
    { knockOutValuation :: Date
    , knockOutSettlement :: Date
    } deriving (Show, Eq)

data SettlementDates = SettlementDates
    { settlementValuation :: Date
    , settlementMaturity :: Date
    } deriving (Show, Eq)

-- // Auxiliary type for expiry and corresponding settlement date
data OptDates = OptDates
    { optDateExpiry :: Date
    , optDateSettlement :: Date
    } deriving (Show, Eq)

-- // Type for picking one of two turbo warrant types: Down-and-Out Call
 ↪  or Up-and-Out Put
data WarrantType =
    DownAndOutCall
  | UpAndOutPut
  deriving (Show, Eq)

-- // Record type to describe a turbo warrant.
-- // For simplicity we assume the dates to be given (as a list).
data TurboWarrantData = TurboWarrantData
    { warrantAgent :: Agent
    , warrantUnderlying :: Underlying
    , warrantUnits :: Integer
    , warrantRefC :: Currency
    , warrantSetC :: Currency
    , strikeAndBarrier :: Integer
    , conversionRatio :: Integer
    , earlyRedemptionAmount :: Integer
    , warrantType :: WarrantType
    , knockOutCheckDates :: [KnockOutCheckDates]
    , expiryAndSettlement :: OptDates
    } deriving (Show, Eq)
-- ////////////////////////////////////////


-- Single events resulting in state b

data Event b where
   Transfer :: Event (Agent, Agent, Resource, Date)
```

```haskell
   Delivery :: Event (Agent, Agent, Resource, Date)
   Notification :: Event (Agent, Agent, Information, Date)
   ClosingStockPrice :: Event (String, Date, Integer)
   PaymentConfirmation :: Event (Agent, Currency, Integer, Date)
   PaymentCorrection :: Event (Agent, Currency, Integer, Date)
   Disruption :: Event (String, String)
   EconomicConditions :: Event (String, String)
   LegalReason :: Event (String, String)
   Observation :: Event (ObservationType, Date, PriceSelection, Rational)
   Config :: Event (Agent, String, Integer, Integer, [(Date, Date)])
   TurboWarrant :: Event TurboWarrantData
   Interest :: Event (String, Rational, (Date, Date))

data EventInfo =
    TransferInfo (Agent, Agent, Resource, Date)
  | DeliveryInfo (Agent, Agent, Resource, Date)
  | NotificationInfo (Agent, Agent, Information, Date)
  | EventInfo (Agent, Agent, Information, Date)
  | ClosingStockPriceInfo (String, Date, Integer)
  | PaymentConfirmationInfo (Agent, Currency, Integer, Date)
  | PaymentCorrectionInfo (Agent, Currency, Integer, Date)
  | DisruptionInfo (String, String)
  | EconomicConditionsInfo (String, String)
  | LegalReasonInfo (String, String)
  | ObservationInfo (ObservationType, Date, PriceSelection, Rational)
  | ConfigInfo (Agent, String, Integer, Integer, [(Date, Date)])
  | TurboWarrantInfo TurboWarrantData
  | InterestInfo (String, Rational, (Date, Date))
  deriving (Show, Eq)

matchevent :: Event a -> EventInfo -> Maybe a
matchevent Transfer (TransferInfo (a, b, r, d)) = Just (a, b, r, d)
matchevent Delivery (DeliveryInfo (a, b, r, d)) = Just (a, b, r, d)
matchevent Notification (NotificationInfo (a, b, r, d)) = Just (a, b, r,
 ↪  d)
matchevent ClosingStockPrice (ClosingStockPriceInfo (a, b, c)) = Just (a,
 ↪  b, c)
matchevent PaymentConfirmation (PaymentConfirmationInfo (a, b, c, d)) =
 ↪  Just (a, b, c, d)
matchevent PaymentCorrection (PaymentCorrectionInfo (a, b, c, d)) = Just
 ↪  (a, b, c, d)
matchevent Disruption (DisruptionInfo (a, b)) = Just (a, b)
matchevent EconomicConditions (EconomicConditionsInfo (a, b)) = Just (a,
 ↪  b)
matchevent LegalReason (LegalReasonInfo (a, b)) = Just (a, b)
matchevent Observation (ObservationInfo (a, b, c, d)) = Just (a, b, c, d)
matchevent Config (ConfigInfo (a, b, c, d, e)) = Just (a, b, c, d, e)
matchevent TurboWarrant (TurboWarrantInfo twData) = Just twData
matchevent Interest (InterestInfo info) = Just info
matchevent _ _ = Nothing

-- Auxiliary functions (in the absence of decent record types in Haskell)

sender :: (Agent, b, c, d) -> Agent
sender (a,_,_,_) = a
receiver :: (a, Agent, c, d) -> Agent
receiver (_,b,_,_) = b
resource :: (a, b, Resource, d) -> Resource
resource (_,_,r,_) = r
information :: (a, b, Information, d) -> Information
information (_,_,i,_) = i
date :: (a, b, c, Date) -> Date
date (_,_,_,d) = d
time :: (a, b, c, Date) -> Time
time (_,_,_,t) = dayNumberGregorian t
stockTicker (st,_,_) = st
date2 (_,d,_) = d
amount (_,_,a,_) = a
payee (a,_,_,_) = a
getCurrency (_,c,_,_) = c
justification (s,_) = s
replacementID (_,s) = s
cfgAgent (a,_,_,_,_) = a
cfgStockTicker (_,st,_,_,_) = st
cfgUnits (_,_,a,_,_) = a
cfgStrike (_,_,_,s,_) = s
cfgDates (_,_,_,_,ds) = ds
obsType (t,_,_,_) = t
obsDate (_,d,_,_) = d
obsPrice (_,_,p,_) = p
obsValue (_,_,_,v) = v

money :: Resource -> Integer
money (_,m) = m
currency :: Resource -> Currency
currency (c,_) = c

data Contract b where
   Fail    :: Contract b
   NoEvent :: b -> Contract b
   Event   :: Event b -> Contract b
   Then    :: Contract a -> (a -> Contract b) -> Contract b
   Par     :: Contract a -> Contract b -> Contract (a, b)
   Alt     :: Contract b -> Contract b -> Contract b

yielding :: Contract a -> (a -> b) -> Contract b
yielding c f = c `Then` (NoEvent . f)

followedBy :: Contract a -> Contract b -> Contract b
followedBy c c' = c `Then` (\_ -> c')

failure :: Contract a
failure = Fail

instance Functor Contract where
  fmap f c = c >>= (return.f)

instance Applicative Contract where
  pure = NoEvent
  f <*> c = f `Then` \f' -> c `yielding` f'

instance Monad Contract where
  return = NoEvent
  (>>=) = Then

instance Alternative Contract where
  empty = failure
  (<|>) = Alt

-- MonadZero for collecting values: lists

type M a = [a]

choose :: a -> M (Either a a)
choose x = mplus (return (Left x)) (return (Right x))

select :: (a -> Bool) -> a -> Either a a
select p x = if p x then Left x else Right x

deselect :: Either a a -> a
deselect (Left x) = x
deselect (Right x) = x

-- Smart contract constructors

noevent :: a -> Contract a
noevent = NoEvent

suchthat :: Contract a -> (a -> Bool) -> Contract a
suchthat c p = c `Then` \x -> if p x then return x else failure

andthen :: Contract a -> Contract b -> Contract (a, b)
andthen c1 c2 = c1 `Then` \x -> (NoEvent x) `Par` c2

```

```haskell
orelse :: Contract a -> Contract a -> Contract a
orelse = Alt

andalso :: Contract a -> Contract b -> Contract (a,b)
andalso = Par

leadsto :: Contract a -> (a -> Contract b) -> Contract b
leadsto = Then

-- State machine semantics (streaming event processing)

manifestlyFailed :: Contract b -> Bool
manifestlyFailed contract =
  case contract of
    Fail ->
      True
    NoEvent b ->
      False
    Event _ ->
      False
    Then c _ ->
      manifestlyFailed c
    Par c c' -> do
      manifestlyFailed c
      ||
      manifestlyFailed c'
    Alt c c' ->
      manifestlyFailed c
      &&
      manifestlyFailed c'

nullify :: Contract b -> [b]
nullify contract =
  case contract of
    Fail ->
      mzero
    NoEvent b ->
      return b
    Event _ ->
      mzero
    Then c c' -> do
      b <- nullify c
      nullify (c' b)
    Par c c' -> do
      b <- nullify c
      b' <- nullify c'
      return (b, b')
    Alt c c' ->
```

```haskell
      (nullify c) ++ (nullify c')

apply :: EventInfo -> Contract b -> Contract b
apply eventinfo contract =
  case contract of
    Fail ->
      failure
    Event event ->
      case matchevent event eventinfo of
        Just info -> return info
        Nothing -> failure
    NoEvent _ ->
      failure
    Then c c' -> do
      (apply eventinfo c) `Then` c'
      <|>
      (foldr (\e a -> apply eventinfo (c' e) <|> a) (failure) (nullify c))
    Par c c' -> do
      (apply eventinfo c) `Par` c'
      <|>
      c `Par` (apply eventinfo c')
    Alt c c' -> do
      apply eventinfo c
      <|>
      apply eventinfo c'

eval :: [EventInfo] -> Contract b -> [b]
eval events contract =
  case events of
    [] ->
      nullify contract
    e:es -> do
      eval es r_contract
      where
        r_contract = apply e contract

succeeds :: String -> [a] -> IO ()
succeeds s [] = putStrLn ("[!] " ++ s ++ " failed")
succeeds s _  = return ()

fails :: String -> [a] -> IO ()
fails s [] = return ()
fails s _ = putStrLn ("[!] " ++ s ++ " failed")
```

11.2 FreeTime.hs

```haskell
module FreeTime where

import Data.Ix

type Year = Integer
type Month = Integer
type Day = Integer

data FreeDate =
  Date Year Month Day
  deriving (Show, Eq)

type Duration = FreeDate

day0Difference :: FreeDate -> FreeDate
day0Difference (Date y m d) =
  Date y' m' d'
  where
    year_offset = ((m-3) `div` 12)
    y' = y + year_offset
    m' = ((m-3) `mod` 12)
    d' = d-1

difference :: FreeDate -> FreeDate -> Duration
difference (Date y1 m1 d1) (Date y2 m2 d2) =
  Date (y2-y1) (m2-m1) (d2-d1)

after :: FreeDate -> Duration -> FreeDate
after (Date y1 m1 d1) (Date y2 m2 d2) =
  Date (y1+y2) (m1+m2) (d1+d2)

before :: FreeDate -> Duration -> FreeDate
before = difference

durationThirty360 :: FreeDate -> FreeDate -> Integer
durationThirty360 start end =
  y*360 + m*30 + d
  where
    Date y m d = difference start end

normalize :: FreeDate -> FreeDate
normalize (Date y m d) =
  Date (y + year_offset) (remMonths + 1) d
  where
    year_offset = (m-1) `div` 12
    remMonths   = (m-1) `mod` 12
```

```haskell

julianLeapYear :: Integer -> Bool
julianLeapYear y = y `mod` 4 == 0

julianDaysInMonth :: FreeDate -> Integer
julianDaysInMonth (Date y m d)
  | m `elem` [4, 6, 9, 11] = 30
  | m == 2 && julianLeapYear y = 29
  | m == 2 = 28
  | otherwise = 31

validJulian :: FreeDate -> Bool
validJulian (Date y m d) =
     m >= 1
  && m <= 12
  && d >= 1
  && d <= julianDaysInMonth (Date y m d)

nextJulian :: FreeDate -> FreeDate
nextJulian (Date y m d)
  | d < julianDaysInMonth (Date y m d) = Date y m (d+1)
  | d == julianDaysInMonth (Date y m d) && m < 12 = Date y (m+1) 1
  | d == julianDaysInMonth (Date y m d) && m == 12 = Date (y+1) 1 1

nextNJulian :: FreeDate -> Integer -> FreeDate
nextNJulian date 0 = date
nextNJulian date n = nextNJulian (nextJulian date) (n-1)

month_length :: [Integer]
month_length = [31, 30, 31, 30, 31, 31, 30, 31, 30, 31, 31, 29]
month_offset = [0, 31, 61, 92, 122, 153, 184, 214, 245, 275, 306, 337]

lookupMonthDay :: Integer -> (Month, Day)
lookupMonthDay n =
  (m, d)
  where
    m = (concat $ map (\(m,n) -> replicate (fromIntegral n) m)
                        (zip (range (0,11)) month_length))
        !! (fromIntegral n)
    d = (concat $ map (\e -> range (0,e-1)) month_length) !! (fromIntegral
     ↪  n)

dayNumberJulian :: FreeDate -> Integer
dayNumberJulian (Date y m d) =
  year_days + offset + d'
  where
    Date y' m' d' = day0Difference (Date y m d)
    year_days = 365*y' + (y' `div` 4)
    offset = month_offset !! (fromIntegral m')

gregorianLeapYear :: Integer -> Bool
gregorianLeapYear y =
     ((y `mod` 4 == 0) && (y `mod` 100 /= 0))
  || (y `mod` 400 == 0)

gregorianDaysInMonth :: FreeDate -> Integer
gregorianDaysInMonth (Date y m d)
  | m `elem` [4, 6, 9, 11] = 30
  | m == 2 && gregorianLeapYear y = 29
  | m == 2 = 28
  | otherwise = 31

validGregorian :: FreeDate -> Bool
validGregorian (Date y m d) =
     m >= 1
  && m <= 12
  && d >= 1
  && d <= gregorianDaysInMonth (Date y m d)

dayNumberGregorian :: FreeDate -> Integer
dayNumberGregorian date =
  year_days + month_days + d
  where
    Date y m d = day0Difference date
    year_days = 365*y + (y `div` 4) - (y `div` 100) + (y `div` 400)
    month_days = month_offset !! (fromIntegral m)

fromDayNumberGregorian :: Integer -> FreeDate
fromDayNumberGregorian n =
  if remDays4 == 1460
  then Date (years400 + years100 + years4 + 4) 2 29
  else if remDays400 == 146096
       then Date (years400 + years100) 2 29
       else normalize (Date y m d)
  where
    epochs400   = n           `div` 146097
    remDays400  = n           `mod` 146097
    years400    = epochs400     *   400
    epochs100   = remDays400  `div` 36524
    remDays100  = remDays400  `mod` 36524
    years100    = epochs100     *   100
    epochs4     = remDays100  `div` 1461
    remDays4    = remDays100  `mod` 1461
    years4      = epochs4       *   4
    years1      = remDays4    `div` 365
    remDays1    = remDays4    `mod` 365
    (m', d')    = lookupMonthDay (remDays1)
    y           = years400 + years100 + years4 + years1
    m           = m' + 3
    d           = d' + 1

year (_,_,_,Date y _ _) = y
month (_,_,_,Date _ m _) = m
day (_,_,_,Date _ _ d) = d

freeDateToGregorian :: FreeDate -> FreeDate
freeDateToGregorian = fromDayNumberGregorian.dayNumberGregorian

afterDays :: FreeDate -> Integer -> FreeDate
afterDays date d =
  fromDayNumberGregorian (dayNumberGregorian date + d)

beforeDays :: FreeDate -> Integer -> FreeDate
beforeDays date d =
  fromDayNumberGregorian (dayNumberGregorian date - d)

comesBefore :: FreeDate -> FreeDate -> Bool
comesBefore date1 date2 =
  (dayNumberGregorian date1) < (dayNumberGregorian date2)

comesAfter :: FreeDate -> FreeDate -> Bool
comesAfter date1 date2 =
  (dayNumberGregorian date1) > (dayNumberGregorian date2)

thirty360 :: FreeDate -> FreeDate -> Rational
thirty360 dateStart dateEnd =
  (fromIntegral y) + (fromIntegral m)/12 + (fromIntegral d)/360
  where
    Date y m d = difference dateStart dateEnd

germanThirty360 :: FreeDate -> FreeDate -> Rational
germanThirty360 dateStart dateEnd =
  let adjustDate =
        (\(Date y m d) ->
           if gregorianDaysInMonth (Date y m d) == d
           then Date y m 30
           else Date y m d) in
    thirty360 (adjustDate dateStart) (adjustDate dateEnd)

usThirty360 :: FreeDate -> FreeDate -> Rational
usThirty360 dateStart dateEnd =
  let adjustDate =
        (\(Date y1 m1 d1) (Date y2 m2 d2) ->
           if d2 == 30 && d1 == gregorianDaysInMonth (Date y1 m1 d1)
```

```haskell
              then (Date y1 m1 30)
              else (Date y1 m1 d1)) in
  let dateStart' = adjustDate dateStart (Date 0 0 30) in
  let dateEnd' = adjustDate dateEnd dateStart' in
    thirty360 dateStart' dateEnd'

instance Ord FreeDate where
  a <= b = not (b `comesBefore` a)
  a < b = a `comesBefore` b
  a > b = a `comesAfter` b
  a >= b = not (b `comesAfter` a)
```

11.3 ZeroCoupon.hs

```haskell
import Contract
import FreeTime

-- type DayCountConvention = (Date -> Date -> Rational)
type BusinessDayConvention = (Date -> Date)

-- Ideally, this should be parametrized
-- roundMoney :: Rational -> Integer
-- roundMoney = round

-- The zero coupon bond
zeroCouponBond :: Agent -> Currency -> Integer -> Date ->
↪ BusinessDayConvention
               -> Contract (Agent, Agent, Resource, Date)
zeroCouponBond issuer paymentCurrency notional maturityDate bdc =
  Event Transfer
  `suchthat`
  (\t -> sender t == issuer
      && (currency.resource) t == paymentCurrency
      && (money.resource) t == notional
      && date t == bdc maturityDate)

zcCorrect = -- Correct transfer
  let c0 = zeroCouponBond "Alice" "DKK" 42 (Date 2021 06 19) id in
  let c1 = apply (TransferInfo ("Alice", "Bob", ("DKK", 42), Date 2021 06
  ↪ 19)) c0 in
    nullify c1

zcWrongAmount = -- Wrong amount of money transferred
  let c0 = zeroCouponBond "Alice" "DKK" 42 (Date 2021 06 19) id in
  let c1 = apply (TransferInfo ("Alice", "Bob", ("DKK", 43), Date 2021 06
  ↪ 19)) c0 in
    nullify c1
```

```haskell

zcWrongCurrency = -- Wrong type of currency transferred
  let c0 = zeroCouponBond "Alice" "DKK" 42 (Date 2021 06 19) id in
  let c1 = apply (TransferInfo ("Alice", "Bob", ("DKK", 43), Date 2021 06
   ↪ 19)) c0 in
    nullify c1

zcWrongDate = -- Wrong date of transfer
  let c0 = zeroCouponBond "Alice" "DKK" 42 (Date 2021 06 19) id in
  let c1 = apply (TransferInfo ("Alice", "Bob", ("DKK", 43), Date 2021 06
   ↪ 19)) c0 in
    nullify c1

main :: IO ()
main = do
  succeeds "zcCorrect" zcCorrect
  fails "zcWrongAmount" zcWrongAmount
  fails "zcWrongCurrency" zcWrongCurrency
  fails "zcWrongDate" zcWrongDate
```

11.4 BulletBond.hs

```haskell
import Contract
import FreeTime

type DayCountConvention = (Date -> Date -> Rational)
type BusinessDayConvention = (Date -> Date)

-- Ideally, this should be parametrized
roundMoney :: Rational -> Integer
roundMoney = round

-- Get the length of the stub in number of days
getStub :: Date -> Date -> Duration -> Integer
getStub startDate maturityDate duration =
  let nextDate = startDate `after` duration in
  if nextDate <= maturityDate
  then getStub nextDate maturityDate duration
  else (dayNumberGregorian maturityDate) - (dayNumberGregorian startDate)

-- Pay a single coupon from startDate to endDate with a given interestRate
payCoupon :: Agent -> Currency -> Date -> Date -> Integer -> Rational
          -> DayCountConvention -> BusinessDayConvention
          -> Contract (Agent, Agent, Resource, Date)
payCoupon issuer bondCurrency startDate endDate notional interestRate dcc
 ↪ bdc =
  let couponFactor = dcc startDate endDate in
  let paymentAmount = (toRational notional) * interestRate * couponFactor
   ↪  in
    Event Transfer
    `suchthat`
    (\t -> sender t == issuer
        && (currency.resource) t == bondCurrency
        && (money.resource) t == roundMoney paymentAmount
        && date t == bdc endDate)

-- Pay multiple coupons from startDate to maturityDate with a given
 ↪  frequency
payCoupons :: Agent -> Currency -> Date -> Date -> Duration -> Integer ->
 ↪  Rational
           -> DayCountConvention -> BusinessDayConvention
           -> Contract ()
payCoupons issuer bondCurrency startDate maturityDate frequency notional
 ↪  interestRate dcc bdc =
  if maturityDate <= startDate -- If there are no more coupons
  then return ()
  else do
       payCoupon issuer bondCurrency startDate (startDate `after`
        ↪  frequency)
                 notional interestRate dcc bdc
       payCoupons issuer bondCurrency (startDate `after` frequency)
        ↪  maturityDate frequency
                  notional interestRate dcc bdc

-- The bullet bond / fixed-rate bond
bulletBond :: Agent -> Currency -> Integer -> Rational
           -> DayCountConvention -> BusinessDayConvention
           -> (Date, Date, Duration, Bool, Duration)
           -> Contract (Agent, Agent, Resource, Date)
bulletBond issuer bondCurrency notional interestRate dcc bdc
  (startDate, maturityDate, frequency, stubAtStart, stubTolerance) = do
  let stub = getStub startDate maturityDate frequency
  if stub == 0 -- No stub
  then payCoupons issuer bondCurrency startDate maturityDate frequency
                 notional interestRate dcc bdc
  else if stubAtStart
       then do
            payCoupon issuer bondCurrency startDate (startDate `afterDays`
             ↪  stub)
                      notional interestRate dcc bdc
            payCoupons issuer bondCurrency (startDate `afterDays` stub)
             ↪  maturityDate frequency
                       notional interestRate dcc bdc
            return ()
       else do
```

```
            payCoupons issuer bondCurrency startDate (maturityDate
            ↪  `beforeDays` stub) frequency
                       notional interestRate dcc bdc
            payCoupon issuer bondCurrency (maturityDate `beforeDays` stub)
            ↪  maturityDate
                      notional interestRate dcc bdc
            return ()
 `followedBy`
 Event Transfer
 `suchthat`
 (\t -> sender t == issuer
     && (currency.resource) t == bondCurrency
     && (money.resource) t == notional
     && date t == maturityDate)

bulletBond_NoStub_2Coupons =
  let c0 = bulletBond "Alice" "DKK" 1000 (4.2/100) (thirty360) id
           (Date 2021 01 01, Date 2021 03 01, Date 0 1 0, True, Date 0 0
           ↪  0) in
  let c1 = apply (TransferInfo ("Alice", "Bob", ("DKK", 4), Date 2021 02
  ↪  01)) c0 in
  let c2 = apply (TransferInfo ("Alice", "Bob", ("DKK", 4), Date 2021 03
  ↪  01)) c1 in
  let c3 = apply (TransferInfo ("Alice", "Bob", ("DKK", 1000), Date 2021
  ↪  03 01)) c2 in
    nullify c3

bulletBond_StartStub_2Coupons =
  let c0 = bulletBond "Alice" "DKK" 1000 (4.2/100) (thirty360) id
           (Date 2021 01 01, Date 2021 03 16, Date 0 1 0, True, Date 0 0
           ↪  0) in
  let c1 = apply (TransferInfo ("Alice", "Bob", ("DKK", 2), Date 2021 01
  ↪  16)) c0 in
  let c2 = apply (TransferInfo ("Alice", "Bob", ("DKK", 4), Date 2021 02
  ↪  16)) c1 in
  let c3 = apply (TransferInfo ("Alice", "Bob", ("DKK", 4), Date 2021 03
  ↪  16)) c2 in
  let c4 = apply (TransferInfo ("Alice", "Bob", ("DKK", 1000), Date 2021
  ↪  03 16)) c3 in
    nullify c4

bulletBond_EndStub_2Coupons =
  let c0 = bulletBond "Alice" "DKK" 1000 (4.2/100) (thirty360) id
           (Date 2021 01 01, Date 2021 03 16, Date 0 1 0, False, Date 0 0
           ↪  0) in
  let c1 = apply (TransferInfo ("Alice", "Bob", ("DKK", 4), Date 2021 02
  ↪  01)) c0 in
  let c2 = apply (TransferInfo ("Alice", "Bob", ("DKK", 4), Date 2021 03
  ↪  01)) c1 in
```

```haskell
  let c3 = apply (TransferInfo ("Alice", "Bob", ("DKK", 2), Date 2021 03 16)) c2 in
  let c4 = apply (TransferInfo ("Alice", "Bob", ("DKK", 1000), Date 2021 03 16)) c3 in
    nullify c4

main :: IO ()
main = do
  succeeds "NoStub_2Coupons" bulletBond_NoStub_2Coupons
  succeeds "StartStub_2Coupons" bulletBond_StartStub_2Coupons
  succeeds "EndStub_2Coupons" bulletBond_EndStub_2Coupons
```

11.5 FloatingBond.hs

```haskell
import Contract
import FreeTime

type InterestRateRef = String
type DayCountConvention = (Date -> Date -> Rational)
type BusinessDayConvention = (Date -> Date)

-- Ideally, this should be parametrized
roundMoney :: Rational -> Integer
roundMoney = round

getStub :: Date -> Date -> Duration -> Integer
getStub startDate maturityDate duration =
  let nextDate = startDate `after` duration in
  if nextDate <= maturityDate
  then getStub nextDate maturityDate duration
  else (dayNumberGregorian maturityDate) - (dayNumberGregorian startDate)

-- Pays a coupon based on a list of interestRates and date intervals
payCoupon :: Agent -> Currency -> Date -> Date -> Integer -> Rational
          -> DayCountConvention -> BusinessDayConvention -> [(Rational, Date, Date)]
          -> Contract Date
payCoupon issuer bondCurrency startDate endDate notional spread
          dcc bdc interestRateList = do
  -- Maximum of d and startDate
  let startDateOr = (\d -> if dayNumberGregorian d < dayNumberGregorian startDate
                          then startDate
                          else d)
  -- Minimum of d and endDate
  let endDateOr = (\d -> if dayNumberGregorian d > dayNumberGregorian endDate
                        then endDate
                        else d)
  -- Get the list of interestRates adjusted for interval size
  let factors = map (\(rate,start,end) ->
                       (rate + spread) * dcc (startDateOr start)
                        ↪  (endDateOr end))
                    interestRateList
  let rNotional = toRational notional
  let paymentAmount = foldr (\factor total -> total + rNotional*factor) 0
   ↪  factors
  (Event Transfer
   `suchthat`
   (\t -> sender t == issuer
       && (currency.resource) t == bondCurrency
       && (money.resource) t == roundMoney paymentAmount
       && date t == bdc endDate)
   `yielding`
   date)

-- Observe the interest rates
getInterestRates :: InterestRateRef -> Date -> Date
                 -> Contract [(Rational, Date, Date)]
getInterestRates interestRateRef mostRecentRateEnd deadline = do
  Event Interest
  `leadsto`
  (\(ref, val, (start, end)) ->
     if ref == interestRateRef && mostRecentRateEnd == start
     then
       if end < deadline
       then do
         interestRates <- getInterestRates interestRateRef end deadline
         return ((val, start, end):interestRates)
       else
         return [(val, start, end)]
     else
       failure)

payCoupons :: Agent -> Currency -> Date -> Date -> Duration
            -> Integer -> (Rational, Date, Date) -> InterestRateRef ->
             ↪  Rational
            -> DayCountConvention -> BusinessDayConvention
            -> Contract (Rational, Date, Date)
payCoupons issuer bondCurrency couponStartDate maturityDate frequency
            notional lastInterestRate interestRateRef spread
            dcc bdc =
  let (_,_,interestExpiry) = lastInterestRate in
  let couponPayDate = couponStartDate `after` frequency in
  if maturityDate <= couponStartDate
  then
    return lastInterestRate
  else
    if maturityDate <= interestExpiry
    then do
      payCoupon issuer bondCurrency couponStartDate couponPayDate notional
      ↪   spread
                dcc bdc [lastInterestRate]
      payCoupons issuer bondCurrency couponPayDate maturityDate frequency
                 notional lastInterestRate interestRateRef spread
                 dcc bdc
    else do
      interestRates <- getInterestRates interestRateRef interestExpiry
      ↪   couponPayDate
      payCoupon issuer bondCurrency couponStartDate couponPayDate notional
      ↪   spread
                dcc bdc (lastInterestRate:interestRates)
      payCoupons issuer bondCurrency couponPayDate maturityDate frequency
                 notional (last interestRates) interestRateRef spread
                 dcc bdc

-- The floating rate bond
floatingRateBond :: Agent -> Currency -> Integer -> InterestRateRef ->
↪   Rational
                 -> DayCountConvention -> BusinessDayConvention
                 -> (Date, Date, Duration, Bool, Duration)
                 -> Contract (Agent, Agent, Resource, Date)
floatingRateBond issuer bondCurrency notional interestRateRef spread dcc
↪   bdc
                 (startDate, maturityDate, frequency, stubAtStart,
                 ↪   stubTolerance) = do
  let stub = getStub startDate maturityDate frequency
  (Event Interest
   `suchthat`
   (\(ref, val, (start, end)) -> ref == interestRateRef
                              && start <= startDate)
   `leadsto`
   (\interestRate ->
       let (ref, val, (start, end)) = interestRate in
       if stub == 0
       then do
         payCoupons issuer bondCurrency startDate maturityDate frequency
                    notional (val, start, end) interestRateRef spread
                    dcc bdc
         return ()
       else
         if stubAtStart
         then
```

```
        let startAfterStub = startDate `afterDays` stub in
        if end < startAfterStub
        then do
          interestRates <- getInterestRates interestRateRef end
          ↪  startAfterStub
          payCoupon issuer bondCurrency startDate startAfterStub
          ↪  notional spread
                    dcc bdc ((val, start, end):interestRates)
          payCoupons issuer bondCurrency startAfterStub maturityDate
          ↪  frequency
                     notional (last interestRates) interestRateRef
                     ↪  spread
                     dcc bdc
          return ()
        else do
          payCoupon issuer bondCurrency startDate startAfterStub
          ↪  notional spread
                    dcc bdc [(val, start, end)]
          payCoupons issuer bondCurrency startAfterStub maturityDate
          ↪  frequency
                     notional (val, start, end) interestRateRef spread
                     dcc bdc
          return ()
      else do
        let lastBeforeStub = maturityDate `beforeDays` stub
        finalInterest <- payCoupons issuer bondCurrency startDate
        ↪  lastBeforeStub frequency
                                    notional (val, start, end)
                                    ↪  interestRateRef spread
                                    dcc bdc
        let (_,_,finalInterestExpiry) = finalInterest
        if finalInterestExpiry <= maturityDate
        then do
          interestRates <- getInterestRates interestRateRef
          ↪  finalInterestExpiry maturityDate
          payCoupon issuer bondCurrency lastBeforeStub maturityDate
          ↪  notional spread
                    dcc bdc (finalInterest:interestRates)
          return ()
        else do
          payCoupon issuer bondCurrency lastBeforeStub maturityDate
          ↪  notional spread
                    dcc bdc [finalInterest]
          return ())
`followedBy`
Event Transfer
`suchthat`
(\t -> sender t == issuer
        && (currency.resource) t == bondCurrency
        && (money.resource) t == notional
        && date t == maturityDate))

floatingBond_NoStub_1Rate =
  let c0 = floatingRateBond "Alice" "DKK" 1000 "1A2B3C" (5/100)
   ↪  (thirty360) id
                            (Date 2021 01 01, Date 2021 02 01, Date 0 1 0,
                             ↪  True, Date 0 0 0) in
  let c1 = apply (InterestInfo ("1A2B3C", (10/100), (Date 2021 01 01,Date
   ↪  2021 02 01))) c0 in
  let c2 = apply (TransferInfo ("Alice", "Bob",
                                 ("DKK", roundMoney (1000 * 15/100 *
                                  ↪  1/12)),
                                 Date 2021 02 01)) c1 in
  let c3 = apply (TransferInfo ("Alice", "Bob", ("DKK", 1000), Date 2021
   ↪  02 01)) c2 in
    nullify c3


floatingBond_NoStub_2Rates =
  let c0 = floatingRateBond "Alice" "DKK" 1000 "1A2B3C" (5/100)
   ↪  (thirty360) id
                            (Date 2021 01 01, Date 2021 02 01, Date 0 1 0,
                             ↪  True, Date 0 0 0) in
  let c1 = apply (InterestInfo ("1A2B3C", (10/100), (Date 2021 01 01,Date
   ↪  2021 01 21))) c0 in
  let c2 = apply (InterestInfo ("1A2B3C", (20/100), (Date 2021 01 21,Date
   ↪  2021 02 01))) c1 in
  let c3 = apply (TransferInfo ("Alice", "Bob",
                                 ("DKK", roundMoney (1000 * 15/100 * 1/12 *
                                  ↪  20/30 + 1000 * 25/100 * 1/12 *
                                  ↪  10/30)),
                                 Date 2021 02 01)) c2 in
  let c4 = apply (TransferInfo ("Alice", "Bob", ("DKK", 1000), Date 2021
   ↪  02 01)) c3 in
    nullify c4

floatingBond_NoStub_3Rates2Coupons =
  let c0 = floatingRateBond "Alice" "DKK" 1000 "1A2B3C" (5/100)
   ↪  (thirty360) id
                            (Date 2021 01 01, Date 2021 03 01, Date 0 1 0,
                             ↪  True, Date 0 0 0) in
  let c1 = apply (InterestInfo ("1A2B3C", (10/100), (Date 2021 01 01,Date
   ↪  2021 01 21))) c0 in
  let c2 = apply (InterestInfo ("1A2B3C", (20/100), (Date 2021 01 21,Date
   ↪  2021 02 11))) c1 in
  let c3 = apply (TransferInfo ("Alice", "Bob",
                                ("DKK", roundMoney (1000 * 15/100 * 1/12 *
                                ↪  20/30 + 1000 * 25/100 * 1/12 *
                                ↪  10/30)),
                                 Date 2021 02 01)) c2 in
  let c4 = apply (InterestInfo ("1A2B3C", (30/100), (Date 2021 02 11,Date
  ↪  2021 03 01))) c3 in
  let c5 = apply (TransferInfo ("Alice", "Bob",
                                ("DKK", roundMoney (1000 * 25/100 * 1/12 *
                                ↪  10/30 + 1000 * 35/100 * 1/12 *
                                ↪  20/30)),
                                 Date 2021 03 01)) c4 in
  let c6 = apply (TransferInfo ("Alice", "Bob", ("DKK", 1000), Date 2021
  ↪  03 01)) c5 in
    nullify c6

main :: IO ()
main = do
  succeeds "NoStub_1Rate" floatingBond_NoStub_1Rate
  succeeds "NoStub_2Rates" floatingBond_NoStub_2Rates
  succeeds "NoStub_3Rates2Coupons" floatingBond_NoStub_3Rates2Coupons
```

11.6 TurboWarrant.hs

```
import FreeTime
import Contract

-- Ideally, this should be parametrized
roundMoney :: Rational -> Integer
roundMoney = round

-- Pay an amount
payment :: (Agent, Date, Rational, Currency) -> Contract ()
payment (payeeAgent, deadlineDate, cashAmount, currencyType) =
  if cashAmount > 0
  then Event Transfer
       `suchthat`
       (\e -> receiver e == payeeAgent
           && date e == deadlineDate
           && money (resource e) == roundMoney cashAmount
           && currency (resource e) == currencyType)
       `yielding`
       (\e -> ())
  else return ()

-- Get either a confirmation or data correction, then make a payment
paymentWithConfirmation :: (Agent, Date, Rational, Currency) -> Contract
↪  ()
```

```haskell
paymentWithConfirmation (payeeAgent, deadlineDate, cashAmount,
 ↪  currencyType) =
  (Event PaymentConfirmation
   `suchthat`
   (\e -> payee e == payeeAgent
       && date e == deadlineDate
       && amount e == roundMoney cashAmount
       && getCurrency e == currencyType)
   `leadsto`
   (\e -> payment (payeeAgent, deadlineDate, cashAmount, currencyType)))
  `orelse`
  (Event PaymentCorrection
   `suchthat`
   (\e -> payee e == payeeAgent)
   `leadsto`
   (\e -> payment (payeeAgent, date e, (toRational.amount) e, getCurrency
    ↪  e)))

-- Events that lead to termination
terminateAndReplace :: Contract (String, String)
terminateAndReplace =
  Event Disruption
  `orelse`
  Event EconomicConditions
  `orelse`
  Event LegalReason

-- Contract wrapper for terminate/replace
withTerminateAndReplace :: Contract a -> Contract (Either (String, String)
 ↪  a)
withTerminateAndReplace contract =
  (terminateAndReplace `yielding` Left)
   `orelse`
  (contract `yielding` Right)

-- Observe an underlying value
observe :: (ObservationType, Date, PriceSelection) -> Contract Rational
observe (oType, oDate, oPrice) =
  Event Observation
  `suchthat`
  (\o -> obsType o == oType
      && obsDate o == oDate
      && obsPrice o == oPrice)
  `yielding`
  obsValue

-- Get the list of all underlying values at specific dates
collectObservations :: [(ObservationType, Date, PriceSelection)]
```

```haskell
  -> Contract (Either (String, String)
                        [((ObservationType, Date,
                         ↪  PriceSelection),Rational)])
collectObservations observationSpecs =
  case observationSpecs of
    [] -> NoEvent (Right [])
    o:os -> do
      obsResult <- withTerminateAndReplace (observe o)
      obsMap <- collectObservations os
      return $ (do val <- obsResult
                   obsMap' <- obsMap
                   return ((o,val):obsMap'))

data BarrierSpec = BarrierSpec
    { observationsRequired :: [(ObservationType, Date, PriceSelection)]
    , barrierStatus :: [((ObservationType, Date, PriceSelection),
     ↪  Rational)] -> Bool
    }

-- Check if the barrier was hit based on observations
knockOutCheck :: (Agent, Currency, Date, BarrierSpec)
              -> Contract (Either (String, String) Bool)
knockOutCheck ( custodian
              , paymentCurrency
              , settlementDate
              , BarrierSpec observations statusFunc) = do
  obsResult <- collectObservations observations
  return (do obsMap <- obsResult;
             return (statusFunc obsMap))

data PaymentSpec = PaymentSpec
    { paymentObservations :: [(ObservationType, Date, PriceSelection)]
    , calculatePaymentAmount :: [((ObservationType, Date, PriceSelection),
     ↪  Rational)] -> Rational
    }

-- Pay the final redemption
finalRedemption :: (Agent, Currency, Date, PaymentSpec)
                -> Contract (Either (String, String) ())
finalRedemption ( payeeAgent
                , settlementCurrency
                , settlementDate
                , PaymentSpec observations paymentFunc) = do
  obsMap <- collectObservations observations
  let paymentArgs = (do obsMap' <- obsMap
                        let settlementAmount = paymentFunc obsMap'
                        return (payeeAgent,
                                settlementDate,
                                settlementAmount,
                                settlementCurrency))
  case paymentArgs of
    Left err ->
      return $ Left err
    Right args -> do
      paymentWithConfirmation args
      return $ Right ()

adjustForCurrencyPaymentSpec :: (PaymentSpec, Currency, Currency, Date) ->
 ↪  PaymentSpec
adjustForCurrencyPaymentSpec (baseSpec, referenceCurrency,
 ↪  settlementCurrency, fxValuationDate) =
  if referenceCurrency == settlementCurrency
  then baseSpec
  else let fxObs = ( FXRate referenceCurrency settlementCurrency
                   , fxValuationDate
                   , Closing) in
       PaymentSpec (fxObs:(paymentObservations baseSpec))
                   (\obsMap ->
                      let Just fxRate = lookup fxObs obsMap in
                      let basePayment = (calculatePaymentAmount baseSpec)
                       ↪  obsMap in
                        fxRate * (toRational basePayment))

mkWarrantPaymentSpec :: TurboWarrantData -> PaymentSpec
mkWarrantPaymentSpec twData =
  let valuationDate = (optDateExpiry . expiryAndSettlement) twData in
  let settlementPriceObs = ( Underlying (warrantUnderlying twData)
                           , valuationDate
                           , Closing ) in
  let cRatio = (toRational . conversionRatio) twData in
  let units = (toRational . warrantUnits) twData in
  let paymentCalculator =
        (\obsMap ->
           let Just settlementPrice = lookup settlementPriceObs obsMap in
           let intrinsic = (settlementPrice -
            ↪  (toRational.strikeAndBarrier) twData)
                           * units / cRatio in
             case warrantType twData of
               DownAndOutCall -> max intrinsic 0.0
               UpAndOutPut -> max (-intrinsic) 0.0) in
  let basePaymentSpec = PaymentSpec [settlementPriceObs] paymentCalculator
  ↪  in
    adjustForCurrencyPaymentSpec ( basePaymentSpec
                                 , warrantRefC twData
                                 , warrantSetC twData
                                 , valuationDate)
```

```haskell

mkKnockOutBarrierSpecs :: TurboWarrantData -> (Date -> BarrierSpec)
mkKnockOutBarrierSpecs twData =
  \valuationDate ->
    let priceObs = ( Underlying (warrantUnderlying twData)
                   , valuationDate
                   , case warrantType twData of
                       DownAndOutCall -> Minimum
                       UpAndOutPut -> Maximum) in
    let statusFunc =
          case warrantType twData of
            DownAndOutCall ->
              (\obsEnv -> let Just price = lookup priceObs obsEnv in
                             price <= (toRational.strikeAndBarrier) twData)
            UpAndOutPut ->
              (\obsEnv -> let Just price = lookup priceObs obsEnv in
                             price >= (toRational.strikeAndBarrier) twData)
                              ↪  in
      BarrierSpec [priceObs] statusFunc


-- Keep doing knockout checks until there are no dates left to check
-- then pay the maturity
turboWarrantRec :: ([KnockOutCheckDates], (Date -> BarrierSpec),
 ↪  PaymentSpec, TurboWarrantData)
                -> Contract (Either (String, String) ())
turboWarrantRec ([], barrierSpecFunc, paymentSpec, twData) =
  finalRedemption ( warrantAgent twData
                  , warrantSetC twData
                  , (optDateSettlement . expiryAndSettlement) twData
                  , paymentSpec)
turboWarrantRec (d:ds, barrierSpecFunc, paymentSpec, twData) = do
  isKnockedOut <- knockOutCheck ( warrantAgent twData
                                , warrantSetC twData
                                , knockOutSettlement d
                                , barrierSpecFunc (knockOutValuation d))
  case isKnockedOut of
    Left err ->
      return $ Left err
    Right isKnockedOut' -> do
      if isKnockedOut'
        then do
          payment paymentArgs
          return $ Right ()
        else
          turboWarrantRec (ds, barrierSpecFunc, paymentSpec, twData)
  where
    paymentArgs = ( warrantAgent twData
                      , knockOutSettlement d
                      , toRational ((earlyRedemptionAmount twData)
                                     * (warrantUnits twData))
                      , warrantSetC twData)

-- The turbo warrant
turboWarrant :: Contract (Either (String, String) ())
turboWarrant =
  Event TurboWarrant
  `leadsto`
  (\twData ->
     let calculateFinalPayment = mkWarrantPaymentSpec twData in
     let knockOutBarrierSpecs = mkKnockOutBarrierSpecs twData in
     turboWarrantRec ( knockOutCheckDates twData
                     , knockOutBarrierSpecs
                     , calculateFinalPayment
                     , twData))


twData = (TurboWarrantData
           "Alice"
           (Index "DAX")
           10
           "EUR"
           "EUR"
           1000
           1
           1
           DownAndOutCall
           []
           (OptDates (Date 2021 01 01) (Date 2021 01 02)))

turboWarrantNoObservations =
  let c1 = apply (TurboWarrantInfo twData) turboWarrant in
  let c2 = apply (ObservationInfo (Underlying (Index "DAX"),
                                   Date 2021 01 01,
                                   Closing,
                                   1050)) c1 in
  let c3 = apply (PaymentConfirmationInfo ("Alice",
                                           "EUR",
                                           500,
                                           Date 2021 01 02)) c2 in
  let c4 = apply (TransferInfo ("Bob",
                                "Alice",
                                ("EUR", 500),
                                Date 2021 01 02)) c3 in
    nullify c4

```

```haskell
turboWarrantNoKnockOut =
  let c1 = apply (TurboWarrantInfo
                   (twData {knockOutCheckDates =
                            [(KnockOutCheckDates (Date 2020 01 01) (Date 2020 01 02))]}))
           turboWarrant in
  let c2 = apply (ObservationInfo (Underlying (Index "DAX"),
                                   Date 2020 01 01,
                                   Minimum,
                                   1050)) c1 in
  let c3 = apply (ObservationInfo (Underlying (Index "DAX"),
                                   Date 2021 01 01,
                                   Closing,
                                   1050)) c2 in
  let c4 = apply (PaymentConfirmationInfo ("Alice",
                                           "EUR",
                                           500,
                                           Date 2021 01 02)) c3 in
  let c5 = apply (TransferInfo ("Bob",
                                "Alice",
                                ("EUR", 500),
                                Date 2021 01 02)) c4 in
    nullify c5

turboWarrantKnockOut =
  let c1 = apply (TurboWarrantInfo
                   (twData {knockOutCheckDates =
                            [(KnockOutCheckDates (Date 2020 01 01) (Date 2020 01 02))]}))
           turboWarrant in
  let c2 = apply (ObservationInfo (Underlying (Index "DAX"), Date 2020 01 01, Minimum, 900)) c1 in
  let c3 = apply (TransferInfo ("Bob", "Alice", ("EUR", 10), Date 2020 01 02)) c2 in
    nullify c3

main :: IO ()
main = do
  succeeds "NoObservations" turboWarrantNoObservations
  succeeds "NoKnockOut" turboWarrantNoKnockOut
  succeeds "KnockOut" turboWarrantKnockOut
```

11.7 BarrierReverse.hs

```haskell
import Contract
import FreeTime

```

```haskell
data Scheduling =
    Scheduling1 (Date, Date, Duration)

type InterestRateRef = String
type DayCountConvention = (Date -> Date -> Rational)
type BusinessDayConvention = (Date -> Date)

-- Ideally, this should be parametrized
roundMoney :: Rational -> Integer
roundMoney = round

type ObsData = (ObservationType, Date, PriceSelection, Rational)

data UnderlyingRef =
    SingleUnderlying Underlying
  | MultiUnderlying [Underlying] ([ObsData] -> Rational)


-- Pays a single coupon at the specified date with the specified rate
payCoupon :: Agent -> Currency -> Date -> Integer -> Rational
          -> Contract (Agent, Agent, Resource, Date)
payCoupon issuer paymentCurrency paymentDate notional couponRate =
  let paymentAmount = (toRational notional) * couponRate in
    Event Transfer
    `suchthat`
    (\t -> sender t == issuer
        && (currency.resource) t == paymentCurrency
        && (money.resource) t == roundMoney paymentAmount
        && date t == paymentDate)

-- Pays coupons at the specified dates
payCoupons :: Agent -> Currency -> [Date] -> Integer -> Rational
           -> Contract ()
payCoupons issuer currency couponDates notional couponRate =
  case couponDates of
    [] ->
      return ()
    d:ds -> do
      payCoupon issuer currency d notional couponRate
      payCoupons issuer currency ds notional couponRate

getUnderlyingValue :: UnderlyingRef -> Date -> Contract Rational
-- Get the value of a single underlying
getUnderlyingValue (SingleUnderlying ref) date =
  Event Observation
  `suchthat`
  (\o -> obsType o == Underlying ref
      && obsDate o == date)
  `yielding`
  obsValue
-- Get the value of multiple underlyings using parallel contracts
getUnderlyingValue (MultiUnderlying (ref:refs) processFunc) date = do
  let firstObs =
        Event Observation
        `suchthat`
        (\o -> obsType o == Underlying ref
            && obsDate o == date)
        `yielding`
        (\o -> [o])
  let makeParallel =
        (\r contract ->
            (Event Observation
             `suchthat`
             (\o -> obsType o == Underlying r
                 && obsDate o == date)
             `andalso`
             contract)
            `yielding`
            (\(o,os) -> o:os))
  foldr makeParallel firstObs refs
  `yielding`
  processFunc

-- Check if the AES level was hit; if it was, make the early payout
checkAES :: Agent -> Currency -> Integer -> UnderlyingRef
         -> Rational -> [(Date,Date)] -> Rational
         -> Contract Bool
checkAES issuer setCurrency notional underlying
         strikePrice aesCheckAndPayDates aesLevel =
  case aesCheckAndPayDates of
    [] ->
      return False
    (checkDate,payDate):ds -> do
      underlyingValue <- getUnderlyingValue underlying checkDate
      if underlyingValue / (toRational strikePrice) >= aesLevel
      then
        Event Transfer
        `suchthat`
        (\t -> sender t == issuer
            && (currency.resource) t == setCurrency
            && (money.resource) t == notional
            && date t == payDate)
        `yielding`
        (\_ -> True)
      else
        checkAES issuer setCurrency notional underlying strikePrice ds
         ↪  aesLevel
```

```

-- Get the strike price based on the underlying value(s)
getStrikePrice' :: UnderlyingRef -> [Date] -> Contract Rational
getStrikePrice' underlyingRef [] =
  return 0
getStrikePrice' underlyingRef (d:ds) = do
  underlyingValue <- getUnderlyingValue underlyingRef d
  underlyingValues <- getStrikePrice' underlyingRef ds
  return (toRational underlyingValue + underlyingValues)

-- Get the strike price over all the dates, then find the average
getStrikePrice :: UnderlyingRef -> [Date] -> Contract Rational
getStrikePrice underlyingRef strikeDates = do
  getStrikePrice' underlyingRef strikeDates
  `yielding`
  (\p -> p / (toRational.length) strikeDates)

type CurrencyConverter = Currency -> Currency -> Rational -> Rational

-- Pay the knock-in amount or just the notional at the maturity
payMaturity :: Agent -> Currency -> Currency -> Integer -> Rational
            -> UnderlyingRef -> CurrencyConverter -> SettlementDates
            -> (Rational, Rational, Rational, Rational, Rational)
            -> Contract (Agent, Agent, Resource, Date)
payMaturity issuer setCurrency refCurrency notional strikePrice
            underlying cc settlementDates
            (knockInLevel, constPercent, strikePercent, gearing,
             ↪  floorPercent) = do
  let ccRefSet = cc refCurrency setCurrency
  underlyingValue <- getUnderlyingValue underlying (settlementValuation
   ↪  settlementDates)
  if underlyingValue / (toRational strikePrice) < toRational knockInLevel
  then
    let putOption = ccRefSet ((underlyingValue / toRational strikePrice) -
     ↪  strikePercent) in
    let payout = (toRational notional) * (max (constPercent + gearing *
     ↪  putOption) floorPercent) in
      Event Transfer
      `suchthat`
      (\t -> sender t == issuer
          && (currency.resource) t == setCurrency
          && (money.resource) t == roundMoney payout
          && date t == settlementMaturity settlementDates)
  else
    Event Transfer
    `suchthat`
    (\t -> sender t == issuer
        && (currency.resource) t == setCurrency
        && (money.resource) t == notional
        && date t == settlementMaturity settlementDates)

-- Pay coupons while checking for AES barrier hits, end early if there is
↪  a hit
payCouponsWithAES :: Agent -> Currency -> Integer -> Rational ->
↪  UnderlyingRef
                  -> [Date] -> [(Date,Date)] -> Rational -> Rational
                  -> Contract Bool
payCouponsWithAES issuer setCurrency notional couponRate underlying
                  [] aesCheckAndPayDates aesLevel strikePrice = do
  checkAES issuer setCurrency notional underlying
           strikePrice aesCheckAndPayDates aesLevel
payCouponsWithAES issuer setCurrency notional couponRate underlying
                  couponDates [] aesLevel strikePrice = do
  payCoupons issuer setCurrency couponDates notional couponRate
  return False
payCouponsWithAES issuer setCurrency notional couponRate underlying
                  couponDates aesCheckAndPayDates aesLevel strikePrice =
                  ↪  do
  let nextCouponDate:couponDates' = couponDates
  let (aesCheckDate,aesPayDate):aesCheckAndPayDates' = aesCheckAndPayDates
  if nextCouponDate < aesCheckDate
  then do
    payCoupon issuer setCurrency nextCouponDate notional couponRate
    payCouponsWithAES issuer setCurrency notional couponRate underlying
                      couponDates' aesCheckAndPayDates aesLevel
                      ↪  strikePrice
  else do
    aesBarrierHit <- checkAES issuer setCurrency notional underlying
                              strikePrice [(aesCheckDate,aesPayDate)]
                              ↪  aesLevel
    if aesBarrierHit
    then return True
    else payCouponsWithAES issuer setCurrency notional couponRate
    ↪  underlying
                           couponDates aesCheckAndPayDates' aesLevel
                           ↪  strikePrice

-- The barrier reverse convertible
barrierReverse :: Agent -> Currency -> Currency -> UnderlyingRef ->
↪  Integer
               -> [(Date,Date)] -> [Date] -> [Date] -> SettlementDates
               -> Rational -> Rational -> CurrencyConverter
               -> (Rational, Rational, Rational, Rational, Rational)
               -> Contract ()
barrierReverse issuer setCurrency refCurrency underlying notional
               aesCheckAndPayDates couponDates strikeDates settlementDates
```

```
              couponRate aesLevel cc
              (knockInLevel, constPercent, strikePercent, gearing,
               ↪  floorPercent) = do
  strikePrice <- getStrikePrice underlying strikeDates
  aesActivated <- payCouponsWithAES issuer setCurrency notional couponRate
   ↪  underlying
                                      couponDates aesCheckAndPayDates
                                       ↪  aesLevel strikePrice
  if aesActivated
  then return ()
  else do
       payMaturity issuer setCurrency refCurrency notional strikePrice
                   underlying cc settlementDates
                   (knockInLevel, constPercent, strikePercent, gearing,
                    ↪  floorPercent)
       return ()

barrierReverse_EarlySettlement =
  let c0 = barrierReverse "Bob" "DKK" "DKK" (SingleUnderlying (Index
   ↪  "1A2B3C")) 1000
                 [((Date 2021 01 20), (Date 2021 01 25)), ((Date 2021 02
                  ↪  20),(Date 2021 02 25))]
                 [(Date 2021 02 01), (Date 2021 03 01)]
                 [(Date 2021 01 05)]
                 (SettlementDates (Date 2021 03 05) (Date 2021 03 10))
                 (10/100) 1 (\_ _ x -> x)
                 (55/100, 1, 1, -1, 0) in
  let c1 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                  Date 2021 01 05,
                                  AtTimePoint (Date 2021 01 05),
                                  500)) c0 in
  let c2 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                  Date 2021 01 20,
                                  AtTimePoint (Date 2021 01 20),
                                  475)) c1 in
  let c3 = apply (TransferInfo ("Bob", "Alice", ("DKK", 100), Date 2021 02
   ↪  01)) c2 in
  let c4 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                  Date 2021 02 20,
                                  AtTimePoint (Date 2021 02 20),
                                  525)) c3 in
  let c5 = apply (TransferInfo ("Bob", "Alice", ("DKK", 1000), Date 2021
   ↪  02 25)) c4 in
    nullify c5

barrierReverse_NoKnockIn =
  let c0 = barrierReverse "Bob" "DKK" "DKK" (SingleUnderlying (Index
   ↪  "1A2B3C")) 1000
```

```
                 [((Date 2021 01 20), (Date 2021 01 25)), ((Date 2021 02
                 ↪  20),(Date 2021 02 25))]
                 [(Date 2021 02 01), (Date 2021 03 01)]
                 [(Date 2021 01 05)]
                 (SettlementDates (Date 2021 03 05) (Date 2021 03 10))
                 (10/100) 1 (\_ _ x -> x)
                 (55/100, 1, 1, -1, 0) in
  let c1 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                   Date 2021 01 05,
                                   AtTimePoint (Date 2021 01 05),
                                   500)) c0 in
  let c2 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                   Date 2021 01 20,
                                   AtTimePoint (Date 2021 01 20),
                                   475)) c1 in
  let c3 = apply (TransferInfo ("Bob", "Alice", ("DKK", 100), Date 2021 02
  ↪  01)) c2 in
  let c4 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                   Date 2021 02 20,
                                   AtTimePoint (Date 2021 02 20),
                                   450)) c3 in
  let c5 = apply (TransferInfo ("Bob", "Alice", ("DKK", 100), Date 2021 03
  ↪  01)) c4 in
  let c6 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                   Date 2021 03 05,
                                   AtTimePoint (Date 2021 03 05),
                                   425)) c5 in
  let c7 = apply (TransferInfo ("Bob", "Alice",
                                ("DKK", 1000),
                               Date 2021 03 10)) c6 in
    nullify c7

barrierReverse_KnockIn =
  let c0 = barrierReverse "Bob" "DKK" "DKK" (SingleUnderlying (Index
  ↪  "1A2B3C")) 1000
                 [((Date 2021 01 20), (Date 2021 01 25)), ((Date 2021 02
                 ↪  20),(Date 2021 02 25))]
                 [(Date 2021 02 01), (Date 2021 03 01)]
                 [(Date 2021 01 05)]
                 (SettlementDates (Date 2021 03 05) (Date 2021 03 10))
                 (10/100) 1 (\_ _ x -> x)
                 (55/100, 1, 1, -1, 0) in
  let c1 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                   Date 2021 01 05,
                                   AtTimePoint (Date 2021 01 05),
                                   500)) c0 in
  let c2 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                   Date 2021 01 20,
                                           AtTimePoint (Date 2021 01 20),
                                           400)) c1 in
  let c3 = apply (TransferInfo ("Bob", "Alice", ("DKK", 100), Date 2021 02
  ↪  01)) c2 in
  let c4 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                           Date 2021 02 20,
                                           AtTimePoint (Date 2021 02 20),
                                           300)) c3 in
  let c5 = apply (TransferInfo ("Bob", "Alice", ("DKK", 100), Date 2021 03
  ↪  01)) c4 in
  let c6 = apply (ObservationInfo (Underlying (Index "1A2B3C"),
                                           Date 2021 03 05,
                                           AtTimePoint (Date 2021 03 05),
                                           200)) c5 in
  let c7 = apply (TransferInfo ("Bob", "Alice",
                                       ("DKK", 1600),
                                       Date 2021 03 10)) c6 in
    nullify c7

main = do
  succeeds "EarlySettlement" barrierReverse_EarlySettlement
  succeeds "NoKnockIn" barrierReverse_NoKnockIn
  succeeds "KnockIn" barrierReverse_KnockIn
```

12 Copyright attribution

The front page image is “Business Contract” by http://informedmag.com/. It is licensed under the CC BY 2.0 license. A copy of the license can be found at https://creativecommons.org/licenses/by/2.0/