Certified Symbolic Management of Financial Multi-Party Contracts *

Patrick Bahr  Jost Berthold  Martin Elsman
Department of Computer Science, University of Copenhagen (DIKU)
{paba,berthold,mael}@di.ku.dk

Abstract
Domain-specific languages (DSLs) for complex financial contracts are in practical use in many banks and financial institutions today. Given the level of automation and pervasiveness of software in the sector, the financial domain is immensely sensitive to software bugs. At the same time, there is an increasing need to analyse (and report on) the interaction between multiple parties. In this paper, we present a multi-party contract language that rigorously relates any artefacts of simulation and computation from its core, which leads to favourable algebraic properties, and therefore allows for formalising domain-specific analyses and transformations using a proof assistant. At the centre of our formalisation is a simple denotational semantics independent of any stochastic aspects. Based on this semantics, we devise certified contract analyses and transformations. In particular, we give a type system, with an accompanying type inference procedure, that statically ensures that contracts follow the principle of causality. Moreover, we devise a reduction semantics that allows us to evolve contracts over time, in accordance with the denotational semantics. From the verified Coq definitions, we automatically extract a Haskell implementation of an embedded contract DSL along with the formally verified contract management functionality. This approach opens a road-map towards more reliable contract management software, including the possibility of analysing contracts based on symbolic instead of numeric methods.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory; D.2.4 [Software Engineering]: Software/Program Verification—Correctness proofs; F.3.2 [Semantics of Programming Languages]: Miscellaneous

General Terms Languages, Verification

Keywords Domain-Specific Language, Financial Contracts, Coq, Haskell, Certified Code

1. Introduction
The modern financial industry is characterised by a large degree of automation and pervasive use of software for many purposes, spanning from day-to-day accounting and management to valuation of financial derivatives, and even automated high-frequency trading. To meet the demand for quick time to market, with respect to supporting new financial products, many banks and financial institutions today use domain-specific languages (DSLs) to describe complex financial contracts, in particular, for specifying how asset transfers for a specific contract depend on underlying observables, such as interest rates, currency and stock prices.

Based on this semantics, we devise certified contract analyses and transformations. In particular, we give a type system, with an accompanying type inference procedure, that statically ensures that contracts follow the principle of causality. Moreover, we devise a reduction semantics that allows us to evolve contracts over time, in accordance with the denotational semantics. From the verified Coq definitions, we automatically extract a Haskell implementation of an embedded contract DSL along with the formally verified contract management functionality. This approach opens a road-map towards more reliable contract management software, including the possibility of analysing contracts based on symbolic instead of numeric methods.

Keywords Domain-Specific Language, Financial Contracts, Coq, Haskell, Certified Code

1. Introduction
The modern financial industry is characterised by a large degree of automation and pervasive use of software for many purposes, spanning from day-to-day accounting and management to valuation of financial derivatives, and even automated high-frequency trading. To meet the demand for quick time to market, with respect to supporting new financial products, many banks and financial institutions today use domain-specific languages (DSLs) to describe complex financial contracts, in particular, for specifying how asset transfers for a specific contract depend on underlying observables, such as interest rates, currency and stock prices.

Based on this semantics, we devise certified contract analyses and transformations. In particular, we give a type system, with an accompanying type inference procedure, that statically ensures that contracts follow the principle of causality. Moreover, we devise a reduction semantics that allows us to evolve contracts over time, in accordance with the denotational semantics. From the verified Coq definitions, we automatically extract a Haskell implementation of an embedded contract DSL along with the formally verified contract management functionality. This approach opens a road-map towards more reliable contract management software, including the possibility of analysing contracts based on symbolic instead of numeric methods.


defined cost observables. The success of this work has been partially supported by the Danish Council for Strategic Research under contract number 10-092299 (HIPERFIT [4]), and the Danish Council for Independent Research under Project 12-132365.

The seminal work by Peyton-Jones, Eber, and Seward on bilateral financial contracts [22] shows how an algebraic approach to contract specification can be used for valuation of contracts (when combined with a model of the underlying observables) and introduces a contract management model where contracts gradually evolve into the empty contract (for which the two parties have no further obligations) when knowledge of underlying observables becomes available and decisions are taken.

In contrast to most previous work on financial contract languages, where contracts are viewed as bilateral agreements held by one of the involved parties, we consider a generalised view of contracts where a contract specifies the obligations and rights of all parties involved in the contract. This generalisation requires the contract writer to be explicit about parties involved in transferring rights and assets. The additional dimension of flexibility allows, for instance, for tools to analyse the overall effect to a contract (or portfolio) of parties defaulting or merging. For valuation purposes, and for other analyses, we can, at any point in time, view a contract (or a portfolio) from the point of view of a particular party, and thereby obtain the classical bilateral contract view. The additional flexibility that this generalisation provides is mandatory for certain kinds of risk analyses, demanded by regulatory requirements for certain financial institutions, such as the daily calculation of Credit Value Adjustments (CVA) [10].

In view of the pervasive automation in the financial world, conceptual as well as accidental software bugs can have catastrophic consequences. Financial companies need to trust their software systems for contract management. For systems where contracts themselves are written independently from the underlying contract management software stack, trust needs to be mitigated at different levels. First, there is the question whether a particular contract behaves according to the contract writer’s intend, and in particular, whether the contract can be executed according to the underlying execution model. In this paper, we address this issue by providing a type system for the contract language, which guarantees that contracts can indeed be executed. In particular, the system guarantees causality of contracts, which means that a transfer of an asset today cannot depend on a decision or the value of an underlying observable tomorrow. Similarly, we demonstrate that essential contract properties can be derived from symbolic contract specifications alone, and that contract management can be described as symbolic goal-directed manipulation of the contract, avoiding any stochastic aspects, which are often added to contract languages for valuation purposes.

Second, one may ask whether the implementation of the underlying contract management framework, and the accompanying con-

The ideas have emerged into the successful company LexiFi, which has become a leading software provider for a range of financial institutions. LexiFi is a partner of the HIPERFIT Research Center [3], hosted at DIKU.
contract analyses, behave correctly over time, not only for the common scenarios, but also in all corner cases and for all possible compositions of contract components. To address this issue, we have based the symbolic contract management operations and the associated contract analyses on a precise cash-flow semantics for contracts, which we have modelled and checked using the Coq proof assistant. Using the code extraction functionality of Coq, the certified contract analyses and transformations are extracted into a Haskell module, which serves as the certified core of a financial contract management library. The two approaches work hand-in-hand and provide a highly desirable and highly trustworthy code base.

In summary, the contributions of this paper are the following:

- We present an expressive multi-party contract DSL (section 2.1) and demonstrate that the contract language allows for expressing real-world contracts and portfolios, such as foreign exchange swaps and options, credit default swaps, and portfolios holding contracts with multiple counter-parties.

- We give the contract language a denotational semantics based on cash-flows (section 4.1), which allows us to precisely and succinctly characterise contract properties (e.g. causality) and transformations (e.g. partial evaluation).

- We devise a type-system that statically ensures that contracts follow the principle of causality, together with an accompanying type inference procedure (section 3.2).

- We derive a reduction semantics for the contract language, which evolves contracts over time in accordance with the denotational semantics (section 4.2). As we will show, our type system is a crucial ingredient for establishing computational adequacy of the reduction semantics.

- We formally verify the correctness of our contract management functionality including type inference, reduction semantics, contract specialisation (partial evaluation), and horizon inference.

- Using the code extraction functionality of the Coq system, we generate an implementation of the certified analyses and transformations in Haskell.

Implementations of the contract language in Haskell and Coq are available online[1] together with machine-checkable proofs of all properties and theorems mentioned in this paper. Currently, the contract framework is being deployed in a contract and portfolio pricing and risk calculation prototype [20], developed at the HIPERFIT Research Center.

2. The Contract Language

A financial contract is an agreement between several parties that stipulates future asset transfers (i.e. cash-flows) between those parties. Amounts in contracts may be scaled using real-valued expressions, which may refer to observable underlying values. Contracts may depend on a variety of different observable values. Common examples include foreign exchange rates, stock prices, and market indexes, but contracts sometimes depend on other observables that are truly outside the financial world, such as the temperature or the amount of rain in an area. Contracts can contain alternatives depending on Boolean predicates, which may refer to these observables, as well as decisions taken by the parties involved in the contract.

Observables and choices in our contract language are “observed” with a given offset (in discrete time units) from the current time. In general, all definitions use this concept of relative time, aiding the compositionality of contracts. Contracts may be easily translated into the future by a positive offset, without having to adjust the time of observing an underlying value.

2.1 Examples and Language Constructs

We shall illustrate contracts using examples from the foreign exchange (FX) market and days as the time unit, but the concepts generalise easily to other settings. Likewise, cash-flows are based on a fixed set of currencies, but could instead allow for arbitrary assets.

At first, we consider the following forward contract, an agreement to purchase an asset in the future for a fixed price.

Example 1 (FX Forward). In 90 days, party X will buy 100 US dollars for a fixed rate 6.5 of Danish kroner from party Y.

\[90 \uparrow 100 \times (\text{USD}(Y \to X) \& 6.5 \times \text{DKK}(X \to Y))\]

The contract \(\text{USD}(Y \to X)\) stipulates that party Y must transfer one unit of USD to party X now. Similarly, \(\text{DKK}(X \to Y)\) stipulates that party X must transfer one unit of DKK to party Y. The combinator \(\&\) allows us to scale a contract by a real-valued expression. In the example, we use it with the constants 6.5 and 100. The combinator \(\&\) combines two contracts conjunctively. Finally, the combinator \(\uparrow\) translates a contract into the future. In the above example, we translate the whole trade of 100 US Dollars for Danish kroner 90 days into the future.

A common contract structure is to repeat a choice between alternatives until a given end date. Our language supports this repetitive check directly using the conditional

\[\text{if } \ldots \text{ within } \ldots \text{ then } \ldots \text{ else } \ldots\]

an iterating generalisation of a simple alternative (if-then-else). As an example, consider a so-called American option, where one party may, at any time before the contract ends, decide to execute the purchase.

Example 2 (FX American Option). Party X may, within 90 days, decide whether to (immediately) buy 100 US dollars for a fixed rate 6.5 of Danish kroner from party Y.

\[
\begin{align*}
\text{if } \text{obs}(X \text{ exercises option}, 0) \text{ within } 90 \\
\text{then } 100 \times (\text{USD}(Y \to X) \& 6.5 \times \text{DKK}(X \to Y)) \\
\text{else } \emptyset
\end{align*}
\]

This contract uses an observable external decision, expressed using \(\text{obs}\) (which uses a time offset, 0 meaning the current day), and the \(\text{if-within}\) construct, which monitors this decision of party X over the 90 days time window. If X chooses to exercise the option before the end of the 90 days time window, the trade comes into effect, consisting of two transfers (combined with \(\&\)) between the parties. Otherwise, the contract becomes empty (\(\emptyset\)) after 90 days. The expression language also features an accumulator combinator \(\text{ace}\), which accumulates a value over a given number of days from the past until the current day. The accumulator can be used to describe so-called Asian options (or average options), for which a price is established from an average of past prices instead of just one observed price.

Example 3 (FX Asian Option). After 90 days, party X may decide to buy USD 100; paying the average of the exchange rate USD to DKK observed over the last 30 days.

\[
\begin{align*}
90 \uparrow \text{if } \text{obs}(X \text{ exercises option}, 0) \text{ within } 0 \\
\text{then } 100 \times (\text{USD}(Y \to X) \& (\text{rate} \times \text{DKK}(X \to Y))) \\
\text{else } \emptyset
\end{align*}
\]

where \(\text{rate} = \text{ace}(\lambda x. r + \text{obs}(\text{FX}(\text{USD}, \text{DKK}), 0), 30, 0)/30\)

---

[2] Examples have been provided by partners of the HIPERFIT research center.

[1] See [https://github.com/HIPERFIT/contracts](https://github.com/HIPERFIT/contracts)
observable, this contract uses an observable expression to observe the exchange rate between USD and DKK (again at offset 0, thus on the current day). Observed values are accumulated at the rate used in an acc expression. The rate is determined as the average of the USD to DKK exchange rates observed over the 30 days before the day when the scaled payment is made (acc has a backwards-stepping semantics with respect to time). More generally, the acc construct can be used to propagate a state through a value computation.

So far, all contracts had only two parties. To illustrate the multiparty aspect of our language, we consider a simple credit default swap (CDS) for a zero-coupon bond, which involves three parties.

Example 4 (CDS for a zero-coupon bond). The issuer $X$ of a zero-coupon bond agrees to pay the holder $Y$ a nominal amount, say DKK 900, in the event that the issuer $X$ of the underlying bond defaults. To this end, we use an observable "X defaults":

$$\text{if } \text{obs}(X \text{ defaults}, 0) \text{ then } 0 \text{ else } 1000 \times \text{DKK}(X \rightarrow Y)$$

The seller $Z$ of a CDS agrees to pay the buyer $Y$ a compensation, say DKK 900, in the event that the issuer $X$ of the underlying bond defaults. In return, the buyer $Y$ of the CDS pays the seller $Z$ a premium. In this case, we consider a simple CDS with a single premium paid up front, say DKK 10. This agreement can be specified in the contract language as follows:

$$(10 \times \text{DKK}(Y \rightarrow Z)) \& \text{if } \text{obs}(X \text{ defaults}, 0) \text{ within } 30 \text{ then } 0 \text{ else } 0$$

Let $C_{\text{bond}}$ and $C_{\text{CDS}}$ be the above bond and CDS contract, respectively. We then combine the two contracts conjunctively to form the contract $C_{\text{bond}} \& C_{\text{CDS}}$ that describes the interaction between the CDS and the underlying bond that the CDS insures. In this compound contract, $X$ acts both as the holder of the bond and the buyer of the CDS, thereby interacting with the two parties $X$ and $Z$.

We will consider more realistic examples of CDSs with regular interest and premium payments in section 3.

2.2 Simple Type System for Contracts

In this section, we present the contract language systematically using a simple type system. As we will see, this type system is too lax to rule out contracts that violate the principle of causality. We shall give a more sophisticated type system in section 3 that takes temporal aspects into account and rules out contracts that violate the principle of causality.

Figure 1 presents the typing rules for the expression language. Real and Bool are the types of real-valued and Boolean-valued expressions, respectively. The typing rules use typing environments $\Gamma$, which are partial mappings from variable names to the set of expression types \{Real, Bool\}.

The expression language includes a number of common real-valued and Boolean operators, which are covered by the judgement $\vdash op : \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, defined in Figure 2. Instead of $\oplus(e_1, e_2)$, we also write $e_1 \oplus e_2$, instead of $¬e$ we write $¬e$, and instead of $if(e_1, e_2, e_3)$ we write if $e_1$ then $e_2$ else $e_3$.

Notice that the obs and acc combinators are indexed by a type $\tau$, which ranges over the set of expression types \{Real, Bool\}. We often omit this type index, if it obvious from the context in which the combinator appears. The obs, combinator uses labels drawn from a set Label. For instance, in our examples in section 3.1 we assume that "FX(U.S. DKK)" $\in$ Label_{Real} and "X exercises option" $\in$ Label_{Bool}. Moreover, we assume that labels in Label_{Real} may have an associated party that has control over it. That is, there is a partial mapping $\pi : \text{Label}_{\text{Real}} \rightarrow \text{Party}$. For instance, we have that $\pi(X \text{ exercises option}) = X$ for all $X \in \text{Party}$. In other words, the label "X exercises option" represents a decision taken by party $X$. As a shorthand we use the notation Label for the set Label_{Real} $\cup$ Label_{Bool} of all labels.

The typing rules for the contract combinators are given in Figure 3. Instead of $\vdash c : \text{Conr}$, we also write $\vdash c : \text{Conr}$, and we call a contract $c$ closed if $\vdash c : \text{Conr}$.

Note that our contract language also features let bindings of the form $\text{let } x = e \text{ in } c$. The intuitive meaning of such a contract is that it evaluates the expression $e$ at the current time and “stores” the resulting value in $x$ for later reference in the contract $c$. Let binding are essential for providing a fixed reference point in time, which is necessary for contracts constructed by the if-within combinator. For instance, we might wish to write an option contract that is cancelled as soon as a foreign exchange rate rises beyond a threshold relative to a previously observed exchange rate:

$\text{let } x = \text{obs}(\text{FX(U.S. USD)}, 0) \text{ in if obs(\text{FX(U.S. USD)}, 0) } \geq 1.1 \times x \text{ within 30 then } 0 \text{ else c}_{\text{option}}$
the time the contract started. Otherwise, the option described by the (elided) contract option becomes available.

Similarly, the let binding is also useful in the then branch of the if-then combinator and in the accumulation function in an expression formed by ace. We shall see more examples of using let bindings in section 2.3.2.

2.3 Denotational Semantics

The denotational semantics of a contract is given with respect to an external environment, which provides values for all observables and choices involved in the contract. A contract’s semantics is then given as a series of cash-flows between parties over time.

2.3.1 External Environments

Given an expression type \( \tau \in \{\text{Real, Bool}\} \), we write \( \llbracket \tau \rrbracket \) for its semantic domain, where \( \llbracket \text{Real} \rrbracket = \mathbb{R} \) and \( \llbracket \text{Bool} \rrbracket = \{\text{true, false}\} \). External environments (or simply environments for short) provide facts about observables and external decisions involved in contracts. An environment \( \rho \in \text{Env} \) maps each time offset \( t \in \mathbb{Z} \) and label \( l \in \text{Label} \) that identifies an observable or a choice, to a value \( \rho(l, t) \) in \( \llbracket \tau \rrbracket \):

\[
\text{Env} = \text{Label} \times \mathbb{Z} \to \llbracket \tau \rrbracket
\]

Notice that the second component of the domain is \( \mathbb{Z} \) and not \( \mathbb{N} \), that is, an environment may provide information about the past as well as the future.

2.3.2 Expressions

Environments are essential to the semantics of Boolean and real-valued expressions, which is otherwise a conventional semantics of arithmetic and logic expressions. In addition to an environment, we also need variable assignments that map free variables of type \( \tau \) to a value of type \( \llbracket \tau \rrbracket \). Given a typing environment \( \Gamma \), we define the set of variable assignments in \( \Gamma \), written \( \llbracket \Gamma \rrbracket \), as the set of all partial mappings \( \gamma \) from variable names to \( \mathbb{R} \cup \mathbb{B} \) such that \( \gamma(x) \in \llbracket \tau \rrbracket \) if \( \gamma(x) \in \llbracket \tau \rrbracket \).

Given an expression typing \( \Gamma \vdash e : \tau \), the semantics of \( e \), denoted \( \llbracket e \rrbracket_\tau \) is a mapping of type \( \llbracket \Gamma \rrbracket \times \text{Env} \to \llbracket \tau \rrbracket \). Instead of \( \llbracket e \rrbracket_\gamma,\rho \), we write \( \llbracket e \rrbracket_\gamma,\rho \). For each operator \( op \) with the typing judgement \( \Gamma \vdash op : \tau_1 \times \cdots \times \tau_n \to \tau \), we define a corresponding semantic function \( [op] : \llbracket \tau_1 \rrbracket \times \cdots \times \llbracket \tau_n \rrbracket \to \llbracket \tau \rrbracket \).

In order to give a semantics to the ace combinator, we need to be able to shift environments in time. To this end, we define for each environment \( \rho : \text{Env} \) and number \( d \in \mathbb{Z} \), the environment \( \rho/d \) as the mapping

\[
\rho/d : (i, l) \mapsto \rho(i + d, l) \quad (i \in \mathbb{Z}, l \in \text{Label})
\]

In other words, \( \rho/d \) is time-shifted \( d \) days into the future. The environment \( \rho/d \) is also called the promotion of \( \rho \) by \( d \).

The semantics of ace iterates the argument \( f \) by stepping backwards in time. This behaviour can be expressed equivalently using promotion of expressions, in analogy to promotion of environments. Promoting an expression by \( d \) translates all contained observables and choices \( d \) days into the future. For any expression \( e \) and \( d \in \mathbb{Z} \), the expression \( e \uparrow d \) is defined as:

\[
d \uparrow e = e \quad \text{if } e \text{ is literal or variable}
\]

\[
d \uparrow \text{obs}_r(p, i) = \text{obs}_r(p, d + i)
\]

\[
d \uparrow \text{op}(e_1, \ldots, e_n) = \text{op}(d \uparrow e_1, \ldots, d \uparrow e_n)
\]

\[
d \uparrow \text{acc}_r(\lambda x. f, k, a) = \text{acc}_r(\lambda x. (d \uparrow f), k, d \uparrow a)
\]

Observe that the left-hand side of a contract is given by its cash-flow trace, a mapping from time into the set \( \text{Trans} \) of asset transfers between
two parties:

\[ \text{Trans} = \text{Party} \times \text{Party} \times \text{Asset} \rightarrow \mathbb{R} \]

\[ \text{Trace} = \mathbb{N} \rightarrow \text{Trans} \]

Given a contract typing \( \Gamma \vdash c : \text{Contr} \), the semantics of \( c \), denoted \( C[c] \), is a mapping of type \( [\Gamma] \times \text{Env} \rightarrow \text{Trace} \). Instead of \( C[c] \) (\( \gamma, \rho \)), we write \( C[c]_{\gamma, \rho} \). Figure 2 shows the denotational contract semantics in full detail. Given a closed contract \( c \) (i.e. \( c : \text{Contr} \)), we simply write \( C[c]_{\gamma, \rho} \) instead of \( C[c]_{\emptyset, \rho} \), where \( \emptyset \) denotes the empty variable assignment.

The semantics of a unit transfer \( a(p_1 \rightarrow p_2) \) may seem confusing at first. If the two parties \( p_1 \) and \( p_2 \) coincide, it is equivalent to the zero contract. Otherwise, the semantics is a trace that has exactly two non-zero cash-flows: one from \( p_1 \) to \( p_2 \) and one in the converse direction but negative. This definition simplifies the definition of the semantics for the remaining contract combinators. A consequence of this approach is that for each contract \( c \), we have the following anti-symmetry property:

**Lemma 2.** For all \( \gamma, \rho, n, p_1, p_2, a \), we have that

\[ C[c]_{\gamma, \rho}(n)(p_1, p_2, a) = -C[c]_{\gamma, \rho}(n)(p_2, p_1, a) \]

In other words, if there is a cash-flow of magnitude \( r \) in one direction, there is a cash-flow of magnitude \(-r\) in the other direction.

The typing rules for the contract language and the expression sub-language ensure that the semantics given above is well-defined.

**Proposition 3** (well-defined semantics). Let \( \Gamma \) be a typing environment, \( \gamma \in [\Gamma] \), and \( \rho \in \text{Env} \).

(i) Given \( \Gamma \vdash c : \tau \), we have that \( C[c]_{\gamma, \rho} \in [\tau] \).

(ii) Given \( \Gamma \vdash c : \text{Contr} \), we have that \( C[c]_{\gamma, \rho} \in \text{Trace} \).

As a corollary, we obtain that each closed contract \( c \) yields a total function \( C[c] : \text{Env} \rightarrow \text{Trace} \).

### 2.4 Contract Equivalences

The denotational semantics provides a natural notion of contract equality. For each typing environment \( \Gamma \), we define the equivalence relation \( \equiv_{\Gamma} \) as follows:

\[ c_1 \equiv_{\Gamma} c_2 \text{ if and only if } \Gamma \vdash c_1 : \text{Contr}, \Gamma \vdash c_2 : \text{Contr}, \text{ and } C[c_1]_{\gamma, \rho} = C[c_2]_{\gamma, \rho} \text{ for all } \gamma \in [\Gamma], \rho \in \text{Env} \]

That means, whenever we have that \( c_1 \equiv_{\Gamma} c_2 \), then we can replace any occurrence of \( c_1 \) in a contract \( e \) by \( c_2 \) without changing the semantics of \( e \). As a shorthand we use the notation \( c_1 \equiv_{\Gamma} c_2 \) if \( c_1 : \text{Contr}, \Gamma \vdash c_2 : \text{Contr} \) implies \( c_1 \equiv_{\Gamma} c_2 \) for all \( \Gamma \).

A number of simple equivalences can be proved easily using the denotational semantics; Figure 2 gives some examples. These contract equivalences can be useful for simplifying a given contract, for instance to achieve a normalised format suitable for further processing.

Many of the equivalences in Figure 2 look similar to the axioms of vector spaces. The reasons for this is the fact that the set \( \text{Trans} \) forms a vector space over the field \( \mathbb{R} \), where the semantics of \( \emptyset, \& \), and \( \times \) are the zero vector, vector addition, and scalar multiplication, respectively.

### 2.5 Observing the Passage of Time

In a contract formed by the if-within construct, say the contract if \( b \) within \( d \) then \( c_1 \) else \( c_2 \), we know how much time

\[ e_1 \times (e_2 \times c) \equiv (e_1 \cdot e_2) \times c \quad d \theta \equiv \emptyset \]

\[ d_1 \uparrow (d_2 \uparrow c) \equiv (d_1 + d_2) \uparrow c \quad r \times \emptyset \equiv \emptyset \]

\[ d \uparrow (c_1 \& c_2) \equiv (d \uparrow c_1) \& (d \uparrow c_2) \quad 0 \times c \equiv 0 \]

\[ e \times (c_1 \& c_2) \equiv (e \times c_1) \& (e \times c_2) \quad c \& \emptyset \equiv c \]

\[ d \uparrow (e \times c) \equiv (d \uparrow e) \times (d \uparrow c) \quad c_1 \& c_2 \equiv c_2 \& c_1 \]

\[ (c_1 \times c) \& (c_2 \times c) \equiv (c_1 + c_2) \times c \]

\[ d \uparrow \text{if } b \text{ within } e \text{ then } c_1 \text{ else } c_2 \equiv \]

\[ \text{if } d \uparrow e \text{ within } d \uparrow c_1 \text{ else } d \uparrow c_2 \]

**Figure 6. Some Contract Equivalences.**

We can use the generalised if-within combinator, for example, to express a callable bond, i.e. a bond where the issuer may decide to redeem the bond prematurely. The amount that has to be paid to the holder of the bond may depend on the time that the issuer decides to call the bond.

\[ \text{if } \text{obs}(X \text{ calls bond, } 0) \text{ within } 30 \]

\[ \text{then } t. ((t - 30) + 100) \times \text{USD}(X \rightarrow Y) \]

\[ \text{else } 100 \times \text{USD}(X \rightarrow Y) \]

For the sake of presentation, the above contract is very simplified, but it illustrates the underlying concept: the issuer of the bond, party \( X \), may decide to call the bond at any time, but if it does so, it has to pay more to the holder of the bond, party \( Y \), depending on the time left until maturity (\( t - 30 \)).

### 3. Temporal Properties of Contracts

With the denotational semantics of contracts at hand, we can characterise a number of temporal properties, which are relevant for managing contracts. We shall consider two examples: causality, the property that a contract does not stipulate cash-flows that depend on "future" observables, and contract horizon, the minimum time span until a contract is certain to be zero.

#### 3.1 Contract Horizon

We define the horizon \( h \in \mathbb{N} \) of a closed contract \( e \) as the minimal time until the last potential cash-flow stipulated by the contract, under any environment. That is, it is the smallest \( h \in \mathbb{N} \) with

\[ C[c]_{\rho}(i)(x) = 0 \text{ for all } \rho \in \text{Env}, i \geq h, \text{ and } x \]

In other words, after \( h \) days, the cashflow for the contract \( c \) remains zero, for any environment \( \rho \). Note that since \( c \) is closed, i.e. \( \Gamma \vdash C : \text{Contr} \), we know that \( C[c]_{\rho}(i) \) is defined for any \( \rho \) and \( i \).

The horizon of a contract can be effectively computed in general (via the decidability of the first order theory of real closed fields).
But for our purposes it suffices to give a sound approximation of it by dropping the minimality requirement:

\[ \text{HOR}(0) = 0 \]

\[ \text{HOR}(a(p_1 \rightarrow p_2)) = 1 \]

\[ \text{HOR}(\text{let } x = e \text{ in } c) = \text{HOR}(c) \]

\[ \text{HOR}(e \times c) = \text{HOR}(c) \]

\[ \text{HOR}(d \uparrow e) = d \oplus \text{HOR}(e) \]

\[ \text{HOR}(c_1 \& c_2) = \max(\text{HOR}(c_1), \text{HOR}(c_2)) \]

\[ \text{HOR}(\text{if } e \text{ within } d \quad \text{then } c_1 \quad \text{else } c_2) = d \oplus \max(\text{HOR}(c_1), \text{HOR}(c_2)) \]

where

\[ a \oplus b = \begin{cases} 0 & \text{if } b = 0 \\ a + b & \text{otherwise} \end{cases} \]

We can show that the semantic contract horizon is never greater than the syntactic horizon computed by \( \text{HOR} \).

**Proposition 4** (soundness of syntactic horizon). Let \( h \) be the horizon of a closed contract \( c \). Then \( h \leq \text{HOR}(c) \).

### 3.2 Contract Causality

When used directly, the contract combinators allow for defining contracts that make no sense in reality as they make stipulations about cash-flow at time \( t \) that depends on input from the external environment strictly after \( t \). In other words, such contracts are not causal. For instance, one could define a transfer to be executed using the denotational semantics, we can give a precise definition of causality. Given \( t \in \mathbb{Z} \), we define an equivalence relation \( \equiv_t \) on \( \text{Env} \) that intuitively expresses that two environments agree until (and including) time \( t \). We define that \( p_1 \equiv_t p_2 \iff s \leq t \) implies \( p_1(l, s) = p_2(l, s) \), for all \( l \in \text{Label} \), and \( s \in \mathbb{Z} \). Causality can then be captured by the following definition: A closed contract \( c \) is causal iff for all \( t \in \mathbb{N} \) and \( p_1, p_2 \in \text{Env} \), we have that \( p_1 \equiv_t p_2 \) implies \( \forall l, Z \in \text{Label} . \quad \forall t \in \mathbb{Z} . \quad (c \mid \sigma_p)(t) = c\mid\sigma_{p_2}(t) \). That is, the cash-flows at any time \( t \) do not depend on observables and decisions after \( t \).

Contract causality is decidable (via the decidability of the first-order theory of real closed fields and the fact that contracts have a computable finite horizon), but computationally expensive. Moreover, causality is not a compositional property, i.e., a contract might be causal even though a subcontract is not causal. Compositionality will be important for the reduction semantics as we shall see in section 4.2. Therefore, we are looking for a compositional, conservative approximation of causality. The simplest such approximation is to require that for every sub-expression of the form \( \text{obs}(l, d) \), we have that \( d \leq 0 \). We call a contract that conforms to this syntactic criterion obviously causal.

Most practical contracts are in fact obviously causal and we have yet to find a contract that cannot be transformed into an equivalent contract that is obviously causal. For example, the following contract is causal but not obviously causal:

\[ \text{obs}(\text{FX}(\text{USD}, \text{DKK}), 1) \times \text{DKK}(X \rightarrow Y) \]

However, the above contract is equivalent to the following obviously causal contract (cf. Figure 6):

\[ 1 \uparrow \text{obs}(\text{FX}(\text{USD}, \text{DKK}), 0) \times \text{DKK}(X \rightarrow Y) \]

A more realistic example is the following chooser option, where the option buyer \( X \) may choose, in 30 days, whether to have a (European) call or put option. The buyer \( X \) may then, 30 days later, exercise the option:

\[ \text{let price } = \text{obs}(\text{FX}(\text{DKK}, \text{USD}), 60) \text{ in} \]

\[ \text{let payout } = \text{if } \text{obs}(X \text{ chooses call option}, 30) \text{ then max(\text{price} \hspace{1mm} - \hspace{1mm} \text{strike}, 0)} \text{ else max(\text{strike} - \text{price}, 0)} \text{ in} 60 \uparrow (\text{payout} \times \text{DKK}(Y \rightarrow X)) \]

Again this contract can be transformed into an equivalent contract that is obviously causal, but the above formulation is closer to the informal description of the contract.

The simple syntactic criterion of obvious causality is rather restrictive for formulating contracts. Most importantly, it is very fragile as it is not necessarily preserved by equivalence preserving contract transformations. For example, applying any of the equivalences from Figure 6 involving promotion of expressions (\( d \uparrow e \)) from left-to right may destroy obvious causality. To overcome this restriction, we refine the typing rules for contracts and expressions by indexing types with time offsets. The intended meaning of an expression \( e \) of type \( \tau \) is that the value of \( e \) is available at time \( t \) and any time after that. In other words, \( e \) does not depend on observations and decisions made strictly after time \( t \). In contrast, if a contract \( c \) is of type \( \text{Contr} \), then \( c \) makes no cash-flow stipulations strictly before \( t \).

The time indices \( t \) range over the set \( \text{Time}_{\rightarrow} = \mathbb{Z} \cup \{-\infty\} \), that is, we assume a time \(-\infty\) that is before any other time \( t \in \mathbb{Z} \). We also assume a total order \( \leq \) on \( \text{Time}_{\rightarrow} \) which is the natural order on \( \mathbb{Z} \) extended by \(-\infty \leq t \) for all \( t \in \text{Time}_{\rightarrow} \). Moreover, we define the addition \( t + d \) of a time \( t \in \text{Time}_{\rightarrow} \) by a number \( d \in \mathbb{Z} \) if \( t \in \mathbb{Z} \); if \( t \notin \mathbb{Z} \), then the addition is just ordinal addition in \( \mathbb{Z} \), otherwise \(-\infty + d = -\infty \). Subtraction \( t - d \) is defined as \( t + (-d) \).

The refined typing rules are given in Figure 7 and Figure 8. To distinguish the refined type system from the simple type system we use the notation \( \vdash_{\rightarrow} \) instead of \( \vdash \). Notice that the typing rules use timed type environments, which map time-indexed types instead of plain types.

The typing rules refine the original simple typing rules from section 4.2. Well-typing \( \dvdash_{\rightarrow} \) implies simple well-typing \( \dvdash \):

**Proposition 5.** Let \( \Gamma \) be a timed type environment and \( \Gamma \vdash \tau \) a type environment such that for every \( t \) such that \( x : \tau \in \Gamma \iff \text{iff } x : \tau \in \Gamma \), Then we have

(i) \( \Gamma \vdash_{\rightarrow} e : \tau \iff \Gamma \vdash e : \tau \), and

(ii) \( \Gamma \vdash_{\rightarrow} c : \text{Contr} \iff \Gamma \vdash c : \text{Contr} \).

Most importantly, we have that well-typed contracts are causal.

**Theorem 6.** If \( \vdash_{\rightarrow} c : \text{Contr} \), then \( c \) is causal.

Finally, we will give a sound and complete type inference procedure that is able to decide whether a given contract \( c \) is well-typed.
The time-indexing of types induces a subtyping order $\leq$ derived from the order $\leq$ on $\text{Time}_{\infty}$ defined as follows:

$$t_1^{\tau_1} \leq t_2^{\tau_2} \text{ iff } t_1 = t_2 \text{ and } t_1 \leq t_2$$

A useful property of expression and contract typing is that both are closed under subtyping, albeit in different directions:

**Lemma 7.**

(i) If $\Gamma \vdash t : \tau$, then $\Gamma \vdash t : \tau^s$ for all $s \geq t$.

(ii) If $\Gamma \vdash c : \text{Contr}^s$, then $\Gamma \vdash c : \text{Contr}^r$ for all $s \leq t$.

Expression typing is upwards closed, whereas contract typing is downwards closed. As a consequence, we know that well-typed expressions have minimal types. Moreover, if we extend the set of time indices $\text{Time}_{\infty}$ with an additional maximal element $\pm \infty$, we also obtain that well-typed contracts have maximal types. This property makes type inference straightforward. The corresponding type inference functions $\text{infer}_c$ and $\text{infer}_e$ are given in Figure 9. $\text{infer}_c$ takes a type environment $\Gamma$ and an expression $e$ and tries to find the minimal type of $e$; whereas $\text{infer}_c$ takes a type environment $\Gamma$ and a contract $c$ and tries to find the maximal type of $c$. If no type can be inferred, $\text{infer}_c(\Gamma, e)$ and $\text{infer}_c(\Gamma, c)$ are undefined. Note that $\text{infer}_c$ is successful, it returns types with type indices in $\text{Time}_{\infty}$, while $\text{infer}_e$ returns time indices from the extended set $\text{Time}_{\infty} \cup \{-\infty, +\infty\}$. The ordering $\leq$ is extended to $\text{Time}_{\infty}$ in the obvious way. Moreover, we define addition of elements $t \in \text{Time}_{\infty}$ with numbers $d \in \mathbb{Z}$ by $t + \infty + d = -\infty, +\infty + d = +\infty$. Otherwise, time indices are considered ordinary numbers in $\mathbb{Z}$.

We can then show that that this type inference procedure is sound and complete:

**Theorem 8 (infer_c is sound and complete).**

(i) If $\text{infer}_c(\Gamma, c) = t$, then $\Gamma \vdash c : \text{Contr}^r$ for all $s \leq t$.

(ii) If $\Gamma \vdash c : \text{Contr}^s$, then $\text{infer}_c(\Gamma, c) = t$ such that $t \leq s$.

Thus, according to Theorem 8, we obtain that if type inference returns a type for a contract $c$, then $c$ is causal.

**Corollary 9.** If $\text{infer}_c(\emptyset, c)$ is defined, then $c$ is causal.

The key ingredient for the simplicity of the type inference procedure is the closure under subtypes respectively supertypes as expressed in Lemma 2. This property will also be important in the next section where we will discuss contract transformations. Lemma 7 is crucial for showing that well-typed is preserved by contract transformations, in particular contract specialisation and contract reduction.

In the light of this observation, it is worthwhile reconsidering the typing rule for the scaling combinator $\times$, in particular the condition $t \leq t'$. This condition seems odd at first. We could get away with requiring that $t = t'$. The resulting type system would still entail causality and we would be able to give a sound and complete type inference procedure. However, we would lose part (ii) of Lemma 7. The condition $t \leq t'$ is in the typing rule for $\times$ decouples the time indices of contract and expression types. This decoupling is necessary due to the different interpretation of time indices for expression types compared to contract types. This different interpretation then also manifests itself in the difference in the subtyping behaviour described in Lemma 7.

### 4. Contract Transformations

The second important aspect of contract management is the transformation of contracts according to the semantics. We will consider two contract transformations, namely *specialisation*, which partially evaluates a contract based on partial information about the external environment, and *advancement*, which moves a contract into the future. The second transformation can be considered an operational semantics that is adequate for the denotational semantics presented in section 2.3.

These contract transformations are based on external knowledge provided by a partial external environment, i.e., on facts about observables and external decisions, which become gradually available as time passes. To this end, we consider the set of partial external environments $\text{Env}_p$, a superset of $\text{Env}$:

$$\text{Env}_p = \text{Label} \times \mathbb{Z} \rightarrow \{\tau\}$$

A contract $c$ can be transformed based on a partial environment $\rho \in \text{Env}_p$ that encodes the available knowledge about observables and decisions which influence $c$, leading to a *specialised* or *advanced* contract.

### 4.1 Contract Specialisation

The objective of *specialisation* is to simplify a given contract $c$ based on partial information about the external environment, that is, based on knowledge about some of the observables and decisions. The resulting contract $c'$ is equivalent to the original contract $c$ under any external environment that is compatible with the partial external environment that was the input to the specialisation.

Before we can define specialisation more formally, we need to introduce some terminology. An environment $\rho' \in \text{Env}$ extends a partial environment $\rho \in \text{Env}_p$ iff $\rho(l, t) = \rho'(l, t)$ for all $(l, t) \in \text{dom}(\rho)$. Furthermore, we define the set of partial variable assignments $\Gamma_p$, for a type environment $\Gamma$ as the set of all partial mappings $\gamma$ from variable names into $\mathbb{R} \cup \{\text{undefined}\}$ such that $\gamma(x) \in \{\tau\}$ for all $x \in \text{dom}(\gamma)$.

Given $\Gamma \vdash c : \text{Contr}$ with $i \in \{1, 2\}$, $\gamma \in \Gamma_p$, and $\rho \in \text{Env}_p$, we say that $c_1$ and $c_2$ are $\gamma, \rho$-equivalent, written $c_1 \equiv_{\gamma, \rho} c_2$, iff $\mathcal{C}[c_1]_{\gamma, \rho'} = \mathcal{C}[c_2]_{\gamma, \rho'}$ for all $\gamma' \in \Gamma$ and $\rho' \in \text{Env}$ that extends $\gamma$ and $\rho$, respectively. The goal for the specialisation function $\text{sp}_{c}$ is that it takes an external environment $\rho$ and a variables assignment $\gamma$ and transforms a contract $c$ into a contract $c'$ with $c \equiv_{\gamma, \rho} c'$.

In order to implement such a function $\text{sp}_{c}$, we also need a corresponding specialisation function $\text{sp}_e$ for expressions. To this end, we define a corresponding notion of $\gamma, \rho$-equivalence on expressions: Given $\Gamma \vdash e_i : \tau$ with $i \in \{1, 2\}$, $\gamma \in \Gamma_p$, and $\rho \in \text{Env}_p$, we say that $e_1$ and $e_2$ are $\gamma, \rho$-equivalent, written $e_1 \equiv_{\gamma, \rho} e_2$, iff $\mathcal{C}[e_1]_{\gamma', \rho'} = \mathcal{C}[e_2]_{\gamma', \rho'}$ for all $\gamma' \in \Gamma$ and $\rho' \in \text{Env}$ that extends $\gamma$ and $\rho$, respectively.

The implementation of $\text{sp}_{c}$ is sketched in Figure 9. The corresponding specialisation function $\text{sp}_e$ for expressions is straightfor-
ward and can be found in the Coq source files associated with this paper.

Notice that the definition of \( \text{spc} \) uses 'primed' versions of \( \times, \uparrow, \& \) and \textit{let}. These are \textit{smart constructors} for the corresponding contract language construct. They are functions that construct a contract that is equivalent to the contract that would have been constructed if we used the original contract language construct. But in addition it tries to simplify the contract. For instance \( x' \) is defined as follows:

\[
e 	imes' e = \begin{cases} e & \text{if } e = 1 \\ 0 & \text{if } e = 0 \lor e = 0 \\ e \times c & \text{otherwise} \end{cases}
\]

The other smart constructors work similarly. In particular, \( \text{let}' x = e \text{ in } e \) is equal to \( e \) if there is no free occurrence of \( x \) in \( e \).

Moreover, \( \text{spc} \) uses an auxiliary function \( \text{traverse} \), also shown in Figure 10 which tries to simplify the \textit{if-within} construct.

**Example 5.** Reconsider the CDS contract from Example 2. We want to see what happens if party \( X \) defaults. To this end, we define the partial environment \( \rho \) such that \( \rho(X \text{ defaults}, i) = \text{true} \) if \( i = 15 \), \( \rho(X \text{ defaults}, i) = \text{false} \) if \( i \neq 15 \), and otherwise \( \rho \) is undefined. In other words, we assume that party \( X \) defaults after 15 days: with this input, \( \text{spc} \) transforms the contract into

\[
(10 \times \text{DKK}(Y \rightarrow Z)) \& (15 \uparrow 900 \times \text{DKK}(Z \rightarrow Y))
\]

That is, \( Y \) pays \( Z \) DKK 10 today and \( Z \) pays \( Y \) DKK 900 in 15 days. On the other hand, if \( X \) does not default, i.e., if we define \( \rho(X \text{ defaults}, i) = \text{false} \) for all \( i \), then \( \text{spc} \) transforms the contract into

\[
(30 \uparrow 1000 \times \text{DKK}(X \rightarrow Y)) \& (10 \times \text{DKK}(Y \rightarrow Z))
\]

That is, \( Y \) pays \( Z \) DKK 10 today and \( X \) pays \( Y \) DKK 100 in 30 days.

We can show that the two specialisation functions \( \text{spc} \) and \( \text{spc} \) do indeed implement specialisation of contracts and expressions, respectively:

**Theorem 10.** Let \( \Gamma \) be a typing environment, \( \gamma \in [\Gamma]_{\rho} \), and \( \rho \in \text{Env} \).

(i) Given \( \Gamma \vdash e : \tau \), we have that \( \text{spc}(e, \gamma, \rho) \equiv_{\gamma, \rho} e \).

(ii) Given \( \Gamma \vdash e : \text{Contr} \), we have that \( \text{spc}(e, \gamma, \rho) \equiv_{\gamma, \rho} e \).

In particular, we have that specialisation preserves the typing, that is, \( \Gamma \vdash e : \text{Contr} \) implies that \( \Gamma \vdash \text{spc}(e, \gamma, \rho) : \text{Contr} \), and analogously for the refined type system.

**4.2 Reduction Semantics and Contract Advancement**

In addition to the denotational semantics, we equip the contract language with a reduction semantics \( \Downarrow \), which \textit{advances} a contract by one time unit. We write \( e \Downarrow_{\gamma, \rho} e' \), to denote that \( e \) is advanced...
to \( c' \) in the partial environment \( \rho \in \text{Env}_T \), where \( T \in \text{Trans} \) represents the transfers that the contract \( c \) stipulates during this time unit, and \( c' \) is the contract that describes all remaining obligations except these transfers (both under the assumptions represented by \( \rho \)). In order to define \( \Rightarrow^T_\rho \), we have to generalise it such that it works on open contracts as well. To this end, we also index the relation with a partial variable assignment \( \gamma \). In sum, the reduction semantics is a relation written as \( c \Rightarrow^T_\gamma \rho \ c' \), and we use the notation \( c \Rightarrow \rho \ c' \) for the special case that \( \gamma \) is the empty variable assignment. The reduction semantics is given in Figure \[I\]1.

We can show that the reduction semantics is computationally adequate w.r.t. the denotational semantics. In order to express this property, we need the notion that partial environment \( \rho \in \text{Env}_T \) is historically complete: by this we mean that \( \rho(t, i) \) is defined for all \( t \in \text{Label} \) and \( i \leq 0 \). In other words, we have complete knowledge about the past. For the sake of a clearer presentation, we formulate the adequacy property in terms of closed contracts only:

**Theorem 11** (Reduction semantics adequacy). Let \( \Gamma \vdash^T c : \text{Contr}_T^T \) and \( \rho \in \text{Env}_T \).

(i) If \( c \vdash^T_\rho c' \), then
   
   \( (a) \; C \vdash [c']_\rho (0) = T \), and
   
   \( (b) \; C \vdash [c']_\rho (i + 1) = C [c']_\rho /_1 (i) \) for all \( i \in \mathbb{N} \), \( \rho' \) that extend \( \rho \).

(ii) If \( c \vdash^T_\rho c' \), then \( \Gamma \vdash^T c' : \text{Contr}_{T-1} \).

(iii) If \( \rho \) historically complete, then there is a unique \( c' \) such that \( c \Rightarrow^T_\rho c' \) and \( T = C [c']_\rho (0) \).

Property (i) expresses that the reduction semantics is sound; (ii) expresses type preservation, and (iii) expresses a progress property.

Combining the three individual properties above, and specialising it to total environments \( \rho \in \text{Env}_T \), we can conclude that any well-typed contract yields an infinite reduction sequence, which reveals the contract’s complete denotational semantics:

\[
\begin{array}{c}
\vdash^T c (0) \\
\vdash^T c (1) \\
\vdash^T c (2) \\
\vdots
\end{array}
\]

Since contracts have a finite horizon (cf. section\[14\]), we know that there is some \( n \in \mathbb{N} \) such that \( c_i = \emptyset \) for all \( i \geq n \). In addition, one can show that there is some \( n \in \mathbb{N} \) such that \( c_i = \emptyset \) for all \( i \geq n \).

It is intuitively expected that we require a contract \( c \) to be causal in order to obtain a reduction \( c \Rightarrow^T_\rho c' \), given that \( \rho \) is only historically complete. For instance, given the contract \( c_1 = \text{obs}((l, i) \times a(p \rightarrow q), q) \), which is clearly not causal, we do not know its cash-flow \( T \) at time \( 0 \) given only knowledge about the environment at time \( 0 \) and earlier, since \( T \) depends on the value of the observable \( l \) at time \( 1 \).

However, even causality is not enough, and indeed our progress result in Theorem\[11\] requires well-typing. In fact, we cannot hope to devise a compositional reduction semantics that is adequate for all causal contracts. The problem is that causality is not a compositional property! For example, similarly to the contract \( c_1 \), also the contract \( c_2 = \text{obs}((l, i) \times a(q \rightarrow p), q) \) is not causal. However, the contract \( c_1 \land c_2 \) is equivalent to \( \emptyset \) and thus is causal. Therefore, being able to capture a conservative notion of causality that is compositional, e.g. in the form of well-typing, is crucial for the adequacy of the reduction semantics.

A central lemma for proving property (iii) of Theorem\[11\] is that the specialisation function \( \text{sps} \) is complete in the sense that it yields a literal if given a partial environment and a partial variable assignment that is “sufficiently defined”. More concretely, we have that if \( \Gamma \vdash^T e : \tau \), then \( \text{sps}(\tau, \gamma, \rho) \in \tau \), given that \( \rho \in \text{Env}_T \) and \( \gamma \in \Gamma \), are sufficiently defined. The “sufficiently defined” condition is dependent on the typing of \( e \). It requires that \( \gamma(x) \) is defined whenever \( x : \sigma \in \Gamma \) and \( s \leq t \), and that \( \rho(t, i) \) is defined whenever \( i \leq t \). In other words, \( \gamma \) and \( \rho \) satisfy the temporal dependencies implicated by the typing \( \Gamma \vdash^T e : \tau \).

In order to make the reduction semantics practically useful, we implement it in the form of a function \( \text{adv} \) that takes a contract \( c \), a partial variable assignment \( \gamma \), and a partial external environment \( \rho \) and returns a contract \( c' \) and the transfer function \( T \in \text{Trans} \) such that \( c \Rightarrow^T_\gamma \rho c' \). The function \( \text{adv} \) can be implemented by transcribing the inference rules from Figure\[I\]1 into a function definition. However, we have to make a small change in order to obtain an effectively computable function. The issue is the second rule for contracts of the form \( e \times c \). To implement this rule we have to check whether the derived transfer function for the contract \( c \) is equal to \( T_0 \), the empty transfer function. This is undecidable if we use the full function space \( \text{Trans} \) of transfer functions. However, transfer functions \( T \) that are the result of the semantics of a contract have finite support, that is, \( T(t) \neq 0 \) for only finitely many \( t \). Hence, we can represent transfer functions using finite maps, with which we can efficiently check whether a transfer function is the empty transfer function \( T_0 \). The implementation for \( \text{adv} \) can be found in the associated Coq source code along with the proof that it adequately implements the reduction semantics. The implementation also makes use of the asymmetric nature of transfer functions, that is, the fact that \( T(p_1, p_2, a) = -T(p_2, p_1, a) \) (cf. Lemma\[2\]), by only storing one of the values \( T(p_1, p_2, a) \) and \( T(p_2, p_1, a) \).

5. Coq Formalisation and Code Extraction

We have formalised the contract language in the Coq theorem prover. To this end, we have chosen an extrinsically typed representation using de Brujin indices. That is, the abstract syntax of contracts and expressions is represented as simple inductive data types and the typing rules are given separately as inductive predicate definitions.

The use of extrinsic typing—as opposed to intrinsic typing, where the type system of the meta language is used to encode the object language’s type system—has two important benefits. First of all, we have two different type systems: the simple type system from Figures\[1\] and\[2\] and the time-indexed type system from Figures\[7\] and\[8\]. With intrinsic typing, we would need to choose a single one. Secondly, the types representing the abstract syntax of contracts and expressions are simple algebraic data types. Coq’s built-in code extraction to generate Haskell or OCaml code does not work very well outside of the realm of algebraic data types—extraction is, at best, difficult with general inductive type families.

The use of extrinsic typing has some drawbacks, though. Some functions that are total on well-typed contracts (e.g., the denotational semantics) are only partial on untyped contracts. Transformations such as contract specialisation and advancement require a separate proof showing that well-typing is preserved.

All propositions and theorems given in sections\[2\] and\[3\] were proved using our formalisation in Coq. In the remainder of this section, we describe how executable Haskell code is produced from this formalisation. The resulting Haskell implementation provides an embedded domain-specific language to write concrete contracts and exposes the contract analysis and management functionality that we discussed in sections\[3\] and\[4\].

5.1 Generating Certified Code

Our goal is to obtain a certified contract management engine written in Haskell. While ideally, one would like the entire software stack (and even hardware stack) on which contracts are being managed to be certified, there are several non-certified components involved. The generated Haskell code has been compiled with a non-
expressions are represented by a type $e$ and we use the approach of Atkey et al. \[3\]. This approach allows us to provide a contract management framework as well as the built-in if-then-else construct both for expression- and contract-level conditionals (i.e., if-within).

\[
\begin{align*}
\emptyset & \Rightarrow_{\gamma, \rho} \emptyset & \text{if } e \text{ within } 0 & \text{then } c_1 \text{ else } c_2 & \Rightarrow_{\gamma, \rho} c' & \text{if } e \text{ within } d & \text{then } c_1 & \text{else } c_2 & \Rightarrow_{\gamma, \rho} c' \\
\end{align*}
\]

\[
\begin{align*}
(a(p_1 \to p_2), \forall p_1, p_2. & \Rightarrow_{\gamma, \rho} c' \text{ if } e \text{ within } d & \text{then } c_1 & \text{else } c_2 & \Rightarrow_{\gamma, \rho} c' \\
& \Rightarrow_{\gamma, \rho} c' \text{ if } e \text{ within } d & \text{then } c_1 & \text{else } c_2 & \Rightarrow_{\gamma, \rho} c' \\
& \text{otherwise} & \Rightarrow_{\gamma, \rho} c' \\
& \end{align*}
\]

\[
\begin{align*}
& \text{where } T_0 = \lambda t.0 & r * T = \lambda t. r \cdot T(t) & T_1 + T_2 = \lambda t. T_1(t) + T_2(t) \\
& T_{p_1, p_2, \alpha} = \lambda(p_1', p_2', \alpha'). & \begin{cases} 1 & \text{if } (p_1, p_2, \alpha) = (p_1', p_2', \alpha') \\
0 & \text{otherwise} \end{cases} \\
& \end{align*}
\]

Figure 11. The reduction semantics of the contract language.

certified Haskell compiler and runs under a non-certified runtime system, most likely on top of a non-certified operating system. Another component that must be trusted is the Coq Haskell extraction itself (which has been addressed to some extent \[13\]). Our work requires trust into these lower-level components.

Instead of extracting Haskell code for types and functions from Coq’s standard library, such as list and option, we map these to the corresponding implementations in Haskell’s standard library. Coq’s code extraction facility provides corresponding customisation features that allow this. In addition, our Coq formalisation uses axiomatic abstract types, i.e. types that are only given by their properties, and we can thus not extract code for them. Examples are the types for assets, parties and finite maps. We use the same customisation mechanism to map these types to corresponding types in Haskell.

Code extraction from Coq into Haskell (or OCaml) does not simply translate function and type definitions from one language to another. It also elides logical parts, i.e. those of sort Prop. Using data types containing proofs and defining functions that operate on them can be useful for establishing invariants that are maintained by those functions. In many cases, this simplifies the proofs drastically.

Code extraction strips those “embedded” proofs from the code.

5.2 Implementing an Embedded Domain-Specific Language

In order to make the contract language usable, we need to provide a frontend that allows the user to write contracts in a convenient surface syntax. In particular, we do not want the user to write contracts using de Bruijn indices. Instead of writing a parser that translates the surface syntax into abstract syntax, we have implemented the contract language as an embedded domain-specific language. This approach allows us to provide a contract management framework with minimal effort. In addition, the approach leads to less untrusted code.

In order to build a combinator library to construct contracts and expressions, we use the approach of Atkey et al. \[3\]. This approach allows us to provide a combinator library that uses higher-order abstract syntax (HOAS) to represent variable binders. For example, expressions are represented by a type $Int \to Expr$, where $Expr$ is the type of expressions extracted from the Coq formalisation and

\[
\begin{align*}
& \text{-- Expressions} \\
& \text{acc :: } E \exp \Rightarrow (exp t \to exp t) \to Int \to exp t \to exp t \\
& \text{iObs :: } E \exp \Rightarrow String \to Int \to exp R \\
& \text{bObs :: } E \exp \Rightarrow String \to Int \to exp B \\
& \text{max, min, (.), (+), (/), (\_\_\_)} & \Rightarrow E \exp \Rightarrow R \exp \Rightarrow R \exp \Rightarrow R \\
& \text{($(\&\&)., (\_\_\_))} & \Rightarrow E \exp \Rightarrow B \exp \Rightarrow B \\
& \text{not} & \Rightarrow E \exp \Rightarrow B \exp B \\
& \text{false, true, } \Rightarrow E \exp \Rightarrow B \\
& \text{-- Contracts} \\
& \text{type Contr = forall exp contr . C exp contr \Rightarrow contr} \\
& \text{transfer :: C exp contr \Rightarrow Party \to Party \to Asset \to contr} \\
& \text{zero :: C exp contr \Rightarrow contr} \\
& \text{letc :: C exp contr \Rightarrow exp t \to (exp t \to contr) \to contr} \\
& \text{(&) :: C exp contr \Rightarrow contr \to contr \to contr} \\
& \text{(\_\_\_)} :: C exp contr \Rightarrow Int \to contr \to contr \\
& \text{horizon :: Contr \Rightarrow Int} \\
& \text{welletyped :: Contr \Rightarrow Bool} \\
& \text{advance :: Contr \Rightarrow ExtEnvP \to (Contr, FMap)} \\
& \text{specialise :: Contr \Rightarrow ExtEnvP \to Contr} \\
& \end{align*}
\]

Figure 12. Interface for the Haskell extracted contract library.

the integer argument is used to keep track of the levels of nested variable binders. In addition, this approach uses type classes in order to keep the representation abstract. This abstraction ensures parametricity, which is needed in order to guarantee that the representation of binders is adequate. The interface of the resulting Haskell combinator library is given in Figure \[12\].

The type classes $E$ and $C$ are used to represent expressions and contracts, respectively. The type $Contr$ represents closed contracts. In addition we use the types $R$ and $B$ to indicate that an expression is of type $Real$ or $Bool$, respectively. We can also use Haskell decimal literals to write Real-typed literals in the expression language as well as the built-in if-then-else construct both for expression- and contract-level conditionals (i.e., if-within).
CDS is different: the buyer has to pay a monthly premium instead of a single up-front premium.

Haskell code for a CDS for a bond.

```haskell
import RebindableEDSL

bondCDS :: Contr
bondCDS = bond (12 :: Int) DKK 10 1000 Y X &
             cds (12 :: Int) DKK 9 1000 Y ZX

cds months cur premium comp buyer seller ref =
step months
  where step i = if i ≤ 0 then zero
    else premium # transfer buyer seller cur &
            comp # transfer seller buyer cur
    else step (i − 1)

bond months cur inter nom holder issuer =
step months
  where step i = if i ≤ 0
    then nom # transfer issuer holder cur
    else inter # transfer issuer holder cur &
            comp # transfer issuer holder cur &
            nominal # transfer issuer issuer cur &
            comp # transfer issuer issuer cur
    else step (i − 1)
```

Figure 14. Haskell code for a CDS for a bond.

Figure 13 illustrates the use of the combinators. It shows a complete Haskell file that imports the contract library and defines two contracts: the Asian and the American option that we have presented in section 2.1.

In a later version of their work, Peyton-Jones and Eber also sketched an operational semantics for contract management [21]. A full reduction semantics is given by Andersen et al. [1] as well as Hvitved et al. [13] along with proofs of their adequacy with respect to the corresponding denotational semantics. The absence of observables in the work of Andersen et al. [1] and Hvitved et al. [13] simplifies the reduction semantics: there is no need to partially evaluate expressions and it is syntactically impossible to write contracts that are not causal.

A different semantic approach that we did not discuss in this paper is an axiomatic semantics. Such an axiomatic treatment has been studied by Schuldenuzker [23]. Interestingly, this axiomatisation of two-party contracts is not based on equality of contracts but rather an order ≤ on contracts. Equality of contracts is then derived from the order ≤.

6. Related Work

Contract Languages

Research on formal contract languages goes back to the work of Lee [16] on electronic contracts. Since then, many different approaches have been studied. An overview over this broad area of contract formalisms can be found in the surveys by Hvitved [11,12].

Most relevant to our work is the pioneering work on financial contracts by Peyton Jones et al. [22]. This work has evolved into the company LexiFi, which has implemented the techniques on top of their MLFi variant of OCaml [19]. The resulting contract management platform runs worldwide in many financial institutions through its integration in key financial institutions, such as Bloomberg and with large asset-management platforms, such as SimCorp Dimension [24], a wall-to-wall asset-management platform for financial institutions.

Based also on earlier work on contract languages [2], in the last decade, domain specific languages for contract specifications have been widely adopted by the financial industry, in particular in the form of payoff languages, such as the payoff language used by Barclays [17]. It has thus become well known that domain specific languages for contract management provide for more agility, shorter time-to-market for new products, and increased assurance of software quality. See also [19] for an overview of resources related to domain specific languages for the financial domain.

Multi-party contracts have been investigated earlier in the work of Andersen et al. [11] and Henglein et al. [20] on establishing a formal transaction system for enterprise resource planning (ERP), which resembles a process calculi that matches up concrete transactions with abstract specifications of transactions agreed upon in a contract. The absence of explicit observables in these languages avoids the issue of syntactically expressible contracts that are not causal.

Semantics

We equipped our contract language with a denotational semantics as well as an operational semantics in the form of a reduction relation. Also Peyton Jones et al. [22] considered a denotational semantics, however, their semantics is based on stochastic processes. Our denotational semantics draws from previous work on trace-based contract formalisms [1,13,15]. However, in order to accommodate the financial domain that we are targeting we needed to add observables to the language and consequently the semantics. While many financial contracts can be formulated without observables, we found examples – such as double barrier options – that we were not able to express without observables.

In the course of the last decade, domain specific languages for contract specifications have been widely adopted by the financial industry, in particular in the form for financial institutions. While many financial contracts can be formulated without observables, we found examples – such as double barrier options – that we were not able to express without observables.

Software Verification and Certified Software

In the course of the last decade, it has been shown that formal software verification is a mature technology. It has been applied to realistic software systems.
such as compilers [17] and operating system kernels [14]. The use of code extraction to obtain executable code from a formally verified implementation is an established technique in the community [5, 8, 18]. Despite the demonstrated feasibility of verification of critical pieces of software, we have yet to see adoption of this technology for financial software in general and for financial contract languages in particular.

7. Conclusions and Future Work

We have presented a symbolic framework for modelling financial contracts, capable of expressing common FX and other derivatives, and equipped with a precise cash-flow semantics. The framework describes multi-party contracts and can therefore model entire portfolios for holistic risk analysis and management purposes. Contracts can be analysed for their temporal dependencies and horizon, and gradually evolved until the horizon has been reached, following a reduction semantics based on gradually-available external knowledge in an environment.

Our model is implemented using the Coq proof assistant, enabling us to certify the intended properties of our contract analyses and transformations, against the denotational cash-flow trace semantics. The Coq proof assistant allows for extracting a Haskell implementation from the implemented contract manipulation functionality. The resulting Haskell module can be used as a certified core of a portfolio management framework, by adding suitable interface modules which define a reduced and protecting API.

As future work, we plan to explore and model more symbolic contract transformations that are central to day-to-day contract management, and to extend the contract analyses with features relevant for contract valuation. We are also considering generalising contracts to use continuous time instead of discrete time. That is, the denotational semantics of contracts is a function of type \( R \rightarrow )\) instead of \( N \rightarrow Trans\). We conjecture that the transformations and analyses can still be performed in this generalised setting. For specialisation and advancement, we would have to make additional assumptions about how the partial external environments—which are input to these transformations—are represented. A reasonable choice would be to assume that partial external environments are given as a finite sampling.

Another natural extension of the language is an iteration combinator \( \text{iter} \), which works like the accumulation combinator \( \text{ace} \), but works on the level of contracts instead of expressions. This combinator allows us to concisely express contracts with repetition, such as the bond and CDS example from Figure 14. At the moment we rely on the meta language (in this case Haskell) in order to construct such contracts. These are still contracts in our core contract language and we can apply all of our contract management tool set to them, but the \( \text{iter} \) combinator would yield a much more compact representation.

Finally, we are interested in exploring the possibility of bridging symbolic techniques with numerical methods, such as stochastic and closed-form contract valuation (which is probably the most important use case of contract DSLs in general). Our contract analyses are geared towards identifying the external entities that need to be modelled in a pricing engine and we are currently working on deploying the certified contract engine in a contract and portfolio pricing and risk calculation prototype [20].

References