Bahr, Berthold, Elsman. Certified Symbolic Management of Financial Multi-party Contracts
In: ICFP 2015, ACM. © 2015, New York: ACM.

Certified Symbolic Management of Financial Multi-party Contracts *

Patrick Bahr
Department of Computer Science,
University of Copenhagen (DIKU)
Copenhagen, Denmark
paba@di.ku.dk

Jost Berthold
Commonwealth Bank of Australia †
Sydney, Australia
jbertholdacm.org

Martin Elsman
Department of Computer Science,
University of Copenhagen (DIKU)
Copenhagen, Denmark
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 relegates 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

General Terms Languages, Verification

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

* This work has been partially supported by the Danish Council for Strategic Research under contract number 10-092299 (HIPERFIT [6]), and the Danish Council for Independent Research under Project 12-132365.

† This research was done at DIKU, University of Copenhagen.

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, many banks and financial institutions today use domain-specific languages (DSLs) to describe complex financial contracts.

The seminal work by Peyton-Jones, Eber, and Seward on bilateral financial contracts [28] shows how an algebraic approach to contract specification can be used for valuation of contracts (when combined with a model of the underlying observables).[1] It also introduces a contract management model where contracts gradually evolve into the empty contract as knowledge of underlying observables becomes available and decisions are taken.

In almost all prior work on financial contract languages, contracts are modelled as bilateral agreements held by one of the involved parties. In contrast, our approach uses a generalised contract model where a contract specifies the obligations and rights of potentially many different 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 effect of parties defaulting or merging. For valuation purposes, and for other analyses, a contract can be viewed from the point of view of a particular party to obtain the classical bilateral contract view. Moreover, portfolios can be expressed simply by composing contracts. On top of that, the multi-party perspective is *required* for certain kinds of risk analyses, demanded by regulatory requirements for certain financial institutions, such as the daily calculation of Credit Value Adjustments (CVA) [12].

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 the contracts 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 intent, and in particular, whether the contract can be executed according to the underlying execution model. In this paper, we partially address this issue by providing a type system for the contract language, which guarantees that contracts can indeed be executed. In particular, the type system guarantees causality of contracts, which means that asset trans-

[1] 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 [6], hosted at DIKU.

fers cannot depend on a decision or the value of an underlying observable which will only be available in the future. 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 contract management framework and the accompanying 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) and demonstrate that it can express real-world contracts and portfolios, such as foreign exchange swaps and options, credit default swaps, and portfolios holding contracts with multiple counter-parties.[2] The contract DSL has been designed for symbolic rather than numerical computation, and cleanly separates all stochastic aspects from the core contract combinators.
- By means of a denotational cash-flow semantics for the DSL (Section 2.3), we 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.

The certified implementation of the contract language is available online[3] together with Coq proofs of all propositions and theorems mentioned in this paper. Currently, the contract framework is being deployed in a contract and portfolio pricing and risk calculation prototype [26], 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. These stipulations may depend on observable underlying values such as foreign exchange rates, stock prices, and market indexes. But they can also be at the discretion of one of the involved parties (e.g., in an option).

Our contract language allows us to express such contracts succinctly and in a compositional manner. To facilitate compositionality, our language employs a relative notion of time. Figure 1 gives an overview of the language's syntax. But before we discuss the language in more detail, we explore it with the help of four concrete example contracts.

2.1 Examples

We shall illustrate our contract DSL using examples from the foreign exchange (FX) market and days as the basic time unit, but the concepts generalise easily to other settings. For the purpose of our examples, cash-flows are based on a fixed set of *currencies*.

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.

$$90 \uparrow 100 \times (\mathsf{USD}(Y \to X) \,\&\, 6.5 \times \mathsf{DKK}(X \to Y))$$

The contract $\mathsf{USD}(Y \to X)$ stipulates that party Y must transfer one unit of USD to party X *immediately*. Similarly, $\mathsf{DKK}(X \to Y)$ stipulates that party X must transfer one unit of DKK to party Y. The combinator $\times$ 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 following conditional, which is an iterating generalisation of a simple alternative (*if-then-else*):

$$\textbf{if } \ldots \textbf{ within } \ldots \textbf{ then } \ldots \textbf{ else } \ldots$$

As an example, consider an *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.

$$\begin{aligned}&\textbf{if } \mathbf{obs}(X \text{ exercises option}, 0) \textbf{ within } 90\\ &\textbf{then } 100 \times (\mathsf{USD}(Y \to X) \,\&\, 6.5 \times \mathsf{DKK}(X \to Y))\\ &\textbf{else } \emptyset\end{aligned}$$

This contract uses an observable external decision, expressed using **obs** (which uses a time offset 0, meaning the current day), and the *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. Otherwise, the contract becomes empty ($\emptyset$) after 90 days.

The expression language also features an accumulation combinator **acc**, which *accumulates* a value over a given number of days from the past until the current day. The accumulator can be used to describe *Asian options* (or average options), for which a price is established from an average of past prices instead of just one observed price.

[2] Examples were provided by partners of the HIPERFIT Research Center.

[3] See `https://github.com/HIPERFIT/contracts`.

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*.

$$90 \uparrow \mathbf{if}\ \mathbf{obs}(X \text{ exercises option}, 0)\ \mathbf{within}\ 0$$
$$\mathbf{then}\ 100 \times (\mathsf{USD}(Y \to X)\ \&(\mathit{rate} \times \mathsf{DKK}(X \to Y)))$$
$$\mathbf{else}\ \emptyset$$

where $\mathit{rate} = \mathbf{acc}(\lambda r.\ r + \mathbf{obs}(\mathsf{FX}(\mathsf{USD}, \mathsf{DKK}), 0), 30, 0)/30$

Here, *rate* is just a meta variable to facilitate the reading of the contract. In addition to the decision expressed as a Boolean observable, this contract uses an **obs** expression to observe the exchange rate between USD and DKK (again at offset 0, thus on the current day). Observed values are accumulated to the *rate* using 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 only had two parties. To illustrate the multi-party 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 1000, at an agreed time in the future, say in 30 days. For this contract we also want to model the eventuality that the issuer X defaults. To this end, we use an observable "X defaults":

$$\mathbf{if}\ \mathbf{obs}(X \text{ defaults}, 0)\ \mathbf{within}\ 30\ \mathbf{then}\ \emptyset$$
$$\mathbf{else}\ 1000 \times \mathsf{DKK}(X \to Y)$$

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:

$$(10 \times \mathsf{DKK}(Y \to Z))\ \&\ \mathbf{if}\ \mathbf{obs}(X \text{ defaults}, 0)\ \mathbf{within}\ 30$$
$$\mathbf{then}\ 900 \times DKK(Z \to Y)$$
$$\mathbf{else}\ \emptyset$$

Let c_{bond} and c_{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, Y 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 5.2.

2.2 Simple Type System for Contracts

In this section, we present the contract language systematically using a simple type system. This type system allows us to give a well-defined denotational semantics (see Section 2.3), but it is too lax to rule out contracts that violate the principle of causality. Therefore, we shall refine this type system in Section 3.2 such that it takes temporal aspects into account, which in turn facilitates a computationally adequate reduction semantics (Section 4.2).

Figure 1 gives an overview of the syntax of the contract language including the expression sub-language. For the syntax we assume a countably infinite set of variables Var, a set of labels Label, a set of assets Asset, a set of parties Party, and a set of operators Op.

Labels are used to refer to observables. To this end, we assume that each label in Label is assigned a unique type τ, and we write

$$\begin{array}{rl}
\text{types} & \tau ::= \mathsf{Real} \mid \mathsf{Bool} \\
\text{expressions} & e ::= x \mid r \mid b \mid \mathbf{obs}(l,t) \mid op(e_1, \ldots, e_n) \mid \\
& \quad \mathbf{acc}(\lambda x.\ e_1, d, e_2) \\
\text{contracts} & c ::= \emptyset \mid \mathbf{let}\ x = e\ \mathbf{in}\ c \mid d \uparrow c \mid c_1\ \&\ c_2 \mid e \times c \mid \\
& \quad a(p \to q) \mid \mathbf{if}\ e\ \mathbf{within}\ d\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2
\end{array}$$

where $x \in \mathsf{Var}, r \in \mathbb{R}, b \in \mathbb{B}, l \in \mathsf{Label}, t \in \mathbb{Z}$, $d \in \mathbb{N}, a \in \mathsf{Asset}, p, q \in \mathsf{Party}, op \in \mathsf{Op}$

Figure 1. Syntax of the contract language.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash r : \mathsf{Real}} \qquad \frac{}{\Gamma \vdash b : \mathsf{Bool}} \qquad \frac{l \in \mathsf{Label}_\tau}{\Gamma \vdash \mathbf{obs}(l,t) : \tau}$$

$$\frac{\Gamma \vdash e_i : \tau_i \quad \vdash op : \tau_1 \times \cdots \times \tau_n \to \tau}{\Gamma \vdash op(e_1, \ldots, e_n) : \tau} \qquad \frac{\Gamma, x : \tau \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{acc}(\lambda x.\ e_1, d, e_2) : \tau}$$

$\boxed{\Gamma \vdash c : \mathsf{Contr}}$

$$\frac{}{\Gamma \vdash \emptyset : \mathsf{Contr}} \qquad \frac{}{\Gamma \vdash a(p \to q) : \mathsf{Contr}}$$

$$\frac{\Gamma \vdash c : \mathsf{Contr}}{\Gamma \vdash d \uparrow c : \mathsf{Contr}} \qquad \frac{\Gamma \vdash c_i : \mathsf{Contr}}{\Gamma \vdash c_1\ \&\ c_2 : \mathsf{Contr}}$$

$$\frac{\Gamma \vdash e : \mathsf{Real} \quad \Gamma \vdash c : \mathsf{Contr}}{\Gamma \vdash e \times c : \mathsf{Contr}} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash c : \mathsf{Contr}}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ c : \mathsf{Contr}}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \quad \Gamma \vdash c_i : \mathsf{Contr}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{within}\ d\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \mathsf{Contr}}$$

Figure 2. Simple typing rules for contracts and expressions.

$\boxed{\vdash op : \tau_1 \times \cdots \times \tau_n \to \tau}$

$\vdash \oplus : \mathsf{Real} \times \mathsf{Real} \to \mathsf{Real}$	for $\oplus \in \{+, -, \cdot, /, max, min\}$
$\vdash \oplus : \mathsf{Real} \times \mathsf{Real} \to \mathsf{Bool}$	for $\oplus \in \{\leq, <, =, \geq, >\}$
$\vdash \oplus : \mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$	for $\oplus \in \{\wedge, \vee\}$
$\vdash \neg : \mathsf{Bool} \to \mathsf{Bool}$	
$\vdash \mathbf{if} : \mathsf{Bool} \times \tau \times \tau \to \tau$	for $\tau \in \{\mathsf{Real}, \mathsf{Bool}\}$

Figure 3. Typing of expression operators.

Label_τ for the set of labels of type τ. For instance, in our examples in Section 2.1, we assume that "$\mathsf{FX}(\mathsf{USD}, \mathsf{DKK})$" $\in \mathsf{Label}_{\mathsf{Real}}$ and "X exercises option" $\in \mathsf{Label}_{\mathsf{Bool}}$. Moreover, we assume that labels in $\mathsf{Label}_{\mathsf{Bool}}$ may have an associated party that has control over it. That is, there is a partial mapping $\pi : \mathsf{Label}_{\mathsf{Bool}} \rightharpoonup \mathsf{Party}$. For instance, we have that $\pi(X \text{ exercises option}) = X$ for all $X \in \mathsf{Party}$. In other words, the label "X exercises option" represents a decision taken by party X.

Figure 2 presents the simple type system for the contract language. The typing rules use typing environments Γ, which are partial mappings from variables to expression types. Instead of $\emptyset \vdash c : \mathsf{Contr}$, we also write $\vdash c : \mathsf{Contr}$, and we call a contract c *closed* if $\vdash c : \mathsf{Contr}$.

The expression sub-language includes a number of common real-valued and Boolean operators, which are covered by the judge-

ment $\vdash op : \tau_1 \times \cdots \times \tau_n \to \tau$, defined in Figure 3. Instead of $\oplus(e_1, e_2)$, we also write $e_1 \oplus e_2$, instead of $\neg(e)$ we write $\neg e$, and instead of $\mathbf{if}(e_1, e_2, e_3)$ we write $\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$.

Notice that our contract language also features let bindings of the form $\mathbf{let}\ x = e\ \mathbf{in}\ c$. 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 bindings 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:

$$\begin{aligned}&\mathbf{let}\ x = \mathbf{obs}(\mathsf{FX}(\mathsf{EUR},\mathsf{USD}), 0)\\ &\mathbf{in}\ \mathbf{if}\ \mathbf{obs}(\mathsf{FX}(\mathsf{EUR},\mathsf{USD}), 0) \geq 1.1 \cdot x\ \mathbf{within}\ 30\\ &\quad \mathbf{then}\ \emptyset\ \mathbf{else}\ c_{\text{option}}\end{aligned}$$

The above contract is equivalent to the zero contract if the exchange rate EUR/USD rises 10 percent above the exchange rate observed at the time the contract started. Otherwise, the option described by the (elided) contract c_{option} becomes available.

Similarly, the let binding is also useful in the **then** branch of the *if-within* combinator and in the accumulation function in an expression formed by **acc**. We shall see more examples of using let bindings in Section 2.5.

In this paper, let bindings are limited to bind expressions only. It is straightforward to extend the language and its metatheory to include let bindings for contracts, and for practical implementations this is very useful in order to obtain compact contract representations. However, such let bindings have no semantic impact, and in the interest of simplicity and conciseness we have elided them.

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.

Given an expression type $\tau \in \{\mathsf{Real}, \mathsf{Bool}\}$, we write $[\![\tau]\!]$ for its semantic domain, where $[\![\mathsf{Real}]\!] = \mathbb{R}$ and $[\![\mathsf{Bool}]\!] = \mathbb{B}$. External environments (or simply *environments* for short) provide facts about observables and external decisions involved in contracts. The set of environments Env consists of functions $\rho : \mathsf{Label} \times \mathbb{Z} \to \mathbb{R} \cup \mathbb{B}$ that map each time offset $t \in \mathbb{Z}$ and label $l \in \mathsf{Label}_\tau$ that identifies an observable or a choice, to a value $\rho(l, t)$ in $[\![\tau]\!]$. Notice that the second argument t is an integer and not necessarily a natural number. That is, an environment may provide information about the past as well as the future. 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 environments, we also need variable assignments that map each free variable of type τ to a value in $[\![\tau]\!]$. Given a typing environment Γ, we define the set of *variable assignments* in Γ, written $[\![\Gamma]\!]$, as the set of all partial mappings γ from variable names to $\mathbb{R} \cup \mathbb{B}$ such that $\gamma(x) \in [\![\tau]\!]$ iff $x : \tau \in \Gamma$.

Figure 4 details the full denotational semantics of expressions and contracts. We first look at the semantics of expressions: Given an expression typing $\Gamma \vdash e : \tau$, the semantics of e, denoted $\mathcal{E}\,[\![e]\!]$ is a mapping of type $[\![\Gamma]\!] \times \mathsf{Env} \to [\![\tau]\!]$. Instead of $\mathcal{E}\,[\![e]\!]\,(\gamma, \rho)$, we write $\mathcal{E}\,[\![e]\!]_{\gamma,\rho}$. For each operator *op* with the typing judgement $\vdash op : \tau_1 \times \cdots \times \tau_n \to \tau$, we define a corresponding semantic function $[\![op]\!] : [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]$. For example, $[\![+]\!]$ is the usual addition on $\mathbb{R}$.

In order to give a semantics to the **acc** combinator, we need to shift environments in time. To this end, we define for each environment $\rho \in \mathsf{Env}$ and time offset $t \in \mathbb{Z}$, the *promotion* of ρ by t as the following mapping:

$$\rho/t\ :\ (l, i) \mapsto \rho(l, i + t) \qquad (i \in \mathbb{Z}, l \in \mathsf{Label})$$

In other words, ρ/t is time-shifted t days into the future.

The semantics of **acc** iterates the argument e_1 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 t days translates all contained observables and choices t days into the future. For any expression e and $t \in \mathbb{Z}$, the expression $t \Uparrow e$, is defined as:

$$\begin{aligned}t \Uparrow e &= e \qquad \text{if } e \text{ is a literal or variable}\\ t \Uparrow \mathbf{obs}(l, t') &= \mathbf{obs}(l, t + t')\\ t \Uparrow op(e_1, \ldots, e_n) &= op(t \Uparrow e_1, \ldots, t \Uparrow e_n)\\ t \Uparrow \mathbf{acc}(\lambda x.\ e_1, d, e_2) &= \mathbf{acc}(\lambda x.(t \Uparrow e_1), d, t \Uparrow e_2)\end{aligned}$$

Observables and choices are translated, and the promotion propagates downwards into all subexpressions.

Promotion of expressions can be semantically characterised by promotion of environments:

$$\boxed{\mathcal{E}\,[\![e]\!] : [\![\Gamma]\!] \times \mathsf{Env} \to [\![\tau]\!]}$$

$$\begin{aligned}\mathcal{E}\,[\![r]\!]_{\gamma,\rho} &= r; \quad \mathcal{E}\,[\![b]\!]_{\gamma,\rho} = b; \quad \mathcal{E}\,[\![x]\!]_{\gamma,\rho} = \gamma(x)\\ \mathcal{E}\,[\![\mathbf{obs}(l,t)]\!]_{\gamma,\rho} &= \rho(l,t)\\ \mathcal{E}\,[\![op(e_1,\ldots,e_n)]\!]_{\gamma,\rho} &= [\![op]\!]\,(\mathcal{E}\,[\![e_1]\!]_{\gamma,\rho}, \ldots, \mathcal{E}\,[\![e_n]\!]_{\gamma,\rho})\\ \mathcal{E}\,[\![\mathbf{acc}(\lambda x.\ e_1, d, e_2)]\!]_{\gamma,\rho} &= \begin{cases}\mathcal{E}\,[\![e_2]\!]_{\gamma,\rho} & \text{if } d = 0\\ \mathcal{E}\,[\![e_1]\!]_{\gamma[x\mapsto v],\rho} & \text{if } d > 0\end{cases}\\ &\text{where} \qquad v = \mathcal{E}\,[\![\mathbf{acc}(e_1, d-1, e_2)]\!]_{\gamma,\rho/-1}\end{aligned}$$

$$\boxed{\mathcal{C}\,[\![c]\!] : [\![\Gamma]\!] \times \mathsf{Env} \to \mathbb{N} \to \mathsf{Party} \times \mathsf{Party} \times \mathsf{Asset} \to \mathbb{R}}$$

$$\begin{aligned}\mathcal{C}\,[\![\emptyset]\!]_{\gamma,\rho} &= \lambda n.\lambda t.0\\ \mathcal{C}\,[\![e \times c]\!]_{\gamma,\rho} &= \lambda n.\lambda(p,q,a).\mathcal{E}\,[\![e]\!]_{\gamma,\rho} \cdot \mathcal{C}\,[\![c]\!]_{\gamma,\rho}\,(n)(p,q,a)\\ \mathcal{C}\,[\![c_1\ \&\ c_2]\!]_{\gamma,\rho} &= \lambda n.\lambda t.\mathcal{C}\,[\![c_1]\!]_{\gamma,\rho}\,(n)(t) + \mathcal{C}\,[\![c_2]\!]_{\gamma,\rho}\,(n)(t)\\ \mathcal{C}\,[\![d \uparrow c]\!]_{\gamma,\rho} &= delay(d, \mathcal{C}\,[\![c]\!]_{\gamma,\rho}), \quad \text{where}\\ delay(d,f) &= \lambda n.\begin{cases}f(n-d) & \text{if } n \geq d\\ \lambda x.0 & \text{otherwise}\end{cases}\\ \mathcal{C}\,[\![a(p \to q)]\!]_{\gamma,\rho} &= \begin{cases}\lambda n.\lambda t.0 & \text{if } p = q\\ unit_{a,p,q} & \text{otherwise,}\end{cases} \quad \text{where}\\ unit_{a,p,q}(n)(p',q',b) &= \begin{cases}1 & \text{if } b = a, p = p', q = q', n = 0\\ -1 & \text{if } b = a, p = q', q = p', n = 0\\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

$$\mathcal{C}\,[\![\mathbf{let}\ x = e\ \mathbf{in}\ c]\!]_{\gamma,\rho} = \mathcal{C}\,[\![c]\!]_{\gamma[x\mapsto v],\rho}, \text{ where } v = \mathcal{E}\,[\![e]\!]_{\gamma,\rho}$$

$$\mathcal{C}\,[\![\mathbf{if}\ e\ \mathbf{within}\ d\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]_{\gamma,\rho} = iter(d,\rho), \text{ where}$$

$$iter(i,\rho') = \begin{cases}\mathcal{C}\,[\![c_1]\!]_{\gamma,\rho'} & \text{if } \mathcal{E}\,[\![e]\!]_{\gamma,\rho'} = true\\ \mathcal{C}\,[\![c_2]\!]_{\gamma,\rho'} & \text{if } \mathcal{E}\,[\![e]\!]_{\gamma,\rho'} = false \wedge i = 0\\ delay(1, iter(i-1, \rho'/1)) & \text{if } \mathcal{E}\,[\![e]\!]_{\gamma,\rho'} = false \wedge i > 0\end{cases}$$

Figure 4. Denotational semantics of expressions and contracts.

Lemma 1. *For all expressions e, $t \in \mathbb{Z}$, variable assignments γ, and environments ρ, we have that $\mathcal{E}[\![t \Uparrow e]\!]_{\gamma,\rho} = \mathcal{E}[\![e]\!]_{\gamma,\rho/t}$.*

Thus, $\mathbf{acc}(\lambda x.\, e_1, d, e_2)$ is semantically equivalent to

$$e_1[x \mapsto (-1 \Uparrow e_1)[x \mapsto \ldots(-(d-1) \Uparrow e_1)[x \mapsto -d \Uparrow e_2]\ldots]]$$

where we use the notation $e[x \mapsto e']$ to denote the substitution of e' for the free variable x in e.

The semantics of a contract is given by its cash-flow *trace*, a mapping from time into the set Trans of *asset transfers* between two parties:[4]

$$\mathsf{Trans} = \mathsf{Party} \times \mathsf{Party} \times \mathsf{Asset} \to \mathbb{R}$$
$$\mathsf{Trace} = \mathbb{N} \to \mathsf{Trans}$$

Given a contract typing $\Gamma \vdash c : \mathsf{Contr}$, the semantics of c, denoted $\mathcal{C}[\![c]\!]$ is a mapping of type $[\![\Gamma]\!] \times \mathsf{Env} \to \mathsf{Trace}$. Instead of $\mathcal{C}[\![c]\!](\gamma,\rho)$, we write $\mathcal{C}[\![c]\!]_{\gamma,\rho}$. Given a closed contract c (i.e., $\vdash c : Contr$), we simply write $\mathcal{C}[\![c]\!]_\rho$ instead of $\mathcal{C}[\![c]\!]_{\emptyset,\rho}$, where $\emptyset$ denotes the empty variable assignment.

The semantics of a unit transfer $a(p \to q)$ may seem confusing at first, but it reflects the nature of cash-flows: If the two parties p and q 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 to q and one in the converse direction but negative. A consequence of this approach is that for each contract c, we have the following anti-symmetry property:

Lemma 2. *For all γ, ρ, n, p, q, a, we have that*

$$\mathcal{C}[\![c]\!]_{\gamma,\rho}(n)(p,q,a) = -\mathcal{C}[\![c]\!]_{\gamma,\rho}(n)(q,p,a)$$

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 Γ be a typing environment, $\gamma \in [\![\Gamma]\!]$, and $\rho \in \mathsf{Env}$.*

(i) Given $\Gamma \vdash e : \tau$, we have that $\mathcal{E}[\![e]\!]_{\gamma,\rho} \in [\![\tau]\!]$.
(ii) Given $\Gamma \vdash c : \mathsf{Contr}$, we have that $\mathcal{C}[\![e]\!]_{\gamma,\rho} \in \mathsf{Trace}$.

As a corollary, we obtain that each closed contract c yields a total function $\mathcal{C}[\![c]\!] : \mathsf{Env} \to \mathsf{Trace}$.

2.4 Contract Equivalences

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

$$c_1 \equiv_\Gamma c_2 \quad \text{iff} \quad \begin{array}{l} \Gamma \vdash c_1 : \mathsf{Contr}, \Gamma \vdash c_2 : \mathsf{Contr}, \text{ and} \\ \mathcal{C}[\![c_1]\!]_{\gamma,\rho} = \mathcal{C}[\![c_2]\!]_{\gamma,\rho} \text{ for all } \gamma \in [\![\Gamma]\!], \rho \in \mathsf{Env} \end{array}$$

That means, whenever we have that $c_1 \equiv_\Gamma c_2$, then we can replace any occurrence of c_1 in a contract c by c_2 without changing the semantics of c. As a shorthand we use the notation $c_1 \equiv c_2$ iff $\Gamma \vdash c_1 : \mathsf{Contr}, \Gamma \vdash c_2 : \mathsf{Contr}$ implies $c_1 \equiv_\Gamma c_2$ for all Γ.

A number of simple equivalences can be proved easily using the denotational semantics; Figure 5 gives some examples. These equivalences can be used to simplify a given contract, for instance to achieve a normalised format suitable for further processing.

Many of the equivalences in Figure 5 look similar to the axioms of vector spaces. The reason is that the set 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.

$$\begin{array}{rr}
e_1 \times (e_2 \times c) \equiv (e_1 \cdot e_2) \times c & d \uparrow \emptyset \equiv \emptyset \\
d_1 \uparrow (d_2 \uparrow c) \equiv (d_1 + d_2) \uparrow c & r \times \emptyset \equiv \emptyset \\
d \uparrow (c_1 \,\&\, c_2) \equiv (d \uparrow c_1) \,\&\, (d \uparrow c_2) & 0 \times c \equiv \emptyset \\
e \times (c_1 \,\&\, c_2) \equiv (e \times c_1) \,\&\, (e \times c_2) & c \,\&\, \emptyset \equiv c \\
d \uparrow (e \times c) \equiv (d \Uparrow e) \times (d \uparrow c) & c_1 \,\&\, c_2 \equiv c_2 \,\&\, c_1 \\
(e_1 \times c) \,\&\, (e_2 \times c) \equiv (e_1 + e_2) \times c & \\
\multicolumn{2}{c}{d \uparrow \mathbf{if}\ b\ \mathbf{within}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \equiv} \\
\multicolumn{2}{c}{\mathbf{if}\ d \Uparrow b\ \mathbf{within}\ e\ \mathbf{then}\ d \uparrow c_1\ \mathbf{else}\ d \uparrow c_2}
\end{array}$$

Figure 5. Some contract equivalences.

2.5 Observing the Passage of Time

In a contract of the form $\mathbf{if}\ b\ \mathbf{within}\ d\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$, we know how much time has passed when we enter subcontract c_2, namely d days. This we do not know for the subcontract c_1; we only know that between 0 and d days have passed. However, the contract language's *let* bindings provide a mechanism to observe the passage of time also when entering the subcontract c_1. To this end, we assume an observable with label $time \in \mathsf{Label}_{\mathsf{Real}}$, whose value is the "current time". We can then modify the above contract such that we can observe how much time has passed before b became true and the subcontract c_1 was entered:

$$\mathbf{let}\ y = \mathbf{obs}(time, 0)$$
$$\mathbf{in\ if}\ b\ \mathbf{within}\ d\ \mathbf{then\ let}\ x = \mathbf{obs}(time, 0) - y\ \mathbf{in}\ c_1\ \mathbf{else}\ c_2$$

The variable x of type Real scopes over the contract c_1 and denotes the time that has passed between entering the whole *if-within* contract and entering the subcontract c_1.

Because the above construction is useful and common when formulating contracts, we give it the following shorthand notation:

$$\mathbf{if}\ b\ \mathbf{within}\ d\ \mathbf{then}\ x.\ c_1\ \mathbf{else}\ c_2$$

The variable y is not explicitly mentioned in this shorthand notation and is assumed to be an arbitrary fresh variable.

We can use this construction, for example, to express a *callable bond*, that is, a bond where the issuer may decide to redeem the bond prematurely. The amount to be paid to the holder of the bond may depend on the time passed before the issuer decided to call the bond.

$$\begin{array}{l}
\mathbf{if}\ \mathbf{obs}(X \text{ calls bond}, 0)\ \mathbf{within}\ 30 \\
\mathbf{then}\ x.\ ((30 - x) + 100) \times \mathsf{USD}(X \to Y) \\
\mathbf{else}\ 100 \times \mathsf{USD}(X \to Y)
\end{array}$$

For the sake of presentation, the above contract is rather simplistic, but illustrates the underlying concept: the issuer of the bond, party X, can call the bond at any time, with the penalty of paying more to the holder of the bond, party Y, depending on the time left until maturity $(30 - x)$.

We still need to formally define the semantics of the $time$ observable. We cannot define the value of the $time$ observable in absolute terms, since our contract language is deliberately constraint to relative time. Consequently, the $time$ observable is defined as a relative concept: each environment ρ satisfies the equation

$$\rho(time, t + t') = \rho(time, t) + t' \qquad \text{for all } t, t' \in \mathbb{Z}$$

Concretely, the above invariant can be achieved by using environments ρ at the top level that satisfy $\rho(time, t) = t$.

[4] The informed reader might notice that this semantics is bound to *adding* all transfers between two parties on one particular day. This so-called "netting" is used throughout in our model. In real-world financial contracts, parties would explicitly agree on netting, or otherwise handle cash-flows from different contracts as separate entities.

2.6 Calendars

A notoriously thorny issue for formal contract languages in the financial domain is the calendar, which is used for expressing and referring to properties about dates such as holidays and business days. Fortunately, observables provide an elegant interface for calendric properties.

To illustrate this, we consider a Boolean observable that is true whenever we have a business day. However, "business day" is not an absolute concept and varies by region. Therefore, we assume that for each currency a, we have a label $\mathit{business}(a) \in \mathsf{Label}_{\mathsf{Bool}}$ denoting whether a given day is a business day for currency a.

In Section 2.1, we considered various examples for foreign exchange options, where the actual exchange was expressed by a contract c such as

$$c = 100 \times (\mathsf{USD}(Y \to X) \,\&\, 6.5 \times \mathsf{DKK}(X \to Y))$$

In reality, however, such exchanges cannot happen on any arbitrary date, but only on days that are business days for all involved currencies. Therefore, real contracts typically state that the exchange must be executed on the first day that is a business day for all involved currencies. With the help of the $\mathit{business}$ observable, this refinement is expressed by the following contract c':

$$c' = \mathbf{if}\ \mathbf{obs}(\mathit{business}(\mathsf{USD}), 0) \wedge \mathbf{obs}(\mathit{business}(\mathsf{DKK}), 0)\ \mathbf{within}\ 365\ \mathbf{then}\ c\ \mathbf{else}\ c$$

The contract c' states that we enter the foreign exchange contract c on the first day that is a business day for both USD and DKK, or in a year at the latest. Thus, a more realistic version of the contract in Example 2 is the following:

$$\mathbf{if}\ \mathbf{obs}(X \text{ exercises option}, 0)\ \mathbf{within}\ 90\ \mathbf{then}\ c'\ \mathbf{else}\ \emptyset$$

Instead of the simple foreign exchange contract c, the above contract uses the refined version c'.

3. Temporal Properties of Contracts

With the denotational semantics of contracts at hand, we can characterise a number of temporal properties that 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*.

In principle, both the causality property and the contract horizon can be effectively computed via the decidability of the first order theory of real closed fields. However, this approach would result in computationally very expensive procedures (even more so since the **acc** combinator and the *if-within* have to be unrolled). But more importantly, minor additions to the expression language such as exponentiation and logarithm, which are common in finance, break the decidability. Therefore, we will devise sound approximations of these temporal properties. Moreover, as we shall see below, the approximation of causality via a type system provides additional benefits—most importantly a computational adequacy result for our reduction semantics in Section 4.2.

3.1 Contract Horizon

We define the *horizon* $h \in \mathbb{N}$ of a closed contract c 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

$$\mathcal{C}\left[\!\left[c\right]\!\right]_{\rho}(i)(x) = 0 \quad \text{for all } \rho \in \mathsf{Env}, i \geq h, \text{ and } x$$

In other words, after h days, the cash-flow for the contract c remains zero, for any environment ρ. Notice that since c is closed, that is, $\vdash C : \mathsf{Contr}$, we know that $\mathcal{C}\left[\!\left[c\right]\!\right]_{\rho}(i)$ is defined for any ρ and i.

$$\begin{aligned}
\mathrm{HOR}(\emptyset) &= 0 & \mathrm{HOR}(e \times c) &= \mathrm{HOR}(c)\\
\mathrm{HOR}(a(p \to q)) &= 1 & \mathrm{HOR}(d \uparrow c) &= d \oplus \mathrm{HOR}(c)\\
\mathrm{HOR}(\mathbf{let}\ x = e\ \mathbf{in}\ c) &= \mathrm{HOR}(c)\\
\mathrm{HOR}(c_1 \,\&\, c_2) &= \max(\mathrm{HOR}(c_1), \mathrm{HOR}(c_2))\\
\mathrm{HOR}\begin{pmatrix}\mathbf{if}\ e\ \mathbf{within}\ d\\ \mathbf{then}\ c_1 \mathbf{else}\ c_2\end{pmatrix} &= d \oplus \max(\mathrm{HOR}(c_1), \mathrm{HOR}(c_2))
\end{aligned}$$

where

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

Figure 6. Symbolic horizon.

By dropping the minimality requirement, we can devise a simple, sound approximation of the horizon, which is given in Figure 6. We can show that the semantic contract horizon is never greater than the symbolic horizon computed by HOR:

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

3.2 Contract Causality

At the moment, the contract language allows us to write 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, we may stipulate a transfer to be executed today using the foreign exchange rate of tomorrow:

$$\mathbf{obs}(\mathsf{FX}(\mathsf{USD}, \mathsf{DKK}), 1) \times \mathsf{DKK}(X \to Y)$$

Using the denotational semantics, we can give a precise definition of *causality*. Given $t \in \mathbb{Z}$, we define an equivalence relation $=_t$ on Env that intuitively expresses that two environments agree until (and including) time t. We define that $\rho_1 =_t \rho_2$ iff $s \leq t$ implies $\rho_1(l, s) = \rho_2(l, s)$, for all $l \in \mathsf{Label}$, 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 $\rho_1, \rho_2 \in \mathsf{Env}$, we have that $\rho_1 =_t \rho_2$ implies $\mathcal{C}\left[\!\left[c\right]\!\right]_{\rho_1}(t) = \mathcal{C}\left[\!\left[c\right]\!\right]_{\rho_2}(t)$. That is, the cash-flows at any time t do not depend on observables and decisions after t.

As mentioned earlier, contract causality is in principle decidable, but it is computationally expensive, and decidability is easily lost with minor additions to the expression language. Moreover, causality is not a compositional property; a contract may be causal even though a subcontract is not causal. For instance, any contract of the form $c \,\&\, (-1 \times c)$ is trivially causal since it is equivalent to $\emptyset$; but c may well be non-causal. Compositionality is important for the reduction semantics as we shall see in Section 4.2. Therefore, we develop a compositional, conservative approximation of causality. The simplest such approximation is to require that for every subexpression of the form $\mathbf{obs}(l, t)$, we have that $t \leq 0$. 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 causal contract that cannot be transformed into an equivalent contract that is obviously causal. For example, the following contract is causal but not obviously causal:

$$\mathbf{obs}(\mathsf{FX}(\mathsf{USD}, \mathsf{DKK}), 1) \times 1 \uparrow \mathsf{DKK}(X \to Y)$$

However, the above contract is equivalent to the following obviously causal contract (cf. Figure 5):

$$1 \uparrow \mathbf{obs}(\mathsf{FX}(\mathsf{USD}, \mathsf{DKK}), 0) \times \mathsf{DKK}(X \to Y)$$

A more realistic example is the following chooser option, where the buyer X may choose, in 30 days, whether to have a (European)

$\boxed{\Gamma \Vdash e : \tau^t}$ where $t \in \mathbb{Z}_{-\infty}$

$$\frac{}{\Gamma \Vdash r : \mathsf{Real}^t} \qquad \frac{}{\Gamma \Vdash r : \mathsf{Bool}^t} \qquad \frac{l \in \mathsf{Label}_\tau \quad t \le t'}{\Gamma \Vdash \mathbf{obs}(l,t) : \tau^{t'}}$$

$$\frac{x : \tau^t \in \Gamma \quad t \le t'}{\Gamma \Vdash x : \tau^{t'}} \qquad \frac{\vdash op : \tau_1 \times \cdots \times \tau_n \to \tau \quad \Gamma \Vdash e_i : \tau_i^t}{\Gamma \Vdash op(e_1, \ldots, e_n) : \tau^t}$$

$$\frac{\Gamma, x : \tau^{-\infty} \Vdash e_1 : \tau^t \quad \Gamma^{+d} \Vdash e_2 : \tau^{t+d}}{\Gamma \Vdash \mathbf{acc}(\lambda x.\, e_1, d, e_2) : \tau^t}$$

$\boxed{\Gamma \Vdash c : \mathsf{Contr}^t}$ where $t \in \mathbb{Z}_{-\infty}$

$$\frac{\Gamma^{-d} \Vdash c : \mathsf{Contr}^{t-d}}{\Gamma \Vdash d \uparrow c : \mathsf{Contr}^t} \qquad \frac{t \le 0}{\Gamma \Vdash a(p \to q) : \mathsf{Contr}^t}$$

$$\frac{}{\Gamma \Vdash \emptyset : \mathsf{Contr}^t} \qquad \frac{\Gamma \Vdash e : \mathsf{Real}^{t'} \quad \Gamma \Vdash c : \mathsf{Contr}^{t'} \quad t \le t'}{\Gamma \Vdash e \times c : \mathsf{Contr}^t}$$

$$\frac{\Gamma \Vdash c_i : \mathsf{Contr}^t}{\Gamma \Vdash c_1 \,\&\, c_2 : \mathsf{Contr}^t} \qquad \frac{\Gamma \Vdash e : \tau^s \quad \Gamma, x : \tau^s \Vdash c : \mathsf{Contr}^t}{\Gamma \Vdash \mathbf{let}\ x = e\ \mathbf{in}\ c : \mathsf{Contr}^t}$$

$$\frac{\Gamma \Vdash e : \mathsf{Bool}^0 \quad \Gamma \Vdash c_1 : \mathsf{Contr}^t \quad \Gamma^{-d} \Vdash c_2 : \mathsf{Contr}^{t-d}}{\Gamma \Vdash \mathbf{if}\ e\ \mathbf{within}\ d\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \mathsf{Contr}^t}$$

Figure 7. Time-indexed type system.

$\boxed{\Gamma \vdash e : \tau^t}$ where $t \in \mathbb{Z}_{-\infty}$

$$\frac{}{\Gamma \vdash b : \mathsf{Bool}^{-\infty}} \qquad \frac{}{\Gamma \vdash r : \mathsf{Real}^{-\infty}} \qquad \frac{l \in \mathsf{Label}_\tau}{\Gamma \vdash \mathbf{obs}(l,t) : \tau^t}$$

$$\frac{x : \tau^t \in \Gamma}{\Gamma \vdash x : \tau^t} \qquad \frac{\Gamma \vdash e_i : \tau_i^{t_i} \quad \vdash op : \tau_1 \times \cdots \times \tau_n \to \tau}{\Gamma \vdash op(e_1, \ldots, e_n) : \tau^{\max_i t_i}}$$

$$\frac{\Gamma, x : \tau^{-\infty} \vdash e_1 : \tau^{t_1} \quad \Gamma^{+d} \vdash e_2 : \tau^{t_2}}{\Gamma \vdash \mathbf{acc}(\lambda x.\, e_1, d, e_2) : \tau^{\max(t_1, t_2 - d)}}$$

$\boxed{\Gamma \vdash c : \mathsf{Contr}^t}$ where $t \in \mathbb{Z}_{\pm\infty}$

$$\frac{\Gamma^{-d} \vdash c : \mathsf{Contr}^t}{\Gamma \vdash d \uparrow c : \mathsf{Contr}^{t+d}} \qquad \frac{}{\Gamma \vdash a(p \to q) : \mathsf{Contr}^0}$$

$$\frac{}{\Gamma \vdash \emptyset : \mathsf{Contr}^{+\infty}} \qquad \frac{\Gamma \vdash e : \mathsf{Real}^{t'} \quad \Gamma \vdash c : \mathsf{Contr}^t \quad t' \le t}{\Gamma \vdash e \times c : \mathsf{Contr}^t}$$

$$\frac{\Gamma \vdash c_i : \mathsf{Contr}^{t_i}}{\Gamma \vdash c_1 \,\&\, c_2 : \mathsf{Contr}^{\min_i t_i}} \qquad \frac{\Gamma \vdash e : \tau^s \quad \Gamma, x : \tau^s \vdash c : \mathsf{Contr}^t}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ c : \mathsf{Contr}^t}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool}^t \quad t \le 0 \quad \Gamma \vdash c_1 : \mathsf{Contr}^{t_1} \quad \Gamma^{-d} \vdash c_2 : \mathsf{Contr}^{t_2}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{within}\ d\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \mathsf{Contr}^{\min(t_1, t_2 + d)}}$$

Figure 8. Type inference algorithm.

call or put option. The buyer X may then, 30 days later, exercise the option. We formulate the contract in terms of the payout with respect to a given *strike* price:

$$\begin{aligned}
&\mathbf{let}\ \mathit{price} &&= \mathbf{obs}(\mathsf{FX}(\mathsf{DKK}, \mathsf{USD}), 60)\ \mathbf{in}\\
&\mathbf{let}\ \mathit{payout} &&= \mathbf{if}\ \mathbf{obs}(X \text{ chooses call option}, 30)\\
&&&\quad \mathbf{then}\ \mathit{max}(\mathit{price} - \mathit{strike}, 0)\\
&&&\quad \mathbf{else}\ \mathit{max}(\mathit{strike} - \mathit{price}, 0)\\
&\mathbf{in}\ 60 \uparrow (\mathit{payout} \times \mathsf{DKK}(Y \to X))
\end{aligned}$$

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. Moreover, it is very fragile as it is not necessarily preserved by equivalence preserving contract transformations. For example, applying any of the equivalences from Figure 5 involving promotion of expressions ($d \Uparrow e$) from left-to right may destroy obvious causality. To address these problems, we refine the typing rules for contracts and expressions by indexing types with time offsets. The intuition of these time indices is the following: If an expression e has type τ^t, then 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 Contr^t, then c makes no cash-flow stipulations strictly before t.

Time indices t range over the set $\mathbb{Z}_{-\infty} = \mathbb{Z} \cup \{-\infty\}$, that is, we assume a time $-\infty$ that is before any other time $t \in \mathbb{Z}$. We also assume a total order $\le$ on $\mathbb{Z}_{-\infty}$, which is the natural order on $\mathbb{Z}$ extended by $-\infty \le t$ for all $t \in \mathbb{Z}_{-\infty}$. Moreover, we define the addition $t + d$ of a time $t \in \mathbb{Z}_{-\infty}$ by a number $d \in \mathbb{Z}$: if $t \in \mathbb{Z}$, then the addition is just ordinary 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. To distinguish the refined type system from the simple type system we use the notation $\Vdash$ instead of $\vdash$. The typing rules use *timed type environments*, which map variables to time-indexed types instead of plain types. Moreover, the typing rules use the notation Γ^{+d} to denote the timed type environment that is obtained from Γ by adding d to all time indices, that is, $x : \tau^{t+d} \in \Gamma^{+d}$ iff $x : \tau^t \in \Gamma$. The notation Γ^{-d} is defined accordingly: $x : \tau^{t-d} \in \Gamma^{-d}$ iff $x : \tau^t \in \Gamma$.

The typing rules provide some insight into the temporal properties of expression and contract constructs. Starting with the typing of expressions, we can see that constants are available at any time and thus have an arbitrary time index; observables at time t are available at any time after t; and operators op have no temporal interaction. The typing rule for **acc** can be difficult to read at first, but it directly reflects its temporal behaviour: e_2 is evaluated d days in the past. This is reflected by the shift of the time indices d days into the future, which means variables become available d days later, but also that e_2 only needs be become available d days later. The other component e_1 is evaluated from d days in the past until the present, hence there is no shift in the time indices. The time index on the accumulation variable x indicates that there are no temporal restrictions on x. Alternatively, we could have avoided the additional $-\infty$ time index and reformulated this typing rule to use any time index $t' \in \mathbb{Z}$ instead of $-\infty$. However, the present approach simplifies the proofs.

Turning to the contract typing rules, we see that $\emptyset$ stipulates no cash-flows, and that $a(p \to q)$ stipulates an immediate cash-flow. The typing for $\&$ indicates that it has no temporal interaction, and the typing for $\uparrow$ directly indicates the temporal shift expressed by $\uparrow$. The typing rule for let binding is also without surprises. It expresses the fact that *let* takes a snapshot in time. The rule for $\times$ is a crucial one as it connects the expression language with the contract language. It is arguably the most important typing rule as it expresses the essential property for causality: an expression e can only meaningfully scale a contract c if e is available at some time t' and c makes no stipulations strictly before t'. The additional inequality $t \le t'$ seems arbitrary and superfluous, but is essential as we will argue at the end of this section. Finally, the typing for *if-within* is somewhat dual to the typing of **acc**: instead of coming

from the past like **acc**, *if-within* moves into the future. Hence, the typing of c_2 is shifted d days into the past. The typing of the predicate e expresses that we need to know its value immediately to decide whether one of the two subcontracts is entered.

The typing rules in Figure 7 refine the original simple typing rules in Figure 2: well-typing ($\Vdash$) implies simple well-typing ($\vdash$):

Proposition 5. *Let Γ be a timed type environment and $|\Gamma|$ a type environment such that for each x and τ there is some t such that $x : \tau^t \in \Gamma$ iff $x : \tau \in |\Gamma|$. Then we have*

(i) $\Gamma \Vdash e : \tau^t$ *implies* $|\Gamma| \vdash e : \tau$, *and*
(ii) $\Gamma \Vdash c : \mathsf{Contr}^t$ *implies* $|\Gamma| \vdash c : \mathsf{Contr}$.

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

Theorem 6. *If $\Vdash c : \mathsf{Contr}^t$, 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 $\mathbb{Z}_{-\infty}$ defined as follows:

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

An essential property of expression and contract typing is that both are closed under subtyping, albeit in opposite directions:

Lemma 7.

(i) If $\Gamma \Vdash e : \tau^t$, then $\Gamma \Vdash e : \tau^s$ for all $s \geq t$.
(ii) If $\Gamma \Vdash c : \mathsf{Contr}^t$, then $\Gamma \Vdash c : \mathsf{Contr}^s$ 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 $\mathbb{Z}_{-\infty}$ with an additional maximal element $+\infty$, we also obtain that well-typed contracts have maximal types. This property allows us to devise a simple type inference algorithm. For the sake of clarity, we present the type inference algorithm in the form of syntax-directed typing rules, which are shown in Figure 8. In contrast to the typing judgement $\Vdash$, the syntax-directed judgement $\mapsto$ assigns contracts (and expressions) at most one type—namely the maximal (resp. minimal) type according to the $\Vdash$ judgement. Notice that contracts are typed with the extended set of time indices set $\mathbb{Z}_{\pm\infty} = \mathbb{Z} \cup \{-\infty, +\infty\}$. The ordering $\leq$ is extended to $\mathbb{Z}_{\pm\infty}$ in the obvious way. Moreover, we define addition of elements $t \in \mathbb{Z}_{\pm\infty}$ with numbers $d \in \mathbb{Z}$ by $-\infty + d = -\infty$, $+\infty + d = +\infty$, and otherwise as ordinary addition in $\mathbb{Z}$.

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

Theorem 8 (Type inference is sound and complete)**.**

(i) If $\Gamma \mapsto c : \mathsf{Contr}^t$, then $\Gamma \Vdash c : \mathsf{Contr}^s$ for all $s \leq t$.
(ii) If $\Gamma \Vdash c : \mathsf{Contr}^s$, then $\Gamma \mapsto c : \mathsf{Contr}^t$ for a unique $t \geq s$.

Thus, according to Theorem 6, we obtain that if type inference returns a type for a contract c, then c is causal.

Corollary 9. *If $\emptyset \mapsto c : \mathsf{Contr}^t$ for some t, 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 7. This property will also be important in the next section where we discuss contract transformations. Lemma 7 is crucial for showing that well-typing 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 $t = t'$. 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 $t \leq t'$ in the typing rule for $\times$ *decouples* the time indices of contract and expression types as much as possible while still enforcing causality. This decoupling is necessary due to the different interpretation of time indices for expression types compared to contract types. This difference in interpretation also manifests itself in the difference in the subtyping behaviour described in Lemma 7.

Apart from this technical problem, changing the typing of $\times$ by requiring $t' = t$, also has the severe practical consequence that fewer contracts would be typeable. Among such contracts that would not be typeable anymore are many realistic contracts that one would expect to be typeable. For example, the contract

$$(\mathbf{obs}(l, 0) \times a(p \to q)) \,\&(\mathbf{obs}(l, 1) \times 1 \uparrow a(p \to q))$$

would not be typeable anymore.

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 computationally adequate for the denotational semantics presented in Section 2.3.

These contract transformations are based on external knowledge provided by a *partial* external environment, that is, 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 Env_P, a superset of Env:

$$\mathsf{Env}_\mathsf{P} = \mathsf{Label}_\tau \times \mathbb{Z} \rightharpoonup [\![\tau]\!]$$

A contract c can be transformed based on a partial environment $\rho \in \mathsf{Env}_\mathsf{P}$ that encodes the available knowledge about observables and decisions that may 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. The primary application of specialisation is the instantiation of a contract to a concrete starting time. A contract may refer to values of observables before the starting time of the contract. Specialisation allows us to instantiate such contracts with these known values of observables. Beyond this simple application, specialisation of expressions is also a crucial ingredient for the reduction semantics in Section 4.2.

Before we can define specialisation more formally, we need to introduce some terminology: An environment $\rho' \in \mathsf{Env}$ extends a partial environment $\rho \in \mathsf{Env}_\mathsf{P}$ iff $\rho(l, t) = \rho'(l, t)$ for all $(l, t) \in \mathsf{dom}(\rho)$. Furthermore, we define the set of partial variable assignments $[\![\Gamma]\!]_\mathsf{P}$ for a type environment Γ as the set of all partial mappings γ from variable names into $\mathbb{R} \cup \mathbb{B}$ such that $\gamma(x) \in [\![\tau]\!]$ if $x : \tau \in \Gamma$. That is, a partial variable assignment may not assign a value to all variables in Γ. A variable assignment $\gamma' \in [\![\Gamma]\!]$ extends a partial variable assignment $\gamma \in [\![\Gamma]\!]_\mathsf{P}$ iff $\gamma(x) = \gamma'(x)$ for all $x \in \mathsf{dom}(\gamma)$.

Given $\Gamma \vdash c_1 : \mathsf{Contr}$, $\Gamma \vdash c_2 : \mathsf{Contr}$, $\gamma \in [\![\Gamma]\!]_\mathsf{P}$, and $\rho \in \mathsf{Env}_\mathsf{P}$, we say that c_1 and c_2 are γ, ρ-equivalent, written $c_1 \equiv_{\gamma,\rho} c_2$,

$$\mathsf{sp_C}(c,\gamma,\rho) = \begin{cases} c & \text{if } c = \emptyset \vee c = a(p \to q) \\ \underline{\mathbf{let}}\ x = e' & \text{if } c = (\mathbf{let}\ x = e\ \mathbf{in}\ c') \wedge \\ \underline{\mathbf{in}}\ \mathsf{sp_C}(c',\gamma',\rho) & \quad e' = \mathsf{sp_E}(e,\gamma,\rho) \wedge \\ & \quad \gamma' = \begin{cases} \gamma[x \mapsto e'] & \text{if } e' \in \mathbb{R} \cup \mathbb{B} \\ \gamma & \text{otherwise} \end{cases} \\ \mathsf{sp_E}(e,\gamma,\rho) \mathbin{\underline{\times}} \mathsf{sp_C}(c',\gamma,\rho) & \text{if } c = e \times c' \\ l \mathbin{\underline{\uparrow}} \mathsf{sp_C}(c',\gamma,\rho/d) & \text{if } c = d \uparrow c' \\ \mathsf{sp_C}(c_1,\gamma,\rho) \mathbin{\underline{\&}} \mathsf{sp_C}(c_2,\gamma,\rho) & \text{if } c = c_1 \,\&\, c_2 \\ \mathsf{trav}(\gamma,\rho,b,c_1,c_2,0,d,c) & \text{if } c = \mathbf{if}\ b\ \mathbf{within}\ d \\ & \quad \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \end{cases}$$

$$\mathsf{trav}(\gamma,\rho,b,c_1,c_2,d',d,c) = \begin{cases} d' \mathbin{\underline{\uparrow}} \mathsf{sp_C}(c_1,\gamma,\rho) & \text{if } \mathsf{sp_E}(b,\gamma,\rho) = \mathit{true} \\ d' \mathbin{\underline{\uparrow}} \mathsf{sp_C}(c_2,\gamma,\rho) & \text{if } \mathsf{sp_E}(b,\gamma,\rho) = \mathit{false} \wedge d = 0 \\ \mathsf{trav}(\gamma,\rho/1,b,c_1,c_2, & \text{if } \mathsf{sp_E}(b,\gamma,\rho) = \mathit{false} \wedge d > 0 \\ \quad d'+1, d-1, c) & \\ c & \text{otherwise} \end{cases}$$

Figure 9. Contract specialisation function $\mathsf{sp_C}$.

iff $\mathcal{C}[\![c_1]\!]_{\gamma',\rho'} = \mathcal{C}[\![c_2]\!]_{\gamma',\rho'}$ for all $\gamma' \in [\![\Gamma]\!]$ and $\rho' \in \mathsf{Env}$ that extend γ and ρ, respectively. The specialisation function $\mathsf{sp_C}$ takes an external environment ρ and a variable assignment γ, and transforms a contract c into a contract c' with $c \equiv_{\gamma,\rho} c'$.

In order to implement such a function $\mathsf{sp_C}$, we also need a corresponding specialisation function $\mathsf{sp_E}$ on expressions. To this end, we define a corresponding notion of γ,ρ-equivalence on expressions: Given $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$, $\gamma \in [\![\Gamma]\!]_\mathsf{P}$, and $\rho \in \mathsf{Env_P}$, we say that e_1 and e_2 are γ,ρ-equivalent, written $e_1 \equiv_{\gamma,\rho} e_2$, iff $\mathcal{E}[\![e_1]\!]_{\gamma',\rho'} = \mathcal{E}[\![e_2]\!]_{\gamma',\rho'}$ for all $\gamma' \in [\![\Gamma]\!]$ and $\rho' \in \mathsf{Env}$ that extend γ and ρ, respectively.

The definition of $\mathsf{sp_C}$ is given in Figure 9. We have elided the definition of $\mathsf{sp_E}$, which is straightforward and can be found in the Coq source files associated with this paper. The definition of $\mathsf{sp_C}$ uses underlined versions of $\times$, $\uparrow$, $\&$ and $\mathbf{let}$. These are *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, $\underline{\times}$ is defined as follows:

$$e \mathbin{\underline{\times}} c = \begin{cases} c & \text{if } e = 1 \\ \emptyset & \text{if } e = 0 \vee c = \emptyset \\ e \times c & \text{otherwise} \end{cases}$$

The other smart constructors work similarly. In particular, $\underline{\mathbf{let}}\ x = e\ \underline{\mathbf{in}}\ c$ is equal to c if there is no free occurrence of x in c.

Moreover, $\mathsf{sp_C}$ uses an auxiliary function trav, also defined in Figure 9, which tries to simplify the *if-within* construct.

Example 5. Reconsider the CDS contract from Example 4. We want to see what happens if party X defaults. To this end, we define the partial environment ρ such that $\rho(X\,\mathsf{defaults}, i) = \mathit{true}$ if $i = 15$, $\rho(X\,\mathsf{defaults}, i) = \mathit{false}$ if $i \neq 15$, and otherwise ρ is undefined. In other words, we assume that party X defaults after 15 days: with this input, $\mathsf{sp_C}$ transforms the contract into

$$(10 \times \mathsf{DKK}(Y \to Z)) \,\&\, (15 \uparrow 900 \times \mathsf{DKK}(Z \to Y))$$

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, that is, if $\rho(X\,\mathsf{defaults}, i) = \mathit{false}$ for all i, then $\mathsf{sp_C}$ transforms the contract into

$$(30 \uparrow 1000 \times \mathsf{DKK}(X \to Y)) \,\&\, (10 \times \mathsf{DKK}(Y \to Z))$$

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

We can show that $\mathsf{sp_C}$ and $\mathsf{sp_E}$ indeed implement specialisation of contracts and expressions, respectively:

Theorem 10. *Let Γ be a typing environment, $\gamma \in [\![\Gamma]\!]_\mathsf{P}$, and $\rho \in \mathsf{Env_P}$.*

(i) Given $\Gamma \vdash e : \tau$, we have that $\mathsf{sp_E}(e,\gamma,\rho) \equiv_{\gamma,\rho} e$.
(ii) Given $\Gamma \vdash c : \mathsf{Contr}$, we have that $\mathsf{sp_C}(c,\gamma,\rho) \equiv_{\gamma,\rho} c$.

In particular, we have that specialisation preserves the typing, that is, $\Gamma \vdash c : \mathsf{Contr}$ implies that $\Gamma \vdash \mathsf{sp_C}(c,\gamma,\rho) : \mathsf{Contr}$, 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 [2], which *advances* a contract by one time unit. This reduction semantics allows us to modify a contract according to the passage of time and the knowledge of observables that gradually becomes available. In addition, the reduction semantics will tell us the concrete asset transfers that have to occur according to the contract.

We write $c \overset{T}{\Longrightarrow}_\rho c'$, to denote that c is advanced to c' in the partial environment $\rho \in \mathsf{Env_P}$, where $T \in \mathsf{Trans}$ 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 ρ). In order to define $\overset{T}{\Longrightarrow}_\rho$, 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 γ. In sum, the reduction semantics is a relation written as $c \overset{T}{\Longrightarrow}_{\gamma,\rho} c'$, and we use the notation $c \overset{T}{\Longrightarrow}_\rho c'$ for the special case that γ is the empty variable assignment. The full definition of the reduction semantics is given in Figure 10.

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 $\rho \in \mathsf{Env_P}$ is *historically complete*. By this we mean that $\rho(l,i)$ is defined for all $l \in \mathsf{Label}$ 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 (Computational adequacy of $\overset{T}{\Longrightarrow}_\rho$)**.** *Let $\Vdash c : \mathsf{Contr}^t$ and $\rho \in \mathsf{Env_P}$.*

(i) If $c \overset{T}{\Longrightarrow}_\rho c'$, then the following holds for all ρ' that extend ρ:
  (a) $\mathcal{C}[\![c]\!]_{\rho'}(0) = T$, and
  (b) $\mathcal{C}[\![c]\!]_{\rho'}(i+1) = \mathcal{C}[\![c']\!]_{\rho'/1}(i)$ for all $i \in \mathbb{N}$,
(ii) If $c \overset{T}{\Longrightarrow}_\rho c'$, then $\Vdash c' : \mathsf{Contr}^{t-1}$.
(iii) If ρ is historically complete, then there is a unique c' such that $c \overset{T}{\Longrightarrow}_\rho c'$ and $T = \mathcal{C}[\![c]\!]_\rho(0)$.

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 $\rho \in \mathsf{Env}$, we can conclude that any well-typed contract yields an infinite reduction sequence, which reveals the contract's complete denotational semantics:

$$c \xrightarrow{\mathcal{C}[\![c]\!]_\rho(0)}_\rho c_0 \xrightarrow{\mathcal{C}[\![c]\!]_\rho(1)}_{\rho/1} c_1 \xrightarrow{\mathcal{C}[\![c]\!]_\rho(2)}_{\rho/2} \quad \ldots$$

$$\frac{c \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}{0 \uparrow c \overset{T}{\Longrightarrow}_{\gamma,\rho} c'} \qquad \frac{}{\emptyset \overset{T_0}{\Longrightarrow}_{\gamma,\rho} \emptyset} \qquad \frac{}{a(p \to q) \overset{T_{p,q,a}}{\Longrightarrow}_{\gamma,\rho} \emptyset}$$

$$\frac{d > 0}{d \uparrow c \overset{T_0}{\Longrightarrow}_{\rho} d-1 \underline{\uparrow} c} \qquad \frac{c \overset{T}{\Longrightarrow}_{\gamma,\rho} c' \quad r = \mathsf{sp_E}(e,\gamma,\rho) \quad r \in \mathbb{R}}{e \times c \overset{r*T}{\Longrightarrow}_{\gamma,\rho} r \underline{\times} c'}$$

$$\frac{c_i \overset{T_i}{\Longrightarrow}_{\gamma,\rho} c_i}{c_1 \,\&\, c_2 \overset{T_1+T_2}{\Longrightarrow}_{\gamma,\rho} c_1 \,\underline{\&}\, c_2} \qquad \frac{c \overset{T_0}{\Longrightarrow}_{\gamma,\rho} c' \quad e' = \mathsf{sp_E}(e,\gamma,\rho)}{e \times c \overset{T_0}{\Longrightarrow}_{\gamma,\rho} (-1 \Uparrow e') \underline{\times} c'}$$

$$\frac{\mathsf{sp_E}(e,\gamma,\rho) = e' \qquad c \overset{T}{\Longrightarrow}_{\gamma',\rho} c' \qquad \gamma' = \begin{cases} \gamma[x \mapsto e'] & \text{if } e' \in \mathbb{R} \cup \mathbb{B} \\ \gamma & \text{otherwise} \end{cases}}{\mathbf{let}\ x = e\ \mathbf{in}\ c \overset{T}{\Longrightarrow}_{\gamma,\rho} \underline{\mathbf{let}}\ x = -1 \Uparrow e'\ \underline{\mathbf{in}}\ c}$$

$$\frac{\mathsf{sp_E}(e,\gamma,\rho) = \mathit{false} \quad c_2 \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}{\mathbf{if}\ e\ \mathbf{within}\ 0\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}$$

$$\frac{\mathsf{sp_E}(e,\gamma,\rho) = \mathit{true} \quad c_2 \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}{\mathbf{if}\ e\ \mathbf{within}\ d\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}$$

$$\frac{\mathsf{sp_E}(e,\gamma,\rho) = \mathit{false} \quad d > 0}{\begin{array}{l}\mathbf{if}\ e\ \mathbf{within}\ d \\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\end{array} \overset{T_0}{\Longrightarrow}_{\gamma,\rho} \begin{array}{l}\mathbf{if}\ e\ \mathbf{within}\ d-1 \\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\end{array}}$$

where $\quad T_0 = \lambda t.0 \qquad r * T = \lambda t.r \cdot T(t)$

$$T_1 + T_2 = \lambda t.T_1(t) + T_2(t)$$

$$T_{p,q,a} = \lambda(p',q',a').\begin{cases} 1 & \text{if } (p',q',a') = (p,q,a) \\ -1 & \text{if } (p',q',a') = (q,p,a) \\ 0 & \text{otherwise} \end{cases}$$

Figure 10. Contract reduction semantics.

Because contracts have a finite horizon (see Section 3.1), we know that there is some $n \in \mathbb{N}$ such that $c_i \equiv \emptyset$ 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 $c \overset{T}{\Longrightarrow}_{\rho} c'$, given that ρ is only historically complete. For instance, given the contract $c_1 = \mathbf{obs}(l,1) \times a(p \to q)$, 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 $c_2 = \mathbf{obs}(l,1) \times a(q \to p)$ is not causal. However, the contract $c_1 \,\&\, c_2$ is equivalent to $\emptyset$ and thus is causal. Therefore, being able to capture a conservative notion of causality that is compositional, for example, in the form of well-typing, is crucial for the computational adequacy of the reduction semantics.

A central lemma for proving property (iii) of Theorem 11, is that the specialisation function $\mathsf{sp_E}$ 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 $\Gamma \Vdash e : \tau^t$, then $\mathsf{sp_E}(e,\gamma,\rho) \in [\![\tau]\!]$, given that $\rho \in \mathsf{Env_P}$ and $\gamma \in [\![\Gamma]\!]_{\mathsf{P}}$ are sufficiently defined. The "sufficiently defined" condition is dependent on the typing of e. It requires that $\gamma(x)$ is defined whenever $x : \sigma^s \in \Gamma$ and $s \leq t$, and that $\rho(l,i)$ is defined whenever $i \leq t$. In other words, γ and ρ satisfy the temporal dependencies implicated by the typing $\Gamma \Vdash e : \tau^t$.

In order to make the reduction semantics practically useful, we implement it in the form of a function `adv` that takes a contract c, a partial variable assignment γ, and a partial external environment ρ and returns a contract c' together with the transfer function $T \in \mathsf{Trans}$ such that $c \overset{T}{\Longrightarrow}_{\gamma,\rho} c'$. The function `adv` can be implemented by transcribing the inference rules from Figure 10 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 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 `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 antisymmetry of transfer functions, that is, the fact that $T(p,q,a) = -T(q,p,a)$ (cf. Lemma 2), by only storing one of the values $T(p,q,a)$ and $T(q,p,a)$.

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 Bruijn 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 Figure 2 and the time-indexed type system from Figure 7. 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, 3, and 4 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 implemented in Haskell. That is, the implementation should satisfy the properties that we have proved in Coq. 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-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 Coq's code extraction itself (which has been addressed to some extent [23]). Our work requires trust in 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 mapping. In addition, our Coq formalisation uses axiomatised abstract types, that is, 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 as well as the type of real numbers. 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, that is, 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 front end 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 uncertified code.

In order to build a combinator library to construct contracts and expressions, we use the approach of Atkey et al. [5]. 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 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 11.

The type classes *E* and *C* are used to provide an abstract interface for constructing expressions and contracts, respectively. The use of these two type classes allows us to use types of the form *exp t* both for expressions and for bound variables (cf. the type signatures of acc and letc in Figure 11). 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*).

Figure 12 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.

Figure 13 shows a more complex contract expressed in the Haskell EDSL: it describes a bond that is insured by a credit default swap (CDS). We have already seen a similar—but simpler—contract in Example 4, where we considered a CDS for a zero-coupon bond. The contract in Figure 13 describes a bond that pays a monthly interest during the 12 months term of the bond. Also the CDS is different: the buyer has to pay a monthly premium instead of a single up-front premium.

```
-- Expresssions
acc  :: E exp ⇒ (exp t → exp t) → Int → exp t → exp t
rObs :: E exp ⇒ RealObs → Int → exp R
bObs :: E exp ⇒ BoolObs → Int → exp B

max, min, (+), (*), (/), (-)    :: E exp ⇒ exp R → exp R → exp R
(==), (<), (≤), (>), (>), (≥)   :: E exp ⇒ exp R → exp R → exp B

(&&), (||)   :: E exp ⇒ exp B → exp B → exp B
not          :: E exp ⇒ exp B → exp B
false , true :: E exp ⇒ exp B

-- Contracts
type Contr = forall exp contr . C exp contr ⇒ contr

transfer :: C exp contr ⇒ Party → Party → Asset → contr
zero     :: C exp contr ⇒ contr
letc     :: C exp contr ⇒ exp t → (exp t → contr) → contr
(&)      :: C exp contr ⇒ contr → contr → contr
(!)      :: C exp contr ⇒ Int → contr → contr
(#)      :: C exp contr ⇒ exp R → contr → contr
ifWithin :: C exp contr ⇒ exp B → Int → contr → contr → contr

-- Contract management
horizon    :: Contr → Int
welltyped  :: Contr → Bool
advance    :: Contr → ExtEnvP → (Contr, FMap)
specialise :: Contr → ExtEnvP → Contr
```

Figure 11. Interface for the Haskell extracted contract library.

```
{-# LANGUAGE RebindableSyntax #-}

import RebindableEDSL

asian :: Contr
asian = 90 ! if bObs (Decision X "exercise") 0
             then 100 # ( transfer Y X USD &
                          (rate # transfer X Y DKK))
             else zero
  where rate = (acc (λr → r +
                      rObs (FX USD DKK) 0) 30 0) / 30

american :: Contr
american = if bObs (Decision X "exercise") 0 `within` 90
           then 100 # ( transfer Y X USD &
                        (6.23 # transfer X Y DKK))
           else zero
```

Figure 12. Complete Haskell code for Asian and American option.

6. Related Work

Contract Languages. Research on formal contract languages can be traced back to the work of Lee [21] 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 [13, 14].

Most relevant to our work is the pioneering work on financial contracts by Peyton Jones et al. [28]. This work has evolved into the company LexiFi, which has implemented the techniques on top of their MLFi variant of OCaml [24]. The resulting con-

```
{−# LANGUAGE RebindableSyntax #−}

import RebindableEDSL

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

cds months cur premium comp buyer seller ref = step months
  where step i = if i ≤ 0 then zero
                 else premium # transfer buyer seller cur &
                   if bObs (Default ref) 0 `within` 30
                   then 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 &
                   if bObs (Default issuer) 0 `within` 30
                   then zero
                   else step (i−1)
```

Figure 13. Complete Haskell code for a CDS for a bond.

tract management platform runs worldwide in many financial institutions through its integration in key financial institutions, such as Bloomberg[5], and with large asset-management platforms, such as SimCorp Dimension [30], a comprehensive asset-management platform for financial institutions.

Based also on earlier work on contract languages [3], 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 [9]. It has thus become well-known that domain specific languages for contract management result in more agility, shorter time-to-market for new products, and increased assurance of software quality. See also [1] for an overview of resources related to domain specific languages for the financial domain.

Multi-party contracts have been investigated earlier in the work by Andersen et al. [2] and Henglein et al. [11] on establishing a formal transaction system for enterprise resource planning (ERP). In this line of work, the contract language resembles a process calculus, which is used to match 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. Likewise, Peyton Jones et al. [28] considered a denotational semantics, however, their semantics is based on stochastic processes. Our denotational semantics draws from previous work on trace-based contract formalisms [2, 15, 20]. However, in order to accommodate the financial domain 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 a later version of their work, Peyton-Jones and Eber also sketched an operational semantics for contract management [27]. A full reduction semantics is given by Andersen et al. [2] as well as Hvitved et al. [15] along with proofs of their adequacy with respect to the corresponding denotational semantics. The absence of observables in the work of Andersen et al. [2] and Hvitved et al. [15] 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 Schuldenzucker [29]. Interestingly, this axiomatisation of two-party contracts is not based on equality of contracts but rather an order $\leq$ on contracts. Equality of contracts is then derived from the order $\leq$.

Software Verification and Certified Software. In the course of the last decade, formal software verification has grown into a mature technology. It has been applied to realistic software systems such as compilers [22] and operating system kernels [18]. The use of code extraction to obtain executable code from a formally verified implementation is an established technique in the community [7, 10, 23]. 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.

The only application of formal software verification in the financial sector we have seen so far is *Imandra* [16], which has been recently developed by the financial technology startup Aesthetic Integration. The core of Imandra is a modelling language for describing financial algorithms. According to Aesthetic Integration, the modelling language has both a formal semantics—for the purpose of formal reasoning—and an executable semantics—for producing executable code. The formal verification capabilities that the Imandra system provides are limited to automated reasoning provided by an SMT solver specialised to the financial domain. While this setup limits the system's reasoning power compared to the abovementioned software verification efforts that use proof assistants, the use of automatic reasoning lowers the burden for proving properties substantially.

Type Systems. The use of type systems to guarantee certain temporal properties of programs has been extensively studied in the literature. Most work in this direction stems from applying the Curry-Howard correspondence to linear-time temporal logic (LTL) or fragments thereof: Davies [8] devises a constructive variant of LTL and uses it as a type system for binding time analysis. Apart from the temporal 'next' modality, Davies's system also features time-indexing of the typing judgement and the variables in the typing context. Jeffrey [17] embeds LTL into a dependently typed programming language to type functional reactive programs. However, his underlying model of *reactive types* is much more expressive than plain LTL and, for example, allows him to define the function space of causal functions. Krishnaswami and Benton [19] present a type system for functional reactive programs using time-indexed types similar to ours. But in addition, they also have a 'next' modality inspired by Nakano's calculus for guarded recursion [25]. More recently, Atkey and McBride [4] extended Nakano's calculus with clock variables for practical programming with coinductive types. Considering the special case of streams (as coinductive types), the type system of Atkey and McBride ensures that all stream transformations satisfy the causality principle.

7. Conclusions and Future Work

We have presented a symbolic framework for modelling financial contracts that is capable of expressing common FX and other derivatives. 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

[5] Press release available at `http://www.lexifi.com/clients/press_release_en/bloomberg`.

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 contract language is implemented using the Coq proof assistant, which enables us to certify the intended properties of our contract analyses and transformations against the denotational cash-flow trace semantics. We used Coq's code extraction functionality to derive a Haskell implementation from the Coq formalisation. The resulting Haskell module can be used as a certified core of a portfolio management framework.

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 $\mathbb{R} \to \mathsf{Trans}$ instead of $\mathbb{N} \to \mathsf{Trans}$. 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 **iter** that behaves like the accumulation combinator **acc**, but works on contracts instead of expressions. Such a combinator would allow us to express concisely contracts with repetition, such as the bond and CDS example from Figure 13. Currently, we rely on the meta language (i.e., Haskell) to construct such contracts. The contracts are represented in our core contract language and all of our contract management tools apply to them. However, the **iter** combinator would provide a more compact representation.

Our contract language can only express contracts with a finite horizon, which covers most practically relevant financial contracts. Although uncommon, there are financial contracts that are perpetual (e.g., perpetual bonds such as UK World War I bonds). To express such contracts, we would need to extend the contract language, for instance, with an *if-within* construct with unbounded horizon or an unbounded version of the **iter** combinator.

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 [26].

Acknowledgements.

We would like to thank Fritz Henglein, LexiFi, the attendees of NWPT 2014 as well as the anonymous referees of NWPT 2014 and ICFP 2015 for many useful comments and suggestions.

References

[1] DSLFin: Financial domain-specific language listing. `http://www.dslfin.org/resources.html`, 2013.

[2] J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen. Compositional specification of commercial contracts. *Int. J. Softw. Tools Technol. Transf.*, 8(6):485–516, 2006.

[3] B. Arnold, A. Van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13, 1995.

[4] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *ICFP*, pages 197–208, 2013.

[5] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *ACM SIGPLAN Symposium on Haskell*, pages 37–48, 2009.

[6] J. Berthold, A. Filinski, F. Henglein, K. Larsen, M. Steffensen, and B. Vinter. Functional High Performance Financial IT – The HIPERFIT Research Center in Copenhagen. In *TFP'11 – Revised Selected Papers*, 2012.

[7] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2013.

[8] R. Davies. A temporal-logic approach to binding-time analysis. In *LICS*, pages 184–195, 1996.

[9] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard. Commercial uses: Going functional on exotic trades. *J. Funct. Program.*, 19(1): 27–45, 2009.

[10] F. Haftmann. From higher-order logic to Haskell: There and back again. In *PEPM*, pages 155–158, 2010.

[11] F. Henglein, K. F. Larsen, J. G. Simonsen, and C. Stefansen. POETS: Process-oriented event-driven transaction systems. *J. Log. Algebr. Program.*, 78(5):381 – 401, 2009.

[12] J. Hull and A. White. CVA and wrong-way risk. *Financ. Anal. J.*, 68 (5):58–69, 2012.

[13] T. Hvitved. A survey of formal languages for contracts. In *FLACOS*, pages 29–32, 2010.

[14] T. Hvitved. *Contract Formalisation and Modular Implementation of Domain-Specific Languages*. PhD thesis, Department of Computer Science, University of Copenhagen, 2011.

[15] T. Hvitved, F. Klaedtke, and E. Zalinescu. A trace-based model for multiparty contracts. *J. Log. Algebr. Program.*, 81(2):72–98, 2012.

[16] D. A. Ignatovich and G. O. Passmore. Creating safe and fair markets. White Paper AI/1501, Aesthetic Integration, Apr. 2015. URL `http://www.aestheticintegration.com/files/ai-wp1501.pdf`.

[17] A. Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *PLPV*, pages 49–60, 2012.

[18] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM T. Comput. Syst.*, 32(1):2:1–2:70, 2014.

[19] N. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *LICS*, pages 257–266, 2011.

[20] M. Kyas, C. Prisacariu, and G. Schneider. Run-time monitoring of electronic contracts. In *ATVA*, pages 397–407, 2008.

[21] R. M. Lee. A logic model for electronic contracting. *Decis. Support Syst.*, 4(1):27–44, 1988.

[22] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.

[23] P. Letouzey. Extraction in Coq: An overview. In *Computability in Europe*, volume 5028 of *LNCS*, pages 359–369, 2008.

[24] LexiFi. Contract description language (MLFi). `http://www.lexifi.com/technology/contract-description-language`.

[25] H. Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.

[26] C. Oancea, J. Berthold, M. Elsman, and C. Andreetta. A financial benchmark for GPGPU compilation. In *CPC*, 2015.

[27] S. Peyton Jones and J.-M. Eber. How to write a financial contract. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*. Palgrave Macmillan, 2003.

[28] S. Peyton Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ICFP*, 2000.

[29] S. Schuldenzucker. Decomposing contracts – a formalism for arbitrage argumentations. Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2014.

[30] SimCorp A/S. XpressInstruments solutions. Company white-paper. Available from `http://simcorp.com`, 2009.