Certified Symbolic Management of Financial Contracts [*]

Patrick Bahr Jost Berthold Martin Elsman

Dept. of Computer Science
University of Copenhagen
Universitetsparken 5
2100 Copenhagen
{paba,berthold,mael}@di.ku.dk

Abstract

Domain-specific languages (DSLs) for complex financial contracts are in practical use in many banks and financial institutions today. Given the level of automation and pervasiveness of software in this sector, the financial domain is immensely sensitive to software bugs. In this paper, we present a small and simple contract language that rigorously relegates any artifacts of modeling and computation from its core, which leads to favorable algebraic properties, and therefore allows for formalising domain-specific analyses and transformations using a proof assistant. Important information can be derived from, and useful manipulations defined on, just the symbolic contract specification, independent of any stochastic aspects of the modeled contracts. Contracts modeled in our language are analysed and transformed for management according to a precise cash-flow semantics, formalised and verified using the Coq proof assistant. A Haskell implementation of the contract management functionality is automatically extracted from the verified Coq formalisation. This approach opens a road-map towards more reliable contract management software, including the possibility of analysing contracts based on symbolic instead of numeric methods.

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

General Terms Languages, Verification

Keywords DSL, Code generation, Coq, Haskell, Financial Contracts

* This work has been partially supported by the Danish Council for Strategic Research under contract number 10-092299 (HIPERFIT [4]), and the Danish Council for Independent Research under Project 12-132365.

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. Many banks and financial institutions use domain-specific languages (DSLs) to describe complex financial contracts, in particular, for specifying how asset transfers for a specific contract depend on underlying observables, such as interest rates, currency rates, and stock prices.

The seminal work by Peyton-Jones and Eber on financial contracts [17] shows how an algebraic approach to contract specification can be used for valuation of contracts (when combined with a model of the underlying observables) and introduces a contract management model where contracts gradually evolve into the empty contract (for which no party has any further obligations) when knowledge of underlying observables becomes available and decisions are taken. The ideas have emerged into the company LexiFi, which has become a leading software provider for a range of financial institutions, with all contract management operations centralised around a domain-specific contract language hosted in the MLFi [15] dialect of the functional programming language OCaml.

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, and also have to fulfill auditing requirements imposed by regulators. An integrated approach of software construction by generating code from a certified model is therefore highly desirable.

In this paper, we present a small simple contract language, which rigorously relegates any artifacts of modeling and computation from its core. The language shares the same vision as the previously mentioned work with the addition that it (a) allows for specifying multi-party contracts (such as entire portfolios), (b) has good algebraic properties, well suited for formal reasoning, and yet (c) allows for expressing many interesting contracts that appear in real-world portfolios, such as various common foreign exchange options.

Essential contract properties can be derived from a symbolic contract specification alone, and 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. Both contract analysis and symbolic contract manipulation in our model are based on a precise cash-flow semantics for contracts, which we have modeled and checked using the Coq proof assistant. Using code extraction functionality in Coq, the certified contract analyses and transformations can be extracted into a Haskell module, which serves as the certified core of a financial contract management library (wrapped in suitable interface modules for better modularity).

In summary, our contributions in this paper are

- a small but expressive contract DSL;
- a precise cash-flow semantics;
- a model of common contract analyses and transformations accessible to correctness and soundness proofs using a proof assistant;
- and a generated implementation of the certified analyses and transformations in Haskell.

Implementations of the contract language in Haskell and Coq are available online[1] together with machine-checkable proofs (in Coq) of the key properties of the contract language.

2. The Contract Language

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

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

Figure 1 presents the typing rules for the expression language. We use $\mathsf{Expr}_{\mathbb{R}}$ and $\mathsf{Expr}_{\mathbb{B}}$ for the real-valued and Boolean-valued expressions, respectively. The typing rules use typing environments Γ, which are partial mappings from variable names to the set $\{\mathbb{R}, \mathbb{B}\}$. Instead of $\emptyset \vdash e : \mathsf{Expr}_\alpha$, where $\emptyset$ denotes the empty environment, we write $\vdash e : \mathsf{Expr}_\alpha$. Expressions e with typing $\vdash e : \mathsf{Expr}_\alpha$ are called *closed*. The expression sub-language features observable values, choices, and a special backwards-accumulating expression. Note that the obs and acc combinators are indexed by a type α, which ranges over the set $\{\mathbb{R}, \mathbb{B}\}$. However, we often omit this type index, if it obvious from the context in which the combinator appears. The next section will introduce the interesting expressions and contracts combinators by small examples of common financial contracts.

2.1 Examples and Language Constructs

In the paper, we explain contracts using examples from the foreign exchange (FX) market and days as the time unit, but the concepts generalise easily. Likewise, cash-flows are based on a fixed set of *currencies*, but could instead allow for arbitrary assets.

As a first simple example, consider the following *forward contract*, an agreement to purchase an asset in the future for a fixed price.

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

$$\frac{r \in \mathbb{R}}{\Gamma \vdash r : \mathsf{Expr}_{\mathbb{R}}} \qquad \frac{b \in \mathbb{B}}{\Gamma \vdash r : \mathsf{Expr}_{\mathbb{B}}} \qquad \frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \mathsf{Expr}_{\alpha}}$$

$$\frac{\oplus \in \{+, -, \cdot, /, max, min\} \quad \Gamma \vdash e_i : \mathsf{Expr}_{\mathbb{R}}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Expr}_{\mathbb{R}}}$$

$$\frac{\oplus \in \{\leq, <, =, \geq, >\} \quad \Gamma \vdash e_i : \mathsf{Expr}_{\mathbb{R}}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Expr}_{\mathbb{B}}}$$

$$\frac{\oplus \in \{\wedge, \vee\} \quad \Gamma \vdash e_i : \mathsf{Expr}_{\mathbb{B}}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Expr}_{\mathbb{B}}} \qquad \frac{\Gamma \vdash e : \mathsf{Expr}_{\mathbb{B}}}{\Gamma \vdash \neg e : \mathsf{Expr}_{\mathbb{B}}}$$

$$\frac{l \in \mathsf{Label}_\alpha \quad d \in \mathbb{Z}}{\Gamma \vdash obs_\alpha(l, d) : \mathsf{Expr}_\alpha}$$

$$\frac{\Gamma[x/\alpha] \vdash f : \mathsf{Expr}_\alpha \quad d \in \mathbb{N} \quad \Gamma \vdash a : \mathsf{Expr}_\alpha}{\Gamma \vdash acc_\alpha(\lambda x.f, d, a) : \mathsf{Expr}_\alpha}$$

Figure 1. Expression sub-language.

EXAMPLE 1 (FX Forward). *In 90 days, party X will buy* 100 *US dollars for a fixed rate* r *of Danish Kroner from Party Y.*

$$\begin{aligned} option &= translate(90, trade) \\ trade &= scale(100, both(transfer(Y, X, \mathsf{USD}), pay)) \\ pay &= scale(r, transfer(X, Y, \mathsf{DKK})) \end{aligned}$$

The *trade* specifies that X *both* receives one USD from Y and *pay*s for it (at rate r), scaled by the amount (100). The use of the *translate* combinator translates the trade 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 directly using *ifWithin*, an iterating generalisation of a simple alternative (*if-then-else*). As an example, consider a so-called *American Option*, where one party may, at any time before the contract ends, decide to execute the purchase.

EXAMPLE 2 (FX American Option). *Party X may, within 90 days, decide whether to (immediately) buy* 100 *US dollars for a fixed rate* r *of Danish Kroner from Party Y.*

$$\begin{aligned} option &= ifWithin(obs_{\mathbb{B}}(X, 0), 90, trade, zero) \\ trade &= scale(100, both(transfer(Y, X, \mathsf{USD}), pay)) \\ pay &= scale(r, transfer(X, Y, \mathsf{DKK})) \end{aligned}$$

This contract uses an observable external decision, expressed using $obs_{\mathbb{B}}$ (which uses a time offset, 0 meaning the current day), and the *ifWithin* construct, which iterates the decision of party X. If X chooses at one day before the end (90 days), the *trade* comes into effect, consisting of two transfers (*both*) between the parties. Otherwise, the contract becomes empty (*zero*) after 90 days.

As a domain-specific element of the expression language, we define a special 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 so-called *Asian options* (or average options), for which a price is established from an average of past prices instead of just one observed price.

EXAMPLE 3 (FX Asian Option). *After 90 days, Party X may decide to buy 100 USD; paying the* average *of the exchange rate USD/DKK* in the last 30 days.

$$\begin{aligned} option &= translate(90, if(obs_{\mathbb{B}}(X, 0), trade, zero) \\ trade &= scale(100, both(transfer(Y, X, \mathsf{USD}), pay)) \\ pay &= scale(rate, transfer(X, Y, \mathsf{DKK})) \\ rate &= \frac{1}{30} \cdot acc(\lambda r.r + obs_{\mathbb{R}}(\mathsf{FX\ USD/DKK}, 0), 30, 0) \end{aligned}$$

In addition to the $obs_{\mathbb{B}}$-modeled decision, this contract uses an $obs_{\mathbb{R}}$ expression to observe the exchange rate between USD and

$$\frac{}{\vdash \mathit{zero} : \mathsf{Contr}} \qquad \frac{p_1, p_2 \in \mathsf{Party} \quad c \in \mathsf{Currency}}{\vdash \mathit{transfer}(p_1, p_2, c) : \mathsf{Contr}}$$

$$\frac{\vdash e : \mathsf{Expr}_{\mathbb{R}} \quad \vdash c : \mathsf{Contr}}{\vdash \mathit{scale}(e, c) : \mathsf{Contr}} \qquad \frac{d \in \mathbb{N} \quad \vdash c : \mathsf{Contr}}{\vdash \mathit{translate}(d, c) : \mathsf{Contr}}$$

$$\frac{\vdash c_i : \mathsf{Contr}}{\vdash \mathit{both}(c_1, c_2) : \mathsf{Contr}}$$

$$\frac{\vdash e : \mathsf{Expr}_{\mathbb{B}} \quad d \in \mathbb{N} \quad \vdash c_i : \mathsf{Contr}}{\vdash \mathit{ifWithin}(e, d, c_1, c_2) : \mathsf{Contr}}$$

Figure 2. Contract Atoms and Combinators.

DKK (again at offset 0, thus on the current day). Observed values are accumulated to the *rate* using an $acc_{\mathbb{R}}$ expression. The *rate* is determined as the average of observed USD/DKK exchange rates on 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.

The formation rules for the contract combinators are given in Figure 2.

2.2 Denotational Semantics

The denotational semantics of a contract is given with respect to an *environment*, which provides values for all observables and choices involved in the contract. A contract semantics is then given as a series of cash-flows between parties over (relative) time. The definitions for contracts are straightforward once some preparation work and terminology are established.

2.2.1 External Environments

External environments (or simply *environments* for short) provide facts about observables and external decisions involved in contracts. An environment $\rho \in \mathsf{Env}$ is a partial mapping from a day offset ($\mathbb{Z}$) and observables or choices, identified by tags in a suitable tag set Label_α, to values of type $\alpha \in \{\mathbb{R}, \mathbb{B}\}$.

$$\mathsf{Env} = \mathsf{Label}_\alpha \times \mathbb{Z} \rightharpoonup \alpha$$

Note that the domain is $\mathbb{Z}$, that is, an environment may provide information about the past as well as the future. While it might make sense to restrict environments to knowledge about the past and present, the more general variant can also describe future scenarios – however, it raises questions about *contract causality* (see Section 3.2.2).

2.2.2 Expressions

Environments are essential to the semantics of Boolean and real-valued expressions, which is otherwise a conventional semantics of arithmetic and logic expression interpretation. In addition to an environment, we also need variable assignments that map free variables of type α to a value of type α. 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 $\gamma(x) \in \alpha$ iff $x : \alpha \in \Gamma$.

Given an expression typing $\Gamma \vdash e : \mathsf{Expr}_\alpha$, the semantics of e, denoted $\mathcal{E}[\![e]\!]$ is a partial mapping of type $[\![\Gamma]\!] \times \mathsf{Env} \rightharpoonup \alpha$. The semantics is a partial mapping because of the partiality of environments. Instead of $\mathcal{E}[\![e]\!](\gamma, \rho)$, we write $\mathcal{E}[\![e]\!]_{\gamma,\rho}$.

$$\begin{aligned}
\mathcal{E}[\![l]\!]_{\gamma,\rho} &= l \text{ if } l \in \mathbb{R} \cup \mathbb{B} \\
\mathcal{E}[\![x]\!]_{\gamma,\rho} &= \gamma(x) \\
\mathcal{E}[\![obs_\alpha(l, d)]\!]_{\gamma,\rho} &= \rho(l, d) \\
\mathcal{E}[\![a \odot b]\!]_{\gamma,\rho} &= \mathcal{E}[\![a]\!]_{\gamma,\rho} \otimes \mathcal{E}[\![b]\!]_{\gamma,\rho} \\
&\qquad (\otimes :: \alpha \times \alpha \to \beta \text{ implements op. } \odot)
\end{aligned}$$

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

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

In other words, ρ/d is time shifted d days into the future. The environment ρ/d is also called the promotion of ρ by d.

The semantics of the accumulator combinator is then recursively defined as follows:

$$\mathcal{E}[\![acc_\alpha(\lambda x.f, k, a)]\!]_{\gamma,\rho} = \begin{cases} \mathcal{E}[\![a]\!]_{\gamma,\rho} & \text{if } k = 0 \\ \mathcal{E}[\![f]\!]_{\gamma[x \mapsto v],\rho} & \text{if } k > 0 \end{cases}$$
$$\text{where } v = \mathcal{E}[\![acc_\alpha(f, k-1, a)]\!]_{\gamma,\rho/-1}$$

The semantics of the *accumulator* combinator iterates the argument f by stepping backwards in time. This behavior can be expressed equivalently using *promotion* of expressions, in analogy to promotion of environments. Promoting an expression by d days translates all contained observables and choices d days into the future. For any expression $e : \mathsf{Expr}_\alpha$ and number $d \in \mathbb{Z}$, the expression e/d, is defined as:

$$\begin{aligned}
x/d &= x \text{ if } x \text{ is literal or variable} \\
obs_\alpha(p, i)/d &= obs_\alpha(p, d + i) \\
(e_1 \odot e_2)/d &= e_1/d \odot e_2/d \\
acc_\alpha(\lambda x.f, k, a)/d &= acc_\alpha(\lambda x.(f/d), k, a/d)
\end{aligned}$$

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

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

$$\mathcal{E}[\![e/d]\!]_{\gamma,\rho} = \mathcal{E}[\![e]\!]_{\gamma,\rho/d}$$

Thus $acc_\alpha(\lambda x.f, k, a)$ is semantically equivalent to the following:

$$f[x \mapsto (f/-1)[x \mapsto \ldots (f/-(d-1))[x \mapsto a/-d]\ldots]]$$

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

Given a closed expression e, we simply write $\mathcal{E}[\![e]\!]_\rho$ instead of $\mathcal{E}[\![e]\!]_{\emptyset,\rho}$, where $\emptyset$ denotes the empty variable assignment.

2.2.3 Contracts

With the expression semantics defined, a semantics for contracts can be defined in a straightforward way. The semantics of a contract is given by its cash-flow *trace*, a partial mapping from time to

$$
\begin{aligned}
\mathcal{C}\left[\!\left[transfer(a,b,cur) \right]\!\right]_\rho &= \lambda n.\lambda(x,y,z).\begin{cases} 1 & \text{if } x = a \wedge y = b \wedge z = cur \wedge n = 0 \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{C}\left[\!\left[zero \right]\!\right]_\rho &= \lambda n.\lambda t.0 \\
\mathcal{C}\left[\!\left[translate(d,c) \right]\!\right]_\rho &= delay(d, \mathcal{C}\left[\!\left[c \right]\!\right]_\rho), \qquad \text{where } delay(d,f) = \lambda n.\begin{cases} f(n-d) & \text{if } n \geq d \\ \lambda x.0 & \text{otherwise} \end{cases} \\
\mathcal{C}\left[\!\left[scale(e,c) \right]\!\right]_\rho &= \lambda n.\lambda(p_1,p_2,c).\begin{cases} \mathcal{E}\left[\!\left[e \right]\!\right]_\rho \cdot \mathcal{C}\left[\!\left[c \right]\!\right]_\rho(n)(p_1,p_2,c) & \text{if } \mathcal{E}\left[\!\left[e \right]\!\right]_\rho \geq 0 \\ -\mathcal{E}\left[\!\left[e \right]\!\right]_\rho \cdot \mathcal{C}\left[\!\left[c \right]\!\right]_\rho(n)(p_2,p_1,c) & \text{if } \mathcal{E}\left[\!\left[e \right]\!\right]_\rho < 0 \end{cases} \\
\mathcal{C}\left[\!\left[both(c_1,c_2) \right]\!\right]_\rho &= \lambda n.\lambda t.\mathcal{C}\left[\!\left[c_1 \right]\!\right]_\rho(n)(t) + \mathcal{C}\left[\!\left[c_2 \right]\!\right]_\rho(n)(t) \\
\mathcal{C}\left[\!\left[ifWithin(b,d,c_1,c_2) \right]\!\right]_\rho &= \begin{cases} \mathcal{C}\left[\!\left[c_1 \right]\!\right]_\rho & \text{if } \ \mathcal{E}\left[\!\left[b \right]\!\right]_\rho \\ \mathcal{C}\left[\!\left[c_2 \right]\!\right]_\rho & \text{if } \neg\mathcal{E}\left[\!\left[b \right]\!\right]_\rho \ \wedge \ d = 0 \\ delay(1, \mathcal{C}\left[\!\left[ifWithin(b, d{-}1, c_1, c_2) \right]\!\right]_{(\rho/1)}) & \text{if } \neg\mathcal{E}\left[\!\left[b \right]\!\right]_\rho \ \wedge \ d > 0 \end{cases}
\end{aligned}
$$

Figure 3. Denotational Semantics for Contracts.

transfers (Trans) between two parties:[2]

$$\mathsf{Trans} = \mathsf{Party} \times \mathsf{Party} \times \mathsf{Currency} \to \mathbb{R}^+$$
$$\mathsf{Trace} = \mathbb{N} \rightharpoonup \mathsf{Trans}$$

The cash-flow trace is a partial mapping since it may not be determined due to insufficient knowledge about observables and external decisions, leading to a partial semantics

$$\mathcal{C}\left[\!\left[\cdot \right]\!\right]_\cdot : \mathsf{Contr} \times \mathsf{Env} \to \mathsf{Trace}$$

Figure 3 shows the denotational contract semantics in full. This denotational semantics is the foundation for the formalisation of symbolic contract analyses, contract management, and contract transformations.

2.2.4 Monotonicity

An important property of the semantics of contracts is *monotonicity*. Intuitively, if an environment ρ_2 provides more information than another (ρ_1), then the semantics of a contract c with respect to environment ρ_2 contains more information as well.

In order to formally capture monotonicity, we define *graph inclusion* to compare the extent to which a partial mapping is defined, by relation $\sqsubseteq$ (a partial ordering on partial functions). Given two partial functions $f_1, f_2 : X \rightharpoonup Y$, we define that $f_1 \sqsubseteq f_2$ iff for all $x \in X$ such that $f_1(x)$ is defined, we have that $f_2(x) = f_1(x)$.

PROPOSITION 1 (monotonicity).
Given $\vdash c : \mathsf{Contr}$ *and* $\rho_1, \rho_2 \in \mathsf{Env}$, *we have that* $\rho_1 \sqsubseteq \rho_2$ *implies* $\mathcal{C}\left[\!\left[c \right]\!\right]_{\rho_1} \sqsubseteq \mathcal{C}\left[\!\left[c \right]\!\right]_{\rho_2}$.

Well-understood, the contract representation using only relative time cannot be used alone for actual contract management. In a real software system, a contract will always be accompanied by its start date; and environments will likewise refer to an absolute reference date. However, our implementation experience is that contracts using relative dates are much easier to compose, analyse and manipulate; therefore we conjecture that start and reference dates should be removed for all internal operations of a contract management software.

[2] 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 to enable considering an entire portfolio as one contract.
In real-world financial contracts, parties would explicitly agree on netting, or otherwise handle cash-flows from different contracts as separate entities. In contrast to our model, cash-flows between the same parties, but in opposite directions would then be joined as well, while we still consider them separate.

3. Contract Analysis and Management

With the denotational semantics of contracts at hand, we can define a number of semantic properties that are relevant when managing contracts.

3.1 Simple Contract Equivalences

The denotational semantics provides a natural notion of contract equality:

$$c_1 \equiv c_2 \quad \text{iff} \quad \mathcal{C}\left[\!\left[c_1 \right]\!\right]_\rho = \mathcal{C}\left[\!\left[c_2 \right]\!\right]_\rho \ \text{ for all } \rho \in \mathsf{Env}$$

However, this notion of equivalence is quite restrictive since it also requires that both contracts behave in the same way regarding partiality. For instance, we do not have the equivalence

$$scale(e, zero) \equiv zero$$

Depending on the environment ρ, the semantics $\mathcal{E}\left[\!\left[e \right]\!\right]_\rho$ and thus the semantics of the left-hand side contract may be undefined, whereas the semantics of the right-hand side contract is always defined.

Therefore, we consider the following weaker but still adequate form of equivalence:

$$c_1 \simeq c_2 \quad \text{iff} \quad \mathcal{C}\left[\!\left[c_1 \right]\!\right]_\rho = \mathcal{C}\left[\!\left[c_2 \right]\!\right]_\rho \quad \text{for all } \rho \in \mathsf{Env} \text{ such that } \mathcal{C}\left[\!\left[c_1 \right]\!\right]_\rho \text{ and } \mathcal{C}\left[\!\left[c_2 \right]\!\right]_\rho \text{ are total functions.}$$

That is the semantics of the two contracts only has to coincide for environments that make the semantics of both contracts total. This notion of equivalence is adequate, since there are environments ρ for which $\mathcal{C}\left[\!\left[c \right]\!\right]_\rho$ is total for any contract c, namely any environment that is a total function.

A number of simple equivalences can be proved easily using the denotational semantics; Figure 4 gives some examples.

Adding expression promotion, as defined earlier, and considering the netting semantics of contracts, more involved equivalences can be proved, such as the ones shown in Figure 5. These contract equivalences can be used to simplify a given contract, for instance to achieve a certain normalised format suitable for further processing.

3.2 Inferring Contract Properties

When dealing with contracts, we are interested in a number of semantic properties, which can be characterised precisely using

$$
\begin{aligned}
\mathit{translate}(d, \mathit{zero}) &\simeq \mathit{zero}\\
\mathit{scale}(r, \mathit{zero}) &\simeq \mathit{zero}\\
\mathit{scale}(0, c) &\simeq \mathit{zero}\\
\mathit{both}(c, \mathit{zero}) &\simeq c\\
\mathit{scale}(s_1, \mathit{scale}(s_2, c)) &\simeq \mathit{scale}(s_1 \cdot s_2, c)\\
\mathit{translate}(d_1, \mathit{translate}(d_2, c)) &\simeq \mathit{translate}(d_1 + d_2, c)\\
\mathit{translate}(d, \mathit{both}(c_1, c_2)) &\simeq \mathit{both}(\mathit{translate}(d, c_1), \mathit{translate}(d, c_2))\\
\mathit{scale}(x, \mathit{both}(c_1, c_2)) &\simeq \mathit{both}(\mathit{scale}(x, c_1), \mathit{scale}(x, c_2))
\end{aligned}
$$

Figure 4. Some Simple Contract Equivalences.

$$
\begin{aligned}
\mathit{translate}(d, \mathit{scale}(s, c)) &\simeq \mathit{scale}(s/d, \mathit{translate}(d, c))\\
\mathit{translate}(d, \mathit{ifWithin}(b, e, c_1, c_2)) &\simeq\\
&\mathit{ifWithin}(b/d, e, \mathit{translate}(d, c_1), \mathit{translate}(d, c_2))\\
\mathit{both}(\mathit{scale}(x, \mathit{transfer}(a, b, c)), &\mathit{scale}(y, \mathit{transfer}(a, b, c)))\\
&\simeq \mathit{scale}(x + y, \mathit{transfer}(a, b, c))
\end{aligned}
$$

Figure 5. Some Contract Equivalences Involving Promotion and Netting.

the denotational contract semantics. In particular, the following properties will be interesting for contract management operations:

Contract Dependence: On which observables do the cash-flows of a contract depend?

Causality: Does a contract contain cash-flows which depend on "future" observables?

Contract Horizon: What is the minimum time span until a contract is certain to be *zero*?

3.2.1 Contract Dependence

The observables and choices on which a contract depends can be easily determined in a syntax-directed (inductive) fashion. A dependence of a contract is composed of an identifying tag and a *time span* for the dependence. Dependence is introduced by use of the *scale* and *ifWithin* constructs, with the latter construct introducing time spans of more than a single day. The definitions of dependence are straightforward and are not shown here.

3.2.2 Contract Causality

When used directly, the contract combinators allow for defining contracts that make no sense in reality. For instance, one could define a transfer to be executed today using the FX rate of tomorrow:

$$\mathit{scale}(\mathit{obs}(\mathsf{FX\ USD/DKK}, 1), \mathit{transfer}(A, B, \mathsf{DKK}))$$

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 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 $\alpha \in \{\mathbb{R}, \mathbb{B}\}$, $l \in \mathsf{Label}_\alpha$, and $s \in \mathbb{Z}$. Causality can then be captured by the following definition: A 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 $\mathcal{C}[\![c]\!]_{\rho_1}(t) = \mathcal{C}[\![c]\!]_{\rho_2}(t)$. That is, the cash-flows at any time t do not depend on observables and decisions after t.

It is in general undecidable whether a contract is causal, but we can provide conservative approximations. We say that a contract c is *syntactically causal* iff, for every sub-expression of the form $\mathit{obs}_\alpha(l, d)$, we have that $d \leq 0$.

We can show that the syntactical notion of causality implies the semantic notion:

PROPOSITION 2 (soundness of syntactic causality).
Every syntactically causal contract is also causal.

The converse is not true. There are causal contracts that are not syntactically causal. Consider for instance the following contract:

$$\mathit{scale}(\mathit{obs}(\mathsf{FX\ USD/DKK}, 1), \mathit{translate}(1, \mathit{transfer}(A, B, \mathsf{DKK})))$$

In this respect, syntactic causality vs. causality is similar to type checking vs. type safety. However, we have not encountered a practically relevant causal contract that cannot be translated into a semantically equivalent contract that is *syntactically* causal. For instance, the above contract is equivalent to the following syntactically causal contract (cf. Figure 5):

$$\mathit{translate}(1, \mathit{scale}(\mathit{obs}(\mathsf{FX\ USD/DKK}, 0), \mathit{transfer}(A, B, \mathsf{DKK})))$$

In Section 4.3, we present a monadic technique for composing contracts that are causal by construction, a technique that is also available to developers working in SimCorp's XpressInstrument developer kit, which is built on top of the LexiFi framework [18].

3.2.3 Contract Horizon

We define the *horizon* $d \in \mathbb{Z}$ of a contract c as the minimal time until the last potential cash-flow specified by the contract, under any environment. That is, it is the smallest $d \in \mathbb{Z}$ with

$$\mathit{delay}(-d, \mathcal{C}[\![c]\!]_\rho) \sqsubseteq \mathcal{C}[\![\mathit{zero}]\!]_\rho \quad \text{for all } \rho \in \mathsf{Env}$$

In other words, after d days, the cashflow for the contract c remains undefined or zero, for any environment ρ.

Similar to causality, the horizon of a contract cannot be effectively computed. But we may give a sound approximation of it, by dropping the minimality requirement:

$$
\begin{aligned}
\mathrm{HOR}(\mathit{zero}) &= 0\\
\mathrm{HOR}(\mathit{transfer}(p_1, p_2, c)) &= 1\\
\mathrm{HOR}(\mathit{scale}(e, c)) &= \mathrm{HOR}(c)\\
\mathrm{HOR}(\mathit{transl}(d, c)) &= \mathrm{HOR}(c) + d\\
\mathrm{HOR}(\mathit{both}(c_1, c_2)) &= \max(\mathrm{HOR}(c_1), \mathrm{HOR}(c_2))\\
\mathrm{HOR}(\mathit{ifWithin}(e, d, c_1, c_2)) &= \max(\mathrm{HOR}(c_1), \mathrm{HOR}(c_2)) + d
\end{aligned}
$$

We can show that the semantic contract horizon is never greater than the syntactic horizon.

PROPOSITION 3 (soundness of syntactic horizon).
Given a contract c, we have that $\mathrm{HOR}(c)$ *is greater than or equal to the horizon of c.*

3.3 Contract Transformations

Apart from a variety of analyses, our framework provides functionality to transform contracts in meaningful ways. The most basic form of such transformations are provided by algebraic laws like those given in Figures 4 and 5. These laws state when it is safe to replace a contract c_1 by an equivalent contract c_2. Using our denotational semantics, these algebraic laws can be proved in a straightforward manner.

More interesting are transformations that are based on external knowledge provided by a particular environment; on facts about observables and external decisions, which become gradually available. A contract c can be transformed based on an environment $\rho \in \mathsf{Env}$ that encodes the available knowledge about observables and decisions which influence c, leading to a *specialised* or *reduced* contract.

3.3.1 Specialisation

A specialisation function f performs a partial evaluation of a contract c under a given environment ρ. The resulting contract $f(c, \rho)$ yields the same cash-flow trace as the original c when evaluated under the environment ρ.

More generally, the semantics of a specialised contract $f(c, \rho)$ under any environment that provides less information than the environment ρ used for its specialisation will yield precisely the same cash-flow trace as the original c under the environment ρ:

$$\rho' \sqsubseteq \rho \Longrightarrow \mathcal{C}\left[\!\left[f(c, \rho)\right]\!\right]_{\rho'} = \mathcal{C}\left[\!\left[c\right]\!\right]_{\rho} \quad \text{for all } \rho, \rho' \in \mathsf{Env}$$

In particular, this includes the case where ρ' is the empty environment – in other words, specialisation with ρ extracts cash-flows that are already certain from the information provided in ρ. Separating those known cash-flows leads to a second semantics, which reduces a contract in a step-wise fashion.

3.3.2 Contract Reduction Semantics

In addition to the denotational semantics, we equipped the contract language with a reduction semantics [1], which *advances* a contract by one time unit. We write $c \xRightarrow{\tau}_{\rho} c'$, to denote that c is advanced to c' in the environment ρ, where $\tau \in \mathsf{Trans}$ are the transfers that the contract c yields during this time unit, and c' is the contract that describes all remaining obligations except these transfers (both considering information in the given environment ρ). The reduction semantics is given in Figure 6

The reduction semantics can be implemented as a recursive function of type

$$f_{\Rightarrow} : \mathsf{Contr} \times \mathsf{Env} \rightharpoonup \mathsf{Contr} \times \mathsf{Trans}$$

The function $f_{\Rightarrow}$ takes a contract c and an environment ρ, and returns the residual contract c' and the transfers τ such that $c \xRightarrow{\tau}_{\rho} c'$. The argument ρ typically contains the knowledge that we have about observables and decisions up to the present time, i.e. for time points ≤ 0. Again, the semantics is partial because an environment might miss values that are required to determine a cash-flow or an alternative.

We can show that the reduction semantics is sound and complete with respect to the denotational semantics:

THEOREM 1 (Reduction semantics correctness).

(i) If $c \xRightarrow{\tau}_{\rho} c'$, then
  (a) $\mathcal{C}\left[\!\left[c\right]\!\right]_{\rho}(0) = \tau$, and
  (b) $\mathcal{C}\left[\!\left[c\right]\!\right]_{\rho}(i+1) = \mathcal{C}\left[\!\left[c'\right]\!\right]_{\rho/1}(i) \quad$ for all $i \in \mathbb{N}$.

(ii) If $\mathcal{C}\left[\!\left[c\right]\!\right]_{\rho}(0) = \tau$, then there is a unique c' with $c \xRightarrow{\tau}_{\rho} c'$.

As a consequence, we have that

$$c \xRightarrow{\mathcal{C}[\![c]\!]_{\rho}(0)}_{\rho} c_0 \xRightarrow{\mathcal{C}[\![c]\!]_{\rho}(1)}_{\rho/1} \quad \dots \quad \xRightarrow{\mathcal{C}[\![c]\!]_{\rho}(n-1)}_{\rho/n-1} c_{n-1}$$

where n is the largest number such that $\mathcal{C}\left[\!\left[c\right]\!\right]_{\rho}(i)$ is defined for all $i < n$ and there is no c' and τ such that $c_{n-1} \xRightarrow{\tau}_{\rho/n} c'$.

3.4 Coq Formalisation

We have formalised the contract language in the Coq theorem prover. For the representation of the syntax of the contract language some care has to be taken when choosing a representation of variables and binders. In principle, we could have chosen a simple representation using de Bruijn indices. However, our goal is to extract the Coq definitions into executable Haskell code (cf. section 4). To this end, we have chosen a representation of variable binders using nested abstract syntax [5]. The use of this representation allows us to give the contract combinators a higher-order abstract syntax interface (HOAS) in the extracted Haskell code using a technique proposed by Bernardy and Pouillard [3], without exposing the internal representation of binders.

$$\frac{}{\mathit{zero} \xRightarrow{\tau_0}_{\rho} \mathit{zero}}$$

$$\frac{}{\mathit{transfer}(p_1, p_2, c) \xRightarrow{\tau_{p_1,p_2,c}}_{\rho} \mathit{zero}}$$

$$\frac{c \xRightarrow{\tau}_{\rho} c' \quad \mathcal{E}\left[\!\left[e\right]\!\right]_{\rho} = v}{\mathit{scale}(e, c) \xRightarrow{v * \tau}_{\rho} \mathit{scale}(e/-1, c')}$$

$$\frac{c \xRightarrow{\tau}_{\rho} c'}{\mathit{translate}(0, c) \xRightarrow{\tau}_{\rho} c'}$$

$$\frac{d > 0}{\mathit{translate}(d, c) \xRightarrow{\tau_0}_{\rho} \mathit{translate}(d-1, c)}$$

$$\frac{c_i \xRightarrow{\tau_i}_{\rho} c_i}{\mathit{both}(c_1, c_2) \xRightarrow{\tau_1 + \tau_2}_{\rho} \mathit{both}(c_1, c_2)}$$

$$\frac{\mathcal{E}\left[\!\left[e\right]\!\right]_{\rho} = \mathit{false} \quad c_2 \xRightarrow{\tau}_{\rho} c'}{\mathit{ifWithin}(e, 0, c_1, c_2) \xRightarrow{\tau}_{\rho} c'}$$

$$\frac{\mathcal{E}\left[\!\left[e\right]\!\right]_{\rho} = \mathit{false} \quad d > 0}{\mathit{ifWithin}(e, d, c_1, c_2) \xRightarrow{\tau_0}_{\rho} \mathit{ifWithin}(e, d-1, c_1, c_2)}$$

$$\frac{\mathcal{E}\left[\!\left[e\right]\!\right]_{\rho} = \mathit{true} \quad c_1 \xRightarrow{\tau}_{\rho} c'}{\mathit{ifWithin}(e, d, c_1, c_2) \xRightarrow{\tau}_{\rho} c'}$$

where

$$\tau_0 = \lambda t.0$$

$$\tau_{p_1,p_2,c} = \lambda(p_1', p_2', c').\begin{cases} 1 & \text{if } (p_1', p_2', c') = (p_1, p_2, c) \\ 0 & \text{otherwise} \end{cases}$$

$$v * \tau = \lambda(p_1, p_2, c).\begin{cases} v \cdot \tau(p_1, p_2, c) & \text{if } v \geq 0 \\ -v \cdot \tau(p_2, p_1, c) & \text{if } v < 0 \end{cases}$$

$$\tau_1 + \tau_2 = \lambda t.\tau_1(t) + \tau_2(t)$$

Figure 6. The reduction semantics of the contract language.

Using this representation of the syntax the formalisation of the denotational semantics as well as the reduction semantics is straightforward. Using this formalisation we have proved the algebraic laws of Figure 4 and 5, the monotonicity property (Proposition 1), the soundness of syntactic causality (Proposition 2) and the syntactic horizon (Proposition 3), and the correctness property for the reduction semantics (Theorem 1).

4. Generating Certified Code

Our work presented here aims at generating a certified contract management engine written in Haskell. While ideally, one would like the entire software stack (and even hardware stack) on which contracts are being managed to be certified, there are several non-certified components involved. The generated Haskell code has probably 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 the Coq Haskell extraction itself (which has been addressed to some extent [14]). Our work requires trust into these lower-level components (which, on the other hand, is not an unrealistic assumption).

In the next section, we describe the techniques used for extracting a useful Haskell library from the contract management code, which is written and reasoned about in Coq. Then, in Section 4.2, we demonstrate how the extracted code can be used as a drop-in library for contract management in a larger framework, written in Haskell.

4.1 Extracting Haskell Code from Coq

Extracting Haskell representations of Coq's native inductive datatypes, such as naturals, booleans, and reals, results in inefficient code and functionality that does not easily integrate with other Haskell code. Instead, we give extract instructions to Coq for making use of Haskell integers, Booleans, and doubles, along with appropriate Haskell operations on these data types. For the same reasons, Coq option values are extracted into Haskell `Maybe` values, Coq sum values are extracted into Haskell `Either` values, and Coq strings are extracted into Haskell `String` values. The extraction is automated in a series of `Makefile` targets and is set up so to ease co-development of Coq code and Haskell code that depends on extracted code.

4.2 Using the Extracted Contract Module from Haskell

The Haskell code that is extracted from Coq does not include a specific interface; all functionality is exported equally. In order to secure the internals of the module and, for instance, prevent users from directly using the constructors, a wrapper module with separate interface functions wraps the extracted module.

The interface of the extracted library in Haskell is given in Figure 7.
The interface provides functionality for constructing boolean and real expressions as well as combinators for constructing contracts. The accumulator combinators are provided with a higher-order abstract syntax interface. Figure 8 lists the Haskell code for the Asian Option from Section 2.1, which makes use of an accumulator expression for maintaining an average.

4.3 A Monadic Front-End Embedding

As discussed in Section 3.2, a straightforward use of the contract combinators does not guarantee that the defined contracts are causal. This problem can be alleviated by providing a monadic interface to contract construction. The idea is to abandon the *translate* combinator and instead have a monad keep track of the current time. An implementation of a monadic contract interface is shown in Figure 9. Inside the monad, the provided *wait* combinator can be used to pause the contract for a number of days. Notice that the interface does not provide a counterpart to the *both* combinator for ordinary contracts; instead, the monadic bind operator is the natural composer for monadic contracts. The monadic contract interface is not as generic as the ordinary contract interface as it does not allow for general use of a construct similar to the *ifWithin* contract combinator. The interface does support conditional contracts, however, through the use of the *ifm* combinator.

The *toContract* function translates a monadic contract into an ordinary contract, which is always causal by construction. A formal proof of this property in Coq is left fot future work, together with an extension of the monadic contract interface that supports a counterpart to the *ifWithin* contract combinator.

```
type Observable = String
type Currency   = String
type Party      = String
type Choice     = String

data BinOp  = Add | Mult | Subt | Div | Min | Max
data Cmp    = EQU | LTH | LTE
data BoolOp = And | Or

−− HOAS support
data ZeroT
data Vars a v
class Elem v a where inj :: v → a

−− Real expressions
type Rexp' a
type Rexp = Rexp' ZeroT −− closed expressions
rLit  :: Double → Rexp' a
rBin  :: BinOp → Rexp' a → Rexp' a → Rexp' a
rNeg  :: Rexp' a → Rexp' a
rObs  :: Observable → Int → Rexp' a
rAcc  :: (forall v. (forall a'. Elem v a' ⇒ Rexp' a')
                    → Rexp' (Vars a v))
         → Int → Rexp' a → Rexp' a

−− Boolean expressions
type Bexp' a
type Bexp = Bexp' ZeroT −− closed expressions
bLit  :: Bool → Bexp' a
bObs  :: Choice → Int → Bexp' a
rCmp  :: Cmp → Rexp → Rexp → Bexp' a
bNot  :: Bexp' a → Bexp' a
bBin  :: BoolOp → Bexp' a → Bexp' a → Bexp' a
bAcc  :: (forall v. (forall a'. Elem v a' ⇒ Bexp' a')
                    → Bexp' (Vars a v))
         → Int → Bexp' a → Bexp' a

−− Contracts
data Contract
zero      :: Contract
transfer  :: Party → Party → Currency → Contract
scale     :: Rexp → Contract → Contract
translate :: Int → Contract → Contract
both      :: Contract → Contract → Contract
ifWithin  :: Bexp → Int → Contract → Contract
             → Contract

−− Environments
type Inp a = Int → Observable → Maybe a
type ObsEnv = Inp Double
type ChoiceEnv = Inp Bool
type Env = (ObsEnv, ChoiceEnv)
obsEnvEmpty    :: ObsEnv
envEmpty       :: Env
choiceEnvEmpty :: ChoiceEnv

−− Contract management
type Trans = Party → Party → Currency → Double
advance    :: Contract → Env → Maybe (Contract,Trans)
specialise :: Contract → Env → Contract
horizon    :: Contract → Int
```

Figure 7. Interface for the Haskell Extracted Contract Library.

5. Related Work

There are several strands of related work. First, there is a bulk of related work concerning domain specific languages for contract management originating with the pioneering work by Peyton-Jones, Eber, and Seward on Composing Contracts [17]. This work has evolved into the company LexiFi that has implemented the techniques on top of LexiFi's MLFi variant of OCaml [15]. The resulting contract management platform runs worldwide in many financial institutions through its integration in key financial institutions, such as Bloomberg[3], and with large asset-management platforms, such as SimCorp Dimension [18], a wall-to-wall asset-management platform for financial institutions.

Based also on earlier work on contract languages [2], in the last decade, domain specific languages for contract specifications have been widely adopted by the financial industry, in particular in the

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

```
iff be c1 c2 = ifWithin be 0 c1 c2

option = translate 90 (iff (bObs "X" 0) trade zero)

trade  = scale (rLit 100)
           (both (transfer "Y" "X" "USD") pay)

pay    = scale rate (transfer "X" "Y" "DKK")

rate   = rBin Div (rAcc (λr →
                          rBin Add r
                            (rObs "FX USD/DKK" 0)
                        )
                        30 (rLit 0)
                   )
                   (rLit 30)
```

Figure 8. Haskell Code for the Asian Option from Section 2.1, Based on the Extracted Contract Library.

```
data CM a = CM ((a→Int→Contract)→Int→Contract)
instance Monad CM where
    return a = CM (λk i → k a i)
    CM f »= g =
      CM (λk i → f (λa i' →
                          case g a of
                            CM h → h k i') i)
    f » m = f »= (λ _ → m)

rObserve :: Observable → CM Rexp
rObserve s = CM (λk i → k (rObs s i) i)

bObserve :: Observable → CM Bexp
bObserve s = CM (λk i → k (bObs s i) i)

transf :: Party → Party → Currency → CM ()
transf a b c =
   CM (λk i → both (transfer a b c) (k () i))

wait :: Int → CM ()
wait t = CM (λk i → translate t (k () i))

skip :: CM ()
skip = return ()

terminate :: CM ()
terminate = CM (λk i → zero)

ifm :: Bexp → CM a → CM a → CM a
ifm b (CM m1) (CM m2) =
  CM (λk i → ifWithin b 0 (m1 k i) (m2 k i))

toContract :: CM () → Contract
toContract (CM m) = m (λ _ _ → zero) 0
```

Figure 9. A Monadic Contract Interface.

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 provide for more agility, shorter time-to-market for new products, and increased assurance of software quality.

The concept of multi-party contracts have been investigated earlier in [1]. See also [7] for an overview of resources related to domain specific languages for the financial domain.

A second line of related work includes verification work in general and verification and certification work in Coq that makes use of the possibility for extracting Haskell or OCaml code [6, 8, 14]. Among the most noticeable pieces of work in this area is the CompCert C compiler developed by Xavier Leroy [12]. The CompCert C compiler consists of more than 40.000 lines of Coq code, which include both the definitions of the compilation phases, specifications of the semantics of the intermediate languages, and proofs that the compilation phases are meaning preserving [13]. Two approaches are taken to provide proofs that each of the individual phases are meaning preserving. The first approach is a direct approach, which establishes the meaning preservation by direct proofs that the translation algorithm is meaning preserving. In the second approach, called the *translation validator approach*, a proof of meaning preservation is established separately from the translation itself. In our work on the verification of the contract library, we make use only of the direct verification approach. Another noticeable piece of work in this area is the work on Vellvm [19], a verification framework for LLVM intermediate representation (IR) transformations [11]. This work provides several different semantics for the LLVM IR and allows IR transformations and analyses to be expressed and proved correct in Coq. Verified transformations and analyses can then be extracted into OCaml code, which (by using the OCaml bindings for LLVM) can be injected into the compiler phases of a real compiler chain.

A third line of related work is work on achieving highly parallel implementations of valuations of financial contracts and portfolios. When a significant part of a portfolio consist of sufficiently complicated, so-called over-the-counter (OTC), contracts, Monte Carlo methods are needed to model the underlying stochastic processes and thereby valuate (i.e., price) the portfolio [10]. In their simplest form, Monte Carlo simulations are embarrassingly parallel, but for valuating portfolios containing so-called path-dependent contracts that reference correlated underlying observables, parallelisation is not so straightforward [16]. For demonstration purposes, we are working towards integrating the certified contract management library with proper parallel pricing techniques.

6. Conclusions and Future Work

We have presented a symbolic framework for modeling financial contracts, capable of expressing common FX and other derivatives, and equipped with a precise cash-flow semantics in two variants. 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 dependencies and horizon, and gradually evolved until the horizon has been reached, following a reduction semantics based on gradually-available external knowledge in an environment.

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

As future work, we plan to explore and model more symbolic contract transformations that are central to day-to-day contract management, and to extend the contract analyses with features relevant for contract valuation. Furthermore, 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 values that need to be modeled in a pricing engine and should be complemented by a code generator for the payoff of a contract, to be integrated with a Monte Carlo pricing engine developed in prior work within HIPERFIT [16].

References

[1] J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen. Compositional specification of commercial contracts. *Interna-*

tional Journal on Software Tools for Technology Transfer, 8(6):485–516, 2006.

[2] B. Arnold, A. V. 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. IEEE, 1995.

[3] J.-P. Bernardy and N. Pouillard. Names for free: polymorphic views of names and binders. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, Haskell '13, pages 13–24, New York, NY, USA, 2013. ACM.

[4] 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.

[5] R. S. Bird and R. Paterson. De bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, Jan. 1999.

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

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

[8] J.-C. Filliâtre and P. Letouzey. Extraction of programs in Objective Caml and Haskell. In T. C. D. Team, editor, *The Coq Proof Assistant, Reference Manual*. INRIA, online, France, 2012. available only online.

[9] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19:27–45, 1 2009.

[10] J. C. Hull. *Options, Futures and Other Derivatives (9th Edition)*. Prentice Hall, 2014.

[11] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[12] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, January 2006.

[13] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[14] P. Letouzey. Extraction in Coq: An overview. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *Logic and Theory of Algorithms*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer-Verlag, 2008.

[15] LexiFi. Contract description language (MLFi). Web page and white paper. `http://www.lexifi.com/technology/contract-description-language`.

[16] C. Oancea, C. Andreetta, J. Berthold, A. Frisch, and F. Henglein. Financial software on GPUs: between Haskell and Fortran. In *Proceedings of International ACM workshop on Functional High-Performance Computing (FHPC'12)*, 2012.

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

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

[19] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *39rd ACM symposium on Principles of Programming Languages*. ACM Press, January 2012.