Towards Certified Compilation of Financial Contracts

Danil Annenkov and Martin Elsman

University of Copenhagen
Dept. of Computer Science (DIKU)
{daan,mael}@di.ku.dk

Abstract

We present an extension to a certified financial contract management system that allows for templated financial contracts and for integration with financial models through verified compilation into so-called payoff-expressions, which readily allow for determining the value of a contract in a given evaluation context, such as contexts created for simulations. The templating mechanism is useful both at the contract specification level, for writing generic reusable contracts, and for reuse of code that, without the templating mechanism, needs to be recompiled for different evaluation contexts. We report on the effect of using the certified system in the context of a GPGPU-based Monte Carlo simulation engine for pricing various over-the-counter (OTC) financial contracts. The full contract-management system, including the payoff-language compilation, is verified in the Coq proof assistant and certified Haskell code is extracted from Coq and integrated with an efficient OpenCL pricing engine.

1 Background and Motivation

New technologies are emerging that have potential for seriously disrupting the financial sector. In particular, blockchain technologies, such as Bitcoins [7] and the Ethereum Smart Contract peer-to-peer platform [9], have entered the realm of the global financial market and it becomes essential to ask to which degree users can trust that the underlying implementations are really behaving according to the specified properties. Unfortunately, the answers are not clear and errors may result in irreversible high-impact events.

The work presented here builds on a series of previous work on specifying financial contracts [1, 2, 4, 6, 8] and in particular on a certified financial contract management engine and its associated contract DSL [3]. This framework allows for expressing a wide variety of financial contracts (a fundamental notion in financial software) and for reasoning about their functional properties (e.g., horizon and causality). As in the previous work, the contract DSL that we consider is equipped with a denotational semantics, which is independent of stochastic aspects and depends only on an *external environment* $\mathsf{ExtEnv} : \mathbb{N} \times \mathsf{Label} \to \mathbb{R} + \mathbb{B}$, which maps observables (e.g., the price of a stock on a particular day) to values. In the work presented here, we present a certified compilation scheme that compiles a contract into a *payoff function*, which aggregates all cashflows in the contract, after discounting them according to some model. The result represents a single "snapshot" value of the contract. The payoff language, which is inspired by traditional payoff languages and is well suited for integration with Monte Carlo simulation techniques for pricing, is essentially a small subset of a C-like expression language enriched with notation for looking up observables in the external environment. We show that compilation from the contract DSL to the payoff language preserves the cashflow semantics.

The contract DSL described in [3], deals with concrete contracts, such as a one year European call option on the AAPL (Apple) stock with strike price $100. The lack of genericity means that each time a new contract is created (even a very similar one), the contract management engine needs to compile the contract into the payoff language and further into a target language for embedding into the pricing engine. To avoid this recompilation problem, we introduce the notion of a *financial instrument*, which allows for templating of contracts and which can be

turned into a concrete contract by instantiating template variables with particular values. For example, a European call option instrument has template parameters such as maturity (the end date of the contract), strike, and the underlying asset that the option is based on. Compiling such a template once allows the engine to reuse compiled code, giving various parameter values as input to the pricing engine.

Moreover, an inherent property of contracts is that they evolve over time. This property is precisely captured by a contract reduction semantics. Each day, a contract becomes a new "smaller" contract, thus, for pricing purposes, contracts need to be recompiled daily, resulting in a dramatic compile time overhead. To avoid recompilation in this case, the generated payoff code is parameterized over the *current time* so that evaluating the payoff code at time t gives us the same result (upto discounting) as first advancing the contract to time t, then compiling it to the payoff code, and then evaluating the result.

The contract analysis and transformation code forms a core code base, which financial software crucially depends on. A certified programming approach using the Coq proof assistant allows us to prove the above desirable properties and to extract certified executable code.

2 The Contract Language

We assume a countably infinite set of program variables, ranged over by v. Moreover, we use n, i, f, and b to range over natural numbers, integers, floating point numbers, and booleans. We use p to range over parties. The contract language that we consider follows the style of [3] and is extended with template variables:

$$c ::= \texttt{zero} \mid \texttt{transfer}(p,p) \mid \texttt{scale}(e,c) \mid \texttt{translate}(t,c) \mid \texttt{checkWithin}(e,t,c,c) \mid \texttt{both}(c,c)$$
$$e ::= op(e,\ e,\ \ldots,\ e) \mid \texttt{obs}(l,i) \mid f \mid b \qquad t ::= n \mid v \qquad op ::= \texttt{add} \mid \texttt{sub} \mid \texttt{mult} \mid \texttt{lt} \mid \texttt{neg} \mid \texttt{cond} \mid \ldots$$

Expressions (e) may contain *observables*, which are interpreted in an external environment. A contract may be empty (`zero`), a transfer of one unit (for simplicity) (`transfer`), a scaled contract (`scale`), a translation of a contract into the future (`translate`), the composition of two contracts (`both`), or a generalized conditional $\texttt{checkWithin}(cond, t, c_1, c_2)$, which checks the condition *cond* repeatedly during the period given by t and evaluates to c_1 if $cond = \texttt{true}$ or to c_2 if *cond* never evaluates to `true` during the period t.

The main difference between the original version of the contract language and the version presented here is the introduction of *template expressions* (t), which, for instance, allows us to write contract templates with the contract maturity as a parameter. This feature requires refined reasoning about the temporal properties of contracts, such as causality. Certain constructs in the original contract language, such as $\texttt{translate}(n, c)$ and $\texttt{checkWithin}(cond, n, c_1, c_2)$, are designed such that basic properties of the contract language, including the property of causality, are straightforward to reason about. In particular, the displacement numbers n in the above constructs are constant positive numbers. For templating, we refine the constructs to support template expressions in place of positive constants. One of the consequences of adding template variables is that the semantics of contracts now depends also on mappings of template variables in a *template environment* $\mathsf{TEnv} : \mathsf{Var} \to \mathbb{N}$, which is also the case for many temporal properties of contracts. For example, the type system for ensuring causality of contracts [3] and the concept of horizon are now parameterized by template environments.

3 The Payoff Intermediate Language

The main motivation behind the payoff language is to bridge the gap between the contract DSL and a traditional expression language, which is usually used to implement pricing engines. The

payoff language should be relatively straightforward to compile to various target languages such as Haskell, Futhark [5], or OpenCL.

$$il ::= \texttt{now} \mid \texttt{model}(l,t) \mid \texttt{if}(il,il,il) \mid \texttt{loopif}(il,il,il,t) \mid \texttt{payoff}(t,p,p) \mid unop(il) \mid binop(il,il)$$
$$unop ::= \texttt{neg} \mid \texttt{not} \qquad binop ::= \texttt{add} \mid \texttt{mult} \mid \texttt{sub} \mid \texttt{lt} \mid \texttt{and} \mid \texttt{or} \mid \ldots \qquad t ::= n \mid i \mid v \mid \texttt{tplus}(t,t)$$

The payoff language is an expression language ($il \in \mathsf{ILExpr}$) with binary and unary operations, extended with conditionals and generalized conditionals `loopif`, behaving similarly to `checkWithin`. Template expressions ($t \in \mathsf{TExprZ}$) in this language are extensions of the template expressions of the contract language with integer literals and addition. Terms in the payoff language can be evaluated given a proper external environment, a proper template environment, and a *discount function* $d : \mathbb{N} \to \mathbb{R}$. The result of the evaluation is a single real value in contrast to the contract language for which the semantics is given in terms of traces.

4 Compiling Contracts to Payoffs

The contract language consist of two levels, namely constructors to build contracts (c) and expressions used in some of these constructors (`scale`, `checkWithin`, etc.). We compile both levels into a single payoff language. The compilation functions $\tau_{\mathrm{e}}[\![-]\!] : \mathsf{Expr} \times \mathsf{TExprZ} \rightharpoonup \mathsf{ILExpr}$ and $\tau_{\mathrm{c}}[\![-]\!] : \mathsf{Contr} \times \mathsf{TExprZ} \rightharpoonup \mathsf{ILExpr}$ are recursively defined on the syntax of expressions and contracts, respectively, taking the starting time $t_0 \in \mathsf{TExprZ}$ as a parameter.

$$\tau_{\mathrm{e}}[\![\texttt{cond}(b,\ e_0,\ e_1])]\!]_{t_0} = \texttt{if}(\tau_{\mathrm{e}}[\![b]\!]_{t_0}, \tau_{\mathrm{e}}[\![e_0]\!]_{t_0}, \tau_{\mathrm{e}}[\![e_1]\!]_{t_0}) \qquad \tau_{\mathrm{c}}[\![\texttt{zero}]\!]_{t_0} = 0$$
$$\tau_{\mathrm{e}}[\![\texttt{obs}(l,i)]\!]_{t_0} = \texttt{model}(l, \texttt{tplus}(t_0,i)) \qquad \tau_{\mathrm{c}}[\![\texttt{translate}(t,c)]\!]_{t_0} = \tau_{\mathrm{c}}[\![c]\!]_{\texttt{tplus}(t_0,t)}$$
$$\tau_{\mathrm{c}}[\![\texttt{transfer}(p_1,p_2)]\!]_{t_0} = \texttt{payoff}(t_0,p_1,p_2) \qquad \tau_{\mathrm{c}}[\![\texttt{both}(c_0,c_1)]\!]_{t_0} = \texttt{add}(\tau_{\mathrm{c}}[\![c_0]\!]_{t_0}, \tau_{\mathrm{c}}[\![c_1]\!]_{t_0})$$
$$\tau_{\mathrm{c}}[\![\texttt{scale}(e,c)]\!]_{t_0} = \texttt{mult}(\tau_{\mathrm{e}}[\![e]\!]_{t_0}, \tau_{\mathrm{c}}[\![c]\!]_{t_0})$$
$$\tau_{\mathrm{c}}[\![\texttt{checkWithin}(e,t,c_1,c_2)]\!]_{t_0} = \texttt{loopif}(\tau_{\mathrm{e}}[\![e]\!]_{t_0}, \tau_{\mathrm{c}}[\![c_0]\!]_{t_0}, \tau_{\mathrm{c}}[\![c_1]\!]_{t_0}, t)$$

Let $\mathcal{E}[\![e]\!] : \mathsf{ExtEnv} \times \mathsf{TEnv} \rightharpoonup \mathbb{R} + \mathbb{B}$, $\mathcal{C}[\![c]\!] : \mathsf{ExtEnv} \times \mathsf{TEnv} \rightharpoonup \mathbb{N} \to \mathsf{Party} \times \mathsf{Party} \to \mathbb{R}$, and $\mathcal{IL}[\![il]\!] : \mathsf{ExtEnv} \times \mathsf{TEnv} \times (\mathbb{N} \to \mathbb{R}) \times \mathsf{Party} \times \mathsf{Party} \rightharpoonup \mathbb{R} + \mathbb{B}$ define the semantics of the contract expression sublanguage, the semantics of contracts, and the semantics of the payoff language, respectively. We also assume a function $HOR : \mathsf{Contr} \times \mathsf{TEnv} \to \mathbb{N}$ that returns a conservative upper bound on the length of a contract. The compilation function satisfies the following properties:

Theorem 1 (Soundness)**.** *Assume parties p_1 and p_2 and discount function $d : \mathbb{N} \to \mathbb{R}$.*

- *If $\tau_e[\![e]\!]_0 = il$ and $\mathcal{E}[\![c]\!]_{\rho,\delta} = v_1$ and $\mathcal{IL}[\![il]\!]_{\rho,\delta,d,p_1,p_2} = v_2$ then $v_1 = v_2$.*
- *If $\tau_c[\![c]\!]_0 = il$ and $\mathcal{C}[\![c]\!]_{\rho,\delta} = trace$, where $trace : \mathbb{N} \to \mathsf{Party} \times \mathsf{Party} \to \mathbb{R}$, and $\mathcal{IL}[\![il]\!]_{\rho,\delta,d,p_1,p_2} = v$ then $\sum_{t=0}^{HOR(c,\delta)} d(t) \times trace(t)(p_1,p_2) = v$.*

To avoid recompilation of a contract when time moves forward, we define a function $\mathsf{cutPayoff}()$. This function is defined recursively on the syntax of intermediate language expressions. The only interesting case is the case for `payoff` expressions:

$$\mathsf{cutPayoff}(\texttt{payoff}(t,p_1,p_2)) = \texttt{if}(\texttt{lt}(t,\texttt{now}), 0, \texttt{payoff}(t,p_1,p_2))$$

The function guards the `payoff` expression with a condition guarding whether this payoff should have effect. For the remaining cases, the function recurses on subexpressions and returns otherwise unmodified expressions.

Avoiding recompilation can significantly improve performance especially on GPGPU devices. On the other hand, additional conditionals are introduced, which results in a number of additional checks at runtime. Experiments conducted with "hand-compiled" OpenCL code, which

was semantically equivalent to the payoff language code, show that for the simple contracts, like European options, additional conditions, introduced by cutPayoff() do not significantly influence performance. The estimated overhead was around 2.5 percent, while compilation time is in the order of a magnitude bigger than the total execution time.

5 Conclusion

This work extends the certified contract management system of [3] with template expressions, which allows for drastic performance improvements and reusability in terms of the concept of instruments. Along with changes to the contract language, we developed a formalization of the payoff intermediate language in Coq. Our approach introduces the abstract syntax of the payoff language as an inductive data type with the semantics of the payoff language and the compilation from the contract DSL defined as partial functions (using Coq's Option data type).

A number of important properties (including soundness) of the translation from contracts to the payoff language have been proven in Coq. Moreover, Coq's code extraction mechanism is used to obtain a certified compiler implementation in the Haskell programming language. We are currently working on establishing the important property of the new time-parameterized payoff evaluation function, which states that given a contract c and its compilation into payoff code il, evaluating il at time t gives us the same result (upto discounting) as first advancing c to time t, then compiling it to payoff code, and then evaluating the result at time 0.

References

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

[2] B.R.T 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.

[3] Patrick Bahr, Jost Berthold, and Martin Elsman. Certified symbolic management of financial multi-party contracts. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP'2015, pages 315–327, September 2015.

[4] Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(1):27–45, 2009.

[5] Troels Henriksen, Martin Elsman, and Cosmin E Oancea. Size slicing: a hybrid approach to size inference in Futhark. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 31–42. ACM, 2014.

[6] Tom Hvitved, Felix Klaedtke, and Eugen Zalinescu. A trace-based model for multiparty contracts. *The Journal of Logic and Algebraic Programming*, 81(2):72–98, 2012.

[7] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[8] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP'2000, September 2000.

[9] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2015. Homestead revision, Founder, Ethereum & Ethcore, gavin@ethcore.io.