Certified Compilation of Financial Contracts

Danil Annenkov, Martin Elsman

▶ **To cite this version:**

Danil Annenkov, Martin Elsman. Certified Compilation of Financial Contracts. PPDP '18 - 20th International Symposium on Principles and Practice of Declarative Programming, Sep 2018, Frankfurt am Main, Germany. pp.1-13, 10.1145/3236950.3236955 . hal-01883559

HAL Id: hal-01883559

https://hal.inria.fr/hal-01883559

Submitted on 2 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Certified Compilation of Financial Contracts

Danil Annenkov
INRIA
GALLINETTE Research team
danil.annenkov@inria.fr

Martin Elsman
University of Copenhagen
Dept. of Computer Science (DIKU)
mael@di.ku.dk

Abstract

We present an extension to a certified financial contract management system that allows for templated declarative financial contracts and for integration with financial stochastic models through verified compilation into so-called payoff-expressions. Such expressions readily allow for determining the value of a contract in a given evaluation context, such as contexts created for stochastic 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 our Coq development along with Futhark code for use in a data-parallel pricing engine.

CCS Concepts

• **Theory of computation** → **Program verification**; *Logic and verification*; • **Software and its engineering** → **Correctness**; **Domain specific languages**; *Source code generation*;

Keywords

financial contracts, contract languages, domain-specific languages, certified programming, software correctness, Coq

ACM Reference Format:
Danil Annenkov and Martin Elsman. 2018. Certified Compilation of Financial Contracts. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3236950.3236955

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPDP '18, September 3–5, 2018, Frankfurt am Main, Germany
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6441-6/18/09. . . $15.00
https://doi.org/10.1145/3236950.3236955

1 Background and Motivation

New technologies are emerging that have potential for seriously disrupting the financial sector. In particular, blockchain technologies, such as Bitcoins [17] and the Ethereum Smart Contract peer-to-peer platform [23], 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.

Contract description languages and payoff languages are used in large scale financial applications [15, 22], although formalisation of such languages in proof assistants and certified compilation schemes are much less explored.

The work presented here builds on a series of previous work on specifying financial contracts [1, 3, 7, 11, 20] and in particular on a certified financial contract management engine and its associated contract DSL [4]. 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* $\texttt{ExtEnv} : \mathbb{N} \times \texttt{Label} \to \mathbb{R} + \mathbb{B}$, which maps observables (e.g., the price of a stock on a particular day) to values. As the first contribution of this work, 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 is inspired by traditional payoff languages and is well suited for integration with Monte Carlo simulation techniques for pricing. It is essentially a small expression language featuring arithmetic and boolean operators, a limited form of a looping construct, and 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 language described in [4], 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. As our second contribution, 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, an 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 at each time step, resulting in a dramatic compilation time overhead. As a third contribution, we introduce a mechanism allowing for avoiding recompilation in relation to contract evolution. A payoff expression can be parameterised over the *current time* so that evaluating the payoff code at time t gives us the same result (up to discounting) as first advancing the contract to time t, then compiling it to the payoff code, and then evaluating the result. Most of the payoff languages used in real-world applications require synchronization of the contract and the payoff code once a contract evolves [16, Contract State and Pricing Synchronization]. In some cases, however, as we mentioned earlier, it is important to capture the reduction semantics in the payoff language as well. Our result allows for using a single compilation procedure for both use cases: compiling a contract upfront and synchronizing at each time step.

The contract analyses and the contract transformation procedures form 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.

We summarize the contributions of this paper as follows:

- We present an extended domain-specific language for expressing financial contracts, called CL, based on the work by [4]. The extended contract language features contract templates, also called *instruments*. The extension allows for parameterisation of contracts with respect to temporal parameters.
- Inspired by traditional payoff languages, we develop a payoff intermediate language, which we demonstrate is well-suited for the integration with Monte Carlo simulation techniques.
- We use the Coq proof assistant to develop a certified compilation procedure of contract templates into a parameterised payoff intermediate language.
- We further parameterise the compiled payoff expressions with the notion of "current time" allowing for capturing the evolution of contracts with the passage of time.
- We develop the proof of an extended soundness theorem in the Coq proof assistant. The theorem establishes a correspondence between the time-parameterised compilation scheme and the contract reduction semantics.
- We demonstrate how the parametric payoff code allows for better performance due to avoiding recompilation with the change of parameters.

$$
\begin{array}{rcl}
c \in \texttt{Contr} & ::= & \texttt{zero} \mid \texttt{transfer}(p_1, p_2, a) \mid \texttt{scale}(e, c) \\
 & \mid & \texttt{translate}(t, c) \mid \texttt{ifWithin}(e, t, c_1, c_2) \\
 & \mid & \texttt{both}(c, c) \\
e \in \texttt{Exp} & ::= & op(e_1,\ e_2,\ \ldots,\ e_n) \mid \texttt{obs}(l, i) \mid f \mid b \\
t \in \texttt{TExp} & ::= & n \mid v \\
op \in \texttt{Op} & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{mult} \mid \texttt{lt} \mid \texttt{neg} \mid \texttt{cond} \mid \ldots
\end{array}
$$

Figure 1: Syntax of contracts, contracts expressions, and template expressions.

2 The Contract Language

The contract language that we consider follows the style of [4] and is a declarative contract language that allows for expressing contractual agreements among parties regarding immediate and future cash flows between the parties. Contracts may refer to observable values and cash flows may thereby depend on the development of such observables, which may include stock prices, interest rates, and decisions made by parties.

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 and a to range over assert symbols (e.g., EUR, USD, and so on). The contract language is given in Figure 1; it follows closely the contract language of [4], but is extended with template variables. Expressions ($e \in \texttt{Exp}$) 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 $\texttt{ifWithin}(cond, t, c_1, c_2)$, 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 $\texttt{ifWithin}(cond, n, c_1, c_2)$, 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* $\texttt{TEnv} : \texttt{Var} \to \mathbb{N}$, which is also the case for many temporal properties of contracts. For example, the type system for ensuring causality of contracts [4] and the concept of horizon are now parameterized by template environments.

Let us consider a few examples of contracts written in English and expressed in CL.

Example 1. A *European option* is a contract that gives the owner the right, but not the obligation, to buy or sell an underlying security at a specific price, known as the strike price, on the option's expiration date (investopedia.com).

Let us take the expiration date to be 90 days into the future and set the strike at USD 100. In CL, we can implement the European option contract with the above parameters as follows:

```
translate(90,
  if(obs(AAPL,0) > 100.0,
    scale(obs(AAPL,0) - 100.0, transfer(you, me, USD)),
    zero))
```

Example 2. A Three month FX swap for which the payment schedule has been settled is easily expressed in CL:

```
scale(1.000.000,
  both(
    all[translate(22, transfer(me, you, EUR)),
        translate(52, transfer(me, you, EUR)),
        translate(83, transfer(me, you, EUR))],
    scale(7.21,
     all[translate(22, transfer(you, me, DKK)),
         translate(52, transfer(you, me, DKK)),
         translate(83, transfer(you, me, DKK))])))
```

In the swap-example, we have written $\texttt{all[}c_1, \cdots, c_n\texttt{]}$ as an abbreviation for the contract $\texttt{both(}c_1, \texttt{both(}\cdots, c_n\texttt{))}$. We use the `all` shortcut with the `translate` combinator to implement a schedule of payments.

Using the CL template extension, we can abstract some of the contract parameters in Example 1 to template variables (T for expiration date, and S for strike)[1]:

```
translate(T,
  if(obs(AAPL,0) > S,
    scale(obs(AAPL,0) - S, transfer(you, me, USD)),
    zero))
```

This possibility for parameterisation plays well with how users would interact with a contract management system. Contract templates could be exposed to users as so-called *instruments*, which a user can instantiate to contracts by supplying concrete values for parameters.

We extend the denotational semantics from [4] to accommodate the idea of template expressions. The semantics for the expression sublanguage stays unchanged, since such expressions do not contain template expressions. That is, the semantics for an expression $e \in \texttt{Exp}$ in Figure 1 is given by the partial function $\mathcal{E}[\![e]\!] : [\![\Gamma]\!] \times \texttt{Env} \rightharpoonup [\![\tau]\!]$. On the other hand, we modify the semantic function for contacts by adding a template environment as an argument:

$$\mathcal{C}[\![c]\!] : [\![\Gamma]\!] \times \texttt{Env} \times \texttt{TEnv} \rightharpoonup \texttt{Trace}$$
$$\texttt{Trace} = \mathbb{N} \to \texttt{Trans}$$
$$\texttt{Trans} = \texttt{Party} \times \texttt{Party} \times \texttt{Asset} \to \mathbb{R}$$

As the original contract semantics, it depends on the external environment $\texttt{Env} : \mathbb{N} \times \texttt{Label} \to \mathbb{R} \cup \mathbb{B}$ and variable assignments that map each free variable of type τ to a value in $[\![\tau]\!]$ with $[\![\texttt{Real}]\!] = \mathbb{R}$ and $[\![\texttt{Bool}]\!] = \mathbb{B}$. Given a typing environment Γ, the set of *variable assignments* in Γ, written $[\![\Gamma]\!]$, is the set of all partial mappings γ from variable names to $\mathbb{R} \cup \mathbb{B}$ such that $\gamma(x) \in [\![\tau]\!]$ iff $x : \tau \in \Gamma$. The typing rules also remain the same for expressions and for contracts.

The semantics for template expressions $\mathcal{T}[\![t]\!] : \texttt{TEnv} \to \mathbb{N}$ is defined as follows:

$$\mathcal{T}[\![n]\!]_\delta = n \qquad \mathcal{T}[\![v]\!]_\delta = \delta(v)$$

We modify the semantics of contract constructors that depend on template expressions in such a way that the corresponding template expression is evaluated using $\mathcal{T}[\![-]\!]$. For example for the `translate` constructor, we have

$$\mathcal{C}[\![\texttt{translate}(t, c)]\!]_{\gamma,\rho,\delta} = \mathit{delay}(\mathcal{T}[\![T]\!]_\delta, \mathcal{C}[\![c]\!]_{\gamma,\rho,\delta})$$

Where $\mathit{delay} : \mathbb{N} \times \texttt{Trace} \to \texttt{Trace}$ is an operation that delays a given trace by a number of time steps (see [4, Figure 4]).

We define an *instantiation function* that takes a contract and a template environment containing values for template variables, and produces another contract that does not contain template variables by replacing all occurrences of template variables with corresponding values from the template environment.

Definition 1 (Instantiation function)**.**

$$\begin{aligned}
\mathsf{inst} &: \texttt{Contr} \times \texttt{TEnv} \to \texttt{Contr} \\
\mathsf{inst}(\texttt{zero}, \delta) &= \texttt{zero} \\
\mathsf{inst}(\texttt{let } e \texttt{ in } c, \delta) &= \mathsf{inst}(c, \delta) \\
\mathsf{inst}(\texttt{transfer}(p_1, p_2, \texttt{a}), \delta) &= \texttt{transfer}(p_1, p_2, \texttt{a}) \\
\mathsf{inst}(\texttt{scale}(e, c), \delta) &= \mathsf{inst}(c, \delta) \\
\mathsf{inst}(\texttt{translate}(t, c), \delta) &= \texttt{translate}(\mathcal{T}[\![t]\!]_\delta, \mathsf{inst}(c, \delta)) \\
\mathsf{inst}(\texttt{both}(c_1, c_2), \delta) &= \texttt{both}(\mathsf{inst}(c_1, \delta), \mathsf{inst}(c_2, \delta)) \\
\mathsf{inst}(\texttt{ifWithin}(e, t, c_1, c_2), \delta) &= \\
&\texttt{ifWithin}(e, \mathcal{T}[\![t]\!]_\delta, \mathsf{inst}(c_1, \delta), \mathsf{inst}(c_2, \delta))
\end{aligned}$$

We further define an inductive predicate that holds only for contract expressions without template variables (Figure 2). We call such contracts *template-closed*.

It is straightforward to establish the following fact.

Lemma 1. *For any contract c and template environment δ, application of the instantiation function gives a template-closed contract:*

$$\mathcal{TC}(\mathsf{inst}(c, \delta))$$

[1] In our implementation, we focus on contract templates allowing for template expressions to represent temporal parameters, such as maturity. Other parameters, such as a strike, can be expressed as constant observable values.

$$\boxed{\mathcal{TC}(c)}$$

$$\frac{}{\mathcal{TC}(\texttt{zero})} \qquad \frac{\mathcal{TC}(c)}{\mathcal{TC}(\texttt{let}\ e\ \texttt{in}\ c)} \qquad \frac{}{\mathcal{TC}(\texttt{transfer}(p_1, p_2, \texttt{a}))}$$

$$\frac{\mathcal{TC}(c)}{\mathcal{TC}(\texttt{scale}(e, c))} \qquad \frac{n \text{ is a numeral} \qquad \mathcal{TC}(c)}{\texttt{translate}(n, c)}$$

$$\frac{\mathcal{TC}(c_1) \qquad \mathcal{TC}(c_2)}{\mathcal{TC}(\texttt{both}(c_1, c_2))} \qquad \frac{n \text{ is a numeral} \qquad \mathcal{TC}(c_1) \qquad \mathcal{TC}(c_2)}{\mathcal{TC}(\texttt{ifWithin}(e, n, c_1, c_2))}$$

Figure 2: Template-closed contracts.

LEMMA 2 (INSTANTIATION SOUNDNESS). *For any contract c, template environments δ and δ', external environment ρ, and any value environment γ, the contract c and $\mathsf{inst}(c, \delta)$ are semantically equivalent. That is, $\mathcal{C}[\![c]\!]_{\gamma,\rho,\delta} = \mathcal{C}[\![\mathsf{inst}(c, \delta)]\!]_{\gamma,\rho,\delta'}$.*

The reduction semantics of the contract language presented in [4] remains the same, although, we make additional assumption that the contract expression is closed with respect to template variables.

3 The Payoff Intermediate Language

The contract language allows for capturing different aspects of financial contracts. We consider a particular use case for the contract language, where one wants to calculate an estimated price of a contract according to some stochastic model by performing simulations. Simulations is often implemented using Monte Carlo techniques, for instance, by evaluating a contract price at current time for randomly generated possible market scenarios and discounting the outcome according to some model. A software component that implements such a procedure is called a *pricing engine* and aims to be very efficient in performing large amount of calculations by exploiting the parallelism [2]. For this use case, one has to take the following aspects into account:

- Contracts should be represented as simple functions that take prices of assets involved in the contract (randomly generated by a pricing engine) and return one value corresponding to the aggregated outcome of the contract.
- The resulting value of the contract should be discounted according to a given discount function.

One way of achieving this would be to implement an interpreter for the contract language as part of a pricing engine. Although this approach is quite general, interpreting a contract in the process of pricing will cause significant performance overhead. Moreover, it will be harder to reason about correctness of the interpreter, since it could require non-trivial encoding in languages targeting GPGPU devices. For that reason we take another approach: translating a contract from CL to an intermediate representation and, eventually, to a function in the pricing engine implementation language. We call this intermediate representation a *payoff language* and expressions in this language we call *payoff expressions*.

We would like the payoff language to contain fewer domain-specific features and being closer to a subset of some general purpose language, making a mapping from the payoff language to a target language straightforward. We demonstrate how payoff expressions can be translated to Haskell and Futhark [8, 9] in Section 6.

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

The payoff language is an expression language ($il \in \texttt{ILExpr}$) with binary and unary operations, extended with conditionals and generalized conditionals `loopif`, behaving similarly to `ifWithin`. Template expressions ($t \in \texttt{TExprZ}$) in this language are extensions of the template expressions of the contract language with integer literals and addition.

The semantics of payoff expressions is given in Figure 3. We use the notation $\overline{\mathcal{P}} = (\rho, \delta, t_0, t, d, p_1, p_2)$ for the vector of arguments to the semantic function. The following notation $\overline{\mathcal{P}}[t_0 := t_0'] = (\rho, \delta, t_0', t, d, p_1, p_2)$ is used to show that the respective argument in the vector receives a certain value. The semantics depends on environments $\rho \in \texttt{Env}$ and $\delta \in \texttt{TEnv}$ similarly to the semantics of the contract language. Payoff expressions can evaluate to a value of type $\mathbb{N}$, $\mathbb{R}$, or $\mathbb{B}$ (in contrast to the contract language for which the semantics is given in terms of traces.). We add $\mathbb{N}$ to the semantic domain, because we need to interpret the `now` construct, which represents the "current time" parameter and template expressions t^e. The semantics also depends on a *discount function* $d : \mathbb{N} \to \mathbb{R}$. The $t_0 \in \mathbb{N}$ parameter is used to add relative time shifts introduced by the semantics of `loopif`; t is a current time, which will be important later, when we introduce a mechanism to cut payoffs before a certain point in time.

The semantics for unary and binary operations is a straightforward mapping to corresponding arithmetic and logical operations, provided that the arguments have appropriate types. For example, $[\![\texttt{add}]\!](v_1, v_2) = v_1 + v_2$, if $v_1, v_2 \in \mathbb{R}$.

The semantic function $\mathcal{IL}[\![-]\!]$ considers only payoffs between two parties p_1 and p_2, which are given as the last two parameters. More precisely, it considers payoffs from party p_1 to party p_2 as positive and as negative, if payoffs go in the opposite direction. Another way of defining the semantics could be a *bilateral view* on payoffs. In this case only cashflows to or from one fixed party to any other party are considered. Then, the semantics for the `payoff` construct would be defined as follows:

$$\mathcal{IL}[\![\texttt{payoff}(t, p_1', p_2')]\!]_{\rho,\delta,t_0,t,d,p} = \begin{cases} d(\mathcal{T}[\![t]\!]_\delta) & \text{if } p_2' = p \\ -d(\mathcal{T}[\![t]\!]_\delta) & \text{if } p_1' = p \\ 0 & \text{otherwise} \end{cases}$$

$$\boxed{\mathcal{IL}\,[\![il]\!] : \texttt{Env} \times \texttt{TEnv} \times \mathbb{N} \times \mathbb{N} \times (\mathbb{N} \to \mathbb{R}) \times \texttt{Party} \times \texttt{Party} \rightharpoonup \mathbb{N} \cup \mathbb{R} \cup \mathbb{B}}$$

$$\begin{aligned}
\overline{\mathcal{P}} &= (\rho, \delta, t_0, t, d, p_1, p_2) \\
\mathcal{IL}\,[\![t^e]\!]_{\overline{\mathcal{P}}} &= \mathcal{T}\,[\![t^e]\!]_\delta + t_0 \\
\mathcal{IL}\,[\![unop(il)]\!]_{\overline{\mathcal{P}}} &= [\![unop]\!]\,(\mathcal{IL}\,[\![il]\!]_{\overline{\mathcal{P}}}) \\
\mathcal{IL}\,[\![binop(il_0, il_1)]\!]_{\overline{\mathcal{P}}} &= [\![binop]\!]\,(\mathcal{IL}\,[\![il_0]\!]_{\overline{\mathcal{P}}},\ \mathcal{IL}\,[\![il_1]\!]_{\overline{\mathcal{P}}}) \\
\mathcal{IL}\,[\![\texttt{model}(l, t^e)]\!]_{\overline{\mathcal{P}}} &= \rho(l, \mathcal{T}\,[\![t^e]\!]_\delta + t_0) \\
\mathcal{IL}\,[\![\texttt{now}]\!]_{\overline{\mathcal{P}}} &= t \\
\mathcal{IL}\,[\![if(il_0, il_1, il_2)]\!]_{\overline{\mathcal{P}}} &= \begin{cases} \mathcal{IL}\,[\![il_1]\!]_{\overline{\mathcal{P}}}, & \text{if } \mathcal{IL}\,[\![il_0]\!]_{\overline{\mathcal{P}}} = true \\ \mathcal{IL}\,[\![il_2,]\!]_{\overline{\mathcal{P}}}, & \text{if } \mathcal{IL}\,[\![il_0]\!]_{\overline{\mathcal{P}}} = false \end{cases}
\end{aligned}$$

$$\mathcal{IL}\,[\![\texttt{payoff}(t^e, p_1', p_2')]\!]_{\overline{\mathcal{P}}} = \begin{cases} d(\mathcal{T}\,[\![t^e]\!]_\delta) & \text{if } p_1' = p_1, p_2' = p_2 \\ -d(\mathcal{T}\,[\![t^e]\!]_\delta) & \text{if } p_1' = p_2, p_2' = p_1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{IL}\,[\![\texttt{loopif}(il_0, il_1, il_2, t^e)]\!]_{\overline{\mathcal{P}}} = iter(\mathcal{T}\,[\![t^e]\!]_\delta, t_0), where$$

$$iter(n, t_0') = \begin{cases} \mathcal{IL}\,[\![il_1]\!]_{\overline{\mathcal{P}}[t_0 := t_0']}, \\ \quad \text{if } \mathcal{IL}\,[\![il_0]\!]_{\overline{\mathcal{P}}[t_0 := t_0']} = true \\ \mathcal{IL}\,[\![il_2]\!]_{\overline{\mathcal{P}}[t_0 := t_0']}, \\ \quad \text{if } \mathcal{IL}\,[\![il_0]\!]_{\overline{\mathcal{P}}[t_0 := t_0']} = false \wedge n = 0 \\ iter(n-1)(t'+1), \\ \quad \text{if } \mathcal{IL}\,[\![il_0]\!]_{\overline{\mathcal{P}}[t_0 := t_0']} = false \wedge i > 0 \end{cases}$$

Figure 3: Semantics of payoff expressions.

4 Compiling Contracts to Payoffs

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

$$\begin{aligned}
\tau_e\,[\![\texttt{cond}(b,\ e_0,\ e_1])]\!]_{t_0} &= \texttt{if}(\tau_e\,[\![b]\!]_{t_0}, \tau_e\,[\![e_0]\!]_{t_0}, \tau_e\,[\![e_1]\!]_{t_0}) \\
\tau_e\,[\![\texttt{obs}(l, i)]\!]_{t_0} &= \texttt{model}(l, \texttt{tplus}(t_0, i)) \\
\tau_c\,[\![\texttt{transfer}(p_1, p_2, a)]\!]_{t_0} &= \texttt{payoff}(t_0, p_1, p_2) \\
\tau_c\,[\![\texttt{scale}(e, c)]\!]_{t_0} &= \texttt{mult}(\tau_e\,[\![e]\!]_{t_0}, \tau_c\,[\![c]\!]_{t_0}) \\
\tau_c\,[\![\texttt{zero}]\!]_{t_0} &= 0 \\
\tau_c\,[\![\texttt{translate}(t, c)]\!]_{t_0} &= \tau_c\,[\![c]\!]_{\underline{\texttt{tplus}}(t_0, t)} \\
\tau_c\,[\![\texttt{both}(c_0, c_1)]\!]_{t_0} &= \texttt{add}(\tau_c\,[\![c_0]\!]_{t_0}, \tau_c\,[\![c_1]\!]_{t_0})
\end{aligned}$$

$$\tau_c\,[\![\texttt{ifWithin}(e, t, c_1, c_2)]\!]_{t_0} = \texttt{loopif}(\tau_e\,[\![e]\!]_{t_0}, \tau_c\,[\![c_0]\!]_{t_0}, \tau_c\,[\![c_1]\!]_{t_0}, t)$$

$$\underline{\texttt{tplus}}(t_1, t_2) = \begin{cases} t_1 + t_2 \text{ if } t_1, t_2 \text{ are numerals} \\ \texttt{tplus}(t_1, t_2) \text{ otherwise} \end{cases}$$

The important point to notice here is that all relative time shifts in an expression in CL are accumulated to the t_0 parameter. The resulting payoff expression only contains lookups in the external environment where time is given explicitly, and does not depend on nesting of time shifts as it was in the case of $\texttt{translate}(t, c)$ in CL. Such a representation allows for a more straightforward evaluation model.[2] We also would like to emphasise that `acc` and `let` constructs are not supported by our compilation procedure. On the supported subset of the contract language, the compilation functions $\tau_e\,[\![e]\!]$, and $\tau_c\,[\![c]\!]$ are total. We use the *smart constructor* $\underline{\texttt{tplus}}$ that adds two integer literals whenever it is possible or returns a syntactic addition expression. This use of smart constructors is useful not only for optimisation purposes; it also allows us to overcome some difficulties in formalisation (see Remark 1 in Section 5).

Example 3. We consider the following contract (`t0` and `t1` denote template variables): the party "you" transfer to the party "me" 100 USD in `t0` days in the future, and after `t1` more days "you" transfers to "me" an amount equal to the difference between the current price of the AAPL stock and 100 USD, provided that the price of AAPL is higher then 100 USD (we use infix notation for arithmetic operations to make code more readable).

```
c =
  translate(t0,
  both(scale(100.0, transfer(you,me)),
       translate(t1,
         if(obs(AAPL,0) > 100.0,
            scale(obs(AAPL,0) - 100.0, transfer(you, me)),
            zero)))
```

This contract compiles to the following code in the payoff intermediate language:

```
e =
  (100.0 * payoff(t0,you,me)) +
  if (model(AAPL,t0+t1) > 100.0,
      (model(AAPL,t0+t1) - 100.0) * payoff(t0+t1,you,me),
      0.0)
```

As one can see, all nested occurrences of the `translate` construct were accumulated from top to bottom. That is, in the `if` case, we calculate payoffs and lookup for values of the AAPL stock at time `(t0+t1)`.

[2]The `loopif` construct still introduces relative time shifts similarly to `ifWithin`. This makes code generation in a target language less trivial. Potentially, it is possible to decompose `loopif` into an iteration and a conditional expression. Also, in the case when number of iterations of `loopif` is a number and not a template variable, it is possible to completely unroll `loopif` into nested conditional expressions, making all the indexing into the external environment explicit.

To be able to reason about soundness of the compilation process, one needs to make a connection between the semantics of the two languages. For the expression sublanguage of CL ($e \in \texttt{Exp}$) we can just compare the values that the original expression and the compiled expression evaluates to. In case of the contract language ($c \in \texttt{Contr}$) the situation is different, since the semantics of a contract is given in terms of a `Trace`, and an expression in the payoff intermediate language evaluates to a single value. However, we know that the compiled expression represents the sum of the contract cashflows after discounting.

We assume a function $HOR : \texttt{TEnv} \times \texttt{Contr} \to \mathbb{N}$ that returns a conservative upper bound on the length of a contract. We often write $\tau_e \llbracket e \rrbracket_0 = il$, or $\tau_c \llbracket c \rrbracket_0 = il$ to emphasise that the compilation function returns some result. The value environment $\llbracket \Gamma \rrbracket$ is not relevant for the present development and we will omit it. The compilation function satisfies the following properties:

Theorem 4 (Soundness). *Assume parties p_1 and p_2 and discount function $d : \mathbb{N} \to \mathbb{R}$, environments $\rho \in \texttt{Env}$, and $\delta \in \texttt{TEnv}$.*

(i) If $\tau_e \llbracket e \rrbracket_0 = il$ and $\mathcal{E} \llbracket e \rrbracket_{\rho,\delta} = v_1$ and $\mathcal{IL} \llbracket il \rrbracket_{\rho,\delta,0,0,d,p_1,p_2} = v_2$ then $v_1 = v_2$.

(ii) If $\tau_c \llbracket c \rrbracket_0 = il$ and $\mathcal{C} \llbracket c \rrbracket_{\rho,\delta} = tr$, where $tr : \mathbb{N} \to \texttt{Party} \times \texttt{Party} \to \mathbb{R}$ then

$$\sum_{t=0}^{\text{HOR}_\delta(c)} d(t) \times tr(t)(p_1, p_2) = \mathcal{IL} \llbracket il \rrbracket_{\rho,\delta,0,0,d,p_1,p_2}$$

Theorem 4 makes an assumption that the compiled expression evaluates to some value. We do not develop a type system for our payoff language to ensure this property. Instead, we show that it is sufficient for a contract to be well-typed to ensure that the compiled expression always evaluates to some value (for details, we refer the reader to the typing rules for the contract language in [4]).

Theorem 5 (Total semantics for compiled contracts). *Assume parties p_1 and p_2 and discount function $d : \mathbb{N} \to \mathbb{R}$, well-typed external environment $\rho \in \texttt{Env}$, template environment $\delta \in \texttt{TEnv}$, and typing context Γ. The following two properties hold:*

(i) for any $e \in \texttt{Exp}$, $t_0 \in \texttt{TExprZ}$, $t'_0 \in \mathbb{N}$, if $\Gamma \vdash e : \tau$ $\tau_e \llbracket e \rrbracket_{t_0} = il$, then

$$\exists v,\ \mathcal{IL} \llbracket il \rrbracket_{\rho,\delta,t'_0,0,d,p_1,p_2} = v, \quad \text{and } v \in \llbracket \tau \rrbracket$$

(ii) for any $c \in \texttt{Contr}$, t_0 t'_0, if $\Gamma \vdash c$ and $\tau_c \llbracket c \rrbracket_{t_0} = il$, then

$$\exists v,\ \mathcal{IL} \llbracket il \rrbracket_{\rho,\delta,t'_0,0,d,p_1,p_2} = v, \quad \text{and } v \in \mathbb{R}$$

Notice that Theorem 5 holds for any $t_0 \in \texttt{TExprZ}$ and $t'_0 \in \mathbb{N}$. These parameters do not affect totality of the semantics and can be arbitrary, since we assume that the external environment is total. Theorems 4 and 5 together ensure that our compilation procedure produces a payoff expression that evaluates to a value reflecting the aggregated price of a contract after discounting.

4.1 Avoiding recompilation

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.

$$\mathsf{cutPayoff} : \texttt{ILExpr} \to \texttt{ILExpr}$$

$$\mathsf{cutPayoff}(\texttt{now}) = \texttt{now}$$

$$\mathsf{cutPayoff}(\texttt{model}(l,t)) = \texttt{model}(l,t)$$

$$\mathsf{cutPayoff}(unop(il)) = unop(\mathsf{cutPayoff}(il))$$

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

$$\mathsf{cutPayoff}(binop(il_1,il_2)) = binop(\mathsf{cutPayoff}(il_1), \mathsf{cutPayoff}(il_2))$$

$$\mathsf{cutPayoff}(\texttt{if}(il_1,il_2,il_3)) = \texttt{if}(\mathsf{cutPayoff}(il_1), \mathsf{cutPayoff}(il_2), \mathsf{cutPayoff}(il_3))$$

The most important case is the case for `payoff`. The function wraps `payoff` with a condition guarding whether this payoff affects the resulting value. For the remaining cases, the function recurses on subexpressions and returns otherwise unmodified expressions.

Example 6. Let us consider Example 3 again and apply the $\mathsf{cutPayoff}()$ function to the expression `e`:

```
cutPayoff(e) =
  (100.0 * disc(t0) * if(t0 < now, 0, payoff(you,me)) +
  if (model(AAPL,t0+t1) > 100.0,
      (model(AAPL,t0+t1) - 100.0) * disc(t0+t1) *
       if(t1+t0 < now, 0, payoff(you,me)),
      0.0)
```

Each `payoff` in the payoff expression is now guarded by the condition, comparing the time of the particular payoff with `now`. Notice that the templates variables `t0` and `t1` are mapped to concrete values in the template environment.

To be able to state a soundness property for the $\mathsf{cutPayoff}()$ function we again need to find a way to connect it to the semantics of CL. Since $\mathsf{cutPayoff}()$ deals with the dynamic behavior of the contract with respect to time, it seems natural to formulate the soundness property in this case in terms of contract reduction ([4, Figure 10]). The semantics of the payoff language takes the "current time" t as a parameter. We should be able to connect the t parameter to the step of contract reduction.

Theorem 7 (Contract compilation soudness wrt. contract reduction). *We assume parties p_1, p_2, discount function $d : \mathbb{N} \to \mathbb{R}$. For any well-typed and template-closed contract c, i.e., we assume $\Gamma \vdash c$, and $\mathcal{TC}(c)$, an external environment $\rho' \in \texttt{Env}$ extending a partial external environment $\rho \in \texttt{Env}_\texttt{p}$, if c steps to some c' by the reduction relation $c \overset{T}{\Longrightarrow}_\rho c'$, for some transfer $T \in \texttt{Trans}$, such that*

$\mathcal{C}\,[\![c']\!]_{(\rho'/1),\emptyset} = trace$, *and* $\tau_c\,[\![c]\!]_0 = il$, *then*

$$\sum_{t'=0}^{\text{HOR}_\delta(c')} d(t'+1) \times trace(t') = \mathcal{IL}\,[\![\mathsf{cutPayoff}(il)]\!]_{\rho',\emptyset,0,1,d,p_1,p_2}$$

where $\rho'/1$ *denotes the external environment* ρ *advanced by one time step:*

$$\rho'/1 = \lambda(l,i).\ \rho'(l,i+1), \quad l \in \texttt{Label}, i \in \mathbb{Z}$$

From the contract pricing perspective, the partial external environment ρ contains *historical data* (e.g., historical stock quotes) and the extended environment ρ' is a union of the two environments ρ and ρ'', where ρ'' contains *simulated data*, produced by means of simulation in the pricing engine (e.g., using Monte Carlo techniques).

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 run-time. We have investigated the influence on performance of these additional conditions for certain contracts. The results of experiments are given in Section 6.2.

One also might be interested in the following property. The following two ways of using our compilation procedure give identical results:

- first reduce a contract (move time forward), compile it to a payoff expression, then evaluate the payoff expression;
- first compile a contract to a payoff expression, apply cutPayoff() to the payoff expression, and then evaluate, specifying the appropriate value for the "current time" parameter.

Let us introduce some notation first. We fix the well-typed external environment ρ, the partial environment ρ', which is historically complete ($\rho'(l,i)$ is defined for all labels l and $i \leq 0$), and a discount function $d : \mathbb{N} \to \mathbb{R}$. Next, we assume that contracts are well-typed, and closed with respect to template variables, the compilation function is applied to supported constructs only, and that the reduction function, corresponding to the reduction relation, is total on ρ' (see [4, Theorem 11]). This gives us the following total functions:

$$red_{\rho'} : \texttt{Contr} \to \texttt{Contr}$$
$$\tau_c\,[\![-]\!]_0 : \texttt{Contr} \to \texttt{ILExpr}$$

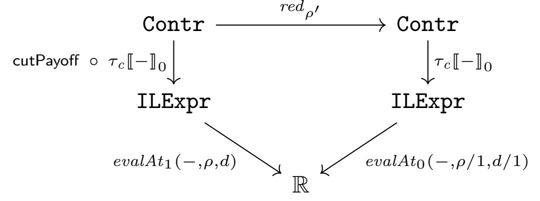
These functions correspond to the contract reduction function and the contract compilation function. We also define an evaluation function for compiled payoff expressions as a shortcut for the payoff expression semantics.

$$evalAt_{-} : \mathbb{N} \to \texttt{ILExpr} \times \texttt{Env} \times \texttt{Disc} \to \mathbb{R} + \mathbb{B}$$
$$evalAt_t(e,\rho,d) = \mathcal{IL}\,[\![e]\!]_{\rho,\emptyset,0,t,d,p_1,p_2}$$

for some parties p_1 and p_2. We know by Theorem 5 that *evalAt* is total on payoff expressions produced by the compilation function from well-typed contracts.

We summarise the property by depicting it as a commuting diagram.

Theorem 8. *The following diagram commutes:*



Here we write $\rho/1$ *and* $d/1$ *for shifted one step external environment and discount function, respectively.*

The above diagram gives rise to the following equation:

$$evalAt_1(-,\rho,d) \circ \mathsf{cutPayoff} \circ \tau_c\,[\![-]\!]_0 = evalAt_0(-,\rho/1,d/1) \circ \tau_c\,[\![-]\!]_0 \circ red_{\rho'}$$

This property shows that our implementation can be used in two different ways. Either a contract is compiled upfront with cutPayoff() or a contract is reduced, at each time of interest, to another contract, which is then compiled to a payoff expression for evaluation. The second use case allows for more flexibility for users. For example, one can develop a system where users define contracts directly in terms of CL working in a specialised IDE. The first case allows for compiling upfront a set of predefined financial instruments (or contract templates) avoiding recompilation when time moves forward. Adding a new instrument is possible, but requires recompilation.

The statements of Theorems 7 and 8 generalise in the obvious way to n-step reduction (by replacing one-step reduction with n-step reduction, replacing environment shifts from $\rho/1$ to ρ/n, and evaluating the compiled payoff expression at $t = n$ instead of $t = 1$). The crucial step for proving these generalised theorems is to use the following theorem.

Theorem 9 (Soundness of cutPayoff() for n time steps). *Assume parties* p_1 *and* p_2*, a discount function* $d : \mathbb{N} \to \mathbb{R}$*, and a well-typed external environment* $\rho \in \texttt{Env}$*. For any well-typed and template-closed contact* c *at a time step* $n \in \mathbb{N}$*, if* $\tau_c\,[\![c]\!]_0 = il$ *and* $\mathcal{C}\,[\![c]\!]_{\rho,\emptyset} = tr$ *then*

$$\sum_{t=n}^{\text{HOR}_\delta(c)} d(t) \times tr(t)(p_1,p_2) = \mathcal{IL}\,[\![\mathsf{cutPayoff}(il)]\!]_{\rho,\emptyset,0,n,d,p_1,p_2}$$

Theorem 9 expresses soundness of n-step cutPayoff() evaluation without explicitly mentioning contract reduction and leads to a better proof structure in the Coq formalisation. Intuitively, this theorem says that the sum of the trace starting at n instead of zero is exactly the value we obtain after evaluating cutPayoff(il) at current time $t = n$.

Theorem 9 combined with the properties of n-step reduction gives us the proofs of Theorems 7 and 8. Our Coq development (see Section 5) contains a full formalisation of these generalised theorems.

5 Formalisation in Coq

Our formalisation in Coq[3] extends the previous work [4] by introducing the concept of template expressions and by

[3] The formalisation presented in this paper is available online: https://github.com/annenkov/contracts. The repository includes the backends

developing a certified compilation technique for translating contracts to payoff expressions. The required modifications to the denotational semantics have been presented in Section 2. These modifications required us to propagate changes to all the proofs affected by the change of syntax and semantics. We start this section with a description of the original formalisation, and then continue with modifications and additions made by the authors of this work.

The formalisation described in [4] uses an extrinsic encoding of CL. That means that syntax is represented using Coq's inductive data types, and a typing relation on these *raw* terms are given separately. For example, the type of the expression sublanguage is defined as follows.

```
Inductive Exp : Set :=
  OpE (op : Op) (args : list Exp)
| Obs (l : ObsLabel) (i : Z)
| VarE (v : Var)
| Acc (f : Exp) (d : nat) (e : Exp).
```

One of the design choices in the definition of `Exp` is to make the constructor of operations `OpE` take "code" for an operation and the list of arguments. Such an implementation makes adding new operations somewhat easier. Although, we would like to point out that this definition is a *nested* inductive definition (see [5, Section 3.8]). In such cases Coq cannot automatically derive a strong enough induction principle, which means that it needs to be defined manually. In the case of `Exp` it is not hard to see, that one needs to add a generalised induction hypothesis in case of `OpE`, saying that some predicate holds for all elements in the argument list.

Although the extrinsic encoding requires more work in terms of proving, it has a big advantage for code extraction, since simple inductive data types are easier to use in the Haskell wrapper for CL.

One of the consequences of this encoding is that semantic functions for contracts `Contr` and expressions `Exp` are partial, since they are defined on raw terms which may not be well-typed. This partiality is implemented with the `Option` type, which is equivalent to Haskell's `Maybe`. To structure the usage of these partial functions, we define the `Option` monad and use monadic binding

```
bind : forall A B : Type,
       option A → (A → option B) → option B
```

to compose calls of partial functions together. The functions

```
liftM: forall A B : Type,
       (A → B) → option A → option B
liftM2 : forall A B C : Type,
         (A → B → C) → option A → option B
         → option C
liftM3 : forall A B C D : Type,
         (A → B → C → D) → option A → option B
         → option C → option D
```

allow for a total function of one, two, or three arguments to be lifted to the `Option` type. The implementation includes poofs of some properties of `bind` and the lifting functions. These properties include cases for which an expression evaluates to some value.

```
bind_some : forall (A B : Type) (x : option A)
                   (v : B) (f : A → option B),
                x >>= f = Some v
                → exists x' : A, x = Some x' ∧ f x' = Some v
```

Similar lemmas were proved for other functions related to the `Option` type. To simplify the work with the `Option` monad, the implementation defines tactics in the `Ltac` language (part of Coq's infrastructure). The tactics `option_inv` and `option_inv_auto` use properties of operations like `bind` and `liftM` to invert hypotheses like `e = Some v`, where e contains the aforementioned functions. The implementation uses some tactics from [21]. Particularly, the `tryfalse` tactic is widely used. It tries to resolve the current goal by looking for contradictions in assumptions, which conveniently removes impossible cases.

The original formalisation of the contract language has been modified by introducing the type of template expressions

```
Parameter TVar : Set.
Inductive TExpr : Set :=
    Tvar (t : TVar)
  | Tnum (n : nat).
```

We keep the type of variables abstract and do not impose any restrictions on it. Although one could add decidability of equality for `TVar`, if required, we do not compare template variables in our formalisation. We modify the definition of the type of contracts `Contr` such that constructors of expressions related to temporal aspects now accept `TExpr` instead of `nat` (`If` corresponds to `ifWithin`):

```
Translate : TExpr → Contr → Contr
If : Exp → TExpr → Contr → Contr → Contr.
```

We leave the other constructors unmodified.

Similarly to how we define an external environment, we define a *template environment* as a function type `TEnv := TVar → nat`. Such a definition allows for easier modification of existing code base in comparison with partial mappings. According to the definitions in Section 2, we modify the semantic function for contracts, and the symbolic horizon function, to take an additional parameter of type `TEnv`. Propagation of these changes was not very problematic and almost mechanical. Because the first attempt to parameterise the reduction relation with a template environment led to some problems, we decided to define the reduction relation only for template-closed contracts. In most cases it is sufficient to instantiate a contract, containing template variables using the instantiation function (Definition 1), and then reduce it to a new contract. Although instantiation requires a template environment containing all the mappings for template variables mentioned in the contract, we do not consider this a big limitation.

The definition of the payoff intermediate language (following Section 4) also uses an extrinsic encoding to represent raw terms as an inductive data type. We define one type

generating Haskell and Futhark code along with the pricing engine implementation in Futhark for benchmarking.

for the payoff language expressions ILExpr, since there is no such separation as in CL on contracts and expressions. The definition of template expressions used in the definition of ILExpr is an extended version of the definition of template expressions TExpr used in the contract language definition.

```
Inductive ILTExpr : Set :=
  ILTplus (e1 : ILTExpr) (e2 : ILTExpr)
| ILTexpr (e : TExpr).

Inductive ILTExprZ : Set :=
  ILTplusZ (e1 : ILTExprZ) (e2 : ILTExprZ)
| ILTexprZ (e : ILTExpr)
| ILTnumZ (z : Z).
```

Notice that we use two different types of template expressions ILTExpr and ILTExprZ. The former extends the definition of TExpr with the addition operation, and the latter extends it further with integer literals and with the corresponding addition operation (recall that template expressions used in CL can be either natural number literals or variables). The reason why we have to extend TExpr with addition is that we want to accumulate time shifts introduced by Translate in one expression using (syntactic) addition. In the expression sublanguage of CL, observables can refer to the past by negative time indices. For that reason we introduce the ILTExprZ type.

The full definition of syntax for the payoff intermediate language in our Coq formalisation looks as follows:

```
Inductive ILExpr : Set :=
| ILIf : ILExpr → ILExpr → ILExpr → ILExpr
| ILFloat : R → ILExpr
| ILNat : nat → ILExpr
| ILBool : bool → ILExpr
| ILtexpr : ILTExpr → ILExpr
| ILNow :  ILExpr
| ILModel : ObsLabel → ILTExprZ → ILExpr
| ILUnExpr : ILUnOp → ILExpr → ILExpr
| ILBinExpr : ILBinOp → ILExpr → ILExpr → ILExpr
| ILLoopIf : ILExpr → ILExpr → ILExpr → TExpr → ILExpr
| ILPayoff : ILTExpr → Party → Party → ILExpr.
```

Notice that we use template expressions, which could represent negative numbers (ILTExprZ) in the constructor ILModel. This constructor corresponds to observable values in the contract language and allows for negative time indices corresponding to access of historical data.

We could have generalised our formalisation to deal with different types of template variables and added a simple type system on top of the template expression language, but we decided to keep our implementation simple, since the main goal was to demonstrate that it is possible to extend the original contract language to contract templates with temporal variables.

All the theorems and lemmas described in the paper are completely formalised in our Coq development. We use a limited amount of proof automation in the soundness proofs. The proof automation is used mainly in the proofs related to compilation of the contract expression sublanguage, since compilation is straightforward and proofs are relatively easily to automate. Moreover, without the proof automation, one would have to consider a large number of very similar cases leading to code duplication. In addition to option_inv_auto mentioned above, we use a tactic that helps to get rid of cases where expressions (a source expression in Exp and a target expression in ILEpxr) evaluate to values of different types (denoted by the corresponding constructor).

```
Ltac destruct_vals :=
  repeat (match goal with
          | [x : Val |- _] ⇒ destruct x; tryfalse
          | [x : ILVal |- _] ⇒ destruct x; tryfalse
          end).
```

Here the Val and IVal types correspond to values of the contract expression sublanguage and the payoff expression language respectively. The tryfalse tactic searches for the contradictions in the goal (see [21]).

Another tactic that significantly reduces the complexity of the proofs is the omega tactic from Coq's standard library. This tactic implements a decision procedure for expressions in Presburger arithmetic. That is, goals can be equations or inequations of integers, or natural numbers with addition and multiplication by a constant. The tactic uses assumptions from the current context to solve the goal automatically.

The principle we use in the organisation of the proofs is to use proof automation to solve the most trivial and tedious goals and to be more explicit about the proof structure in cases requiring more sophisticated reasoning.

Remark 1. The first version of the soundness proof was developed for the original contract language without template expressions. The proof was somewhat easier, since the aggregation of nested time shifts introduced by $\texttt{translate}(n, c)$ constructs during compilation was implemented as addition of natural numbers, corresponding to time shifts. In the presence of template expressions, the compilation function builds a syntactic expression using the tplus constructor. There are some places in proofs where it was crucial to use associativity of addition to prove the goal, but this does not work for template expressions. For example, $\texttt{tplus}(\texttt{tplus}(t_1, t_2), t_3)$ is not equal to $\texttt{tplus}(t_1, \texttt{tplus}(t_2, t_3))$, because these expressions represent different syntactic trees, although semantically equivalent. Instead of restating proofs in terms of this semantic equivalence (significantly complicating the proofs), we used the following approach. The compilation function uses the smart constructor $\underline{\text{tplus}}$ instead of just plain construction of the template expression. This allowed us to recover the property we needed to complete the soundness proof without altering too much of its structure.

There are a number of aspects that introduce complications to the development of proofs of the compilation properties.

- Accumulation of relative time shifts during compilation. To obtain a general enough induction hypothesis we have to generalise our lemmas to take as parameter an initial time t0. The same holds for the semantics of

loopif, since there is an additional parameter in the semantics to implement iterative behavior.
- Presence of template expressions. The complications we faced due to template expressions are described in Remark 1. We have resolved these complications with smart constructors, but template expressions still add some overhead.
- Conversion between types of numbers. We use integers and natural numbers (nat and Z type from the standard library of Coq). In some places, including the semantics of template expressions, we use a conversion from natural numbers to integers. This conversion makes automation with the omega tactic more complicated, because it requires first to use the properties of conversion, which is harder to automate. With the accumulation aspect, conversions add even more overhead.
- We use the definition of contract horizon in the statement of the soundness theorems, which leads to additional case analysis in proofs.

5.1 Code Extraction

The Coq proof assistant allows for extracting Coq functions into programs in some functional languages [14]. The implementation described in [4] supports code extraction of the contract type checker and contract manipulation functions into the Haskell programming language. We extend the code extraction part of the implementation with features related to contract templates and contract compilation. Particularly, we extract Haskell implementations of the following functions:

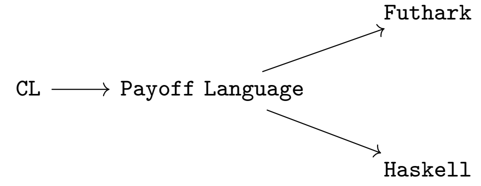
- inst_contr function that instantiates a given contract according to given template environment;
- fromExp function for compiling the contract expression sublanguage;
- fromContr function for compiling contract language constructs;
- cutPayoff function for parameterising a payoff expression with the "current time".
- ILsem semantic function for payoff expressions, which can be used as an interpreter.

For supporting templates, we have updated the Haskell front end and exposed the full contract language in a convenient form. We have kept the original versions of extended combinators, such as translate and within without changes and added translateT and withinT combinators, which support template variables.

Our implementation contains an extended collection of contract examples, examples of contract compilation, and evaluation of resulting payoff expressions.

6 Code Generation

To exemplify how the payoff language can be used to produce a payoff function in a subset of some general purpose language, we have implemented a code generation procedure to Haskell and Futhark, as illustrated in the following diagram:



We make use of the code extraction mechanism described in Section 5.1 to obtain a certified compilation function, which we use to translate expressions in CL to expressions in the payoff language.

6.1 The Haskell Backend

The code generation procedure is (almost) a one-to-one mapping of the payoff language constructs to Haskell expressions. One primitive, which we could not map directly to Haskell build-in functions was the loopif construct. We have solved this issue by implementing loopif as a higher-order function in Haskell. The implementation essentially follows the definition of the semantics of loopif in Coq:

```
loopif :: Int → Int → (Int → Bool) → (Int → a) → (Int → a) → a
loopif n t0 b e1 e2 = let b' = b t0 in
                      case b' of
                       True → e1 t0
                       False → case n of
                               0 → e2 t0
                               _ → loopif (n−1) (t0+1) b e1 e2
```

The resulting payoff function has the following signature:

```
payoff :: Map.Map ([Char], Int) Double → Map.Map [Char] Int
          → Int → Party → Party → Double
```

That is, the function takes as parameters an external environment, a template environment, the current time, and two parties. The payoff function calls the payoffInternal function, which takes an additional parameter—an initial value for the loopif function, which serves as a loop counter.

Example 10. We apply the code generation procedure to the expression e from Example 3. Here is the result of code generation:

```
module Examples.PayoffFunction where
import qualified Data.Map as Map
import BaseTypes
import Examples.BasePayoff

payoffInternal ext tenv t0 t_now p1 p2 =
   (100.0 ∗ (if  (X== p1 && Y== p2) then 1
             else if (X== p2 && Y== p1) then −1 else 0)) +
   (if ((100.0 < (ext Map.!  ("AAPL",(0 + (tenv Map.! "t1") +
                                 (tenv Map.!  "t0") + 0+ t0)))))
    then (((( ext Map.!  ("AAPL",(0 + (tenv Map.! "t1") +
                             (tenv Map.!  "t0") + 0+ t0))) ∗ 100.0)
          ∗ (if  (X== p1 && Y== p2) then 1 else
             if  (X== p2 && Y== p1) then −1 else 0))) else 0.0)

payoff ext tenv t_now p1 p2 = payoffInternal ext tenv 0 t_now p1 p2
```

The external environment and the template environment are represented using Haskell's Data.Map, and Map.! is an infix notation for the lookup function. To obtain the code above we apply a simple optimisation, replacing the loopif with zero as the first argument with the regular if. One could also

add more optimisations to our Coq implementation along with proofs of soundness.

A module declaring the `payoff` function can be used as an ordinary Haskell module as a part of the development requiring the payoff functions. For example, it could be used in the context of the FinPar benchmark [2], which contains a Haskell implementation of pricing among other routines. Moreover, the cutPayoff() function can be used to obtain a parameterised version of a payoff function in Haskell, allowing us to reproduce the contract reduction behavior.

6.2 The Futhark Backend

Futhark is a data-parallel functional language for programming nested, regular programs to be executed efficiently on a GPU [8, 9]. The language has a rich core language, which provides a number of second-order functional array combinators, such as `map`, `reduce`, `filter`, and `scan`, but it also provides seemingly imperative features, including sequential loops and array updates, which are based on a uniqueness type system that allows for an efficient implementation of functional array updates. On top of the core language, Futhark is enriched with a higher-order module language, for which constructs are compiled away at compile time due to a static interpretation technique.

Generating code for Futhark is quite similar to the code generation approach described for Haskell. With the Futhark backend, however, the aim is to integrate generated payoff functions with an efficient parallel Monte Carlo based pricing engine, which is achieved by making the pricing engine a parameterised module that takes as argument a module containing a payoff-function.

Regarding the particular Futhark payoff function generation, the implementation differs from the Haskell implementation in two ways, namely (i) with respect to the representation of the `loopif` construct and (ii) with respect to external environment access.

The first difference is related to the fact that Futhark does not support recursive functions, but instead includes various iteration constructs. The payoff language `loopif` construct is therefore compiled into a Futhark `while` loop construct. For example, consider the following payoff expression (corresponding to a simple contract with a barrier):

```
loopif(model("AAPL", 0) <= 4000.0,
       0.0, 2000.0 * payoff(0,X,Y), t)
```

This payoff expression is translated into the following fragment of Futhark code:

```
let payoffInternal(ext : [][] f32, tenv : [] i32,
                   disc : [] f32, t0 : i32, t_now : i32) : f32 =
  let t0 = loop t0 = t0
      while (!( ext[t0,0]  <= 4000.0) && (t0 < tenv[0])) do t0+1
      in if (ext[t0,0]  <= 4000.0)
         then 0.0
         else (2000.0 * disc[t0])

let payoff ext tenv t_now = payoffInternal(ext,tenv,0,t_now)
```

The `ext` variable is a two-dimensional array containing model data (the first index corresponds to time and the second corresponds to an observable), `tenv` is an array with template parameter values, and `disc` is an array containing discount factors (indexed by time).

The second difference, which is related to the way we work with the model data environment, is concerned with translation of environment indexing to the form used in the FinPar pricing code. For example, we translate time indices 100 and 200 in the following payoff expression

```
payoff(100,X,Y) + payoff(200,X,Y)
```

to 0 and 1 respectively. This reindexing corresponds to the order in which time indices appear in a payoff expression.

The output of the generation procedure is a Futhark module that can be directly passed to the parameterised Futhark pricing engine module. A key feature of the implemented template mechanism combined with the cutPayoff() functionality is that the code base needs to be compiled into efficient GPU code only when new instruments are introduced; the generated payoff expressions are generic with respect to the time at which the price is calculated.

Table 1 shows the timings for pricing three different financial contracts using the FinPar Monte Carlo pricing engine [2]. The contracts include a *vanilla European call option*, which allows a holder at some time t to purchase a particular stock at a predetermined price, and a *discrete barrier option*, which forces a holder to exercise the option before maturity if any of three particular underlying stocks at certain dates cross certain barrier levels. Finally, the contracts include a *double vanilla European option*, which allows a holder to exercise any of two European options on two different underlying stocks. The three contracts cover well the possible scope of supported contracts, including the support for dealing with multiple underlyings and multiple measurement days. Moreover, the contracts are instances of real financial contracts appearing in real financial portfolios.

The experiments were executed on a commodity MacBook Pro laptop with a 2.7GHz Intel i7 CPU and an AMD Radeon Pro 460 GPU using `futhark-bench`, which was configured to report the average runtime of five different runs. Pricing of the Vanilla option is based on 8388608 individual Monte Carlo simulation paths, whereas pricing of the two other contracts is based on 1048576 individual paths. The Fut-C column shows timings for executables generated using `futhark-c`, the CPU sequential-code compiler for Futhark. The Fut-OpCL and the Fut-OpCL-Cut columns show timings for executables generated with `futhark-opencl` and with the Fut-OpCL-Cut column providing timings for the case where the cutPayoff() functionality allows for pricing of the contract at different times during the contract's lifetime. The experiments show that for the vanilla European option, a speedup of roughly 310 was achieved comparing the Futhark program compiled into C (and further into x86 machine code) with a version compiled into OpenCL using Futhark's OpenCL backend.

There are a number of observations to draw from the benchmark results. First, notice that the speedup obtained from using the commodity GPU instead of the laptop CPU

Table 1: Price timings in milliseconds. The measurements show the time it takes to price three different financial contracts using the FinPar Monte Carlo based generic pricing. The Fut-C column specifies sequential performance and the Fut-OpCL and the Fut-OpCL-Cut specify parallel performance with the Fut-OpCL-Cut column showing timings with the cutPayoff() functionality enabled.

	Fut-C	Fut-OpCL	Fut-OpCL-Cut
Vanilla option	6,779.4ms	21.4ms	21.7ms
Barrier option	1,521.7ms	54.7ms	55.1ms
Double option	983.0ms	12.4ms	12.3ms

ranges from a factor of 27 to a factor of 317.[4] Second, notice that the introduction of the cutPayoff() function in the Fut-OpCL-Cut column has neglectable impact on performance. We can therefore conclude that, at least for the contracts represented by the three examples, the template feature makes it possible to avoid recompilation of pricing code and that the generalisation can have a dramatical positive effect on the performance of risk calculations, each of which often consists of thousands of pricing tasks.

7 Related Work

There is a large body of work related to using domain specific languages for specifying and managing financial contracts [4, 10, 11, 15, 19, 20, 22] and for specifying financial contract payoff expressions [7]. Only parts of this work investigate the certification aspects of the devised solutions [4]. Compared to the previous work, the present work considers how declarative certified contracts can be compiled into generic payoff functions for efficient use in a practical pricing framework.

Another line of related work investigates the possibility of implementing financial contracts on distributed ledgers such as blockchains [6]. Included in this work is work on establishing a certified foundation for executing programs (also called smart contracts) on such architectures [18].

Finally, there is a large body of related work on developing techniques for certifying implementations of programming languages, including the seminal work on CompCert [13], a fully certified compiler for the C programming language and the verified LLVM project [24], which aims at providing a pluggable toolkit for composing certified LLVM [12] compiler phases.

8 Conclusion

This work extends the certified contract management system of [4] with template expressions, which allows for drastic performance improvements and reusability in terms of the concept of instruments (i.e., contract templates). We consider a practical application of the declarative contract specifications in the context of contract valuation (i.e., pricing). For the purposes of interacting with pricing engines, we introduce a language for payoff expressions (the payoff intermediate language). We have developed a formalisation of the payoff intermediate language and a certified compilation procedure in Coq. Our approach uses an extrinsic encoding, which allows us to make use of Coq's code extraction feature for obtaining a correct implementation of the compiler function that translates expressions in CL to payoff expressions. We have introduced a parameterisation technique for payoff expressions allowing for capturing contract development over time. The developed technique is consistent with the notion of contract reduction from [4].

A number of important properties, including soundness of the translation from CL to the payoff language have been proved in Coq. We have exemplified how the payoff intermediate language can be used to generate code in a target language by mapping payoff expressions to a subset of Haskell and Futhark. We have conducted performance measurements with the generated Futhark code in the context of an efficient parallel pricing engine and shown that, for three types of contracts included in the experiment, the template scheme does not significantly influence performance. On the contrary, the template scheme allows for avoiding recompilation caused by changes to an instrument's parameters and by simplification of a contract due to the passage of time.

There are number of possibilities for future work. First, some work is needed for the payoff intermediate language to support the expression-level accumulation functionality from [4]. As part of a solution, one may consider generalising the somewhat ad-hoc `loopif` construct and, instead, provide a more general language construct for iteration, which could involve compiling `ifWithin` to a combination of iteration and conditions (resulting in simpler target code generation).

Second, the representation of traces as functions $\mathbb{N} \rightarrow$ `Trans` is equivalent to infinite streams of transfers. It would be interesting to explore this idea of using streams further, since observable values also can be naturally represented as streams.

Finally, a possibility for future work is to formalise further the infrastructure for working with external environment representations. For instance, the reindexing scheme used in our Futhark backend for accessing the external environment is currently considered trusted code in the same way as the Futhark pretty printing.

Acknowledgments

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center "HIPERFIT: Functional High Performance Computing for Financial Information Technology" (http://hiperfit.dk) under contract number 10-092299 and by the CoqHoTT ERC Grant 637339. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Danish Strategic Research Council.

[4]The multi-underlying nature of the non-vanilla contracts results in smaller speedups relative to the vanilla case due to the complexity and the sequential dependencies involved in dealing with correlations between underlyings.

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] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. FinPar: A parallel financial benchmark. *ACM Trans. Archit. Code Optim.*, 13(2):18:1–18:27, June 2016.

[3] 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.

[4] 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.

[5] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013.

[6] Benjamin Egelund-Müller, Martin Elsman, Fritz Henglein, and Omri Ross. Automated execution of financial contracts on blockchains. *Business & Information Systems Engineering*, 59(6):457–467, Dec 2017.

[7] 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.

[8] 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.

[9] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

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

[11] 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.

[12] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

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

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

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

[16] LexiFi. Structuring, Pricing, and Processing Complex Financial Products with MLFi. http://www.lexifi.com/files/resources/MLFiWhitePaper.pdf, 2008. White paper.

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

[18] Russell O'Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, PLAS '17, pages 107–120, New York, NY, USA, 2017. ACM.

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

[20] 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.

[21] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2016. Version 4.0. http://www.cis.upenn.edu/~bcpierce/sf.

[22] SimCorp A/S. XpressInstruments solutions. Company whitepaper. Available from http://simcorp.com, 2009.

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

[24] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 427–440, New York, NY, USA, 2012. ACM.