Towards Certified Compilation of Financial Contracts

Danil Annenkov Martin Elsman

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

NWPT, November 2016

Motivation

Why do we need languages for financial contracts?

- precise formulation of a contract
- symbolic contract analysis and transformation
- portfolio management
- input for “pricing engines” through compilation to payoff expressions
- increasing interest in “smart contracts” running on blockchain platforms

Motivation

Why do we need languages for financial contracts?

- precise formulation of a contract
- symbolic contract analysis and transformation
- portfolio management
- input for "pricing engines" through compilation to payoff expressions
- increasing interest in "smart contracts" running on blockchain platforms

Why go "certified"?

- by "certified" we mean that we have a formal proof that a program has the desired properties
- correctness is crucial
- proof assistants can be used to write proofs and even "extract" a correct implementation!

What is a Contract DSL[1]?

- allows for expressing a large variety of financial contracts
- supports multi-party contracts
- has a formal semantics
- contract management operations and transformations are proven correct wrt. the specified semantics
- the contract DSL semantics along with all proofs are formalized in the Coq proof assistant

[1] Patrick Bahr, Jost Berthold, Martin Elsman. Certified Symbolic Management of Financial Multi-Party Contracts, ICFP2015

Contract DSL language constructs

`zero`	empty contract
`transfer`(p_1, p_2)	transfer of one unit
`scale`(e, c)	scaled contract
`translate`(t, c)	translation into the future
`both`(c_1, c_2)	composition of two contracts
`checkWithin`(e, t, c_1, c_2)	generalized conditional

An expression sublanguage (e) features arithmetic and boolean expressions along with *observable* values (stock prices etc.)

Contract DSL : Contract Templates

Template feature extension: The Contract DSL allows template variable instead of just fixed numbers for some language constructs

Contract DSL : Contract Templates

Template feature extension: The Contract DSL allows template variable instead of just fixed numbers for some language constructs

Original	Extended
$\texttt{translate}(n, c)$	$\texttt{translate}(t, c)$
$\texttt{checkWithin}(e, n, c_1, c_2)$	$\texttt{checkWithin}(e, t, c_1, c_2)$

Where $t ::= n \mid v$. Variables v are interpreted in a *template environment* TEnv

Contract DSL : Contract Templates

Template feature extension: The Contract DSL allows template variable instead of just fixed numbers for some language constructs

Original	**Extended**
$\texttt{translate}(n, c)$	$\texttt{translate}(t, c)$
$\texttt{checkWithin}(e, n, c_1, c_2)$	$\texttt{checkWithin}(e, t, c_1, c_2)$

Where $t ::= n \mid v$. Variables v are interpreted in a *template environment* TEnv

Compiling contract templates leads to extensive code reuse

Example

"European options are contracts that give the owner the right, but not the obligation, to buy or sell the underlying security at a specific price, known as the strike price, on the option's expiration date"
(taken from: investopedia.com)

Example

"European options are contracts that give the owner the right, but not the obligation, to buy or sell the underlying security at a specific price, known as the strike price, on the option's expiration date"
(taken from: investopedia.com)

Take: expiration date = 90 days into the future, strike = 100.0

Example

"European options are contracts that give the owner the right, but not the obligation, to buy or sell the underlying security at a specific price, known as the strike price, on the option's expiration date"
(taken from: investopedia.com)

Take: expiration date = 90 days into the future, strike = 100.0

European Call Option

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

Example

"European options are contracts that give the owner the right, but not the obligation, to buy or sell the underlying security at a specific price, known as the strike price, on the option's expiration date"
(taken from: investopedia.com)

Take: expiration date = T days into the future, strike = S

European Call Option Template

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

Contract DSL Semantics

The semantics of a contact is given by a Trace:

$$\mathcal{C}\,[\![c]\!] : \mathsf{ExtEnv} \times \mathsf{TEnv} \rightharpoonup \mathsf{Trace}$$

The Trace is a mapping from time to transfers between parties:

$$\mathsf{Trace} = \mathbb{N} \rightarrow \mathsf{Party} \times \mathsf{Party} \rightarrow \mathbb{R}$$

Contract DSL Semantics

The semantics of a contact is given by a Trace:

$$\mathcal{C}\,[\![c]\!] : \mathsf{ExtEnv} \times \mathsf{TEnv} \rightharpoonup \mathsf{Trace}$$

The Trace is a mapping from time to transfers between parties:

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

The trace of a contract depends on an *external environment* ExtEnv containing information about *observable* values.
The *template environment* TEnv maps template variables to values.

Contract DSL Semantics

The semantics of a contact is given by a Trace:

$$\mathcal{C}\left[\!\left[c \right]\!\right] : \mathsf{ExtEnv} \times \mathsf{TEnv} \rightharpoonup \mathsf{Trace}$$

The Trace is a mapping from time to transfers between parties:

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

The trace of a contract depends on an *external environment* ExtEnv containing information about *observable* values.
The *template environment* TEnv maps template variables to values. The semantics does not depend on any stochastic aspects

Motivation for a Payoff Language

- Pricing (simulation using Monte-Carlo techniques) requires "snapshot" value of the contract
- Discounting should be taken into account
- Compiling to a target language should be (relatively) easy
- Several target languages (depends on the pricing engine)

Payoff Language: syntax and semantics

Syntax

Expression language with conditionals

$$il \ ::= \ unop(il) \mid binop(il, il) \mid \texttt{if}(il, il, il) \mid \texttt{loopif}(il, il, il, t)$$

Payoff Language: syntax and semantics

Syntax

Expression language with conditionals

$$il \ ::= \ unop(il) \mid binop(il, il) \mid \mathtt{if}(il, il, il) \mid \mathtt{loopif}(il, il, il, t)$$

some additional domain-specific constructs

$$\mathtt{model}(l, t) \mid disc(t) \mid \mathtt{payoff}(t, p, p) \mid \mathtt{now}$$

Payoff Language: syntax and semantics

Syntax

Expression language with conditionals

$$il \ ::= \ unop(il) \mid binop(il, il) \mid \texttt{if}(il, il, il) \mid \texttt{loopif}(il, il, il, t)$$

some additional domain-specific constructs

$$\texttt{model}(l, t) \mid disc(t) \mid \texttt{payoff}(t, p, p) \mid \texttt{now}$$

and template subexpressions

$$t ::= \ n \mid i \mid v \mid \texttt{tplus}(t, t)$$

Payoff Language: syntax and semantics

Syntax

Expression language with conditionals

$$il \;::=\; unop(il) \mid binop(il, il) \mid \texttt{if}(il, il, il) \mid \texttt{loopif}(il, il, il, t)$$

some additional domain-specific constructs

$$\texttt{model}(l, t) \mid disc(t) \mid \texttt{payoff}(t, p, p) \mid \texttt{now}$$

and template subexpressions

$$t ::= \; n \mid i \mid v \mid \texttt{tplus}(t, t)$$

Semantics

$$\mathcal{IL}\,[\![il]\!] : \text{ExtEnv} \times \text{TEnv} \times \text{Disc} \times \text{Party} \times \text{Party} \rightharpoonup \mathbb{R} + \mathbb{B}$$

where $\text{Disc} = \mathbb{N} \to \mathbb{R}$ is a discounting function.

Compiling Contracts to Payoffs

We compile both "levels" - contracts *c* and expressions *e* - into single payoff expression language

The compilation functions:

$$\tau_{\mathrm{e}} \llbracket - \rrbracket : \mathsf{Expr} \times \mathsf{TExprZ} \rightharpoonup \mathsf{ILExpr}$$

$$\tau_{\mathrm{c}} \llbracket - \rrbracket : \mathsf{Contr} \times \mathsf{TExprZ} \rightharpoonup \mathsf{ILExpr}$$

Compiling Contracts to Payoffs

$$\begin{aligned}
\tau_{\mathrm{e}} \left[\!\left[\mathtt{obs}(l, i) \right]\!\right]_{t_0} &= \mathtt{model}(l, \mathtt{tplus}(t_0, i)) \\
\tau_{\mathrm{c}} \left[\!\left[\mathtt{transfer}(p_1, p_2) \right]\!\right]_{t_0} &= \mathtt{mult}(\mathit{disc}(t_0), \mathtt{payoff}(t_0, p_1, p_2)) \\
\tau_{\mathrm{c}} \left[\!\left[\mathtt{scale}(e, c) \right]\!\right]_{t_0} &= \mathtt{mult}(\tau_{\mathrm{e}} \left[\!\left[e \right]\!\right]_{t_0}, \tau_{\mathrm{c}} \left[\!\left[c \right]\!\right]_{t_0}) \\
\tau_{\mathrm{c}} \left[\!\left[\mathtt{zero} \right]\!\right]_{t_0} &= 0 \\
\tau_{\mathrm{c}} \left[\!\left[\mathtt{translate}(t, c) \right]\!\right]_{t_0} &= \tau_{\mathrm{c}} \left[\!\left[c \right]\!\right]_{\mathtt{tplus}(t_0, t)} \\
\tau_{\mathrm{c}} \left[\!\left[\mathtt{both}(c_0, c_1) \right]\!\right]_{t_0} &= \mathtt{add}(\tau_{\mathrm{c}} \left[\!\left[c_0 \right]\!\right]_{t_0}, \tau_{\mathrm{c}} \left[\!\left[c_1 \right]\!\right]_{t_0}) \\
\tau_{\mathrm{c}} \left[\!\left[\mathtt{checkWithin}(e, t, c_1, c_2) \right]\!\right]_{t_0} &= \mathtt{loopif}(\tau_{\mathrm{e}} \left[\!\left[e \right]\!\right]_{t_0}, \tau_{\mathrm{c}} \left[\!\left[c_0 \right]\!\right]_{t_0}, \tau_{\mathrm{c}} \left[\!\left[c_1 \right]\!\right]_{t_0}, t)
\end{aligned}$$

Compiling Contracts to Payoffs: An Example

Original contract

```
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)))
```

Compiling Contracts to Payoffs: An Example

Original contract

```
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)))
```

Compiles to (using infix notation for binary operations)

```
... + ...
```

Compiling Contracts to Payoffs: An Example

Original contract

```
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)))
```

Compiles to (using infix notation for binary operations)

```
100.0 * disc(t0) + ...
```

Compiling Contracts to Payoffs: An Example

Original contract

```
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)))
```

Compiles to (using infix notation for binary operations)

```
100.0 * disc(t0) + if (...,
                        ...,
                        ...)
```

Compiling Contracts to Payoffs: An Example

Original contract

```
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)))
```

Compiles to (using infix notation for binary operations)

```
100.0 * disc(t0) + if (model(AAPL,t0+t1) > 100.0,
                       ...,
                       ...)
```

Compiling Contracts to Payoffs: An Example

Original contract

```
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)))
```

Compiles to (using infix notation for binary operations)

```
100.0 * disc(t0) + if (model(AAPL,t0+t1) > 100.0,
                       (model(AAPL,t0+t1) - 100.0) * disc(t0+t1),
                        0.0)
```

Why "certified"?

- The semantics and the symbolic contract transformations are verified in the Coq proof assistant[1]
- The correctness of the compilation into payoff expressions is crucial for the result of pricing
- Having a proof of correctness, we can use *code extraction* techniques to obtain a correct implementation

[1] formalization by Bahr et al.

Compilation soundness

Assume parties p_1 and p_2, discount function $d : \mathbb{N} \to \mathbb{R}$ and environments ρ : ExtEnv, δ : TEnv

Theorem (soundness for contract expressions)

If $\tau_{\mathrm{e}} [\![e]\!] = il$ and $\mathcal{E} [\![e]\!]_{\rho,\delta} = v_1$ and $\mathcal{IL} [\![il]\!]_{\rho,\delta,d,p_1,p_2} = v_2$

then $v_1 = v_2$.

Theorem (soundness for contracts)

If $\tau_{\mathrm{c}} [\![c]\!] = il$ and $\mathcal{C} [\![c]\!]_{\rho,\delta} = trace$,

where $trace : \mathbb{N} \to \text{Party} \times \text{Party} \to \mathbb{R}$,

and $\mathcal{IL} [\![il]\!]_{\rho,\delta,d,p_1,p_2} = v$

then $\sum_{t=0}^{HOR(c,\delta)} d(t) \times trace(t)(p_1, p_2) = v$.

Performance challenges

- Contracts evolution: at each time step a contract becomes a new "smaller" contract (by reduction)
- Reductions thus requires us to recompile a contract to a payoff expression and further to a target language
- Pricing engine could use inlining for optimization purposes $\rightarrow$ contract recompilation could require to recompile part (or whole) pricing engine

Performance challenges: possible solutions

- Write an interpreter for the payoff language
  ✓ general
  ✗ hard to implement efficiently on GPUs
  ✗ still requires compilation contracts to payoffs
- Parameterize payoff expression with "current time" t_{now} and "cut" payoffs before t_{now} at runtime
  ✓ requires less modification to the pricer
  ✓ less work to prove correctness and implement in Coq
  ✗ overhead due to additional checks

Performance challenges: possible solutions

- Write an interpreter for the payoff language
  ✓ general
  ✗ hard to implement efficiently on GPUs
  ✗ still requires compilation contracts to payoffs
- Parameterize payoff expression with "current time" t_{now} and "cut" payoffs before t_{now} at runtime
  ✓ requires less modification to the pricer
  ✓ less work to prove correctness and implement in Coq
  ✗ **overhead due to additional checks (but not that bad)**

Experiment

The estimated overhead was around 2.5 percent for hand-written implementation of guard condition for simple contracts

Cutting payoffs

The cutPayoff() function adds guard condition to each `payoff` construct (showing only most important case):

$$\begin{aligned}
\text{cutPayoff} &: \text{ILExpr} \to \text{ILExpr} \\
&\cdots \\
\text{cutPayoff}(\texttt{payoff}(t, p_1, p_2)) &= \texttt{if}(t < \mathit{now}, 0, \texttt{payoff}(t, p_1, p_2))
\end{aligned}$$

cutPayoff properties

If we evaluate a compiled payoff expression after application of cutPayoff() with $t_{now} = 0$ we should get the same result as evaluating the expression before applying cutPayoff()

Theorem

Assume parties p_1 and p_2, discount function $d : \mathbb{N} \to \mathbb{R}$ and environments ρ : ExtEnv, δ : TEnv.

For any P : ILExpr, if $t_{now} = 0$ then

$$\mathcal{IL}\left[\!\left[P\right]\!\right]_{\rho,\delta,d,t_{now},p_1,p_2} = \mathcal{IL}\left[\!\left[\text{cutPayoff}(P)\right]\!\right]_{\rho,\delta,d,t_{now},p_1,p_2}$$

cutPayoff properties

The cutPayoff() function should be sound with respect to contract reduction semantics.

Theorem (soudness wrt. contract reduction)

Assume parties p_1 and p_2, discount function $d : \mathbb{N} \to \mathbb{R}$ and environments $\rho' \ : \ \text{ExtEnv}$, $\delta \ : \ \text{TEnv}$ and a *partial environment* $\rho \in \text{ExtEnvP}$ such that $\rho \subseteq \rho'$.

For any well-typed contract c, if $c \implies_\rho c'$, $\mathcal{C}\left[\!\left[c' \right]\!\right]_{\rho'/1,\delta} = \mathit{trace}$, $\tau_{\mathrm{c}}\left[\!\left[c \right]\!\right] = P$ and $\mathcal{IL}\left[\!\left[\text{cutPayoff}(P) \right]\!\right]_{\rho',\delta,1,d,p_1,p_2} = v$ then $\sum_{t=0}^{HOR(c')} d(t+1) \times \mathit{trace}(t) = v$

Where $\rho'/1$ means environment ρ “advanced” one step.

Conclusion

- Contract DSL extended with template expressions $\rightarrow$ improved code reuse

Conclusion

- Contract DSL extended with template expressions $\rightarrow$ improved code reuse
- designed Payoff Intermediate Language

Conclusion

- Contract DSL extended with template expressions $\rightarrow$ improved code reuse
- designed Payoff Intermediate Language
- formalization of the Payoff Language semantics in Coq (including proofs on compilation soundness)

Conclusion

- Contract DSL extended with template expressions $\rightarrow$ improved code reuse
- designed Payoff Intermediate Language
- formalization of the Payoff Language semantics in Coq (including proofs on compilation soundness)
- payoff expressions parameterized by time $\rightarrow$ captures contract reduction in runtime $\rightarrow$ allows to avoid recompilation

Conclusion

- Contract DSL extended with template expressions $\rightarrow$ improved code reuse
- designed Payoff Intermediate Language
- formalization of the Payoff Language semantics in Coq (including proofs on compilation soundness)
- payoff expressions parameterized by time $\rightarrow$ captures contract reduction in runtime $\rightarrow$ allows to avoid recompilation
- use of code extraction Coq code extraction mechanism to obtain correct compilation function

Future Work

- proof of soundness wrt. multi-step contract reduction
- external environments as arrays (implement reindexing and prove properties)
- make proofs nicer (clean and modular)
- integration of extracted code to the HIPERFIT Portfolio Management Prototype
- investigate connection to Smart Contract and to blockchain-related technology

Thank you!

Thank you! Questions?