Certified Compilers and Program Analyses

Patrick Bahr

University of Copenhagen,
Department of Computer Science
paba@diku.dk

1st December, 2014

Overview

1. Deriving Certified Compilers from Specification
2. Certified Management and Analysis of Financial Contracts

Part I:

Deriving Certified Compilers from Specification

joint work with Graham Hutton

Introduction

The problem: Implementing a correct compiler.

Introduction

The problem: Implementing a correct compiler.

$$\text{Language} \xrightarrow{\text{compiler}} \text{Code}$$

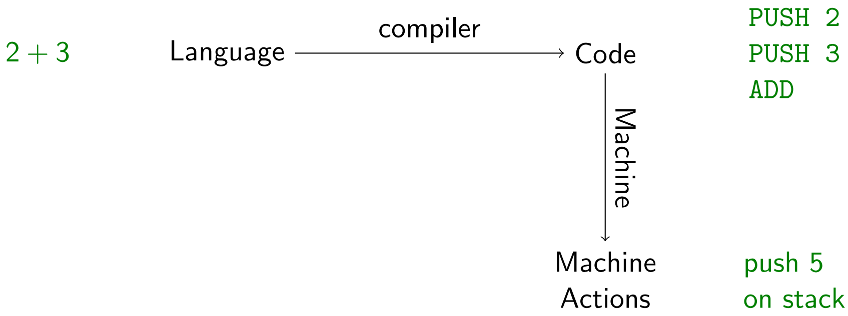
Introduction

The problem: Implementing a correct compiler.

$$\begin{tikzcd}
\text{Language} \arrow[r, "\text{compiler}"] & \text{Code} \arrow[d, "\text{Machine}"] \\
& \begin{array}{c}\text{Machine}\\ \text{Actions}\end{array}
\end{tikzcd}$$

Introduction

The problem: Implementing a correct compiler.



$2+3$

Language $\xrightarrow{\text{compiler}}$ Code

Code $\xrightarrow{\text{Machine}}$ Machine Actions

```
PUSH 2
PUSH 3
ADD
```

push 5
on stack

Introduction

The problem: Implementing a correct compiler.

$$\begin{tikzcd}
2+3 & \text{Language} \arrow[r, "\text{compiler}"] \arrow[d, "{[\![\cdot]\!]}"'] & \text{Code} \arrow[d, "\text{Machine}"] & \begin{array}{l}\texttt{PUSH 2}\\ \texttt{PUSH 3}\\ \texttt{ADD}\end{array} \\
 & \begin{array}{c}\text{Semantic}\\ \text{Domain}\end{array} & \begin{array}{c}\text{Machine}\\ \text{Actions}\end{array} & \begin{array}{l}\text{push 5}\\ \text{on stack}\end{array}
\end{tikzcd}$$

Introduction

The problem: Implementing a correct compiler.

$$\begin{tikzcd}
2+3 & \text{Language} \arrow[rr, "\text{compiler}"] \arrow[d, "{[\![\cdot]\!]}"'] & & \text{Code} \arrow[d, "\text{Machine}"] & \begin{array}{l}\texttt{PUSH 2}\\ \texttt{PUSH 3}\\ \texttt{ADD}\end{array} \\
5 & \begin{array}{c}\text{Semantic}\\ \text{Domain}\end{array} & & \begin{array}{c}\text{Machine}\\ \text{Actions}\end{array} & \begin{array}{l}\text{push 5}\\ \text{on stack}\end{array}
\end{tikzcd}$$

Introduction

The problem: Implementing a correct compiler.

$$\begin{tikzcd}
\text{Language} \arrow[r, "\text{compiler}"] \arrow[d, "[\![\cdot]\!]"'] & \text{Code} \arrow[d, "\text{Machine}"] \\
\text{Semantic Domain} \arrow[r, "\text{rep}"] & \text{Machine Actions}
\end{tikzcd}$$

$2 + 3$

```
PUSH 2
PUSH 3
ADD
```

5

push 5
on stack

Introduction

The problem: Implementing a correct compiler.

$$\begin{tikzcd}
2+3 & \text{Language} \arrow[r, "\text{compiler}"] \arrow[d, "[\![\cdot]\!]"'] & \text{Code} \arrow[d, "\text{Machine}"] & \begin{array}{l}\texttt{PUSH 2}\\ \texttt{PUSH 3}\\ \texttt{ADD}\end{array} \\
5 & \begin{array}{c}\text{Semantic}\\ \text{Domain}\end{array} \arrow[r, "\text{rep}"] & \begin{array}{c}\text{Machine}\\ \text{Actions}\end{array} & \begin{array}{l}\text{push 5}\\ \text{on stack}\end{array}
\end{tikzcd}$$

Introduction

The problem: Implementing a correct compiler.

$$\begin{tikzcd}
2+3 & \text{Language} \arrow[rr, "\text{compiler}"] \arrow[d, "\llbracket\cdot\rrbracket"'] & & \text{Code} \arrow[d, "\text{Machine}"] & \begin{array}{l}\texttt{PUSH 2}\\ \texttt{PUSH 3}\\ \texttt{ADD}\end{array} \\
5 & \begin{array}{c}\text{Semantic}\\ \text{Domain}\end{array} \arrow[rr, "\text{rep}"] & & \begin{array}{c}\text{Machine}\\ \text{Actions}\end{array} & \begin{array}{l}\text{push 5}\\ \text{on stack}\end{array}
\end{tikzcd}$$

Goal:
- Systematically derive compiler from ⟦·⟧ & rep

Introduction

The problem: Implementing a correct compiler.

$$\begin{tikzcd}
2+3 & \text{Language} \arrow[r, "\text{compiler}"] \arrow[d, "\llbracket \cdot \rrbracket"'] & \text{Code} \arrow[d, "\text{Machine}"] & \begin{array}{l}\texttt{PUSH 2}\\ \texttt{PUSH 3}\\ \texttt{ADD}\end{array} \\
5 & \begin{array}{c}\text{Semantic}\\ \text{Domain}\end{array} \arrow[r, "\text{rep}"] & \begin{array}{c}\text{Machine}\\ \text{Actions}\end{array} & \begin{array}{l}\text{push 5}\\ \text{on stack}\end{array}
\end{tikzcd}$$

Goal:

- Systematically derive compiler from $\llbracket \cdot \rrbracket$ & rep
- Derivation is rigorous & machine-checked

Toy Example: Simple Arithmetic Language

Step 1: Semantics of the language

Syntax

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

Toy Example: Simple Arithmetic Language

Step 1: Semantics of the language

Syntax

e.g. $2 + 3 \quad \rightsquigarrow \quad \mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3)$

$\mathbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$

Toy Example: Simple Arithmetic Language

Step 1: Semantics of the language

Syntax

e.g. $2 + 3 \quad \rightsquigarrow \quad \mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3)$

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

Semantics

$$\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Int} \\
&\mathit{eval}\ (\mathit{Val}\ n) \quad\ \ = n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}$$

Step 2: Compiler Correctness Property

The compiler

data *Instr* = ...
type *Code* = [*Instr*]   -- list of instructions

$comp :: Expr \to Code$

Step 2: Compiler Correctness Property

The compiler

data $Instr = \ldots$
type $Code = [Instr]$ -- list of instructions
$comp :: Expr \rightarrow Code$

The machine

type $Stack = [Int]$ -- list of integers
$exec :: Code \rightarrow Stack \rightarrow Stack$

Step 2: Compiler Correctness Property

The compiler

data *Instr* = ...
type *Code* = [*Instr*]   -- list of instructions
comp :: *Expr* → *Code*

The machine

type *Stack* = [*Int*]   -- list of integers
exec :: *Code* → *Stack* → *Stack*

Compiler correctness property

For all $e :: \mathit{Expr}$, $s :: \mathit{Stack}$

$$\mathit{exec}\ (\mathit{comp}\ e \qquad)\ s = \qquad \mathit{eval}\ e : s$$

Step 2: Compiler Correctness Property

The compiler

$\textbf{data}\ \mathit{Instr} = \ldots$
$\textbf{type}\ \mathit{Code} = [\mathit{Instr}]$ -- list of instructions
$\mathit{comp} :: \mathit{Expr} \rightarrow \mathit{Code}$

The machine

$\textbf{type}\ \mathit{Stack} = [\mathit{Int}]$ -- list of integers
$\mathit{exec} :: \mathit{Code} \rightarrow \mathit{Stack} \rightarrow \mathit{Stack}$

Compiler correctness property

For all $e :: \mathit{Expr}$, $s :: \mathit{Stack}$, $c :: \mathit{Code}$

$$\mathit{exec}\ (\mathit{comp}\ e \mathbin{+\!\!+} c)\ s = \mathit{exec}\ c\ (\mathit{eval}\ e : s)$$

Step 2: Compiler Correctness Property

The compiler

$$\textbf{data}\ \mathit{Instr} = \ldots$$
$$\textbf{type}\ \mathit{Code} = [\mathit{Instr}] \quad \text{-- list of instructions}$$
$$\mathit{comp} :: \mathit{Expr} \rightarrow \mathit{Code}$$

The machine

$$\textbf{type}\ \mathit{Stack} = [\mathit{Int}] \quad \text{-- list of integers}$$
$$\mathit{exec} :: \mathit{Code} \rightarrow \mathit{Stack} \rightarrow \mathit{Stack}$$
$$\mathit{exec}\ [\,]\ s = s$$

Compiler correctness property

For all $e :: \mathit{Expr}$, $s :: \mathit{Stack}$, $c :: \mathit{Code}$

$$\mathit{exec}\ (\mathit{comp}\ e +\!\!+\ c)\ s = \mathit{exec}\ c\ (\mathit{eval}\ e : s)$$

Step 3: Calculate!

Compiler correctness property

$$exec\ (comp\ e \mathbin{+\!\!+} c)\ s = \ exec\ c\ (eval\ e : s)$$

Step 3: Calculate!

Compiler correctness property

$$exec\ (comp\ e \mathbin{+\!\!+} c)\ s = \ exec\ c\ (eval\ e : s)$$

Strategy

- structural induction on e
- transform $exec\ c\ (eval\ e : s)$ into $exec\ (c' \mathbin{+\!\!+} c)\ s$
- conclude that $comp\ e = c'$

Step 3: Calculate!

Compiler correctness property

$$exec\ (comp\ e \mathbin{+\!\!+} c)\ s = \ exec\ c\ (eval\ e : s)$$

$$exec\ c\ (eval\ e : s)$$

Strategy

- structural induction on e
- transform $exec\ c\ (eval\ e : s)$ into $exec\ (c' \mathbin{+\!\!+} c)\ s$
- conclude that $comp\ e = c'$

Step 3: Calculate!

Compiler correctness property

$$exec\ (comp\ e +\!\!+ c)\ s = \ exec\ c\ (eval\ e : s)$$

$$\leftrightsquigarrow exec\ c\ (eval\ e : s)$$

Strategy

- structural induction on e
- transform $exec\ c\ (eval\ e : s)$ into $exec\ (c' +\!\!+ c)\ s$
- conclude that $comp\ e = c'$

Step 3: Calculate!

Compiler correctness property

$$exec\ (comp\ e +\!\!+ c)\ s = \ exec\ c\ (eval\ e : s)$$

$$exec\ (\quad c' \qquad +\!\!+ c)\ s \leftsquigarrow exec\ c\ (eval\ e : s)$$

Strategy

- structural induction on e
- transform $exec\ c\ (eval\ e : s)$ into $exec\ (c' +\!\!+ c)\ s$
- conclude that $comp\ e = c'$

Step 3: Calculate!

Compiler correctness property

$$\begin{array}{rcccl} exec\ (& comp\ e & \mathbin{+\!\!+} c)\ s & = & exec\ c\ (eval\ e : s) \\ & \Vert & & & \\ exec\ (& c' & \mathbin{+\!\!+} c)\ s & \leftsquigarrow & exec\ c\ (eval\ e : s) \end{array}$$

Strategy

- structural induction on e
- transform $exec\ c\ (eval\ e : s)$ into $exec\ (c' \mathbin{+\!\!+} c)\ s$
- conclude that $comp\ e = c'$

Case $e = Val\ n$

Compiler correctness property

$$exec\ (comp\ e \mathbin{+\!\!+} c)\ s = exec\ c\ (eval\ e : s)$$

Proof

$$exec\ c\ (eval\ (Val\ n) : s)$$

$$exec\ (c' \mathbin{+\!\!+} c)\ s$$

Case $e = Val\ n$

Compiler correctness property

$$exec\ (comp\ e + c)\ s = exec\ c\ (eval\ e : s)$$

Proof

$$\begin{aligned} & exec\ c\ (eval\ (Val\ n) : s) \\ = & \quad \{\ \text{definition of } eval\ \} \\ & exec\ c\ (n : s) \end{aligned}$$

$$exec\ (c' + c)\ s$$

Case $e = Val\ n$

Compiler correctness property

$$exec\ (comp\ e +\!\!+\ c)\ s = exec\ c\ (eval\ e : s)$$

Proof

$$
\begin{aligned}
& exec\ c\ (eval\ (Val\ n) : s) \\
= \quad & \{\ \text{definition of } eval\ \} \\
& exec\ c\ (n : s) \\
= \quad & \{\ \text{define: } exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s)\ \} \\
& exec\ (PUSH\ n : c)\ s
\end{aligned}
$$

$$exec\ (c' +\!\!+\ c)\ s$$

Case $e = \mathit{Val}\ n$

Compiler correctness property

$$\mathit{exec}\ (\mathit{comp}\ e \mathbin{+\!\!+} c)\ s = \mathit{exec}\ c\ (\mathit{eval}\ e : s)$$

Proof

$$\begin{aligned}
& \mathit{exec}\ c\ (\mathit{eval}\ (\mathit{Val}\ n) : s) \\
= \quad & \{\ \text{definition of } \mathit{eval}\ \} \\
& \mathit{exec}\ c\ (n : s) \\
= \quad & \{\ \text{define: } \mathit{exec}\ (\mathit{PUSH}\ n : c)\ s = \mathit{exec}\ c\ (n : s)\ \} \\
& \mathit{exec}\ (\mathit{PUSH}\ n : c)\ s
\end{aligned}$$

data $\mathit{Instr} = \mathit{PUSH}\ \mathit{Int} \mid \ldots$

$$\mathit{exec}\ (c' \mathbin{+\!\!+} c)\ s$$

Case $e = Val\ n$

Compiler correctness property

$$exec\ (comp\ e \mathbin{+\!\!\!+} c)\ s = exec\ c\ (eval\ e : s)$$

Proof

$$\begin{array}{ll}
 & exec\ c\ (eval\ (Val\ n) : s) \\
= & \quad \{\ \text{definition of } eval\ \} \\
 & exec\ c\ (n : s) \\
= & \quad \{\ \text{define: } exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s)\ \} \\
 & exec\ (PUSH\ n : c)\ s \\
= & \quad \{\ \text{definition of } \mathbin{+\!\!\!+}\ \} \\
 & exec\ (c' \mathbin{+\!\!\!+} c)\ s
\end{array}$$

Case $e = Val\ n$

Compiler correctness property

$$exec\ (comp\ e +\!\!+\ c)\ s = exec\ c\ (eval\ e : s)$$

Proof

$$
\begin{aligned}
& exec\ c\ (eval\ (Val\ n) : s) \\
= \quad & \{\ \text{definition of } eval\ \} \\
& exec\ c\ (n : s) \\
= \quad & \{\ \text{define: } exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s)\ \} \\
& exec\ (PUSH\ n : c)\ s \\
= \quad & \{\ \text{definition of } +\!\!+\ \} \\
& exec\ ([PUSH\ n] +\!\!+\ c)\ s
\end{aligned}
$$

Case $e = Val\ n$

Compiler correctness property

$$exec\ (comp\ e +\!\!+\ c)\ s = exec\ c\ (eval\ e : s)$$

Proof

$$\begin{aligned}
& exec\ c\ (eval\ (Val\ n) : s) \\
= \quad & \{\ \text{definition of } eval\ \} \\
& exec\ c\ (n : s) \\
= \quad & \{\ \text{define: } exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s)\ \} \\
& exec\ (PUSH\ n : c)\ s \\
= \quad & \{\ \text{definition of } +\!\!+\ \} \\
& exec\ ([PUSH\ n] +\!\!+\ c)\ s
\end{aligned}$$

Conclude: $comp\ (Val\ n) = [PUSH\ n]$

Case $e = Add\ x\ y$

Compiler correctness property

$$exec\ (comp\ e \mathbin{+\!\!+} c)\ s = exec\ c\ (eval\ e : s)$$

Proof

$$exec\ c\ (eval\ (Add\ x\ y) : s)$$

$$exec\ (c' \mathbin{+\!\!+} c)\ s$$

Case $e = Add\ x\ y$

Induction hypothesis

$$exec\ (comp\ x \mathbin{+\!\!+} c'')\ s' = exec\ c''\ (eval\ x : s')$$
$$exec\ (comp\ y \mathbin{+\!\!+} c'')\ s' = exec\ c''\ (eval\ y : s')$$

Proof

$$exec\ c\ (eval\ (Add\ x\ y) : s)$$

$$exec\ (c' \mathbin{+\!\!+} c)\ s$$

Case $e = Add\ x\ y$

Induction hypothesis

$$exec\ (comp\ x +\!\!+\ c'')\ s' = exec\ c''\ (eval\ x : s')$$
$$exec\ (comp\ y +\!\!+\ c'')\ s' = exec\ c''\ (eval\ y : s')$$

Proof

$$\begin{array}{ll} & exec\ c\ (eval\ (Add\ x\ y) : s) \\ = & \quad \{\ \text{definition of } eval\ \} \\ & exec\ c\ (eval\ x + eval\ y : s) \end{array}$$

$$exec\ (c' +\!\!+\ c)\ s$$

Case $e = Add\ x\ y$

Induction hypothesis

$$exec\ (comp\ x \mathbin{+\!\!+} c'')\ s' = exec\ c''\ (eval\ x : s')$$

$$exec\ (comp\ y \mathbin{+\!\!+} c'')\ s' = exec\ c''\ (eval\ y : s')$$

Proof

$$
\begin{aligned}
& exec\ c\ (eval\ (Add\ x\ y) : s) \\
= \quad & \{\ \text{definition of } eval\ \} \\
& exec\ c\ (eval\ x + eval\ y : s) \\
= \quad & \{\ \text{define: } exec\ (ADD : c)\ (m : n : s) = exec\ c\ ((n + m) : s)\ \} \\
& exec\ ([ADD] \mathbin{+\!\!+} c)\ (eval\ y : eval\ x : s)
\end{aligned}
$$

$$exec\ (c' \mathbin{+\!\!+} c)\ s$$

Case $e = Add\ x\ y$

Induction hypothesis

$$exec\ (comp\ x \mathbin{+\!\!+} c'')\ s' = exec\ c''\ (eval\ x : s')$$
$$exec\ (comp\ y \mathbin{+\!\!+} c'')\ s' = exec\ c''\ (eval\ y : s')$$

Proof

$$\begin{aligned}
& exec\ c\ (eval\ (Add\ x\ y) : s) \\
= \quad & \{\ \text{definition of } eval\ \} \\
& exec\ c\ (eval\ x + eval\ y : s) \\
= \quad & \{\ \text{define: } exec\ (ADD : c)\ (m : n : s) = exec\ c\ ((n + m) : s)\ \} \\
& exec\ ([ADD] \mathbin{+\!\!+} c)\ (eval\ y : eval\ x : s) \\
= \quad & \{\ \text{induction hypothesis for } y\ \} \\
& exec\ (comp\ y \mathbin{+\!\!+} [ADD] \mathbin{+\!\!+} c))\ (eval\ x : s)
\end{aligned}$$

$$exec\ (c' \mathbin{+\!\!+} c)\ s$$

Case $e = Add\ x\ y$

Induction hypothesis

$$exec\ (comp\ x \mathbin{+\!\!+} c'')\ s' = exec\ c''\ (eval\ x : s')$$
$$exec\ (comp\ y \mathbin{+\!\!+} c'')\ s' = exec\ c''\ (eval\ y : s')$$

Proof

$$
\begin{aligned}
& exec\ c\ (eval\ (Add\ x\ y) : s) \\
= \quad & \{\ \text{definition of } eval\ \} \\
& exec\ c\ (eval\ x + eval\ y : s) \\
= \quad & \{\ \text{define: } exec\ (ADD : c)\ (m : n : s) = exec\ c\ ((n + m) : s)\ \} \\
& exec\ ([ADD] \mathbin{+\!\!+} c)\ (eval\ y : eval\ x : s) \\
= \quad & \{\ \text{induction hypothesis for } y\ \} \\
& exec\ (comp\ y \mathbin{+\!\!+} [ADD] \mathbin{+\!\!+} c))\ (eval\ x : s) \\
= \quad & \{\ \text{induction hypothesis for } x\ \} \\
& exec\ (c' \mathbin{+\!\!+} c)\ s
\end{aligned}
$$

Case $e = Add\ x\ y$

Induction hypothesis

$$exec\ (comp\ x +\!\!+\ c'')\ s' = exec\ c''\ (eval\ x : s')$$

$$exec\ (comp\ y +\!\!+\ c'')\ s' = exec\ c''\ (eval\ y : s')$$

Proof

$$\begin{aligned}
& exec\ c\ (eval\ (Add\ x\ y) : s) \\
= \quad & \{\ \text{definition of } eval\ \} \\
& exec\ c\ (eval\ x + eval\ y : s) \\
= \quad & \{\ \text{define: } exec\ (ADD : c)\ (m : n : s) = exec\ c\ ((n + m) : s)\ \} \\
& exec\ ([ADD] +\!\!+\ c)\ (eval\ y : eval\ x : s) \\
= \quad & \{\ \text{induction hypothesis for } y\ \} \\
& exec\ (comp\ y +\!\!+\ [ADD] +\!\!+\ c))\ (eval\ x : s) \\
= \quad & \{\ \text{induction hypothesis for } x\ \} \\
& exec\ (comp\ x +\!\!+\ comp\ y +\!\!+\ [ADD] +\!\!+\ c)\ s
\end{aligned}$$

Case $e = Add\ x\ y$

Induction hypothesis

$$exec\ (comp\ x +\!\!+\ c'')\ s' = exec\ c''\ (eval\ x : s')$$

$$exec\ (comp\ y +\!\!+\ c'')\ s' = exec\ c''\ (eval\ y : s')$$

Proof

$$\begin{aligned}
& exec\ c\ (eval\ (Add\ x\ y) : s) \\
= & \quad \{\ \text{definition of } eval\ \} \\
& exec\ c\ (eval\ x + eval\ y : s) \\
= & \quad \{\ \text{define: } exec\ (ADD : c)\ (m : n : s) = exec\ c\ ((n + m) : s)\ \} \\
& exec\ ([ADD] +\!\!+\ c)\ (eval\ y : eval\ x : s) \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
& exec\ (comp\ y +\!\!+\ [ADD] +\!\!+\ c))\ (eval\ x : s) \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
& exec\ (comp\ x +\!\!+\ comp\ y +\!\!+\ [ADD] +\!\!+\ c)\ s
\end{aligned}$$

Conclude: $comp\ (Add\ x\ y) = comp\ x +\!\!+\ comp\ y +\!\!+\ [ADD]$

Derived Compiler Implementation

The compiler

```
data Instr = PUSH Int | ADD
type Code = [Instr]   -- list of instructions

comp :: Expr → Code
comp (Val n)     = [PUSH n]
comp (Add x y) = comp x ++ comp y ++ [ADD]
```

Derived Compiler Implementation

The compiler

```
data Instr = PUSH Int | ADD
type Code = [Instr]   -- list of instructions

comp :: Expr → Code
comp (Val n)     = [PUSH n]
comp (Add x y) = comp x ⧺ comp y ⧺ [ADD]
```

The machine

```
type Stack = [Int]   -- list of integers

exec :: Code → Stack → Stack
exec []                    s  = s
exec (PUSH n : c)      s  = exec c (n : s)
exec (ADD : c) (m : n : s) = exec c ((n + m) : s)
```

Summary

- simple calculations without the need for dependent types
- little prior knowledge needed
  (e.g. "Target machine has a stack.")
- scales to wide variety of language features

Summary

- simple calculations without the need for dependent types
- little prior knowledge needed
  (e.g. “Target machine has a stack.”)
- scales to wide variety of language features:
  - arithmetic expressions
  - exceptions (synchronous and asynchronous)
  - state (global and local)
  - lambda calculi (call-by-value, call-by-name, call-by-need)
  - loops (bounded and unbounded)
  - non-determinism

Summary

- simple calculations without the need for dependent types
- little prior knowledge needed
  (e.g. “Target machine has a stack.”)
- scales to wide variety of language features:
  - arithmetic expressions
  - exceptions (synchronous and asynchronous)
  - state (global and local)
  - lambda calculi (call-by-value, call-by-name, call-by-need)
  - loops (bounded and unbounded)
  - non-determinism
- Underlying techniques: continuation-passing style & defunctionalisation (Reynolds, 1972)

Summary

- simple calculations without the need for dependent types
- little prior knowledge needed
  (e.g. "Target machine has a stack.")
- scales to wide variety of language features:
  - arithmetic expressions
  - exceptions (synchronous and asynchronous)
  - state (global and local)
  - lambda calculi (call-by-value, call-by-name, call-by-need)
  - loops (bounded and unbounded)
  - non-determinism
- Underlying techniques: continuation-passing style & defunctionalisation (Reynolds, 1972)
- Formalised in Coq $\rightsquigarrow$ proof automation

Future Work

- Register-based machines
- Reason about concurrency
- Modular reasoning
  (e.g. abstraction from language features)
- "Real" target machines (e.g. JVM)
- Derive translation between calculi
  (e.g. λ-calculus $\rightarrow$ π-calculus)

Part II:

Certified Management and Analysis of Financial Contracts

joint work with Jost Berthold & Martin Elsman

Introduction

What are financial contracts?

- stipulate future transactions between different parties
- have time constraints
- may depend on stock prices, exchange rates etc.

Introduction

What are financial contracts?

- stipulate future transactions between different parties
- have time constraints
- may depend on stock prices, exchange rates etc.

Example (Foreign Exchange Option)

At any time within the next 90 days, party X may decide to buy USD 100 from party Y, for a fixed rate r of Danish Kroner.

Introduction

What are financial contracts?

- stipulate future transactions between different parties
- have time constraints
- may depend on stock prices, exchange rates etc.

Example (Foreign Exchange Option)

At any time within the next 90 days, party X may decide to buy USD 100 from party Y, for a fixed rate *r* of Danish Kroner.

Goals

- Express such contracts in a formal language
- Symbolic manipulation and analysis of such contracts.

Introduction

What are financial contracts?

- stipulate future transactions between different parties
- have time constraints
- may depend on stock prices, exchange rates etc.

Example (Foreign Exchange Option)

At any time within the next 90 days, party X may decide to buy USD 100 from party Y, for a fixed rate r of Danish Kroner.

Goals

- Express such contracts in a formal language
- Symbolic manipulation and analysis of such contracts.
- Formally verified!

Contract Language Goals in Detail

- **Compositionality**.
  Contracts are time-relative ⇒ facilitates compositionality
- **Multi-party**.
  Specify obligations and opportunities for multiple parties,
  (which opens up the possibility for specifying portfolios)
- **Contract management**.
  Contracts can be managed and symbolically evolved;
  a contract gradually reduces to the empty contract.
- **Contract utilities (symbolic)**.
  Contracts can be analysed in a variety of ways
- **Contract pricing (numerical, staged)**.
  Code for payoff can be generated from contracts
  (input to a stochastic pricing engine)

Example

Contract in natural language

- At any time within the next 90 days,
- party X may decide to
- buy USD 100 from party Y,
- for a fixed rate r of Danish Kroner.

Example

Contract in natural language

- At any time within the next 90 days,
- party X may decide to
- buy USD 100 from party Y,
- for a fixed rate r of Danish Kroner.

Translation into contract language

$$\begin{array}{l}\textbf{if } \mathit{obs}_{\mathbb{B}}(X \text{ exercises option}, 0) \textbf{ within } 90 \\ \textbf{then } 100 \times (\mathit{USD}(Y \to X) \mathbin{\&} r \times \mathit{DKK}(X \to Y)) \\ \textbf{else } \emptyset\end{array}$$

Contributions

- Denotational semantics based on cash-flows
- Reduction semantics (sound and complete)
- Correctness proofs for common contract analyses and transformations
- Formalised in the Coq theorem prover
- Certified implementation via code extraction

An Overview of the Contract Language

Core Calculus of Contracts

$$\frac{}{\vdash \emptyset : \mathsf{Contr}} \qquad \frac{p_1, p_2 \in \mathsf{Party} \quad a \in \mathsf{Asset}}{\vdash a(p_1 \to p_2) : \mathsf{Contr}}$$

$$\frac{\vdash e : \mathsf{Expr}_{\mathbb{R}} \quad \vdash c : \mathsf{Contr}}{\vdash e \times c : \mathsf{Contr}} \qquad \frac{d \in \mathbb{N} \quad \vdash c : \mathsf{Contr}}{\vdash d \uparrow c : \mathsf{Contr}}$$

$$\frac{\vdash c_i : \mathsf{Contr}}{\vdash c_1 \,\&\, c_2 : \mathsf{Contr}} \qquad \frac{\vdash e : \mathsf{Expr}_{\mathbb{B}} \quad d \in \mathbb{N} \quad \vdash c_i : \mathsf{Contr}}{\vdash \textbf{if } e \textbf{ within } d \textbf{ then } c_1 \textbf{ else } c_2 : \mathsf{Contr}}$$

An Overview of the Contract Language

Core Calculus of Contracts

$$\frac{}{\vdash \emptyset : \mathsf{Contr}} \qquad \frac{p_1, p_2 \in \mathsf{Party} \quad a \in \mathsf{Asset}}{\vdash a(p_1 \to p_2) : \mathsf{Contr}}$$

$$\frac{\vdash e : \mathsf{Expr}_{\mathbb{R}} \quad \vdash c : \mathsf{Contr}}{\vdash e \times c : \mathsf{Contr}} \qquad \frac{d \in \mathbb{N} \quad \vdash c : \mathsf{Contr}}{\vdash d \uparrow c : \mathsf{Contr}}$$

$$\frac{\vdash c_i : \mathsf{Contr}}{\vdash c_1 \,\&\, c_2 : \mathsf{Contr}} \qquad \frac{\vdash e : \mathsf{Expr}_{\mathbb{B}} \quad d \in \mathbb{N} \quad \vdash c_i : \mathsf{Contr}}{\vdash \textbf{if } e \textbf{ within } d \textbf{ then } c_1 \textbf{ else } c_2 : \mathsf{Contr}}$$

Expression Language

$\mathsf{Expr}_{\mathbb{R}}$, $\mathsf{Expr}_{\mathbb{B}}$: real-valued resp. Boolean-valued expressions.

An Overview of the Contract Language

Core Calculus of Contracts

$$\frac{}{\vdash \emptyset : \mathsf{Contr}} \qquad \frac{p_1, p_2 \in \mathsf{Party} \quad a \in \mathsf{Asset}}{\vdash a(p_1 \to p_2) : \mathsf{Contr}}$$

$$\frac{\vdash e : \mathsf{Expr}_{\mathbb{R}} \quad \vdash c : \mathsf{Contr}}{\vdash e \times c : \mathsf{Contr}} \qquad \frac{d \in \mathbb{N} \quad \vdash c : \mathsf{Contr}}{\vdash d \uparrow c : \mathsf{Contr}}$$

$$\frac{\vdash c_i : \mathsf{Contr}}{\vdash c_1 \,\&\, c_2 : \mathsf{Contr}} \qquad \frac{\vdash e : \mathsf{Expr}_{\mathbb{B}} \quad d \in \mathbb{N} \quad \vdash c_i : \mathsf{Contr}}{\vdash \textbf{if } e \textbf{ within } d \textbf{ then } c_1 \textbf{ else } c_2 : \mathsf{Contr}}$$

Expression Language

$\mathsf{Expr}_{\mathbb{R}}$, $\mathsf{Expr}_{\mathbb{B}}$: real-valued resp. Boolean-valued expressions.

$$obs_\alpha : \mathsf{Label}_\alpha \times \mathbb{Z} \to \mathsf{Expr}_\alpha$$

$$acc_\alpha : (\mathsf{Expr}_\alpha \to \mathsf{Expr}_\alpha) \times \mathbb{N} \times \mathsf{Expr}_\alpha \to \mathsf{Expr}_\alpha$$

Example: Asian Option

$$\begin{aligned} 90 \uparrow\ & \textbf{if } \mathit{obs}_{\mathbb{B}}(X \text{ exercises option}, 0) \textbf{ within } 0 \\ & \textbf{then } 100 \times (\mathit{USD}(Y \to X)\,\&(\mathit{rate} \times \mathit{DKK}(X \to Y))) \\ & \textbf{else } \emptyset \end{aligned}$$

where

$$\mathit{rate} = \frac{1}{30} \cdot \mathit{acc}(\lambda r.r + \mathit{obs}_{\mathbb{R}}(\mathit{FX\ USD/DKK}, 0), 30, 0)$$

Denotational Semantics

The semantics of a contract is given by the cash-flow it stipulates.

$$\mathcal{C}\left[\!\left[\cdot\right]\!\right]_{\cdot} : \mathsf{Contr} \qquad\qquad \to \mathsf{CashFlow}$$

Denotational Semantics

The semantics of a contract is given by the cash-flow it stipulates.

$$\mathcal{C}\left[\!\left[\cdot\right]\!\right]_{\cdot} : \mathsf{Contr} \qquad \to \mathsf{CashFlow}$$

$$\begin{aligned} \mathsf{CashFlow} &= \mathbb{N} \rightharpoonup \mathsf{Transactions} \\ \mathsf{Transactions} &= \mathsf{Party} \times \mathsf{Party} \times \mathsf{Asset} \to \mathbb{R} \end{aligned}$$

Denotational Semantics

The semantics of a contract is given by the cash-flow it stipulates.

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

$$\mathsf{Env} = \mathsf{Label} \times \mathbb{Z} \rightharpoonup \mathbb{B} \cup \mathbb{R}$$

$$\mathsf{CashFlow} = \mathbb{N} \rightharpoonup \mathsf{Transactions}$$

$$\mathsf{Transactions} = \mathsf{Party} \times \mathsf{Party} \times \mathsf{Asset} \to \mathbb{R}$$

Denotational Semantics

The semantics of a contract is given by the cash-flow it stipulates.

$$\mathcal{C}\left[\!\left[\cdot\right]\!\right]_{\cdot} : \mathsf{Contr} \times \mathsf{Env} \to \mathsf{CashFlow}$$
$$\mathsf{Env} = \mathsf{Label}_{\alpha} \times \mathbb{Z} \rightharpoonup \alpha$$

$$\mathsf{CashFlow} = \mathbb{N} \rightharpoonup \mathsf{Transactions}$$
$$\mathsf{Transactions} = \mathsf{Party} \times \mathsf{Party} \times \mathsf{Asset} \to \mathbb{R}$$

Contract Analyses

Examples

- contract dependencies
- contract causality
- contract horizon

Contract Analyses

Examples

- contract depender
- contract causality
- contract horizon

$obs_{\mathbb{R}}(FX\ \ USD/DKK, 1) \times DKK(X \rightarrow Y)$

Contract Analyses

Examples

- contract dependencies
- contract causality
- contract horizon

Semantics vs. Syntax

- these analyses have precise semantic definition
- they cannot be effectively computed
- we provide sound approximations, e.g. type system

Contract Causality

Refined Types

- $e : \mathrm{Expr}_{\alpha}^{t}$ value of e available at time t (or later)
- $c : \mathrm{Contr}^{t}$ no obligations strictly before t

Contract Causality

Refined Types

- $e : \mathrm{Expr}_\alpha^{t}$ value of e available at time t (or later)
- $c : \mathrm{Contr}^{t}$ no obligations strictly before t

Typing Rules

$$\frac{t_1, t_2 \in \mathbb{Z} \quad l \in \mathrm{Label}_\alpha \quad t_1 \leq t_2}{\Gamma \vdash obs_\alpha(l, t_1) : \mathrm{Expr}_\alpha^{t_2}} \qquad \frac{p_1, p_2 \in \mathrm{Party} \quad a \in \mathrm{Asset}}{\vdash a(p_1 \to p_2) : \mathrm{Contr}^0}$$

Contract Causality

Refined Types

- $e : \mathsf{Expr}_\alpha^{t}$ value of e available at time t (or later)
- $c : \mathsf{Contr}^{t}$ no obligations strictly before t

Typing Rules

$$\frac{t_1, t_2 \in \mathbb{Z} \quad l \in \mathsf{Label}_\alpha \quad t_1 \leq t_2}{\Gamma \vdash obs_\alpha(l, t_1) : \mathsf{Expr}_\alpha^{t_2}} \qquad \frac{p_1, p_2 \in \mathsf{Party} \quad a \in \mathsf{Asset}}{\vdash a(p_1 \to p_2) : \mathsf{Contr}^0}$$

$$\frac{\vdash e : \mathsf{Expr}_\mathbb{R}^{t} \quad \vdash c : \mathsf{Contr}^{t}}{\vdash e \times c : \mathsf{Contr}^{t}} \qquad \frac{d \in \mathbb{N} \quad \vdash c : \mathsf{Contr}^{t}}{\vdash d \uparrow c : \mathsf{Contr}^{t+d}}$$

Contract Causality

Refined Types

- $e : \mathrm{Expr}_\alpha^t$ value of e available at time t (or later)
- $c : \mathrm{Contr}^t$ no obligations strictly before t

Typing Rules

$$\frac{t_1, t_2 \in \mathbb{Z} \quad l \in \mathrm{Label}_\alpha \quad t_1 \leq t_2}{\Gamma \vdash obs_\alpha(l, t_1) : \mathrm{Expr}_\alpha^{t_2}} \qquad \frac{p_1, p_2 \in \mathrm{Party} \quad a \in \mathrm{Asset}}{\vdash a(p_1 \to p_2) : \mathrm{Contr}^0}$$

$$\frac{\vdash e : \mathrm{Expr}_\mathbb{R}^t \quad \vdash c : \mathrm{Contr}^t}{\vdash e \times c : \mathrm{Contr}^t} \qquad \frac{d \in \mathbb{N} \quad \vdash c : \mathrm{Contr}^t}{\vdash d \uparrow c : \mathrm{Contr}^{t+d}}$$

$$\vdots$$

Contract Transformations

Contract equivalences

When can we replace a sub-contract with another one, without changing the semantics of the contract?

Reduction semantics

What does the contract look like after n days have passed?

Contract Specialisation

What does the contract look like after we learned the actual value of some observables?

Contract Equivalences

$$\begin{array}{cc}
e_1 \times (e_2 \times c) \simeq (e_1 \cdot e_2) \times c & d \uparrow \emptyset \simeq \emptyset \\
d_1 \uparrow (d_2 \uparrow c) \simeq (d_1 + d_2) \uparrow c & r \times \emptyset \simeq \emptyset \\
d \uparrow (c_1 \mathbin{\&} c_2) \simeq (d \uparrow c_1) \mathbin{\&} (d \uparrow c_2) & 0 \times c \simeq \emptyset \\
e \times (c_1 \mathbin{\&} c_2) \simeq (e \times c_1) \mathbin{\&} (e \times c_2) & c \mathbin{\&} \emptyset \simeq c \\
d \uparrow (e \times c) \simeq (d \Uparrow e) \times (d \uparrow c) & c_1 \mathbin{\&} c_2 \simeq c_2 \mathbin{\&} c_1
\end{array}$$

$$\begin{array}{l}
d \uparrow \textbf{if } b \textbf{ within } e \textbf{ then } c_1 \textbf{ else } c_2 \simeq \\
\qquad \textbf{if } d \Uparrow b \textbf{ within } e \textbf{ then } d \uparrow c_1 \textbf{ else } d \uparrow c_2
\end{array}$$

$$(e_1 \times a(p_1 \to p_2)) \mathbin{\&} (e_2 \times a(p_1 \to p_2)) \simeq (e_1 + e_2) \times a(p_1 \to p_2)$$

Reduction Semantics

$$c \xRightarrow{\tau}_{\rho} c'$$

Reduction Semantics

$$c \overset{\tau}{\Longrightarrow}_{\rho} c'$$

$$\frac{}{a(p_1 \rightarrow p_2) \overset{\tau_{a,p_1,p_2}}{\Longrightarrow}_{\rho} \emptyset}$$

Reduction Semantics

$$c \overset{\tau}{\Longrightarrow}_{\rho} c'$$

$$\frac{}{a(p_1 \to p_2) \overset{\tau_{a,p_1,p_2}}{\Longrightarrow}_{\rho} \emptyset} \qquad \frac{c \overset{\tau}{\Longrightarrow}_{\rho} c' \quad \mathcal{E}\left[\!\left[e \right]\!\right]_{\rho} = v}{e \times c \overset{v*\tau}{\Longrightarrow}_{\rho} (-1 \Uparrow e) \times c'}$$

Reduction Semantics

$$c \xrightarrow{\tau}_{\rho} c'$$

$$\frac{}{a(p_1 \rightarrow p_2) \xrightarrow{\tau_{a,p_1,p_2}}_{\rho} \emptyset} \qquad \frac{c \xrightarrow{\tau}_{\rho} c' \quad \mathcal{E}\left[\!\left[e\right]\!\right]_{\rho} = v}{e \times c \xrightarrow{v*\tau}_{\rho} (-1 \Uparrow e) \times c'}$$

$$\vdots$$

Reduction Semantics

$$c \xrightarrow{\tau}_{\rho} c'$$

$$\frac{}{a(p_1 \to p_2) \xrightarrow{\tau_{a,p_1,p_2}}_{\rho} \emptyset} \qquad \frac{c \xrightarrow{\tau}_{\rho} c' \quad \mathcal{E}\,[\![e]\!]_{\rho} = v}{e \times c \xrightarrow{v*\tau}_{\rho} (-1 \Uparrow e) \times c'}$$

$$\vdots$$

Theorem (Reduction semantics correctness)

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

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

Code Extraction

Coq formalisation

- Denotational & reduction semantics
- Meta-theory of contracts (causality, monotonicity, . . .)
- Definition of contract transformations and analyses
- Correctness proofs

Code Extraction

Coq formalisation

- Denotational & reduction semantics
- Meta-theory of contracts (causality, monotonicity, . . .)
- Definition of contract transformations and analyses
- Correctness proofs

Code Extraction

Coq formalisation

- Denotational & reduction semantics
- Meta-theory of contracts (causality, monotonicity, . . .)
- Definition of contract transformations and analyses
- Correctness proofs

Extraction of executable Haskell code

- efficient Haskell implementation
- embedded domain-specific language for contracts
- contract analyses and contract management

Future Work

- improve code extraction
- advanced analyses and transformations
  (e.g. scenario generation and “zooming”)
- combine this work with numerical methods