Proving Compiler Correctness with Dependent Types

João Paulo Pizani Flor
Wout Elsinghorst

Department of Information and Computing Sciences
Utrecht University

Wednesday 16^{th} April, 2014

Universiteit Utrecht

Table of Contents

Table of contents

Source language, Target language

- Example source code (expression language):

```
Add (Val 1) (Add (Val 1) (Val 3))
```

- Example target code (for a stack machine):

```
PUSH 1 >> PUSH 1 >> PUSH 3 >> ADD >> ADD
```

Evaluation, execution

- An **eval** function gives the semantics for the **source** language
  - Denotational semantics
  - Maps terms to values
- An **exec** function gives the semantics for the "**machine**" language
  - For each instruction, an operation to perform on the machine state (stack)

Table of contents

What does "correct" mean?

- Both semantics (before and after compilation) should be "equivalent"
- Compiling then executing must give the same result as direct evaluation

$$
\begin{tikzcd}
\mathit{Exp} \arrow[r, "\mathit{eval}"] \arrow[d, "\mathit{compile}"'] & \mathit{Value} \arrow[d, "\mathit{push}"] \\
\mathit{Code} \arrow[r, "\mathit{exec}"'] & \mathit{Stack} \to \mathit{Stack}
\end{tikzcd}
$$

Reference paper

- "A type-correct, stack-safe, provably correct expression compiler in Epigram"
  - James McKinna, Joel Wright
- Basic ideas and proofs, which we extended...

Table of contents

Extending the source language

- More "realistic" languages have sharing constructs
- We wanted the "simplest possible" extension with sharing behaviour.
- **Chosen extension: if_then_else + sequencing**

```
if c then t else e >> common-suffix
```

- The "naïve" compile function will duplicate the suffix
- Having Bytecode defined as graph (structured graph) instead of tree would solve this problem
  - But proofs would be more complex

Table of contents

Universiteit Utrecht

What we ideally want

- Have a "smart" graph-based compiler, generating code which uses sharing
- Write the correctness proof only for the "dumb" compiler, have correctness **derived** for the smart version.

Reference paper

- "Proving Correctness of Compilers using Structured Graphs"
  - Patrick Bahr (visiting researcher)

Our project's goals

- Integrating the best of both “reference” papers
- Our contributions:
  - (Simplest possible) language extension showing sharing behaviour.
  - Proof of correctness for the **stack-safe** “naïve” compiler
    - The one that just duplicates code.
  - A way to to lift this **stack-safe** ”naïve” correctness proof
    - Into a proof concerning the more **efficient** compiler.

Table of contents

Source

- Source types:

$$\begin{aligned}&\mathsf{data}\ \mathsf{Ty}_s : \mathsf{Set}\ \mathsf{where}\\&\quad \mathbb{N}_s : \mathsf{Ty}_s\\&\quad \mathbb{B}_s : \mathsf{Ty}_s\end{aligned}$$

- Source terms (snippet):

$$\begin{aligned}&\mathsf{data}\ \mathsf{Src} : (t : \mathsf{Ty}_s) \to (z : \mathsf{Size}_s) \to \mathsf{Set}\ \mathsf{where}\\&\quad \mathsf{v}_s \quad : \forall \{t\} \to (v : \{\!\{ t \}\!\}) \to \mathsf{Src}\ t\ 1\\&\quad \_{+_s}\_ \quad : (e_1\ e_2 : \mathsf{Src}\ \mathbb{N}_s\ 1) \to \mathsf{Src}\ \mathbb{N}_s\ 1\end{aligned}$$

- Denotational semantics (snippet):

$$\begin{aligned}&[\![\_]\!] : \{t : \mathsf{Ty}_s\}\ \{z : \mathsf{Size}_s\} \to (e : \mathsf{Src}\ t\ z) \to \mathsf{Vec}\ \{\!\{ t \}\!\}\ z\\&[\![\ \mathsf{v}_s\ v\ ]\!] \qquad = [\ v\ ]\\&[\![\ e_1 +_s e_2\ ]\!] \quad = [\ [\![\ e_1\ ]\!]' + [\![\ e_2\ ]\!]'\ ]\end{aligned}$$

Bytecode

- Typed stack:

$$\mathsf{StackType} : \mathsf{Set}$$
$$\mathsf{StackType} = \mathsf{List}\ \mathsf{Ty}_s$$

$$\begin{array}{l}\mathsf{data}\ \mathsf{Stack} : \mathsf{StackType} \to \mathsf{Set}\ \mathsf{where} \\ \quad \varepsilon \quad : \mathsf{Stack}\ [] \\ \quad \_\nabla\_ \quad : \forall\ \{t\ s'\} \to \{\ t\ \} \to \mathsf{Stack}\ s' \to \mathsf{Stack}\ (t :: s')\end{array}$$

- Typed bytecode (snippet):

$$\begin{array}{l}\mathsf{data}\ \mathsf{Bytecode} : \mathsf{StackType} \to \mathsf{StackType} \to \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{SKIP} : \forall\ \{s\} \to \mathsf{Bytecode}\ s\ s \\ \quad \mathsf{PUSH} : \forall\ \{t\ s\} \to (x : \{\ t\ \}) \to \mathsf{Bytecode}\ s\ (t :: s) \\ \quad \mathsf{ADD} \ : \forall\ \{s\} \to \mathsf{Bytecode}\ (\mathbb{N}_s :: \mathbb{N}_s :: s)\ (\mathbb{N}_s :: s)\end{array}$$

Compiler correctness

$$
\begin{aligned}
&\mathsf{compile} : \forall\ \{t\ z\ s\} \to \mathsf{Src}\ t\ z \to \mathsf{Bytecode}\ s\ (\mathsf{replicate}\ z\ t\ \mathbin{+\!\!+}\ s)\\
&\mathsf{compile}\ (\mathsf{v}_s\ x) \qquad\quad = \mathsf{PUSH}\ x\\
&\mathsf{compile}\ (e_1 +_s e_2) \quad = \mathsf{compile}\ e_2 \rangle\!\rangle\ \mathsf{compile}\ e_1 \rangle\!\rangle\ \mathsf{ADD}
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{correct}\ :\ \{\mathtt{t}\ :\ \mathtt{Ty}_s\}\ \{\mathtt{z}\ :\ \mathtt{Size}_s\}\ (\mathtt{e}\ :\ \mathtt{Src}\ \mathtt{t}\ \mathtt{z})\\
&\qquad \to \mathtt{exec}\ (\mathtt{compile}\ \mathtt{e}) \equiv [\![\ \mathtt{e}\ ]\!]
\end{aligned}
$$

Table of contents

Tree fixpoints

- Fixed Point for standard Functors

```
data Tree (f : Set -¿ Set) : Set where
  In : f (Tree f) -¿ Tree f
```

- Fixed Point for indexed Functors

Bytecode Tree Representation

$$
\begin{array}{l}
\textsf{data Bytecode} : \textsf{StackType} \to \textsf{StackType} \to \textsf{Set where} \\
\quad \textsf{SKIP} : \forall \; \{s\} \qquad \to \textsf{Bytecode}\; s\; s \\
\quad \textsf{PUSH} : \forall \; \{t\; s\} \quad \to (x : \{\, t \,\}) \to \textsf{Bytecode}\; s\; (t :: s) \\
\quad \textsf{ADD} \;\; : \forall \; \{s\} \qquad \to \textsf{Bytecode}\; (\mathbb{N}_s :: \mathbb{N}_s :: s)\; (\mathbb{N}_s :: s)
\end{array}
$$

$$
\begin{array}{l}
\textsf{data BytecodeF}\; (r : \textsf{StackType -¿ StackType -¿ Set}) : (\textsf{StackType -¿} \\
\quad \textsf{SKIP'} : \forall \; \{s\} \qquad \to \textsf{BytecodeF}\; r\; s\; s \\
\quad \textsf{PUSH'} : \forall \; \{a\; s\} \quad \to (x : \{\, a \,\}) \to \textsf{BytecodeF}\; r\; s\; (a :: s) \\
\quad \textsf{ADD'} \;\; : \forall \; \{s\} \qquad \to \textsf{BytecodeF}\; r\; (\mathbb{N}_s :: \mathbb{N}_s :: s)\; (\mathbb{N}_s :: s)
\end{array}
$$

- Bytecode is isomorphic to `HTree BytecodeF`
  - `fromTree` ∘ `toTree` ≡ `id`
  - `toTree` ∘ `fromTree` ≡ `id`

Correctness on Trees

compileT : ∀ {t z s} → Src t z → HTree BytecodeF s (replicate z t ++

execT : ∀ {s s'} → HTree BytecodeF s s' -¿ Stack s -¿ Stack s'

```
correctT : ∀ {t z s'} → (e : Src t z)
         → execT (compileT e) ≡ ⟦ e ⟧
```

- Proof of `correctT` can be derived from `correct`
  - Because ‘Bytecode‘ is structurally the same as ‘HTree BytecodeF‘

Graphs

```
data HGraph .. : ... -> Set where ...
```

- `HGraph` is similar ("includes") `HTree`
  - But with extra constructors to represent *shared subgraphs*
- `Bytecode` is not *exactly* isomorphic to `HGraph BytecodeF`:
  - We have: `fromGraph` $\circ$ `toGraph` $\equiv$ `id`
  - But: `toGraph` $\circ$ `fromGraph` $\not\equiv$ `id`
  - `HGraph` $\rightarrow$ `Bytecode` $\rightarrow$ `HGraph` loses sharing

Bytecode Graph Representation

$$\text{compileG} : \{s : \text{StackType}\} \to \forall\ \{z\ t\}\ \text{-¿}\ \text{Src}\ t\ z \to \text{HGraph Bytecode}$$

$$\text{execG} : \forall\ \{s\ s'\} \to \text{HGraph BytecodeF}\ s\ s'\ \text{-¿}\ \text{Stack}\ s\ \text{-¿}\ \text{Stack}\ s'$$

```
correctG : ∀ {t z} → (e : Src t z)
        → execG (compileG e) ≡ ⟦ e ⟧
```

Using machinery, we get this proof derived from ‘correctT‘

Universiteit Utrecht

Achieved

- Agda "framework" for deriving compiler correctness proofs
  - Compilers with **typed source** and **typed target**
  - Given correctness of a "naïve" compiler, derive correctness of "optimized" version
- Correctness proof for an expression language (with sequencing)
  - As "instance" of this framework

Agda limitations we faced

- Strict positivity requirement
  - When defining fixed point type operators
- Totality checker
  - When defining folds
- Type checking with positivity check disabled made debugging hard
  - Stack overflow, memory consumption

Yet to be done

- Sequence clause of “basic” (non-lifted) correctness proof
- Prove a final lemma to complete the lifting (fusion law)

Thank you!

Questions?