Proving Compiler Correctness with Dependent Types

João Paulo Pizani Flor
Wout Elsinghorst

Department of Information and Computing Sciences
Utrecht University

Wednesday 16th April, 2014

Universiteit Utrecht

Table of Contents

Universiteit Utrecht

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

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

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

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:

```
data Tys : Set where
  ℕs : Tys
  𝔹s : Tys
```

$$\mathsf{data}\ \mathsf{Ty}_s : \mathsf{Set}\ \mathsf{where}$$
$$\mathbb{N}_s : \mathsf{Ty}_s$$
$$\mathbb{B}_s : \mathsf{Ty}_s$$

Source terms (snippet):

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

Denotational semantics (snippet):

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

Bytecode

Typed stack:

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

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

Typed bytecode (snippet):

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

Compiler correctness

Table of contents

Tree fixpoints

Universiteit Utrecht

Bytecode Tree Representation

Universiteit Utrecht

Correctness on Trees

Graphs

Bytecode Graph Representation

Universiteit Utrecht

Achieved

- Developed a framework for producing compiler correctness proofs for *typed* languages with sharing constructs, given proofs which don't involve sharing.
- Gave a specific proof for a simple example language, as "instance" of this framework.

Agda limitations we faced

- Strict positivity requirement (when defining fixpoints)
- 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
- Proof a final lemma to complete the lifting (fusion law)

Thank you!

Questions?