Proving Compiler Correctness with Dependent Types

João Paulo Pizani Flor
Wout Elsinghorst

Department of Information and Computing Sciences
Utrecht University

Tuesday 15th April, 2014

Universiteit Utrecht

Table of Contents

Source code, Bytecode, eval, compile, exec

Source:

$$\begin{array}{l}\textsf{data}\ \textsf{Ty}_s : \textsf{Set}\ \textsf{where} \\ \quad \mathbb{N}_s : \textsf{Ty}_s \\ \quad \mathbb{B}_s : \textsf{Ty}_s\end{array}$$

Bytecode (stack code, for example):

```
PUSH 2 >> PUSH 3 >> ADD
```

What does "correct" mean?

- Given a semantics for the source language ("eval")
- This semantics is *respected* by the compiler.

Correctness equation:

```
eval e = exec (compile e)
```

Reference paper

- "A type-correct, stack-safe, provably correct expression compiler in Epigram"
  - James McKinna, Joel Wright

Extending the source language

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

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

Our project's goals

Integrating the approaches of the two "reference" papers.
Our contributions:

- (Simplest possible) language extension showing sharing behaviour.
- Proof of correctness for the **stack-safe** "simple" compiler that just duplicates code.
- A way to to lift this **stack-safe** correctness proof to one for a more **efficient** compiler.

Source

Source types:

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

Source terms (snippet):

$$\begin{array}{lll}\multicolumn{3}{l}{\mathsf{data}\ \mathsf{Src} : (t : \mathsf{Ty}_s) \to (z : \mathsf{Size}_s) \to \mathsf{Set}\ \mathsf{where}}\\ \quad \mathsf{v}_s & : & \forall\ \{t\} \to (v : \{\ t\ \}) \to \mathsf{Src}\ t\ 1\\ \quad \_{+_s}\_ & : & (e_1\ e_2 : \mathsf{Src}\ \mathbb{N}_s\ 1) \to \mathsf{Src}\ \mathbb{N}_s\ 1\end{array}$$

Denotational semantics (snippet):

$$\begin{array}{lll}\multicolumn{3}{l}{[\![\_]\!] : \{t : \mathsf{Ty}_s\}\ \{z : \mathsf{Size}_s\} \to (e : \mathsf{Src}\ t\ z) \to \mathsf{Vec}\ \{\ t\ \}\ z}\\ [\![\ \mathsf{v}_s\ v\ ]\!] & = & [\ v\ ]\\ [\![\ e_1 +_s e_2\ ]\!] & = & [\ [\![\ e_1\ ]\!]' + [\![\ e_2\ ]\!]'\ ]\end{array}$$

Bytecode

Typed stack:

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

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

Typed bytecode (snippet):

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

Compiler correctness

Tree fixpoints

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?