Systems and Formalisms Lab

Bootstrapping a verified compiler for an imperative language in Rocq

Kacper F. Korban, SYSTEMF, EPFL

Supervised by **Magnus O. Myreen**, Chalmers University of Technology, University of Gothenburg and **Clément Pit-Claudel**, EPFL

Why bootstrap a verified compiler?

Bootstrapping a verified compiler -> No extraction

Bootstrapping *noun* /ˈbuːtˌstræpɪŋ/
1. the process of creating a compiler executable in which the compiler compiles its own source code

Self-hosting *adjective* /ˌselfˈhəʊstɪŋ/
1. of a language whose compiler is implemented in itself

Verified compiler setup

IMPL language

$$
\begin{aligned}
e &::= x \mid n \mid e_1 + e_2 \\
&\mid e_1 - e_2 \mid e_1 \setminus e_2 \mid e_1[e_2] \\
cmp &::= < \mid = \\
t &::= e_1\ cmp\ e_2 \mid \text{not } t \\
&\mid t_1 \wedge t_2 \mid t_1 \vee t_2 \\
c &::= \varepsilon \mid c_1 ; c_2 \mid x = e \\
&\mid e_1[e_2] = e_3 \mid \textbf{if } t \textbf{ then } c_1 \textbf{ else } c_2 \\
&\mid \textbf{while } t\ c \mid x = f(e_1, ..., e_k) \\
&\mid \textbf{return } e \mid x = \text{alloc}(e) \mid \text{abort} \\
&\mid \text{putchar } e \mid \text{getchar } x
\end{aligned}
$$

ASM langage

$$
\begin{aligned}
c &::= \text{always} \mid \text{less } r_1\ r_2 \mid \text{equal } r_1\ r_2 \\
i &::= \text{const } r\ w \mid \text{add } r_1\ r_2 \mid \text{sub } r_1\ r_2 \\
&\mid \text{div } r \mid \text{jump } c\ n \mid \text{call } n \\
&\mid \text{mov } r_1\ r_2 \mid \text{ret} \mid \text{pop } r \\
&\mid \text{push } r \mid \text{add\_rsp } n \\
&\mid \text{sub\_rsp } n \mid \text{load\_rsp } r\ n \\
&\mid \text{store\_rsp } r\ n \mid \text{load } r_1\ r_2\ o \\
&\mid \text{store } r_1\ r_2\ o \mid \text{getchar} \mid \text{putchar} \\
&\mid \text{exit} \mid \text{comment } s
\end{aligned}
$$

impl_to_asm_Rocq

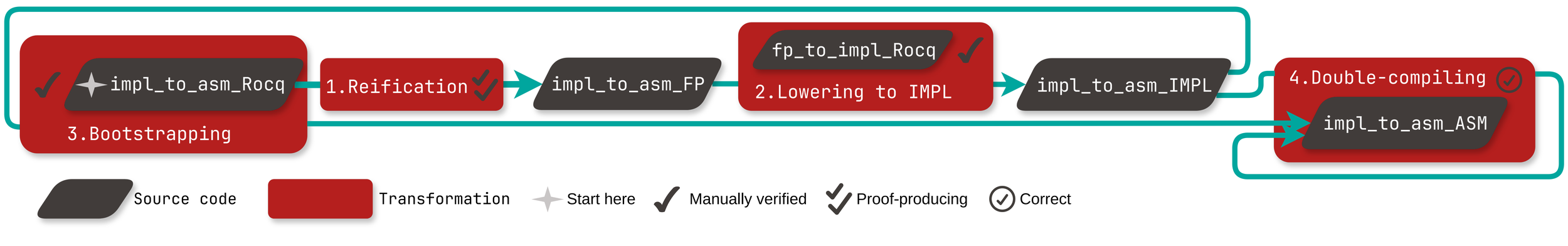
```
...
Definition c_add: asm_appl :=
  let/d pop_rdi :=
    ASMSyntax.Pop RDI in
  let/d add_rax_rdi :=
    ASMSyntax.Add RAX RDI in
  let/d asm1 :=
    [pop_rdi; add_rax_rdi] in
  let/d res := List asm1 in
  res.
...
Definition impl_to_asm_Rocq
  (prog: IMPL): ASM := ...
```

We bootstrapped a verified compiler for an imperative language in Rocq

Contributions:
- First standalone bootstrapping of a verified compiler in Rocq
- First standalone bootstrapping of a verified compiler for an imperative language
- New variant of relational compilation for proof-producing reification of a subset of Gallina

Bootstrap a verified compiler in 3 steps!

```
(* Reify the compiler code to FP using a tactic *)
let impl_to_asm_FP: FP := ltac2:(reify '@impl_to_asm_Rocq) in
(* Evaluate fp_to_impl_Rocq code generator in-logic on *)
(*  the reified deeply embedded compiler in FP *)
let impl_to_asm_IMPL: IMPL := fp_to_impl_Rocq impl_to_asm_FP in
(* Evaluate the compiler's own code generator in-logic *)
(*  on the reified deeply embedded compiler in IMPL *)
let impl_to_asm_ASM: ASM := impl_to_asm_Rocq impl_to_asm_IMPL in
(* Prettify the resulting ASM code of the compiler *)
pretty_print impl_to_asm_ASM
```



1. Reification (Gallina -> FP)

a. Define a target language

$$
\begin{aligned}
o &::= + \mid - \mid \times \mid / \mid \text{cons} \\
&\mid \text{head} \mid \text{tail} \mid \text{read} \mid \text{write} \\
t &::= < \mid = \\
e &::= n \mid x \mid o(e_1, \ldots, e_k) \mid f(e_1, \ldots, e_k) \\
&\mid \textbf{if } e_1\ t\ e_2 \textbf{ then } e_3 \textbf{ else } e_4 \mid \textbf{let } x = e_1 \textbf{ in } e_2
\end{aligned}
$$

b. State shallow to deep reification (compilation) as a proof search problem,

$$
\begin{aligned}
&\exists\ (impl\_to\_asm\_FP : \text{FP}). \\
&\quad \forall\ (inp : \text{string}). \\
&\qquad (impl\_to\_asm\_FP,\ inp) \\
&\qquad\quad \Downarrow_{\text{FP}} \text{impl\_to\_asm\_Rocq } inp
\end{aligned}
$$

c. Define compilation lemmas e.g.,

$$
\begin{aligned}
&x1 \Downarrow_{\text{FP}} n1 \rightarrow x2 \Downarrow_{\text{FP}} n2 \rightarrow \\
&\texttt{(Op Add [}x1\texttt{;}x2\texttt{])} \Downarrow_{\text{FP}} n1 + n2
\end{aligned}
$$

d. Write a compiler in Ltac2 for this decision procedure

```
repeat (match! goal with
  | ... ⇒
    eapply compilation_lemmaX
  ...
end).
```

```
Defun (name_enc "c_add") []
  (Let (name_enc "pop_rdi")
    (Op Cons
      [FunSyntax.Const (name_enc "Pop");
       Op Cons [FunSyntax.Const (name_enc "RDI"); FunSyntax.Const 0]])
    (Let (name_enc "add_rax_rdi")
      (Op Cons
        [FunSyntax.Const (name_enc "Add");…
```

2. Lowering to IMPL (FP -> IMPL)

a. Define a code generator from FP to IMPL

```
Definition fp_to_impl_Rocq
  (p: FP): option IMPL := ...
```

b. Verify it

$$
\begin{aligned}
&\forall\ (fp : \text{FP}), (inp : \text{string}), (o_1\, o_2 : \text{string}). \\
&\quad (fp,\ inp) \Downarrow_{\text{FP}} o_1 \rightarrow \\
&\quad (\text{fp\_to\_impl\_Rocq } fp,\ inp) \Downarrow_{\text{IMPL}} o_2 \rightarrow \\
&\qquad o_2 \approx o_1
\end{aligned}
$$

```
Func 426801980516 []
  (Seq
    (Call 31647725819946089 124416
      [Const 5271408; Const 5391433; Const 0])
  (Seq
    (Call 1177398927029801159965010 97 124672
      [Const 4285540; Const 5390680; Const 5391433; Const 0])
  (Seq…
```

3. Bootstrapping (IMPL -> ASM)

1. Compile `impl_to_asm` with itself

   impl_to_asm_ASM ≜ impl_to_asm_Rocq `impl_to_asm_IMPL`

2. Apply `impl_to_asm_Rocq` correctness

$$
\begin{aligned}
&\forall\ (inp : \text{string}), (o : \text{string}). \\
&\quad (\text{impl\_to\_asm\_ASM},\ inp) \Downarrow_{\text{ASM}} o \rightarrow \\
&\qquad o \approx \text{impl\_to\_asm\_Rocq } inp
\end{aligned}
$$

```
[ASMSyntax.Sub_RSP 4;
 ASMSyntax.Push ASMSyntax.RAX;
 ASMSyntax.Const ASMSyntax.RAX 5271408;
 ASMSyntax.Push ASMSyntax.RAX;
 ASMSyntax.Const ASMSyntax.RAX 5391433;
 ASMSyntax.Push ASMSyntax.RAX;
 ASMSyntax.Const ASMSyntax.RAX 0;
 ASMSyntax.Pop ASMSyntax.RDI;…
```

4. Double-compiling (IMPL -> ASM)

1. Compile `impl_to_asm` with its assembly

$$
\begin{aligned}
&(\texttt{impl\_to\_asm\_ASM},\ \texttt{impl\_to\_asm\_IMPL}) \\
&\quad \Downarrow_{\text{ASM}}\ \texttt{impl\_to\_asm\_ASM}_{DC}
\end{aligned}
$$

2. Check that bootstrapping fixpoints

$$
\texttt{impl\_to\_asm\_ASM}_{DC} = \texttt{impl\_to\_asm\_ASM}
$$

```
[ASMSyntax.Sub_RSP 4;
 ASMSyntax.Push ASMSyntax.RAX;
 ASMSyntax.Const ASMSyntax.RAX 5271408;
 ASMSyntax.Push ASMSyntax.RAX;
 ASMSyntax.Const ASMSyntax.RAX 5391433;
 ASMSyntax.Push ASMSyntax.RAX;
 ASMSyntax.Const ASMSyntax.RAX 0;
 ASMSyntax.Pop ASMSyntax.RDI;…
```

Ask me about:

- Why don't we need to verify the parser?
- Why is fp_to_impl_Rocq easy to write and verify?
- How we reify non-structural recursion?
- How does this relate to connected efforts in HOL4?
- How we made HOL4-style proofs work in Rocq?

Related work

(Curzon, 1993) (Myreen & Owens, 2014) (Kumar et al., 2014) (Mullen et al., 2018) (Myreen, 2021) (Abrahamsson et al., 2020) (Pit-Claudel et al., 2022) (Forster et al., 2024)