Formally Verified Native Code Generation in an Effectful JIT
Artifact Documentation

1 Kick-The-Tires Instructions

The artifact is available as a Virtual Machine image.
Login: popl23
Password: popl23

The virtual machine already contains compiled files for the JIT. For the kick-the-tires phase, you can simply execute it. Note that the keyboard layout can be changed from AZERTY to QWERTY by clicking on the **fr** button in the top right of the screen.

- Open the VM and a terminal and go to the develoment directory.
  `cd /home/popl23/FM-JIT/coqjit`
- Run the pre-made tests.
  `./RunTests.sh`
  This runs the JIT on a prime search program located in `progs_IR/prime.ir`. The first run compiles the internal function after 2 calls, then calls the dynamically generated native codes. The second run disables native code generation and simply interprets the entire program. This should take around 2 minutes. The expected result is that the first run executes much faster than the second one.

For the full evaluation instructions, check the following sections.

2 Artifact Description

Our development is located in the `FM-JIT` directory.

- The `FM-JIT/coqjit` directory contains the code and proofs that has been written for the JIT compiler.
- Files ending in `_proof.v` contain the various proofs presented in the submission.
- `FM-JIT/c_primitives` is the C library of primitive implementations.

- `coqjit/compcert_coq` is taken as is from CompCert 3.8. Coq definitions and proofs.
- `coqjit/compcert_ocaml` is similarly the OCaml development from the same CompCert 3.8.
- `coqjit/flocq` and `coqjit/MenhirLib` are Coq libraries, also from CompCert 3.8.
- `coqjit/parsing` contains a custom unverified parser for this version of CoreIR, using Menhir.
- `coqjit/progs_IR` contains a few example CoreIR programs, that can be parsed and executed using the JIT.

Coq files can be stepped through using either Proof General for Emacs or CoqIDE, both installed on the virtual machine. For instance, you can inspect the `jit_proof.v` file using either `emacs jit_proof.v` or `coqide jit_proof.v`.

2.1 Renamings

A few things have different names between the submission and the development. We list them here:

Name in the submission	Name in the develoment	Location
Monad specification	`monad_impl`	`monad_impl.v`
Reference Specification	`naive_impl`	`monad_impl.v`
Primitive Specification	`array_impl`	`monad_impl.v`
Theorem `jit_correctness_simulation`	`jit_correctness_array`	`jit_proof.v`
Theorem `jit_correctness`	`jit_preservation_array`	`jit_proof.v`
CoreIR semantics	`input_sem`	`mixed_sem.v`
`Prim_Pop` and `Prim_Push`	`Prim_Save` and `Prim_Load`	`monad.v`
`Prim_HeapSet` and `Prim_HeapGet`	`Prim_MemSet` and `Prim_MemGet`	`monad.v`
`get_prim`	`exec_prim`	`monad_impl.v`
`free_to_state`	`exec`	`monad_impl.v`
`start_` and `end_`	`load_` and `ret_` in `na_spec`	`monad_properties.v`
`free_interpreter`	`nm_exec`	`main.ml`

2.2 Axioms

The three axioms discussed in the submission (Section 6.4) are

- `no_builtin` in `backend_proof.v`
- `ext_prim_axiom` in `primitives.v`
- `external_in_memory` in `flattenRTL_proof.v`

Other axioms that may appear during a `Print Assumptions` are either parameters realized during extraction, or axioms from the CompCert development. We can check that these are the only Axioms in the JIT development by running, from the `coqjit` directory:
`grep 'Axiom' *.v && grep 'Admitted' *.v`

3 Full Compilation Instructions

The artifact virtual machine contains pre-compiled files. In this section, we show how to recompile everything.

3.1 Compiling the JIT

- Go to the development directory.
  `cd /home/popl23/FM-JIT/coqjit`
- Clean the existing development.
  `make clean_all`
  `clean_all` removes everything, including the CompCert compiled files that are part of our development. `clean` allows to clean the JIT files without cleaning the CompCert compiled files.
- Build the JIT.
  `make`
  This first compiles the Coq files. There might some Coq warnings. Some of them are due to us (and the CompCert files we included) using deprecated Coq features that are still supported by the Coq version in the VM. Other warnings state that some Coq names are redefined. This is because some definitions are adapted from CompCert but slightly modified for the JIT. There should be warnings, but there should not be any compilation error.

  After the Coq files are compiled, this will build the C library of primitives (in `../c_primitives`). Then, the OCaml extracted JIT is compiled. Finally, the JIT correctness proofs are compiled.

  The entire compilation process should take around 15 minutes.

3.2 Compiling CompCert

We also use CompCert 3.8 to compile the C library. We have included CompCert 3.8 in the home directory. You can also re-compile CompCert.

- Move to the CompCert directory.
  `cd /home/popl23/CompCert-3.8`
- Clean the CompCert development.
  `make clean`

- Build CompCert.
  `make`
  This should take around 10 minutes.
- Install CompCert.
  `sudo make install`

4 List of Claims

CompCert reuse We claim reusing the CompCert backend and its proofs to formally verify the native code generation in a JIT. In particular, the `transf_rtl_program_` function in `backend.v` uses multiple passes from the CompCert backend, located in the `compcert_coq` directory. The `compcert_backend_forward` theorem of `backend_proof.v` reuses proofs from CompCert.

JIT correctness proof The JIT theorems described in the submission can be found at the end of the `jit_proof.v` file, with some renamings listed in Section 2.1 of this artifact documentation.

Speedups We claim that generating native code produces substantial speedups in JIT execution (Section 6.1). Running `./RunTests.sh` runs the prime search example that is mentioned in the submission. With the version given in the artefact, we can observe around 40 times speedup in the VM. Note that when not in a VM, we have observed even greater speedups.

Modifying the prime search bound on line 7 of `coqjit/progs_IR/prime.ir` can change the observed speedup. With a greater bound comes greater speedups.

Testing other features of FM-JIT In Section 6.1, we also claim having tested the JIT on other features. For instance, a program that uses the heap can be executed:

```
cd /home/popl23/FM-JIT/coqjit
./jit progs_IR/heap.ir
```

Expected result: 28 (the value that is stored and retrieved in the heap).

A program that contains an `Assume` speculation can also be executed:

```
cd /home/popl23/FM-JIT/coqjit
./jit progs_IR/deopt.ir
```

Expected result: 1. Using the `-c` option (`./jit progs_IR/deopt.ir -c`) shows that deoptimization is triggered from native code execution.