SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

EXTRACTION

EXTRACTING ML FROM COQ

Basic Extraction

In its simplest form, extracting an efficient program from one written in Coq is completely straightforward.

First we say what language we want to extract into. Options are OCaml (the most mature), Haskell (mostly works), and Scheme (a bit out of date).

```
Require Coq.extraction.Extraction.
Extraction Language Ocaml.
```

Now we load up the Coq environment with some definitions, either directly or by importing them from other modules.

```
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import ImpCEvalFun.
```

Finally, we tell Coq the name of a definition to extract and the name of a file to put the extracted code into.

```
Extraction "imp1.ml" ceval_step.
```

When Coq processes this command, it generates a file `imp1.ml` containing an extracted version of `ceval_step`, together with everything that it recursively depends on. Compile the present `.v` file and have a look at `imp1.ml` now.

Controlling Extraction of Specific Types

We can tell Coq to extract certain `Inductive` definitions to specific OCaml types. For each one, we must say

- how the Coq type itself should be represented in OCaml, and
- how each constructor should be translated.

```
Extract Inductive bool ⇒ "bool" [ "true" "false" ].
```

Also, for non-enumeration types (where the constructors take arguments), we give an OCaml expression that can be used as a "recursor" over elements of the type. (Think Church numerals.)

```
Extract Inductive nat ⇒ "int"
  [ "0" "(fun x → x + 1)" ]
  "(fun zero succ n →
      if n=0 then zero () else succ (n-1))".
```

We can also extract defined constants to specific OCaml terms or operators.

```
Extract Constant plus ⇒ "( + )".
Extract Constant mult ⇒ "( * )".
Extract Constant beq_nat ⇒ "( = )".
```

Important: It is entirely *your responsibility* to make sure that the translations you're proving make sense. For example, it might be tempting to include this one

```
Extract Constant minus ⇒ "( - )".
```

but doing so could lead to serious confusion! (Why?)

```
Extraction "imp2.ml" ceval_step.
```

Have a look at the file `imp2.ml`. Notice how the fundamental definitions have changed from `imp1.ml`.

A Complete Example

To use our extracted evaluator to run Imp programs, all we need to add is a tiny driver program that calls the evaluator and prints out the result.

For simplicity, we'll print results by dumping out the first four memory locations in the final state.

Also, to make it easier to type in examples, let's extract a parser from the `ImpParser` Coq module. To do this, we first need to set up the right correspondence between Coq strings and lists of OCaml characters.

```
Require Import ExtrOcamlBasic.
Require Import ExtrOcamlString.
```

We also need one more variant of booleans.

```
Extract Inductive sumbool ⇒ "bool" ["true" "false"].
```

The extraction is the same as always.

```
Require Import Imp.
Require Import ImpParser.

Require Import Maps.
Definition empty_state := { --> 0 }.
Extraction "imp.ml" empty_state ceval_step parse.
```

Now let's run our generated Imp evaluator. First, have a look at `impdriver.ml`. (This was written by hand, not extracted.)

Next, compile the driver together with the extracted code and execute it, as follows.

```
ocamlc -w -20 -w -26 -o impdriver imp.mli imp.ml impdriver.ml
./impdriver
```

(The `-w` flags to `ocamlc` are just there to suppress a few spurious warnings.)

Discussion

Since we've proved that the `ceval_step` function behaves the same as the `ceval` relation in an appropriate sense, the extracted program can be viewed as a *certified* Imp interpreter. Of course, the parser we're using is not certified, since we didn't prove anything about it!

Going Further

Further details about extraction can be found in the Extract chapter in *Verified Functional Algorithms* (*Software Foundations* volume 3).