Verified, Validated, or Certified?

Harry R. Schwartz

February 17, 2017

What's the difference between a *verifying* compiler, a *validating* compiler, and a *certifying* compiler?

All three of these strategies accomplish the same goal: given source code

$$S$$

, we should be able to prove that a compiler generates target code

$$C$$

such that

$$S$$

and

$$C$$

satisfy the same specification. We write this as

$$S \approx C$$

, which indicates that

$$S$$

and

$$C$$

are semantically equivalent.

I didn't understand the difference between these strategies—I think I thought they were all vaguely synonymous?—but §2 of Leroy's *Formal Verification of a Realistic Compiler*[1] clarifies the distinctions.

A ***verified*** compiler is accompanied by a proof that, for any

$$S$$

[1]Xavier Leroy, *Formal verification of a realistic compiler*. Communications of the ACM 52(7), July 2009.

, it'll either throw an error[2] or generate code

$$C$$

such that

$$S \approx C$$

. This is the direct approach that usually manifests as "write it in Coq and export it to OCaml." A verified compiler is correct by construction.

A ***validated*** compiler has two components: an unverified compiler and a verified *validator*. The validator is a Boolean function that takes

$$S$$

and

$$C$$

and returns *true* if

$$S \approx C$$

. In other words:

$$\forall S, C, \mathit{Validate}(S, C) \rightarrow S \approx C$$

Note that the general problem of determining whether

$$S \approx C$$

for any

$$S$$

and

$$C$$

is undecidable, so most validators would probably conservatively return *false* if

$$S \approx C$$

couldn't be determined.

A ***certified*** compiler passes the verification buck along to its users. Given source code

$$S$$

, it'll either fail to compile or produce code

$$C$$

[2]Note that a compiler that *only* throws errors satisfies this definition. I was delighted to learn that I've successfully (if unknowingly) written a verified compiler!

and certificate

$$\pi$$

, where

$$\pi$$

is a proof that

$$C \approx S$$

. The user (or, more likely, a client library) can then check that

$$\pi$$

is correct with a verified proof checker.

The compiler itself doesn't need to be verified, since users can ascertain for themselves (using the certificate) that the compiled code has the same specification as the source.

To sum up, these three approaches all guarantee that

$$S \approx C$$

:

- A verified compiler,
- An unverified compiler with a verified validator, and
- An unverified certifying compiler paired with a verified client-side proof checker.

If you want to prove that your compiler preserves semantics, then, you'd probably want to pick one of these three strategies. You could also structure your compiler as a series of passes, each of which would use one of these. Passes compose trivially, so a chain of correct passes would yield a correct compiler.

These terms don't just apply to compilers! They're used all over the place in the study of formal methods and can be applied to any kind of algorithm. I'm only emphasizing compilers in this post because they're a convenient example (and because they're the topic of the CompCert paper).

Now entering the realm of speculation. . .

Since all three of these strategies make the same guarantees, could we automatically change strategies? For example, given a validating compiler, could we automatically construct an equivalent verified compiler for the same specification? Leroy notes that it's theoretically possible to construct a certifying compiler from a verified compiler, but I wonder if that's true more generally. Do these form an equivalence class? If so, what's the name of that class, and what else is in it? This sounds terribly hard to implement, but I'd be curious to know if it's theoretically possible.