Bootstrapping Compiler-Verifiers

Mario Carneiro
Carnegie Mellon University
Pittsburgh, PA, USA
mcarneir@andrew.cmu.edu

1 Introduction

Fix an input language $\mathcal{L}$ and output set $\mathcal{M}$, and let $R \subseteq \mathcal{L} \times \mathcal{M}$. We will write $M \in R_s$ for $(s, M) \in R$. A compiler-verifier for R is a program P such that given inputs s, x, if $s \in \mathcal{L}$, then either $P(s, x)$ does not terminate or indicates failure, or $P(s, x) \in R_s$.

This is a very broad definition, but we are primarily interested in the case where $\mathcal{L}$ is the collection of valid program specifications in some language, $\mathcal{M}$ is the collection of programs in some formalism (lambda terms, Turing machines, x86 machine code files), and $M \in R_s$ whenever the program M satisfies the specification s.

The input x does not apparently play a role in the definition, so one may wonder what its purpose is. The idea is that x is some kind of "implementation hint" for the compiler to actually produce an $M \in R_s$. Failure is always an option, and there is no requirement that there exist an x making the compiler successful, but the x can play the role of an oracle to guide the compiler to a solution. The point is that the compiler only ever produces $M \in R_s$, regardless of the value of x, although the particular M produced may depend on x. So x need not be "trusted" to trust M or its behavior.

2 Bootstrapping

With this simple formalization it is easy to describe a bootstrapping compiler verifier. Let s be an encoding of $(\mathcal{L}, \mathcal{M}, \mathrm{eval}, R)$. Then $P \in R_s$ if $\mathrm{eval}(P)$ is a compiler-verifier for R.

Okay, that's a bit too terse. We have three kinds of encoding happening: s is encoding a (pair of) languages, a property of (pairs of) strings in those languages, and P is encoding a program via eval. To encode the language we can use context free grammars, which are sufficiently general for our purposes. To encode the property we need at least a first order language; the property of being a compiler verifier is Π_1 so we need at least this much complexity in the specification language. To encode the machine, we need an evaluation function eval from M to partial functions $\{0,1\}^* \to \{0,1\}^*$ or similar.

Finally, there is the task of actually making a nontrivial (not always failing) compiler verifier that is capable of compiling itself. We are given an encoding of $(\mathcal{L}, \mathcal{M}, \mathrm{eval}, R)$, and need to produce a compiler-verifier for R (which is itself nontrivial). Really we don't need the input x, we can just do a brute force search for programs provably satisfying R, but x lets us "brute force" in exponentially less time, by telling us what decision to make at each turn.

Nevertheless, it is still not an easy task. Note that the notion of "proof" has appeared now; we have to check a Π_1 property in finite time so we need to use proofs to reduce the search space. But this also means that the program must be given some axiomatic strength, so it can relate provability to truth. We aren't going to completely close the loop here by Gödel incompleteness.

The axioms here appear in the statement that our program indeed produces compiler-verifiers. The property we need of the proof system is that anything provable is true (we can relativize this, but then we get statements like "$T \vdash$ 'the program P is a compiler verifier'" which are harder to interpret). So let us introduce a proof theory T, with a language $\mathcal{T}$ of proof terms. The property of P we wish to hold is "for all s, x, M, if $s \in \mathcal{L}$ and $\mathrm{eval}(P)(s, x) = M$ terminates successfully then $M \in R_s$". We abbreviate this as $T \vdash \mathrm{eval}(P) \triangleleft R$; note that it depends on an encoding of $\mathcal{L}$, eval, and R in the language of $\mathcal{T}$.

Thus, we take X, the auxiliary input to our compiler-verifier generator, to be a pair (p, D), where $P = C(p)$ is the compiler, and $T \vdash D : p\ \mathsf{ok} \wedge \mathrm{eval}'(p) \triangleleft R$ is checked. Then, the assumptions needed for correctness are:

1. If $T \vdash \phi$, that is, $T \vdash D : \phi$ for some D, then the interpretation of ϕ is true. (i.e. T is sound)
2. If $T \vdash p\ \mathsf{ok}$, then $T \vdash \mathrm{eval}'(p) = \mathrm{eval}(C(p))$. (soundness of the compiler)
3. If the program verifies that $T \vdash D : \phi$, then $T \vdash D : \phi$. (correctness of the verifier)
4. If the program computes $C(p) = P$, then $C(p) = P$. (correctness of the compiler)

The first assumption reduces to the truth of all the axioms and rules of inference of T. This includes the correctness of the interpretations of constant symbols in T that correspond to $\mathcal{L}$, eval and R. In other words, the execution model is correct. Also C, which appears both as a function in the language $\mathcal{T}$ and a real function from p to P, should be interpreted correctly.

The second assumption is a theorem about programs in the language used for p; it need not be a separate assumption (it can be trivially satisfied if $C = \mathsf{id}$ and $\mathrm{eval}' = \mathrm{eval}$) but is useful for when there is a nontrivial compilation component.

PL'18, January 01–03, 2018, New York, NY, USA
2018. ACM ISBN …$15.00
https://doi.org/

We can also dispense with this by proving $T \vdash \forall p,\ (p\ \mathrm{ok} \to \mathrm{eval}'(p) = \mathrm{eval}(C(p)))$.

The third assumption reduces to checking that the program adequately manipulates proof steps according to the rules of T, and the fourth assumption reduces to checking that the program correctly implements the compilation function. Only these two require direct inspection of the code of the compiler-verifier generator itself. The rest requires only the correctness of the abstract objects with respect to the theory T.

2.1 Bootstrapping again

Now imagine we step up one meta-level; we wish to describe a compiler-verifier that produces *this* compiler-verifier. The language $\widehat{\mathcal{L}}$ is the set of codes for tuples $(\mathcal{L}, \mathcal{M}, \mathrm{eval}, R)$. To code a language, we can use a CFG written in e.g. Backus-Naur form. For the relation R, we can use the encoding in theory T. Everything must be embeddable in theory T, although at this stage we only require it be encoded in some form.

We can let the set of machines $\widehat{\mathcal{M}}$ be the machines in our favorite interpretation; it can be real computers if you like. eval in the tuple above is an encoding of a function $\mathcal{M} \to \widehat{\mathcal{M}}$, while the interpretation of $\mathrm{eval}(M)$ in T will perform steps in the same way as the program $\mathrm{eval}(M)$. The function $\widehat{\mathrm{eval}}(M)$ where $M \in \widehat{\mathcal{M}}$ is just evaluation in the chosen metalanguage.

The relation $\widehat{R}$ holds at $((\mathcal{L}, \mathcal{M}, \mathrm{eval}, R), P)$ if $P \in \widehat{\mathcal{M}}$ is a program such that $(s, P(s, x)) \in R$ (interpreted), for all $s \in \mathcal{L}$ and all x.

Note that we don't have to mention the theory T here. We only care about pure correctness, and the theory and proof language was only a means to that end, "internal" to the compiler-verifier implementation. This is important because it means we can freely change theories from one level to the next. Additionally, the target language $\mathcal{M}$ is not fixed, so we can compile from more complicated languages.

Now we have described a specification $(\widehat{\mathcal{L}}, \widehat{\mathcal{M}}, \widehat{\mathrm{eval}}, \widehat{R})$, so we can run it through the compiler-verifier of the previous section with auxiliary input $(\hat{p}, \hat{D})$ to get a program $\widehat{P}$ such that $\widehat{\mathrm{eval}}(\widehat{P}) \lhd \widehat{R}$. That means that on input $((\mathcal{L}, \mathcal{M}, \mathrm{eval}, R), x)$, if P produces a program M then $\mathrm{eval}(M) \lhd R$. This is the same property as we started with; we have successfully bootstrapped.

But there is a catch, of course, as we haven't dealt with Gödel yet. For the auxiliary input $(\hat{p}, \hat{D})$ to work, $\hat{p}$ should be a high level description of the operation of the proof checker, and $\hat{D}$ should be a proof in T that $\hat{p}$ is well formed and $\mathrm{eval}'(\hat{p}) \lhd \widehat{R}$. But that means that whenever $\hat{p}$ (or its compiled counterpart $\hat{P}$) receives a specification $(\mathcal{L}, \mathcal{M}, \mathrm{eval}, R)$ and a proof in T that a program meets its specification, then the program does meet its specification. This is impossible to prove in T, since it implies the consistency of T, so no such $\hat{D}$ exists.

2.2 Relative compiler verifiers

The solution is fairly simple, but it helps to set up some definitions to express it. Given $\mathcal{L}$, $\mathcal{M}$, eval, R as before, a relative compiler verifier for R is a program P such that given s, T, x, if $s \in \mathcal{L}$ and P terminates with $P(s, T, x) = M$, then $T \vdash M \in R_s$.

This is basically just what our compiler verifier was doing already, although the inputs have been shuffled about a bit. With minor modifications, we can use the same approach to make a relative compiler verifier. (Here s, T are "trusted inputs" and x is "untrusted", in the sense that the properties of the resulting program depend on s and T but not x.) Specifically, we take $K_T(s, (p, D))$ to be a function with auxiliary input (p, D) where $T \vdash D : p\ \mathrm{ok} \wedge \mathrm{eval}'(p) \lhd R$, and assume that $T \vdash \mathrm{eval}(C(p)) = \mathrm{eval}'(p)$ is proven separately as before (over some base theory T_0 and extended to T, failing if T is not an extension of T_0), then $T \vdash \mathrm{eval}(P) \lhd R$ and hence for any particular s and x with $K_T(s, x) = M$, reflecting them into the language of T, we have that $T \vdash M \in R_s$, that is, this is a relative compiler verifier for R.

But now we can say what we wanted to in the bootstrapper. Let the relation $\widehat{R}_{\hat{s}}$ hold of $\hat{s} := (\mathcal{L}, \mathcal{M}, \mathrm{eval}, R)$ and $P \in \widehat{\mathcal{M}}$ if for all $s \in \mathcal{L}$, and x, T, if $P(s, T, x) = M$ then $T \vdash M \in R_s$. Then we can plug $(\widehat{\mathcal{L}}, \widehat{\mathcal{M}}, \widehat{\mathrm{eval}}, \widehat{R})$ and $\hat{T} := T + \mathrm{Con}(T)$ into the relative compiler verifier K, provided we can find the right auxiliary, and get a program $\widehat{M}$ such that $\hat{T} \vdash \hat{M} \in \hat{R}_{\hat{s}}$, that is, whenever $s \in \mathcal{L}$ and x, T' are chosen, if $\hat{M}(s, T', x) = M$ then $T' \vdash M \in R_s$. Thus $\hat{M}$ will be a relative compiler verifier for R, assuming $\hat{T}$ is sound.

The auxiliary we need is a pair $(\hat{p}, \hat{D})$ such that $\hat{T} \vdash \hat{D} : \hat{p}\ \mathrm{ok} \wedge \mathrm{eval}'(\hat{p}) \lhd \hat{R}$. We let $\hat{p} = K_T$, and the first conjunct of $\hat{D}$ is straightforward. The second says that $\hat{T}$ proves that for all $\hat{s} := (\mathcal{L}, \mathcal{M}, \mathrm{eval}, R) \in \widehat{\mathcal{L}}$ and $\hat{x}$, if $K_T(\hat{s}, \hat{x}) = P$ then for all $s \in \mathcal{L}$ and x, T', if $P(s, T', x) = M$ then $T' \vdash M \in R_s$.

This last condition is just saying that P is a relative compiler verifier for R, except everything is relativized to $\hat{T}$.