Foundational Proof-Carrying Code

Presentation notes

Enríquez Ballester, Adrián Isasa Martín, Carlos Ignacio
Mata Aguilar, Luis Bak, Mieczyslaw

January 8, 2022

In 1996, a Gorge Necula's paper [8] introduces the idea of Proof-Carrying code:

1. **Code producer**: generates an executable together with a proof (certificate) that the program adheres to some safety policy.
2. **Code consumer**: receives some untrusted executable with its proof and can validate it before running.

The idea is not about cryptography, but about type systems for machine code (i.e. a deductive system defined over machine instructions which is proved to guarantee or preserve some properties).

Based on that work, Andrew W. Appel, known for being a major contributor of the StandardML compiler, starts its research in Foundational Proof-Carrying Code, which he defines as [3]:

A framework for mechanical verification of safety properties of machine language...

until here it is Proof-Carrying Code

...with the smallest possible runtime and verifier.

and this last part is the Foundational one.

The drawback that he sees for Proof-Carrying Code as initially described is that it is ad-hoc for each specific case, and built-in type rules and lemmas must be a trusted part because they are human-verified. Foundational Proof-Carrying Code relies on a primitive logic (e.g. high order with some arithmetic axioms) which is powerful enough to encode the required type system and lemmas. It means that they are instead proved in this foundational logic. With this, the aim is to make the trusted part as small as possible.

At that moment, they chose Twelf for defining the logic and the required encodings, this is just an example to illustrate how it looks like:

```
tp   : type.
tm   : tp -> type.
num  : tp.
pair : tp -> tp -> tp.
```

One of the parts that must be modeled within this logic is the target machine architecture. The behavior (i.e. semantic) and encoding (i.e. syntax) of machine instructions must be defined, and they believe that it is possible for every usual architecture in a similar way (i.e. as a step relation $(r, m) \mapsto (r', m')$ where r and r' are states of the register bank and m and m' of the memory). For example, they encoded the SPARK architecture by means of 1035 Twelf LOC for the syntactic part, generated with a 151 LOC of a higher level language due to redundancies, and 600 Twelf LOC for the semantic one. This is an example of an *add* instruction encoding:

$$\begin{aligned}&\mathsf{add}(i, j, k) = \\ &\quad \lambda r, m, r', m'.\ r'(i) = r(j) + r(k) \\ &\qquad\qquad \wedge (\forall x \neq i.\ r'(x) = r(x)) \\ &\qquad\qquad \wedge m' = m\end{aligned}$$

Safety requirements can be specified in the syntax and semantics themselves, by making the step relation deliberately partial or by making some syntax forbidden just by not defining it. This is a dumb example of the previous instruction to be not allowed for a certain register:

$$\begin{aligned}&\mathsf{add}(i, j, k) = \\ &\quad \lambda r, m, r', m'.\ r'(i) = r(j) + r(k) \\ &\qquad\qquad \wedge (\forall x \neq i.\ r'(x) = r(x)) \\ &\qquad\qquad \wedge m' = m \\ &\qquad\qquad \wedge i \neq 42\end{aligned}$$

Now, for proving the adherence to the safety requirements, an appropriate type system must be defined for the machine instructions and, as they follow a so called semantic approach, it requires the following to be encoded in the foundational logic as proofs, not as built-ins:

- Type judgements are assigned a truth value.
- If the premise judgements are true, then the conclusion judgement must be true.

- If a type judgement is true, then it corresponds to a safe state.

One of the most challenging parts has been to find an appropriate model for encoding type systems. Its first approach [1] was to model types as sets of values, and model values in a direct way like a pair consisting of the memory and a memory address, but they encounter some limitations:

- They were unable to model mutable fields.
- They were unable to model certain kinds of recursive datatype definitions.

Their second approach [2] was to model types as sets of pairs $\langle k, v\rangle$ where k is an approximation index and v a value. The judgement $\langle k, v\rangle \in \tau$ means informally that v can be considered to have type τ for a program running for less than k steps. This model solved their problem with recursion, but the one with mutable fields remained. A PhD thesis of a student of A.W. Appel offered later a model which solved also that problem.

A sad ending for this presentation

Ten years later, A.W. Appel says that now it is practical to prove safety and correctness with type systems for source code instead of machine code and they are trustworthy if compiled with a formally verified compiler [4], so he is now involved in projects of this kind (e.g. CertiCoq [5], CompCert [7], CertiKOS [6]).

However, although the results of this research seems to have not so much practical interest nowadays, we wanted to show how they encoded type systems in a foundational way, which is not only applicable to type systems for machine code as they show for example with a usual typed lambda calculus.

References

[1] Amy P Felty Andrew W Appel. A semantic model of types and machine instructions for proof-carrying code. *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 243–253, 2000.

[2] David McAllester Andrew W Appel. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23:657–683, 2001.

[3] Andrew W Appel. Foundational proof-carrying code. *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–256, 2001.

[4] Andrew W Appel. Proof-carrying code with correct compilers. -, 2009.

[5] `https://certicoq.org`.

[6] `http://flint.cs.yale.edu/certikos/`.

[7] `https://compcert.org`.

[8] George C Necula. Proof-carrying code. *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.