A Foundational Model for Types

Luis Mata Adrian Enríquez Carlos Ignacio Isasa Mieczyslaw Bak

January 17, 2022

Universidad Complutense de Madrid (UCM)

UNIVERSIDAD
COMPLUTENSE
MADRID

Agenda

- Foundational Proof Carrying Code
- Interlude
- A Foundational Model for Types
- Indexed Model for Types
- Conclusion

Foundational Proof Carrying Code

Foundational Proof Carrying Code (FPCC)

In 1996, a Gorge Necula's paper (Necula, 1997) introduces the idea of Proof-Carrying code:

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

Foundational Proof Carrying Code (FPCC)

The idea is about type systems for machine code (i.e. a deductive system defined over machine instructions which is proved to guarantee or preserve some properties).

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.

(Appel, 2001)

Foundational Proof Carrying Code (FPCC)

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:

Foundational Proof Carrying Code (FPCC)

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).

Foundational Proof Carrying Code (FPCC)

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:

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

Foundational Proof Carrying Code (FPCC)

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:

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

Foundational Proof Carrying Code (FPCC)

One of the most challenging parts has been to find an appropriate model for encoding type systems. Its first approach (A. P. F. Andrew W Appel, 2000) 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.

Foundational Proof Carrying Code (FPCC)

Their second approach (D. M. Andrew W Appel, 2001) 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.

Interlude

Interlude

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 (Appel, 2009), so he is now involved in projects of this kind (e.g. CertiCoq (n.d.-a), CompCert (n.d.-b), CertiKOS (n.d.-c)).

Interlude

However, although the results of this research seem to have not so much practical interest nowadays, we want 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.

A Foundational Model for Types

A Foundational Model for Types

The first key insight of FPCC -and of PCC in reality- is realizing that you can write type-inference rules for machine language and states, for example:

$$\frac{m \vdash x : \tau_1 \times \tau_2}{m \vdash m(x) : \tau_1 \wedge m(x+1) : \tau_2}$$

which means that if x has type $\tau_1 \times \tau_2$ -meaning a pointer to a pair of those types-, then the contents of direction x will be of type τ_1 and the contents of direction $x+1$ will be of type τ_2. We can even add safety policies into the mix:

$$\frac{m \vdash x : \tau_1 \times \tau_2}{\text{readable}(x) \wedge \text{readable}(x+1) \wedge m \vdash m(x) : \tau_1 \wedge m(x+1) : \tau_2}$$

A Foundational Model for Types

The second key insight of FPCC -which really makes it foundational-, is realizing that instead of looking at this from a syntactical standpoint we can do it from a semantical standpoint. Viewing things from a syntactic standpoint is significantly harder as they only allow subject-reduction over a set of axiomatic syntactic rules.

Using semantics, however, allows us to have these expressions as lemmas provable from their syntactic meaning instead of axioms. Let us define $m \vdash x : \tau$ as an application of predicate τ on memory m and integer (or address) x. So $m \vdash x : \tau \equiv \tau(m)(x)$.

A Foundational Model for Types

To note the change from syntactic semantic we shall note the types as $\models x : \tau$ We can now give the semantic meaning to the rules written before of:

$$\begin{aligned}\mathsf{record}(\tau_1, \tau_2)mx = & \\ & \mathsf{readable}(x) \wedge \mathsf{readable}(x+1) \\ & \wedge \models m(x) : \tau_1, m(x+1) : \tau_2\end{aligned}$$

Which not only allows us to prove the expressions that we've already seen but also:

$$\frac{\mathsf{readable}(x) \qquad \mathsf{readable}(x+1) \qquad \models m(x) : \tau_1 \qquad \models m(x+1) : \tau_2}{\models x : \tau_1 \times \tau_2}$$

Indexed Model for Types

Indexed Model for Types

The type system for FPCC that we have discussed has some issues, mainly in dealing with recursion. In order to solve them, we need a new definition for type that lets us come with more advanced proofs.

A type is modeled as a set of pairs of the form $\langle k, v \rangle$ where k is a nonnegative integer called index, v is a value and it is closed under decreasing index (i.e.whenever $\langle k, v \rangle$ belongs to a type τ, we also have that $\langle j, v \rangle$ belongs to τ for all $j \geq k$).

We write $e :_k \tau$ if $e \rightarrow^j v$ for some $j < k$ implies $\langle k - j, v \rangle \in \tau$, where $\rightarrow$ is the step relation given by the corresponding small step semantics.

Indexed Model for Types

In each case, the type sets must be defined, but some conventional ones which do not depend on a specific use case are for example:

$$\bot \equiv \{\}$$
$$\top \equiv \{\langle k, v\rangle \mid k \geq 0\}$$

Type environment and states are modeled respectively as mappings from variables to types and variables to values. The consistency of a state σ and environment Γ is written $\sigma :_k \Gamma$ and defined as $\forall x \in dom(\Gamma)$ we have $\sigma(x) :_k \Gamma(x)$.

Indexed Model for Types

The entailment relation $\Gamma \models_k e$ is the semantic counterpart of a type judgement and means $\sigma(e) :_k \alpha$ for all state σ consistent with Γ, where $\sigma(e)$ is the result of replacing all the free variables of e with their values i n σ.

Also, $\Gamma \models e : \alpha$ means $\Gamma \models e :_k \alpha$ for all $k \geq 0$.

Conclusion

Conclusion

- Foundational Proof Carrying Code seems to be out of interest nowadays, but it motivated a research about modeling type systems in a foundational way.
- To model types directly as sets of values was not enough for some features such as contravariant recursive definitions and mutable fields.

Conclusion

- A more sophisticated model, which we are presenting here, allowed them to model better recursion, and further research seems to have solved even the problem they had about mutable fields.
- These results may help in studying type systems from the point of view of foundational mathematics and also where the introduction of rules must be correct within a logic in a machine-checkable way.

References i

(n.d.-a). https://certicoq.org.

(n.d.-b). https://compcert.org.

(n.d.-c). http://flint.cs.yale.edu/certikos/.

Andrew W Appel, A. P. F. (2000). 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*, 243–253.

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

References ii

Appel, A. W. (2001). Foundational proof-carrying code. *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, 247–256.

Appel, A. W. (2009). Proof-carrying code with correct compilers. -.

Necula, G. C. (1997). Proof-carrying code. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 106–119.