# A Foundational Model for Types

Luis Mata  Adrian Enríquez  Carlos Ignacio Isasa

January 17, 2022

Universidad Complutense de Madrid (UCM)


UNIVERSIDAD
COMPLUTENSE
MADRID

# Agenda

- Foundational Proof-Carrying Code
- A Foundational Model for Types
- Conclusions

# Foundational Proof-Carrying Code

# Proof-Carrying Code

1. **Code producer**: generates executable + proof that of adherance to a safety policy
2. **Code consumer**: validates the executable and the proof before running it.

Approached via type systems for machine code, not cryptography.

(Necula, 1997)

# Foundational Proof-Carrying Code

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

until here it's still Proof-Carrying Code

*...with the smallest possible runtime and verifier.*

and this last part is the Foundational one.

(Appel, 2001)

# Foundational Proof-Carrying Code

**How?**

- Choose a foundational logic (e.g. higher-order + arithmetic).
- Model the machine architecture (e.g. a step relation $(r, m) \mapsto (r', m')$ between memory and register bank states).
- Model the security requirements (e.g. by making the step relation deliberately partial).
- Model a type system (e.g. values as memory data and types as sets of values).

# Foundational Proof-Carrying Code: Logic

They chose higher+order logic with some arithmetic axioms, and implement the specifications using the Twelf language.

# Foundational Proof-Carrying Code: Architecture

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: Safety policy

Example of safety requirement specified in the semantics of the previous instruction itself:

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

# Foundational Proof-Carrying Code: Bad news

Nowadays this seems to have evolved to 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) (e.g. CertiCoq (n.d.-a), CompCert (n.d.-b), CertiKOS (n.d.-c)).

# Foundational Proof-Carrying Code: Type systems

However, this motivated a research about encoding 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

# Fst. attempt

Values are defined in a straightforward way for the use case, and types are sets of values.

We are going to see a machine instruction example where values are memory data and adresses.

# Fst. attempt: Machine instructions PCC example

Pair projections:

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

Adding safety policies into the mix:

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

But this is PCC yet.

# Fst. attempt: Machine instructions FPCC example

Let us define $m \vdash x : \tau$ as an application of predicate $\tau$ on memory $m$ and integer $x$:

$$m \vdash x : \tau \equiv \tau(m)(x)$$

We write $\models_m x : \tau$ to note the change from a syntactic viewpoint to a semantic one.

# Fst. attempt: Machine instructions FPCC example

The previous rule defined as

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

stands for

$$\begin{aligned} \text{pair}(\tau_1, \tau_2)(m)(x) = \ & \text{readable}(x) \\ & \wedge \text{readable}(x+1) \\ & \wedge \tau_1(m)(x) \\ & \wedge \tau_2(m)(x+1) \end{aligned}$$

# Fst. attempt: Results

Achievements:

- Usual type system features.
- Datatype declarations.
- Function types.

But they had trouble in encoding some highly desirable features:

- Recursive datatype definitions where the recursion is in contravariant position.
- Mutable fields.

# Snd. attempt

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 $\tau$, we also have that $\langle j, v \rangle$ belongs to $\tau$ 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.

# Snd. attempt

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 $\sigma$ and environment $\Gamma$ is written $\sigma :_k \Gamma$ and defined as $\forall x \in \mathit{dom}(\Gamma)$ we have $\sigma(x) :_k \Gamma(x)$.

# Snd. attempt

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

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

# Snd. attempt: Lambda calculus example

# Conclusions

# Conclusions

Foundational Proof Carrying Code, although out of interest nowadays, motivated a research about modeling type systems in a foundational way.

# Conclusions

Their 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.

# Conclusion

After several attempts, they modeled succesfully some basic and advanced type system features:

- Usual type systems of functional languages.
- Usual type systems of imperative languages.
- Specific type systems for machine instructions.
- Function types (e.g. first-class, continuations and closures).
- Universal and existential quantification.
- Type equalities ($\tau_1 \sim \tau_2$).

# References i

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

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

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

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.