A Foundational Model for Types

Luis Mata Carlos Ignacio Isasa Adrian Enríquez Mieczyslaw Bak

January 18, 2022

Universidad Complutense de Madrid (UCM)

UNIVERSIDAD
COMPLUTENSE
MADRID

Agenda

- Foundational Proof-Carrying Code
- A Foundational Model for Types: fst. attempt
- A Foundational Model for Types: snd. attempt
- Conclusions

Foundational Proof-Carrying Code

Proof-Carrying Code

1. **Code producer**: generates executable + proof of adherence 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

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).

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).

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).

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:

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

Foundational Proof-Carrying Code: Safety policy

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

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

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

Fst. attempt

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

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 addresses.

Fst. attempt: Machine instructions PCC example

Pair projections:

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

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:

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

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 τ on memory m and integer x:

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

Fst. attempt: Machine instructions FPCC example

Let us define $m \vdash x : \tau$ as an application of predicate τ 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: Machine instructions FPCC example

They keep adding ingredients according to their needs, for example, adding a heap allocation state to the relation:

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

Fst. attempt: Machine instructions FPCC example

And a predicate to check if a type is consistent according to heap allocation:

$$\begin{aligned}\mathsf{valid}(\tau) = \; & \forall a, a', m, v.\ (a \subset a') \implies \tau(a, m)(v) \implies \tau(a', m)(v) \\ & \wedge \forall a, m, m', v.\ (\forall x\ m(x) = m'(x)) \implies \tau(a, m)(v) \implies \tau(a, m')(v)\end{aligned}$$

Fst. attempt: Results

Achievements:

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

Fst. attempt: Results

Achievements:

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

But they had trouble with encoding some highly desirable features:

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

A Foundational Model for Types: snd. attempt

Snd. attempt

A type is modeled as a set of pairs of index and value of the form $\langle k, v \rangle$.

Snd. attempt

A type is modeled as a set of pairs of index and value of the form $\langle k, v \rangle$.

It has to be closed under decreasing index:

$$\langle k, v \rangle \in \tau \implies \langle j, v \rangle \in \tau \; \forall j \leq k$$

Snd. attempt

Type sets defined for each use case, but here are some conventional examples:

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

Snd. attempt

Type sets defined for each use case, but here are some conventional examples:

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

Type environment and states are modeled respectively as mappings from variables to types and variables to values. $\sigma :_k \Gamma$ denotes consistency:

$$\sigma(x) :_k \Gamma(x)\ \forall x \in dom(\Gamma)$$

Snd. attempt

We write $e :_k \tau$ if $e \to^j e'$ for some $j < k$ and e' which do not admit more steps implies $\langle k - j, e' \rangle \in \tau$, where $\to$ is the step relation given by the corresponding small step semantics.

Snd. attempt

We write $e :_k \tau$ if $e \to^j e'$ for some $j < k$ and e' which do not admit more steps implies $\langle k - j, e' \rangle \in \tau$, where $\to$ is the step relation given by the corresponding small step semantics.

With this, the entailment relation $\Gamma \models_k e : \alpha$ is the semantic counterpart of a type judgement and means

$$\sigma(e) :_k \alpha \; \forall \sigma \text{ s.t. } \sigma :_k \Gamma$$

Also, $\Gamma \models e : \alpha$ means that the relation is satisfied for all $k \geq 0$.

Snd. attempt: Lambda calculus example

$$e ::= x \mid 0 \mid \langle e_1, e_2 \rangle \mid \pi_1(e) \mid \pi_2(e) \mid \lambda x.e \mid e_1\ e_2$$

Snd. attempt: Lambda calculus example

Values: 0, closed abstraction terms and pairs of values.

Snd. attempt: Lambda calculus example

Values: 0, closed abstraction terms and pairs of values.

Small step semantics: standard.

Snd. attempt: Lambda calculus example

Values: 0, closed abstraction terms and pairs of values.

Small step semantics: standard.

Safeness: to not reach a stuck term, trivially modeled from the definition of $\models e : \alpha$.

Snd. attempt: Lambda calculus example

Type sets:

$$
\begin{aligned}
\text{int} &\equiv \{\langle k, 0\rangle \mid \forall k.\ k \geq 0\} \\
\tau_1 \times \tau_2 &\equiv \{\langle k, \langle v_1, v_2\rangle\rangle \mid \forall j.\ j < k \wedge \langle j, v_1\rangle \in \tau_1 \wedge \langle j, v_2\rangle \in \tau_2\} \\
\alpha \to \tau &\equiv \{\langle k, \lambda x.e\rangle \mid \forall j.\ j < k \wedge \langle j, v\rangle \in \alpha \implies e[v/x] :_j \tau\} \\
\mu F &\equiv \{\langle k, v\rangle \mid \langle k, v\rangle \in F^{k+1}(\bot)\}
\end{aligned}
$$

Snd. attempt: Lambda calculus example

Type sets:

$$
\begin{aligned}
\text{int} &\equiv \{\langle k, 0\rangle \mid \forall k.\ k \geq 0\} \\
\tau_1 \times \tau_2 &\equiv \{\langle k, \langle v_1, v_2\rangle\rangle \mid \forall j.\ j < k \wedge \langle j, v_1\rangle \in \tau_1 \wedge \langle j, v_2\rangle \in \tau_2\} \\
\alpha \to \tau &\equiv \{\langle k, \lambda x.e\rangle \mid \forall j.\ j < k \wedge \langle j, v\rangle \in \alpha \implies e[v/x] :_j \tau\} \\
\mu F &\equiv \{\langle k, v\rangle \mid \langle k, v\rangle \in F^{k+1}(\bot)\}
\end{aligned}
$$

They are closed under decreasing index by definition.

Snd. attempt: Lambda calculus example

Type inference lemmas:

$$\Gamma \models x : \Gamma(x) \qquad \Gamma \models 0 : \text{int}$$

$$\frac{\Gamma \models f : \alpha \to \beta \quad \Gamma \models e : \alpha}{\Gamma \models f\, e : \beta} \qquad \frac{\Gamma[x := \alpha] \models e : \beta}{\Gamma \models \lambda x.e : \alpha \to \beta}$$

$$\frac{\Gamma \models e_1 : \alpha \quad \Gamma e_2 : \beta}{\Gamma \models \langle e_1, e_2 \rangle : \alpha \times \beta}$$

$$\frac{\Gamma \models e : \alpha \times \beta}{\Gamma \models \pi_1(e) : \alpha} \qquad \frac{\Gamma \models e : \alpha \times \beta}{\Gamma \models \pi_2(e) : \beta}$$

$$\frac{\Gamma \models e : \mu F}{\Gamma \models e : F(\mu F)} \qquad \frac{\Gamma \models e : F(\mu F)}{\Gamma e : \mu F}$$

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 successfully 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.