A foundational model for types

Presentation notes

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

January 18, 2022

Foundational Proof-Carrying Code

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:

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

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:

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

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

- If the premise judgements are true, then the conclusion judgement must be true.

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.

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 [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 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

We are going to show incrementally an example for machine instructions with the reasoning of their initial attempt to model types within a foundational logic.

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 a pair projection:

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

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:

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

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 $m \vdash x : \tau \equiv \tau(m)(x)$. To note the change from syntactic to semantic we shall note the type judgements as $\models_m x : \tau$. We can now give the semantic meaning to the rules written before of:

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

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

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

However, all of this poses a problem, what do we do if the memory changes? Let us suppose an initial state with $\models_m r_1 : \mathsf{int} \times \mathsf{int}, r_3 : \mathsf{int}$ and a set of instructions:

$$\begin{aligned} & \mathsf{m}(r_2) \leftarrow r_3 \\ & \mathsf{m}(r_2 + 1) \leftarrow r_3 \end{aligned}$$

How can we prove that $\models r_1 :_{m'} \mathsf{int} \times \mathsf{int}$ when the fact was stated in the initial state of the memory but it has changed? We can certainly prove theorems of the form:

$$\frac{\models_m v : \mathsf{int} \times \mathsf{int} \qquad \mathsf{upd}(m, x, y, m') \qquad x \neq v \qquad x \neq v + 1}{\models_{m'} v : \mathsf{int} \times \mathsf{int}}$$

But how can we prove $x \neq v$ and $x \neq v + 1$ in a simple way? The authors of [1] came up with using the heap allocation pointer a as a way to check if types are maintained. Giving the pair example again, we shall define it as:

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

Now, using as an example the register r_6, we can define what being in a is:

$$\mathsf{stda}(r, m)(v) = v < r_6$$

As the heap expands, new values are stored without deleting the old ones. In order to check if the registers are kept within this type we must define a function to check if they are valid (i.e. consistent):

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

The first part of the definition means that the type is invariant under increases of the allocation set while the second part is that it is invariant under the storing of new unallocated variables. With these new concepts, going back to our examples we can assert that the register r_1 maintains the typing $\mathsf{int} \times \mathsf{int}$ as long as that type is valid.

With this kind of reasoning, apart from defining specific typing rules for machine instructions as in the example, they were also able to model the usual features of a type system like type constructors, function types and a restricted version of recursive datatypes.

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 $j \leq k$).

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

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 $\sigma(x) :_k \Gamma(x)$.

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

Lambda calculus example

As an example of type system modeled in this way, they consider a lambda calculus with pairs and the constant 0:

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

Values are the constant 0, a closed abstraction $\lambda x.e$ (i.e. no free variables) and a pair of values $\langle v_1, v_2\rangle$.

For the semantics, they consider a pretty standard small step and safeness is to not reach a stuck term.

The defined type sets which they define for this use case are, together with $\bot$ and $\top$:

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

The relation $\models e : \alpha$ models safety trivially from its definition, and the rules which correspond to the axioms of variables and the constant 0 can also be trivially proved from the definitions of consistency and the type set int.

The same happens in order to prove that the type sets product and abstraction are also valid types (i.e. they satisfy the closed under decreasing index property).

With that, the remaining rules can also be proved. For example, the idea of the proof for the application rule consists on the fact that, for a given k, the left subterm either does not reach a normal form yet or it reduces to a lambda expression. By checking then the result of beta reducing it and the corresponding type set definitions, the rule follows.

(Show a slide with the typing rules and comment that they are written with their entailment relation syntax but in fact they are defined within the logic they use).

This new approach allows them to model more advanced features such as quantified types and type equalities.

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.

In that research, to model types directly as sets of values was not enough for some features such as contravariant recursive definitions and mutable fields.

A more sophisticated model allowed them to better model 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.

This is a list of some of the features successfully modeled within their research:

- Usual features of a functional language.
- Usual features of an imperative and OO language.
- Specific type systems for machine instructions.
- Recursive datatype definitions.
- Function types (i.e. first-class functions, continuations and closures).
- Universal and existential quantification.
- Type equality (i.e. $\tau_1 \sim \tau_2$).

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.