A Syntactic Approach to Foundational Proof-Carrying Code*

Nadeem Abdul Hamid
Advisor: Zhong Shao

January 24, 2002 (OGST)

* Joint work with Zhong Shao, Valery Trifonov, Stefan Monnier, Zhaozhong Ni. Submitted to LICS 2002.
Conventional Compilation

- C Program
- C Compiler
- Machine Code
- CPU

Safe
Type-preserving Compilation

Java/ML Program

Type-preserving Compiler

Typed Assembly/Byte-code

Verifier

Trusted type system

Trusted Assembler/JIT Compiler

Machine Code

Machine Code
Proof-Carrying Code (PCC)

Source Program

Verification Condition Generator

Compiled machine code

Type annotations

Type-preserving Compiler

Safety Predicate

Prover

Safety Proof

Machine Code

PCC Binary

Proof Checker

Safety Policy

OK

Compiled machine code

Type annotations

[Necula, Lee 1996; Necula 1997]
PCC: Advantages

✓ Proofs directly on machine code
  – No further compilation

✓ Fully automatic proof construction
  – Sufficiently simply safety policy
  – Type-safe source language
Original PCC: Shortcomings

● Logic: Built-in type system
   *(Soundness of the logic proved by hand as metatheorem)*

![Logical expressions and typing rules]

😊 Potential bugs
   – VCGen (relatively large program)
   – Typing rules  [League, Shao, Trifonov 2001]
   – Proof checker
Foundational Proof-Carrying Code (FPCC)

- Build proofs on foundations of mathematical logic
  - Model/encode/express type system in terms of higher-order logic + axioms of arithmetic

- More Flexible
  - Not specialized to any type system

- More Secure
  - Only trust the verification system for higher-order logic (very small program)

- More difficult to construct

FPCC Overview

FPCC Assembler

Typed Assembly Code

FPCC Binary

Safety Proof

Machine Code

Encoded Type System
Safety Policy

Proof Checker

OK
Original FPCC: “Semantic” Approach

- Powerful
  - Sophisticated semantic models of types

- Complex
  - Indexed model of recursive types
  - Hierarchy of Godel numberings

- Difficult to extend to new type system features
  - New features require re-working the entire framework

**FPCC: Syntactic Approach**

- **FPCC Logic**
  - Machine encoding
  - Safety policy specification

- **Typed Assembly Language (FTAL)**

- **Compiling FTAL ➔ FPCC**
FPCC Logic

- Expressive enough to encode machine semantics, safety policies

- The Calculus of Inductive Constructions (CiC)

\[ A, B ::= \text{Set} | \text{Type} | X | \lambda X : A. B | A \ B | \Pi X : A. B \]

+ inductive definitions

- Supported by Coq Proof Assistant
Logic

- Propositions: $P, Q, ...$
- Implication, conjunction, disjunction, etc.:
  - $P \Rightarrow Q, \quad P \land Q, \quad P \lor Q$
- Predicates, quantifiers: $\forall x. P(x)$
- Inference Rules

Lambda Calculus

- Types: $A, B ::= A | A \rightarrow B$
- Terms: $M, N ::= x | \lambda x : A . M | (M N)$
Curry-Howard Isomorphism

Propositions $\iff$ Types

Proofs $\iff$ Terms

<table>
<thead>
<tr>
<th>Logic</th>
<th>Lambda Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>assumption</td>
<td>term variable</td>
</tr>
<tr>
<td>proof</td>
<td>term</td>
</tr>
<tr>
<td>propositional variable</td>
<td>type variable</td>
</tr>
<tr>
<td>formula/proposition</td>
<td>type</td>
</tr>
<tr>
<td>connective</td>
<td>type constructor</td>
</tr>
<tr>
<td>proof tree with redundancy</td>
<td>redex</td>
</tr>
<tr>
<td>normalization</td>
<td>reduction</td>
</tr>
</tbody>
</table>

Consider *modus ponens*:

\[
P \quad P \rightarrow Q
\]

\[
\quad Q
\]

\[
N : A \quad M : A \rightarrow B
\]

\[
(M \ N) \ : B
\]
CiC: Types (Propositions)

\[ A, B ::= \text{Set} \mid \text{Type} \mid X \mid \lambda X:A.\ B \mid A \ B \mid \Pi X:A.\ B \]

- \( \Pi X:A.\ B \) (or, \( A \rightarrow B \) ) read in two ways:
  - \( \Pi n:\text{nat}.\ (\text{lt}\ n\ n+1) \) : “for all” quantifier
  - \( (x=y)\rightarrow(y=x) \) : “implies”

Natural numbers are an inductive definition:

Inductive \( \text{nat} : \text{Set} \) := \( O : \text{nat} \mid S : \text{nat} \rightarrow \text{nat} \).

Less-than (proposition), also:

Inductive \( \text{lt} : \text{nat} \rightarrow \text{nat} \rightarrow \text{Set} \) := \( \text{lt}_o : \Pi n:\text{nat}.\ (\text{lt}\ n\ (S\ n)) \)
| \( \text{lt}_s : \Pi n,m:\text{nat}.\ (\text{lt}\ n\ m) \rightarrow (\text{lt}\ n\ (S\ m)). \)

\( \forall n.\ n < (n+1) \)
\( \forall n,m.\ (n < m) \rightarrow (n < (m+1)) \)
A proof of the proposition \((\text{lt } n \ m)\) is built using the appropriate **constructors**

- *E.g.*

\[
(\text{lt}_S \ O \ (S \ O) \ (\text{lt}_n \ O)) : (\text{lt} \ O \ (S \ (S \ O))
\]

A proof of implication \((A \rightarrow B)\) is a “function”, \(\lambda x:A. M\).

CiC logic provides elimination over inductive types
CiC: Inductive Elimination

- Fix-point (recursion) + Case analysis

Lemma lem1 : \( \forall n: \text{nat}. (\text{lt} \; O \; (S \; n)) \)
\( := \lambda n: \text{nat}. \)
\[ \text{Cases } n \text{ of } O \Rightarrow (\text{lt}_n \; O) \]
\[ | (S \; m) \Rightarrow (\text{lt}_S \; O \; (S \; m) \; (\text{lem1} \; m)) \]
end.

- Constraints on inductive definitions keep the logic consistent.
Our Idealized Machine

- Simple instruction set
- 32 Registers
- Infinite word-addressed memory of infinite-size words

Machine state
  - Memory + Registers + Program Counter
Memory, Registers, etc.

\[ \overline{r} \in \text{Regnum} = \{ \overline{r}_0, \overline{r}_1, \ldots \overline{r}_{31} \} \]
\[ w, pc \in \text{Word} = \{ 0, 1, \ldots \} \]
\[ M \in \text{Mem} = \text{Word} \rightarrow \text{Word} \]
\[ \overline{R} \in \text{Regfile} = \text{Regnum} \rightarrow \text{Word} \]
\[ S \in \text{State} = \text{Mem} \times \text{Regfile} \times \text{Word} \]

\[ \text{Instr} \ni \nu ::= \text{add} \ \overline{r}_d, \overline{r}_s, \overline{r}_t | \text{addi} \ \overline{r}_d, \overline{r}_s, w | \text{movi} \ \overline{r}_d, w \]
\[ | \text{bgt} \ \overline{r}_s, \overline{r}_t, w | \text{jd} \ w | \text{jmp} \ \overline{r} \]
\[ | \text{ld} \ \overline{r}_d, \overline{r}_s(w) | \text{st} \ \overline{r}_d(w), \overline{r}_s | \text{illegal} \]

+ Decoding function \((Dc : \text{Word} \rightarrow \text{Instr})\)
+ Step function \((\text{Step} : \text{State} \rightarrow \text{State})\)
## Step function

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if Dc(M(pc)) =</strong></td>
<td>then Step(M, R, pc) =</td>
</tr>
<tr>
<td><strong>add ( \overline{r_d}, \overline{r_s}, \overline{r_t} )</strong></td>
<td>((M, R{\overline{r_d} \mapsto \overline{R}(\overline{r_s}) + \overline{R}(\overline{r_t})}, pc+1))</td>
</tr>
<tr>
<td><strong>addi ( \overline{r_d}, \overline{r_s}, w )</strong></td>
<td>((M, R{\overline{r_d} \mapsto \overline{R}(\overline{r_s}) + w}, pc+1))</td>
</tr>
<tr>
<td><strong>movi ( \overline{r_d}, w )</strong></td>
<td>((M, R{\overline{r_d} \mapsto w}, pc+1))</td>
</tr>
</tbody>
</table>
| **bgt \( \overline{r_s}, \overline{r_t}, w \)** | \((M, R, pc+1), \text{ when } \overline{R}(\overline{r_s}) \leq \overline{R}(\overline{r_t})\)  
\((M, R, w), \text{ when } \overline{R}(\overline{r_s}) > \overline{R}(\overline{r_t})\) |
| **jd \( w \)** | \((M, R, w)\) |
| **jmp \( \overline{r} \)** | \((M, R, \overline{R}(\overline{r}))\) |
| **ld \( \overline{r_d}, \overline{r_s}(w) \)** | \((M, R\{\overline{r_d} \mapsto M(\overline{R}(\overline{r_s}) + w)\}, pc+1)\) |
| **st \( \overline{r_d}(w), \overline{r_s} \)** | \((M\{\overline{R}(\overline{r_d}) + w \mapsto \overline{R}(\overline{r_s})\}, R, pc+1)\) |
| **illegar** | \((M, R, pc)\) |
Safety Policy

- Program counter must point to a legal instruction in memory (i.e. not decode to an illegal instruction).

\[ SP \ (M, R, pc) = (Dc \ (M \ (pc)) \neq \text{illegal}) \]

- Safety = For any number of steps, starting from a state satisfying the safety policy, the safety policy must still be satisfied.

\[ \text{Safe} \ (S) = \Pi n : \text{Nat.} \ SP \ \ (\text{Step}^n \ (S)) \]
FPCC Code Package

- Initial machine state (includes program code)
- Safety proof

\[ F = (S_0 : State, A : Safe (S_0)) \]
Generating the safety proof

- Define a “global invariant” (induction hypothesis) and prove properties:

  Initial Condition: \( \text{Inv} (S_0) \)

  Preservation: \( \Pi S : \text{State}. \text{Inv} (S) \rightarrow \text{Inv} (\text{Step} (S)) \)

  Progress: \( \Pi S : \text{State}. \text{Inv} (S) \rightarrow \text{SP} (S) \)

\[ \Pi n : \text{Nat}. \text{Inv} (\text{Step}^n (S_0)) \]

\[ \text{Safe} (S) = \Pi n : \text{Nat}. \text{SP} (\text{Step}^n (S)) \]
Featherweight Typed Assembly Language (FTAL)

- Critical for defining the global invariant
- “Featherweight” …
  - No polymorphism, existentials
- … But realistic
  - Recursive types
  - Memory allocation
  - Mutable records
- Similar to TAL [Morrisett, et al. 1998] but novel
FTAL: Syntax

(word val) \( v ::= l \mid i \mid ?\tau \mid \text{fold } v \text{ as } \tau \)

(heap val) \( h ::= \langle v_1, \ldots, v_n \rangle \mid \text{code}[\Gamma.I] \)

(heap) \( H ::= \{0 \mapsto h_0, \ldots, n \mapsto h_n\} \)

(regfile) \( R ::= \{r0 \mapsto v_0, \ldots, r31 \mapsto v_{31}\} \)

(instr) \( i ::= \text{add } r_d, r_s, r_t \mid \text{addi } r_d, r_s, i \)
\( \mid \text{alloc } r_d[\overline{\tau}] \mid \text{bgt } r_s, r_t, l \mid \text{bump } i \)
\( \mid \text{fold } r_d[\overline{\tau}], r_s \mid \text{ld } r_d, r_s(i) \mid \text{mov } r_d, r_s \)
\( \mid \text{movi } r_d, i \mid \text{movl } r_d, l \mid \text{st } r_d(i), r_s \)
\( \mid \text{unfold } r_d, r_s \)

(instr seq) \( I ::= i; I \mid \text{jd } l \mid \text{jmp } r \)

(program) \( P ::= (H, R, I) \)
FTAL: Types

\[(\text{type}) \quad \tau ::= \alpha \mid \text{int} \mid \forall[].\Gamma \mid \langle\tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n}\rangle \mid \mu\alpha.\tau\]

\[(\text{init flag}) \quad \varphi ::= 0 \mid 1\]

\[(\text{heap ty}) \quad \Psi ::= \{0: \tau_0, \ldots, n: \tau_n\}\]

\[(\text{alloc pt ty}) \quad \rho ::= \text{fresh} \mid \text{used}(n)\]

\[(\text{regfile ty}) \quad \Gamma ::= \{r_0: \tau_0, \ldots, r_n: \tau_n, r31: \rho\}\]
Alloc/Bump Instructions

- `malloc` (in TAL) is a “macro instruction”
  - Expanded in assembly to machine code
  - Breaks one-to-one correspondence between FTAL and machine, needed for the invariant
  - So we decompose it into `alloc` / `bump` and develop typing rules with special register:

\[
\frac{
\Psi; \Gamma\{r_d: \langle \tau_1^0, \ldots, \tau_n^0 \rangle \}\{r31: \text{used}(n)\} \vdash I}{
\Psi; \Gamma\{r31: \text{fresh}\} \vdash \text{alloc } r_d[\tau_1, \ldots, \tau_n]; I}
\text{(ALLOC)}
\]

\[
\frac{
\Psi; \Gamma\{r31: \text{fresh}\} \vdash I}{
\Psi; \Gamma\{r31: \text{used}(n)\} \vdash \text{bump } n; I}
\text{(BUMP)}
\]
FTAL Soundness

- Encode the complete syntax, typing rules in CiC (Coq) and formally prove preservation and progress.

Theorem 1 (Progress)
If $\vdash P$, then there exists $P'$ such that $P \leftrightarrow P'$.

Theorem 2 (Preservation)
If $\vdash P$ and $P \leftrightarrow P'$, then $\vdash P'$.

*** The entire proof is mechanically checkable ***
and built solely on foundational CiC logic
Not there yet ...

- Need to have a safety proof for machine state, not for FTAL programs
- Recall, we need:

  Initial Condition: \( \text{Inv} (S_0) \)
  Preservation: \( \forall S : \text{State}. \text{Inv} (S) \rightarrow \text{Inv} (\text{Step} (S)) \)
  Progress: \( \forall S : \text{State}. \text{Inv} (S) \rightarrow \text{SP} (S) \)

- So, now we define the invariant as:

\[
\text{Inv}(S) = \exists P : \text{program}. \exists D : (\vdash P). P \Rightarrow S
\]

“FTAL program, \( P \), translates to machine state, \( S \)”
FTAL \Rightarrow \text{Machine State Translation}

\textbullet \textit{Layout} function translates FTAL labels to memory addresses
\hspace{1cm} – flattens the FTAL heap

\begin{align*}
\text{Layout}(H) \vdash H &\Rightarrow M \\
\text{Layout}(H) \vdash R &\Rightarrow \overline{R} \\
\text{Layout}(H) \vdash I &\Rightarrow M[pc..j] \\
\hline
(H, R, I) &\Rightarrow (M, \overline{R}, pc)
\end{align*}
# Translation Judgments

<table>
<thead>
<tr>
<th>Relation</th>
<th>Correspondence</th>
</tr>
</thead>
<tbody>
<tr>
<td>((H, R, I) \Rightarrow (M, \overline{R}, pc))</td>
<td>FTAL program to machine state</td>
</tr>
<tr>
<td>(L \vdash H \Rightarrow M)</td>
<td>FTAL heap to memory</td>
</tr>
<tr>
<td>(L \vdash R \Rightarrow \overline{R})</td>
<td>register files</td>
</tr>
<tr>
<td>(L \vdash I \Rightarrow_{s} M[i..j])</td>
<td>sequence of instructions to memory layout</td>
</tr>
<tr>
<td>(L \vdash i \Rightarrow_{i} w)</td>
<td>instruction translation</td>
</tr>
<tr>
<td>(L \vdash h \Rightarrow_{h} M[i..j])</td>
<td>heap value to memory layout</td>
</tr>
<tr>
<td>(L \vdash v \Rightarrow_{w} w)</td>
<td>word value to machine word</td>
</tr>
</tbody>
</table>

**Word values:**

\[ L \vdash l \Rightarrow_{w} L(l) \quad L \vdash i \Rightarrow_{w} i \]

\[
\text{for any } w \\
L \vdash ?\tau \Rightarrow_{w} w
\]

\[
L \vdash v \Rightarrow_{w} w \\
L \vdash \text{fold } v \text{ as } \tau \Rightarrow_{w} w
\]
Translating Instructions

\[
L \vdash \text{add } r_d, r_s, r_t \Rightarrow_i \text{add } \overline{r_d}, \overline{r_s}, \overline{r_t}
\]

\[
L \vdash \text{addi } r_d, r_s, i \Rightarrow_i \text{addi } \overline{r_d}, \overline{r_s}, i
\]

\[
L \vdash \text{alloc } r_d[\overline{r}] \Rightarrow_i \text{addi } \overline{r_d}, \overline{r_{31}}, 0
\]

\[
L \vdash \text{bump } i \Rightarrow_i \text{addi } \overline{r_{31}}, \overline{r_{31}}, i
\]

\[
L \vdash \text{fold } r_d[\overline{r}], r_s \Rightarrow_i \text{addi } \overline{r_d}, \overline{r_s}, 0
\]

\[
L \vdash \text{unfold } r_d, r_s \Rightarrow_i \text{addi } \overline{r_d}, \overline{r_s}, 0
\]

\[
L \vdash \text{id } r_d, r_s(i) \Rightarrow_i \text{id } \overline{r_d}, \overline{r_s(i)}
\]

\[
L \vdash \nu \Rightarrow_i \text{Dc}(M(l)) \quad L \vdash I \Rightarrow_s M[(i + 1) \ldots j]
\]

\[
L \vdash \nu; I \Rightarrow_s M[i \ldots j]
\]

\[
\text{Dc}(M(i)) = \text{jd } (L(l'))
\]

\[
L \vdash \text{jd } l' \Rightarrow_s M[i \ldots i]
\]

\[
\text{Dc}(M(i)) = \text{jmp } \overline{r}
\]

\[
L \vdash \text{jmp } r \Rightarrow_s M[i \ldots i]
\]
**FPCC Proofs**

Initial Condition: \( \text{Inv} \left( S_0 \right) \)

Preservation: \( \forall S : \text{State}. \text{Inv} \left( S \right) \rightarrow \text{Inv} \left( \text{Step} \left( S \right) \right) \)

Progress: \( \forall S : \text{State}. \text{Inv} \left( S \right) \rightarrow \text{SP} \left( S \right) \)

- Initial Condition obtained during the initial translation from the FTAL program
- Progress is easy …

\[
\text{Inv} (S) = \exists P : \text{program}. \ \exists D : (\neg P). \ P \Rightarrow S
\]

\[
\text{SP} (M, \overline{R}, pc) = (\text{Dc} (M (pc)) \neq \text{illegal})
\]
FPCC Preservation Proof

$$\Pi S : State. \text{Inv}(S) \rightarrow \text{Inv}(\text{Step}(S))$$

- **Premise** (for any $S$):
  - We have a well-typed FTAL program which translates to $S$

- **Desired conclusion**:
  - There exists a well-typed FTAL program which translates to $(\text{Step}(S))$

\[
\text{Inv}(S) = \exists P : program. \exists D : (\vdash P). P \Rightarrow S
\]
...Use the FTAL Soundness Theorems!

Theorem 1 (Progress)
If $\vdash P$, then there exists $P'$ such that $P \hookrightarrow P'$.

Theorem 2 (Preservation)
If $\vdash P$ and $P \hookrightarrow P'$, then $\vdash P'$.

Given well-typed $P$, we know that there exists a well-typed $P'$.

Now, we need to show that $P'$ translates to Step($S$) ...
Relationship between FTAL evaluation and machine semantics

![Diagram showing the relationship between FTAL evaluation and machine semantics]

- Use induction on structure of $P$'s typing derivation to get necessary information about all possible forms of $P$, $P'$, $S$, and Step($S$).
That's It!

- Compile high-level typed program (e.g. Java/ML/C#) to FTAL(++)..
- Type-check FTAL program to get a typing derivation.
- Translate the FTAL program to initial machine state.
- Plug in the initial typing derivation.
- Apply Preservation and Progress Thms.
- And you have a package of proof-carrying machine code!!!
Does it really work?

- Coq “implementation”
  - Tedious but workable
  - Almost fully complete…
Technical Requirements

- Corresponding typed assembly language (TAL) for the machine
- Type system, soundness proof of the TAL must be encodable in the logic
- Criteria required by the safety policy needs to be reflected in the type system
  - In present case, safety policy only cares about getting “stuck”, but may specify more complex policies such as memory access, etc.
FPCC code packages can be constructed following a simple, “generic” syntactic recipe

First comprehensive study on the syntactic approach

- Supports variety of powerful language features (recursive types, mutable fields, etc.)

Express an exact relationship between TAL and FPCC technologies

Design of a novel, realistic TAL