Proof-Carrying Code:
From 19th Century Logic
To 21st Century Computing

Nadeem Abdul Hamid
Yale University

February 17, 2004
A Bit of Trivia
triv·i·um n. pl. triv·i·a [tri-, three- + via, road, way]
The lower division of the seven liberal arts in medieval schools, consisting of grammar, rhetoric, and **logic** – being a triple way, as it were, to eloquence.
trivium \( n. \) pl. trivium \( a \) [tri-, three- + via, road, way]

The lower division of the seven liberal arts in medieval schools, consisting of grammar, rhetoric, and \textit{logic} – being a triple way, as it were, to eloquence.

- What is \textit{Logic}?
trivium n. pl. trivias [tri-, three- + via, road, way]
The lower division of the seven liberal arts in medieval schools, consisting of grammar, rhetoric, and logic – being a triple way, as it were, to eloquence.

- What is Logic?

Charles Lutwidge Dodgson (1832-1898) ...
“Contrariwise,” continued Tweedleddee, “if it was so, it might be; and if it were so, it would be; but as it isn’t, it ain’t. That’s logic.”

[Lewis Carroll, Through the Looking Glass]
The First Age of Logic

- Symbolic logic (500BC - 19th Century)
  - All Greeks are men.
  - All men are mortal.
  → All Greeks are mortal.

Paradoxes: “This sentence is false.”
Symbolic logic (500BC - 19th Century)

- All toves are bojums.
- All bojums are slithy.
→ All toves are slithy.
The First Age of Logic

- Symbolic logic (500BC - 19th Century)
  - All toves are bojums.
  - All bojums are slithy.
  → All toves are slithy.

- Paradoxes: “This sentence is false.”
The Age of Mathematical Logic

- Mid–19th to mid–20th century
- Logic as a (the) language for mathematics
- Boole, Frege, Cantor, Hilbert, Russell, Gödel, Church, Turing

Paradoxes again (naïve set theory: –is a member of itself?)

Incompleteness...
The Age of Mathematical Logic

- Mid–19th to mid–20th century
- Logic as a (the) language for mathematics
- Boole, Frege, Cantor, Hilbert, Russell, Gödel, Church, Turing
- Paradoxes again (naïve set theory: \( T = \{S | S \notin S\} \) – is \( T \) a member of itself?)
- Incompleteness . . .
The New Age of Logic

Logic in Computer Science

- Boolean circuits
- Combinatorial analysis (NP-completeness)
- Databases: SQL (standard first-order logic)
- Formal semantics (programming languages)
- Design validation and verification (hardware)
- AI: mechanized reasoning and expert systems
- Network security: proof-carrying code
Back to Frege

- Gottlob Frege (1848-1925)
- *Begriffsschrift* (“Conceptual Notation”), 1879

If *today is Tuesday* then *we are in Georgia.*

*Today is Tuesday.*

We are in Georgia.
Begriffsschrift, 1879
New Notation

\[ B \rightarrow A \]

“\( B \) implies \( A \)”

\[
\begin{align*}
\vdash B \rightarrow A & \quad \vdash B \\
\vdash A & \\
\end{align*}
\]

(modus ponens)

\[
\begin{align*}
\vdash A \rightarrow A & \quad (1) \\
\vdash A \rightarrow (B \rightarrow A) & \quad (2) \\
\vdash (A \rightarrow (B \rightarrow C)) & \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)) & \quad (3)
\end{align*}
\]
More Logical Connectives, Quantifiers

\[ A \text{ implies } B \quad A \rightarrow B \]
\[ A \text{ and } B \quad A \land B \]
\[ A \text{ or } B \quad A \lor B \]
\[ \text{not } A \quad \neg A \]
\[ \text{for all } \ldots \quad \forall \ldots \]
\[ : \]
Natural Deduction

- Gerhard Gentzen, 1934

- Use of assumptions: \( B_1, \ldots B_n \vdash A \)
  \( (\Delta, \Gamma = B_1, \ldots B_n) \)
Natural Deduction

- Gerhard Gentzen, 1934
- Use of assumptions: $B_1, \ldots, B_n \vdash A$
  $(\Delta, \Gamma = B_1, \ldots, B_n)$

\[
\Delta, A \vdash A \quad \text{ID}
\]

\[
\Delta, B \vdash A \quad \Delta \vdash B \rightarrow A \quad \rightarrow I
\]

\[
\Delta \vdash B \rightarrow A \quad \Delta, \Gamma \vdash A \quad \Gamma \vdash B \quad \rightarrow E
\]

\[
\Delta \vdash A \quad \Gamma \vdash B \quad \Delta, \Gamma \vdash A \wedge B \quad \wedge I
\]

\[
\Delta \vdash A \wedge B \quad \Delta \vdash A \quad \wedge E_1
\]

\[
\Delta \vdash A \wedge B \quad \Delta \vdash B \quad \wedge E_2
\]
A Roundabout Proof of $A \land B$

\[
\begin{align*}
(B \land A) \vdash (B \land A) & \quad \text{ID} \\
(B \land A) \vdash A & \quad \land \text{-}E^2 \\
(B \land A) \vdash B & \quad \land \text{-}E^1 \\
(B \land A) \vdash A \land B & \quad \land \text{-}I \\
\vdash (B \land A) \rightarrow (A \land B) & \quad \rightarrow \text{-}I \\
A, B \vdash A \land B & \quad \land \text{-}E
\end{align*}
\]
• Lambda calculus: 1932
• New formulation of logic
• Church-Turing thesis: problems are either solvable or unsolvable by mechanical methods of computation
The Lambda ($\lambda$) Calculus

- All computation reduced to notion of substitution
- Compact notation for writing functions

The function $f$ where $f(x) = x \times x$. \\

$$\Rightarrow$$ \\

$\lambda x . x \times x$

$$f(3) = 3 \times 3 = 9$$ \\

$$\Rightarrow$$ \\

$$(\lambda x . x \times x)(3) \Rightarrow (x \times x)[3/x] \Rightarrow 3 \times 3 \Rightarrow 9$$
Church-Rosser Theorem

- Reduction order does not matter

\[(\lambda x.x \times x)((\lambda y.y + 1)(2))\]
Church-Rosser Theorem

- Reduction order does not matter

\[
(\lambda x.x \times x)((\lambda y.y + 1)(2))
\]

\[
(\lambda x.x \times x)(2 + 1) \quad ((\lambda y.y + 1)(2)) \times ((\lambda y.y + 1)(2))
\]
Church-Rosser Theorem

- Reduction order does not matter

\[
(\lambda x.x \times x)((\lambda y.y + 1)(2)) \Rightarrow \quad (\lambda x.x \times x)(2 + 1) \quad \Rightarrow \quad ((\lambda y.y + 1)(2)) \times ((\lambda y.y + 1)(2)) \Rightarrow \quad (2 + 1) \times (2 + 1)
\]
Currying Multiple Arguments

- Functions return functions

\[ g(x, y) = x \times x + y \times y \]
\[ g(3, 4) = 3 \times 3 + 4 \times 4 = 25 \]

\[ ((\lambda x. \lambda y. x \times x + y \times y)(3))(4) \]
\[ \Rightarrow (\lambda y. 3 \times 3 + y \times y)(4) \]
\[ \Rightarrow 3 \times 3 + 4 \times 4 \]
\[ \Rightarrow 25 \]
Data Structures: Pairs

\[ \langle t, u \rangle \]  
\[ \langle t, u \rangle . \text{fst} \Rightarrow t \]  
\[ \langle t, u \rangle . \text{snd} \Rightarrow u \]  

build a pair  
(reduction rules)
Data Structures: Pairs

\[ \langle t, u \rangle \text{ build a pair} \]

\[ \langle t, u \rangle . \text{fst} \Rightarrow t \text{ (reduction rules)} \]

\[ \langle t, u \rangle . \text{snd} \Rightarrow u \]

\[ \lambda z. \langle z . \text{snd}, z . \text{fst} \rangle \text{ swap elements} \]
Data Structures: Pairs

\[ \langle t, u \rangle \quad \text{build a pair} \]
\[ \langle t, u \rangle . \text{fst} \Rightarrow t \quad \text{(reduction rules)} \]
\[ \langle t, u \rangle . \text{snd} \Rightarrow u \]

\[ \lambda z. \langle z . \text{snd}, z . \text{fst} \rangle \quad \text{swap elements} \]

\[ (\lambda z. \langle z . \text{snd}, z . \text{fst} \rangle)(\langle y, x \rangle) \]
\[ \Rightarrow \langle \langle y, x \rangle . \text{snd}, \langle y, x \rangle . \text{fst} \rangle \]
\[ \Rightarrow \langle x, y \rangle \]
No Extensions Needed!

- **Numbers:** $n = \lambda f.\lambda x. f(\cdots f(x))$
- Addition, multiplication, subtraction, . . .
- Booleans, pairs, other data structures, . . .
- Recursive functions

- Any function on numbers computable by a machine can be represented by a lambda term, built from (1) $\lambda x. t$, (2) $t(u)$, and (3) $x!$
Typed Lambda Calculus

- Introduced 1940, to avoid paradoxes

- Function types: $A \rightarrow B$
- Type of a pair: $A \land B$
- Typing judgment: $x_1 : B_1, \ldots x_n : B_n \vdash t : A$

- Function application rule:
  $$\Delta \vdash t : B \rightarrow A \quad \Delta \vdash u : B$$
  $$\Delta \vdash t(u) : A$$
\[ \lambda \text{ Typing Rules} \]

\[
\begin{align*}
\Delta, x : B \vdash t : A & \quad \rightarrow \text{I} \\
\Delta \vdash \lambda x.t : B \rightarrow A & \quad \rightarrow \text{E}
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash t : A & \quad \Delta \vdash u : B \\
\Delta \vdash \langle t, u \rangle : A \land B & \quad \land \text{I} \\
\Delta \vdash t \cdot \text{fst} : A & \quad \land \text{E}_1 \\
\Delta \vdash t \cdot \text{snd} : B & \quad \land \text{E}_2
\end{align*}
\]
Type of “pair-swap”

\[
\begin{align*}
\frac{z : (B \land A) \vdash z : (B \land A)}{\text{ID}} & \quad \frac{z : (B \land A) \vdash z : (B \land A)}{\text{ID}} \\
\frac{z : (B \land A) \vdash z.\text{snd} : A}{\wedge \text{E}2} & \quad \frac{z : (B \land A) \vdash z.\text{fst} : B}{\wedge \text{E}1} \\
\frac{\frac{z : (B \land A) \vdash \langle z.\text{snd}, z.\text{fst} \rangle : A \land B}{\text{ID}}}{\frac{\frac{\Rightarrow \lambda z.\langle z.\text{snd}, z.\text{fst} \rangle : (B \land A) \rightarrow (A \land B)}{\text{ID}}}{\downarrow \text{I}}} \quad \frac{\frac{\Rightarrow \lambda z.\langle z.\text{snd}, z.\text{fst} \rangle : (B \land A) \rightarrow (A \land B)}{\text{ID}}}{\downarrow \text{I}}
\end{align*}
\]

\[
\begin{align*}
\frac{\Rightarrow \lambda z.\langle z.\text{snd}, z.\text{fst} \rangle : (B \land A) \rightarrow (A \land B)}{\text{E}} & \quad \frac{x : A, y : B \vdash \langle y, x \rangle : B \land A}{\text{E}} \\
x : A, y : B \vdash (\lambda z.\langle z.\text{snd}, z.\text{fst} \rangle)(\langle y, x \rangle) : A \land B
\end{align*}
\]
The Curry-Howard Isomorphism

Correspondence between natural deduction and the lambda calculus

- First noted by Haskell Curry (1900-1982) in 1960
- Written down by William Howard in 1969
- Finally published in 1980
Proofs are Programs!

<table>
<thead>
<tr>
<th>λ →</th>
<th>N.D.</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>assumption</td>
</tr>
<tr>
<td>type constructor</td>
<td>connective</td>
</tr>
<tr>
<td>type</td>
<td>proposition</td>
</tr>
<tr>
<td>term (program)</td>
<td>proof</td>
</tr>
<tr>
<td>inhabitation</td>
<td>provability</td>
</tr>
<tr>
<td>reducable expression</td>
<td>redundant proof tree</td>
</tr>
<tr>
<td>reduction</td>
<td>normalization</td>
</tr>
</tbody>
</table>

I ideas and observations about logic are ideas and observations about programming languages
Proof-Carrying Code


- Foundational PCC: Appel and Felty, 2000–02
- Syntactic FPCC: Hamid, Shao, et al., 2002
- Ongoing work at Berkeley, CMU, Princeton, Yale, …
A Logical Approach

Please install and execute this program.

OK, but let me quickly look over the instructions first.

Code producer

Host
A Logical Approach

This store instruction looks dangerous!

Code producer

Host
A Logical Approach

Code producer

Host

Can you prove that it is always safe?
A Logical Approach

Yes! Here’s the proof I got from my certifying compiler!

Can you prove that it is always safe?

Code producer

Host
Your proof checks out. I believe you because I believe in logic.
The Code Safety Problem

Is this safe to execute?
The Code Safety Problem

“Trusted Computing Base” (TCB)
The Code Safety Problem

```
CPU

Theorem Prover

00111101
10010011

Native Compiler

Code

“Good” Java program

Flexible and powerful. But really really really hard and must be correct.
```
Proving vs. Checking

rlrlrlrlrlrlrlrlrlrlrlrlrlrl...
Basic Concept of PCC

Proof Checker → CPU

Proof

00111101 10010011

Proving Compiler

Code

“Good” Java program
First Proof-Carrying Code Systems

[Necula, Lee 1996]
#1: How large is the trusted computing base?

- Machine (hardware) specification and semantics
- Safety policy
- Proof checker (and VC generator)
- The Logic itself

#2: How big are the proofs?

#3: How long does it take to produce/check the proofs?
#1: How large is the trusted computing base?

\[
\begin{align*}
\Delta; \Psi; \Gamma & \vdash o_1 : \text{int} \\
\Delta; \Psi; \Gamma & \vdash o_2 : \text{set}_=(B) \\
\Delta; \Psi; \Gamma & \vdash \text{rco}(r) : \tau_1 \lor \tau_2 \\
\Delta & \vdash \tau_1 \lor \tau_2 : TW \\
\Delta; \Psi; \Gamma\{r:\tau_1\} & \vdash o_1 : \tau'_1 \\
\Delta; \Psi; \Gamma\{r:\tau_2\} & \vdash o_1 : \tau'_2 \\
\Delta & \vdash \tau'_1 \land \tau'_{\text{unsat}} \leq \text{void} \\
\Delta & \vdash \tau'_2 \land \tau'_{\text{sat}} \leq \text{void} \\
\Delta; \Psi; \Gamma & \vdash o_3 : (\Gamma\{r:\tau_1\}) \rightarrow 0 \\
\Delta; \Psi; \Gamma\{r:\tau_2\} & \vdash I \\
\Delta; \Psi; \Gamma & \vdash \text{cmpjcc} \ o_1, o_2, \kappa, o_3; I
\end{align*}
\]
**Foundational Proof-Carrying Code**

(as opposed to type-specialized “conventional” PCC)

- Use a completely foundational mathematical logic with no (or very few) built-in axioms.
- Safety policy, machine state and semantics, and safety proof built from the foundations of mathematical logic

- Flexible
- Secure (much smaller TCB)
- Complex?
FPCC System

[Appel, Felty 2000; Hamid et al. 2002]
Summary: Proof-Carrying Code

- Representation of logic/proofs
- Burden of proving safety on the code producer
- Consumer doesn’t care how proofs constructed
- “Tamperproof”
- No cryptography/trusted third parties needed
- No effect on execution time after checking

- In practice? Lots of work remaining . . .