Proof Carrying Code by Geroge Necula

Arunothia Marappan

Indian Institute of Technology Kanpur

arunothi@iitk.ac.in

April 13, 2016

Acknowledgement

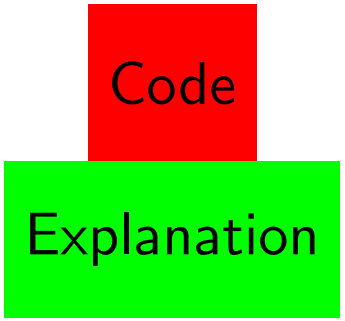
I thank Prof. Anil Seth for giving me the opportunity to read, understand and present this topic.

Overview

PCC is a general framework that allows the host to do a quick check for certain safety properties of the agent.

Host It defines a framework on which the explanation can be tested for the following -

- The explanation lies within established framework.
- It pertains to the safety policy.
- The explanation matches the actual code of the agent.



Agent

- The explanation has reasonings as to why the agent's code complies to safety policy.

Overview

Figure taken from page-179 of book [1]

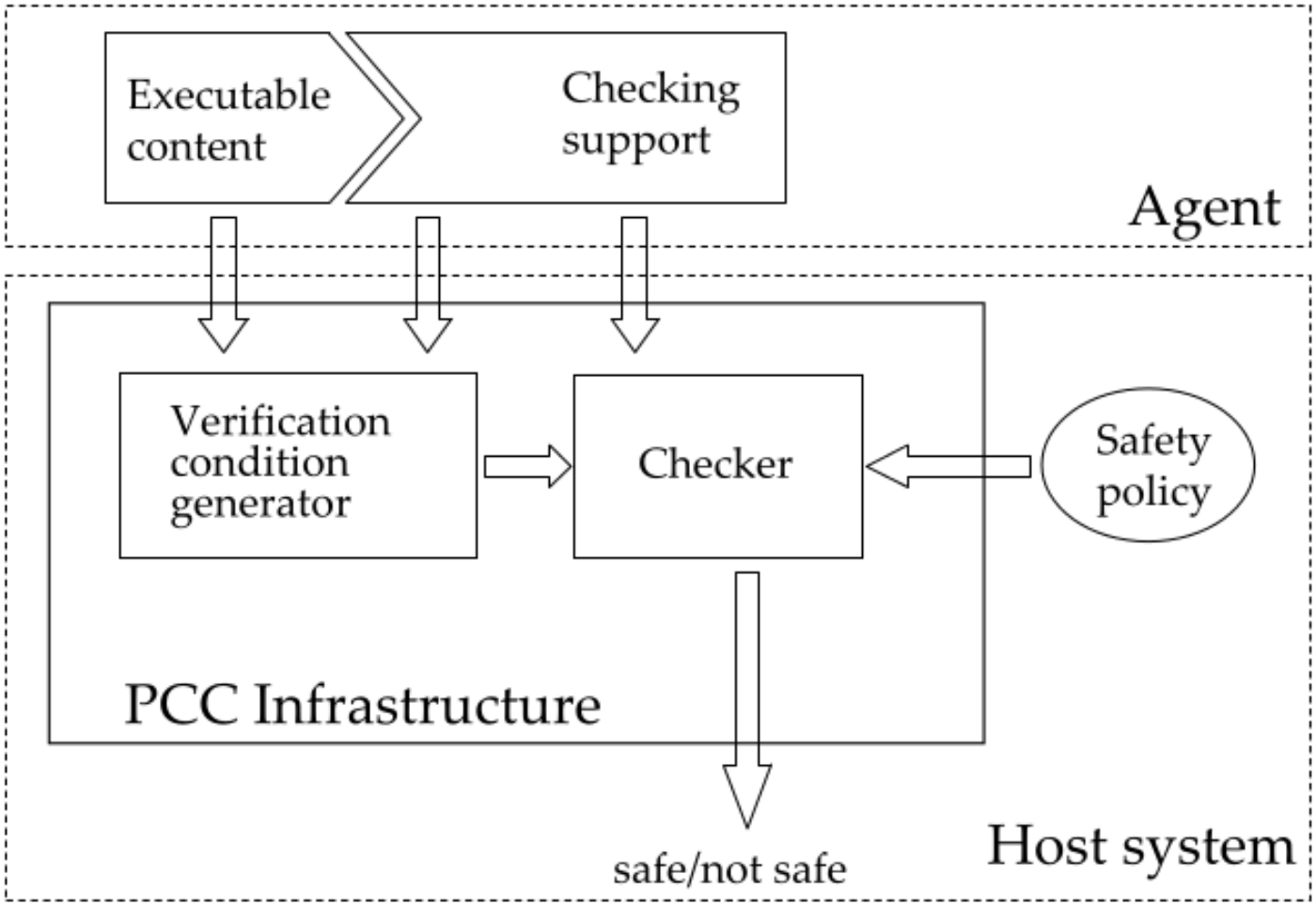


Figure 5-1: The Touchstone PCC architecture

Overview

We will be dealing with the example of *maybepairlist* or *mplist* throughout. *mplist* is defined as a list containing either of

- *Int* (which will be taken odd for a better representation. (i.e). x will be represented as $2x + 1$).
- $\{Int; Int\}$. (A pointer pointing to a pair of integers).

$r_x := e$
$r_x := Mem[e]$
$Mem[e'] = e$
jump L
if e jump L
return

Example - [Int 2; Pair(3,4)]

Formalizing the Safety Policy

- Safety conditions specify a set of instructions that will be dangerous to execute without the specified verification for each.
- These usually comprise of memory operations, function calls and returns.

Elements of Safety Policy

- A language of symbolic expressions and formulae for the verification conditions.
- A set of function precondition and postcondition for all functions that form the interface between agent and host.
- A set of proof rules for verification conditions.

Formalizing the Safety Policy

We use First Order Logic for our formalization.

Formalization

- Formulas - $F ::= True | F_1 \wedge F_2 | F_1 \vee F_2 | F_1 \rightarrow F_2 | \forall x.F | \exists x.F$
  $| \text{ addr } E_a | E_1 = E_2 | E_1 \neq E_2 | f(E_1, \cdots, E_n)$
- Expressions - $E ::= x |$ Sel E_m $E_a |$ Upd E_m E_a $E_v | f(E_1, \cdots E_n)$

- addr $[E_a]$ - It is specified by VC-Gen when a memory read or write is called on E_a. It is true when E_a is a valid address.
- Sel $[E_m \ E_a]$ - Denotes contents of memory address E_a at memory state E_m.
- Upd $[E_m \ E_a \ E_v]$ - Denotes a new memory state that is obtained by storing value E_v at E_a in the memory state E_m.

Extending the Safety Policy

The following are the extensions required to safety policy formulations to solve our example -

Formalization

- Word Types - $W ::= Int|Ptr\{S\}|listW|\{x|f(x)\}$
- Structure Types - $S ::= W|W;S$
- Formulas - $F ::= \cdots|E : W|$ listinv E_m

- $\{x|f(x)\}$ - Set of all those values for which $f(x) = True$.
- listinv M holds true when each memory address that is assigned a pointer type contains values that are assigned appropriate types.

Examples

Typing judgement for $x : mplist$ in this system now will be

$x : mplist$

$$x : list\{y|(\text{even } y) \rightarrow y : ptr\{int; int\}\}$$

Some more examples -

x :singleton with v

$$x : \{x|x = v\}$$

Alternate representation for *mplist*

$$\{x|x : ptr\{\{y|y = 0\}; int\} \vee x : ptr\{\{y|y = 1\}; int; int\}\}$$

Preconditions and Postconditions

The function precondition and postcondition for our agent are -

Problem - Sum of elements of *mplist*

- $Pre_{sum} = r_x$: mplist $\wedge$ listinv r_M
- $Post_{sum}$ = listinv r_M

Are these well formed formulae?

Another example -

Function of type $(int * int) * int\ list \rightarrow int\ list$

- $Pre = r_1 : ptr\{int; int\} \wedge r_2$: list $int \wedge$ listinv r_M
- $Post = r_R$: list $int \wedge$ listinv r_M

Proof Rules

Figure taken from page-186 of book [1]

$$\frac{F_1 \quad F_2}{F_1 \wedge F_2}\ (\text{ANDI}) \qquad \frac{F_1 \wedge F_2}{F_1}\ (\text{ANDEL}) \qquad \frac{F_1 \wedge F_2}{F_2}\ (\text{ANDER})$$

$$\frac{\begin{array}{c} F_1 \\ \vdots \\ F_2 \end{array}}{F_1 \Rightarrow F_2}\ (\text{IMPI}) \qquad \frac{F_1 \Rightarrow F_2 \quad F_1}{F_2}\ (\text{IMPE})$$

$$\frac{A = A'}{\mathsf{sel}\ (\mathsf{upd}\ M\ A\ V)\ A' = V}\ (\text{MEM0}) \qquad \frac{A \neq A'}{\mathsf{sel}\ (\mathsf{upd}\ M\ A\ V)\ A' = \mathsf{sel}\ M\ A'}\ (\text{MEM1})$$

Figure 5-5: Built-in proof rules

Proof Rules for our Example

Figure taken from page-187 of book [1]

$$\frac{}{0 : \texttt{list}\ W}\ (\text{NIL})$$

$$\frac{E : \texttt{list}\ W \qquad E \neq 0}{E : \texttt{ptr}\ \{W; \texttt{list}\ W\}}\ (\text{CONS})$$

$$\frac{E : \{y \mid F(y)\}}{F(E)}\ (\text{SET})$$

$$\frac{E : \texttt{ptr}\ \{W;S\}}{E : \texttt{ptr}\ \{W\}}\ (\text{THIS})$$

$$\frac{E : \texttt{ptr}\ \{W;S\}}{E+4 : \texttt{ptr}\ \{S\}}\ (\text{NEXT})$$

$$\frac{A : \texttt{ptr}\ \{W\} \qquad \texttt{listinv}\ M}{(\texttt{sel}\ M\ A) : W}\ (\text{SEL})$$

$$\frac{\texttt{listinv}\ M \qquad A : \texttt{ptr}\ \{W\} \qquad V : W}{\texttt{listinv}\ (\texttt{upd}\ M\ A\ V)}\ (\text{UPD})$$

$$\frac{A : \texttt{ptr}\ \{W\}}{\texttt{addr}\ A}\ (\text{PTRADDR})$$

Figure 5-6: Proof rules specific to the example safety policy

Verification-Condition Generation

This is the method of enforcing safety policy. PCC Infrastructure is divided into two parts to perform low-level checking on Assembly code that has almost no structure.

- VC - Gen
  - This does a **syntactic scanning** of the code.
  - This also identifies **what must** be checked for each instruction.
- Checker
  - This identifies **how** to perform the checking.
  - It uses the support of the proof provided by the checking support.

Problems posed by Low-Level Languages

High Level

- match x with
  $\_ :: t \rightarrow e$

Low Level

- $r_t := r_x$
  $r_t := r_t + 4$
  if $r_x = 0$ jump $LNIL$
  $r_t := Mem[r_t]$

Where x is allocated to r_x and e is compiled with the assumption that on entry, variable t is assigned to r_t.

- To find $tail(x : x_s)$
- We need an address computation and a load.

- $r_t + 4$ - Not only type-check but, also check for constant 4.
- Low level languages are flow and path sensitive.
- Type of variables vary with context as all we have is limited registers.

Past Attempts - Java Byte-Code Verifier

This verifies the Java Virtual Machine Language (JVML).

- This is a relatively high-level design.
- Load and addition operations are bundled into a single byte-code.
- It is designed such that the outcome of conditionals does NOT matter for type-checking.

The Problem!

It does not allow much scope for optimizations.

Solution Proposed by our Model

Symbolic Evaluation

Symbolic Evaluation

This technique has the effect of collecting the results of intermediate computations to create the final result as a complex expression whose meaning is equivalent to the combined expression.

- For each register we maintain a symbolic expression.
- σ : is the range over symbolic states. (It mentions the register to S.E mapping)

At the start, we initialize all the symbolic expressions to fresh variables as we have no idea of the big picture as yet.

$$\sigma_0 = \{r_t = t, r_x = x, r_M = m\}$$

where, t, x are fresh variables. Then at the first program point, the following invariant holds,

$$\exists t.\exists x.\exists m.(r_t = t) \wedge (r_x = x) \wedge (r_M = m)$$

Symbolic Evaluation - Example

Figure taken from page-191 of book [1]

$$\sigma = \{r_t = t, r_x = x, r_M = m\}, A$$

$r_t := r_x$

$$\sigma = \{r_t = x, r_x = x, r_M = m\}, A$$

$r_t := r_t + 4$

$$\sigma = \{r_t = x + 4, r_x = x, r_M = m\}, A$$

if $r_x = 0$ jump LNil

$$\sigma = \{r_t = x + 4, r_x = x, r_M = m\}, A \wedge x \neq 0$$

$r_t :=$ Mem$[r_t]$

$$\sigma = \{r_t = \mathtt{sel}\ m\ (x + 4), r_x = x, r_M = m\}, A \wedge x \neq 0$$

...

LNil:
$$\sigma = \{r_t = x + 4, r_x = x, r_M = m\}, A \wedge x = 0$$

Symbolic Evaluation - Example

The Checker module would have to check the following verification condition for the load instruction.

Check

$$\forall t.\forall x.\forall m.(r_t = x \wedge r_x = x + 4 \wedge r_M = m \wedge A \wedge x \neq 0) \rightarrow \text{addr}\ (x + 4)$$

Sum - An Example Program

Figure taken from page-182 of book [1]

```
1  sum:                                          ;r_x : maybepair list
2  Loop:
3          if r_x ≠ 0 jump LCons                      ; Is r_x empty?
4          r_R := r_acc
5          return
6  LCons:  r_t := Mem[r_x]                    ; Load the first data
7          if even(r_t) jump LPair
8          r_t := r_t div 2
9          r_acc := r_acc + r_t
10         jump LTail
11 LPair:  r_s := Mem[r_t]                ; Get the first pair element
12         r_acc := Mem[r_acc + r_s]
13         r_t := Mem[r_t + 4]                ; and the second element
14         r_acc := r_acc + r_t
15 LTail:  r_x := Mem[r_x + 4]
16         jump Loop
```

Figure 5-4: Assembly code for the function in Figure 5-2

The Role of Program Annotations

VC-Gen attempts to execute untrusted code symbolically in-order to signal all potential danger associated with any instruction. In order to achieve this in finite time, we need the program to be annotated with invariants.

Invariant for block Loop

Loop: INV$= r_x = mplist \wedge$ listinv r_M

Different methods to annotate our code

- By hand by programmer. (A feature that will allow untrusted optimizations by hand)
- Automatically by a certifying compiler. (For a simple type safety policy like ours, annotations will consist of type declarations for the live registers.) [Necula, 1998]

The Verification Condition Generation

For a uniform treatment of functions and loops, we consider that the first instruction in each agent function is an invariant annotation with the precondition predicate.

Inv

Inv : Program Counter $\rightarrow$ Invariant Predicates

$(\sigma\ e)$

Results of substituting register names in e with expression given by σ.

$\sigma[r \leftarrow e]$

Same as σ with r being mapped to e.

π_i

Annotation at Program Counter i

Symbolic Evaluation Function

$SE(i, \sigma)$

It produces a formula that captures all of the verification conditions from i until the next return instruction or invariant.

The symbolic evaluator also ensures that each loop in the code has at least one invariant annotation. This property of it ensures the termination of the SE function.

SE Function

Figure taken from page-194 of book [1]

$$
SE(i,\sigma) = \begin{cases}
SE(i+1, \sigma[r \leftarrow \sigma\ e]) & \text{if } \Pi_i = r := e \\
(\sigma\ e) \Rightarrow SE(L,\sigma) \wedge & \text{if } \Pi_i = \texttt{if } e \texttt{ jump L} \\
\quad (\texttt{not } (\sigma\ e)) \Rightarrow SE(i+1,\sigma) & \\
\texttt{addr } (\sigma\ a) \wedge & \text{if } \Pi_i = r := \texttt{Mem}[a] \\
\quad SE(i+1, \sigma[r \leftarrow (\sigma\ (\texttt{sel } \mathtt{r}_M\ a))]) & \\
\texttt{addr } (\sigma\ a) \wedge & \text{if } \Pi_i = \texttt{Mem}[a] := e \\
\quad SE(i+1, \sigma[r_M \leftarrow (\sigma\ (\texttt{upd } r_M\ a\ e))] & \\
\sigma\ Post & \text{if } \Pi_i = \texttt{return} \\
\sigma\ I & \text{if } \Pi_i = \texttt{INV } I
\end{cases}
$$

Verification Condition Example

Figure taken from page- 196 of book [1]

1: Generate fresh values $\mathsf{r}_M = m_0$, $\mathsf{r}_R = r_0$, $\mathsf{r}_x = x_0$, $\mathsf{r}_{acc} = acc_0$,
$\mathsf{r}_t = t_0$ and $\mathsf{r}_s = s_0$

1: Assume Invariant $\underline{x_0 : \texttt{mp\_list}}$
$\underline{\texttt{listinv}\ m_0}$

2: Invariant $\boxed{x_0 : \texttt{mp\_list}}$
$\boxed{\texttt{listinv}\ m_0}$

2: Generate fresh values $\mathsf{r}_M = m_1$, $\mathsf{r}_R = r_1$, $\mathsf{r}_x = x_1$, $\mathsf{r}_{acc} = acc_1$,
$\mathsf{r}_t = t_1$ and $\mathsf{r}_s = s_1$

2: Assume Invariant $\underline{x_1 : \texttt{mp\_list}}$
$\underline{\texttt{listinv}\ m_1}$

3: Branch 3 taken $\underline{x_1 \neq 0}$

6: Check load $\boxed{\texttt{addr}\ x_1}$

7: Branch 7 taken $\underline{\texttt{even}\ (\texttt{sel}\ m_1\ x_1)}$

Verification Condition Example

Figure taken from page- 196 of book [1]

11: Check load		$\boxed{\texttt{addr}\ (\texttt{sel}\ m_1\ x_1)}$
13: Check load		$\boxed{\texttt{addr}\ ((\texttt{sel}\ m_1\ x_1) + 4)}$
15: Check load		$\boxed{\texttt{addr}\ (x_1 + 4)}$
16: Goto Loop		
2: Invariant		$\boxed{(\texttt{sel}\ m_1\ (x_1 + 4)) : \texttt{mp\_list}}$
		$\boxed{\texttt{listinv}\ m_1}$
7: Branch 7 not taken	$\underline{\texttt{odd}\ (\texttt{sel}\ m_1\ x_1)}$	
10: Goto LTail		
15: Check load		$\boxed{\texttt{addr}\ (x_1 + 4)}$
16: Goto Loop		
2: Invariant		$\boxed{(\texttt{sel}\ m_1\ (x_1 + 4)) : \texttt{mp\_list}}$
		$\boxed{\texttt{listinv}\ m_1}$
3: Branch 3 not taken	$\underline{x_1 = 0}$	
5: Return		$\boxed{\texttt{listinv}\ m_1}$

Figure 5-7: The sequence of actions taken by VCGen

Verification Condition

Global Verification Condition

$VC = \bigwedge_{i \in Dom(Inv)} \forall x_1, \cdots, x_n \cdot \sigma_0 Inv_i \rightarrow SE(i+1, \sigma_0)$
where $\sigma_0 = \{r_1 = x_1, \cdots, r_n = x_n\}$

Notice that invariants act similar to predicates in an inductive proof. They make the assumptions as well as help us get the verification condition.
Consider, this simple example of proof -
$maybepair \triangleq \{y | even(y) \rightarrow y : ptr\{int; int\}\}$

Proving PTRADDR

$x_1 : mplist \qquad x_1 \neq 0$

$x_1 : ptr\{maybepair; mplist\}$

---

addr x_1

Proof - Another Example

Figure taken from page- 197 of book [1]

$$\dfrac{\dfrac{\dfrac{\dfrac{x_1 : \texttt{mp\_list} \quad x_1 \neq 0}{x_1 : \texttt{ptr}\,\{\texttt{maybepair;mp\_list}\}}\,\text{CONS}}{x_1 : \texttt{ptr}\,\{\texttt{maybepair}\}}\,\text{THIS}}{(\texttt{sel}\ m_1\ x_1) : \texttt{maybepair}}\,\text{SEL}}{\dfrac{\dfrac{\dfrac{\texttt{listinv}\ m_1 \qquad \texttt{even}\,(\texttt{sel}\ m_1\ x_1) \Rightarrow (\texttt{sel}\ m_1\ x_1) : \texttt{ptr}\,\{\texttt{int;int}\}}{(\texttt{sel}\ m_1\ x_1) : \texttt{ptr}\,\{\texttt{int;int}\}}\,\text{SET} \qquad \texttt{even}\,(\texttt{sel}\ m_1\ x_1)}{(\texttt{sel}\ m_1\ x_1) : \texttt{ptr}\,\{\texttt{int;int}\}}\,\text{IMPE}}{(\texttt{sel}\ m_1\ x_1) : \texttt{ptr}\,\{\texttt{int}\}}\,\text{THIS}}{\texttt{addr}\,(\texttt{sel}\ m_1\ x_1)}\,\text{PTRADDR}$$

Figure 5-8: Proof of a verification condition

Soundness

If the global verification condition for a program is provable using the proof rules given by the safety policy, then the program is guaranteed to execute without violating memory safety.

Proof-Idea

- First, using VC-Gen they exhibit a derivation of the global verification condition using the safety policy proof rules.
- Then they split the soundness proof into a proof of soundness of the set of safety policy rules and a proof of soundness of the VCGen algorithm.

Representation and Checking of Proofs

In PCC, a derivation using a sound system of proof rules must be attached with the untrusted code so that the checker module can verify it.

Edinburgh Logical Framework (Harper, Honsell, Plotkin, 1993) [1]

- The framework can be used to encode judgements and derivations from a wide variety of logics, including first-order and higher-order logics.
- The implementation of the proof checker is parameterized by a high-level description of the logic. This allows a unique implementation of the proof checker to be used with many logics and safety policies.
- The proof checker performs a directed, one-pass inspection of the proof object, without having to perform search. This leads to a simple implementation of the proof checker that is easy to trust and install in existing extensible systems.
- Even though the proof representation is detailed, it is also compact.

Logic Framework

The LF type theory is a language with entities at three levels -

Levels in LF

- Kinds $K ::= Type | \prod x : A \cdot K$
- Types $A ::= a | A\ M\ | \prod x : A_1 \cdot A_2$
- Objects $M ::= x | c | M_1 M_2 | \lambda x : A \cdot M$

LF-Example

Figure taken from page- 207 of book [1]

expressions(l), formulas (o), word types (w), and structure types (s)

ι	:	Type	true	:	o
o	:	Type	and	:	$o \to o \to o$
w	:	Type	impl	:	$o \to o \to o$
s	:	Type	all	:	$(\iota \to o) \to o$
			eq	:	$\iota \to \iota \to o$
zero	:	ι	neq	:	$\iota \to \iota \to o$
sel	:	$\iota \to \iota \to \iota$	addr	:	$\iota \to o$
upd	:	$\iota \to \iota \to \iota \to \iota$	hastype	:	$\iota \to w \to o$
			ge	:	$\iota \to \iota \to o$
int	:	w			
list	:	$w \to w$			
seq1	:	$w \to s$			
seq2	:	$w \to s \to s$			
ptr	:	$s \to w$			
settype	:	$(\iota \to o) \to w$			

(a) (b)

Figure 5-10: LF signature for the syntax of first-order predicate logic with equality and subscripted variables, showing expression (a) and predicate (b) constructors

LF Representation Function

Figures taken from pages- 208,209 of book [1]

The LF representation function $\ulcorner \cdot \urcorner$ is defined inductively on the structure of expressions, types and formulas. For example:

$$\ulcorner P \Rightarrow (P \wedge P) \urcorner = \texttt{imp}\ \ulcorner P \urcorner\ (\texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner)$$

$$\ulcorner \forall x.\texttt{addr}\ x \urcorner = \texttt{all}\ (\lambda x : \iota.\texttt{addr}\ x)$$

$$\ulcorner \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ F_1 & F_2\end{array}}{F1 \wedge F_2} \urcorner = \texttt{andi}\ \ulcorner F_1 \urcorner\ \ulcorner F_2 \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner$$

$$\ulcorner \dfrac{\begin{array}{c}[F_1] \\ \vdots\ \mathcal{D}^u \\ F_2\end{array}}{F_1 \Rightarrow F_2} \urcorner = \texttt{impi}\ \ulcorner F_1 \urcorner\ \ulcorner F_2 \urcorner\ (\lambda u{:}\texttt{pf}\ \ulcorner F_1 \urcorner.\ulcorner \mathcal{D}^u \urcorner)$$

$$M = \texttt{impi}\ \ulcorner F \urcorner\ (\texttt{and}\ \ulcorner F \urcorner\ \ulcorner F \urcorner)\ (\lambda x{:}\texttt{pf}\ \ulcorner F \urcorner.\texttt{andi}\ \ulcorner F \urcorner\ \ulcorner F \urcorner\ x\ x)$$

Figure 5-12: LF representation of a proof of $F \Rightarrow (F \wedge F)$

LF Representation - Another Example

Try $\forall a \cdot a : ptr\{int\} \rightarrow \text{addr } a$

LF Representation - Another Example

$\forall a \cdot a : ptr\{int\} \rightarrow$ addr a

SOLUTION: The proof of the predicate $\forall a.a : \texttt{ptr}\ \{\texttt{int}\} \Rightarrow \texttt{addr}\ a$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : \texttt{ptr}\ \{\texttt{int}\}}\,\mathrm{u}}{\texttt{addr}\ a}\,\mathrm{PTRADDR}}{a : \texttt{ptr}\ \{\texttt{int}\} \Rightarrow \texttt{addr}\ a}\,\mathrm{IMPI}^{u}}{\forall a.a : \texttt{ptr}\ \{\texttt{int}\} \Rightarrow \texttt{addr}\ a}\,\mathrm{ALLI}^{a}$$

The LF representation of this proof is shown below. Notice how the parameter a and the hypothesis u are properly scoped by using higher-order representation.

The LF Type System

To check that the *LF* object M is the representation of a valid proof of the predicate F we use the *LF* typing to verify that M has type pf $[F]$ in the context of the signature $\sum$ declaring the valid proof rules.

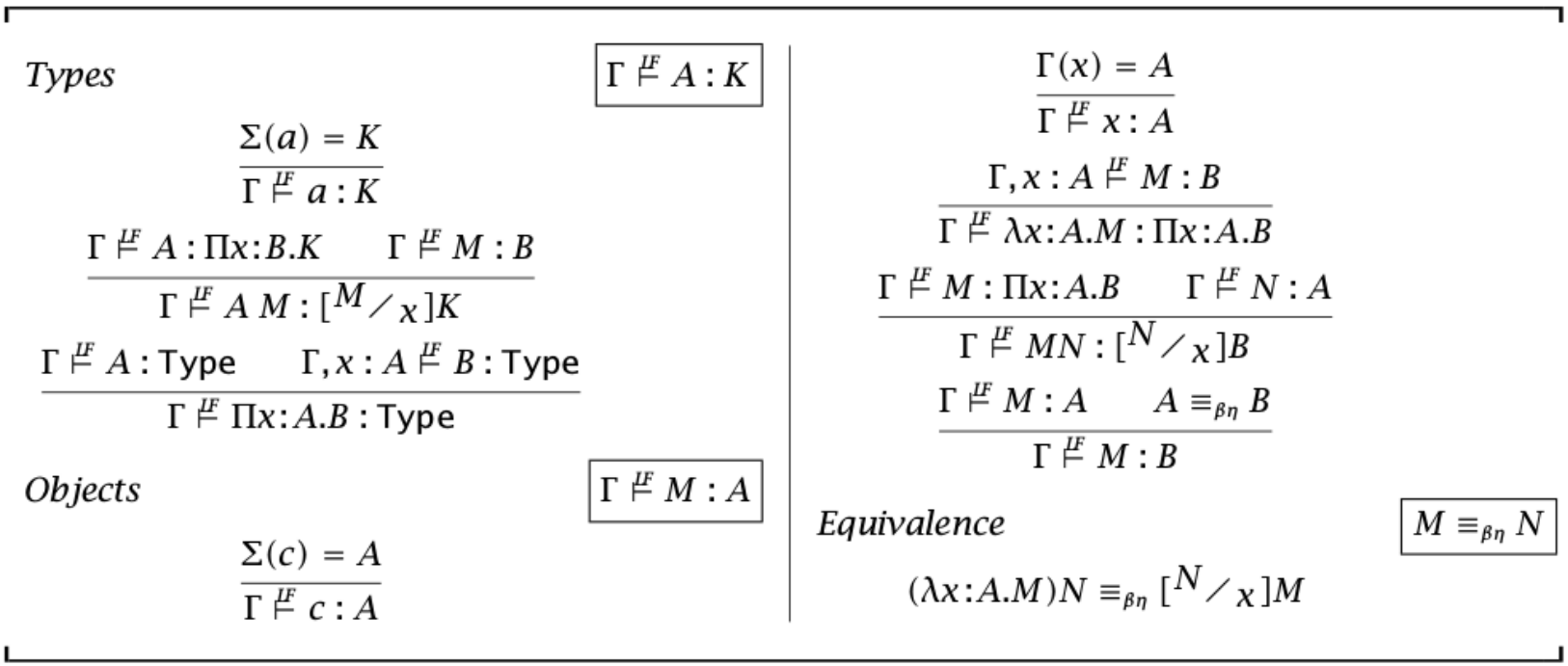
Figure taken from page- 209 of book [1]

Type checking in the LF type system is defined by means of four judgments described below:

$\Gamma \overset{LF}{\vdash} A : K$	A is a valid type of kind K
$\Gamma \overset{LF}{\vdash} M : A$	M is a valid object of type A
$A \equiv_{\beta\eta} B$	type A is $\beta\eta$-equivalent to type B
$M \equiv_{\beta\eta} N$	object M is $\beta\eta$-equivalent to object N

The LF Type System

Figure taken from page- 210 of book [1]



Types $\boxed{\Gamma \vdash^{LF} A : K}$

$$\frac{\Sigma(a) = K}{\Gamma \vdash^{LF} a : K}$$

$$\frac{\Gamma \vdash^{LF} A : \Pi x{:}B.K \qquad \Gamma \vdash^{LF} M : B}{\Gamma \vdash^{LF} A\ M : [M/x]K}$$

$$\frac{\Gamma \vdash^{LF} A : \mathsf{Type} \qquad \Gamma, x : A \vdash^{LF} B : \mathsf{Type}}{\Gamma \vdash^{LF} \Pi x{:}A.B : \mathsf{Type}}$$

Objects $\boxed{\Gamma \vdash^{LF} M : A}$

$$\frac{\Sigma(c) = A}{\Gamma \vdash^{LF} c : A}$$

$$\frac{\Gamma(x) = A}{\Gamma \vdash^{LF} x : A}$$

$$\frac{\Gamma, x : A \vdash^{LF} M : B}{\Gamma \vdash^{LF} \lambda x{:}A.M : \Pi x{:}A.B}$$

$$\frac{\Gamma \vdash^{LF} M : \Pi x{:}A.B \qquad \Gamma \vdash^{LF} N : A}{\Gamma \vdash^{LF} MN : [N/x]B}$$

$$\frac{\Gamma \vdash^{LF} M : A \qquad A \equiv_{\beta\eta} B}{\Gamma \vdash^{LF} M : B}$$

Equivalence $\boxed{M \equiv_{\beta\eta} N}$

$$(\lambda x{:}A.M)N \equiv_{\beta\eta} [N/x]M$$

Figure 5-13: The LF type system

The LF Type System - Adequacy Proof

The adequacy of LF type checking for proof checking in the logic under consideration is arrived at by using two theorems -

1. Theorem-1 : Adequacy of syntax representation
2. Theorem-2 : Adequacy of Derivation Representation

The LF Type System - Disadvantages

Though it has all the mentioned advantages, the code is very large due to a lot of redundancy.

- Check implication Example.
- If we wanted to check the typing of say a binary tree structure, then the redundancy could go from ideal ($nlog(n)$) when balanced to a very large $n^2/2$ when completely skewed.

The implicit LF

We will eliminate all the copies of a given subterm from the proof and rely instead on the copy that exists within the predicate to be proved given by the VC-Gen. We modify the LF type-checking algorithm to reconstruct the missing subterms while it performs type checking.

Figure taken from page- 214 of book [1]

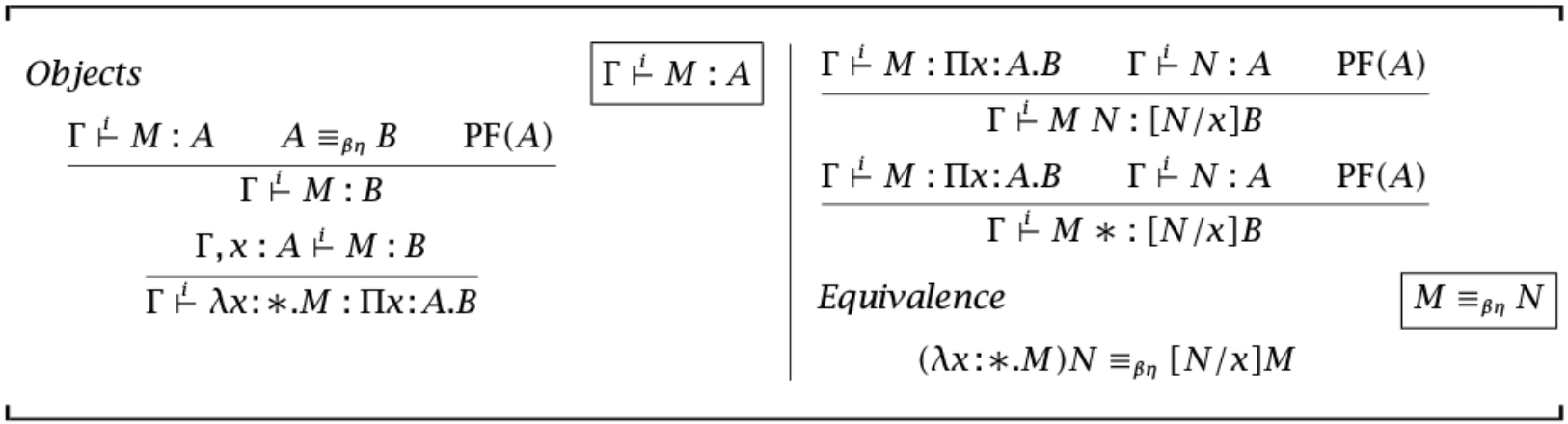


Figure 5-14: The rules that are new in the LF_i type system

Implicit LF Example

Figures taken from pages- 212,213 of book [1]

$$\texttt{impi}\ *_1\ *_2\ (\lambda u{:}*_3.\texttt{andi}\ *_4\ *_5\ u\ u)$$

$$\begin{array}{lllll} & *_1 & \equiv & \ulcorner F \urcorner \\ & *_2 & \equiv & \mathsf{and}\ \ulcorner F \urcorner\ \ulcorner F \urcorner \\ \vdash & (\lambda u{:}*_3.\texttt{andi}\ *_4\ *_5\ u\ u) & : & \mathsf{pf}\ \ulcorner F \urcorner \to \mathsf{pf}\ (\mathsf{and}\ \ulcorner F \urcorner\ \ulcorner F \urcorner) \end{array}$$

$$\begin{array}{llll} & *_3 & \equiv & \mathsf{pf}\ \ulcorner F \urcorner \\ u : \mathsf{pf}\ \ulcorner F \urcorner \vdash & \texttt{andi}\ *_4\ *_5\ u\ u & : & \mathsf{pf}\ (\mathsf{and}\ \ulcorner F \urcorner\ \ulcorner F \urcorner) \end{array}$$

$$\begin{array}{llll} & *_4 & \equiv & \ulcorner F \urcorner \\ & *_5 & \equiv & \ulcorner F \urcorner \\ u : \mathsf{pf}\ \ulcorner F \urcorner \vdash & u & : & \mathsf{pf}\ \ulcorner F \urcorner \\ u : \mathsf{pf}\ \ulcorner F \urcorner \vdash & u & : & \mathsf{pf}\ \ulcorner F \urcorner \end{array}$$

Proof Generation

- The first difficult task, is to generate the code annotations consisting of loop invariants for all loops and of function specifications for all local functions.
- The other difficult task is to prove the verification condition produced by the verification-condition generator.

Figure taken from pages- 215 of book [1]

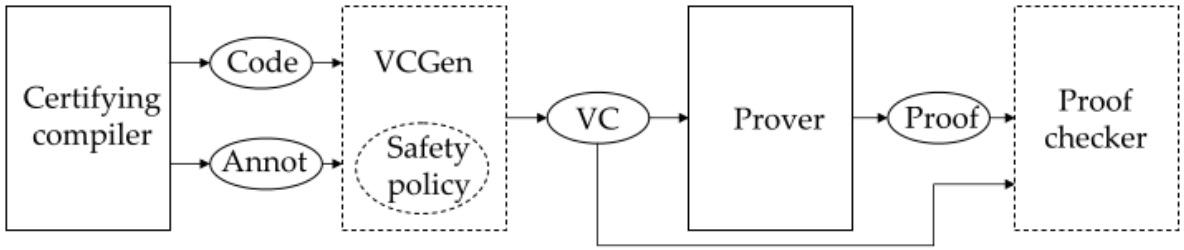


Figure 5-15: Interaction between untrusted PCC tools (continuous lines) and trusted PCC infrastructure (interrupted lines)

Future Reading

The following has not been covered in this presentation.

- The Soundness Proof.
- The Adequacy Proof of LF type system.
- PCC beyond Types.

Thank you!

Questions?

References

Benjamin C. Pierce

Advanced topics in types and programming languages

Published by MIT Press