Mechanizing Metatheory – The Next Chapter

Brigitte Pientka
McGill University
Montreal, Canada
What is *mechanized* metatheory?
Theory Def.
Operational Semantics
Type System
Type Theory
Program Transformations
Type Inference Algorithm
Logic
...

...
1. Mechanizing formal systems together with proofs about them is the talk of the town.
Everybody talks about it.
From Simple Types To Dependent Types
2. Mechanizing formal systems together with proofs about them establishes trust.
2. Mechanizing formal systems together with proofs about them establishes trust... and avoids flaws.
Programs go wrong.

- GCC and LLVM had over 195 bugs
- Compcert, the only compiler where no bugs were found: “This is a strong testimony to the promise and quality of verified compilers.”

[Vu et al. PLDI'14]
Testing correctness of C compilers [Vu et al. PLDI’14]:

- GCC and LLVM had over 195 bugs
- Compcert the only compiler where no bugs were found

“This is a strong testimony to the promise and quality of verified compilers.”

[Vu et al. PLDI’14]
Programming lang. designs and implementations go wrong.

Type Safety of Java (20 years ago)

Java is Type Safe — Probably

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
email: sd and se @doc.ic.ac.uk

Abstract. Amidst rocketing numbers of enthusiastic Java programmers and internet applet users, there is growing concern about the security of executing Java code produced by external, unknown sources. Rather than waiting to find out empirically what damage Java programs do, we aim to examine first the language and then the environment looking for points of weakness. A proof of the soundness of the Java type system is a first, necessary step towards demonstrating which Java programs won’t compromise computer security.

We consider a type safe subset of Java describing primitive types, classes, inheritance, instance variables and methods, interfaces, shadowing, dynamic method binding, object creation, null and arrays. We argue that for this subset the type system is sound, by proving that program execution preserves the types, up to subclasses/subinterfaces.
Type Safety of Java (20 years ago)

Java is not type-safe

Vijay Saraswat

AT&T Research, 180 Park Avenue, Florham Park NJ 07932

Java is not type-safe, though it was intended to be. A Java object may read and modify fields (and invoke methods) private to another object. It may read and modify fields (and invoke methods) private to another object. It may invoke operations not even defined for that object. Thus Java security, guaranteed by the type checking, is compromised.

A safe subset of Java describing primitive types, classes, objects, casting, variables and methods, interfaces, shadowing, dynamic binding, object creation, null and arrays. We argue that this subset of Java is sound, by proving that program execution preserves the types, up to subclasses/subinterfaces.
Programming lang. designs and implementations go wrong.

Type Safety of Java and Scala (20 years later)

Java is Type-safe — Probably

Java and Scala’s Type Systems are Unsound

The Existential Crisis of Null Pointers

Nada Amin
EPFL, Switzerland
nada.amin@epfl.ch

Ross Tate
Cornell University, USA
ross@cs.cornell.edu

A Java object may read and n.
modify internal Java Virtual M.
ance variables.

- Object: A Java object is a subset the type system is
which depends strongly on type.

A Java object may read and n.
modify internal Java Virtual M.
ance variables.

- Object: A Java object is a subset the type system is
which depends strongly on type.

Java is not type-safe, though!

Coke operations not even defined for that
(core dumps). Thus Java security,

Java programmers
about the security
ources. Rather
m do, we
king for
is a
Programming lang. designs and implementations go wrong.

Type Safety of Java and Scala (20 years later)

Java is Type-safe — Probably

Java and Scala’s The Exit

Nada Amin
EPFL, Switzerland
nada.amin@epfl.ch

Type Soundness for Dependent Object Types (DOT)

Tiark Rompf*, Nada Amin†

*Purdue University, USA: {firstname}@purdue.edu
†EPFL, Switzerland: {first.last}@epfl.ch

Java is a language, object oriented. Subset the type system is sound. Option preserves the types, up to subclasses.
Why is it hard to get theories and implementations right?
It’s a tricky business.

“The truth of the matter is that putting languages together is a very tricky business. When one attempts to combine language concepts, unexpected and counterintuitive interactions arise.”

- J. Reynolds
Correct proofs are tricky to write.

On paper:

- Challenging to keep track of all the details
- Easy to skip over details
- Difficult to understand interaction between different features
- Difficulties increase with size

In a proof assistant:

- A lot of overhead in building basic infrastructure
- May get lost in the technical, low-level details
- Time consuming
- Experience, experience, experience
“To those that doubted de Bruijn, I wished to prove them wrong, or discover why they were right. Now, after some years and many hundred hours of labor, I can say with some authority: they were right. De Bruijn indices are foolishly difficult for this kind of proof. [...] The full proof runs to 3500 lines, although that relies on a further library of 1900 lines of basic facts about lists and sets. [...] the cost of de Bruijn is partly reflected in the painful 1600 lines that are used to prove facts about “shifting” and “substitution”.

Ezra Cooper (PhD Student)
What are good high-level proof languages that make it easier to mechanize metatheory?
“The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.”

B. Liskov [1974]
“To know your future you must know your past.” – G. Santayana
Back in the 80s...


• LF = Dependently Typed Lambda Calculus ($\lambda\Pi$) serves as a Meta-Language for representing formal systems

• Higher-order Abstract Syntax (HOAS): Uniformly model binding structures in Object Language with (intensional) functions in LF
Back in the 80s...

1987  
**R. Harper, F. Honsell, G. Plotkin:** A Framework for Defining Logics, LICS’87

1988  
**F. Pfenning and C. Elliott:** Higher-Order Abstract Syntax, PLDI’88

- **LF =** DependentlyTyped Lambda Calculus ($\lambda^\Pi$) serves as a Meta-Language for representing formal systems

- **Higher-order Abstract Syntax (HOAS):**
  Uniformly model binding structures in Object Language with (intensional) functions in LF
Types $A, B ::= \text{nat} \mid A \Rightarrow B$

Terms $M ::= x \mid \text{lam } x:A.M \mid \text{app } M N$
Representing Types and Terms in LF – In a Nutshell

Types $A, B ::= \text{nat} \mid A \Rightarrow B$

Terms $M ::= x \mid \text{lam } x: A. M \mid \text{app } M N$

**LF Representation**

```
obj: type.
nat: obj.
arr: obj → obj → obj.
tm: type.
lam: obj → (tm → tm) → tm.
app: tm → tm → tm.
```

**On Paper (Object Language)**

<table>
<thead>
<tr>
<th>On Paper (Object Language)</th>
<th>In LF (Meta Language)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{lam } x: \text{nat}. x$</td>
<td>$\text{lam } \text{nat } \lambda x.x$</td>
</tr>
<tr>
<td>$\text{lam } x: \text{nat}. (\text{lam } x: \text{nat} \Rightarrow \text{nat}. x)$</td>
<td>$\text{lam } \text{nat } \lambda x.(\text{lam } (\text{arr } \text{nat } \text{nat}) \lambda x.x)$</td>
</tr>
<tr>
<td>$\text{lam } x: \text{nat}. (\text{lam } f: \text{nat} \Rightarrow \text{nat}. \text{app } f \ x)$</td>
<td>$\text{lam } \text{nat } \lambda x.(\text{lam } (\text{arr } \text{nat } \text{nat}) \lambda f. \text{app } f \ x)$</td>
</tr>
</tbody>
</table>

**Higher-order Abstract Syntax (HOAS):**

- Uniformly model bindings with (intensional) functions in LF
- Inherit $\alpha$-renaming and single substitutions
Uniformly Model Binding Structures using LF Functions

Types $A, B ::= \text{nat} \mid A \Rightarrow B \mid \alpha \mid \forall \alpha. A$

Terms $M ::= x \mid \text{lam} \ x:A. M \mid \text{app} \ M \ N \mid \text{let} \ x = M \ \text{in} \ N \mid \text{tlam} \ \alpha. M \mid \ldots$
Uniformly Model Binding Structures using LF Functions

Types $A, B ::= \text{nat} \mid A \Rightarrow B \mid \alpha \mid \forall \alpha. A$

Terms $M ::= x \mid \text{lam } x: A. M \mid \text{app } M \ N \mid \text{let } x = M \text{ in } N \mid \text{tlam } \alpha. M \mid \ldots$

LF Representation

<table>
<thead>
<tr>
<th>obj: type.</th>
<th>tm: type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nat: obj.</td>
<td>lam: obj $\rightarrow$ (tm $\rightarrow$ tm) $\rightarrow$ tm.</td>
</tr>
<tr>
<td>arr: obj $\rightarrow$ obj $\rightarrow$ obj.</td>
<td>app: tm $\rightarrow$ tm $\rightarrow$ tm.</td>
</tr>
<tr>
<td>all: (obj $\rightarrow$ obj) $\rightarrow$ obj.</td>
<td>let: tm $\rightarrow$ (tm $\rightarrow$ tm) $\rightarrow$ tm.</td>
</tr>
<tr>
<td></td>
<td>tlam: (obj $\rightarrow$ tm) $\rightarrow$ tm.</td>
</tr>
</tbody>
</table>

On Paper (Object Language) | In LF (Meta Language)
---|---
$t\text{lam } \alpha. (t\text{lam } x: \alpha. x)$ | $t\text{lam } \lambda a.(t\text{lam } a \lambda x. x)$
$\forall \alpha. \forall \beta. \alpha \Rightarrow \beta$ | $\forall \alpha. \forall \beta. \alpha \Rightarrow \beta$

12
Uniformly Model Binding Structures using LF Functions

Types $A, B ::= \text{nat} \mid A \Rightarrow B \mid \alpha \mid \forall \alpha.

Terms $M ::= x \mid \text{lam } x : A . M \mid \text{app } M \ N \mid \text{let } x = M \text{ in } N \mid \text{tlam } \alpha . M \mid \ldots$

LF = Dependent Lambda Calculus $\lambda \Pi$

- LF functions only encode variable scope
- no recursion, no pattern matching, etc.

- HOAS trees = Syntax trees with binders

- Benefit: $\alpha$-renaming and substitution principles

- Scales: Model derivation trees
  - Hypothetical derivations as LF functions
  - Parametric derivations as LF functions

$\text{obj: type, \ nat: obj, \ arr: obj \to obj \to obj, \ all: (obj \to obj) \to obj.}$

$\text{tm: type, \ lam: obj \to (tm \to tm) \to tm, \ app: tm \to tm \to tm, \ let: tm \to (tm \to tm) \to tm, \ tlam: (obj \to tm) \to tm.}$
Sounds cool... can I do this in OCaml or Agda?
## OCaml

1. `type tm = Lam of (tm -> tm)`
2. `let apply = function (Lam f) -> f`
3. `let omega = Lam (function x -> apply x x)`

What happens, when we try to evaluate `apply omega omega`?
OCaml

1. `type tm = Lam of (tm -> tm)`
2. `let apply = function (Lam f) -> f`
3. `let omega = Lam (function x -> apply x x)`

What happens, when we try to evaluate `apply omega omega`?

It will loop.
An Attempt in OCaml and Agda

OCaml

1  type tm = Lam of (tm -> tm)
2  let apply = function (Lam f) -> f
3  let omega = Lam (function x -> apply x x)

What happens, when we try to evaluate `apply omega omega`?

It will loop.

Agda

data tm : type = lam : (tm -> tm) -> tm

Violates positivity restriction
### OCaml

1. `type tm = Lam of (tm -> tm)`
2. `let apply = function (Lam f) -> f`
3. `let omega = Lam (function x -> apply x x)`

What happens when we try to evaluate `apply omega omega`?

It will loop.

### Agda

```
data tm : type = lam : (tm -> tm) -> tm
```

Functions in OCaml and Agda are opaque (black box).

- We **can observe** the result that a function computes
- We **cannot pattern match** to inspect the function body

**Violates positivity restriction**
OK... so, how do we write recursive programs over with HOAS trees? We clearly want pattern matching, since a HOAS tree is a data structure.
An Attempt to Compute the Size of a Term

size (lam \(\lambda x.\) lam \(\lambda f.\) app f x) 
\[\implies\] size (lam \(\lambda f.\) app f x) + 1 
\[\implies\] size (app f x) + 1 + 1 
\[\implies\] size f + size x + 1 + 1 + 1 
\[\implies\] 0 + 0 + 1 + 1 + 1

“the whole HOAS approach by its very nature disallows a feature that we regard of key practical importance: the ability to manipulate names of bound variables explicitly in computation and proof. ”

[Pitts, Gabbay'97]
Back in 2008...
In LF (Meta Lang.)

\[ \text{lam } \lambda x. \text{lam } \lambda f. \text{app } f \ x \]

\[ \text{lam } \lambda x. \text{lam } \lambda f. \text{app } f \ x \]

LF Typing Judgment:

\[ \psi \vdash \ M : A \]

\uparrow

LF Context

\uparrow

LF Term

\uparrow

LF Type
LF and Holes in HOAS trees – Revisited

In LF (Meta Lang.)

\[ \text{lam } \lambda x. \text{lam } \lambda f. \text{app } f \ x \]

\[ \text{lam } \lambda x. \text{lam } \lambda f. \text{app } f \ x \]

LF Typing Judgment:

\[ x: \text{tm} \quad \vdash \quad \text{lam } \lambda f. \text{app } f \ x : \ \text{tm} \]

\[ \uparrow \quad \vdash \quad \uparrow \]

LF Context \quad LF Term \quad LF Type
LF and Holes in HOAS trees – Revisited

In LF (Meta Lang.)

\[ \text{lam } \lambda x. \text{lam } \lambda f. \text{app } f \ x \]

\[ \text{lam } \lambda x. \text{lam } \lambda f. \text{app } f \ x \]

LF Typing Judgment:

\[ x : \text{tm} \vdash \text{lam } \lambda f. \text{app } f \ x : \text{tm} \]

\[ \uparrow \]

LF Context

\[ \uparrow \]

LF Term

\[ \uparrow \]

LF Type
In LF (Meta Lang.) | Contextual Type
---|---
\( \text{lam } \lambda x. \text{lam } \lambda f. \text{app } f \ x \) | \([x:tm \vdash tm]\)
\( \text{lam } \lambda x. \text{lam } \lambda f. \text{app } f \ x \) | \([x:tm, f:tm \vdash tm]\)

LF Typing Judgment:

\[
x:tm \vdash \text{lам } \lambda f. \text{app } f \ x : tm
\]

What is the type of \( \text{app } f \ x \)? – Its type is \([x:tm, f:tm \vdash tm]\).
**Contextual Types** [Nanevski, Pfenning, Pientka’08]

- **h** is a contextual variable
- It has the contextual type \([x:tm, f:tm \vdash tm]\)
- It can be instantiated with a contextual term \([x, f \vdash \text{app } f x]\)
- Contextual types (\(\vdash\)) reify LF typing derivations (\(\vdash\))
Contextual Types [Nanevski, Pfenning, Pientka’08]

\[ h : [x : \text{tm}, f : \text{tm} \vdash \text{tm}] ; x : \text{tm} \vdash \lambda f. \ h : \text{tm} \]

- **\( h \)** is a contextual variable
- It has the contextual type \([x : \text{tm}, f : \text{tm} \vdash \text{tm}]\)
- It can be instantiated with a contextual term \([x, f \vdash \text{app } f \ x]\)
- Contextual types (\(\vdash\)) reify LF typing derivations (\(\vdash\))

**WAIT!** . . . whatever we plug in for \( h \) may contain free LF variables?
Contextual Types [Nanevski, Pfenning, Pientka’08]

- $h$ is a contextual variable
- It has the contextual type $[y:tm, g:tm \vdash tm]$.
- It can be instantiated with a contextual term $[y,g \vdash app\ g\ y]$.
- Contextual types ($\vdash$) reify LF typing derivations ($\vdash$).

**WAIT!** . . . whatever we plug in for $h$ may contain free LF variables? and we want it to be stable under $\alpha$-renaming . . .
Contextual Types [Nanevski, Pfenning, Pientka’08]

- $h$ is a contextual variable
- It has the contextual type $[y:tm, g:tm \vdash tm]$
- It can be instantiated with a contextual term $[y, g \vdash app g y]$
- Contextual types ($\vdash$) reify LF typing derivations ($\vdash\vdash$)

**WAIT!** . . . whatever we plug in for $h$ may contain free LF variables? and we want it to be stable under $\alpha$-renaming . . .

**Solution:** Contextual variables are associated with LF substitutions
Contextual Type Theory\(^1\) (CTT) \([\text{Nanevski, Pfenning, Pientka’08}]\)

\(\Gamma; \Psi \vdash M : A\)

\(\Gamma; \Psi\ \vdash x : A\)

\(\Gamma; \Psi \vdash x[\sigma] : [\sigma]A\)

\(\text{Closure} \quad \text{Apply subst. } \sigma\)

\(^1\)Footnote for nerds: CTT is a generalization of modal S4.
Proofs as Functional Programs

Terms $t ::= \left[ \Psi \vdash M \right] \mid \ldots$

Types $T ::= \left[ \Psi \vdash A \right] \mid \ldots$

Contextual Logical Framework LF

Renaming
Scope
Binding
Hypothesis
Variables
Substitution
Context

Eigenvariables

Derivation Tree

$\Gamma; \Psi \vdash M : A$

$\Gamma \vdash t : T$
Revisiting the program size

\[
\begin{align*}
\text{size} \quad & \left[ \vdash \text{lam } \lambda x. \text{lam } \lambda f. \text{app } f \ x \right] \\
\implies \text{size} \quad & \left[ x \vdash \text{lam } \lambda f. \text{app } f \ x \right] + 1 \\
\implies \text{size} \quad & \left[ x,f \vdash \text{app } f \ x \right] + 1 + 1 \\
\implies \text{size} \quad & \left[ x,f \vdash f \right] + \text{size} \left[ x,f \vdash x \right] + 1 + 1 + 1 \\
\implies & \quad 0 + 0 + 1 + 1 + 1
\end{align*}
\]

Corresponding program:

\[
\begin{align*}
\text{size} : & \Pi \gamma: \text{ctx. } \left\lfloor \gamma \vdash \text{tm} \right\rfloor \to \text{int} \\
\text{size} \left\lfloor \gamma \vdash \#p \right\rfloor & = 0 \\
\text{size} \left\lfloor \gamma \vdash \text{lam } \lambda x. M \right\rfloor & = \text{size} \left\lfloor \gamma, x \vdash M \right\rfloor + 1 \\
\text{size} \left\lfloor \gamma \vdash \text{app } M \ N \right\rfloor & = \text{size} \left\lfloor \gamma \vdash M \right\rfloor + \text{size} \left\lfloor \gamma \vdash N \right\rfloor + 1;
\end{align*}
\]

• Abstract over context \( \gamma \) and introduce special variable pattern \#p

• Higher-order pattern matching [Miller'91]
Revisiting the program size

\[
\begin{align*}
\text{size } [\gamma \vdash \text{lam } \lambda x. \text{lam } \lambda f. \text{app } f \ x] &
\Rightarrow \text{size } [x \vdash \text{lam } \lambda f. \text{app } f \ x] + 1 \\
\Rightarrow \text{size } [x, f \vdash \text{app } f \ x] + 1 + 1 \\
\Rightarrow \text{size } [x, f \vdash f] + \text{size } [x, f \vdash x] + 1 + 1 + 1 \\
\Rightarrow 0 + 0 + 1 + 1 + 1
\end{align*}
\]

Corresponding program:

\[
\begin{align*}
\text{size : } \Pi \gamma : \text{ctx}. \ [\gamma \vdash \text{tm}] &\rightarrow \text{int} \\
\text{size } [\gamma \vdash \#p] &= 0 \\
\text{size } [\gamma \vdash \text{lam } \lambda x. \ M] &= \text{size } [\gamma, x \vdash M] + 1 \\
\text{size } [\gamma \vdash \text{app } M \ N] &= \text{size } [\gamma \vdash M] + \text{size } [\gamma \vdash N] + 1;
\end{align*}
\]

- Abstract over context \( \gamma \) and introduce special variable pattern \( \#p \)
- Higher-order pattern matching [Miller’91]
What Programs / Proofs Can We Write?

- **Certified programs:**
  Type-preserving closure conversion and hoisting [CPP’13]
  Joint work with O. Savary-Bélanger, S. Monnier

- **Inductive proofs:**
  Logical relations proofs (Kripke-style) [MSCS’18]
  Joint work with A. Cave

- **Coinductive proofs:**
  Bisimulation proof using Howe’s Method [MSCS’18]
  Joint work with D. Thibodeau and A. Momigliano
Remember the PhD student who mechanized strong normalization for STLC in Coq using de Bruijn?
4 Beyond the Challenge

The POPLMARK Challenge is not meant to be exhaustive: other aspects of programming language theory raise formalization difficulties that are interestingly different from the problems we have proposed—to name a few: more complex binding constructs such as mutually recursive definitions, logical relations proofs, coinductive simulation arguments, undecidability results, and linear handling of

Strong Normalization for STLC using Kripke-style Logical Relations Joint work with A. Abel, G. Allais, A. Hameer, A. Momigliano, S. Schäfer, K. Stark

- Easily accessible problem, while still being worthwhile
- Survey the state of the art
- Compare proof assistants
- Encourage development to make them more robust
Lesson 1: Choosing an inductive definition for SN makes proofs modular and simpler – on paper and in mechanizations.

Lesson 2: Beluga exploits high-level abstractions and primitives (HOAS, contexts, substitutions, renamings, etc.) leading to a compact implementation (416 LOC).

Lesson 3: Contextual types provide an abstract, conceptual view of syntax trees within a context of assumptions.
Sounds cool... but how can we get this into type theories (like Agda)?
The strict separation between contextual LF and computations means we cannot embed computation terms directly.

**Contextual Variable Rule**

\[ \frac{x : [\Phi \vdash A] \in \Gamma}{\Gamma; \Psi \vdash x[\sigma] : [\sigma]A} \]

- **Closure**: \([\sigma]A\)
- **Apply subst.**: \(\sigma\)
What if we did?

Rule for Embedding Computations

\[ \Gamma;\Psi \vdash t : [\Phi \vdash A] \quad \Gamma;\Psi \vdash \sigma : \Phi \]

\[ \underline{\Gamma;\Psi \vdash [t]_\sigma : [\sigma]A} \]

Closure  Apply subst.  \( \sigma \)
Cocon: A Type Theory for Defining Logics
Joint work with A. Abel, F. Ferreira, D. Thibodeau, R. Zucchi

- Hierarchy of universes
- Type-level computation
- Writing proofs about functions (such as size)

\[ \text{quote / box } \left[ \Psi \vdash M \right] \]
\[ \text{unquote / unbox } \left[ t \right]_\sigma \]
**Sketch: Translation Between STLC and CCC**

STLC

\[ \text{tm: obj } \rightarrow \text{type.} \]

Cartesian Closed Categories (CCC)

\[ \text{mor:obj } \rightarrow \text{obj } \rightarrow \text{type} \]

Translate an LF context \( \gamma \) to cross product:

\[ \text{ictx}: \Pi \gamma: \text{ctx.} [ \vdash \text{obj}] \]

Example: \( \text{ictx} (x_1:A_1, x_2:A_2) \rightarrow (\text{cross (cross unit } A_1) A_2) \)

Translate STLC to CCC

\[ \text{itm}: \Pi \gamma: \text{ctx.} \Pi A: [ \vdash \text{obj}] \cdot [\gamma \vdash \text{tm [A]}] \rightarrow [ \vdash \text{mor [ictx } \gamma [A]] \]
Bridging the Gap between LF and Martin Löf Type Theory
What’s Next?

Theory

- Decidable equality
- Categorical semantics
- ...

Implementation and Case Studies

- Build an extension to Coq/Agda/Beluga
- Case studies: Equivalence of STLC and CCC
- Meta-Programming (Tactics)
- Compilation
- Proof Search
- ...

Lesson 1: Contextual types provide a type-theoretic framework to think about syntax trees within a context of assumptions.

Lesson 2: Abstractions for variable binding, contexts, and substitution, etc. are useful when we mechanize metatheory.
Lesson 1: Contextual types provide a type-theoretic framework to think about syntax trees within a context of assumptions.

Lesson 2: Abstractions for variable binding, contexts, and substitution, etc. are useful when we mechanize metatheory.

Last but not least: There are many other abstractions and primitives we should explore: heaps, linearity, resources, ...