POPLMark Reloaded:
Mechanizing Logical Relations Proofs

Brigitte Pientka
McGill University

Joint work with A. Abel (Chalmers), A. Hameer (McGill), A. Momigliano (Milan), S. Schäfer (Saarbrücken), K. Stark (Saarbrücken)
Mechanizing formal systems together with proofs establishes trust.
Mechanizing formal systems together with proofs establishes trust... and avoid flaws.
Programs go wrong.
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]
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
e-mail: 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.
Java is not type-safe

Abstract. A monograph on the design and implementation of Java, in press.

Vijay Saraswat

ARJ Research, 180 Park Avenue, Florham Park, NJ 07932

Java is type-safe. Probably.

Java is not type-safe, though it was intended to be.

Java object may read and modify fields (and invoke methods) private to another object. It may invoke operations not even defined for that object.

We argue that allowing dynamic fields, classes, and interfaces (in accordance with existing Java Virtual Machine (JVM) data-structures), is completely unpredictable results, including JVM crashes (core dumps). Thus Java security;

Type Safety of Java (20 years ago)

Programming lang. designs and implementations go wrong.
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

Java is not type-safe, etc.

Java object may read a
Java object internal Java Virtu

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

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

Java is Type Safe

The Existential Crisis of Null Pointers

Java object may read a
Java object internal Java Virtu

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

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

Java is Type Safe

The Existential Crisis of Null Pointers

Java object may read a
Java object internal Java Virtu

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

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

Java is Type Safe

The Existential Crisis of Null Pointers

Java object may read a
Java object internal Java Virtu

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

Nada Amin
EPFL, Switzerland
nada.amin@epfl.ch
Programming language designs and implementations go wrong.

Type Safety of Java and Scala (20 years later)
“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. At this point, even the most experienced designers intuition must be butressed by a rigorous definition of what the language means. ”

- J. Reynolds
The problem: Correct proofs are tricky to write.

- a lot of overhead
  (on paper and even more so in a proof assistant)
- challenging to keep track of details
- hard to understand interaction between different features
- difficulties increase with size
What are good high-level proof languages that make it easier to mechanize and maintain formal guarantees?
POPLMark – A Look Back ...
Spotlight on

“type preservation and soundness, unique decomposition properties of operational semantics, proofs of equivalence between algorithmic and declarative versions of type systems.”

- Structural induction proofs (syntactic)
- Representing and reasoning about structures with binders
- Easy to be understood; text book description (TAPL)
- Small (can be mechanized in a couple of hours or days)
- Explore different encoding techniques for representing bindings
POPLMark Challenge – The Good

✓ Popularized the use of proof assistants
✓ Many submitted solutions
✓ Good way to learn about a technique
✓ Mechanizing proofs is addictive!
POPLMark Challenge – The Bad

• Did we achieve “a future where the papers in conferences such as POPL and ICFP are routinely accompanied by mechanically checkable proofs of the theorems they claim.”?

• Did we get better tool support for mechanizing proofs?
POPLMark Challenge – The Ugly

✗ Did not identify bugs or flaws in existing systems
✗ Did not inspire the development of new theoretical foundations
✗ Did not push existing systems to their limit

“Type soundness results are two a penny.”
Andrew Pitts
Beyond the POPLMark Challenge!
Beyond the POPLMark 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 type environments.” [Aydemir et. al. 2005]
POPLMark Reloaded –
Goals and Target Audience
User Community Including Students

- Learn logical relations proofs a modern way
- Be able to grow the development to rich type theories (for example dependently typed systems)
- Understand the trade-offs in choosing a particular proof environment when tackling such a proof
Framework Developers

- Highlight features that are ideally suited for built-in support
- Highlight current shortcomings (theoretical and practical) in existing proof environments
- Signpost to advertise a given system
- Stimulate research on foundations of proof environments
- Benchmark for evaluating and comparing systems
POPLMark Reloaded:
Strong normalization for the simply-typed lambda-calculus with typed-reductions using Kripke-style logical relations
Simply Typed $\lambda$-Calculus with Type-Directed Reductions

Simply Typed $\lambda$-calculus:

Terms $M, N ::= x | \lambda x : A. M | M N | ()$

Types $A, B ::= A \Rightarrow B | \text{unit}$
Simply Typed $\lambda$-Calculus with Type-Directed Reductions

Simply Typed $\lambda$-calculus:

Terms $M, N ::= x \mid \lambda x: A.M \mid M N \mid ()$

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

Type-directed reductions [Goguen'95]:

\[
\begin{align*}
\Gamma \vdash M \rightarrow N : A \\
\Gamma \vdash \lambda x: A. M : A \Rightarrow B \quad \Gamma \vdash N : A \\
\Gamma \vdash (\lambda x: A. M) N \rightarrow [N/x]M : B \\
\beta \\
M \neq () \\
\Gamma \vdash M \rightarrow () : \text{unit}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M \rightarrow M' : A \Rightarrow B \\
\Gamma \vdash N : A \\
\Gamma \vdash M N \rightarrow M' N : B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M : A \Rightarrow B \\
\Gamma \vdash N \rightarrow N' : A \\
\Gamma \vdash M N \rightarrow M N' : B
\end{align*}
\]

\[
\begin{align*}
\Gamma, x: A \vdash M \rightarrow M' : B \\
\Gamma \vdash \lambda x: A. M \rightarrow \lambda x: A. M' : A \Rightarrow B
\end{align*}
\]
Why Type-directed Reductions?

- Simplifies the study of its meta-theory.
- Concise presentation of the important issues that arise.
- Widely applicable in studying subtyping, type-preserving compilation, etc.
- Types are necessary if we want $\eta$-expansion.

\[
\frac{M \neq \lambda y:A. M'}{\Gamma \vdash M \rightarrow \lambda x:A. M \ x : A \Rightarrow B} \quad * 
\]

WARNING: This rule doesn’t actually work, if we want strong normalization. Use at your own risk.
Normalization

- A term $M$ is said to be weakly normalising if there is a rewrite sequence starting in $M$ that eventually ends in a normal form.
- A term $M$ is said to be strongly normalising if all rewrite sequences starting in $M$ end eventually in a normal form.
Often defined as an accessibility relation:

\[ \forall N. \Gamma \vdash M \rightarrow N : A \quad \Rightarrow \quad \Gamma \vdash N : A \in \text{sn} \]
\[ \Gamma \vdash M : A \in \text{sn} \]

“the reduct analysis becomes increasingly annoying in normalization proofs for more and more complex systems.” Joachimski and Matthes [2003]
• A term $M$ is said to be *weakly normalising* if there is a rewrite sequence starting in $M$ that eventually ends in a normal form.
• A term \( M \) is said to be *weakly normalising* if there is a rewrite sequence starting in \( M \) that eventually ends in a normal form.

• Set of all weakly normalising terms: the smallest set of all normal forms closed under expansion.
A term $M$ is said to be weakly normalising if there is a rewrite sequence starting in $M$ that eventually ends in a normal form.

Set of all weakly normalising terms: the smallest set of all normal forms closed under expansion.

How to obtain the set of all strongly normalising terms?

$\Rightarrow$ Similar ... with a few restrictions
A Modular Approach to Strongly Normalizing Terms
[F. van Raamsdonk and P. Severi 1995]

- Inductive characterization of normal forms ($\Gamma \vdash M : A \in SN$)
- Leads to modular proofs – on paper and in mechanizations

“the new proofs are essentially simpler than already existing ones.”  F. van Raamsdonk and P. Severi
Inductive definition of well-typed strongly normalizing terms

Neutral terms
\[ x : A \in \Gamma \]
\[ \Gamma \vdash x : A \in \text{SNe} \]
\[ \Gamma \vdash R : A \Rightarrow B \in \text{SNe} \]
\[ \Gamma \vdash M : A \in \text{SN} \]
\[ \Gamma \vdash R M : B \in \text{SNe} \]

Normal terms
\[ \Gamma \vdash (\) : \text{unit} \in \text{SN} \]
\[ \Gamma \vdash R : A \in \text{SNe} \]
\[ \Gamma \vdash \lambda x : A. M : A \Rightarrow B \in \text{SN} \]
\[ \Gamma \vdash M \rightarrow_{\text{SN}} M' : A \]
\[ \Gamma \vdash M' : A \in \text{SN} \]
\[ \Gamma \vdash M : A \in \text{SN} \]

Strong head reduction
\[ \Gamma \vdash N : A \in \text{SN} \]
\[ \Gamma \vdash (\lambda x . M) N \rightarrow_{\text{SN}} [N/x]M : B \]
\[ \Gamma \vdash R \rightarrow_{\text{SN}} R' : A \Rightarrow B \]
\[ \Gamma \vdash M : A \]
\[ \Gamma \vdash R M \rightarrow_{\text{SN}} R' M \]
Challenge 1: Equivalence between accessibility and inductive definition of strongly normalizing terms:

\[ \Gamma \vdash M : A \in \text{sn} \iff \Gamma \vdash M : A \in \text{SN}. \]
Definition (Reducibility Candidates: $\Gamma \vdash M \in R$)

- $\Gamma \vdash M \in R$ unit iff $\Gamma \vdash M : \text{unit} \in SN$
- $\Gamma \vdash M \in R \Rightarrow B$ iff $\Gamma \vdash M : A \Rightarrow B$ and for all $N, \Delta$ such that $\Gamma \leq \rho \Delta$, if $\Delta \vdash N \in R \ A$ then $\Delta \vdash (\rho M) N \in R \ B$.

- Contexts arise naturally.
- They are necessary!
- The definition scales to dependently typed setting and stating properties about type-directed equivalence of lambda-terms.

Do we really need to model terms in a "local" context and use Kripke-style context extensions?
Definition (Reducibility Candidates: $\Gamma \vdash M \in \mathcal{R}_A$)

- $\Gamma \vdash M \in \mathcal{R}_{\text{unit}}$ iff $\Gamma \vdash M : \text{unit} \in \text{SN}$
- $\Gamma \vdash M \in \mathcal{R}_{A \Rightarrow B}$ iff $\Gamma \vdash M : A \Rightarrow B$ and
  
  for all $N, \Delta$ such that $\Gamma \leq_\rho \Delta$,
  
  if $\Delta \vdash N \in \mathcal{R}_A$ then $\Delta \vdash ([\rho]M)N \in \mathcal{R}_B$.

- Contexts arise naturally.
- They are necessary!
- The definition scales to dependently typed setting and stating properties about type-directed equivalence of lambda-terms.
Definition (Reducibility Candidates: $\Gamma \vdash M \in R_A$)

- $\Gamma \vdash M \in R_{\text{unit}}$ iff $\Gamma \vdash M : \text{unit} \in SN$
- $\Gamma \vdash M \in R_{A \Rightarrow B}$ iff $\Gamma \vdash M : A \Rightarrow B$ and
  
  for all $N, \Delta$ such that $\Gamma \leq_{\rho} \Delta$,
  
  if $\Delta \vdash N \in R_A$ then $\Delta \vdash ([\rho]M)N \in R_B$.

- Contexts arise naturally.
- They are necessary!
- The definition scales to dependently typed setting and stating properties about type-directed equivalence of lambda-terms.

*Do we really need the weakening substitution $\rho$?*
Definition (Reducibility Candidates: $\Gamma \vdash M \in R_A$)

- $\Gamma \vdash M \in R_{\text{unit}}$ iff $\Gamma \vdash M : \text{unit} \in SN$
- $\Gamma \vdash M \in R_{A\Rightarrow B}$ iff $\Gamma \vdash M : A \Rightarrow B$ and
  
  for all $N, \Delta$ such that $\Gamma \leq_\rho \Delta$,
  
  if $\Delta \vdash N \in R_A$ then $\Delta \vdash ([\rho]M)N \in R_B$.

- Contexts arise naturally.
- They are necessary!
- The definition scales to dependently typed setting and stating properties about type-directed equivalence of lambda-terms.

*Do we really need to model terms in a “local” context and use Kripke-style context extensions?*
Challenge 2: Strong normalization for simply typed \(\lambda\)-calculus

**CR 1:** If \(\Gamma \vdash M \in \mathcal{R}_A\) then \(\Gamma \vdash M : A \in SN\).

**CR 2:** If \(\Gamma \vdash R : A \in SNe\) then \(\Gamma \vdash R \in \mathcal{R}_A\).

**CR 3:** If \(\Gamma \vdash M \rightarrow_{SN} M' : A\) and \(\Gamma \vdash M' \in \mathcal{R}_A\), then \(\Gamma \vdash M \in \mathcal{R}_A\).

**Main fundamental lemma:**

If \(\Gamma \vdash M : A\) and \(\Gamma' \vdash \sigma \in \mathcal{R}_\Gamma\) then \(\Gamma' \vdash [\sigma]M \in \mathcal{R}_A\).
Challenges in the Proof(s)

- Definitions use well-typed terms
- Stratified definitions for reducibility candidates (not strictly positive!)
- Simultaneous substitutions and weakenings
- Basic infrastructure
  - Substitution properties about terms
  - Weakening and Strengthening of type-directed reductions
  - Weakening, Exchange, and Strengthening for typing
  - Weakening, Anti-weakening for strongly normalizing terms
  - Weakening for reducibility candidates
- Induction principles
Towards solving the challenge problems
Beluga: Programming and Proof Environment

- Below the surface: Support for key concepts based on Contextual LF
  [TOCL’08, POPL’08, LFMTP’13, ESOP’17, ...]

- Above the surface: (Co)Inductive Proofs
  as (Co)Recursive Programs using (Co)pattern Matching
  [POPL’08, IJCAR’10, POPL’12, POPL’13, CADE’15, ICFP’16, ...]
Demo
• Use HOAS to characterize simply typed terms
• Define SN inductively
• Use stratified definition for reducibility
• Extension to disjoint sums.
Lessons Learned – The Good, the Bad and the Ugly

✓ HOAS is great to model binding structure!
✓ Built-in support for substitutions and weakening is very useful!
✓ Take advantage of dependent types to model intrinsically typed terms, typed-reductions, typed SN, etc.
✓ Compact (256 lines total)
✓ Great to investigate and motivate extensions, first-class weakenings, to the theory of simultaneous substitutions Unification in the presence of renamings.
✓ Great to find bugs and make system more robust. Particularly coverage and termination checking
✗ Interactive proof development mode clearly needs work.
✗ No proof automation – know what you want to do.
Other Mechanizations:  
**Agda [A. Abel] and Coq [S. Schäfer, K. Stark]**

Common set up:

- Use well-typed de Bruijn encoding for simply typed terms
- Model typed reduction
- Weakens (renamings) are functions $\text{Nat} \rightarrow \text{Nat}$

Where they differ at the moment ....

- Coq: **Accessibility characterization of SN**
- Agda: **Inductive def. of SN**
✓ Everything could be proven "as expected"

✓ Substitution lemmas
  - Coq (Well-scoped): Autosubst
  - Coq (Well-typed): (≈ 180 lines of code) (proof scripts)
  - Agda (Well-typed): (≈ 270 lines of code) (proof terms)

✗ Still need to work with de Bruijn encodings

✗ Need to apply equational theory of substitution
  can be partially automated using tactics in Coq
Isn’t proving strong normalization in a proof assistant an old hat?
T. Altenkirch [TLCA’93] : SN for System F
C. Coquand [1999] : NbE for $\lambda\sigma$
S. Berghofer [TYPES 2004] : WN for STL

Characteristic Features:

- Terms are not well-scoped or well-typed
- Candidate relation is untyped and does not enforce well-scoped terms

$\implies$ does not scale to typed-directed evaluation or equivalence
$\implies$ today we may have better techniques to structure proofs (inductive def. of SN)
Beyond strong normalization ...
Additional Challenge Problems

- Weak normalization
  (good starting point) [LFMTP’13, MSCS’17] ✓

- Type-directed algorithmic equality
  (Tutorial by K. Crary in ATPL; similar issues as in strong normalization with typed reductions) [LFMTP’15, MSCS’17] ✓

- Adding $\eta$-expansion

- Normalization of System F
  (excellent suggestion) ✗

Let’s systematically compare different mechanization.
Benchmarks can be great!
A Call for Action

- Complete the challenge

- Let’s stick to the given set up
  in particular the inductive def. of SN ...

  https://github.com/andreasabel/strong-normalization/
  blob/master/sn-proof/sn-proof.pdf

  ✓ makes it easier to compare mechanization
  ✓ it’s good for you :-)

- Be part of formulating and tackling the challenge and building a repository for challenge problems
Let’s get started... talk to me for the challenge problem set up and questions to keep in mind.

Thank you!
If you submit a solution, please answer the following set of questions to help us compare and evaluate different mechanizations.
Questions About the Set Up

- How are bindings, substitutions, and necessary infrastructure represented? How big is your initial set up? – If you use libraries, explain briefly what they are.
- How are well-typed terms modelled?
- How are Kripke-style context extensions modelled?
- How does the formalization deal with renamings / weakenings?
Questions About the Proof Development

- How does it compare to the proof given in the online tutorial?
- Were there any additional lemmas required besides the ones given in the tutorial?
- How straightforward was it to extend the language to unit and disjoint sums? Did anything in the set-up needed to be changed?
Questions About General Lessons

• Did you find solving the problem interesting? Did it expose you to a new perspective on logical relations proofs?
• Did solving this problem expose any issues with the system you were working with? Did it inspire extensions?
• Do you have any general lessons / take-aways?
Let’s get started... We are looking forward seeing your solutions.