How to use formal proofs to detect bugs

Alexandre Miquel

ENTROPY 2018 – December 26th – Villeneuve d'Ascq
In many situations, the correctness of a high-level program depends on the correctness of its low-level components, whose implementation may be unknown.

The correctness of the high-level program is thus pre-conditioned by the correctness of its low-level components:

\[ \text{Correct(low-level components)} \implies \text{Correct(high-level program)} \]

The formal proof of correctness can only assume the correctness of the low-level components (as axioms).

But what happens when the high-level program does not meet its specification, i.e., when \( \neg \text{Correct(high-level program)} \)?

Surely, one of the low-level components is defective...

**Problem:** how to determine which low-level component is defective, and for which set of parameters?
The modus tollens of program certification

- $U_1, \ldots, U_\ell = \text{specification of low-level components}$ (Π$^0_1$-formulas)
- $V = \text{specification of high-level program}$ (Π$^0_1$-formula)

Formal proof

\[
\begin{align*}
U_1, \ldots, U_\ell & \vdash V \\
\vdash \neg V & \text{ for some } i \\
\vdash \neg U_i & \text{ bug report}
\end{align*}
\]

Two possible solutions:

1. **Dynamic debugging:** Run the high-level program in a sandbox, checking the correctness of each call to a low-level component (using the appropriate assertions)

2. **Static debugging:** Use the formal proof to guide the search for the defective low-level component (this talk)
The problem of the experimental modus tollens


Consider two falsifiable theories $U, V$ such that:
- $U \Rightarrow V$ is *mathematically* provable
- $V$ is *experimentally* falsifiable

Can we deduce from this an *experimental falsification* of $V$?

**Experimental modus tollens**

$$
\text{math } \vdash U \Rightarrow V \quad \text{exp } \models \neg V \\
\text{exp } \models \neg U
$$

- M., 2007: *The experimental effectiveness of mathematical proof*
  - Solving Popper’s problem using techniques of *classical realizability*
  - The same technique could be used for debugging (obs. by V. Balat)
Traditionally, proof-assistants based on the Curry-Howard correspondence (Coq, Agda, etc.) are based on constructive logics, where proofs are interpreted as purely functional programs:

\[ f : A \xrightarrow{\text{flow of execution}} B \]

Execution goes from \( A \) to \( B \); there is now way to go the other way around.

But since the 90’s, we know (Griffin, Krivine, Parigot, ...) how to interpret classical proofs within Curry-Howard, by the means of functional programs with continuations (backtrack):

\[ f : A \xrightarrow{\text{normal flow of execution}} B \]

We can retrieve (negative) information on \( A \) from (negative) information on \( B \).
What is realizability?

- Realizability is a **generalized form of typing** that is based on the computational behavior of programs rather than on typing rules.

- **Example:** Why does $\lambda x . x$ have type $\text{nat} \rightarrow \text{nat}$?

<table>
<thead>
<tr>
<th><strong>Typing</strong> ($t : A$)</th>
<th><strong>Realizability</strong> ($t \vDash A$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : \text{nat} \vdash x : \text{nat}$</td>
<td>for all $n \in \text{nat}$ $\vdash (\lambda x . x) , n \succ n \in \text{nat}$</td>
</tr>
<tr>
<td>$\vdash \lambda x . x : \text{nat} \rightarrow \text{nat}$</td>
<td></td>
</tr>
</tbody>
</table>

- Syntactic analysis of terms
- Does not look at computation
- At least semi-decidable
- Simple justification: derivation

- Computational analysis of terms
- Does not look at the syntax
- Strongly undecidable
- External justification (proof)

**Adequacy:** Each term of type $A$ is also a realizer of $A$
What is realizability?

- Adequacy is the property that ensures that all well-typed programs are computationally correct.
  
  As a matter of fact, all strong normalization proofs (from $\lambda \rightarrow$ to CIC) rely on a realizability model + a proof of adequacy.

- Typing appears as a decidable approximation of realizability.

But historically, realizability was invented for logic.

- Kleene ’45. *On the interpretation of intuitionistic number theory*
  
  Realizability as a semantics for intuitionistic provability.

- In the mid-90’s, Krivine reformulated the principles of realizability to make them compatible with the correspondence between classical reasoning and control operators discovered by Griffin’90:

  $$\text{call/cc} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$$

  (Peirce’s law)

  $$\Rightarrow \text{classical realizability}$$
Plan

1. Introduction

2. A primer of classical realizability

3. The modus tollens of program certification

4. Classical realizability and forcing
<table>
<thead>
<tr>
<th>Introduction</th>
<th>Realizability</th>
<th>Modus tollens</th>
<th>Classical realizability and forcing</th>
</tr>
</thead>
</table>

**Plan**

1. **Introduction**

2. **A primer of classical realizability**

3. **The modus tollens of program certification**

4. **Classical realizability and forcing**
What is classical realizability? [Krivine ’94, ’00, ’03, ’09, ’11, ’12, …]

- A complete reformulation of the principles of Kleene realizability to make them compatible with classical reasoning

Reformulation ≠ Extension. Recall that by design, Kleene realizability and all its extensions (e.g. with PCAs) are incompatible with classical logic

- Based on the connection between classical reasoning and control operators discovered by Griffin ’90:

  \[
  \text{call/cc} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A
  \]  
  (Peirce’s law)

- Initially designed for PA2, but extends to
  - Higher-order arithmetic (PAω)
  - Zermelo-Fraenkel set theory (ZF)
  - The calculus of (inductive) constructions (CCω, CIC)
  - Interprets the Axiom of Dependent Choices (DC)

- Deep connections with Cohen forcing
Main idea underlying classical realizability

- Traditionally, classical proofs are turned into intuitionistic proofs (via some translation/interpretation from LK into LJ) before being interpreted as purely functional programs.

- Rather than restricting to LJ a priori, interpret classical proofs directly, using functional programs with control operators.
Control operators give to the programs the ability to capture their evaluation context (the “continuation”), so that they can backtrack when something goes wrong.

Allow programs to use the method of trial and error.

Technically: Extend the pure λ-calculus with a new binder $Ck . t$ that captures the current continuation in the bound variable $k$:

$$
\frac{k : A \Rightarrow B \vdash t : A}{\vdash Ck . t : A}
$$

The variable $k : A \Rightarrow B$ captures the current $A$-continuation, that is: the evaluation context asking for a value of type $A$.

When applied to an object of type $A$ (the “new answer”), the $A$-continuation $k : A \Rightarrow B$ restores the evaluation context that was saved in $k$, with the new answer of type $A$. The current context is aborted, hence $B$ can be any type (typically: $B \equiv \bot$).
In practice, the binder $Ck.t$ is implemented from the control operator $\alpha$ ("call/cc"), letting $Ck.t \equiv \alpha(\lambda k.t)$.

We have: $\alpha : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ (Peirce’s law)

**Question:** $A \lor \neg A$ ?

**Answer:** $EM \equiv \alpha(\lambda k. right(\lambda x. k(left x))) : A \lor \neg A$

where $left : \forall X \forall Y (X \Rightarrow X \lor Y)$

$right : \forall X \forall Y (Y \Rightarrow X \lor Y)$

Note that $EM$ does not even need to know the formula $A$! It is actually polymorphic in $A$: $EM : \forall X (X \lor \neg X)$
Krivine’s $\lambda_c$-calculus

Terms, stacks and processes

Terms

$t, u ::= \lambda x \cdot t | tu | \alpha | \cdots | k_\pi$

Stacks

$\pi ::= \emptyset | u \cdot \pi$

Processes

$p ::= t \star \pi$

Krivine’s Abstract Machine (KAM)

(Push) $tu \star \pi \succeq t \star u \cdot \pi$

(Grab) $\lambda x \cdot t \star u \cdot \pi \succeq t\{x := u\} \star \pi$

(Save) $\alpha \star t \cdot \pi \succeq t \star k_\pi \cdot \pi$

(Restore) $k_\pi \star t \cdot \pi' \succeq t \star \pi'$

\cdots
Classical realizability: principles

- **Intuitions:**
  - term = “defender”
  - stack = “attacker”
  - process = “contradiction”  
    (slogan: never trust a classical realizer!)

- Classical realizability model parameterized by:
  - a model $\mathcal{M}$ of the input theory (PA2, PA$\omega$, ZF, etc.)
  - a pole $\perp \subseteq \Lambda \ast \Pi$ (closed under anti-evaluation)

- Each formula $A$ is interpreted as two sets:
  - **Falsity value** $\|A\| \subseteq \Pi$ (primitive, defined by induction on $A$)
  - **Truth value** $|A| \subseteq \Lambda$ (defined by orthogonality from $\|A\|$)
Construction of the realizability model

- Realizability model $\mathcal{M}_\bot$ parameterized by:
  - a ground model $\mathcal{M}$ (of PA2, PA$\omega$, ZF, etc.)
  - a pole $\bot \subseteq \Lambda \star \Pi$ (closed under anti-evaluation)

- Falsity value $\|A\| \subseteq \Pi$ defined by induction on $A$

\[
\|A \Rightarrow B\| = \|A\| \cdot \|B\| = \{t \cdot \pi : t \in \|A\|, \pi \in \|B\|\}
\]

\[
\|\forall x^\tau A(x)\| = \bigcup_{v \in \mathcal{M}_\tau} \|A(v)\|
\]

- Truth value $|A| \subseteq \Lambda$ defined by orthogonality:

\[
|A| = \|A\|_{\bot} = \{t \in \Lambda : \forall \pi \in \|A\|, \ t \star \pi \in \bot\}
\]

- Realizability relation: $t \vdash A \equiv t \in |A|$ (depends on the pole $\bot$)
Adequacy

Use proof-like terms as Curry-style proof terms:

**Typing rules**

\[
\begin{align*}
\Gamma &\vdash x : A \\
\Gamma, x : A &\vdash t : B \\ 
\Gamma &\vdash \lambda x \cdot t : A \Rightarrow B \\
\Gamma &\vdash t : A \Rightarrow B, \Gamma &\vdash u : A \\
\Gamma &\vdash tu : B \\
\Gamma &\vdash t : A \\
\Gamma &\vdash t : \forall x^\tau A \\
\Gamma &\vdash t : \forall^x A \{ x^\tau := M^\tau \} \\
\Gamma &\vdash \alpha : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A
\end{align*}
\]

**Theorem (Adequacy)**

If \( \vdash t : A \), then \( t \vdash A \) (in all realizability models)
Other results

- **Dependent choices** (DC) can be realized using the extra instruction:
  \[ \text{quote} \star t \cdot u \cdot \pi \succ u \star \bar{n} \cdot \pi \]
  where \( n = \lceil t \rceil \) is the Gödel code of term \( t \)

- Direct witness extraction techniques for \( \Sigma^0_1 \) and \( \Pi^0_2 \)-formulas
  (equivalent to Friedman’s trick via the Lafont-Reus-Streicher translation)

- Game-theoretic techniques to solve the specification problem:
  
  *Given a formula \( A\), characterize its universal realizers from their computational behavior*

- Study of some interesting realizability models \( \mathcal{M}_\bot \):
  
  *Which interesting formulas are realized in \( \mathcal{M}_\bot \) that were not already true in the ground model \( \mathcal{M} \)?
Plan

1. Introduction
2. A primer of classical realizability
3. The modus tollens of program certification
4. Classical realizability and forcing
Implementing the experimental modus tollens

- Classical realizability can be defined in
  - Classical second-order arithmetic (PA2) [Krivine]
  - Classical higher-order arithmetic (PAω) [Raffalli-Ruyer’08]
  - The calculus of constructions with universes (CCω) [M.’07]
  - The calculus of inductive constructions (CIC) [M.’09, unpublished]
  - Zermelo-Fraenkel set theory (ZF) [Krivine’01]

Note that in CCω/CIC, excluded middle only lives in Prop

- In all the above frameworks, classical realizability interprets excluded middle + axiom of dependent choices (DC) [Krivine’03]

- For simplicity, we shall now work in (an extension of) PA2
  But the same methodology also works in PAω, ZF, CCω or CIC

- Note: The construction presented here is a variant of the one given in [M.’07]
The language of experimental arithmetic (xPA2)

Symbols

\[x, y, z, \ldots\] Variables of individuals (integers)
\[X, Y, Z, \ldots\] Predicate variables (of all arities \(k \geq 0\))
\[h, h', h_1, \ldots\] Experimental functions
\[f, f', f_1, \ldots\] User-defined functions (programs) \((+, \times, \uparrow, \ldots)\)

Syntax

**Arith. expr.** \[e, e' ::= x \mid h(e_1, \ldots, e_k) \mid f(e_1, \ldots, e_k)\]

**Formulas** \[A, B, C ::= X(e_1, \ldots, e_k) \mid A \Rightarrow B \mid \forall x \ B \mid \forall X \ B\]

- Predicate variable \(\approx\) set of integers \(\approx\) real number
- 2nd-order arithmetic \(\approx\) Analysis
Abbreviations

- Logical system based on $\Rightarrow$ and $\forall$ (1st and 2nd order)

**Definition of other connectives**

(Absurdity) $\perp := \forall Z \, Z$

(Triviality) $\top := \forall Z \, (Z \Rightarrow Z)$

(Negation) $\neg A := A \Rightarrow \perp$

(Conjunction) $A \land B := \forall Z \, ((A \Rightarrow B \Rightarrow Z) \Rightarrow Z)$

(Disjunction) $A \lor B := \forall Z \, ((A \Rightarrow Z) \Rightarrow (B \Rightarrow Z) \Rightarrow Z)$

(Existence-1) $\exists x \, A(x) := \forall Z \, (\forall x \, (A(x) \Rightarrow Z) \Rightarrow Z)$

(Existence-2) $\exists X \, A(X) := \forall Z \, (\forall x \, (A(X) \Rightarrow Z) \Rightarrow Z)$

(Equality) $x = y := \forall Z \, (Z(x) \Rightarrow Z(y))$

- $1 := s(0)$, $2 := s(1)$, $3 := s(2)$, etc.
Deduction in xPA2: inference rules

(Axiom) \[ \Gamma \vdash A \quad (A \in \Gamma \cup \mathcal{A}) \]

(\Rightarrow) \[ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \]

(\forall^1) \[ \frac{\Gamma \vdash A}{\Gamma \vdash \forall x A} \quad (x \notin \text{FV}(\Gamma)) \quad \frac{\Gamma \vdash \forall x A}{\Gamma \vdash A\{x := e\}} \]

(\forall^2) \[ \frac{\Gamma \vdash A}{\Gamma \vdash \forall X A} \quad (X \notin \text{FV}(\Gamma)) \quad \frac{\Gamma \vdash \forall X A}{\Gamma \vdash A\{X(x_1, \ldots, x_k) := B\}} \]

(Peirce’s law) \[ \frac{\Gamma \vdash ((A \Rightarrow B) \Rightarrow A)}{\Gamma \vdash A} \]

- \( \mathcal{A} = \) set of axioms of xPA2
- The usual rules of \( \neg, \wedge, \vee, \exists, \ldots \) are derivable from their encodings
Relativization and induction

- In PA2, 1st-order universal quantification $\forall x \ A(x)$ is uniform
  (= ML-style, intersection type)

- To get non-uniform universal quantification (Coq-style, $\Pi$-type, ...) we relativize it to the set $\mathbb{N}$ of Dedekind numerals

  $$(\forall x \in \mathbb{N}) \ A(x) \equiv \ \forall x \ (x \in \mathbb{N} \Rightarrow A(x)) \quad (\equiv \ \forall x : \mathbb{N}. \ A(x))$$

where

  $$x \in \mathbb{N} \equiv \ \forall Z \ (Z(0) \Rightarrow \ \forall y \ (Z(y) \Rightarrow Z(s(y)))) \Rightarrow Z(x))$$

- Thanks to this trick, the relativized induction scheme is derivable:

  $$A(0) \Rightarrow (\forall x \in \mathbb{N}) \ (A(x) \Rightarrow A(s(x))) \Rightarrow (\forall x \in \mathbb{N}) \ A(x)$$
The axioms (\(\mathcal{A}\)) of xPA2

- Peano axioms:
  \[ \forall x \ \forall y \ (s(x) = s(y) \Rightarrow x = y), \quad \forall x \ \neg (s(x) = 0) \]

  Remember that (relativized) induction is derivable

- The axioms expressing the totality of experimental functions:
  \[ (\forall x_1, \ldots, x_k \in \mathbb{N}) \ h(x_1, \ldots, x_k) \in \mathbb{N} \]

- Defining equalities of user-defined functions:
  \[ 0 + y = y \quad 0 \times y = 0 \]
  \[ s(x) + y = s(x + y) \quad s(x) \times y = (x \times y) + y \]

  etc.

We only allow primitive recursive definitions (possibly using experimental functions), to ensure the totality of all user-defined functions.
The axioms ($\mathcal{A}$) of xPA2

- The experimental functions ("$h$") are specified by the means of $\Pi^0_1$-axioms $U_1, \ldots, U_\ell$, of the form

$$U_i \equiv (\forall x_1, \ldots, x_{k_i} \in \mathbb{N}) \ e_i(x_1, \ldots, x_{k_i}) = 0$$

where $e_i(x_1, \ldots, x_{k_i})$ may contain experimental functions ("$h$")

- To sum up:

$$\text{xPA2} = \text{deduction rules of 2nd-order logic}$$
$$+ \text{Peano axioms}$$
$$+ \text{totality of experimental functions ("$h$")}$$
$$+ \text{defining equations of used-defined functions ("$f$")}$$
$$+ \text{specification } U_1, \ldots, U_\ell \text{ of experimental functions}$$
The experimental modus tollens

- In this framework, we assume that the correctness of the high-level program is given in the form of a $\Pi^0_1$-formula

$$V \equiv (\forall x_1, \ldots, x_k \in \mathbb{N}) \ e(x_1, \ldots, x_k) = 0$$

where $e(x_1, \ldots, x_k)$ may contain experimental functions ("h")

- We assume given:
  - a formal proof (a derivation) $d$ of the formula $V$ (in xPA2)
  - a bug report for $V$, that is: a tuple
    $$(n_1, \ldots, n_k) \in \mathbb{N}^k \quad \text{s.t.} \quad e(n_1, \ldots, n_k) \neq 0$$

- From these ingredients, we want to extract a bug report on some of the hypotheses $U_1, \ldots, U_\ell$, that is: a tuple
  $$(i, m_1, \ldots, m_{k_i}) \in \mathbb{N}^{k_i+1} \quad \text{s.t.} \quad i \in [1..\ell] \text{ and } e_i(m_1, \ldots, m_{k_i}) \neq 0$$
The language for program extraction

We extend Krivine’s $\lambda_c$-calculus with the following instructions:

- For each experimental function $h$, an instruction $\hat{h}$ that computes the function (calling the actual low-level component or oracle)

- An instruction `stop` with no evaluation rule (used to return the expected bug report)

- For each $i \in [1..\ell]$, an instruction `test_i` that tests the axiom $U_i$ with a given set of parameters $(m_1, \ldots, m_{k_i}) \in \mathbb{N}^{k_i}$:

  \[
  \text{test}_i \star \vec{m}_1 \cdots \vec{m}_{k_i} \cdot t \cdot \pi \quad \triangleright \quad \begin{cases} 
  t \star \pi & \text{if } e_i(m_1, \ldots, m_{k_i}) = 0 \\
  \text{stop} \star i \cdot \vec{m}_1 \cdots \vec{m}_{k_i} \cdot \Diamond & \text{otherwise}
  \end{cases}
  \]
Program extraction

From the derivation $d$ of the formula $V$ (in $\times$PA2), we extract a $\lambda_c$-term $d^*$ as follows:

- Logical constructions are extracted as usual, according to the Curry-Howard correspondence ($\Rightarrow$-intro. by $\lambda$, $\Rightarrow$-elim. by app., ...)
- Peirce’s law is extracted as $\alpha$
- Peano axioms are extracted as $1 \equiv \lambda x . x$ and $\lambda y . y \ 1$, respectively
- Definitional axioms of user-defined functions (possibly involving $h$’s) are extracted as $1 \equiv \lambda x . x$
- Totality axiom for each experimental function $h$ is extracted as $\hat{h}$
- Each axiom $U_i$ ($1 \leq i \leq \ell$) is extracted as the $\lambda_c$-term $M_i \ tests_i$ (where $M_i$ is a storage operator of arity $k_i$, just for technical reasons)
Correctness of the extracted program

**Theorem**

If \( d : V \) (in xPA2) and \((n_1, \ldots, n_k)\) is a bug report for \( V \), then:

\[
d^* \star \bar{n}_1 \cdots \bar{n}_k \cdot I \cdot \triangleright \text{stop} \star \bar{i} \cdot \bar{m}_1 \cdots \bar{m}_{k_i} \cdot \triangleright
\]

for some \( i \in [1..\ell] \) and \((m_1, \ldots, m_{k_i})\) \( \in \mathbb{N}^{k_i} \) s.t. \( e_i(m_1, \ldots, m_{k_i}) \neq 0 \)

**Proof.**

- We work in the pole \( \perp \) formed by all processes \( p \succ \text{stop} \star \bar{i} \cdot \bar{m}_1 \cdots \bar{m}_{k_i} \cdot \triangleright \)
  for some \( i \in [1..\ell] \) and \((m_1, \ldots, m_{k_i})\) \( \in \mathbb{N}^{k_i} \) s.t. \( e_i(m_1, \ldots, m_{k_i}) \neq 0 \).

- We check by induction on \( d \) that each extracted term is a realizer of its type *in this particular pole* \( \perp \). The only interesting case is the case of instruction \( \text{test}_i \), that appears to be a realizer of \( U_i \) due to the particular definition of \( \perp \).

- We conclude by observing that \( d^* \bar{n}_1 \cdots \bar{n}_k \models \perp \)
To sum up

- Given
  - a formal proof of $U_1, \ldots, U_\ell \vdash V$
  - a bug report on $V$

  we have seen how to construct a program that determines which of $U_1, \ldots, U_\ell$ is wrong, and for which set of parameters

- Note that the formulas $U_1, \ldots, U_\ell$ and $V$ must be $\Pi^0_1$ (falsifiable, according to Popper’s terminology)

- We have presented the method in PA2, but the same technique also works in PA\(\omega\), ZF, CC\(\omega\) or CIC (cf next slide)

- Moreover, the extraction procedure allows many optimizations:
  - Removing proofs of $\Pi^0_1$-formulas (e.g. commutativity of $+$)
  - Using binary (arbitrary precision) natural numbers
Applying the same technique in Coq

Relies on the existence of a classical realizability model for CC$^\omega$/CLIC
+ extraction function $M \mapsto M^*$ from CC$^\omega$/CLIC to $\lambda_c$ [M.'07]

**Adequacy:** If $M : T$, then $M^* \vdash_{[T]} [M]$ (dependent realizability)

- Experimental functions $h$ are of the form:
  \[ \text{Axiom } h : \prod \vec{x} : \vec{T}. S \]
  where $\vec{T}, S$ are purely algebraic datatypes (no $\rightarrow$/Π)

- No need to introduce user-defined functions: we can use Coq programming language instead (no restriction)

- Formulas $U_1, \ldots, U_\ell$ and $V$ still need to be $\prod^1_1$:
  \[ U_i/V := \prod \vec{x} : \vec{T}. M_1 = M_2 \]
  where $\vec{T}$ are purely algebraic datatypes (no $\rightarrow$/Π)
Plan

1. Introduction
2. A primer of classical realizability
3. The modus tollens of program certification
4. Classical realizability and forcing
What is forcing?

- A technique invented by Cohen ('63) to prove the independence of the continuum hypothesis (CH) w.r.t. ZFC:

  \[ \text{Cons}(\text{ZFC} + \neg \text{CH}), \quad \text{or:} \quad \text{ZFC} \nvdash \text{CH} \]

Gödel ('38) proved \( \text{Cons}(\text{ZFC} + \text{CH}) \) (or: \( \text{ZFC} \nvdash \neg \text{CH} \)) introducing constructible sets

- Forcing can be understood as
  - A technique to transform models of ZFC, using generic sets
  - A way to build Boolean-valued models [Scott, Solovay, Vopěnka]
  - A translation of formulas and proofs (proof theorist’s point of view)

- Now standard item of the toolbox of model theorists
  - Used to prove the consistency/independence of many axioms
How does forcing work?

Exploit the under-specification of the power set $\mathcal{P}(X)$ (when $X$ is infinite)
Forcing: the proof-theoretic point of view

- Works in strong classical theories (PA3, PAω, ZF, ZFC)
- Parameterized by a poset \( (P, \leq) \) of conditions (expressed in the theory)
- Forcing translation: \( A \leftrightarrow p \text{IF} A \) \( (p \in P) \)

General properties

1. \( \vdash A \) entails \( \vdash (p \text{IF} A) \) (for all conditions \( p \in P \))
2. But \( \vdash (p \text{IF} A) \) for more formulas \( A \) (depending on \( P \))
3. \( \vdash p \text{IF} \bot \) (consistency)

Remark: Forcing commutes with \( \bot, \top, \land \) and \( \forall \), but not with \( \Rightarrow, \neg, \lor, \exists \):

\[
\begin{align*}
(p \text{IF} \bot) & \iff \bot \\
(p \text{IF} \top) & \iff \top \\
(p \text{IF} A \land B) & \iff (p \text{IF} A) \land (p \text{IF} B) \\
(p \text{IF} \forall x A) & \iff \forall x (p \text{IF} A)
\end{align*}
\]
Classical realizability and Cohen forcing

- **Forcing in classical realizability**
  - Introduce realizability algebras, generalizing the $\lambda_c$-calculus
  - Discover the program transformation underlying forcing
  - Extend iterated forcing to classical realizability
  - Show how to force the existence of a well-ordering over $\mathbb{R}$ (while keeping evaluation deterministic)

- **Computational analysis of forcing**
  - Hard-wire the program transformation into the abstract machine

**Underlying methodology**

Translation of formulas & proofs $\rightsquigarrow$ Classical program transformation $\rightsquigarrow$ New abstract machine (no transformation)
A glimpse of the forcing translation in PA_\omega

- Translating sorts: \( \tau \mapsto \tau^* \)
  
  \[
  i^* \equiv i \quad o^* \equiv \kappa \to o \quad (\tau \to \sigma)^* \equiv \tau^* \to \sigma^*
  \]

- Translating higher-order terms: \( M \mapsto M^* \)
  
  \[
  (x^\tau)^* \equiv x^{\tau^*} \quad 0^* \equiv 0 \\
  (\lambda x^\tau \cdot M)^* \equiv \lambda x^{\tau^*} \cdot M^* \quad s^* \equiv s \\
  (MN)^* \equiv M^* N^* \quad rec^*_\tau \equiv rec_{\tau^*} \\
  (A \Rightarrow B)^* \equiv \lambda r^\kappa \cdot \forall q^\kappa \forall r'^\kappa (r = qr' \mapsto (\forall s^\kappa (C[qs] \Rightarrow A^* s) \Rightarrow B^* r')) \\
  (\forall x^\tau A)^* \equiv \lambda r^\kappa \cdot \forall x^{\tau^*} A^* r \\
  (M_1 = M_2 \mapsto A)^* \equiv \lambda r^\kappa \cdot M_1^* = M_2^* \mapsto (A^* r)
  \]

- Forcing propositions (terms of type \( o \)):
  
  \[
  p \text{ IF } A \equiv \forall r^\kappa (C[pr] \Rightarrow A^* r)
  \]
A glimpse of the underlying program transformation

- The program transformation is parameterized by combinators $\alpha_*, \alpha_1, \ldots, \alpha_{15}$ expressing that $(P, \leq)$ is a (suitable) poset

\[
\begin{align*}
x^* & \equiv x \\
(\alpha c)^* & \equiv \lambda x. \alpha (\lambda k. x (\alpha_{14} c) (\gamma_4 k)) \\
(t u)^* & \equiv \gamma_3 t^* u^* \\
(\lambda x. t)^* & \equiv \gamma_1 (\lambda x. t^* \{x \equiv \beta_4 x\} \{x_i \equiv \beta_3 x_i\}_{i=1}^n)
\end{align*}
\]

\[
\begin{align*}
\gamma_4 & \equiv \lambda x y. x (\alpha_{15} c) \\
\gamma_3 & \equiv \lambda x y. x (\alpha_{11} c) y \\
\gamma_1 & \equiv \lambda x y. x y (\alpha_6 c) \\
\beta_3 & \equiv \lambda x. x (\alpha_9 c) \\
\beta_4 & \equiv \lambda x. x (\alpha_{10} c)
\end{align*}
\]

Soundness

If \( x_1 : A_1, \ldots, x_n : A_n \vdash t : B \)
then \( x_1 : (p \text{ IF } A_1), \ldots, x_n : (p \text{ IF } A_n) \vdash t^* : (p \text{ IF } B) \)
Krivine Forcing Abstract Machine (KFAM)

<table>
<thead>
<tr>
<th>Terms</th>
<th>$t, u ::= x \mid \lambda x \cdot t \mid tu \mid c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environments</td>
<td>$e ::= \emptyset \mid e; x := c$</td>
</tr>
<tr>
<td>Closures</td>
<td>$c ::= t[e] \mid k_\pi \mid t[e]^* \mid k_\pi^*$</td>
</tr>
<tr>
<td>Stacks</td>
<td>$\pi ::= \diamond \mid c \cdot \pi$ forcing closures</td>
</tr>
</tbody>
</table>

- **Real mode:**
  
  - $x[e; y := c] \star \pi \triangleright x[e] \star \pi \quad (y \neq x)$
  - $x[e; x := c] \star \pi \triangleright c \star \pi$
  - $(\lambda x \cdot t)[e] \star c \cdot \pi \triangleright t[e; x := c] \star \pi$
  - $(tu)[e] \star \pi \triangleright t[e] \star u[e] \cdot \pi$
  - $\alpha[e] \star uc \cdot \pi \triangleright c \star k_\pi \cdot \pi$
  - $k_\pi \star c \cdot \pi' \triangleright c \star \pi$

- **Forcing mode:**
  
  - $x[e; y := c]^* \star c_0 \cdot \pi \triangleright x[e]^* \star \alpha_9 c_0 \cdot \pi \quad (y \neq x)$
  - $x[e; x := c]^* \star c_0 \cdot \pi \triangleright c \star \alpha_10 c_0 \cdot \pi$
  - $(\lambda x \cdot t)[e]^* \star c_0 \cdot c \cdot \pi \triangleright t[e; x := c]^* \star \alpha_6 c_0 \cdot \pi$
  - $(tu)[e]^* \star c_0 \cdot \pi \triangleright t[e]^* \star \alpha_11 c_0 \cdot u[e]^* \cdot \pi$
  - $\alpha[e]^* \star c_0 \cdot c \cdot \pi \triangleright c \star \alpha_{14} c_0 \cdot k_\pi^* \cdot \pi$
  - $k_\pi^* \star c_0 \cdot c \cdot \pi' \triangleright c \star \alpha_{15} c_0 \cdot \pi$