How to use formal proofs to detect bugs

Alexandre Miquel

UNIVERSIDAD DE LA REPUBLICA URUGUAY

FACULTAD DE INGENIERIA

ENTROPY 2018 – December 26th – Villeneuve d'Ascq

The problem

- 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:

  $$\text{Correct(low-level components)} \Rightarrow \text{Correct(high-level program)}$$

  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

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

$$\frac{\overbrace{U_1, \ldots, U_\ell \vdash V}^{\text{formal proof}} \qquad \overbrace{\models \neg V}^{\text{bug report}}}{\underbrace{\models \neg U_i}_{\text{bug report}} \ \text{ for some } i} \quad ?$$

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

- K. R. Popper, 1934: *The Logic of Scientific Discovery* (ch. 3, § 18)

  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

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

- 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)

Why using classical logic?

- 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):

$$\begin{tikzcd} f \ : \ A \arrow[r, "\text{normal flow of execution}"] & B \arrow[l, bend left, dotted, "\text{backtrack}"] \end{tikzcd}$$

We can retrieve (negative) information on A from (negative) information on B

What is realizability? (1/2)

- 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 $\mathrm{nat} \to \mathrm{nat}$?

Typing $(t : A)$	**Realizability** $(t \Vdash A)$
$$\dfrac{\dfrac{}{x : \mathrm{nat} \ \vdash \ x : \mathrm{nat}}}{\vdash \ \lambda x\,.\,x \ : \ \mathrm{nat} \to \mathrm{nat}}$$	for all $n \in \mathrm{nat}$ $\quad (\lambda x\,.\,x)\,n \ \succ \ n \in \mathrm{nat}$
• Syntactic analysis of terms	• Computational analysis of terms
• Does not look at computation	• Does not look at the syntax
• At least semi-decidable	• Strongly undecidable
• Simple justification: derivation	• External justification (proof)

Adequacy: Each term of type A is also a realizer of A

What is realizability? (2/2)

- Adequacy is the property that ensures that all well-typed programs are computationally correct

  As a matter of fact, all strong normalization proofs (from $\lambda_{\to}$ 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:

$$\text{call/cc} \;:\; ((A \Rightarrow B) \Rightarrow A) \Rightarrow A \qquad \text{(Peirce's law)}$$

$\Rightarrow$ classical realizability

Plan

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

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 $\neq$ 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:

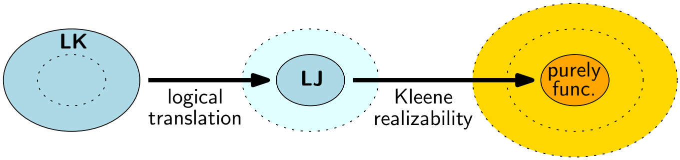
$$\mathrm{call/cc} \;:\; ((A \Rightarrow B) \Rightarrow A) \Rightarrow A \qquad \text{(Peirce's law)}$$

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

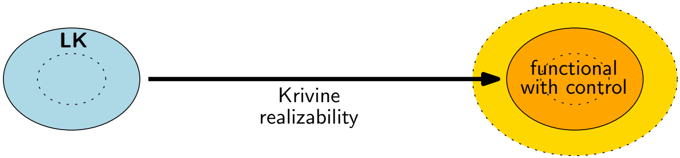
- Deep connections with Cohen forcing

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



Programming with continuations (1/2)

- Control operators give to the programs the ability to capture their evaluation context (the "continuation"), so that they can backtrack when something goes wrong.

  $\rightsquigarrow$ Allow programs to use the method of trial and error.

- **Technically:** Extend the pure λ-calculus with a new binder $\mathcal{C}k\,.\,t$ that captures the current continuation in the bound variable k:

$$\frac{k : A \Rightarrow B \quad \vdash \quad t \ : \ A}{\vdash \ \mathcal{C}k\,.\,t \ : \ A}$$

  - 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: $B \equiv \bot$).

Programming with continuations (2/2)

- In practice, the binder $\mathcal{C}k\,.\,t$ is implemented from the control operator æ ("call/cc"), letting $\mathcal{C}k\,.\,t \equiv \text{æ}\,(\lambda k\,.\,t)$.

  We have: $\text{æ} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ (Peirce's law)

- **Question:** $A \vee \neg A$?

- **Answer:** $\mathbf{EM} \equiv \text{æ}\,(\lambda k\,.\,\mathbf{right}(\lambda x\,.\,k\,(\mathbf{left}\,x))) \quad : \quad A \vee \neg A$

  where
  $$\begin{array}{rcl} \mathbf{left} & : & \forall X\,\forall Y\,(X \Rightarrow X \vee Y) \\ \mathbf{right} & : & \forall X\,\forall Y\,(Y \Rightarrow X \vee Y) \end{array}$$

- Note that **EM** does not even need to know the formula A!

  It is actually polymorphic in A: $\mathbf{EM} \quad : \quad \forall X\,(X \vee \neg X)$

Krivine's λ_c-calculus

Terms, stacks and processes

Terms $\quad t, u \quad ::= \quad \underbrace{\overbrace{x \mid \lambda x\,.\,t \mid tu}^{\lambda\text{-calculus}} \mid \overbrace{\text{œ} \mid \cdots}^{\text{control op.}}}_{\text{proof-like terms}} \mid \underbrace{\mathsf{k}_\pi}_{\text{continuations}}$

Stacks $\quad \pi \quad ::= \quad \diamond \mid u \cdot \pi \qquad (u, \pi \text{ closed})$

Processes $\quad p \quad ::= \quad t \star \pi \qquad (t, \pi \text{ closed})$

Krivine's Abstract Machine (KAM)

(Push)	tu	$\star$	π	$\succ$	t	$\star$	$u \cdot \pi$
(Grab)	$\lambda x\,.\,t$	$\star$	$u \cdot \pi$	$\succ$	$t\{x := u\}$	$\star$	π
(Save)	œ	$\star$	$t \cdot \pi$	$\succ$	t	$\star$	$\mathsf{k}_\pi \cdot \pi$
(Restore)	k_π	$\star$	$t \cdot \pi'$	$\succ$	t	$\star$	π
		$\cdots$				$\cdots$	

Classical realizability: principles

- **Intuitions:**
  - term = "defender"
  - stack = "attacker"
  - process = "contradiction" (slogan: never trust a classical realizer!)
- Classical realizability model parameterized by:
  - a model $\mathscr{M}$ of the input theory (PA2, PAω, ZF, etc.)
  - a pole $\perp\!\!\!\perp \subseteq \Lambda \star \Pi$ (closed under anti-evaluation)
- Each formula A is interpreted as two sets:

Falsity value	$\|A\|$	$\subseteq$	Π	(primitive, defined by induction on A)
Truth value	$\lvert A\rvert$	$\subseteq$	Λ	(defined by orthogonality from $\|A\|$)

Construction of the realizability model

- Realizability model $\mathscr{M}_{\perp\!\!\!\perp}$ parameterized by:
  - a ground model $\mathscr{M}$ (of PA2, PAω, ZF, etc.)
  - a pole $\perp\!\!\!\perp \subseteq \Lambda \star \Pi$ (closed under anti-evaluation)
- Falsity value $\|A\| \subseteq \Pi$ defined by induction on A

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

$$\|\forall x^\tau \ A(x)\| = \bigcup_{v \in \mathscr{M}_\tau} \|A(v)\|$$

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

$$|A| = \|A\|^{\perp\!\!\!\perp} = \{t \in \Lambda : \forall \pi \in \|A\|, \quad t \star \pi \in \perp\!\!\!\perp\}$$

- **Realizability relation:** $t \Vdash A \ \equiv \ t \in |A|$ (depends on the pole $\perp\!\!\!\perp$)

Adequacy

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

Typing rules

$$\frac{}{\Gamma \vdash x : A}\ (x{:}A)\in\Gamma$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x\,.\,t : A \Rightarrow B} \qquad \frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall x^\tau\, A}\ x^\tau \notin FV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall x^\tau\, A}{\Gamma \vdash t : A\{x^\tau := M^\tau\}}$$

$$\frac{}{\Gamma \vdash \text{æ} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A}$$

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:

$$\mathsf{quote} \star t \cdot u \cdot \pi \quad \succ \quad u \star \overline{n} \cdot \pi$$

where $n = \ulcorner t \urcorner$ is the Gödel code of term t

- Direct witness extraction techniques for Σ^0_1- and Π^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 $\mathscr{M}_{\perp\!\!\perp}$:

Which interesting formulas are realized in $\mathscr{M}_{\perp\!\!\perp}$ that were not already true in the ground model $\mathscr{M}$?

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 (CC^ω) [M.'07]
  - The calculus of inductive constructions (CIC) [M.'09, unpublished]
  - Zermelo-Fraenkel set theory (ZF) [Krivine'01]

  Note that in CC^ω/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, CC^ωor 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. $\quad e, e' \quad ::= \quad x \quad | \quad h(e_1, \ldots, e_k) \quad | \quad f(e_1, \ldots, e_k)$

Formulas $\quad A, B, C \quad ::= \quad X(e_1, \ldots, e_k) \quad | \quad A \Rightarrow B \quad | \quad \forall x\ B \quad | \quad \forall X\ B$

- 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)	$\bot$	$:=$	$\forall Z\ Z$
(Triviality)	$\top$	$:=$	$\forall Z\ (Z \Rightarrow Z)$
(Negation)	$\neg A$	$:=$	$A \Rightarrow \bot$
(Conjunction)	$A \wedge B$	$:=$	$\forall Z\ ((A \Rightarrow B \Rightarrow Z) \Rightarrow Z)$
(Disjunction)	$A \vee B$	$:=$	$\forall Z\ ((A \Rightarrow Z) \Rightarrow (B \Rightarrow Z) \Rightarrow Z)$
(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) $$\frac{}{\Gamma \vdash A}\ (A \in \Gamma \cup \mathscr{A})$$

($\Rightarrow$) $$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

($\forall^1$) $$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\, A}\ (x \notin FV(\Gamma)) \qquad \frac{\Gamma \vdash \forall x\, A}{\Gamma \vdash A\{x := e\}}$$

($\forall^2$) $$\frac{\Gamma \vdash A}{\Gamma \vdash \forall X\, A}\ (X \notin FV(\Gamma)) \qquad \frac{\Gamma \vdash \forall X\, A}{\Gamma \vdash A\{X(x_1, \ldots, x_k) := B\}}$$

(Peirce's law) $$\frac{}{\Gamma \vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A}$$

- $\mathscr{A}$ = set of axioms of xPA2
- The usual rules of $\neg$, $\wedge$, $\vee$, $\exists$, ... 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, Π-type, ...) we relativize it to the set $\mathbb{N}$ of Dedekind numerals

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

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 ($\mathscr{A}$) of xPA2 (1/2)

- Peano axioms:

$$\forall x\ \forall y\ (s(x) = s(y) \Rightarrow x = y), \qquad \forall x\ \neg(s(x) = 0)$$

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:

$$\begin{array}{rcl} 0 + y & = & y \\ s(x) + y & = & s(x + y) \end{array} \qquad \begin{array}{rcl} 0 \times y & = & 0 \\ s(x) \times y & = & (x \times y) + y \end{array} \qquad \text{etc.}$$

We only allow primitive recursive definitions (possibly using experimental functions), to ensure the totality of all user-defined functions

The axioms ($\mathscr{A}$) of xPA2 (2/2)

- The experimental functions (“h”) are specified by the means of Π^0_1-axioms $U_1, \dots, U_\ell$, of the form

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

where $e_i(x_1, \dots, x_{k_i})$ may contain experimental functions (“h”)

- To sum up:

$$\begin{array}{lcl} \text{xPA2} & = & \text{deduction rules of 2nd-order logic} \\ & + & \text{Peano axioms} \\ & + & \text{totality of experimental functions (“}h\text{”)} \\ & + & \text{defining equations of used-defined functions (“}f\text{”)} \\ & + & \text{specification } U_1, \dots, U_\ell \text{ of experimental functions} \end{array}$$

The experimental modus tollens

- In this framework, we assume that the correctness of the high-level program is given in the form of a Π^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 λ_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 $(m_1, \ldots, m_{k_i}) \in \mathbb{N}^{k_i}$:

$$\mathsf{test}_i \star \bar{m}_1 \cdots \bar{m}_{k_i} \cdot t \cdot \pi \quad \succ$$

$$\begin{cases} t \star \pi & \text{if } e_i(m_1, \ldots, m_{k_i}) = 0 \\ \mathsf{stop} \star \bar{i} \cdot \bar{m}_1 \cdots \bar{m}_{k_i} \cdot \diamond & \text{otherwise} \end{cases}$$

Program extraction

From the derivation d of the formula V (in xPA2),
we extract a λ_c-term d^* as follows:

- Logical constructions are extracted as usual, according to the Curry-Howard correspondence (⇒-intro. by λ, ⇒-elim. by app., ...)
- Peirce's law is extracted as $\text{\ae}$
- Peano axioms are extracted as $\mathbf{I} \equiv \lambda x \,.\, x$ and $\lambda y \,.\, y\,\mathbf{I}$, respectively
- Definitional axioms of user-defined functions (possibly involving h's) are extracted as $\mathbf{I} \equiv \lambda x \,.\, x$
- 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 λ_c-term $M_i\ \mathsf{test}_i$ (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:

$$d^* \star \bar{n}_1 \cdots \bar{n}_k \cdot \mathbf{I} \cdot \diamond \quad \succ \quad \text{stop} \star \bar{i} \cdot \bar{m}_1 \cdots \bar{m}_{k_i} \cdot \diamond$$

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 $\perp\!\!\!\perp$ formed by all processes $p \succ \text{stop} \star \bar{i} \cdot \bar{m}_1 \cdots \bar{m}_{k_i} \cdot \diamond$ 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* $\perp\!\!\!\perp$. The only interesting case is the case of instruction test_i, that appears to be a realizer of U_i due to the particular definition of $\perp\!\!\!\perp$.
- We conclude by observing that $d^* \bar{n}_1 \cdots \bar{n}_k \mathbf{I} \Vdash \perp$ □

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 Π^0_1 (falsifiable, according to Popper's terminology)

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

- Moreover, the extraction procedure allows many optimizations:
  - Removing proofs of Π^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^ω/CIC
+ extraction function $M \mapsto M^*$ from CC^ω/CIC to λ_c [M.'07]

Adequacy: If $M : T$, then $M^* \Vdash_{\llbracket T \rrbracket} \llbracket M \rrbracket$ (dependent realizability)

- Experimental functions h are of the form:

$$\texttt{Axiom}\ \ h\ :\ \Pi\vec{x}:\vec{T}.\ S$$

  where $\vec{T}, S$ are purely algebraic datatypes (no $\to$/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 Π^0_1:

$$U_i/V\ ::=\ \Pi\vec{x}:\vec{T}.\ M_1 = M_2$$

  where $\vec{T}$ are purely algebraic datatypes (no $\to$/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:

$$\mathrm{Cons}(\mathrm{ZFC} + \neg\mathrm{CH}), \qquad \text{or:} \qquad \mathrm{ZFC} \nvdash \mathrm{CH}$$

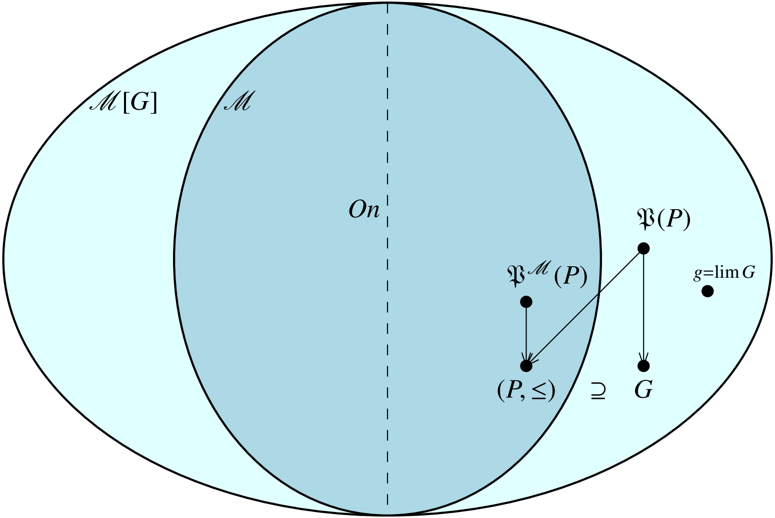
  Gödel ('38) proved $\mathrm{Cons}(\mathrm{ZFC} + \mathrm{CH})$ (or: $\mathrm{ZFC} \nvdash \neg\mathrm{CH}$) introducing constructible sets

- 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-specificiation of the power set $\mathfrak{P}(X)$ (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: $A \quad \mapsto \quad p \Vdash A$ $(p \in P)$

General properties

1. $\vdash A$ entails $\vdash (p \Vdash A)$ (for all conditions $p \in P$)
2. But $\vdash (p \Vdash A)$ for more formulas A (depending on P)
3. $\vdash p \nVdash \bot$ (consistency)

- **Remark:** Forcing commutes with $\bot$, $\top$, $\wedge$ and $\forall$, but not with $\Rightarrow$, $\neg$, $\vee$, $\exists$:

$$\begin{array}{rcl} (p \Vdash \bot) & \Leftrightarrow & \bot \\ (p \Vdash \top) & \Leftrightarrow & \top \end{array} \qquad \begin{array}{rcl} (p \Vdash A \wedge B) & \Leftrightarrow & (p \Vdash A) \wedge (p \Vdash B) \\ (p \Vdash \forall x\, A) & \Leftrightarrow & \forall x\, (p \Vdash A) \end{array}$$

Classical realizability and Cohen forcing

- **Forcing in classical realizability** [Krivine '09]
  - Introduce realizability algebras, generalizing the λ_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** [Miquel '11]
  - Hard-wire the program transformation into the abstract machine

Underlying methodology

Translation of formulas & proofs	$\leadsto$	Classical program transformation	$\leadsto$	New abstract machine (no transformation)

A glimpse of the forcing translation in PAω

- Translating sorts: $\tau \mapsto \tau^*$

$$\iota^* \equiv \iota \qquad o^* \equiv \kappa \to o \qquad (\tau \to \sigma)^* \equiv \tau^* \to \sigma^*$$

- Translating higher-order terms: $M \mapsto M^*$

$$\begin{array}{rclrcl}
(x^\tau)^* & \equiv & x^{\tau^*} & 0^* & \equiv & 0 \\
(\lambda x^\tau . M)^* & \equiv & \lambda x^{\tau^*} . M^* & s^* & \equiv & s \\
(MN)^* & \equiv & M^* N^* & \mathrm{rec}_\tau^* & \equiv & \mathrm{rec}_{\tau^*}
\end{array}$$

$$\begin{array}{rcl}
(A \Rightarrow B)^* & \equiv & \lambda r^\kappa . \forall q^\kappa \forall r'^\kappa (r = qr' \mapsto (\forall s^\kappa (C[qs] \Rightarrow A^* s) \Rightarrow B^* r')) \\
(\forall x^\tau A)^* & \equiv & \lambda r^\kappa . \forall x^{\tau^*} A^* r \\
(M_1 = M_2 \mapsto A)^* & \equiv & \lambda r^\kappa . M_1^* = M_2^* \mapsto (A^* r)
\end{array}$$

- Forcing propositions (terms of type o):

$$p \Vdash A \equiv \forall r^\kappa (C[pr] \Rightarrow A^* r)$$

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

$$x^* \equiv x \qquad \text{æ}^* \equiv \lambda cx \,.\, \text{æ}\,(\lambda k \,.\, x\,(\alpha_{14}\, c)\,(\gamma_4\, k))$$

$$(t\,u)^* \equiv \gamma_3\, t^*\, u^*$$

$$(\lambda x \,.\, t)^* \equiv \gamma_1\,(\lambda x \,.\, t^*\, \underbrace{\{x := \beta_4 x\}}_{\text{bounded var}}\, \underbrace{\{x_i := \beta_3 x_i\}_{i=1}^{n}}_{\text{other free vars of } t})$$

$\gamma_4 \equiv \lambda xcy \,.\, x\,(y\,(\alpha_{15}\, c))$

$\gamma_3 \equiv \lambda xyc \,.\, x\,(\alpha_{11}\, c)\, y$

$\gamma_1 \equiv \lambda xcy \,.\, x\, y\,(\alpha_6\, c)$

$\beta_3 \equiv \lambda xc \,.\, x\,(\alpha_9\, c)$

$\beta_4 \equiv \lambda xc \,.\, x\,(\alpha_{10}\, c)$

Soundness

If $\quad x_1 : A_1, \;\ldots,\; x_n : A_n \vdash t : B$

then $\quad x_1 : (p \Vdash A_1), \;\ldots,\; x_n : (p \Vdash A_n) \vdash t^* : (p \Vdash B)$

Krivine Forcing Abstract Machine (KFAM)

$$
\begin{array}{llcl}
\textbf{Terms} & t, u & ::= & x \mid \lambda x\,.\,t \mid tu \mid \text{œ} \\
\textbf{Environments} & e & ::= & \emptyset \mid e; x := c \\
\textbf{Closures} & c & ::= & t[e] \mid \mathsf{k}_\pi \mid \underbrace{t[e]^* \mid \mathsf{k}^*_\pi}_{\text{forcing closures}} \\
\textbf{Stacks} & \pi & ::= & \diamond \mid c \cdot \pi
\end{array}
$$

- Real mode:

$$
\begin{array}{rcl l}
x[e; y =: c] \star \pi & \succ & x[e] \star \pi & (y \not\equiv x) \\
x[e; x =: c] \star \pi & \succ & c \star \pi & \\
(\lambda x\,.\,t)[e] \star c \cdot \pi & \succ & t[e; x := c] \star \pi & \\
(tu)[e] \star \pi & \succ & t[e] \star u[e] \cdot \pi & \\
\text{œ}[e] \star uc \cdot \pi & \succ & c \star \mathsf{k}_\pi \cdot \pi & \\
\mathsf{k}_\pi \star c \cdot \pi' & \succ & c \star \pi &
\end{array}
$$

- Forcing mode:

$$
\begin{array}{rcl l}
x[e; y := c]^* \star c_0 \cdot \pi & \succ & x[e]^* \star \alpha_9\, c_0 \cdot \pi & (y \not\equiv x) \\
x[e; x := c]^* \star c_0 \cdot \pi & \succ & c \star \alpha_{10}\, c_0 \cdot \pi & \\
(\lambda x\,.\,t)[e]^* \star c_0 \cdot c \cdot \pi & \succ & t[e; x := c]^* \star \alpha_6\, c_0 \cdot \pi & \\
(tu)[e]^* \star c_0 \cdot \pi & \succ & t[e]^* \star \alpha_{11}\, c_0 \cdot u[e]^* \cdot \pi & \\
\text{œ}[e]^* \star c_0 \cdot c \cdot \pi & \succ & c \star \alpha_{14}\, c_0 \cdot \mathsf{k}^*_\pi \cdot \pi & \\
\mathsf{k}^*_\pi \star c_0 \cdot c \cdot \pi' & \succ & c \star \alpha_{15}\, c_0 \cdot \pi &
\end{array}
$$