The Static Debugger: classical realizability applied to debugging

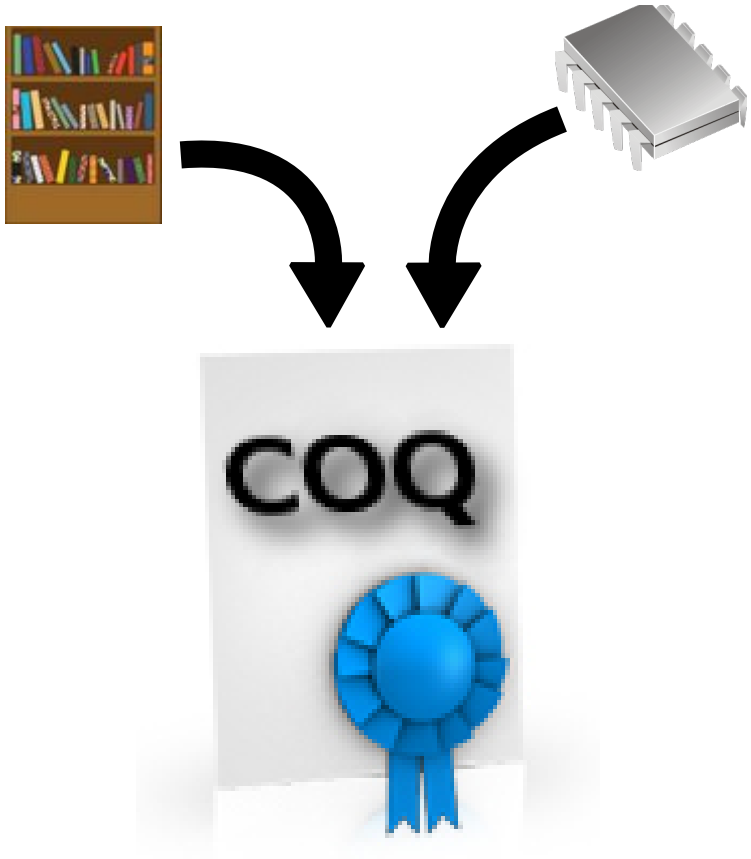
Lionel Rieg

LIP, ENS Lyon

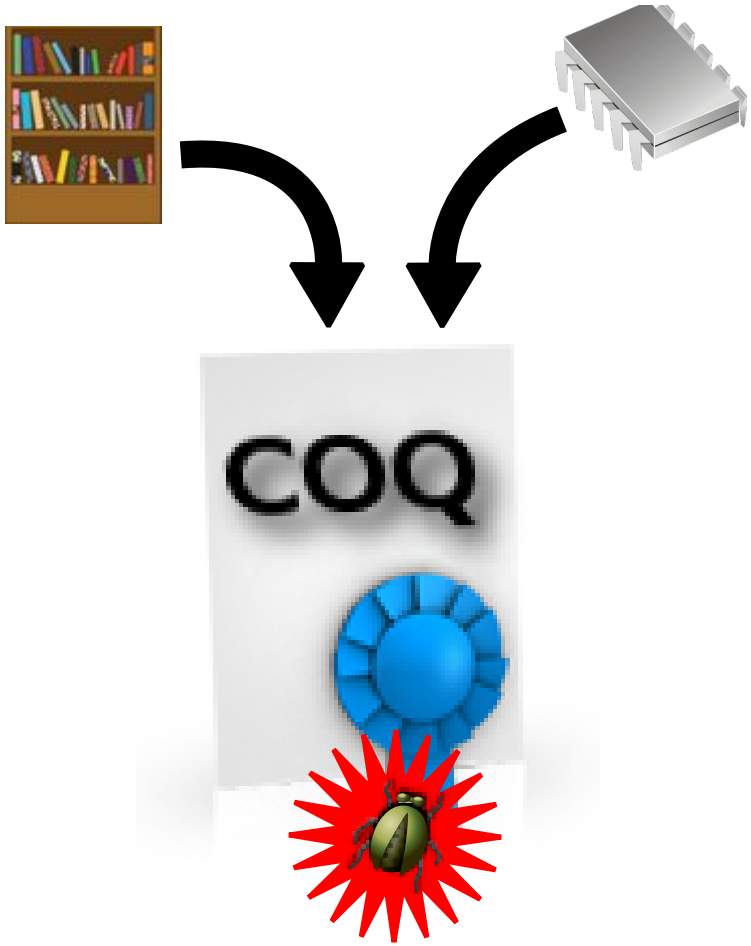
Coq Workshop, August 26th 2011

Motivation: Static debugger

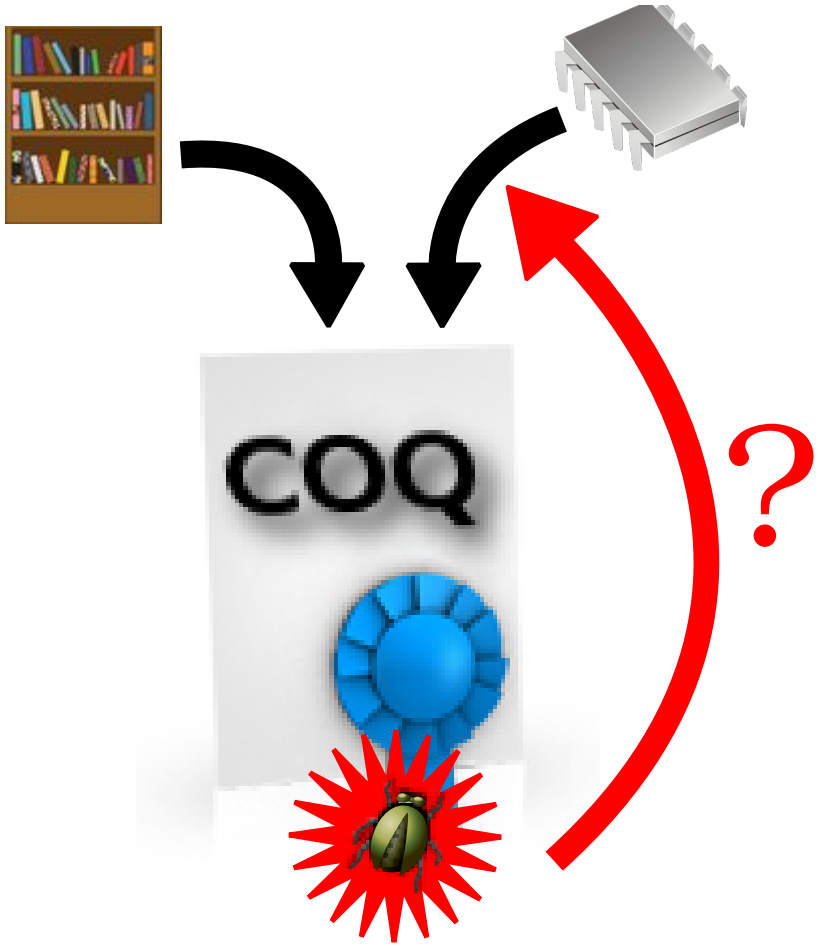
Motivation: Static debugger



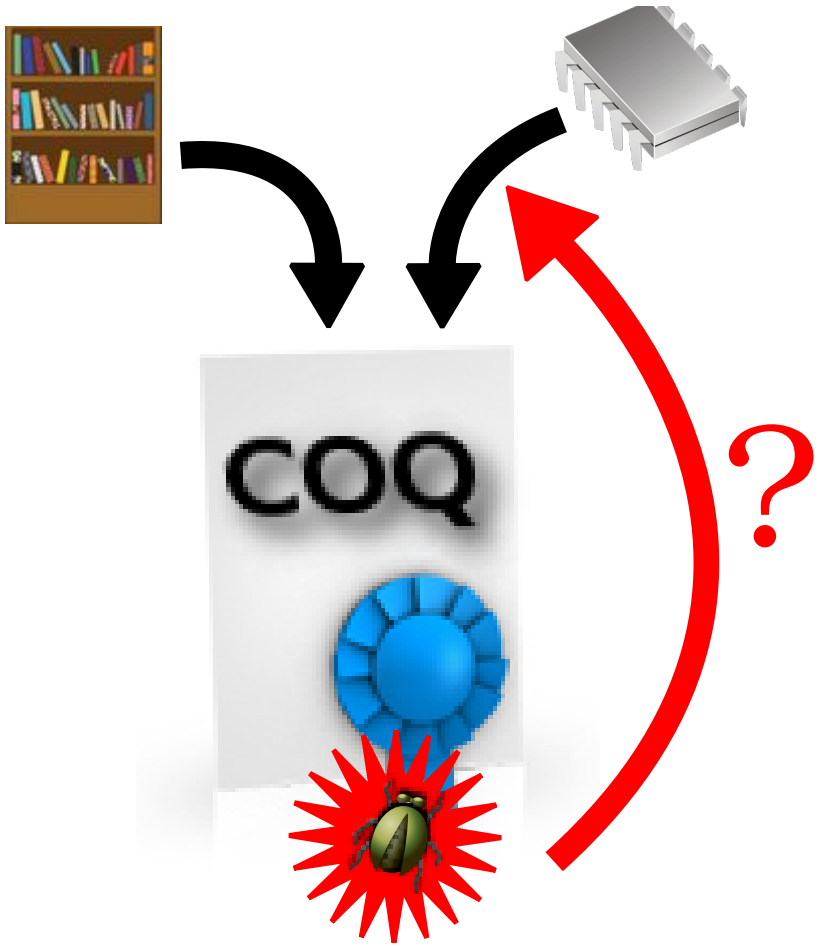
Motivation: Static debugger



Motivation: Static debugger



Motivation: Static debugger



We will need:

- the correctness proof of the program
- the bug report
- the implementations of the external tools

➥ independent of the programming language

Logical formalization

$$\frac{\overbrace{\vdash (\forall x, Ax) \Rightarrow \forall y, By}^{\text{program certification}} \qquad \overbrace{\not\models By_0}^{\text{bug report}}}{\underbrace{\not\models A\ x_?}_{\text{bug report for one external tool}}}\ \text{Experimental Modus Tollens}$$

A and B $\forall/\exists$-free formulæ containing external predicates

Logical formalization

$$\frac{\overbrace{\vdash (\forall x, Ax) \Rightarrow \forall y, By}^{\text{program certification}} \qquad \overbrace{\not\models By_0}^{\text{bug report}}}{\underbrace{\not\models A\ x_?}_{\text{bug report for one external tool}}}\ \text{Experimental Modus Tollens}$$

A and B $\forall/\exists$-free formulæ containing external predicates

which can be further reduced to

$$\frac{\forall x, A\,x \vdash \bot}{\not\models A\ x_?}\ \text{Experimental Effectiveness}$$

➥ We want to exhibit a counter-example ($\not\models A\ x_?$)
from a proof of contradiction ($\forall x, A\,x \vdash \bot$)

From a counter-example to a Herbrand tree

- In the universal theory $\mathsf{U} = \forall x, A\,x, \quad A\,x$ is $\forall/\exists$-free and contains external predicates and functions
  $\rightsquigarrow$ *e.g.* library functions

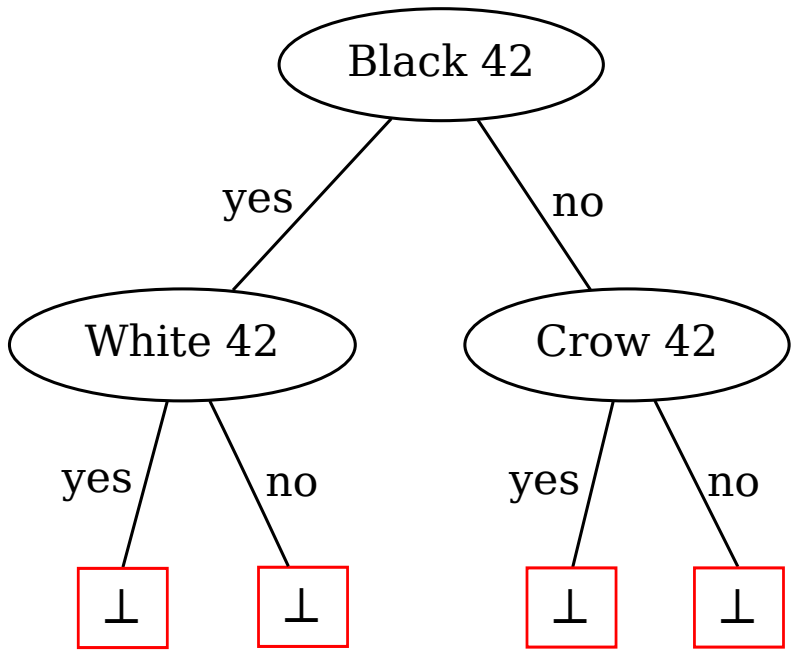
- We want to abstract on the interpretation
  (*i.e.* a specific implementation of the external tools)
  $\rightsquigarrow$ no test to perform anymore

➥ we need a Herbrand tree = tree of possible interpretations
= BDD of counter-examples

Herbrand trees through an example

- atoms (= atomic formulæ)
  - Crow n
  - Black n
  - White n
- a (finite) inconsistent theory
  - $A_1 : \forall n, \text{Crow } n \Longrightarrow \text{Black } n$
  - $A_2 : \forall n, \neg(\text{Black } n \wedge \text{White } n)$
  - $A_3 : \text{Crow } 42$
  - $A_4 : \text{White } 42$
- contradictions on the leaves

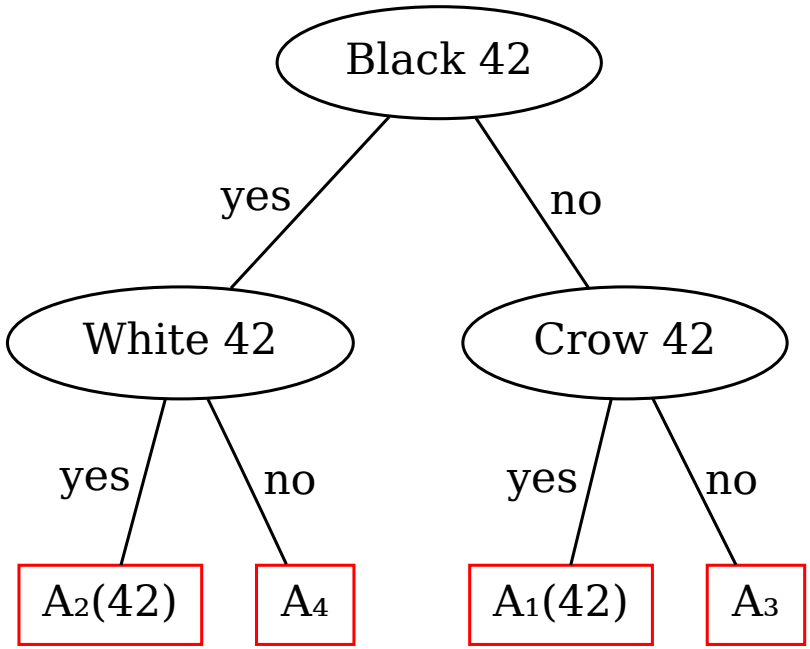
White crow example



Herbrand trees through an example

- atoms (= atomic formulæ)
  - Crow n
  - Black n
  - White n
- a (finite) inconsistent theory
  - $A_1 : \forall n, \text{Crow } n \Longrightarrow \text{Black } n$
  - $A_2 : \forall n, \neg(\text{Black } n \wedge \text{White } n)$
  - $A_3 : \text{Crow } 42$
  - $A_4 : \text{White } 42$
- contradictions on the leaves plus counter-examples

White crow example

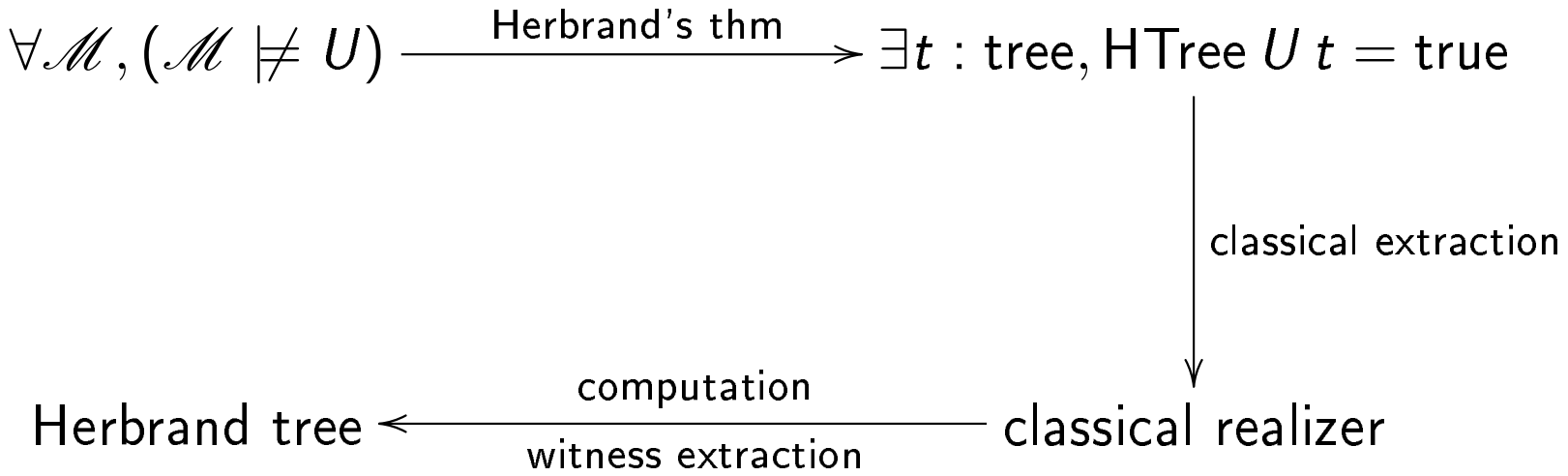


The proposed solution

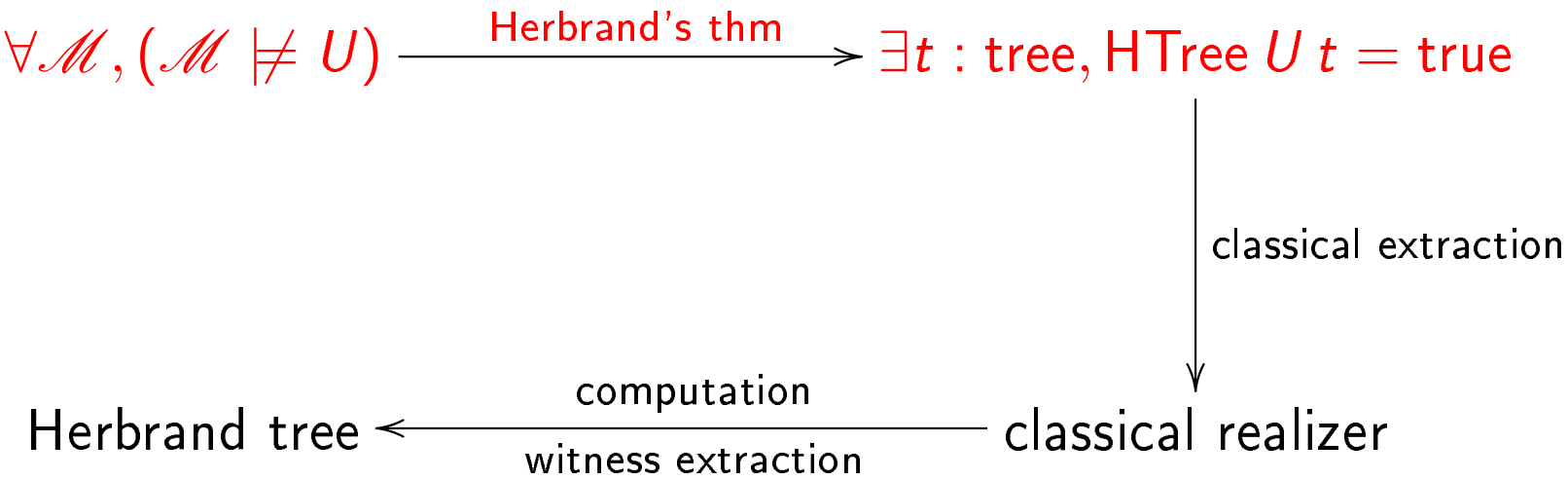
Extract Herbrand's theorem with classical realizability

Theorem (Herbrand)

Let U be a universal theory.
If for all interpretations $\mathscr{M}$, $\mathscr{M} \not\models U$, then U has a Herbrand tree.



Herbrand's theorem



Herbrand's Theorem proof

Theorem (Herbrand)

If for all interpretations $\mathscr{M}$, $\mathscr{M} \not\models U$, then U has a Herbrand tree.

Herbrand's Theorem proof

Theorem (Herbrand)

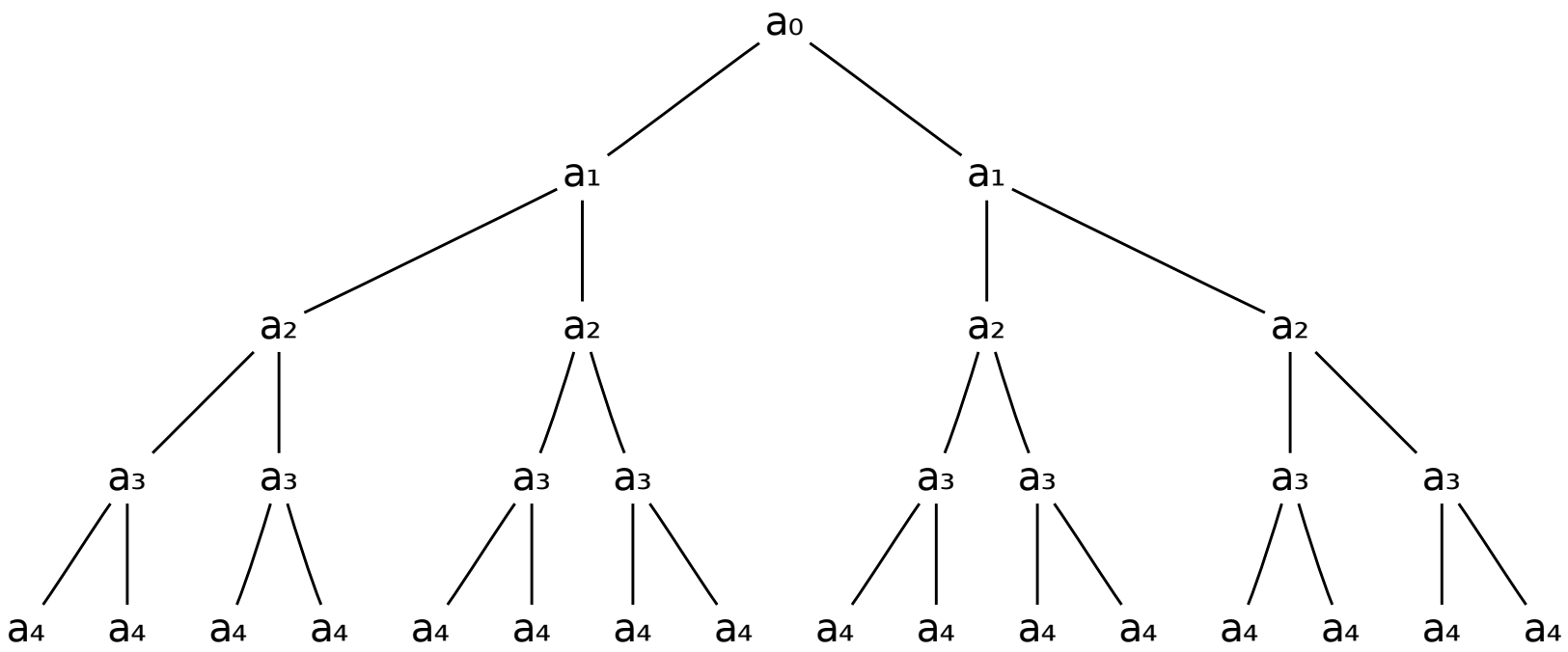
If for all interpretations $\mathscr{M}$, $\mathscr{M} \not\models U$, then U has a Herbrand tree.

Let us fix an enumeration $(a_i)_{i \in \mathbb{N}}$ of the atoms.
(atoms = atomic formulæ)

Herbrand's Theorem proof

Theorem (Herbrand)

If for all interpretations $\mathscr{M}$, $\mathscr{M} \not\models U$, then U has a Herbrand tree.

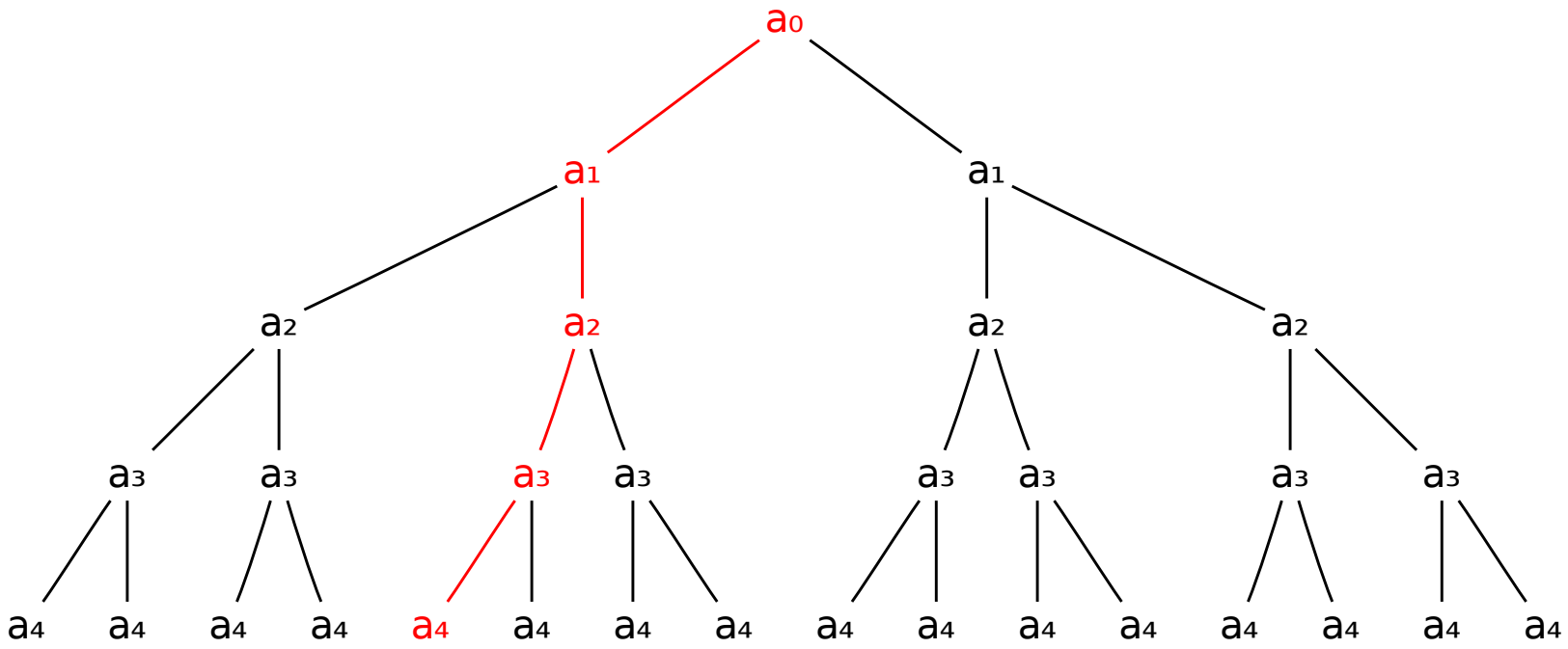


consider the atom-enumerating complete infinite tree

Herbrand's Theorem proof

Theorem (Herbrand)

If for all interpretations $\mathscr{M}$, $\mathscr{M} \not\models U$, then U has a Herbrand tree.

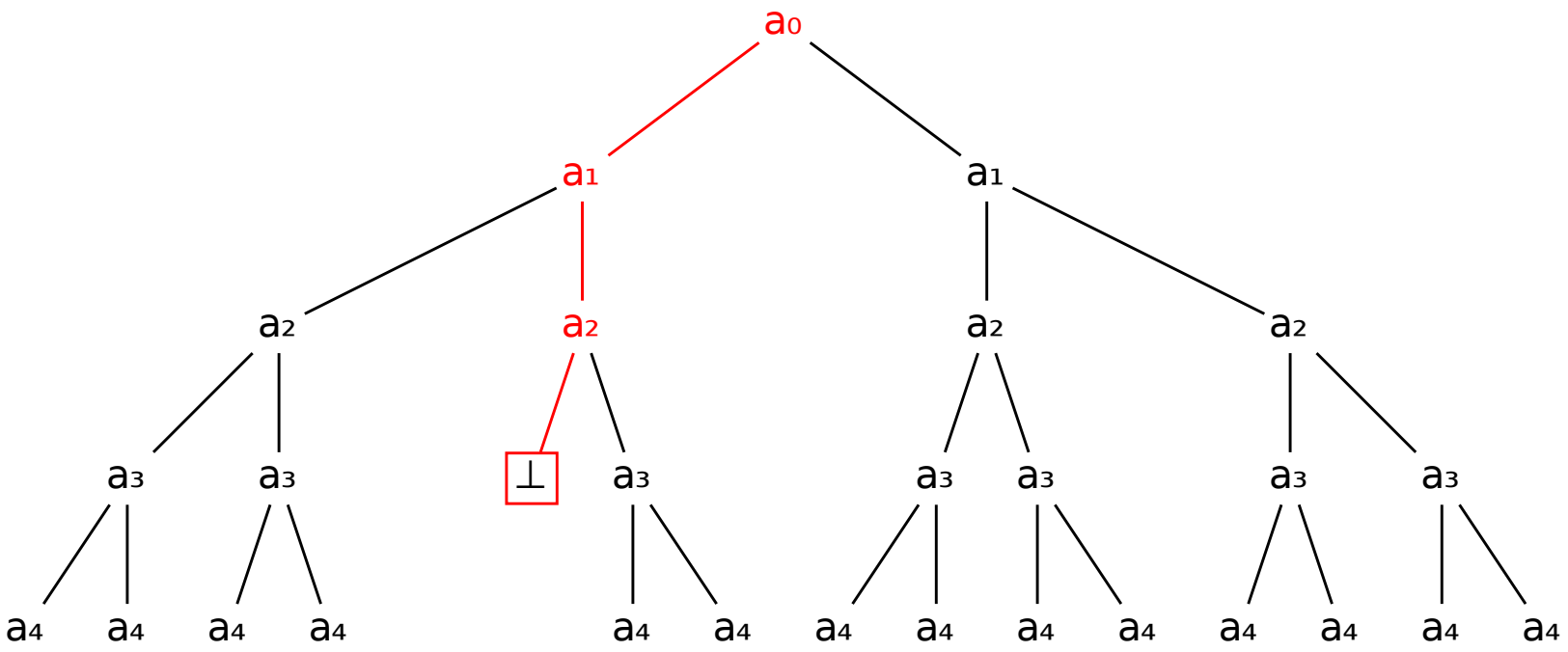


pick any infinite branch

Herbrand's Theorem proof

Theorem (Herbrand)

If for all interpretations $\mathscr{M}$, $\mathscr{M} \not\models U$, then U has a Herbrand tree.

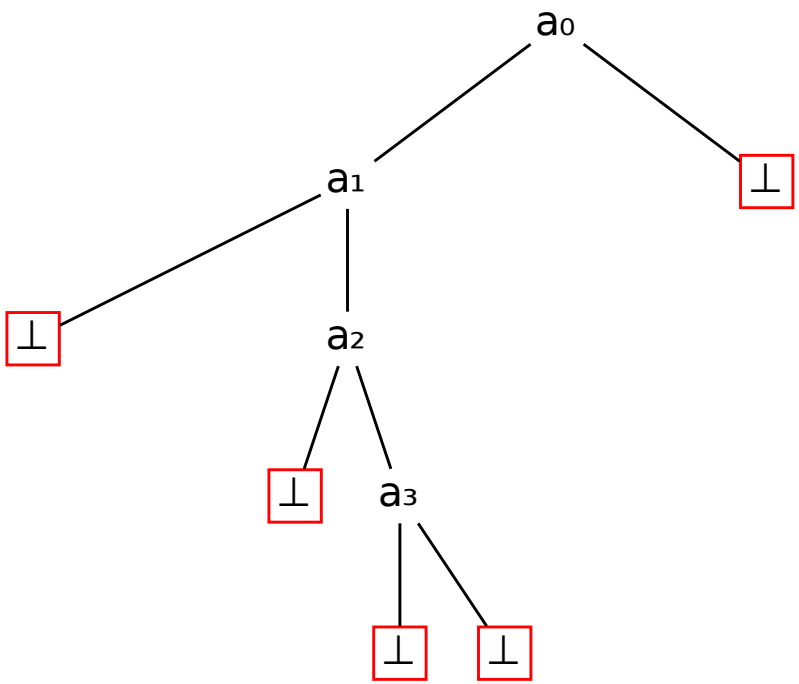


by hypothesis (and compactness), we can cut it at finite depth

Herbrand's Theorem proof

Theorem (Herbrand)

If for all interpretations $\mathscr{M}$, $\mathscr{M} \not\models U$, then U has a Herbrand tree.



conclude using Kőnig's lemma

Formalized proof

Usual proof of Herbrand's theorem

➥ uses the compactness theorem and Kőnig's lemma

Formalized proof

Usual proof of Herbrand's theorem

➥ uses the compactness theorem and Kőnig's lemma

Uses *reductio ad absurdum*.

Builds an infinite branch in a "potential Herbrand tree".

Formalized proof

Usual proof of Herbrand's theorem
➥ uses the compactness theorem and Kőnig's lemma

Uses *reductio ad absurdum*.
Builds an infinite branch in a "potential Herbrand tree".

1. suppose there is no Herbrand tree
2. show that any partial interpretation consistent with the theory can be extended into a longer one
3. let u be the union of the increasing sequence built by extension from the empty path
   ➥ it is the infinite branch
4. show that u contains no contradiction and is a model of U (by syntactic compactness)

Coq implementation

We need the proof to be classically extractable

Some aspects of the proof:

- the proof is parametrized by two abstract data types
  - the signature
  - the theory U
- 2 axioms added to Coq: (in Prop only)
  - excluded middle: $\forall P : \text{Prop}, P \vee \neg P$
  - proof irrelevance: $\forall P : \text{Prop}, \forall p_1\ p_2 : P,\ p_1 = p_2$
- decidable *vs.* undecidable
  - extraction requires finite objects with decidable properties
  - but the proof uses (at some point) infinite objects

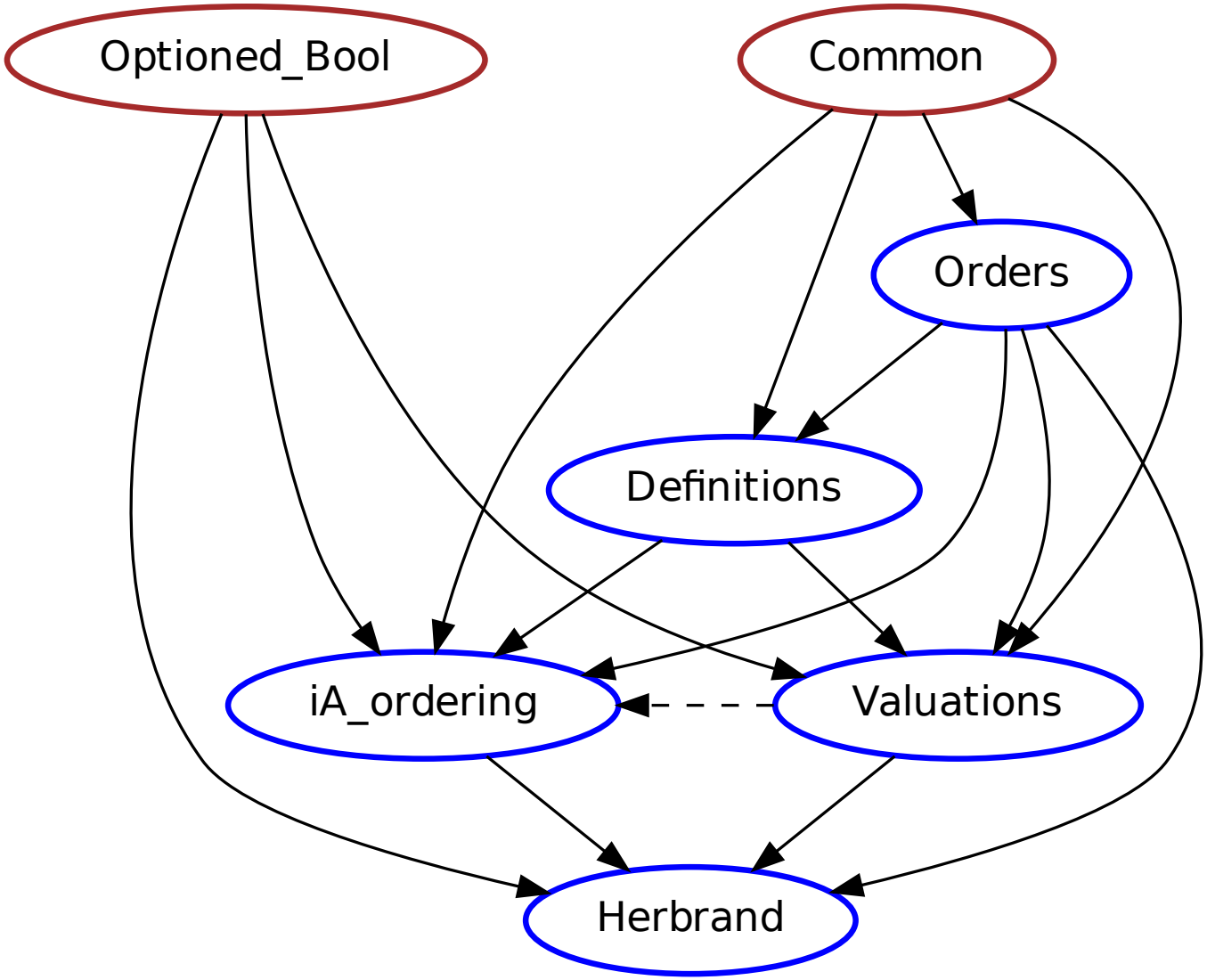
The guts of the proof

1. The real statement

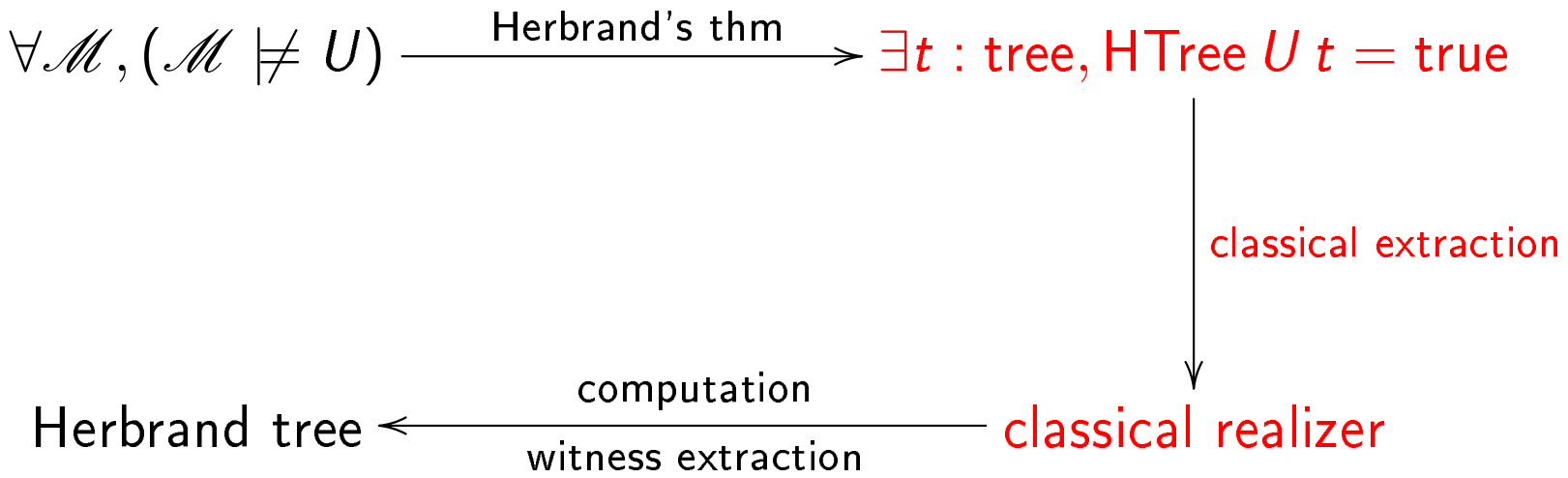
$$\forall\, \text{atom} : \text{Set},$$
$$\forall\, \text{index} : \text{Set}, \forall\, \text{U} : \text{index} \to \text{compound},$$
$$\left(\forall \mathit{val} : \text{atom} \to \text{Prop}, \neg(\forall i : \text{index}, \text{eval}\ \mathit{val}\ (\text{U}\ i)) \right)$$
$$\to \exists t : \text{tree}, \text{HTree}\ \text{U}\ t = \text{true}$$

2. Boolean equality and order on dependent pairs $\langle i, a \rangle$ $(a \in \text{U}\ i)$ (requires the same for atom and index)
   ➥ used to prove that the infinite branch we build is a model

The structure of the proof



Extraction


$\forall \mathscr{M}, (\mathscr{M} \not\models U)$
Herbrand's thm
$\exists t : \text{tree}, \text{HTree}\ U\ t = \text{true}$
classical extraction
Herbrand tree
computation
witness extraction
classical realizer

The syntactic extraction function

- source language: CoC_ω + Peirce's law
- target language: λ-calculus + call/cc

$$
\begin{array}{rcl}
\mathit{CoC}_\omega + \mathrm{Peirce} & \xrightarrow{*} & \lambda_c \\
x^* & = & x \\
(\lambda x : T.M)^* & = & \lambda x.M^* \\
(MN)^* & = & M^* N^* \\
(\Pi \_)^* & = & \text{any term} \\
s^* & = & \text{any term} \\
\mathrm{Pierce}^* & = & \lambda \_.\mathrm{call/cc}
\end{array}
$$

The syntactic extraction function

- source language: CoC_ω + Peirce's law
- target language: λ-calculus + call/cc

$$
\begin{array}{rcll}
CoC_\omega + \text{Peirce} & \xrightarrow{*} & \lambda_c & \\
x^* & = & x & \\
(\lambda x : T.M)^* & = & \lambda x.M^* & \text{for all sorts} \\
(MN)^* & = & M^*N^* & \text{for all sorts} \\
(\Pi\_)^* & = & \text{any term} & \\
s^* & = & \text{any term} & \\
\text{Pierce}^* & = & \lambda\_.\text{call/cc} &
\end{array}
$$

Remarks

- types are erased because they have no computational content
- all sorts are treated the same way

Toward CIC$_\omega$: adding inductive

Inductive data structure are encoded by elimination

Toward CIC$_\omega$: adding inductive

Inductive data structure are encoded by elimination

- in Coq

  Inductive foo p_1 p_2 :=
  | C_1 : foo p_1 p_2
  | C_2 a : foo p_1 p_2
  | C_3 b_1 b_2 b_3 : foo p_1 p_2

- in λ_c (`Jivaro` syntax)

$$\begin{array}{llll}
\text{Define } C_1 = & \lambda p_1 \lambda p_2 & \lambda e_1 \lambda e_2 \lambda e_3 & e_1 \\
\text{Define } C_2 = & \lambda p_1 \lambda p_2 \; \lambda a & \lambda e_1 \lambda e_2 \lambda e_3 & e_2\ a \\
\text{Define } C_3 = & \lambda p_1 \lambda p_2 \; \lambda b_1 \lambda b_2 \lambda b_3 & \lambda e_1 \lambda e_2 \lambda e_3 & e_3\ b_1\, b_2\, b_3
\end{array}$$

➥ matching is the identity!

Correctness of the extraction: realizability model for PA2

Classical realizability is a "negative interpretation" of formulæ
but keeps intuitionistic typing rules

➥ formulæ are interpreted by sets of stacks ("falsity values")
➥ realizers are defined by orthogonality to those stacks

Correctness of the extraction: realizability model for PA2

Classical realizability is a "negative interpretation" of formulæ but keeps intuitionistic typing rules

➥ formulæ are interpreted by sets of stacks ("falsity values")
➥ realizers are defined by orthogonality to those stacks

Examples

$$\begin{array}{rcl} \lambda x.\, x & \Vdash & \forall Z, Z \Rightarrow Z \\ \text{call/cc} & \Vdash & \forall A \forall B, ((A \Rightarrow B) \Rightarrow A) \Rightarrow A \end{array}$$

➥ intuition: realizers of $\bot$ trigger backtracks

Correctness of the extraction: realizability model for PA2

Classical realizability is a "negative interpretation" of formulæ but keeps intuitionistic typing rules

➥ formulæ are interpreted by sets of stacks ("falsity values")
➥ realizers are defined by orthogonality to those stacks

Examples

$$\begin{array}{rcl} \lambda x.\, x & \Vdash & \forall Z, Z \Rightarrow Z \\ \text{call/cc} & \Vdash & \forall A \forall B, ((A \Rightarrow B) \Rightarrow A) \Rightarrow A \end{array}$$

➥ intuition: realizers of $\bot$ trigger backtracks

Theorem (Adequacy)

Every formula provable in PA2 has a universal realizer.

What happens beyond PA2?

Different realizability models:

- PA2 [Krivine, 2010]
- ZF [Krivine, 2001]
- CoC_ω + some inductive types [Miquel, 2007]

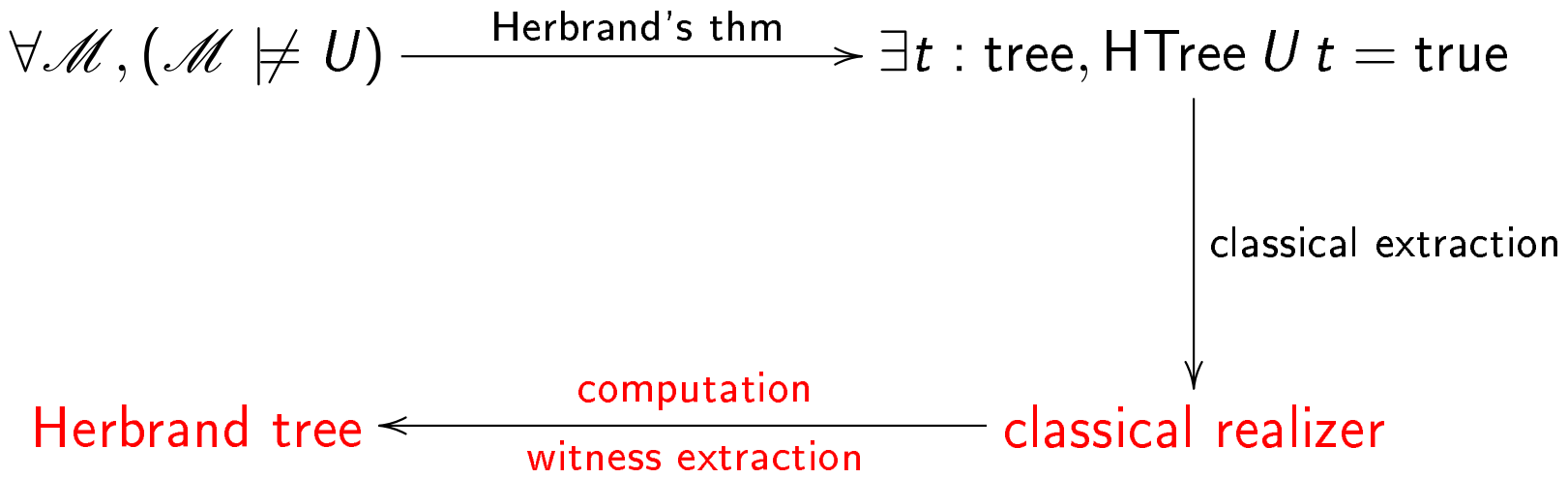
What happens beyond PA2?

Different realizability models:

- PA2 [Krivine, 2010]
- ZF [Krivine, 2001]
- CoC_ω + some inductive types [Miquel, 2007]

Hopefully, the 2^{nd} order fragments of their realizability models are isomorphic

Witness extraction



Computation framework: Krivine's machine for λ -calculus

- terms: $t = x \mid \lambda x.t \mid t\ t$
- stacks: $\pi = \varepsilon \mid t \cdot \pi$ (t closed)
- processes: $t \star \pi$ (t closed)
- evaluation relation $\succ$:

$$\begin{array}{lrcl} \text{Grab} & \lambda x.\, t \star u \cdot \pi & \succ & t[u/x] \star \pi \\ \text{Push} & t\ u \star \pi & \succ & t \star u \cdot \pi \end{array}$$

Computation framework: Krivine's machine for λ_c-calculus

- terms: $t = x \mid \lambda x.t \mid t\,t \mid \text{call/cc} \mid k_\pi \mid \ldots$
- stacks: $\pi = \varepsilon \mid t \cdot \pi$ (t closed)
- processes: $t \star \pi$ (t closed)
- evaluation relation $\succ$:

$$
\begin{array}{rrcl}
\text{Grab} & \lambda x.\, t \star u \cdot \pi & \succ & t[u/x] \star \pi \\
\text{Push} & t\,u \star \pi & \succ & t \star u \cdot \pi \\
\text{Save} & \text{call/cc} \star t \cdot \pi & \succ & t \star k_\pi \cdot \pi \\
\text{Restore} & k_\pi \star t \cdot \pi' & \succ & t \star \pi \\
\vdots & \vdots \star \vdots & \succ & \vdots \star \vdots
\end{array}
$$

Witness extraction

When we have a realizer of a decidable Σ_1 formula

$$t \quad \Vdash \quad \exists x : T, f(x) = 0$$
$$\equiv \forall Z, (\forall x, T(x) \to f(x) = 0 \to Z) \to Z$$

it eventually evaluates into a pair $\langle w, j \rangle$ where

- w is a witness (a realizer of $x : T$)
- j is the justification of w (a realizer of $f(w) = 0$)

Witness extraction

When we have a realizer of a decidable Σ_1 formula

$$t \quad \Vdash \quad \exists x : T, f(x) = 0$$
$$\equiv \forall Z, (\forall x, T(x) \to f(x) = 0 \to Z) \to Z$$

it eventually evaluates into a pair $\langle w, j \rangle$ where

- w is a witness (a realizer of $x : T$)
- j is the justification of w (a realizer of $f(w) = 0$)

But j (and w) can backtrack!

Witness extraction

When we have a realizer of a decidable Σ_1 formula

$$t \quad \Vdash \quad \exists x : T, f(x) = 0$$
$$\equiv \forall Z, (\forall x, T(x) \to f(x) = 0 \to Z) \to Z$$

it eventually evaluates into a pair $\langle w, j \rangle$ where

- w is a witness (a realizer of $x : T$)
- j is the justification of w (a realizer of $f(w) = 0$)

But j (and w) can backtrack!

$\rightsquigarrow$ w is not necessarily a correct witness

$\rightsquigarrow$ we need to evaluate the proof

➥ it explains why we keep proof information during extraction

We use $t\ (M_T\ (\lambda w\ \lambda p.\ j\ (\mathsf{stop}\ w)))$

General usage

1. apply Herbrand's theorem to the inconsistency proof
2. extract the resulting theorem with `kextraction`
3. realize extra axioms used
   - excluded middle: $\lambda\_\lambda g \lambda h.\, \mathrm{call/cc}(\lambda f.\, h\,(\lambda x.\, f\,(g\,x)))$
   - proof irrelevance: $\lambda\_\lambda\_\lambda\_\lambda x.\, x$
4. optimize realizers to speed up execution
   $\rightsquigarrow$ especially for code extracted from proofs (in Prop)
5. evaluate and retrieve actual trees with the wrapper

General usage

0. TODO: tactics to convert the inconsistency proof and automate the creation of the required data types
1. apply Herbrand's theorem to the inconsistency proof
2. extract the resulting theorem with `kextraction`
3. realize extra axioms used
   - excluded middle: $\lambda\_\lambda g\lambda h.\ \mathrm{call/cc}(\lambda f.\ h\,(\lambda x.\ f\,(g\,x)))$
   - proof irrelevance: $\lambda\_\lambda\_\lambda\_\lambda x.\ x$
4. optimize realizers to speed up execution
   $\rightsquigarrow$ especially for code extracted from proofs (in Prop)
5. evaluate and retrieve actual trees with the wrapper

General usage

0. TODO: tactics to convert the inconsistency proof and automate the creation of the required data types
1. apply Herbrand's theorem to the inconsistency proof
2. extract the resulting theorem with `kextraction`
3. realize extra axioms used
   - excluded middle: $\lambda\_\lambda g \lambda h.\, \mathrm{call/cc}(\lambda f.\, h\,(\lambda x.\, f\,(g\,x)))$
   - proof irrelevance: $\lambda\_\lambda\_\lambda\_\lambda x.\, x$
4. optimize realizers to speed up execution
   $\rightsquigarrow$ especially for code extracted from proofs (in Prop)
5. evaluate and retrieve actual trees with the wrapper

➥ certified program but slow
➥ depends on the quality of the proof of Herbrand's theorem

(Old) Experimentation results: why we need optimization

Example	T_{extract} (s.)	T_{optim} (s.)
1 ($k = 10$)	0.20	0.10
1 ($k = 50$)	1.92	0.29
1 ($k = 100$)	6.70	0.54
1 ($k = 500$)	163.66	2.51
1 ($k = 1000$)	646.63	5.03
2 ($k = 42$)	4.30	1.10
2 ($k = 1337$)	2832.41	29.90

$1 \quad = \quad \forall n, P\ n \rightarrow P\ (S\ n) \wedge P\ 0 \wedge \neg P\ k$

$2 \quad = \quad$ White Crow theory

Realizer optimization

Semantic optimization

Change the data type representation during extraction

it changes the realizability model and the adequacy lemma

e.g. primitive integers

- space savings: unary integers $\rightsquigarrow$ binary integers
- time savings: unary functions $\rightsquigarrow$ native operations

Realizer optimization

Semantic optimization

Change the data type representation during extraction

it changes the realizability model and the adequacy lemma

e.g. primitive integers

- space savings: unary integers $\rightsquigarrow$ binary integers
- time savings: unary functions $\rightsquigarrow$ native operations

Code optimization (mostly for Prop)

Replace extracted realizers by more efficient ones

currently only for obvious cases:
e.g. chains of arithmetical equalities $\rightsquigarrow \lambda x.x$

Example: commutativity of $+_{\mathsf{nat}}$ (extracted realizer)

```
Coq.Init.Datatypes.nat_rect =
  \P\f\f0 .fix_1_1 (\F\n Coq.Init.Datatypes.nat%case n f (\n f0 n (F n)))
Coq.Init.Datatypes.nat_ind =
  \P Coq.Init.Datatypes.nat_rect P
Coq.Init.Peano.plus_n_O =
  \n
    Coq.Init.Datatypes.nat_ind
      .type (Coq.Init.Logic.refl_equal .type (nat 0))
      (\n\IHn
         Coq.Init.Logic.f_equal
           .type .type Coq.Init.Datatypes.S n (Coq.Init.Peano.plus n (nat 0))
           IHn) n
Coq.Init.Peano.plus_n_Sm =
  \n\m
    Coq.Init.Datatypes.nat_ind
      .type (Coq.Init.Logic.refl_equal .type (Coq.Init.Datatypes.S m))
      (\n\IHn
         Coq.Init.Logic.f_equal
           .type .type Coq.Init.Datatypes.S
           (Coq.Init.Datatypes.S (Coq.Init.Peano.plus n m))
           (Coq.Init.Peano.plus n (Coq.Init.Datatypes.S m)) IHn) n
Coq.Arith.Plus.plus_comm =
  \n\m
    Coq.Init.Datatypes.nat_ind
      .type (Coq.Init.Peano.plus_n_O m)
      (\y\H
         Coq.Init.Logic.eq_ind
           .type (Coq.Init.Datatypes.S (Coq.Init.Peano.plus m y)) .type
           (Coq.Init.Logic.f_equal
              .type .type Coq.Init.Datatypes.S (Coq.Init.Peano.plus y m)
              (Coq.Init.Peano.plus m y) H)
           (Coq.Init.Peano.plus m (Coq.Init.Datatypes.S y))
```

Example: commutativity of $+_{\mathsf{nat}}$ (optimized realizer)

```
Coq.Arith.Plus.plus_comm =
  \n\m\z z
```

Conclusion & Perspectives

- first real use of classical realizability & extraction
- certified algorithm to extract Herbrand trees
- strong links between inconsistency proof and extracted tree
  $\rightsquigarrow$ improvement over Herbrand's proof
- we still do not understand how the program computes

- improve performances to face real-life examples
- optimization theory for classical realizability

Conclusion & Perspectives

- first real use of classical realizability & extraction
- certified algorithm to extract Herbrand trees
- strong links between inconsistency proof and extracted tree
  $\rightsquigarrow$ improvement over Herbrand's proof
- we still do not understand how the program computes

- improve performances to face real-life examples
- optimization theory for classical realizability

Thank you