From Z3 to Lean, Efficient Verification

Turing Gateway to Mathematics, 19 July 2017

Leonardo de Moura, Microsoft Research

Z3 Theorem Prover

Joint work with Nikolaj Bjorner
and Christoph Wintersteiger

DPLL

Simplex

Rewriting

Superposition

Z3 is a collection of
Symbolic Reasoning Engines

Congruence Closure

Groebner Basis

∀∃ elimination

Euclidean Solver

Satisfiability

Solution/Model

$x^2 + y^2 < 1 \; and \; xy > 0.1$ → sat, $x = \frac{1}{8}, y = \frac{7}{8}$

$x^2 + y^2 < 1 \; and \; xy > 1$ → unsat, Proof

Is execution path *P* feasible? Is assertion *X* violated?

SAGE

Is Formula *F* Satisfiable?

Symbolic Reasoning Engine

- Test Case Generation
- Verifying Compilers
- Invariant Generation
- Model Based Testing
- Type Checking
- Model Checking

Some Applications at Microsoft

SAGE

BOOGIE

HAVOC

The Spec# Programming System

Vcc

SLAM

FORMULA
Modeling Foundations.

TERMINATOR

Spec Explorer

Yogi

Vigilante

Hyper-V
Microsoft | Virtualization

Pex

Impact

Used by many research groups

Awards:

- "The most influential tool paper in the first 20 years of TACAS"(> 3500 citations)
- Programming Languages Software Award from ACM SIGPLAN

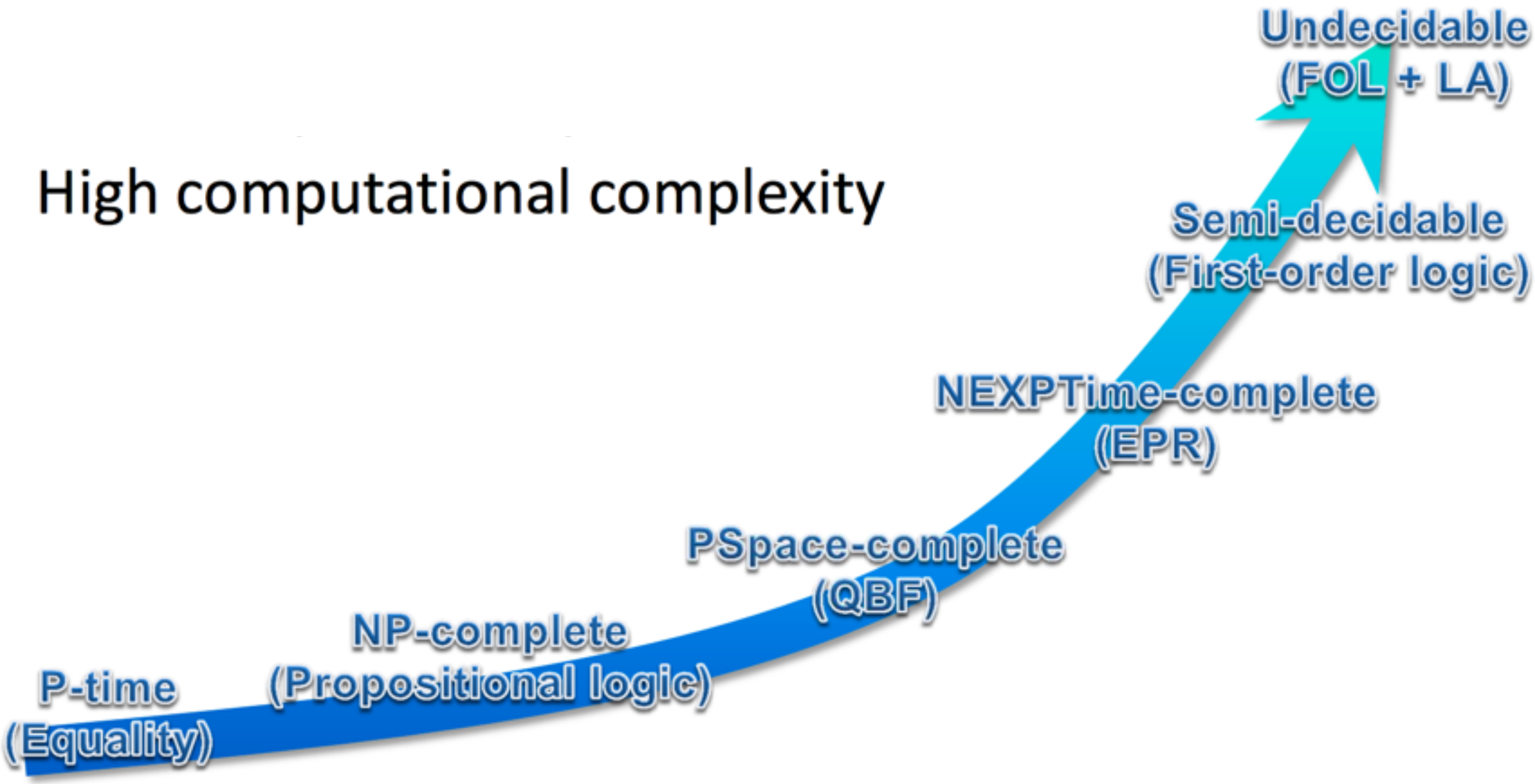
Ships with many popular systems

- Isabelle, Pex, SAGE, SLAM/SDV, Visual Studio, …

Solved more than 5 billion constraints created by SAGE when checking Win8/Office

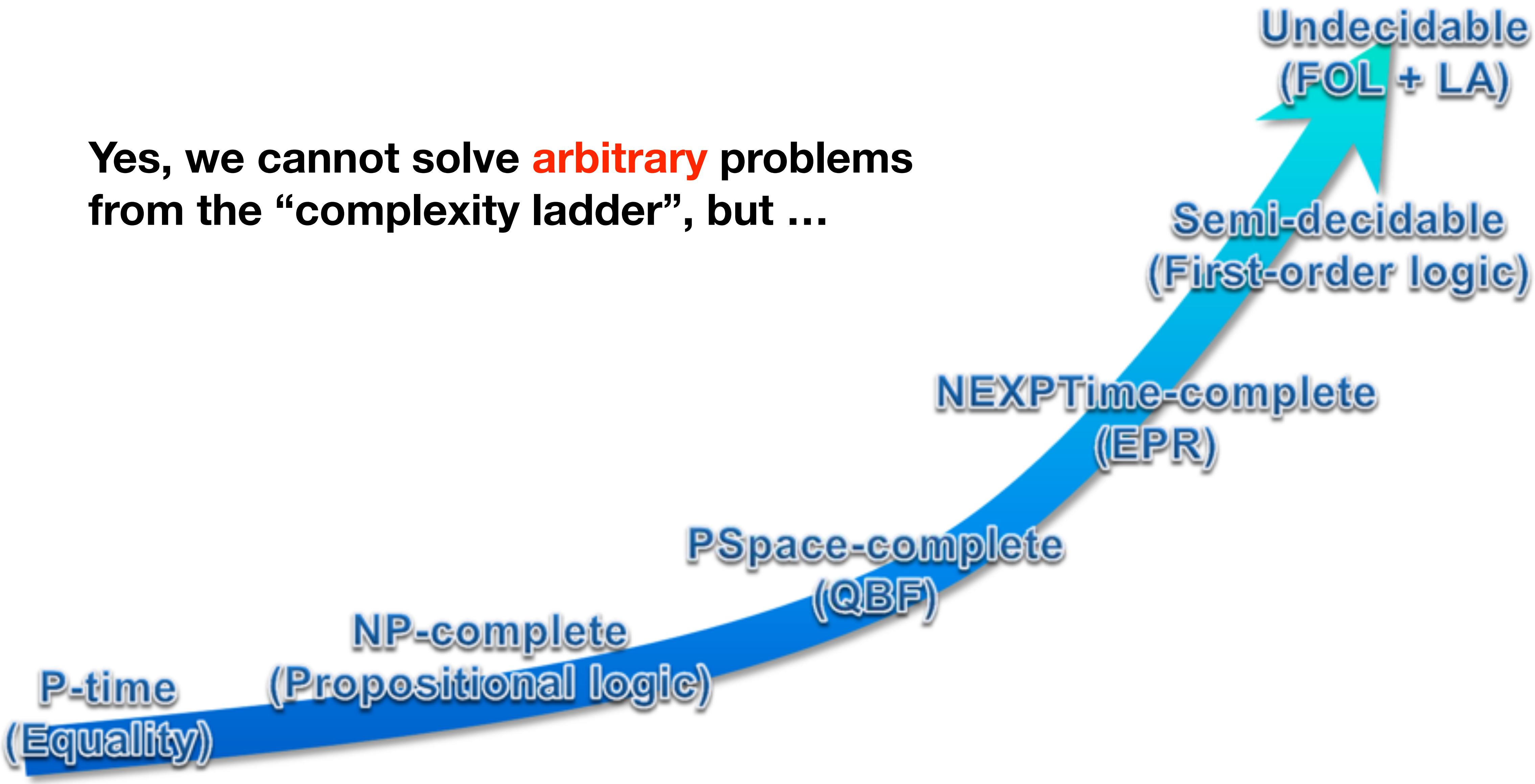
Logic is “the calculus of computer science”
(Z. Manna)

High computational complexity



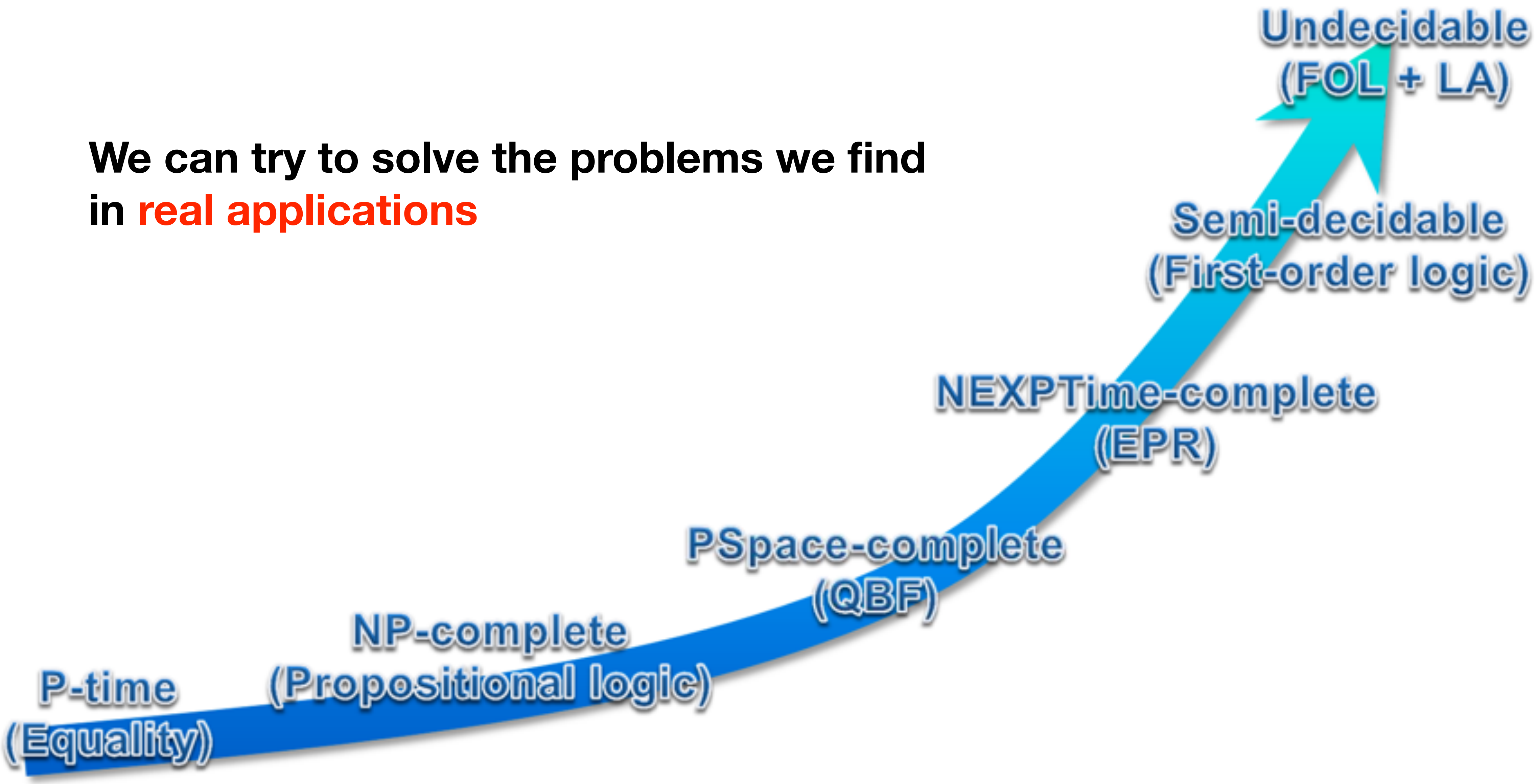
Logic is “the calculus of computer science” (Z. Manna)

Yes, we cannot solve **arbitrary** **problems from the “complexity ladder”, but …**



Logic is "the calculus of computer science" (Z. Manna)

We can try to solve the problems we find in real applications

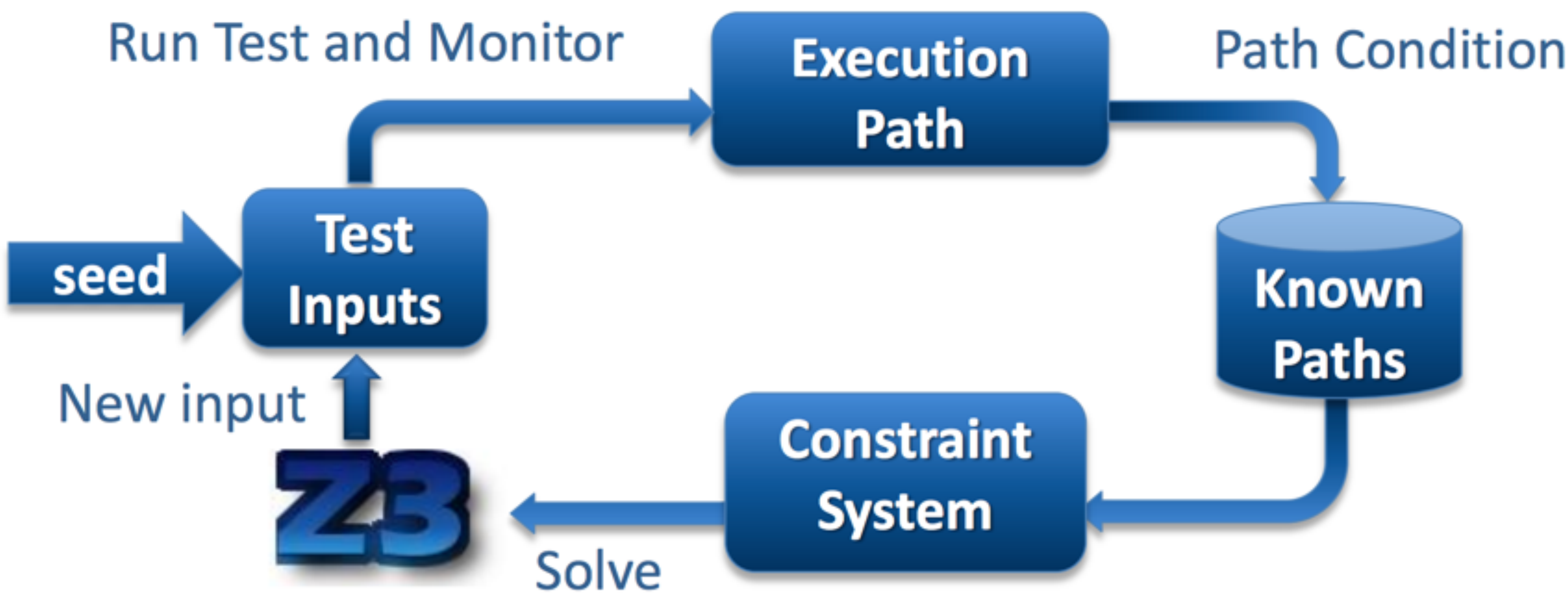


Security is critical

- Security bugs can be very expensive
  - Cost of each MS security bulletin: $millions
  - Cost due to worms: $billions
- Most security exploits are initiated via files or packets
  - Ex: Internet browsers parse dozens of file formats
- Security testing: hunting million dollar bugs

Directed Automated Random Testing

SAGE (one of the most successful Z3 applications)
developed by Patrice Godefroid

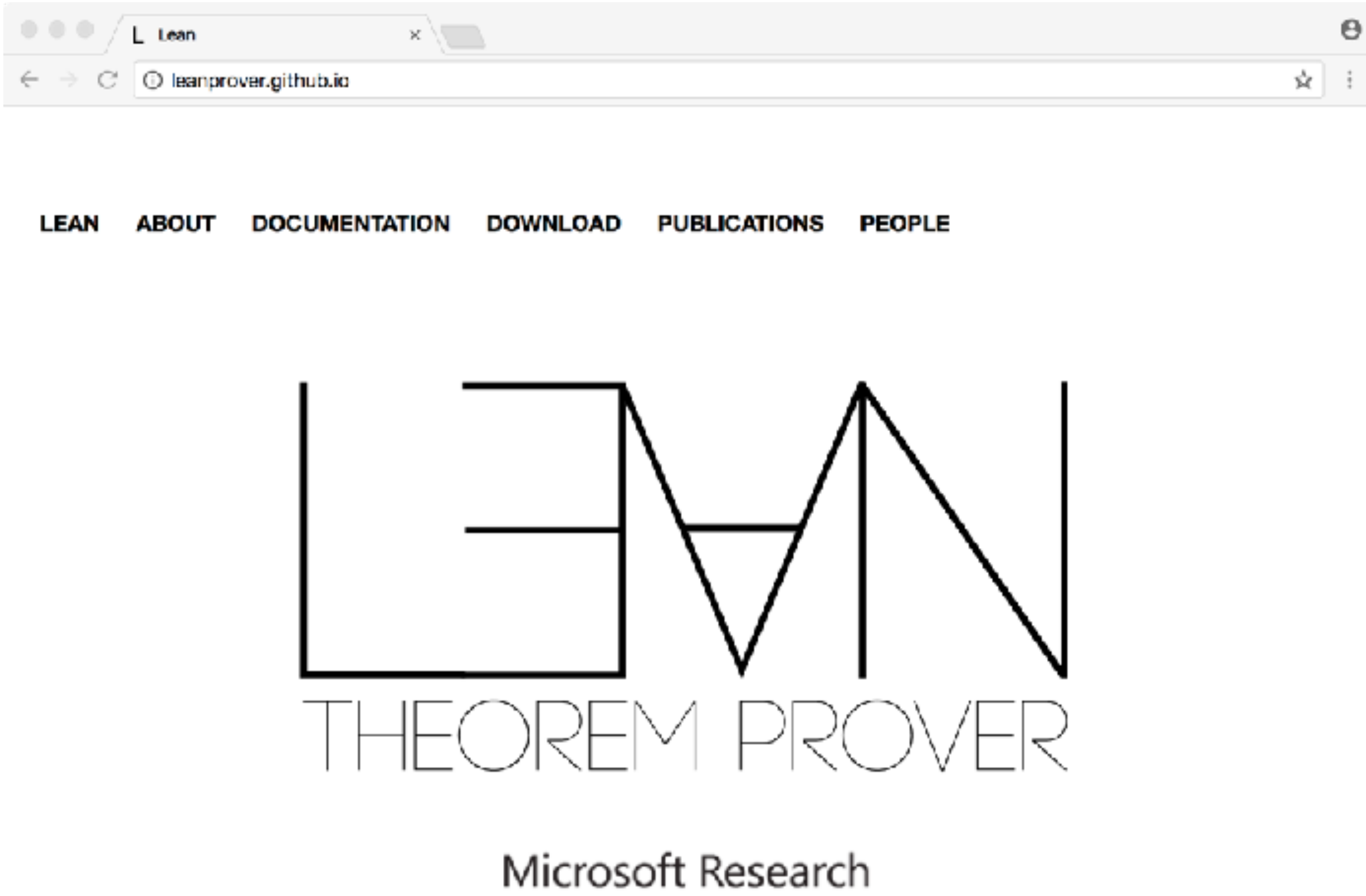


Software verification

- Specifications
  - Methods contracts
  - Invariants
  - Field and type annotations
- Program logic: Dijkstra's weakest precondition
- Verification condition generation

Software verification & automated provers

- Easy to use for simple properties
- Main problems:
  - Scalability issues
  - Proof stability
- in many verification projects:
  - Hyper-V
  - Ironclad & Ironfleet (https://github.com/Microsoft/Ironclad)
  - Everest (https://project-everest.github.io/)



joint work with Jeremy Avigad, Mario Carneiro, Floris van Doorn, Gabriel Ebner, Johannes Hölzl, Rob Lewis, Jared Roesch, Daniel Selsam and Sebastian Ullrich

Lean aims to bridge the gap between interactive and automated theorem proving

Lean

- New open source theorem prover (and programming language)

  Soonho Kong and I started coding in the Fall of 2013
- Platform for
  - Software verification
  - Formalized Mathematics
  - Domain specific languages
- de Bruijn's principle: small trusted kernel
- Dependent Type Theory
- Partial constructions: automation fills the "holes"

Inductive Families

```
inductive nat
| zero : nat
| succ : nat → nat

inductive tree (α : Type u)
| leaf : α → tree
| node : tree → tree → tree

inductive vector (α : Type) : nat → Type
| nil  : vector zero
| cons : Π {n : nat}, α → vector n → vector (succ n)
```

Recursive equations

```
def fib : nat → nat
| 0     := 1
| 1     := 1
| (a+2) := fib a + fib (a + 1)
```

```
def ack : nat → nat → nat
| 0     y     := y+1
| (x+1) 0     := ack x 1
| (x+1) (y+1) := ack x (ack (x+1) y)
```

Proofs

```
theorem ring_mul_zero {α : Type u} [ring α] (a : α) : a * 0 = 0 :=
have a * 0 + 0 = a * 0 + a * 0, from calc
     a * 0 + 0 = a * 0           : add_zero (a*0)
           ... = a * (0 + 0)     : by simp
           ... = a * 0 + a * 0 : left_distrib a 0 0,
show a * 0 = 0, from (add_left_cancel this).symm
```

Metaprogramming

- Introduced in Lean 3 (mid 2016)
- Extend Lean using Lean
- Access Lean internals using Lean
  - Type inference
  - Unifier
  - Simplifier
  - Decision procedures
  - Type class resolution
  - …
- Proof/Program synthesis

Metaprogramming

```
meta def find : expr → list expr → tactic expr
| e []          := failed
| e (h :: hs) :=
  do t ← infer_type h,
     (unify e t >> return h) <|> find e hs

meta def assumption : tactic unit :=
do { ctx ← local_context,
     t   ← target,
     h   ← find t ctx,
     exact h }
<|> fail "assumption tactic failed"

lemma simple (p q : Prop) (h₁ : p) (h₂ : q) : q :=
by assumption
```

Reflecting expressions

```
inductive level
| zero  : level
| succ  : level → level
| max   : level → level → level
| imax  : level → level → level
| param : name → level
| mvar  : name → level
```

```
inductive expr
| var    : nat → expr
| lconst : name → name → expr
| mvar   : name → expr → expr
| sort   : level → expr
| const  : name → list level → expr
| app    : expr → expr → expr
| lam    : name → binfo → expr → expr → expr
| pi     : name → binfo → expr → expr → expr
| elet   : name → expr → expr → expr → expr
```

```
meta def num_args : expr → nat
| (app f a) := num_args f + 1
| e         := 0
```

Superposition prover

- 2200 lines of code

```
example {α} [monoid α] [has_inv α] : (∀ x : α, x * x⁻¹ = 1) →
                                       ∀ x : α, x⁻¹ * x = 1 :=
by super with mul_assoc mul_one

meta structure prover_state :=
(active passive : rb_map clause_id derived_clause)
(newly_derived : list derived_clause) (prec : list expr)
(locked : list locked_clause) (sat_solver : cdcl.state)
...
meta def prover := state_t prover_state tactic
```

dlist

```
structure dlist (α : Type u) :=
(apply     : list α → list α)
(invariant : ∀ l, apply l = apply [] ++ l)
```

```
def to_list : dlist α → list α
| ⟨xs, _⟩ := xs []
```

```
local notation `♯`:max := by abstract {intros, rsimp}
```

```
/-- `O(1)` Append dlists -/
protected def append : dlist α → dlist α → dlist α
| ⟨xs, h₁⟩ ⟨ys, h₂⟩ := ⟨xs ∘ ys, ♯⟩

instance : has_append (dlist α) :=
⟨dlist.append⟩
```

transfer tactic

- Developed by Johannes Hölzl (approx. 200 lines of code)

```
lemma to_list_append (l₁ l₂ : dlist α) : to_list (l₁ ++ l₂) = to_list l₁ ++ to_list l₂ :=
show to_list (dlist.append l₁ l₂) = to_list l₁ ++ to_list l₂, from
by cases l₁; cases l₂; simp; rsimp
```

```
protected def rel_dlist_list (d : dlist α) (l : list α) : Prop :=
to_list d = l
```

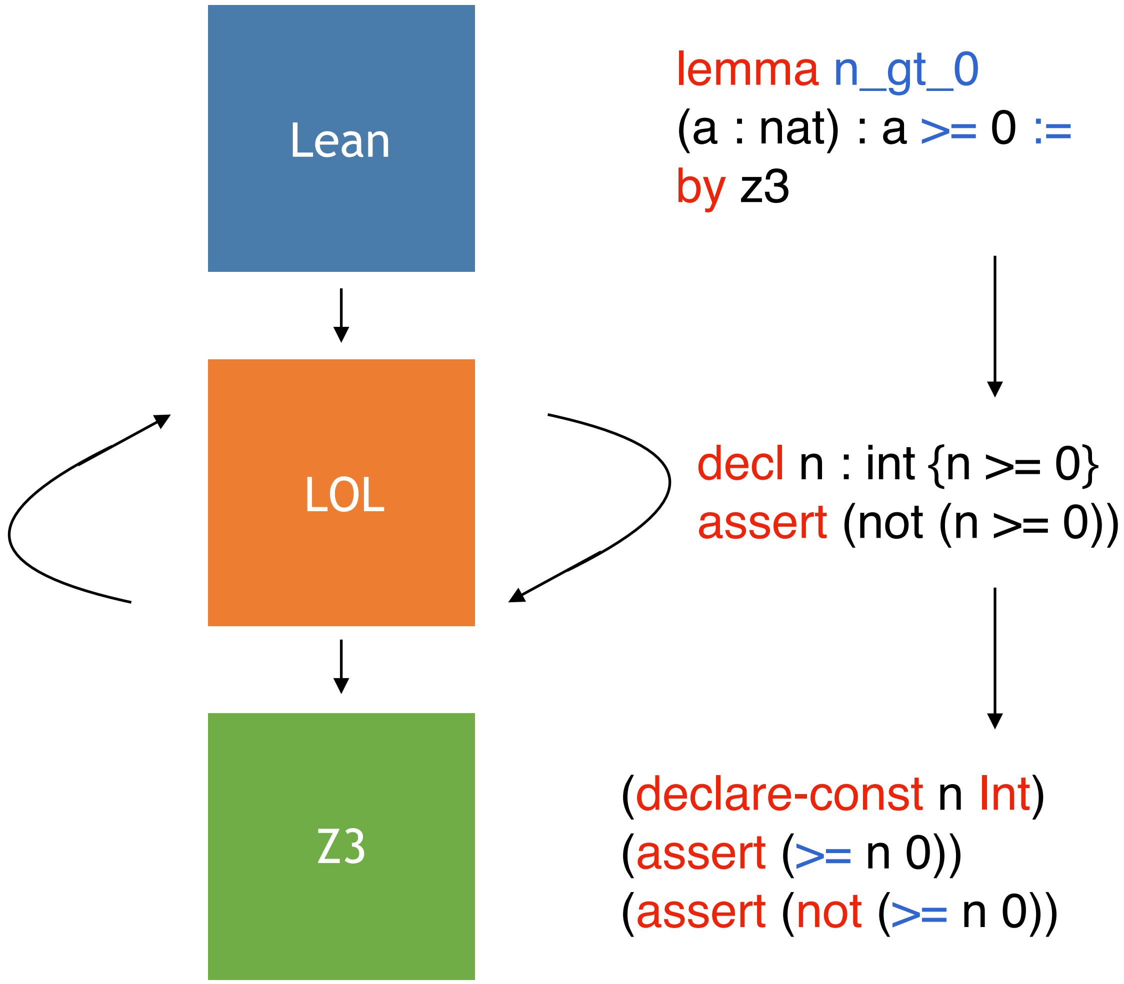
```
protected meta def transfer : tactic unit := do
  _root_.transfer.transfer [`relator.rel_forall_of_total, `dlist.rel_eq, `dlist.rel_empty,
    `dlist.rel_singleton, `dlist.rel_append, `dlist.rel_cons, `dlist.rel_concat]

example : ∀(a b c : dlist α), a ++ (b ++ c) = (a ++ b) ++ c :=
begin
  dlist.transfer,
  intros,
  simp
end
```

- We also use it to transfer results from nat to int.

Lean to Z3

- Goal: translate a Lean local context, and goal into Z3 query.
- Recognize fragment and translate to low-order logic (LOL).
- Logic supports some higher order features, is successively lowered to FOL, finally Z3.



simple expression language

```
inductive exp : Type
| Const (n : nat) : exp
| Plus (e1 e2 : exp) : exp
| Mult (e1 e2 : exp) : exp

def eeval : exp → nat
| (Const n)     := n
| (Plus e1 e2) := eeval e1 + eeval e2
| (Mult e1 e2) := eeval e1 * eeval e2

def times (k : nat) : exp → exp
| (Const n)     := Const (k * n)
| (Plus e1 e2) := Plus (times e1)
                       (times e2)
| (Mult e1 e2) := Mult (times e1) e2
```

```
def reassoc : exp → exp
| (Const n)     := (Const n)
| (Plus e1 e2) :=
  let e1' := reassoc e1 in
  let e2' := reassoc e2 in
  match e2' with
  | (Plus e21 e22) := Plus (Plus e1' e21) e22
  | _              := Plus e1' e2'
  end
| (Mult e1 e2) :=
  let e1' := reassoc e1 in
  let e2' := reassoc e2 in
  match e2' with
  | (Mult e21 e22) := Mult (Mult e1' e21) e22
  | _              := Mult e1' e2'
  end
```

Writing your own search strategies

```
meta def try_list {α} (tac : α → tactic unit) : list α → tactic unit
| []      := failed
| (e::es) := (tac e >> done) <|> try_list es

meta def induct (tac : tactic unit) : tactic unit :=
collect_inductive_hyps >>= try_list (λ e, induction' e; tac)

meta def split (tac : tactic unit) : tactic unit :=
collect_inductive_from_target >>= try_list (λ e, cases e; tac)

meta def search (tac : tactic unit) : nat → tactic unit
| 0     := try tac >> done
| (d+1) := try tac >> (done <|> all_goals (split (search d)))

meta def nano_crush (depth : nat := 1) :=
do hs ← mk_relevant_lemmas, induct (search (rsimp' hs) depth)
```

simple expression language

```
lemma eeval_times (k e) : eeval (times k e) = k * eeval e := by nano_crush
lemma reassoc_correct (e) : eeval (reassoc e) = eeval e := by nano_crush
```

Conclusion

- SMT solvers (like Z3) are very successful in bug finding tools
- Scalability and proof stability issues
- Lean aims to bring the best of automated and interactive systems
- Users can create their on automation, extend and customize Lean
- Domain specific automation
- Internal data structures and procedures are exposed to users
- Whitebox automation