Lost in translation

how easy problems become hard due to bad encodings

Vampire Workshop 2015

Leonardo de Moura
Microsoft Research

I wanted to give the following talk

LEAN THEOREM PROVER

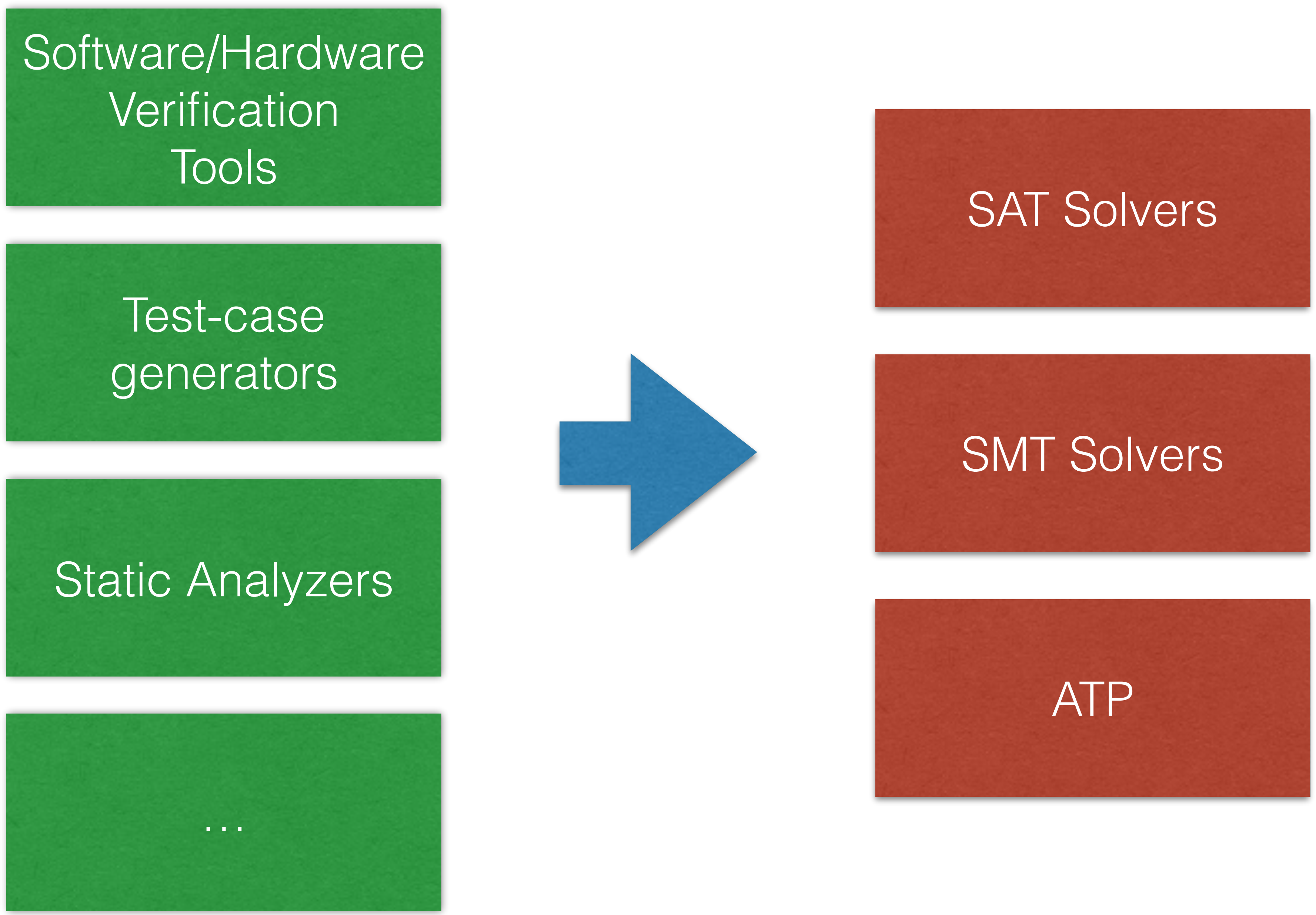
Microsoft Research

↔

Vampire

http://leanprover.github.io/

Automated Reasoning Tools as a service



The “dream”

Automated reasoning tools as black boxes

SAT Solvers

SMT Solvers

ATP

Example 1:
SAT solvers and Tseitin encoding

- Most SAT solvers expect the input formula to be in CNF

- In practice, it is not feasible to convert formulas into CNF using equivalences such as

eliminate $\Rightarrow$	$A \Rightarrow B \equiv \neg A \vee B$
reduce the scope of $\neg$	$\neg(A \vee B) \equiv \neg A \wedge \neg B$, $\neg(A \wedge B) \equiv \neg A \vee \neg B$
apply distributivity	$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$, $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$

Example 1:
SAT solvers and Tseitin encoding

However, there is a *linear time* translation to CNF that produces an *equisatisfiable* formula. Replace the distributivity rules by the following rules:

$$\frac{F[l_i \ op \ l_j]}{F[x], x \Leftrightarrow l_i \ op \ l_j}*$$

$$\frac{x \Leftrightarrow l_i \vee l_j}{\neg x \vee l_i \vee l_j, \neg l_i \vee x, \neg l_j \vee x}$$

$$\frac{x \Leftrightarrow l_i \wedge l_j}{\neg x \vee l_i, \neg x \vee l_j, \neg l_i \vee \neg l_j \vee x}$$

(*) x must be a fresh variable.

Example 1:
SAT solvers and Tseitin encoding

- Tseitin encoding is easy to implement.
- However, there are several important improvements.
  - Example: detect common sub- formulas.
- SAT preprocessors (such as SatELite) “fix” naive CNF encodings before invoking the actual SAT solver
- Good: preprocessors are reused by different research groups

Example 2:
Finite model finding & symmetry breaking

- Given a first-order logic formula F, find a finite model M for it
- Procedures: reduce to SAT (or SMT), reduce to EPR
- MACE-style reduction
  - Fix domain D = {1, …, n}
  - Create propositional variables for each predicate P and argument vector (d_1, …, d_k) where k is the rarity of P and d_i in D
  - Similarly, one proposition $p_{f,v,r}$ application for each function f argument vector v = (d_1, …, d_k) and “result” r in D
  - Convert F into CNF, instantiate, and add
    - function definition constraint: (not $p_{f,v,r}$) or (not $p_{f,v,r'}$)
    - totality constraints: ($p_{f,v,1}$ or … or $p_{f,v,n}$)

Example 2:
Finite model finding & symmetry breaking

- The encoding into SMT is simpler. Example: we can use the theory of uninterpreted functions and avoid function definition and totality constraints.

- Symmetry reduction is a very important optimization (in both cases).

  - The MACE-style encoding implies that for each model, all of its isomorphic valuations (obtained by permuting the domain elements) are also models.

  - Idea: add symmetry breaking constraints that force that the model we are looking for has a certain canonical form.

Example 2:
Finite model finding & symmetry breaking

- Suppose the problem encoder did not include the symmetry breaking problems.

- Now, to achieve good performance the solver developer must try to infer the symmetries (a much harder problem). See

  "SyMT: finding symmetries in SMT formulas", by Carlos Areces, David Deharbe, Pascal Fontaine and Ezequiel Orbe

- SMT-LIB has as huge set of finite model finding (QF_UF) problems where symmetry breaking constraints have not been added.

- Consequently, many SMT solvers (e.g., CVC4, veriT, Yices, Z3) do implement SyMT-like procedures to be able to solve these problems efficiently.

Example 3: Sledgehammer

- Sledgehammer is a very successful tool available in the Isabelle Proof assistant
- It converts HOL into FOL and invokes many ATPs (Vampire) and SMT solvers (Z3)
- A lot is lost in the translation.
- Sledgehammer may fail in very simple queries because they are higher-order, but it will succeed once the user has, for example, manually unfolded some definitions.
- We need provers/solvers that can understand HOL and perform proofs by induction. Even if it is just thin layer.

Example 4:
Nonlinear arithmetic solvers

- Nonlinear (real polynomial) arithmetic is decidable

$$x^2 - 4x + y^2 - y + 8 < 1$$
$$xy - 2x - 2y + 4 > 1$$

- Expensive decision procedure

- Most efficient complete solvers are based on Cylindrical Algebraic Decomposition (CAD)

- Perform computations with real algebraic numbers

Example 4:
Nonlinear arithmetic solvers

- Real algebraic numbers

Polynomial + Isolating Interval

$$x^2 - 2, (1, 2)$$

$$\sqrt[3]{\frac{1}{9}} - \sqrt[3]{\frac{2}{9}} + \sqrt[3]{\frac{4}{9}} \overset{?}{=} \sqrt[3]{\sqrt[3]{2} - 1}$$

$$x^9 + 3x^6 + 3x^3 - 1, (0,1)$$

Example 4: Nonlinear arithmetic solvers

- Bad application: object placement in 3D virtual world (constraints of the form *distance(a, b)* $< n$)
  - Precision is not important: algebraic numbers are an overkill for this kind of application
- Avoiding real-algebraic numbers
  - replace $p = 0$ with $-\delta < p < \delta$ (for a small δ)
  - replace $p \leqq 0$ with $p < \delta$
  - The resulting problem is satisfiable iff it has a rational model. This trick only works if the solver takes the property into account.
  - Remark: we should not apply this transformation to linear equalities since they can be easily eliminated using variable substitution
    - Example, given $x+y+2=0$, replace x with $-y-2$ “everywhere” and delete equation.

Example 4:
Nonlinear arithmetic solvers

- Bad application: object placement in 3D virtual world (constraints of the form *distance(a, b)* $< n$)
  - Precision is not important: algebraic numbers are an overkill for this kind of application
- Avoiding real-algebraic numbers
  - replace $p = 0$ with $-\delta < p < \delta$ (for a small δ)
  - replace $p \leqq 0$ with $p < \delta$
  - The resulting problem is satisfiable iff it has a rational model. This trick only works if the solver takes the property into account.
  - Remark: we should not apply this transformation to linear equalities since they can be easily eliminated using variable substitution
    - Example, given $x+y+2=0$, replace x with $-y-2$ “everywhere” and delete equation.

Example 4:
Nonlinear arithmetic solvers

- Bad idea: convert nonlinear real arithmetic into nonlinear integer arithmetic (using fixed point encoding).
  - Replace x with $10^k y$ where y is a fresh integer variable and k is the number of decimal places
- This approximation also avoids algebraic numbers.
- The resulting problem is in an undecidable fragment (Hilbert's 10th problem).
- This encoding was used by a Z3 user.
- Lesson: users must have a rough idea on how the solver works.

Example 5:
Proof checking in dependent type theory

- Proof assistants based on dependent type theory (e.g., Agda, Coq and Lean) have a builtin notion of reduction.
  - Beta-reduction (λx, f x x) (g a) → f (g a) (g a)
  - Eta-reduction (λx, f x) → f
  - Iota-reduction

    nat.primrec c f 0 → c

    nat.primrec c f (succ n) → f n (nat.primrec c f n)
- In these systems, we say that *t* and *s* are *definitionally equal* if there is an *r* such that $t \to r \leftarrow s$
- Zero-step proofs: we can use reflexivity to prove that definitionally equal terms are equal. Example: (refl 4) is a proof for 2+2 = 4 since 2+2 is convertible to 4.

Example 5: Proof checking in dependent type theory

- A naive definitional equality checker for *t* and *s* will simply compute the normal forms for *t* and *s* and check whether they are syntactically equal or not.

- In practice, the naive checker will fail in examples such as

  fact (99+1) and *fact* 100.

- Most proof assistants use the following heuristics for checking whether (*f s*) is definitionally equal to (*f t*).

  - If *s* and *t* are definitionally equal, then return *yes.*

  - Otherwise, unfold *f* and try again.

- (*refl* (*fact* 100)) is a compact proof for *fact* (99+1) = *fact* 100, but it is only feasible to check it if the type/proof checker implements an optimization like the one above.

Flexible solvers and provers

We need more flexible tools.

Customized solutions should be easy to build.

Reuse preprocessors and problem encoders.

Solvers should not be big monolithic black boxes, but a collection of tools and procedures.

Open source tools is a must have.

Efforts such as TPTP and SMT-Lib are fundamental

The strategy challenge

To build theoretical and practical tools allowing users to exert strategic control over core heuristic aspects of high performance prover and solvers.

What is a strategy?

Theorem proving as an exercise of combinatorial search.

Strategies are adaptations of general search mechanisms which reduce the search space by tailoring its exploration to a particular class of formulas.

Different strategies for different domains

From timeout to 0.05secs

QBVF = Quantifiers + Bit-vectors + uninterpreted functions

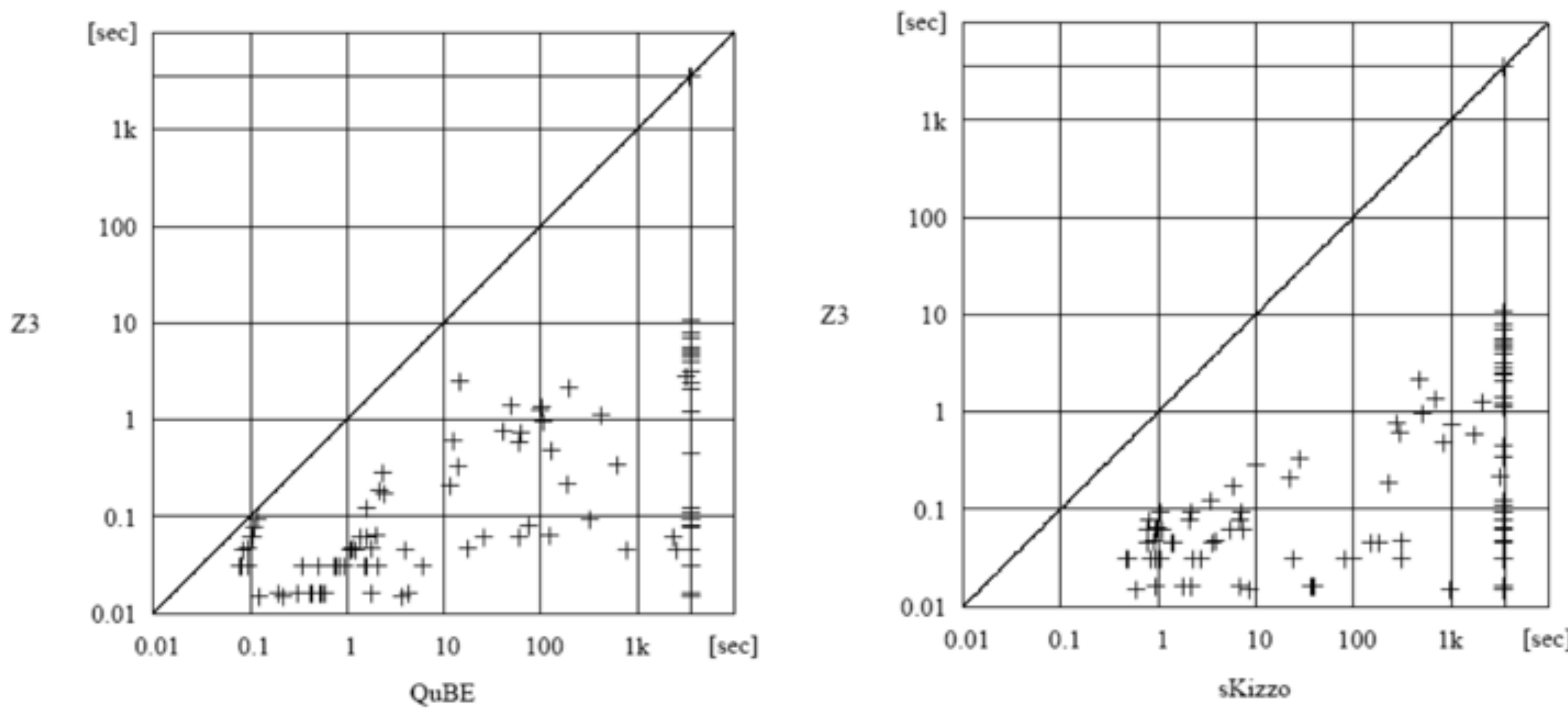
Hardware Fixpoint Checks.

Given: $I[x]$ and $T[x, x']$

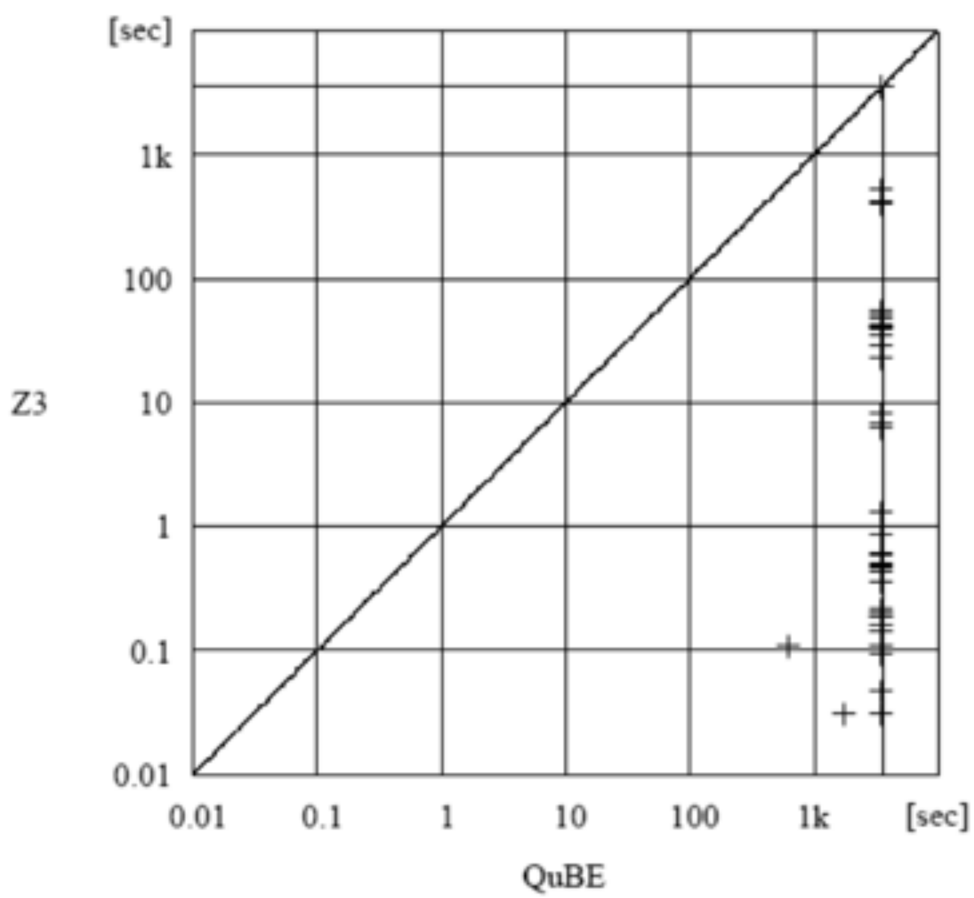
$$\forall x, x' \;.\; I[x] \wedge T^k[x, x'] \rightarrow \exists y, y' \;. I[y] \wedge T^{k-1}[y, y']$$

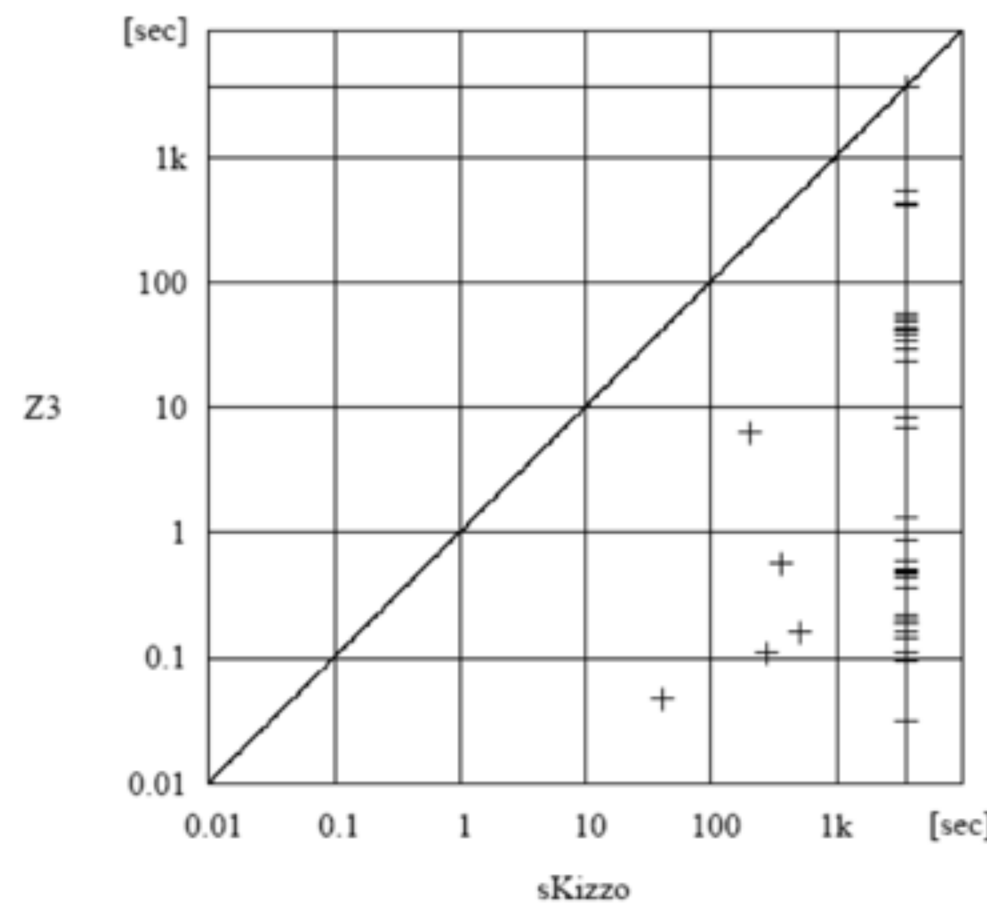
Ranking function synthesis.

Hardware fixpoint checks



Ranking function synthesis



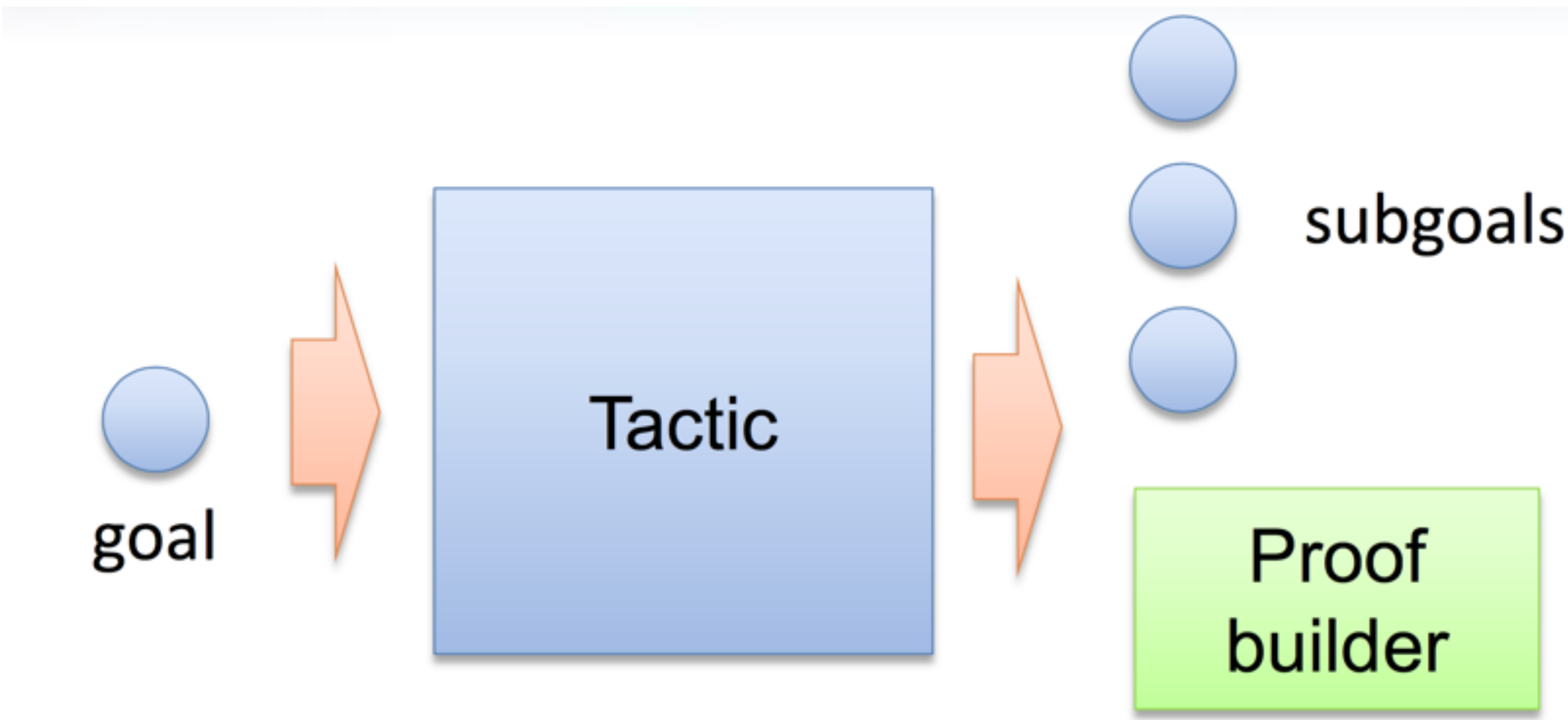


Why is Z3 fast in these benchmarks?

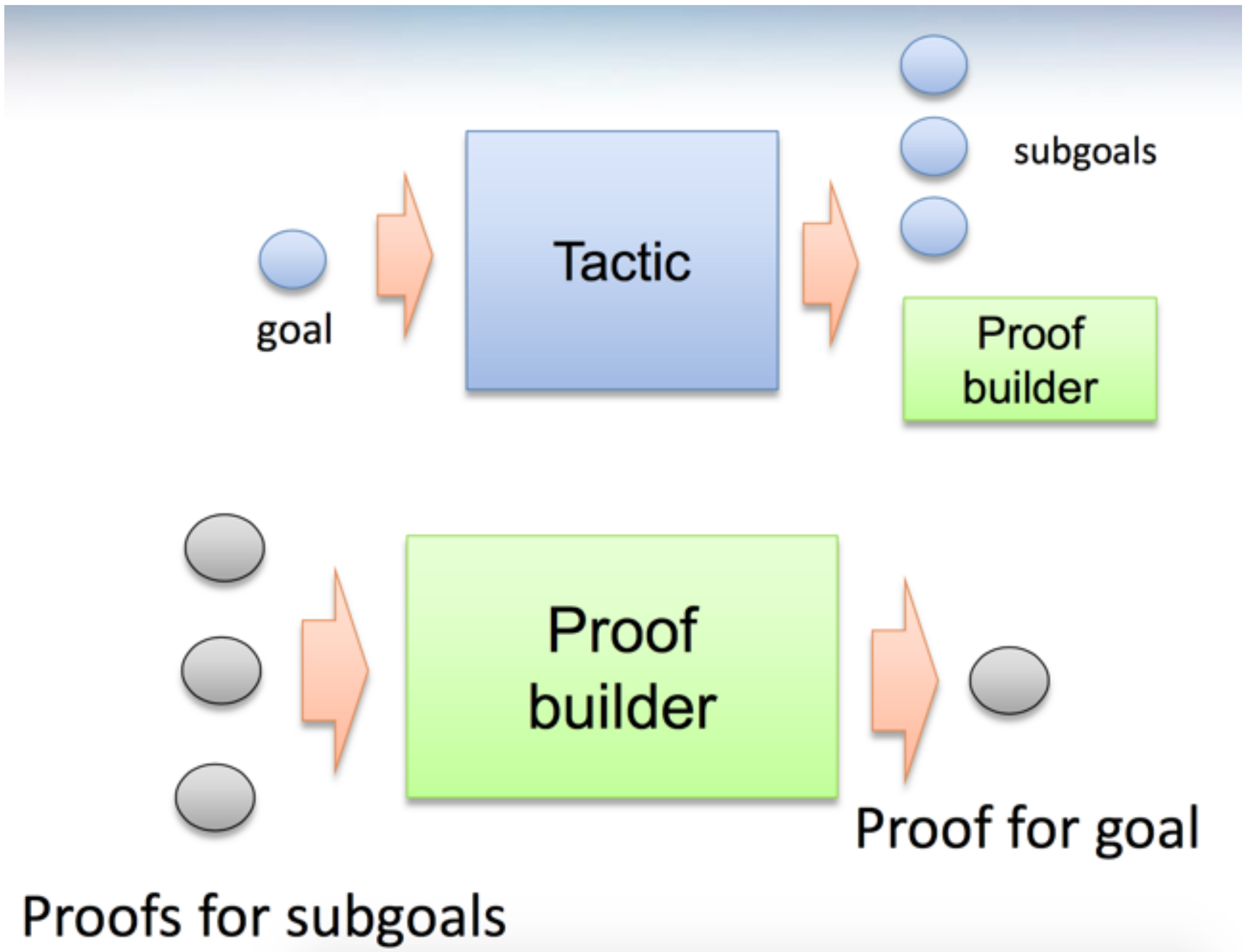
Z3 is using a custom strategy that combines:

- rewriting, SAT, model based quantifier instantiation

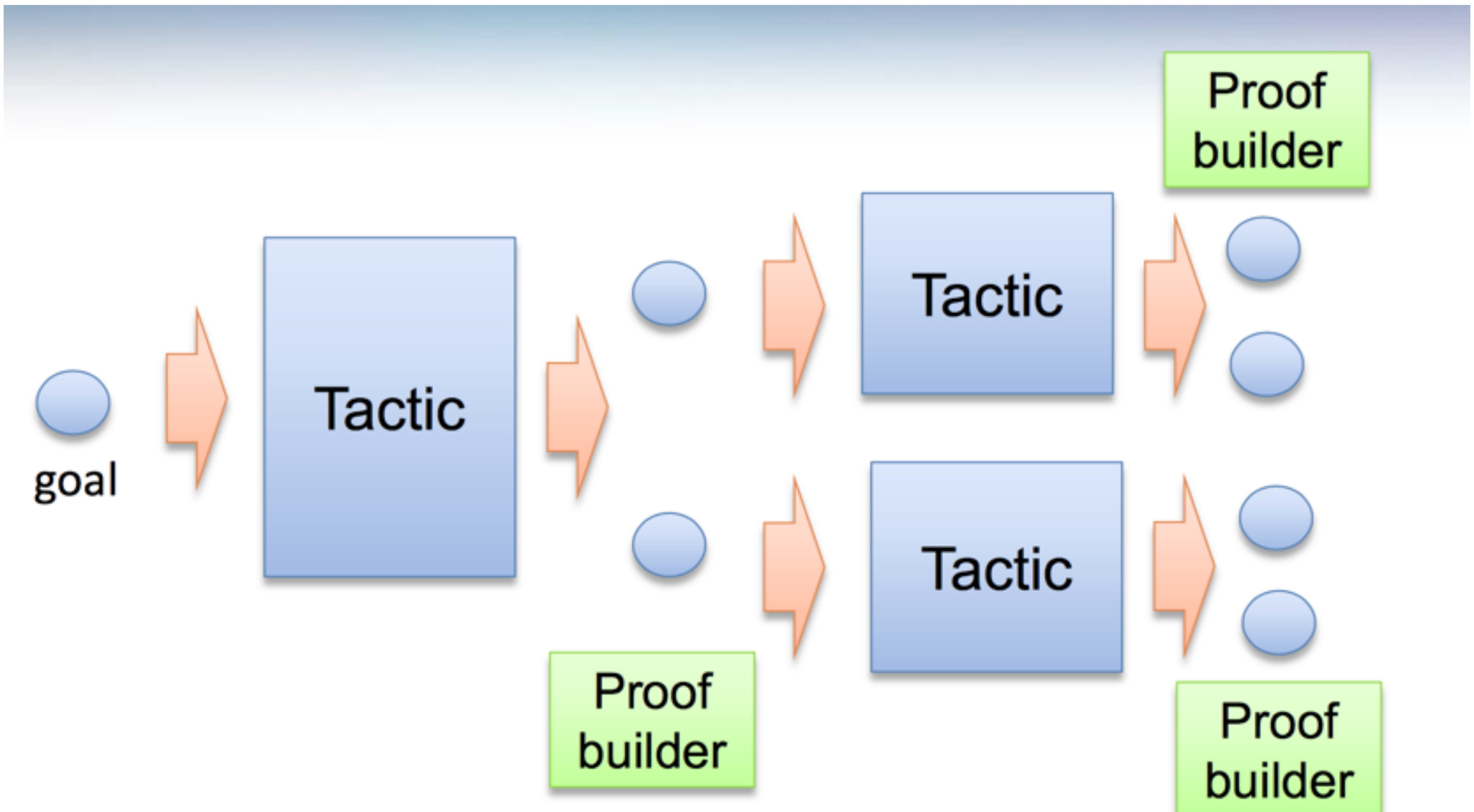
Combining Strategies
Main inspiration: LCF-approach



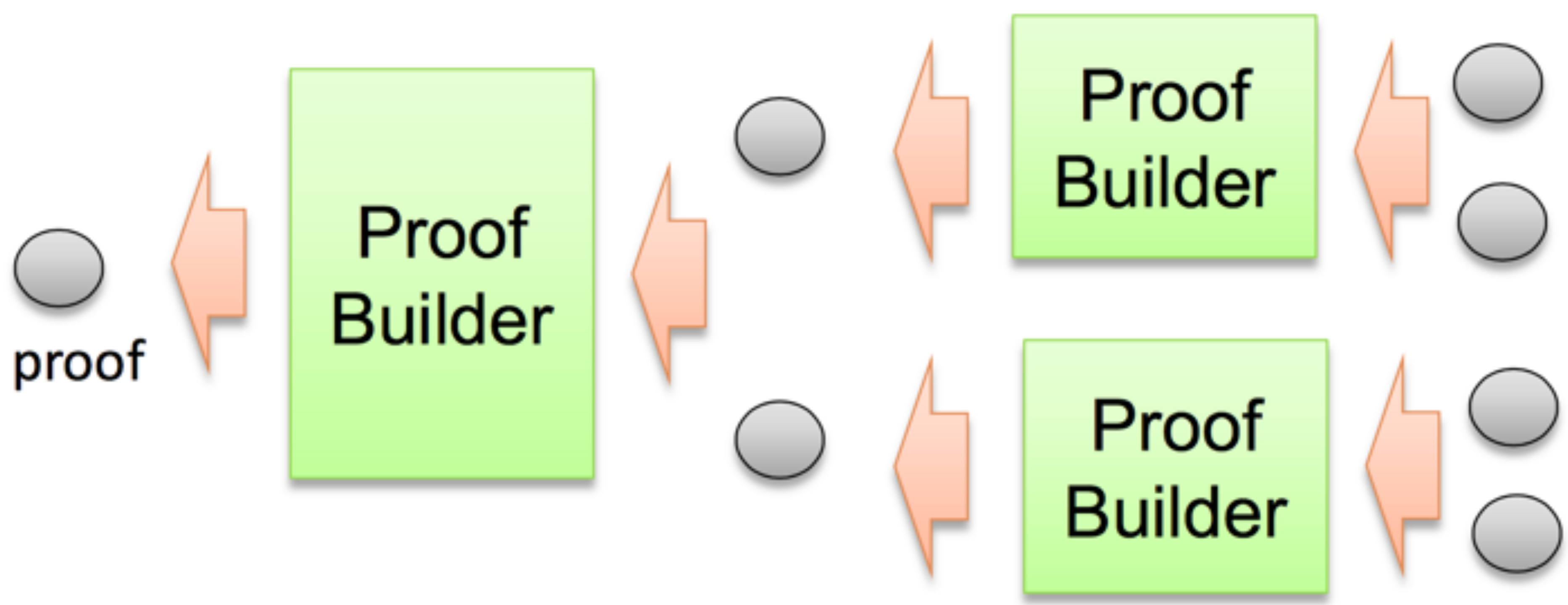
Combining Strategies
Main inspiration: LCF-approach



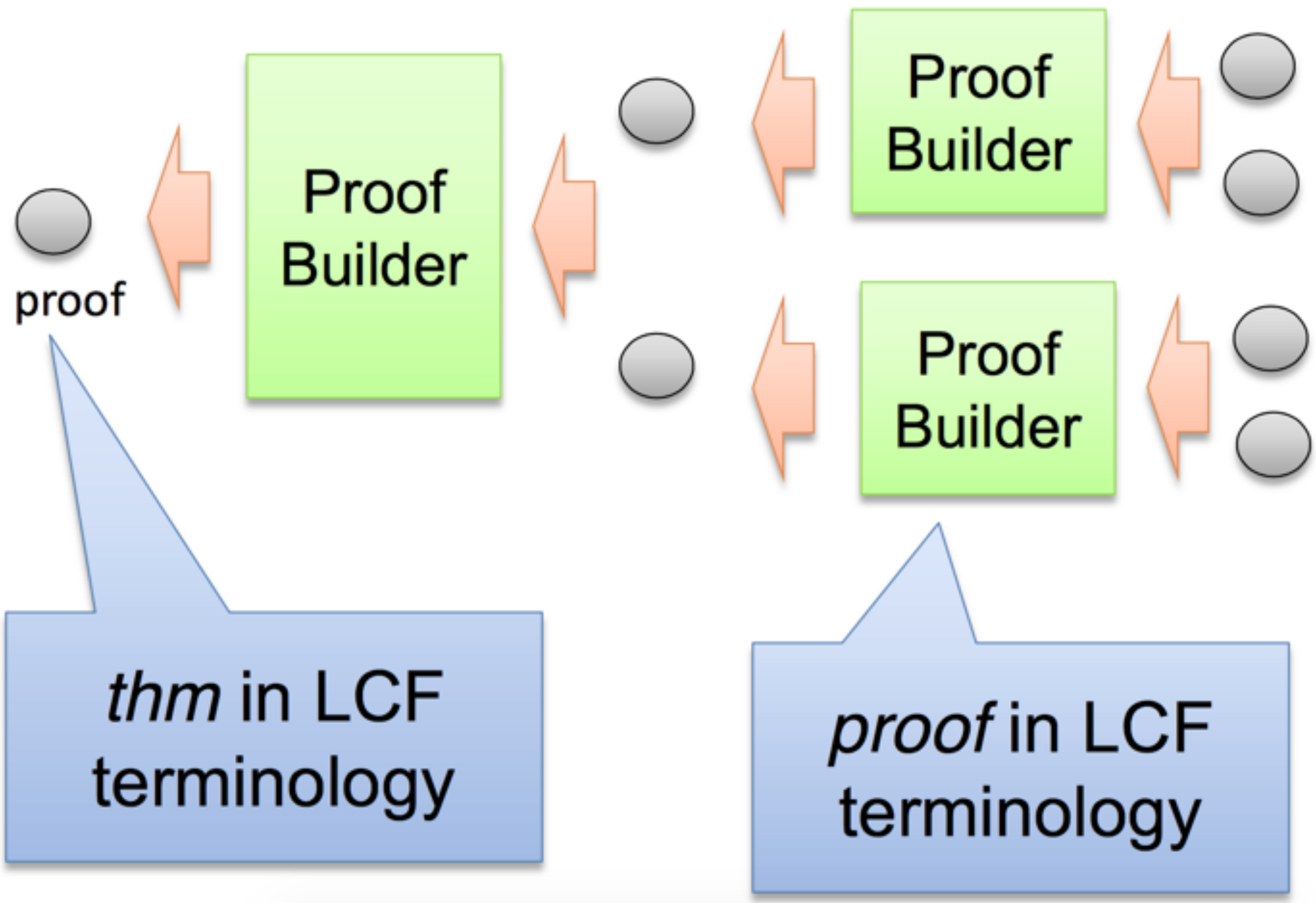
Combining Strategies
Main inspiration: LCF-approach



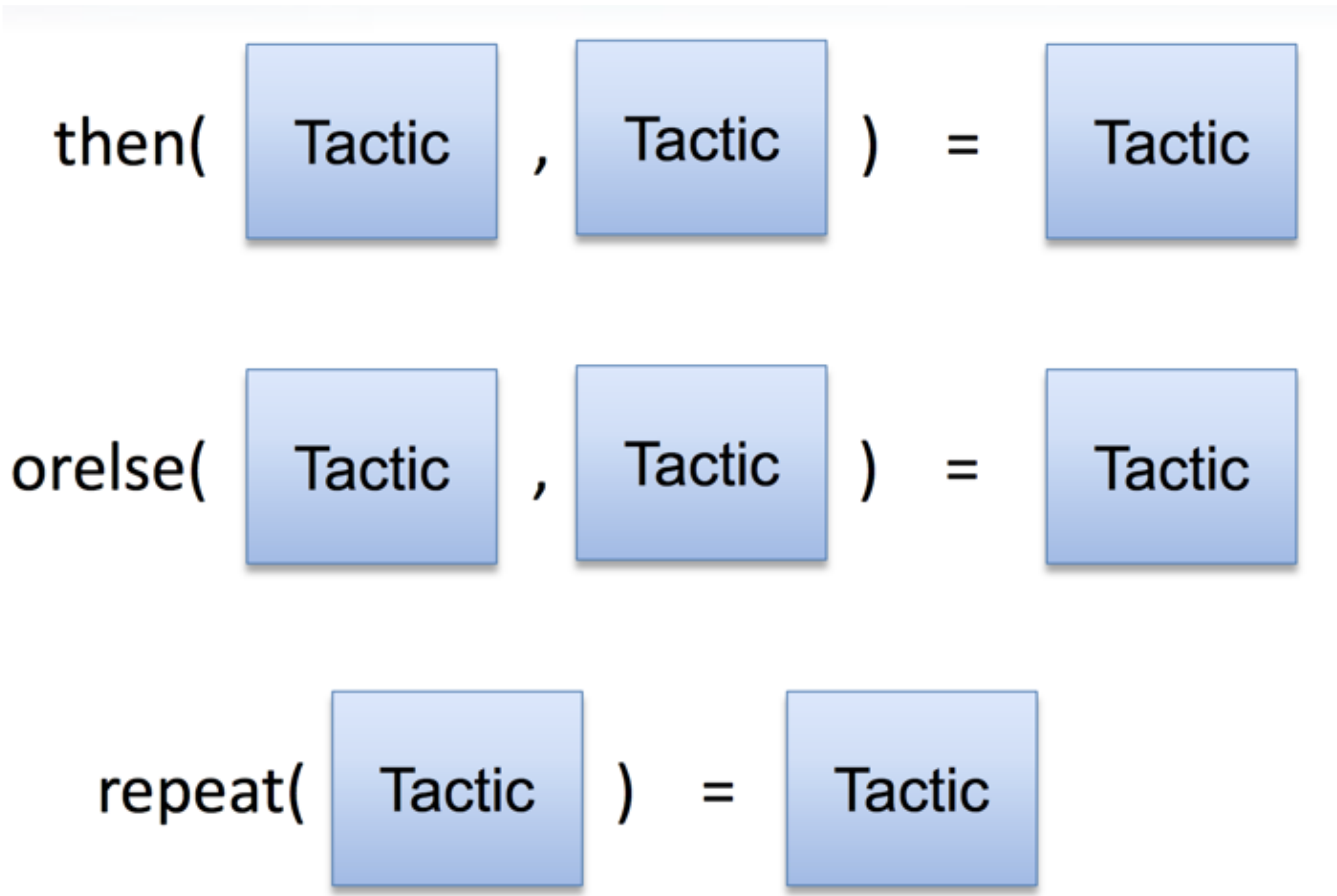
Combining Strategies
Main inspiration: LCF-approach



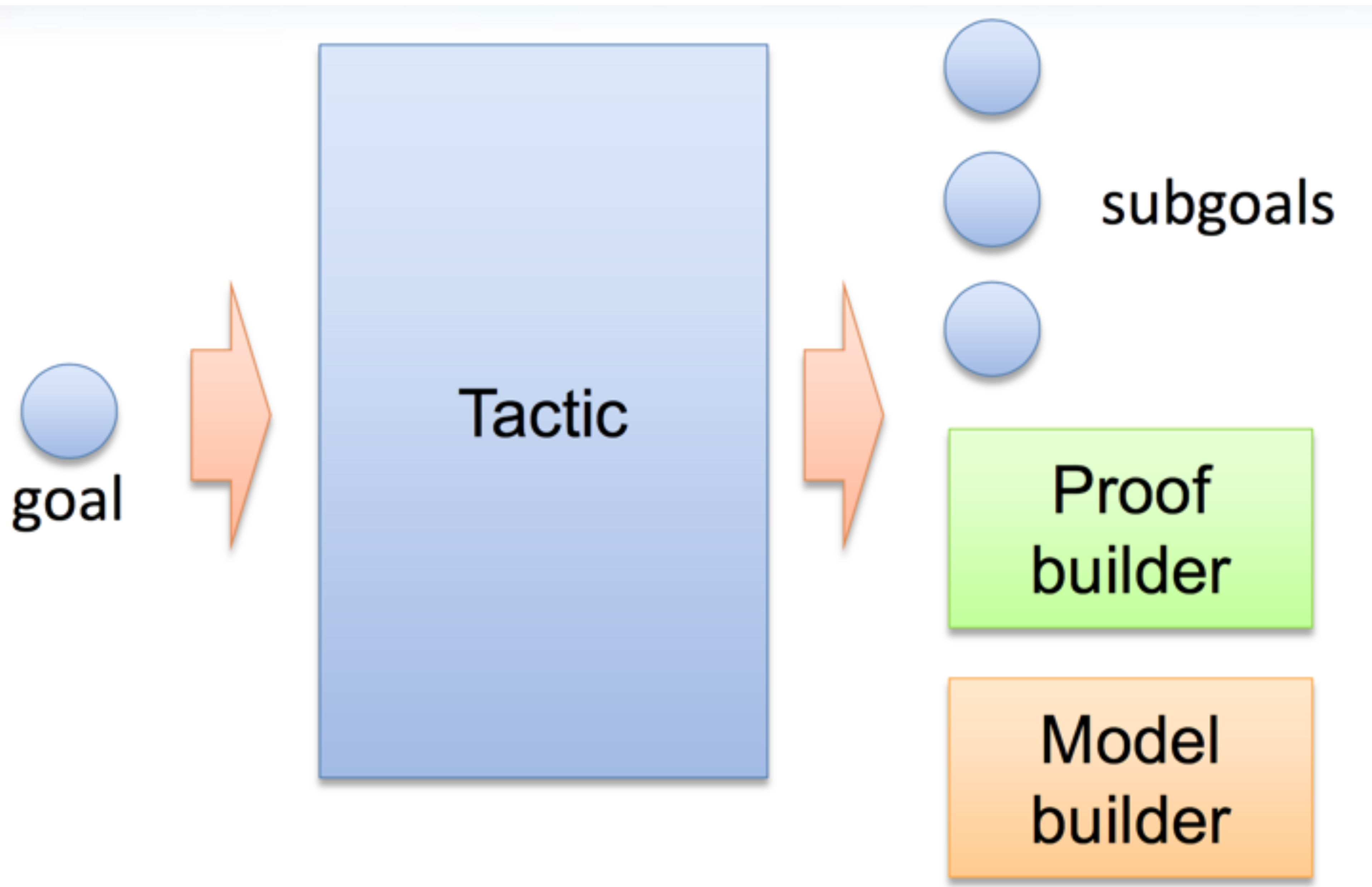
Combining Strategies
Main inspiration: LCF-approach



Tactical: combinators



SMT Tactic



SMT Tactic

$$
\begin{aligned}
goal &= formula\ sequence \times attribute\ sequence \\
\\
proofconv &= proof\ sequence \rightarrow proof \\
modelconv &= model \times nat \rightarrow model \\
trt &= \mathsf{sat}\ model \\
&\mid\ \mathsf{unsat}\ proof \\
&\mid\ \mathsf{unknown}\ goal\ sequence \times modelconv \times proofconv \\
&\mid\ \mathsf{fail} \\
tactic &= goal \rightarrow trt
\end{aligned}
$$

SMT Tactic

$$
\begin{array}{lcl}
\mathit{goal} & = & \mathit{formula\ sequence} \times \mathit{attribute\ sequence} \\
\\
\mathit{proofconv} & = & \mathit{proof\ sequence} \rightarrow \mathit{proof} \\
\mathit{modelconv} & = & \mathit{model} \times \mathit{nat} \rightarrow \mathit{model} \\
\mathit{trt} & = & \mathsf{sat}\ \mathit{model} \\
 & | & \mathsf{unsat}\ \mathit{proof} \\
 & | & \mathsf{unknown}\ \mathit{goal\ sequence} \times \mathit{modelconv} \times \mathit{proofconv} \\
 & | & \mathsf{fail} \\
\mathit{tactic} & = & \mathit{goal} \rightarrow \mathit{trt}
\end{array}
$$

end-game tactics:
never return unknown(sb, mc, pc)

SMT Tactic

$$goal = formula\ sequence \times attribute\ sequence$$

$$\begin{aligned} proofconv &= proof\ sequence \rightarrow proof \\ modelconv &= model \times nat \rightarrow model \\ trt &= \mathsf{sat}\ model \\ &\mid\ \mathsf{unsat}\ proof \\ &\mid\ \mathsf{unknown}\ goal\ sequence \times modelconv \times proofconv \\ &\mid\ \mathsf{fail} \\ tactic &= goal \rightarrow trt \end{aligned}$$

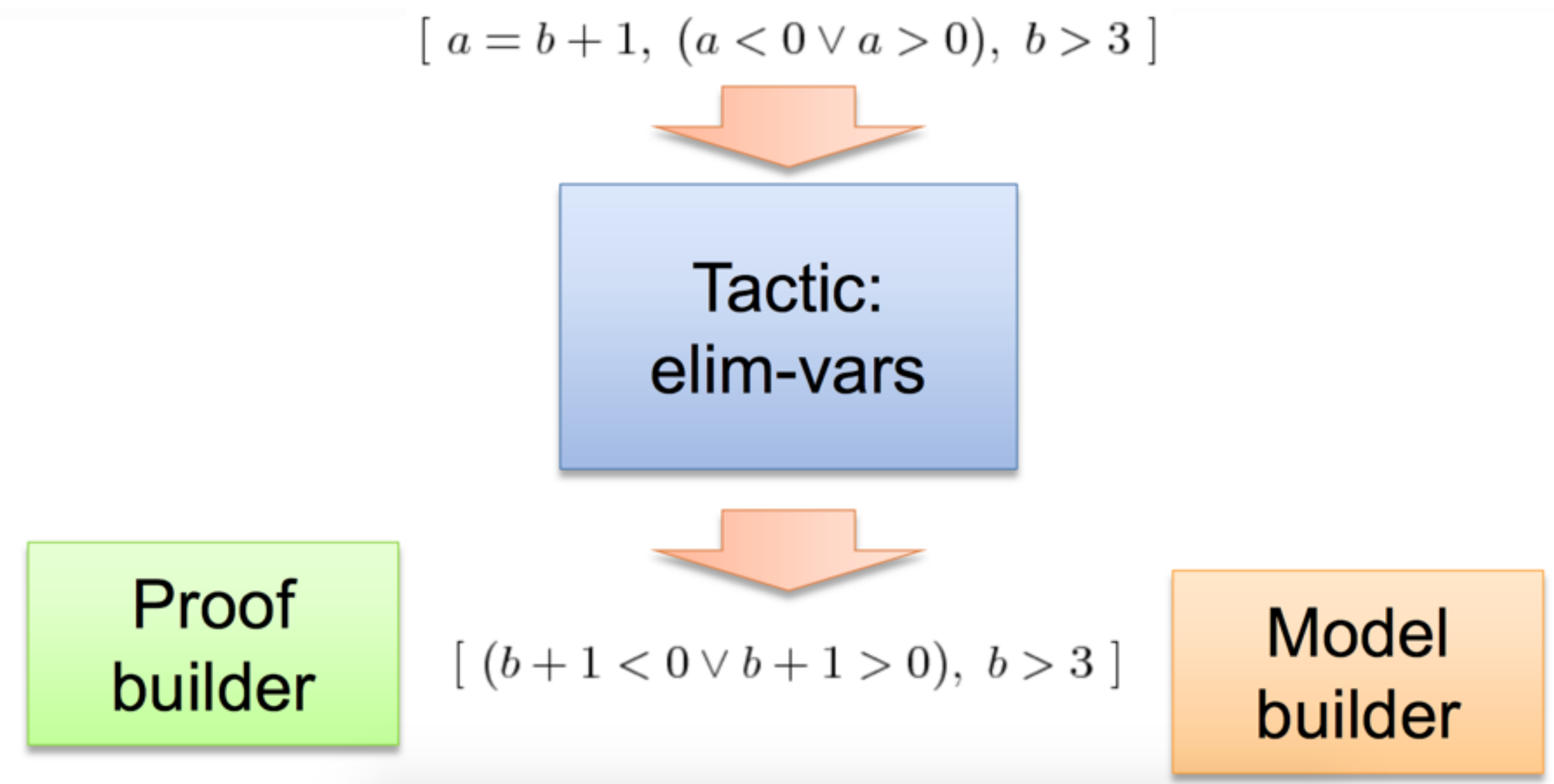
non-branching tactics:
sb is a sigleton in
unknown(sb, mc, pc)

Trivial goals

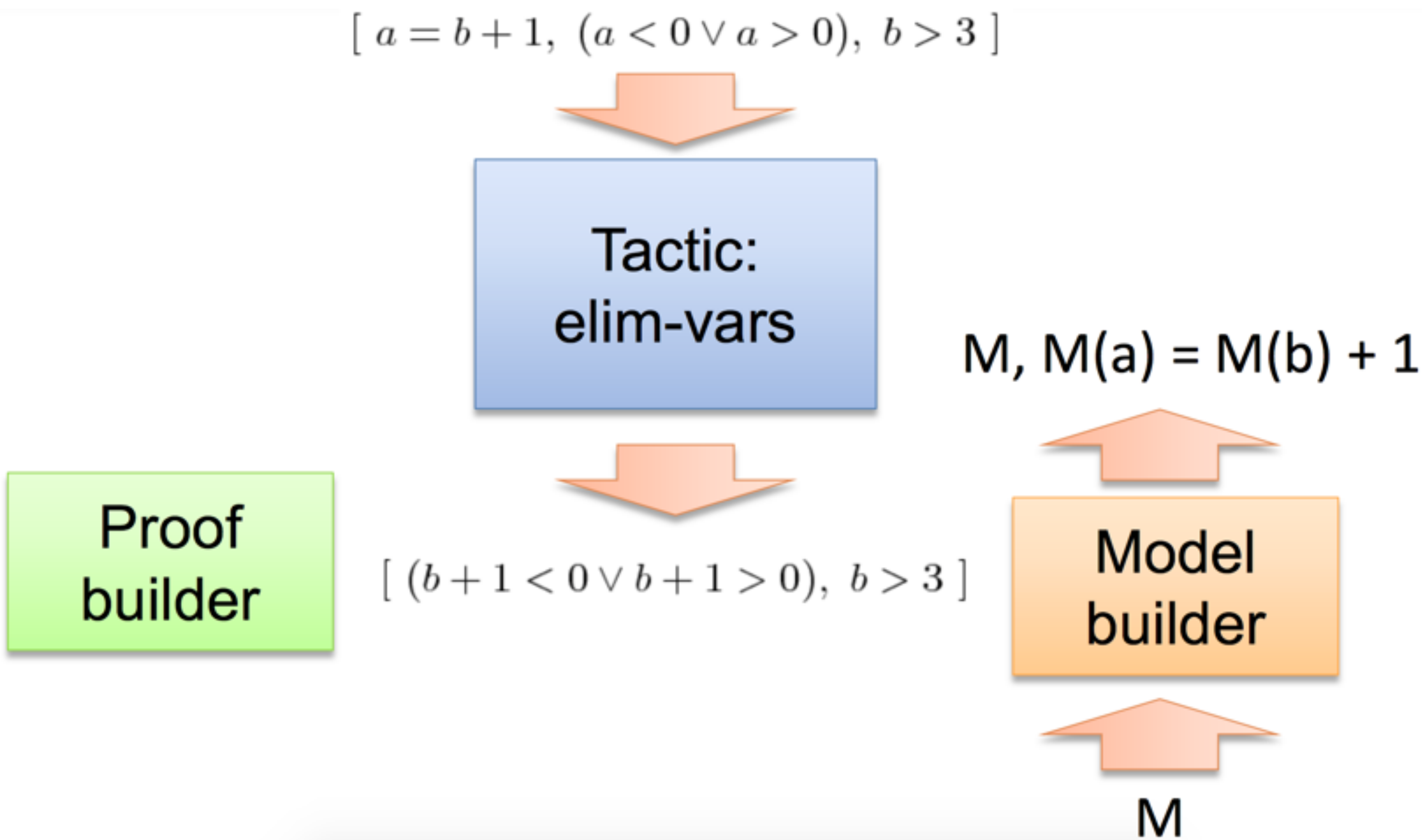
Empty goal [ ] is trivially satisfiable

False goal [..., false, ...] is trivially unsatisfiable

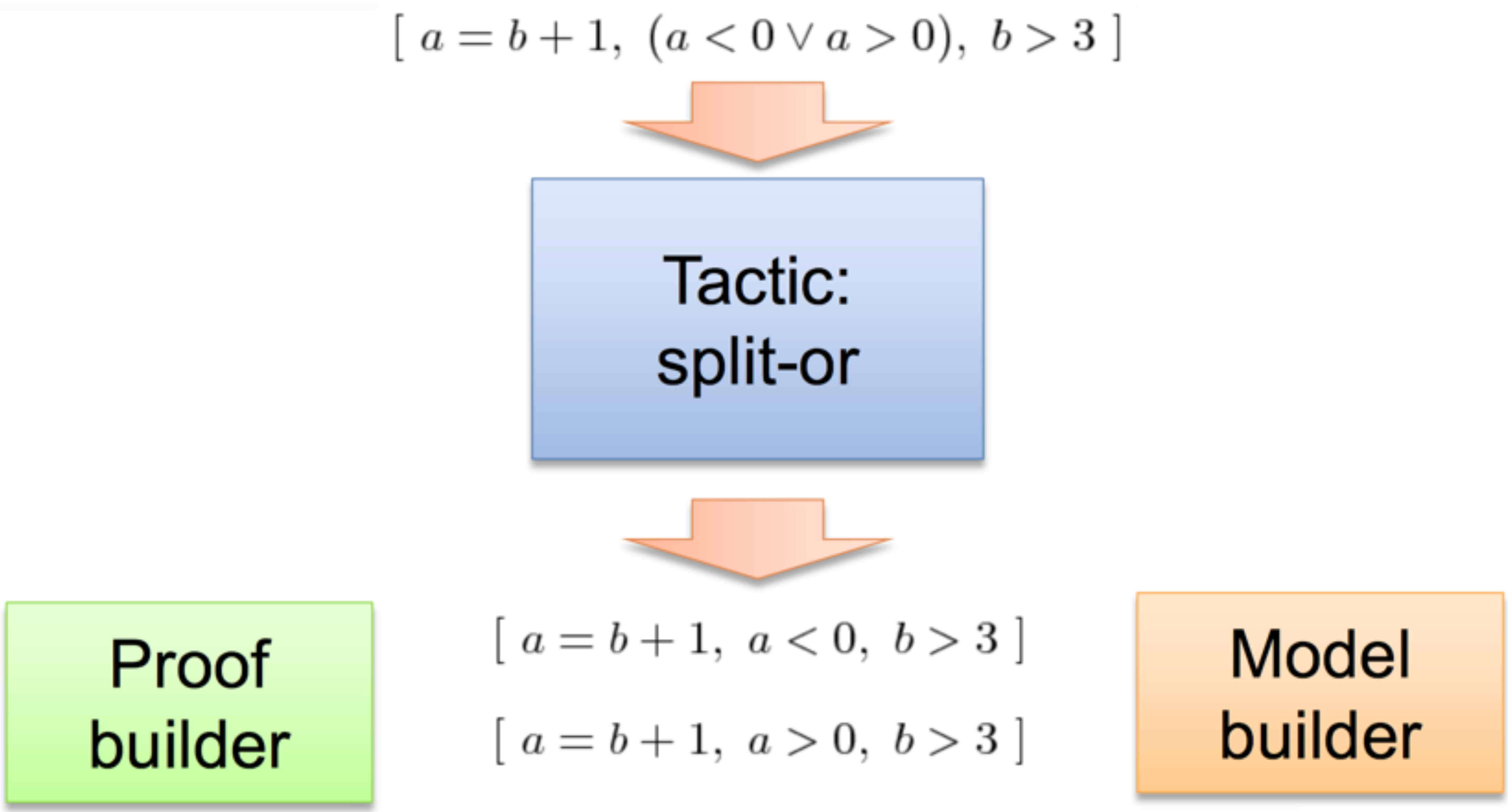
SMT Tactic: example



SMT Tactic: example



SMT Tactic: example



SMT Tactic

simplify	propagate-bounds
nnf	propagate-values
cnf	split-ineqs
tseitin	split-eqs
lift-if	rewrite
bitblast	p-cad
gb	sat
vts	solve-eqs

SMT Tacticals

then : $(tactic \times tactic) \rightarrow tactic$

then(t_1, t_2) applies t_1 to the given goal and t_2 to every subgoal produced by t_1.

then$*$: $(tactic \times tactic\ sequence) \rightarrow tactic$

then$*(t_1, [t_{2_1}, ..., t_{2_n}])$ applies t_1 to the given goal, producing subgoals $g_1, ..., g_m$. If $n \neq m$, the tactic fails. Otherwise, it applies t_{2_i} to every goal g_i.

orelse : $(tactic \times tactic) \rightarrow tactic$

orelse(t_1, t_2) first applies t_1 to the given goal, if it fails then returns the result of t_2 applied to the given goal.

par : $(tactic \times tactic) \rightarrow tactic$

par(t_1, t_2) excutes t_1 and t_2 in parallel.

SMT Tacticals

$$\mathsf{then}(\mathsf{skip}, t) = \mathsf{then}(t, \mathsf{skip}) = t$$

$$\mathsf{orelse}(\mathsf{fail}, t) = \mathsf{orelse}(t, \mathsf{fail}) = t$$

SMT Tacticals

repeat : $tactic \to tactic$

Keep applying the given tactic until no subgoal is modified by it.

repeatupto : $tactic \times nat \to tactic$

Keep applying the given tactic until no subgoal is modified by it, or the maximum number of iterations is reached.

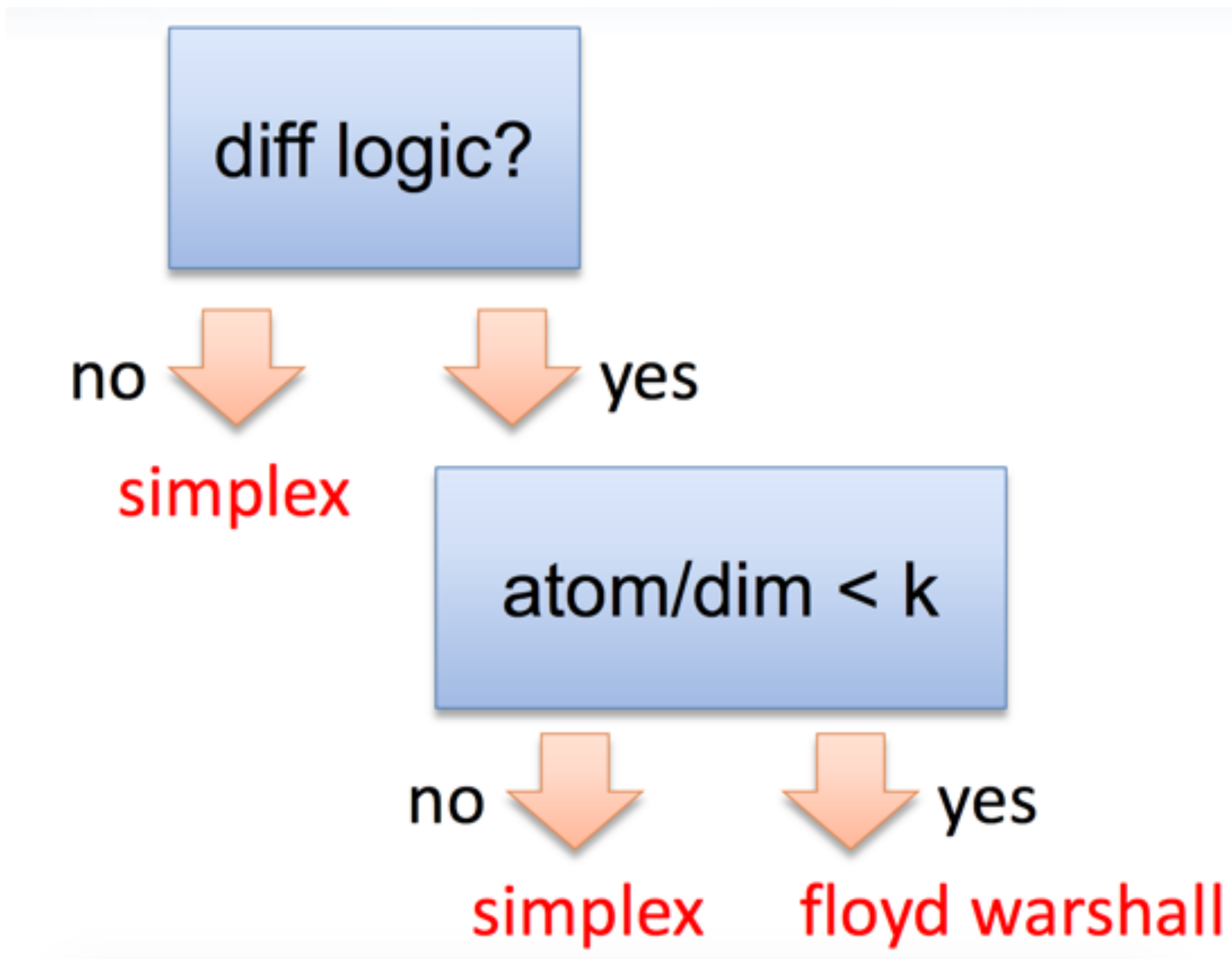
tryfor : $tactic \times seconds \to tactic$

tryfor(t, k) returns the value computed by tactic t applied to the given goal if this value is computed within k seconds, otherwise it fails.

Features/Measures

Probing structural features of formulas

Features/Measures: Yices 1.0 strategy



Features/Measures: Yices 1.0 strategy

$$\mathsf{orelse}\Big(\mathsf{then}\Big(\mathsf{failif}\Big(\mathsf{diff} \wedge \frac{\mathsf{atom}}{\mathsf{dim}} > \mathsf{k}\Big),\ \mathsf{simplex}\Big),\ \mathsf{floydwarshall}\Big)$$

Fail if condition is not satisfied.
Otherwise, do nothing.

Features/Measures

bw: Sum total bit-width of all rational coefficients of polynomials in case.
diff: True if the formula is in the difference logic fragment.
linear: True if all polynomials are linear.
dim: Number of arithmetic constants.
atoms: Number of atoms.
degree: Maximal total multivariate degree of polynomials.
size: Total formula size.

Tacticals: syntax sugar

$$\mathsf{if}(c,\ t_1,\ t_2) = \mathsf{orelse}(\mathsf{then}(\mathsf{failif}(\neg c), t_1), t_2)$$

$$\mathsf{when}(c,\ t) = \mathsf{if}(c,\ t,\ \mathsf{skip})$$

Abstraction/Refinement

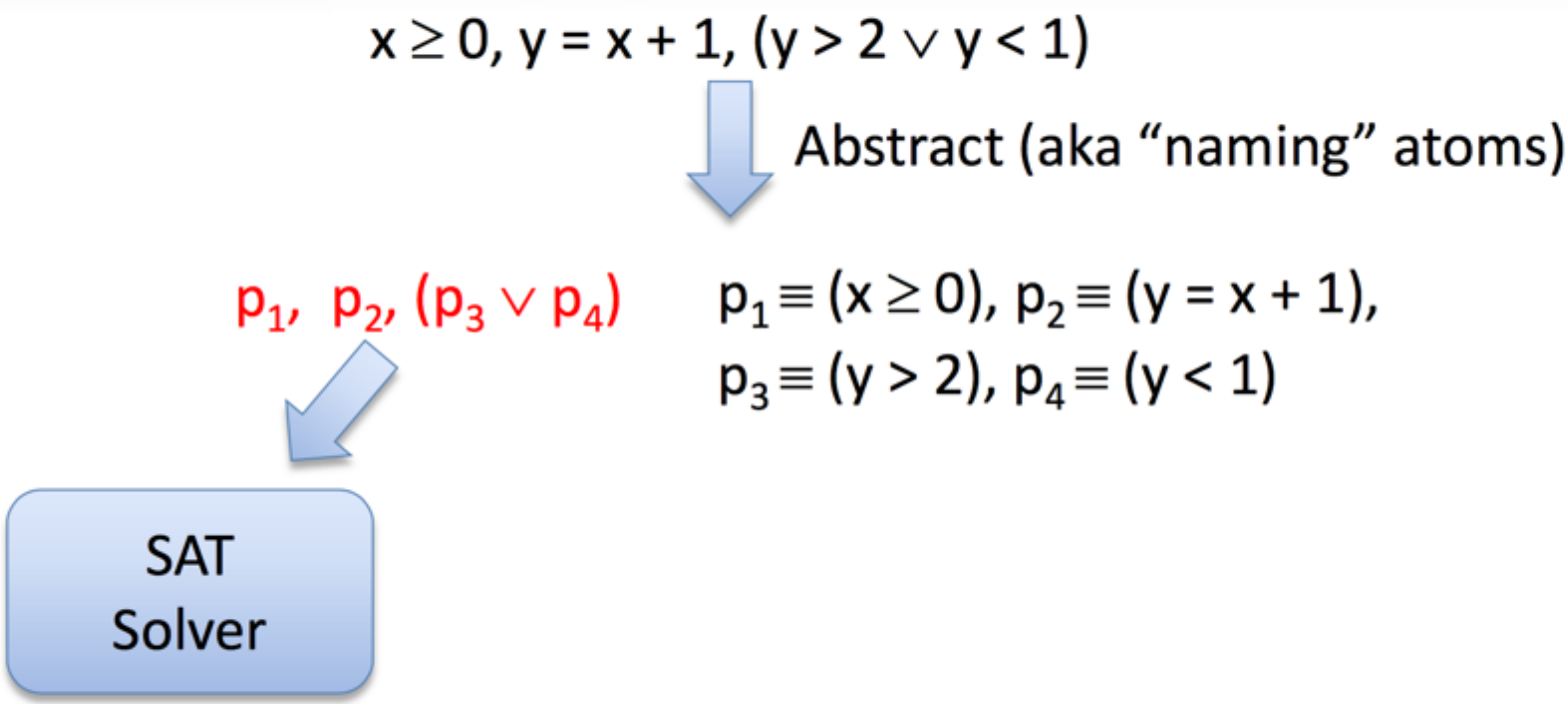
$x \geq 0, y = x + 1, (y > 2 \vee y < 1)$

Abstract (aka “naming” atoms)

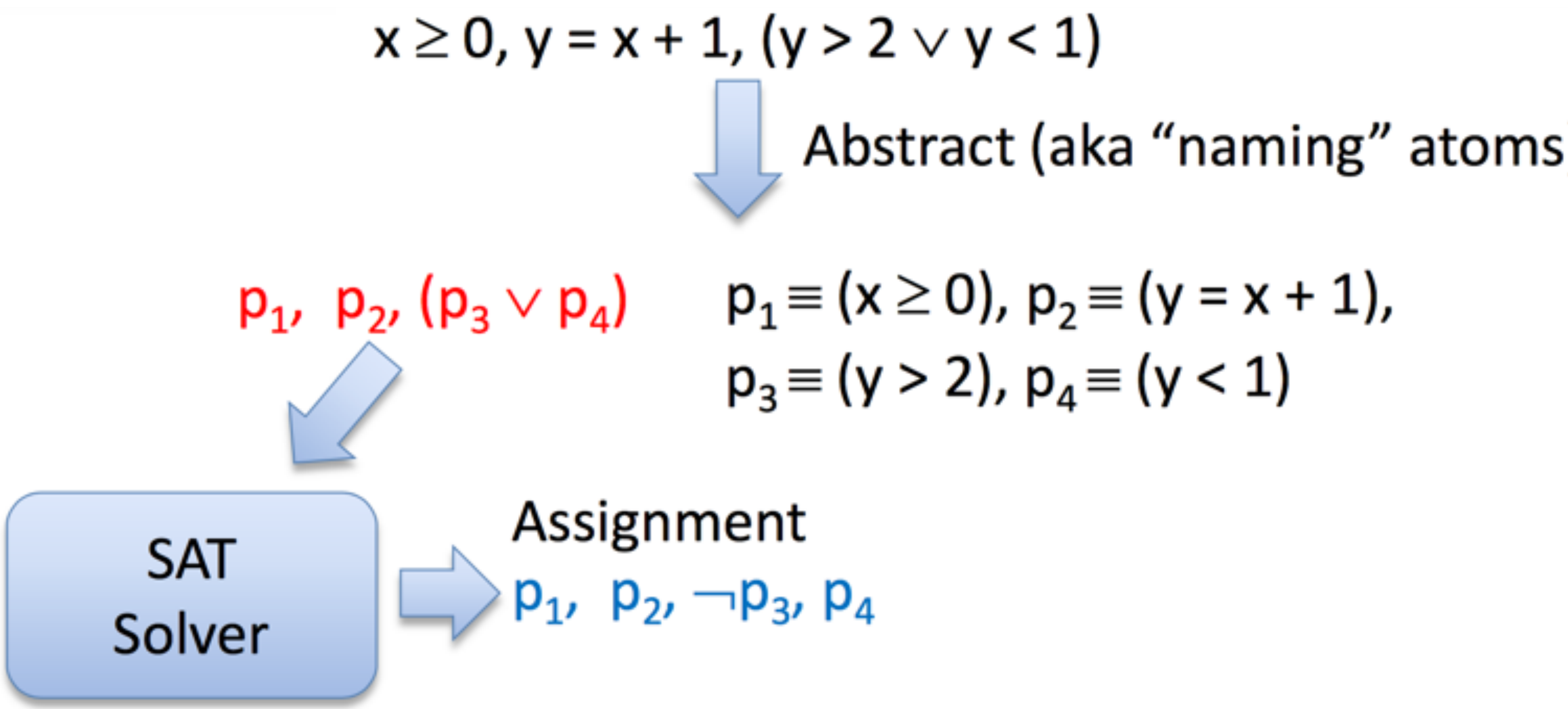
$p_1,\ \ p_2, (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

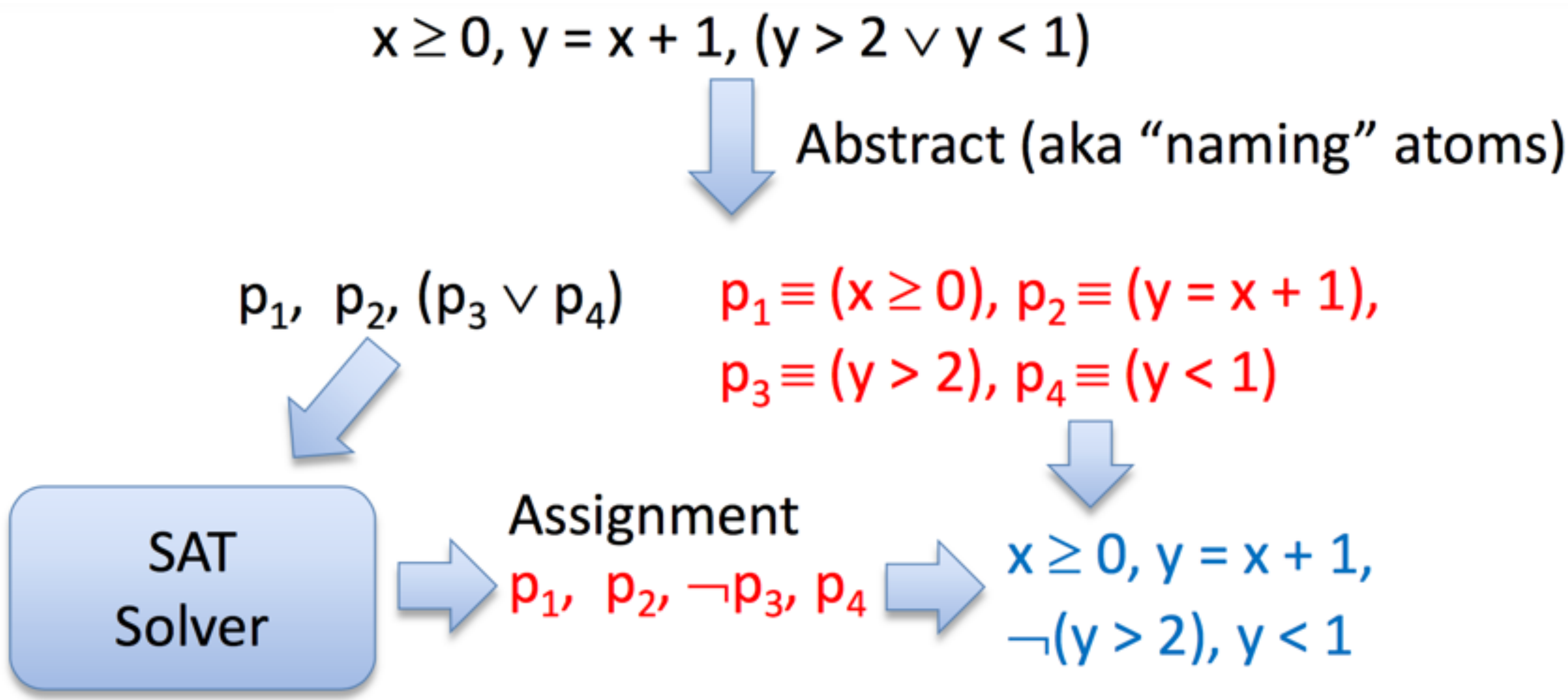
Abstraction/Refinement



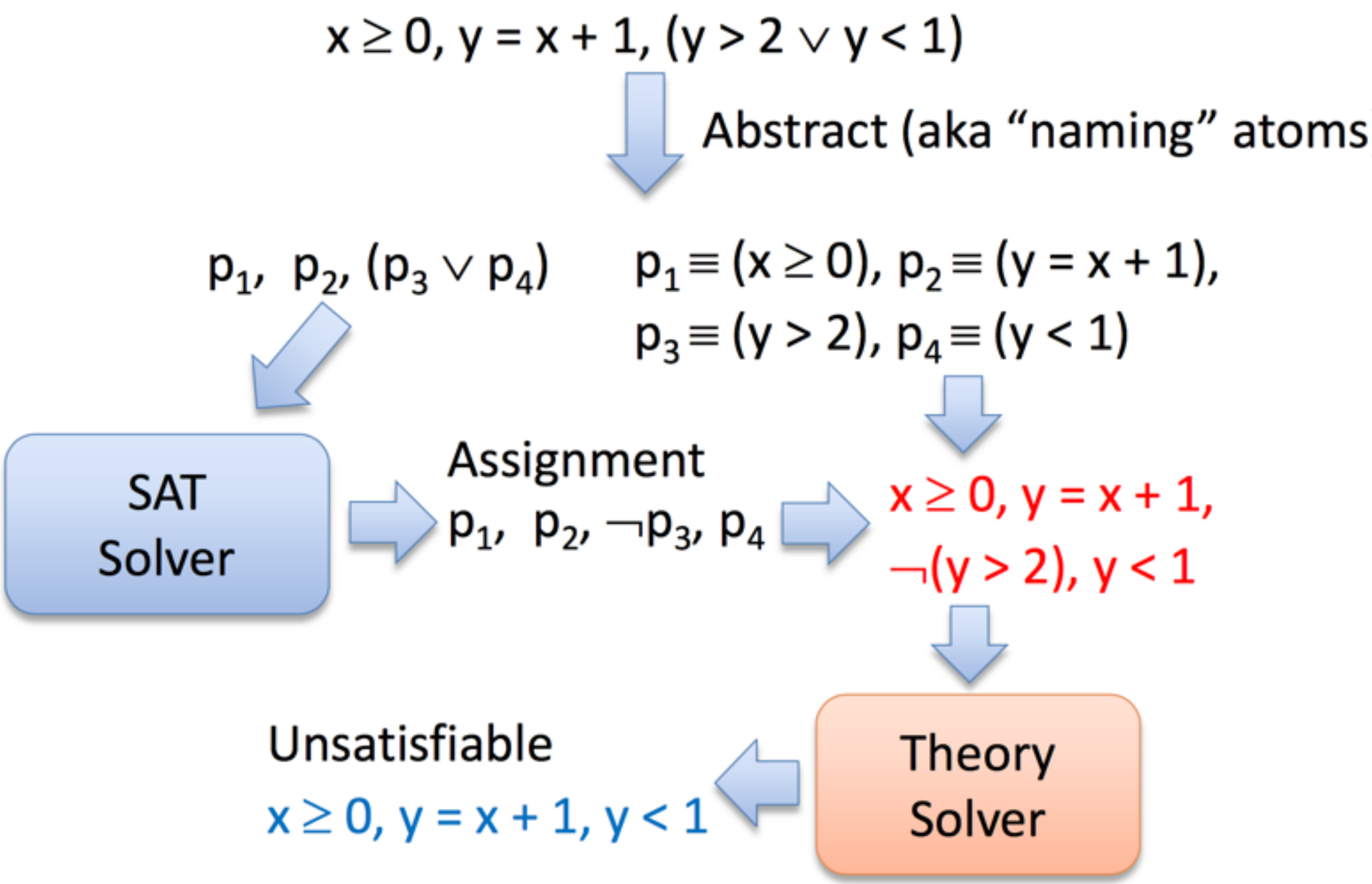
Abstraction/Refinement



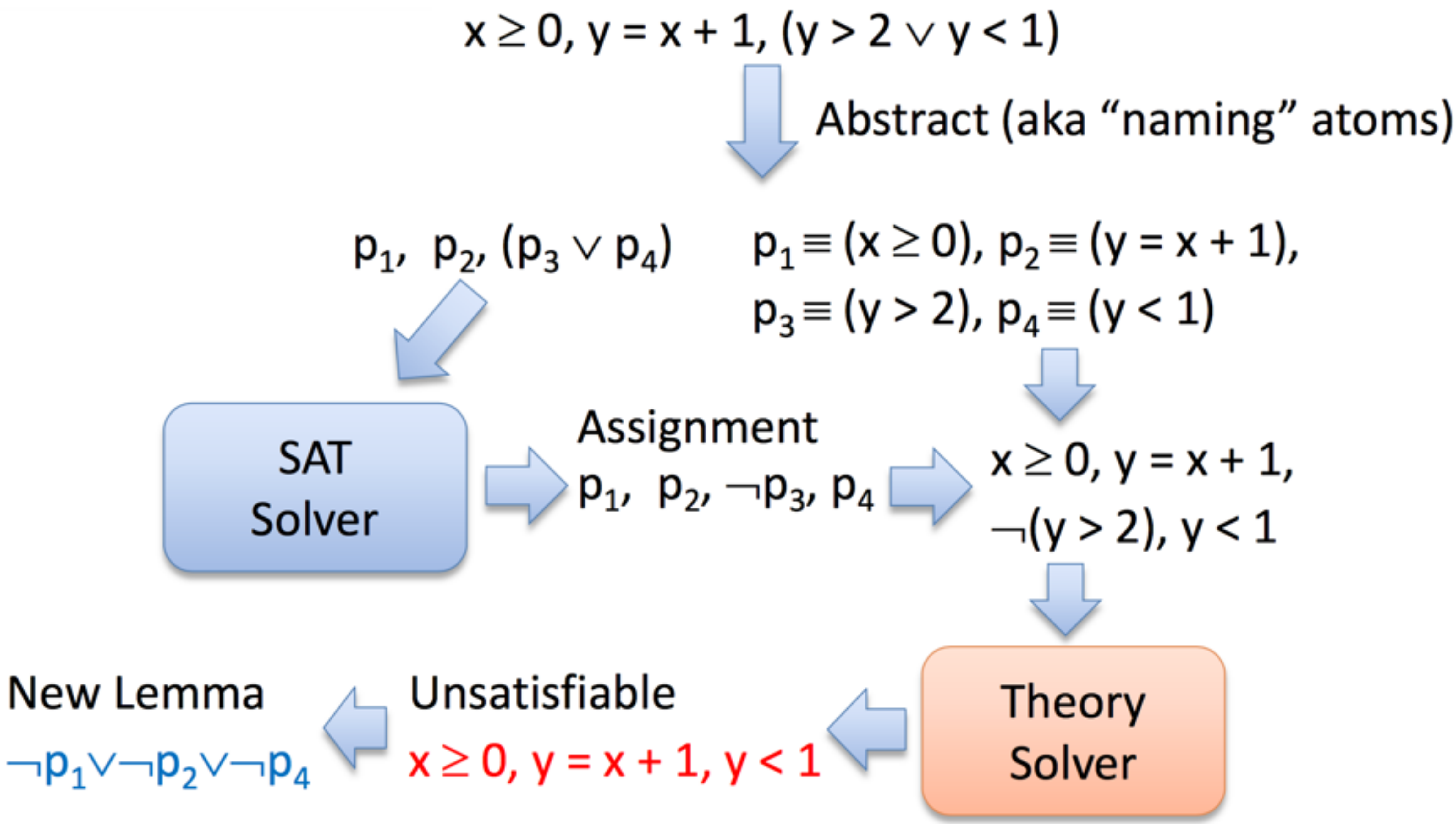
Abstraction/Refinement



Abstraction/Refinement



Abstraction/Refinement



Design engines as tacticals

then(preprocess, smt(finalcheck))

Apply "cheap" propagation/pruning steps;
and then apply complete "expensive" procedure

Conclusions

- Flexible solver/prover architectures
- “Good encodings” are solver/prover dependent
- Transparency (open source) is essential
- Separation of concerns: problem encoders x solvers
- “Orchestrating smaller/simpler procedures”