Experimenting with superposition in iProver

André Duarte and Konstantin Korovin
{andre.duarte,konstantin.korovin}@manchester.ac.uk

3 September 2019

The University of Manchester

iProver

- iProver is an automated theorem prover for first-order logic.

iProver

- iProver is an automated theorem prover for first-order logic.
- Instantiation calculus, modular architecture.

iProver

- iProver is an automated theorem prover for first-order logic.
- Instantiation calculus, modular architecture.
- Refutationally complete.

iProver

- iProver is an automated theorem prover for first-order logic.
- Instantiation calculus, modular architecture.
- Refutationally complete.
- Powerful redundancy checks.

iProver

- iProver is an automated theorem prover for first-order logic.
- Instantiation calculus, modular architecture.
- Refutationally complete.
- Powerful redundancy checks.
- Decides fragments such as EPR (winner of CASC/EPR almost every year since 2008).

iProver

- iProver is an automated theorem prover for first-order logic.
- Instantiation calculus, modular architecture.
- Refutationally complete.
- Powerful redundancy checks.
- Decides fragments such as EPR (winner of CASC/EPR almost every year since 2008).

Combination provers

Rules of thumb:

Combination provers

Rules of thumb:

- There's no one calculus that is clearly superior,
  - Some problems are solved easily by some techniques and not at all by others.

Combination provers

Rules of thumb:

- There's no one calculus that is clearly superior,
  - Some problems are solved easily by some techniques and not at all by others.
- Performance degrades very fast.
  - (Half of problems that are solved in $< 5\,\mathrm{min}$ are solved in $\sim 1\,\mathrm{s}$)
  - ($\sim 90\%$ of problems that are solved in $< 5\,\mathrm{min}$ are solved in $< 30\,\mathrm{s}$)

Combination provers

Rules of thumb:

- There's no one calculus that is clearly superior,
  - Some problems are solved easily by some techniques and not at all by others.
- Performance degrades very fast.
  - (Half of problems that are solved in $< 5\,\mathrm{min}$ are solved in $\sim 1\,\mathrm{s}$)
  - ($\sim 90\%$ of problems that are solved in $< 5\,\mathrm{min}$ are solved in $< 30\,\mathrm{s}$)
- (In iProver) clauses are shared for simplifications.

Combination provers

Rules of thumb:

- There's no one calculus that is clearly superior,
  - Some problems are solved easily by some techniques and not at all by others.
- Performance degrades very fast.
  - (Half of problems that are solved in $< 5\,\mathrm{min}$ are solved in $\sim 1\,\mathrm{s}$)
  - ($\sim 90\%$ of problems that are solved in $< 5\,\mathrm{min}$ are solved in $< 30\,\mathrm{s}$)
- (In iProver) clauses are shared for simplifications.

Corollary:

It's better to run many strategies for a little time than one strategy for a long time.

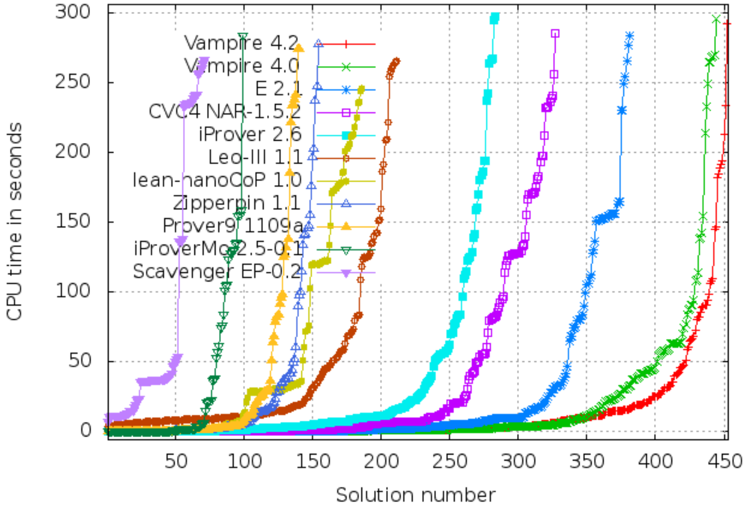


Figure: Performance graph for provers entered CASC-26/FOF.

Superposition

$$\text{Superposition} \qquad \frac{l = r \vee C \quad t[s] \doteq u \vee D}{(t[s \mapsto r] \doteq u \vee C \vee D)\theta}$$

where $\theta = \mathrm{mgu}(l, s)$, $l\theta \not\preceq r\theta$, $t\theta \not\preceq u\theta$, and s not a variable,

$$\text{Eq. Resolution} \qquad \frac{l \neq r \vee C}{C\theta}$$

where $\theta = \mathrm{mgu}(l, r)$,

$$\text{Eq. Factoring} \qquad \frac{l = r \vee l' = r' \vee C}{(l = r \vee r \neq r' \vee C)\theta}$$

where $\theta = \mathrm{mgu}(l, l')$, $l\theta \not\preceq r\theta$ and $r\theta \not\preceq r'\theta$.

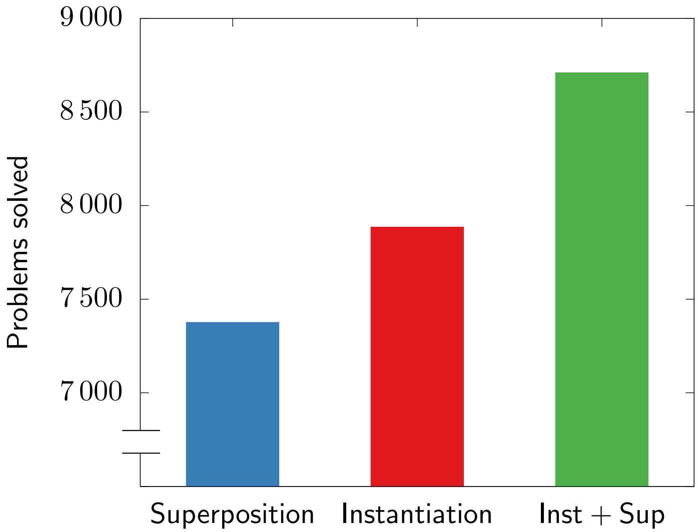


Figure: Number of problems solved over TPTP-v7.2.0, in less than 300 s.

Simplifications

As the number of kept clauses increases, the performance of the prover degrades.

Simplifications

As the number of kept clauses increases, the performance of the prover degrades.

We can control this by:

- Restricting generating inferences as much as possible.

Simplifications

As the number of kept clauses increases, the performance of the prover degrades.

We can control this by:

- Restricting generating inferences as much as possible.
  - Examples: literal selection, dismatching constraints.

Simplifications

As the number of kept clauses increases, the performance of the prover degrades.

We can control this by:

- Restricting generating inferences as much as possible.
  - Examples: literal selection, dismatching constraints.
- Performing simplifying inferences to delete clauses.

Simplifications

As the number of kept clauses increases, the performance of the prover degrades.

We can control this by:

- Restricting generating inferences as much as possible.
  - Examples: literal selection, dismatching constraints.
- Performing simplifying inferences to delete clauses.
  - Examples: subsumption, tautology deletion, rewriting by unit equalities.

Simplifications

As the number of kept clauses increases, the performance of the prover degrades.

We can control this by:

- Restricting generating inferences as much as possible.
  - Examples: literal selection, dismatching constraints.
- Performing simplifying inferences to delete clauses.
  - Examples: subsumption, tautology deletion, rewriting by unit equalities.

Simplifying inferences are key!

Simplifications

As the number of kept clauses increases, the performance of the prover degrades.

We can control this by:

- Restricting generating inferences as much as possible.
  - Examples: literal selection, dismatching constraints.
- Performing simplifying inferences to delete clauses.
  - Examples: subsumption, tautology deletion, rewriting by unit equalities.

> Simplifying inferences are key. . .
> but we can't spend too much time on them!

Simplifications

Tautology deletion

$$\frac{\cancel{l \vee \bar{l} \vee C}}{} \qquad \frac{\cancel{t = t \vee C}}{}$$

Syntactic eq. res.

$$\frac{\cancel{t \neq t \vee C}}{C}$$

Subsumption

$$\frac{\cancel{C\theta \vee D} \quad C}{}$$

Subset subsumption

$$\frac{\cancel{C \vee D} \quad C}{}$$

Demodulation

$$\frac{l = r \quad \cancel{C[l\theta]}}{C[l\theta \mapsto r\theta]}, \quad \begin{array}{l} l\theta \succ r\theta \\ \{l\theta = r\theta\} \prec C \end{array}$$

Simplifications

Tautology deletion	$\dfrac{\cancel{l \vee \bar{l} \vee C}}{}$ $\quad$ $\dfrac{\cancel{t = t \vee C}}{}$
Syntactic eq. res.	$\dfrac{\cancel{t \neq t \vee C}}{C}$
Subsumption	$\dfrac{\cancel{C\theta \vee D} \quad C}{}$
Subset subsumption	$\dfrac{\cancel{C \vee D} \quad C}{}$
Demodulation	$\dfrac{l = r \quad \cancel{C[l\theta]}}{C[l\theta \mapsto r\theta]}, \quad \begin{array}{l} l\theta \succ r\theta \\ \{l\theta = r\theta\} \prec C \end{array}$

Simplifications

Tautology deletion
$$\frac{\cancel{l \vee \bar{l} \vee C}}{} \qquad \frac{\cancel{t = t \vee C}}{}$$

Syntactic eq. res.
$$\frac{\cancel{t \neq t \vee C}}{C}$$

Subsumption
$$\frac{\cancel{C\theta \vee D} \quad C}{}$$

Subset subsumption
$$\frac{\cancel{C \vee D} \quad C}{}$$

Demodulation
$$\frac{l = r \quad \cancel{C[l\theta]}}{C[l\theta \mapsto r\theta]}, \quad \begin{array}{l} l\theta \succ r\theta \\ \{l\theta = r\theta\} \prec C \end{array}$$

Light normalisation
$$\frac{l = r \quad \cancel{C[l]}}{C[l \mapsto r]}, \quad l \succ r$$

Light normalisation

We introduce the simplification rule

$$\text{Light normalisation} \qquad \frac{l = r \quad \cancel{C[l]}}{C[l \mapsto r]}$$

where $l \succ r$, and l occurs outside a maximal side of an equality literal.

While a restricted case of demodulation, it's also much faster.

Light normalisation

We introduce the simplification rule

$$\text{Light normalisation} \qquad \frac{l = r \quad \cancel{C[l]}}{C[l \mapsto r]}$$

where $l \succ r$, and l occurs outside a maximal side of an equality literal.

While a restricted case of demodulation, it's also much faster:

- No indexing.

Light normalisation

We introduce the simplification rule

$$\text{Light normalisation} \qquad \frac{l = r \quad \cancel{C[l]}}{C[l \mapsto r]}$$

where $l \succ r$, and l occurs outside a maximal side of an equality literal.

While a restricted case of demodulation, it's also much faster:

- No indexing.
- No instantiation of unit equalities.

Light normalisation

We introduce the simplification rule

$$\text{Light normalisation} \qquad \frac{l = r \quad \cancel{C[l]}}{C[l \mapsto r]}$$

where $l \succ r$, and l occurs outside a maximal side of an equality literal.

While a restricted case of demodulation, it's also much faster:

- No indexing.
- No instantiation of unit equalities.
- No ordering checks.

Light normalisation

We introduce the simplification rule

$$\text{Light normalisation} \qquad \frac{l = r \quad \cancel{C[l]}}{C[l \mapsto r]}$$

where $l \succ r$, and l occurs outside a maximal side of an equality literal.

While a restricted case of demodulation, it's also much faster:

- No indexing.
- No instantiation of unit equalities.
- No ordering checks.
- Long demodulation chains are done in 1 step.

Simplifications

Important: lots of freedom to choose *how* we do simplifications.

Simplifications

Important: lots of freedom to choose *how* we do simplifications:

- which rules to perform,

Simplifications

Important: lots of freedom to choose *how* we do simplifications:

- which rules to perform,
- in what order,

Simplifications

Important: lots of freedom to choose *how* we do simplifications:

- which rules to perform,
- in what order,
- when,

Simplifications

Important: lots of freedom to choose *how* we do simplifications:

- which rules to perform,
- in what order,
- when,
- and with respect to what clauses.

Simplifications

Important: lots of freedom to choose *how* we do simplifications:

- which rules to perform,
- in what order,
- when,
- and with respect to what clauses.

Also, these require indices to implement. Some indices support several simplification rules. We must choose:

Simplifications

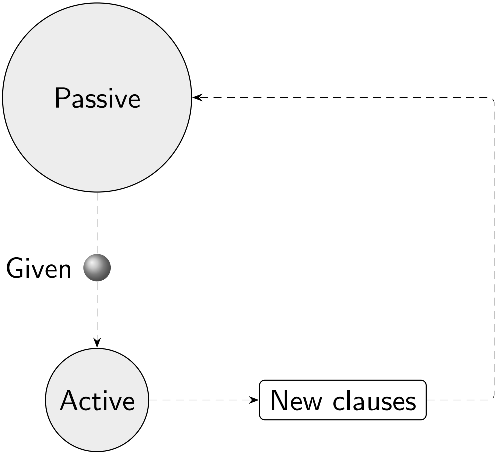
Important: lots of freedom to choose *how* we do simplifications:

- which rules to perform,
- in what order,
- when,
- and with respect to what clauses.

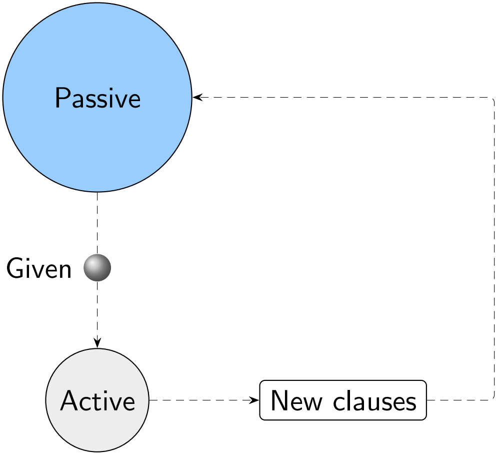
Also, these require indices to implement. Some indices support several simplification rules. We must choose:

- which clauses to add to which indices,
- and when.

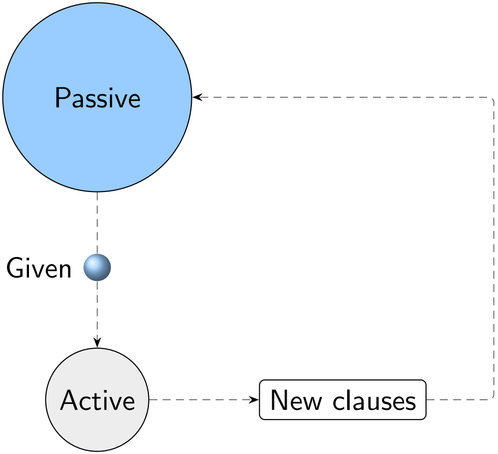
Simplification setup — Given clause loop overview



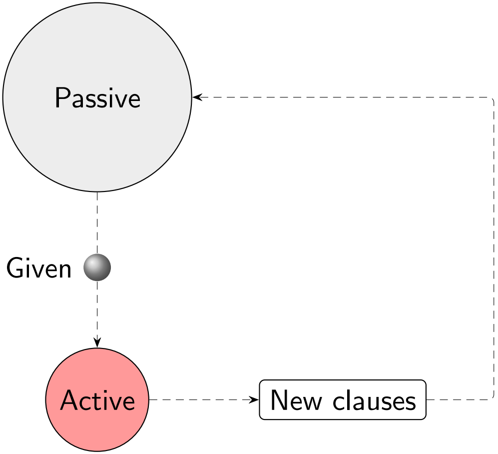
Simplification setup — Given clause loop overview



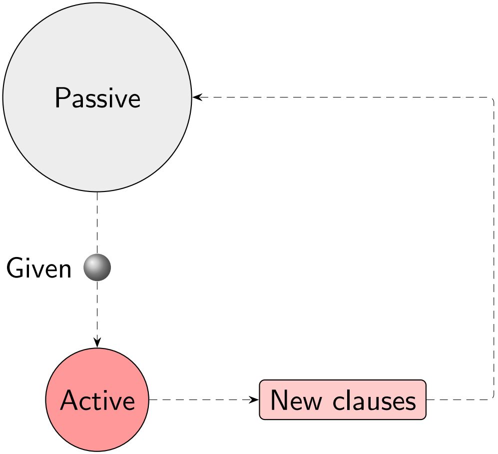
Simplification setup — Given clause loop overview



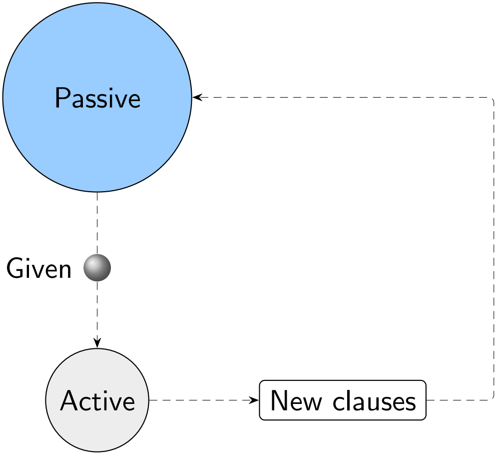
Simplification setup — Given clause loop overview



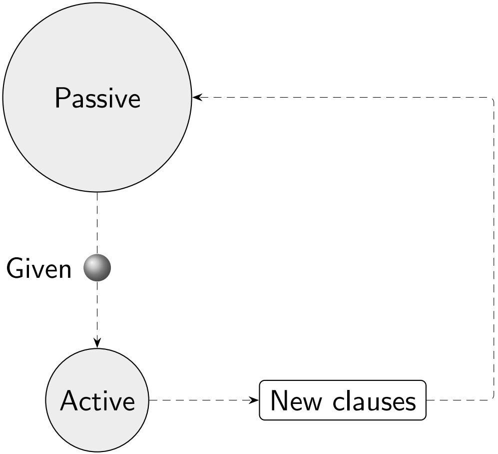
Simplification setup — Given clause loop overview



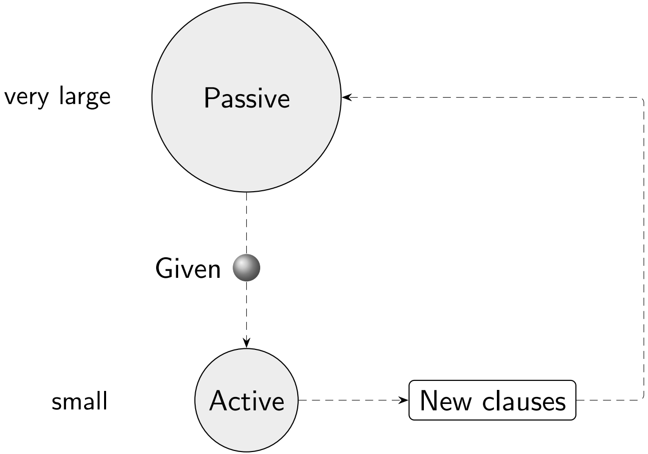
Simplification setup — Given clause loop overview



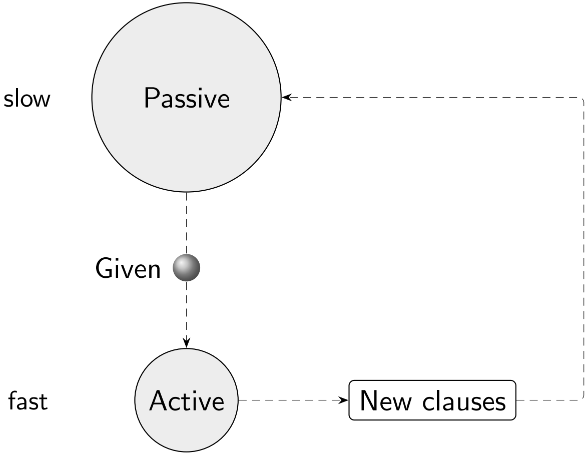
Simplification setup — Given clause loop overview



Simplification setup — Given clause loop overview

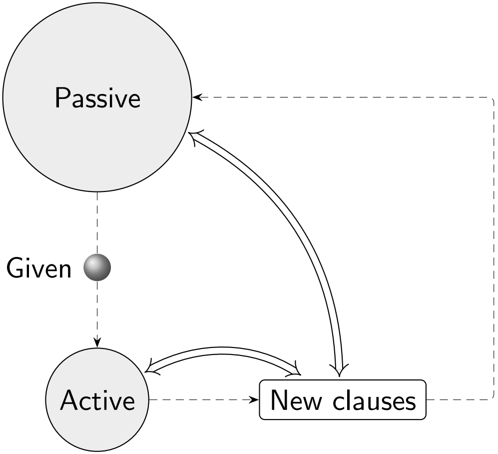


Simplification setup — Given clause loop overview



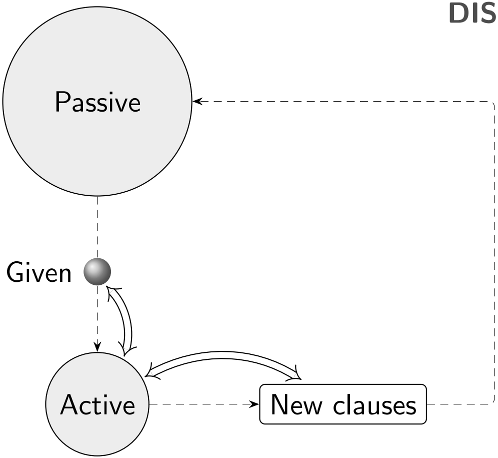
Simplification setup

Otter



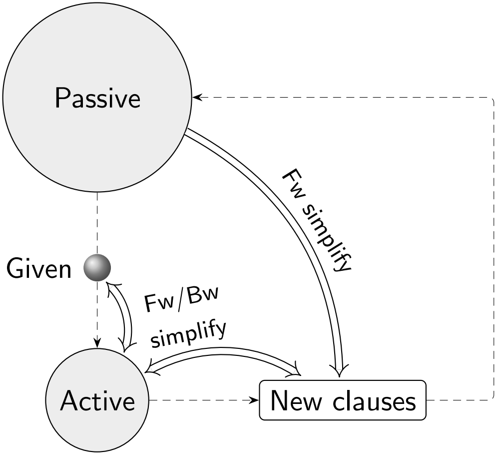
Simplification setup

DISCOUNT

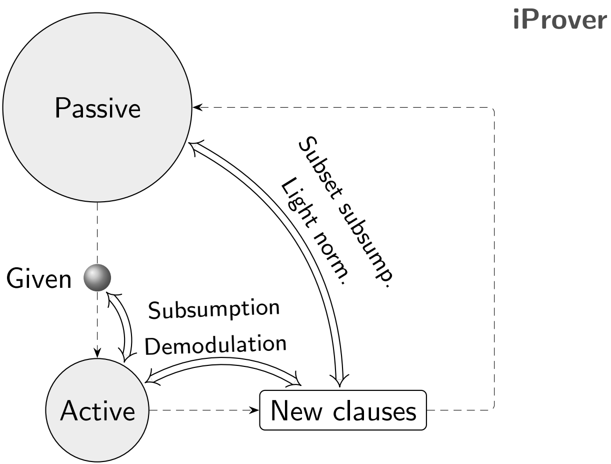


Simplification setup

iProver



Simplification setup



Immediate simplification

Intuition:

Immediate simplification

Intuition:

- Clauses that are derived in each loop are more “related” to each other.

Immediate simplification

Intuition:

- Clauses that are derived in each loop are more “related” to each other.
- The passive set grows very large, but the set of new clauses in each loop stays comparatively small.

Immediate simplification

Intuition:

- Clauses that are derived in each loop are more “related” to each other.
- The passive set grows very large, but the set of new clauses in each loop stays comparatively small.
- Can check if a new clause deletes a parent clause. If yes, then:
  - we can throw away all its children,
  - and avoid trying to generate any new clauses with it.

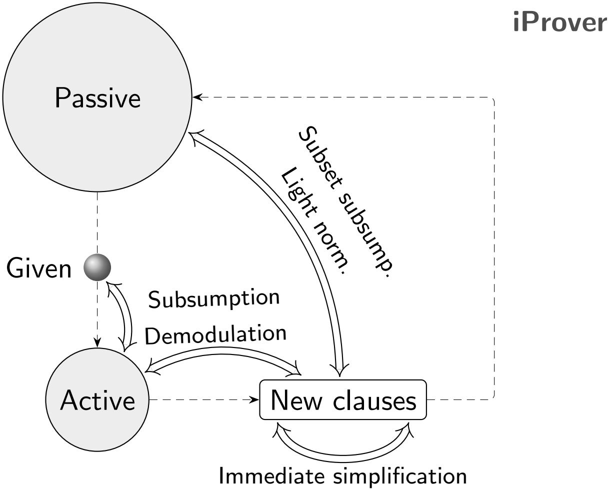
Immediate simplification

Intuition:

- Clauses that are derived in each loop are more "related" to each other.
- The passive set grows very large, but the set of new clauses in each loop stays comparatively small.
- Can check if a new clause deletes a parent clause. If yes, then:
  - we can throw away all its children,
  - and avoid trying to generate any new clauses with it.

Hypothesis: it may be useful to keep new clause $\cup$ parents inter-simplified.

Simplification setup



Simplification setup — iProver

```
$ ./iprover --schedule none | grep '--sup_'
--sup_indices_passive     [SubsetSubsumption]
--sup_indices_active      [Subsumption;LightNormNoReduce;FwDemod;...
--sup_indices_immed       [SubsetSubsumption;Subsumption;LightNor...
--sup_indices_input       [SubsetSubsumption;Subsumption;LightNor...
--sup_light_triv          [TrivRules]
--sup_light_fw            [FwLightNorm]
--sup_light_bw            []
--sup_full_triv           [TrivRules;PropSubs]
--sup_full_fw             [FwDemodLightNormLoopTriv;FwSubsumption...
--sup_full_bw             [BwDemod]
--sup_immed_triv          [TrivRules]
--sup_immed_fw_main       [FwDemodLightNormLoopTriv;FwSubsumption...
--sup_immed_fw_immed      [FwDemodLightNormLoopTriv;FwSubsumption...
--sup_immed_bw_main       []
--sup_immed_bw_immed      [BwDemod;BwSubsumption;BwSubsumptionRes...
--sup_input_triv          [TrivRules]
--sup_input_fw            [FwDemodLightNormLoopTriv;FwSubsumption...
--sup_input_bw            [BwDemod;BwSubsumption;BwSubsumptionRes]
```

Summary

- It is (generally) better to combine many strategies/options than to run just one.
  - Instantiation + superposition is better than just instantiation or superposition.
- Applying simplification rules is crucial for performance. But spending too much time on them may hurt more than help.
- Huge freedom in choosing how to do them, but no clear path.
  - Work on hyperparameter optimisation may help here.
- "Immediate simplification" may block many redundant generating inferences, and is relatively inexpensive.