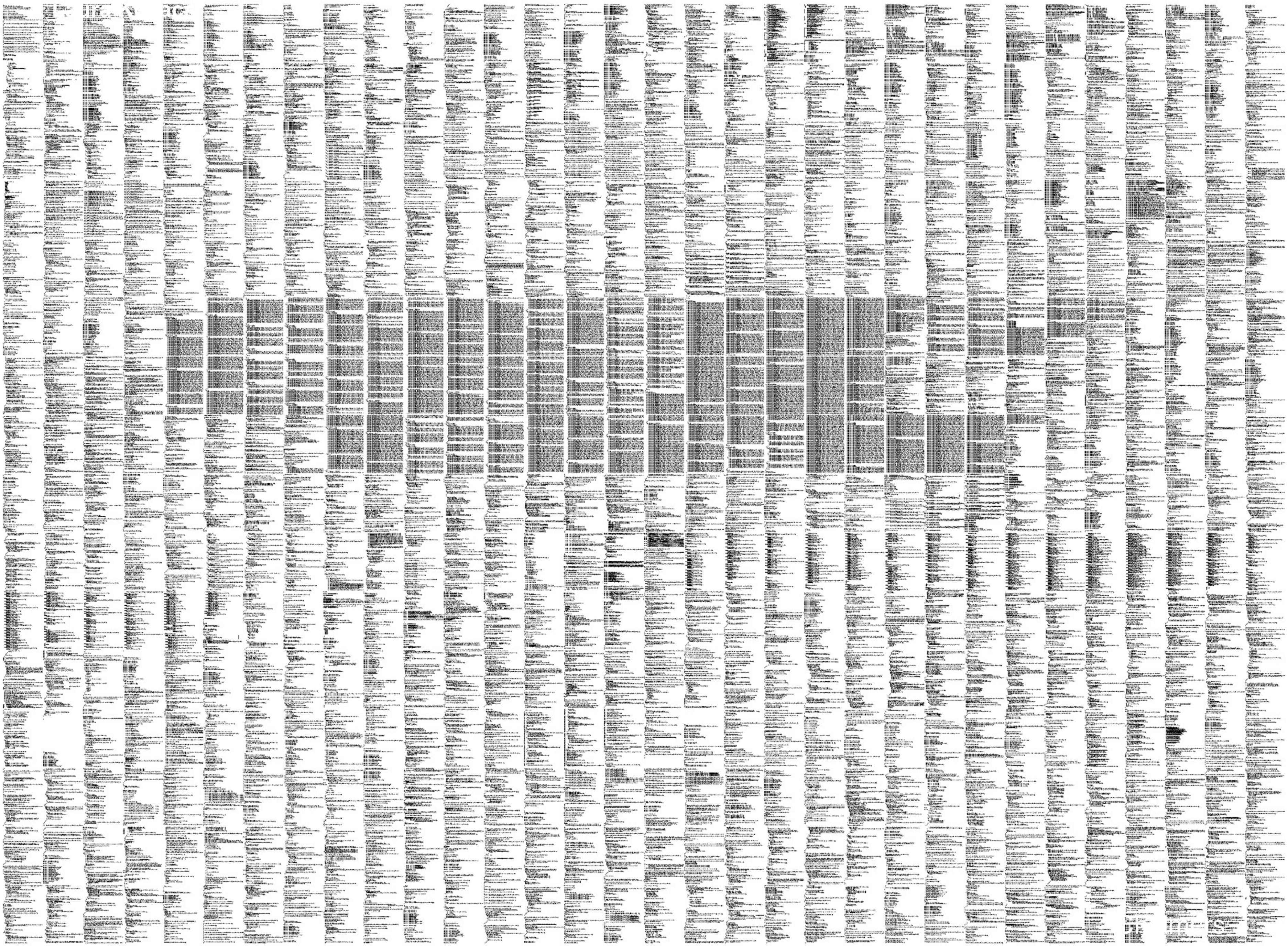
First-Order Theorem Proving and Program Analysis

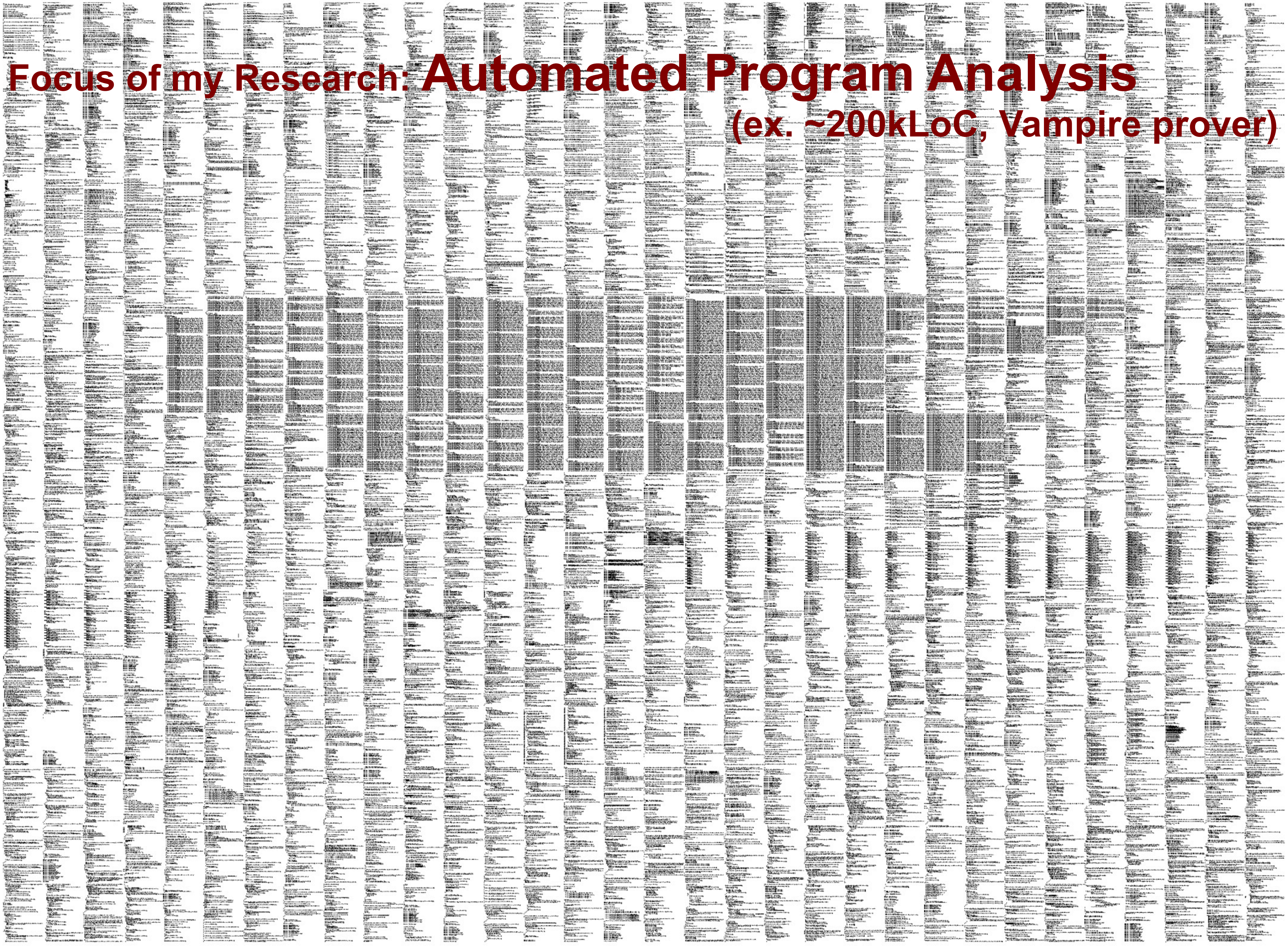
Laura Kovács

Chalmers University of Technology



Focus of my Research: Automated Program Analysis

(ex. ~200kLoC, Vampire prover)



Focus of my Research: Automated Program Analysis

```
a=0, b=0, c=0;
while (a<n) do

if A[a]>0 then B[b]=A[a]+h(b); b=b+1;
          else C[c]=A[a]; c=c+1;

a=a+1;
end do
```

Focus of my Research: Automated Program Analysis

```
a=0, b=0, c=0;
while (a<n) do

if A[a]>0 then B[b]=A[a]+h(b); b=b+1;
          else C[c]=A[a]; c=c+1;

a=a+1;
end do
```

Program property:

$$(\forall p)(0 \le p < b \Rightarrow$$
$$(\exists q)(0 \le q < a \wedge B[p]=A[q]+h(p) \wedge A[q]>0)$$

Focus of my Research: Automated Program Analysis

```
cnt=0, fib1=1, fib2=0;
while (cnt<n) do

t=fib1; fib1=fib1+fib2; fib2=t; cnt++;

end do
```

h

```
a=0, b=0, c=0;
while (a<n) do

if A[a]>0 then B[b]=A[a]+h(b); b=b+1;
               else C[c]=A[a]; c=c+1;

a=a+1;
end do
```

Focus of my Research: Automated Program Analysis

```
cnt=0, fib1=1, fib2=0;
while (cnt<n) do
  t=fib1; fib1=fib1+fib2; fib2=t; cnt++;
end do
```

h

Program property:

$$fib1^4 + fib2^4 + 2*fib1*fib2^3 - 2\ fib1^3*fib2 - fib1^2*fib2^2 - 1 = 0$$

```
a=0, b=0, c=0;
while (a<n) do
  if A[a]>0 then B[b]=A[a]+h(b); b=b+1;
         else C[c]=A[a]; c=c+1;
  a=a+1;
end do
```

Focus of my Research: Automated Program Analysis

```
cnt=0, fib1=1, fib2=0;
while (cnt<n) do
t=fib1; fib1=fib1+fib2; fib2=t; cnt++;
end do
```

$fib1^4 + fib2^4 + 2*fib1*fib2^3 - 2\ fib1^3*fib2 - fib1^2*fib2^2 - 1 = 0$

Math

```
a=0, b=0, c=0;
while (a<n) do
if A[a]>0 then B[b]=A[a]+h(b); b=b+1;
          else C[c]=A[a]; c=c+1;
a=a+1;
end do
```

h

Logic

$(\forall p)(0 \le p < b \Rightarrow$

$(\exists q)(0 \le q < a \wedge B[p]=A[q]+h(p) \wedge A[q]>0)$

Math

Logic

My Research

Program Analysis

Symbolic Computation

Automated Theorem Proving

My Research

funded by:

European Research Council
Supporting top researchers from anywhere in the world

Knut and Alice Wallenberg Foundation

WALLENBERG ACADEMY FELLOWS

Vetenskapsrådet

Program Analysis

Symbolic Computation

Automated Theorem Proving

My Research

funded by:

European Research Council
Supporting top researchers from anywhere in the world

Knut and Alice Wallenberg Foundation

Wallenberg Academy Fellows

Vetenskapsrådet

Need industrial partners/interest!

(We have the funding!)

Program Analysis

Outline

Example: Array Partition

$a := 0$; $b := 0$; $c := 0$;
while $(a \leq k)$ **do**
  if $A[a] \geq 0$
    then $B[b] := A[a]$; $b := b + 1$;
    else $C[c] := A[a]$; $c := c + 1$;
  $a := a + 1$;
end while

A :

1	3	-1	-5	8	0	-2

$a = 0$

B :

*	*	*	*	*	*	*

$b = 0$

C :

*	*	*	*	*	*	*

$c = 0$

Example: Array Partition

$a := 0$; $b := 0$; $c := 0$;
while $(a \leq k)$ **do**
  if $A[a] \geq 0$
    then $B[b] := A[a]$;$b := b + 1$;
    else $C[c] := A[a]$;$c := c + 1$;
  $a := a + 1$;
end while

A :

1	3	-1	-5	8	0	-2

$a = 7$

B :

1	3	8	0	*	*	*

$b = 4$

C :

-1	-5	-2	*	*	*	*

$c = 3$

Example: Array Partition

$a := 0$; $b := 0$; $c := 0$;
while $(a \leq k)$ **do**
  if $A[a] \geq 0$
    then $B[b] := A[a]$;$b := b + 1$;
    else $C[c] := A[a]$;$c := c + 1$;
  $a := a + 1$;
end while

A :

1	3	-1	-5	8	0	-2

$a = 7$

B :

1	3	8	0	*	*	*

$b = 4$

C :

-1	-5	-2	*	*	*	*

$c = 3$

Invariants with ∀ ∃

- Each of $B[0], \ldots, B[b-1]$ is non-negative and equal to one of $A[0], \ldots, A[a-1]$.

$$(\forall p)(0 \leq p < b \rightarrow B[p] \geq 0 \wedge (\exists i)(0 \leq i < a \wedge A[i] = B[p]))$$

Example: Array Partition

$a := 0$; $b := 0$; $c := 0$;
while $(a \leq k)$ **do**
  if $A[a] \geq 0$
    then $B[b] := A[a]$;$b := b + 1$;
    else $C[c] := A[a]$;$c := c + 1$;
  $a := a + 1$;
end while

A :

1	3	-1	-5	8	0	-2

$a = 7$

B :

1	3	8	0	*	*	*

$b = 4$

C :

-1	-5	-2	*	*	*	*

$c = 3$

Invariants with $\forall$ $\exists$

- Each of $B[0], \ldots, B[b-1]$ is non-negative and equal to one of $A[0], \ldots, A[a-1]$.

$$(\forall p)(0 \leq p < b \rightarrow B[p] \geq 0 \wedge (\exists i)(0 \leq i < a \ \wedge \ A[i] = B[p]))$$

Invariant Generation – Overview of Our Method

- Given loop $\mathcal{L}$;
- Extend $\mathcal{L}$ to $\mathcal{L}'$;
- Extract a set P of loop properties in $\mathcal{L}'$;
- Generate loop property p in $\mathcal{L}$ s.t. $P \rightarrow p$.

Invariant Generation – Overview of Our Method

- Given loop $\mathcal{L}$;
- Extend $\mathcal{L}$ to $\mathcal{L}'$;
- Extract a set P of loop properties in $\mathcal{L}'$;
- Generate loop property p in $\mathcal{L}$ s.t. $P \rightarrow p$.

Invariant Generation – Overview of Our Method

- Given loop $\mathcal{L}$;
- Extend $\mathcal{L}$ to $\mathcal{L}'$;
- Extract a set P of loop properties in $\mathcal{L}'$;
- Generate loop property p in $\mathcal{L}$ s.t. $P \rightarrow p$.

  $\leftarrow$ Symbol elimination!

Invariant Generation - The Method

```
a := 0; b := 0; c := 0;
while (a ≤ k) do
  if A[a] ≥ 0
    then B[b] := A[a];b := b + 1;
    else C[c] := A[a];c := c + 1;
  a := a + 1;
end while
```

1. Extend the language $\mathcal{L}$ to $\mathcal{L}'$:
   - variables as functions of n: $v^{(i)}$ with $0 \leq i < n$
   - predicates as loop properties: *iter*

$$(\forall i)(i \in iter \Leftrightarrow 0 \leq i \wedge i < n)$$

2. Collect loop properties:

$$a = b + c, \; a \geq 0, \; b \geq 0, \; c \geq 0$$

$$(\forall i \in iter)(a^{(i+1)} > a^{(i)})$$

$$(\forall i \in iter)(b^{(i+1)} = b^{(i)} \vee b^{(i+1)} = b^{(i)} + 1)$$

$$(\forall i \in iter)(a^{(i)} = a^{(0)} + i)$$

$$(\forall j, k \in iter)(k \geq j \rightarrow b^{(k)} \geq b^{(j)})$$

$$(\forall j, k \in iter)(k \geq j \rightarrow b^{(j)} + k \geq b^{(k)} + j)$$

$$(\forall p)(b^{(0)} \leq p < b^{(n)} \rightarrow (\exists i \in iter)(b^{(i)} = p \wedge A[a^{(i)}] \geq 0))$$

$$(\forall i)\neg upd_B(i, p) \rightarrow B^{(n)}[p] = B^{(0)}[p]$$

$$upd_B(i, p, x) \wedge (\forall j > i)\neg upd_B(j, p) \rightarrow B^{(n)}[p] = x$$

$$(\forall i \in iter)(A[a^{(i)}] \geq 0 \rightarrow B^{(i+1)}[b^{(i)}] = A[a^{(i)}] \wedge b^{(i+1)} = b^{(i)} + 1 \wedge c^{(i+1)} = c^{(i)} \;)$$

Invariant Generation - The Method

```
a := 0; b := 0; c := 0;
while (a ≤ k) do
  if A[a] ≥ 0
    then B[b] := A[a];b := b + 1;
    else C[c] := A[a];c := c + 1;
  a := a + 1;
end while
```

1. Extend the language $\mathcal{L}$ to $\mathcal{L}'$:
   - variables as functions of n: $v^{(i)}$ with $0 \leq i < n$
   - predicates as loop properties: *iter*

$$(\forall i)(i \in iter \Leftrightarrow 0 \leq i \wedge i < n)$$

2. Collect loop properties:

- Polynomial scalar properties
- Monotonicity properties of scalars
- Update predicates of arrays
- Translation of guarded assignments

$$a = b + c,\ a \geq 0,\ b \geq 0,\ c \geq 0$$

$$(\forall i \in iter)(a^{(i+1)} > a^{(i)})$$

$$(\forall i \in iter)(b^{(i+1)} = b^{(i)} \vee b^{(i+1)} = b^{(i)} + 1)$$

$$(\forall i \in iter)(a^{(i)} = a^{(0)} + i)$$

$$(\forall j, k \in iter)(k \geq j \rightarrow b^{(k)} \geq b^{(j)})$$

$$(\forall j, k \in iter)(k \geq j \rightarrow b^{(j)} + k \geq b^{(k)} + j)$$

$$(\forall p)(b^{(0)} \leq p < b^{(n)} \rightarrow (\exists i \in iter)(b^{(i)} = p \wedge A[a^{(i)}] \geq 0))$$

$$(\forall i)\neg upd_B(i, p) \rightarrow B^{(n)}[p] = B^{(0)}[p]$$

$$upd_B(i, p, x) \wedge (\forall j > i)\neg upd_B(j, p) \rightarrow B^{(n)}[p] = x$$

$$(\forall i \in iter)(A[a^{(i)}] \geq 0 \rightarrow B^{(i+1)}[b^{(i)}] = A[a^{(i)}] \wedge b^{(i+1)} = b^{(i)} + 1 \wedge c^{(i+1)} = c^{(i)}\ )$$

Invariant Generation - The Method

```
a := 0; b := 0; c := 0;
while (a ≤ k) do
  if A[a] ≥ 0
    then B[b] := A[a];b := b + 1;
    else C[c] := A[a];c := c + 1;
  a := a + 1;
end while
```

1. Extend the language $\mathcal{L}$ to $\mathcal{L}'$:
   - variables as functions of n: $v^{(i)}$ with $0 \leq i < n$
   - predicates as loop properties: $iter$

$$(\forall i)(i \in iter \Leftrightarrow 0 \leq i \wedge i < n)$$

2. Collect loop properties:

- Polynomial scalar properties
- Monotonicity properties of scalars
- Update predicates of arrays
- Translation of guarded assignments

$$a = b + c,\ a \geq 0,\ b \geq 0,\ c \geq 0$$

$$(\forall i \in iter)(a^{(i+1)} > a^{(i)})$$

$$(\forall i \in iter)(b^{(i+1)} = b^{(i)} \vee b^{(i+1)} = b^{(i)} + 1)$$

$$(\forall i \in iter)(a^{(i)} = a^{(0)} + i)$$

$$(\forall j, k \in iter)(k \geq j \rightarrow b^{(k)} \geq b^{(j)})$$

$$(\forall j, k \in iter)(k \geq j \rightarrow b^{(j)} + k \geq b^{(k)} + j)$$

$$(\forall p)(b^{(0)} \leq p < b^{(n)} \rightarrow (\exists i \in iter)(b^{(i)} = p \wedge A[a^{(i)}] \geq 0))$$

$$(\forall i) \neg upd_B(i, p) \rightarrow B^{(n)}[p] = B^{(0)}[p]$$

$$upd_B(i, p, x) \wedge (\forall j > i) \neg upd_B(j, p) \rightarrow B^{(n)}[p] = x$$

$$(\forall i \in iter)(A[a^{(i)}] \geq 0 \rightarrow B^{(i+1)}[b^{(i)}] = A[a^{(i)}] \wedge b^{(i+1)} = b^{(i)} + 1 \wedge c^{(i+1)} = c^{(i)}\ )$$

Invariant Generation - The Method

```
a := 0; b := 0; c := 0;
while (a ≤ k) do
  if A[a] ≥ 0
    then B[b] := A[a];b := b + 1;
    else C[c] := A[a];c := c + 1;
  a := a + 1;
end while
```

1. Extend the language $\mathcal{L}$ to $\mathcal{L}'$:
   - variables as functions of n: $v^{(i)}$ with $0 \leq i < n$
   - predicates as loop properties: $iter$

$$(\forall i)(i \in iter \Leftrightarrow 0 \leq i \wedge i < n)$$

2. Collect loop properties:

- Polynomial scalar properties
- Monotonicity properties of scalars
- Update predicates of arrays
- Translation of guarded assignments

$$a = b + c,\ a \geq 0,\ b \geq 0,\ c \geq 0$$

$$(\forall i \in iter)(a^{(i+1)} > a^{(i)})$$

$$(\forall i \in iter)(b^{(i+1)} = b^{(i)} \vee b^{(i+1)} = b^{(i)} + 1)$$

$$(\forall i \in iter)(a^{(i)} = a^{(0)} + i)$$

$$(\forall j, k \in iter)(k \geq j \rightarrow b^{(k)} \geq b^{(j)})$$

$$(\forall j, k \in iter)(k \geq j \rightarrow b^{(j)} + k \geq b^{(k)} + j)$$

$$(\forall p)(b^{(0)} \leq p < b^{(n)} \rightarrow (\exists i \in iter)(b^{(i)} = p \wedge A[a^{(i)}] \geq 0))$$

$$(\forall i)\neg upd_B(i, p) \rightarrow B^{(n)}[p] = B^{(0)}[p]$$

$$upd_B(i, p, x) \wedge (\forall j > i)\neg upd_B(j, p) \rightarrow B^{(n)}[p] = x$$

$$(\forall i \in iter)(A[a^{(i)}] \geq 0 \rightarrow B^{(i+1)}[b^{(i)}] = A[a^{(i)}] \wedge b^{(i+1)} = b^{(i)} + 1 \wedge c^{(i+1)} = c^{(i)}\ )$$

3. Eliminate symbols $\rightarrow$ Invariants

Invariant Generation - The Method

$a := 0; b := 0; c := 0;$
while $(a \leq k)$ **do**
  if $A[a] \geq 0$
    then $B[b] := A[a]; b := b + 1;$
    else $C[c] := A[a]; c := c + 1;$
  $a := a + 1;$
end while

1. Extend the language $\mathcal{L}$ to $\mathcal{L}'$:
   - variables as functions of n: $v^{(i)}$ with $0 \leq i < n$
   - predicates as loop properties: *iter*

$$(\forall i)(i \in iter \Leftrightarrow 0 \leq i \wedge i < n)$$

2. Collect loop properties:

- Polynomial scalar properties
- Monotonicity properties of scalars
- Update predicates of arrays
- Translation of guarded assignments

$$a = b + c,\ a \geq 0,\ b \geq 0,\ c \geq 0$$

$$(\forall i \in iter)(a^{(i+1)} > a^{(i)})$$

$$(\forall i \in iter)(b^{(i+1)} = b^{(i)} \vee b^{(i+1)} = b^{(i)} + 1)$$

$$(\forall i \in iter)(a^{(i)} = a^{(0)} + i)$$

$$(\forall j, k \in iter)(k \geq j \rightarrow b^{(k)} \geq b^{(j)})$$

$$(\forall j, k \in iter)(k \geq j \rightarrow b^{(j)} + k \geq b^{(k)} + j)$$

$$(\forall p)(b^{(0)} \leq p < b^{(n)} \rightarrow (\exists i \in iter)(b^{(i)} = p \wedge A[a^{(i)}] \geq 0))$$

$$(\forall i) \neg upd_B(i, p) \rightarrow B^{(n)}[p] = B^{(0)}[p]$$

$$upd_B(i, p, x) \wedge (\forall j > i) \neg upd_B(j, p) \rightarrow B^{(n)}[p] = x$$

$$(\forall i \in iter)(A[a^{(i)}] \geq 0 \rightarrow B^{(i+1)}[b^{(i)}] = A[a^{(i)}] \wedge b^{(i+1)} = b^{(i)} + 1 \wedge c^{(i+1)} = c^{(i)})$$

3. Eliminate symbols $\rightarrow$ Invariants

HOW?

Invariant Generation by Symbol Elimination

$(\forall i)(i \in iter \Leftrightarrow 0 \leq i \wedge i < n)$
$upd_B(i,p) \Leftrightarrow i \in iter \wedge p = b^{(i)} \wedge A[a^{(i)}] \geq 0$
$upd_B(i,p,x) \Leftrightarrow upd_B(i,p) \wedge x = A[a^{(i)}]$
$a = b + c,\ a \geq 0,\ b \geq 0,\ c \geq 0$
$(\forall i \in iter)(a^{(i+1)} > a^{(i)})$
$(\forall i \in iter)(b^{(i+1)} = b^{(i)} \vee b^{(i+1)} = b^{(i)} + 1)$
$(\forall i \in iter)(a^{(i)} = a^{(0)} + i)$
$(\forall j,k \in iter)(k \geq j \rightarrow b^{(k)} \geq b^{(j)})$
$(\forall j,k \in iter)(k \geq j \rightarrow b^{(j)} + k \geq b^{(k)} + j)$
$(\forall p)(b^{(0)} \leq p < b^{(n)} \rightarrow (\exists i \in iter)(b^{(i)} = p \wedge A[a^{(i)}] \geq 0))$
$(\forall i)\neg upd_B(i,p) \rightarrow B^{(n)}[p] = B^{(0)}[p]$
$upd_B(i,p,x) \wedge (\forall j > i)\neg upd_B(j,p) \rightarrow B^{(n)}[p] = x$
$(\forall i \in iter)(A[a^{(i)}] \geq 0 \rightarrow B^{(i+1)}[b^{(i)}] = A[a^{(i)}] \wedge b^{(i+1)} = b^{(i)} + 1 \wedge c^{(i+1)} = c^{(i)})$

$\xrightarrow[\text{Theorem Proving}]{\text{First-Order}}$ $I_1, I_2, I_3, I_4, I_5, \ldots$

Outline

First-Order Theorem Proving. Example

Group theory theorem: if a group satisfies the identity $x^2 = 1$, then it is commutative.

First-Order Theorem Proving. Example

Group theory theorem: if a group satisfies the identity $x^2 = 1$, then it is commutative.
More formally: in a group "assuming that $x^2 = 1$ for all x prove that $x \cdot y = y \cdot x$ holds for all x, y."

First-Order Theorem Proving. Example

Group theory theorem: if a group satisfies the identity $x^2 = 1$, then it is commutative.
More formally: in a group "assuming that $x^2 = 1$ for all x prove that $x \cdot y = y \cdot x$ holds for all x, y."
What is implicit: axioms of the group theory.

$$
\begin{array}{l}
\forall x(1 \cdot x = x) \\
\forall x(x^{-1} \cdot x = 1) \\
\forall x \forall y \forall z((x \cdot y) \cdot z = x \cdot (y \cdot z))
\end{array}
$$

Formulation in First-Order Logic

Axioms (of group theory):	$\forall x(1 \cdot x = x)$ $\forall x(x^{-1} \cdot x = 1)$ $\forall x \forall y \forall z((x \cdot y) \cdot z = x \cdot (y \cdot z))$
Assumptions:	$\forall x(x \cdot x = 1)$
Conjecture:	$\forall x \forall y(x \cdot y = y \cdot x)$

In the TPTP Syntax

The TPTP library (Thousands of Problems for Theorem Provers), `http://www.tptp.org` contains a large collection of first-order problems. For representing these problems it uses the TPTP syntax, which is understood by all modern theorem provers, including Vampire.

In the TPTP Syntax

The TPTP library (Thousands of Problems for Theorem Provers), `http://www.tptp.org` contains a large collection of first-order problems. For representing these problems it uses the TPTP syntax, which is understood by all modern theorem provers, including Vampire.

First-Order Logic (FOL)	TPTP
$\bot, \top$	`$false, $true`
$\neg F$	`~F`
$F_1 \land \ldots \land F_n$	`F1 & ... & Fn`
$F_1 \lor \ldots \lor F_n$	`F1 | ... | Fn`
$F_1 \rightarrow F_n$	`F1 => Fn`
$(\forall x_1) \ldots (\forall x_n) F$	`! [X1,...,Xn] : F`
$(\exists x_1) \ldots (\exists x_n) F$	`? [X1,...,Xn] : F`

In the TPTP Syntax

The TPTP library (Thousands of Problems for Theorem Provers), http://www.tptp.org contains a large collection of first-order problems. For representing these problems it uses the TPTP syntax, which is understood by all modern theorem provers, including Vampire.

In the TPTP syntax this group theory problem can be written down as follows:

```
%---- 1 * x = 1
fof(left_identity,axiom,
! [X] : mult(e,X) = X).
%---- i(x) * x = 1
fof(left_inverse,axiom,
! [X] : mult(inverse(X),X) = e).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,
! [X,Y,Z] : mult(mult(X,Y),Z) = mult(X,mult(Y,Z))).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
! [X] : mult(X,X) = e).
%---- prove x * y = y * x
fof(commutativity,conjecture,
! [X] : mult(X,Y) = mult(Y,X)).
```

More on the TPTP Syntax

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
       mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

More on the TPTP Syntax

- Comments;

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
      mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

More on the TPTP Syntax

- Comments;
- Input formula names;

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
       mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

More on the TPTP Syntax

- Comments;
- Input formula names;
- Input formula roles (very important);

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
       mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

More on the TPTP Syntax

- Comments;
- Input formula names;
- Input formula roles (very important);
- Equality

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
       mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

Running Vampire on a TPTP file

is easy: simply use

```
vampire <filename>
```

Running Vampire on a TPTP file

is easy: simply use

```
vampire <filename>
```

One can also run Vampire with various options, some of them will be explained later. For example, save the group theory problem in a file `group.tptp` and try

```
vampire                  group.tptp
```

Running Vampire on a TPTP file

is easy: simply use

```
vampire <filename>
```

One can also run Vampire with various options, some of them will be explained later. For example, save the group theory problem in a file `group.tptp` and try

```
vampire --thanks LCCC group.tptp
```

Proof by Vampire (Slightliy Modified)

```
Refutation found.
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

Proof by Vampire (Slightliy Modified)

```
Refutation found.
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;

Proof by Vampire (Slightliy Modified)

```
Refutation found.
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus

Proof by Vampire (Slightliy Modified)

```
Refutation found.
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus

Proof by Vampire (Slightliy Modified)

```
Refutation found.
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus

Proof by Vampire (Slightliy Modified)

```
Refutation found.
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus

Proof by Vampire (Slightliy Modified)

```
Refutation found.
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus
- Proof by refutation, generating and simplifying inferences, unused formulas ...

Proof by Vampire (Slightliy Modified)

```
Refutation found.
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus
- Proof by refutation, generating and simplifying inferences, unused formulas . . .

Proof by Vampire (Slightliy Modified)

```
Refutation found.
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus
- Proof by refutation, generating and simplifying inferences, unused formulas . . .

Vampire

- Completely automatic: once you started a proof attempt, it can only be interrupted by terminating the process.

Vampire

- Completely automatic: once you started a proof attempt, it can only be interrupted by terminating the process.
- Champion of the CASC world-cup in first-order theorem proving: won CASC 30 times.

What an Automatic Theorem Prover is Expected to Do

Input:

- a set of axioms (first order formulas) or clauses;
- a conjecture (first-order formula or set of clauses).

Output:

- proof (hopefully).

Proof by Refutation

Given a problem with axioms and assumptions $F_1, \ldots, F_n$ and conjecture G,

1. negate the conjecture ($\neg G$);
2. establish unsatisfiability of the set of formulas $F_1, \ldots, F_n, \neg G$.

Proof by Refutation

Given a problem with axioms and assumptions $F_1, \ldots, F_n$ and conjecture G,

1. negate the conjecture ($\neg G$);
2. establish unsatisfiability of the set of formulas $F_1, \ldots, F_n, \neg G$.

Thus, we reduce the theorem proving problem to the problem of checking unsatisfiability.

Proof by Refutation

Given a problem with axioms and assumptions $F_1, \ldots, F_n$ and conjecture G,

1. negate the conjecture ($\neg G$);
2. establish unsatisfiability of the set of formulas $F_1, \ldots, F_n, \neg G$.

Thus, we reduce the theorem proving problem to the problem of checking unsatisfiability.

In this formulation the negation of the conjecture $\neg G$ is treated like any other formula. In fact, Vampire (and other provers) internally treat conjectures differently, to make proof search more goal-oriented.

General Scheme (simplified)

- Read a problem;
- Determine proof-search options to be used for this problem;
- Preprocess the problem;
- Convert it into CNF;
- Run a saturation algorithm on it, try to derive *false*.
- If *false* is derived, report the result, maybe including a refutation.

General Scheme (simplified)

- Read a problem;
- Determine proof-search options to be used for this problem;
- Preprocess the problem;
- Convert it into CNF;
- Run a saturation algorithm on it, try to derive *false*.
- If *false* is derived, report the result, maybe including a refutation.

Trying to derive *false* using a saturation algorithm is the hardest part, which in practice may not terminate or run out of memory.

Inference System

First-order theorem provers prove using an inference system.

- An inference has the form

$$\frac{F_1 \quad \ldots \quad F_n}{G},$$

  where $n \geq 0$ and $F_1, \ldots, F_n, G$ are formulas.
- The formula G is called the conclusion of the inference;
- The formulas $F_1, \ldots, F_n$ are called its premises.
- An inference rule R is a set of inferences.
- An inference system $\mathbb{I}$ is a set of inference rules.
- Axiom: inference rule with no premises.

Inference System

First-order theorem provers prove using an inference system.

- An inference has the form

$$\frac{F_1 \quad \ldots \quad F_n}{G} \; ,$$

  where $n \geq 0$ and $F_1, \ldots, F_n, G$ are formulas.
- The formula G is called the conclusion of the inference;
- The formulas $F_1, \ldots, F_n$ are called its premises.
- An inference rule R is a set of inferences.
- An inference system $\mathbb{I}$ is a set of inference rules.
- Axiom: inference rule with no premises.

Inference System

First-order theorem provers prove using an inference system.

- An inference has the form

$$\frac{F_1 \quad \ldots \quad F_n}{G},$$

where $n \geq 0$ and $F_1, \ldots, F_n, G$ are formulas.
- The formula G is called the conclusion of the inference;
- The formulas $F_1, \ldots, F_n$ are called its premises.
- An inference rule R is a set of inferences.
- An inference system $\mathbb{I}$ is a set of inference rules.
- Axiom: inference rule with no premises.

Derivation, Proof

- Derivation in an inference system $\mathbb{I}$: a tree built from inferences in $\mathbb{I}$.
- Proof of E: a finite derivation whose leaves are axioms.

Clauses

- Literal: either an atom A or its negation $\neg A$.
- Clause: a disjunction $L_1 \vee \ldots \vee L_n$ of literals, where $n \geq 0$.

Clauses

- Literal: either an atom A or its negation $\neg A$.
- Clause: a disjunction $L_1 \vee \ldots \vee L_n$ of literals, where $n \geq 0$.
- Empty clause, denoted by $\square$: clause with 0 literals, that is, when $n = 0$.

Clauses

- Literal: either an atom A or its negation $\neg A$.
- Clause: a disjunction $L_1 \vee \ldots \vee L_n$ of literals, where $n \geq 0$.
- Empty clause, denoted by $\square$: clause with 0 literals, that is, when $n = 0$. The $\square$ is equivalent to false.

Clauses

- Literal: either an atom A or its negation $\neg A$.
- Clause: a disjunction $L_1 \vee \ldots \vee L_n$ of literals, where $n \geq 0$.
- Empty clause, denoted by $\square$: clause with 0 literals, that is, when $n = 0$. The $\square$ is equivalent to false.
- A formula in Clausal Normal Form (CNF): a conjunction of clauses.

Soundness

- An inference is sound if the conclusion of this inference is a logical consequence of its premises.
- An inference system is sound if every inference rule in this system is sound.

Soundness

- An inference is sound if the conclusion of this inference is a logical consequence of its premises.
- An inference system is sound if every inference rule in this system is sound.

Consequence of soundness: let S be a set of clauses. If $\square$ can be derived from S in a sound inference system $\mathbb{I}$, then S is unsatisfiable.

Can this be used for checking (un)satisfiability

1. What happens when the empty clause cannot be derived from S?

Can this be used for checking (un)satisfiability

1. Completeness of an inference system $\mathbb{I}$.

   Let S be an unsatisfiable set of clauses. Then there exists a derivation of $\square$ from S in $\mathbb{I}$.

Can this be used for checking (un)satisfiability

1. Completeness of an inference system $\mathbb{I}$.

   Let S be an unsatisfiable set of clauses. Then there exists a derivation of $\square$ from S in $\mathbb{I}$.

2. How to establish unsatisfiability?

How to Establish Unsatisfiability?

Completess is formulated in terms of derivability of the empty clause $\square$ from a set S_0 of clauses in an inference system $\mathbb{I}$. However, this formulations gives no hint on how to search for such a derivation.

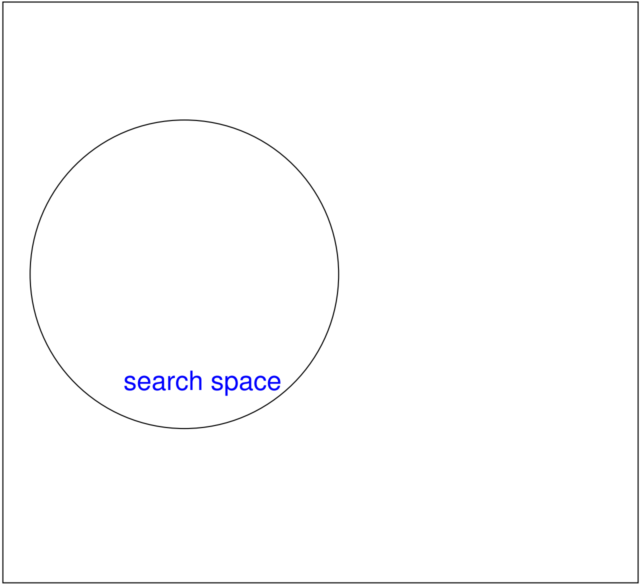
How to Establish Unsatisfiability?

Completess is formulated in terms of derivability of the empty clause $\square$ from a set S_0 of clauses in an inference system $\mathbb{I}$. However, this formulations gives no hint on how to search for such a derivation.

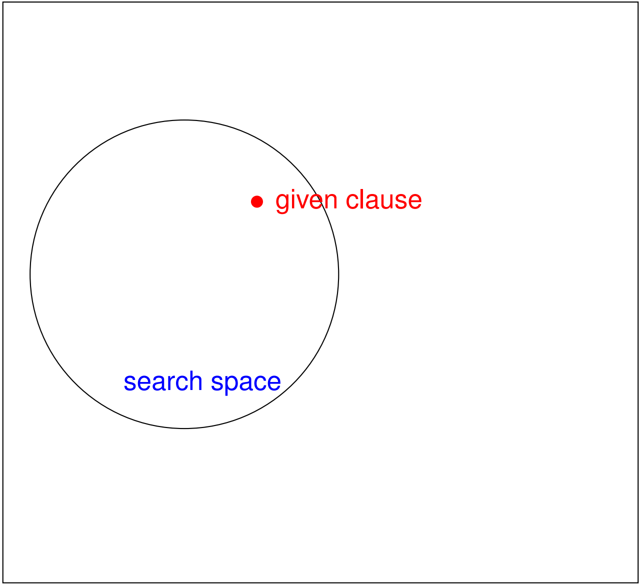
Idea:

- Take a set of clauses S (the search space), initially $S = S_0$. Repeatedly apply inferences in $\mathbb{I}$ to clauses in S and add their conclusions to S, unless these conclusions are already in S.
- If, at any stage, we obtain $\square$, we terminate and report unsatisfiability of S_0.

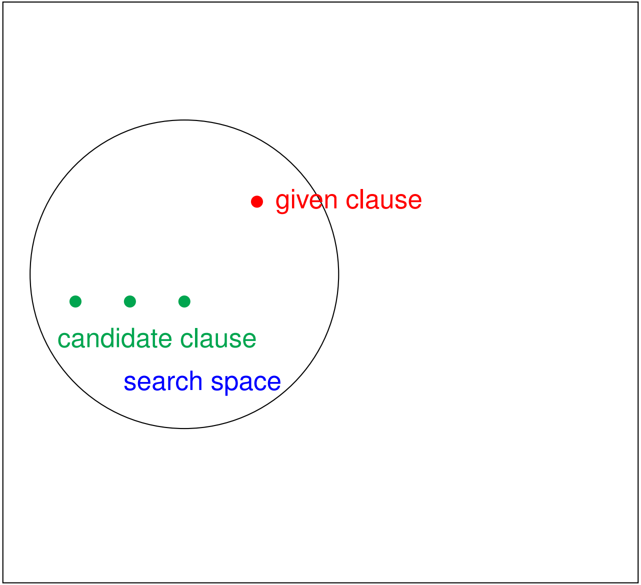
Saturation Algorithms



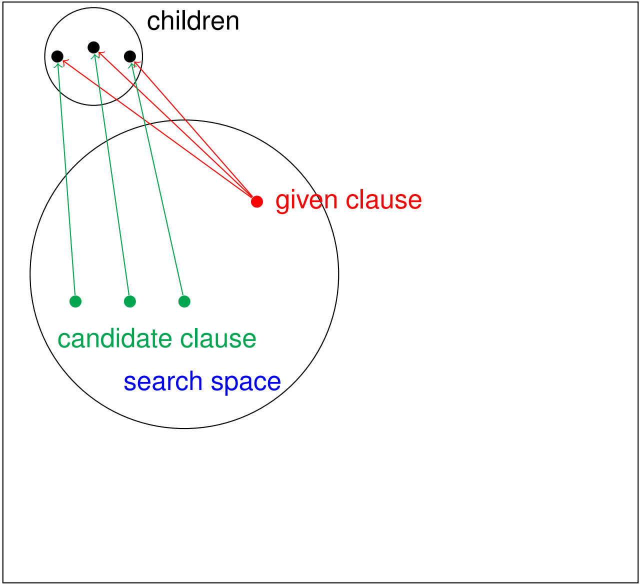
Saturation Algorithms



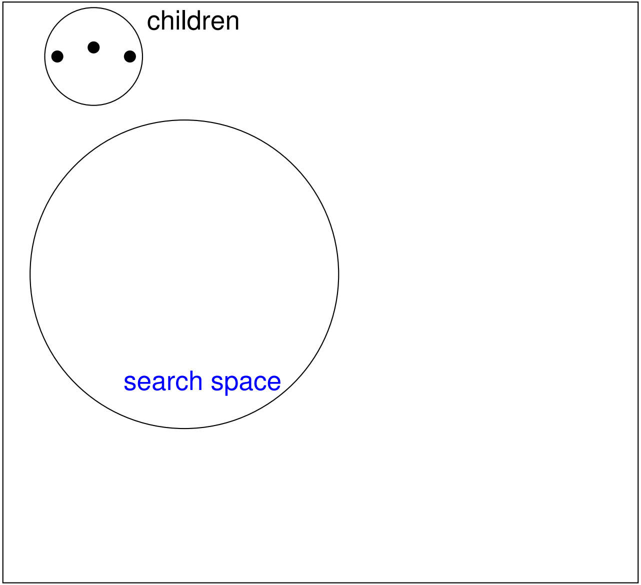
Saturation Algorithms



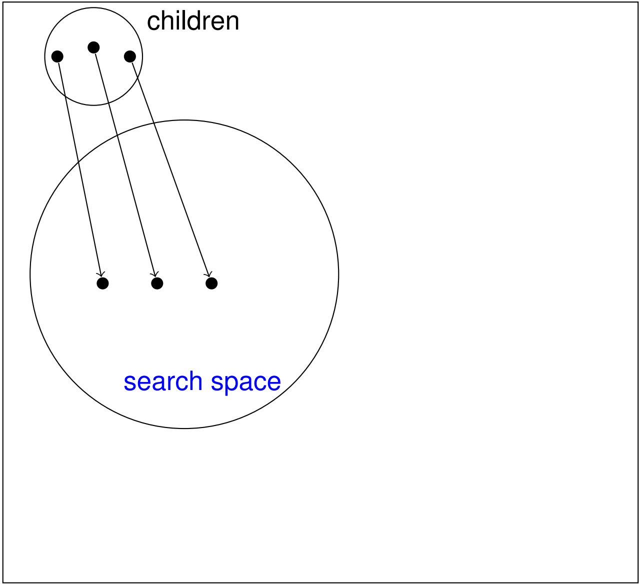
Saturation Algorithms



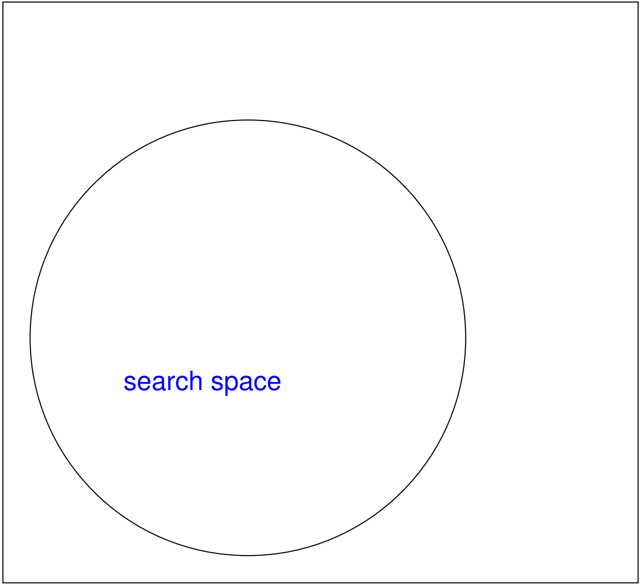
Saturation Algorithms



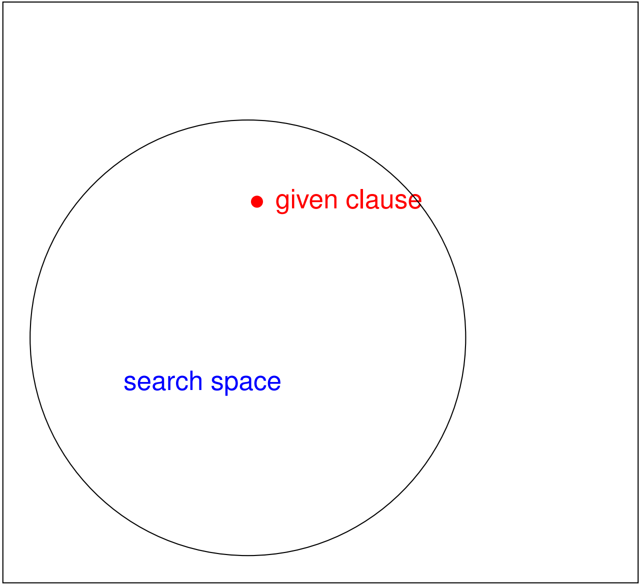
Saturation Algorithms



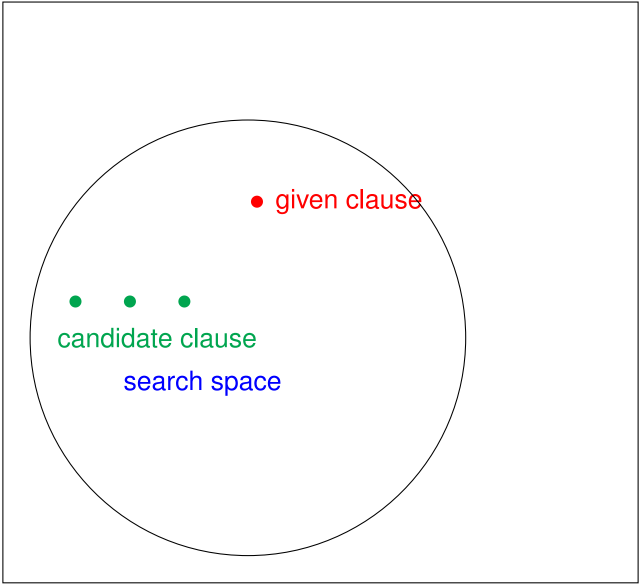
Saturation Algorithms



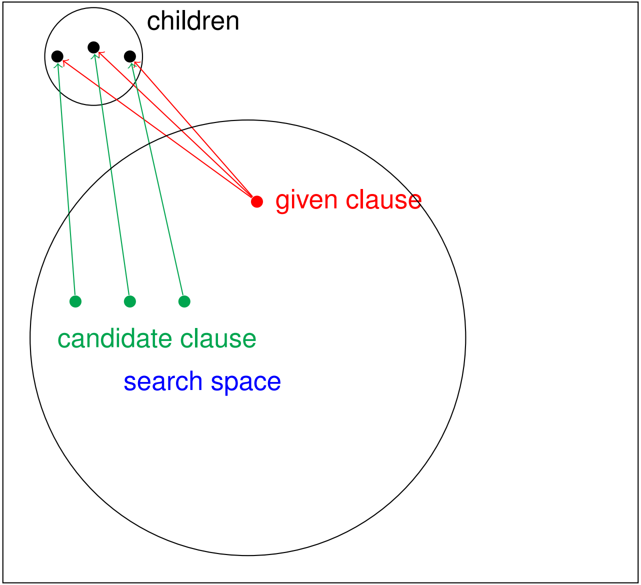
Saturation Algorithms



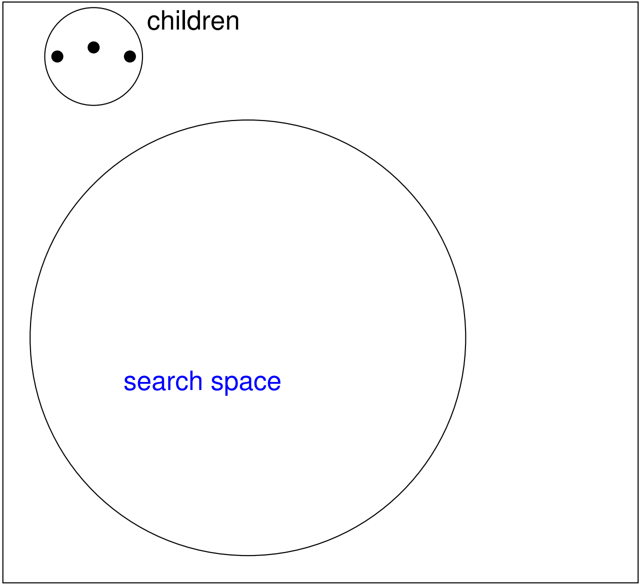
Saturation Algorithms



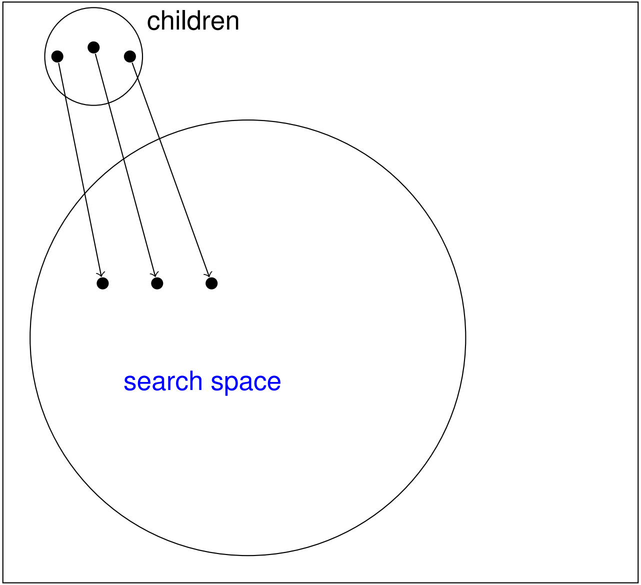
Saturation Algorithms



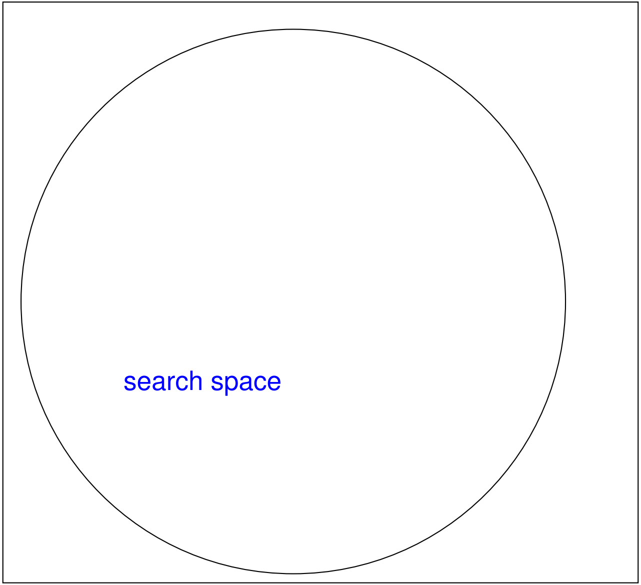
Saturation Algorithms



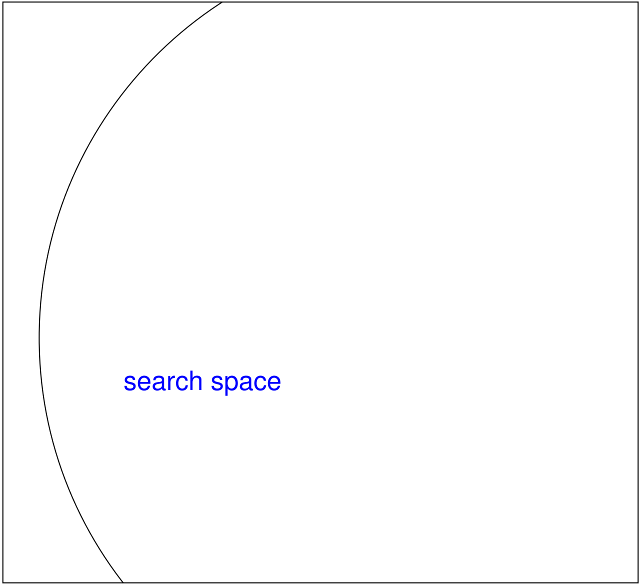
Saturation Algorithms



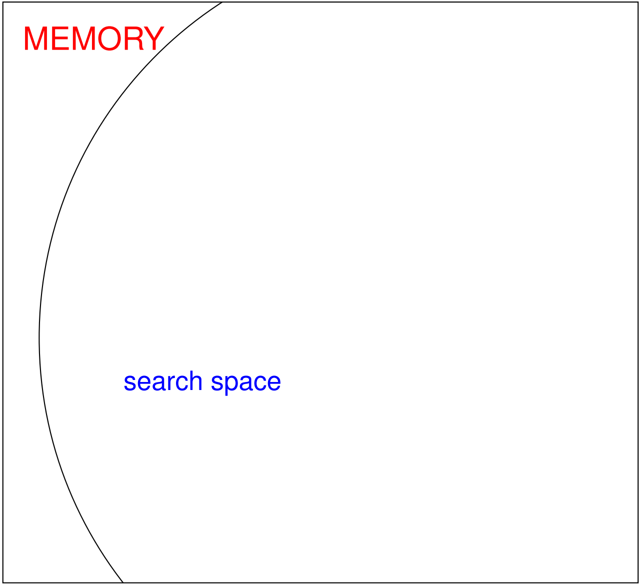
Saturation Algorithms



Saturation Algorithms



Saturation Algorithms



Saturation Algorithm

A saturation algorithm tries to saturate a set of clauses with respect to a given inference system.
In theory there are three possible scenarios:

1. At some moment the empty clause $\square$ is generated, in this case the input set of clauses is unsatisfiable.
2. Saturation will terminate without ever generating $\square$, in this case the input set of clauses in satisfiable.
3. Saturation will run forever, but without generating $\square$. In this case the input set of clauses is satisfiable.

Saturation Algorithm in Practice

In practice there are three possible scenarios:

1. At some moment the empty clause $\square$ is generated, in this case the input set of clauses is unsatisfiable.
2. Saturation will terminate without ever generating $\square$, in this case the input set of clauses in satisfiable.
3. Saturation will run until we run out of resources, but without generating $\square$. In this case it is unknown whether the input set is unsatisfiable.

From Theory to Practice

In practice, saturation theorem provers implement:

- Preprocessing and CNF transformation;
- Superposition system;
- Orderings and selection functions;
- Fairness (saturation algorithms);
- Deletion and generation of clauses in the search space;
- Many, many proof options and stragegies

From Theory to Practice

In practice, saturation theorem provers implement:

- Preprocessing and CNF transformation;
- Superposition system;
- Orderings and selection functions;
- Fairness (saturation algorithms);
- Deletion and generation of clauses in the search space;
- Many, many proof options and stragegies

.

From Theory to Practice

In practice, saturation theorem provers implement:

- Preprocessing and CNF transformation;
- Superposition system;
- Orderings and selection functions;
- Fairness (saturation algorithms);
- Deletion and generation of clauses in the search space;
- Many, many proof options and stragegies
  – example: limited resource strategy.

From Theory to Practice

In practice, saturation theorem provers implement:

- Preprocessing and CNF transformation;
- Superposition system;
- Orderings and selection functions;
- Fairness (saturation algorithms);
- Deletion and generation of clauses in the search space;
- Many, many proof options and stragegies
  – example: limited resource strategy.

Try:

```
vampire --age_weight_ratio 10:1
  --backward_subsumption off
  --time_limit 86400
      GRP140-1.p
```

Outline

Invariant Generation by Symbol Elimination

$$(\forall i)(i \in iter \Leftrightarrow 0 \leq i \land i < n)$$
$$upd_B(i,p) \Leftrightarrow i \in iter \land p = b^{(i)} \land A[a^{(i)}] \geq 0$$
$$upd_B(i,p,x) \Leftrightarrow upd_B(i,p) \land x = A[a^{(i)}]$$
$$a = b + c,\ a \geq 0,\ b \geq 0,\ c \geq 0$$
$$(\forall i \in iter)(a^{(i+1)} > a^{(i)})$$
$$(\forall i \in iter)(b^{(i+1)} = b^{(i)} \lor b^{(i+1)} = b^{(i)} + 1)$$
$$(\forall i \in iter)(a^{(i)} = a^{(0)} + i)$$
$$(\forall j,k \in iter)(k \geq j \to b^{(k)} \geq b^{(j)})$$
$$(\forall j,k \in iter)(k \geq j \to b^{(j)} + k \geq b^{(k)} + j)$$
$$(\forall p)(b^{(0)} \leq p < b^{(n)} \to (\exists i \in iter)(b^{(i)} = p \land A[a^{(i)}] \geq 0))$$
$$(\forall i)\neg upd_B(i,p) \to B^{(n)}[p] = B^{(0)}[p]$$
$$upd_B(i,p,x) \land (\forall j > i)\neg upd_B(j,p) \to B^{(n)}[p] = x$$
$$(\forall i \in iter)(A[a^{(i)}] \geq 0 \to B^{(i+1)}[b^{(i)}] = A[a^{(i)}] \land b^{(i+1)} = b^{(i)} + 1 \land c^{(i+1)} = c^{(i)})$$

$\xrightarrow[\text{Theorem Proving}]{\text{Saturation}}$ $I_1, I_2, I_3, I_4, I_5, \ldots$

Conclusions: Program Analysis by First-Order Theorem Proving

Given a loop:

1. Express loop properties in a language containing extra symbols (loop counter, predicates expressing array updates, etc.);
2. Every logical consequence of these properties is a valid loop property, but not an invariant;
3. Run a theorem prover for eliminating extra symbols;
4. Every derived formula in the language of the loop is a loop invariant;
5. Invariants are **consequences of symbol-eliminating inferences (SEI)**.

SEI: premise contains extra symbols, conclusion is in the loop language.