# *{log}*

## programming and automated proof in set theory


maximiliano cristiá

universidad nacional de rosario and cifasis 

cristia@cifasis-conicet.gov.ar

argentina

joint work with

gianfranco rossi

università di parma

italy


tutorial for abz 2023 — nancy, france — may, 2023

revised version april 2025

# install {*log*}

install swi-prolog

`www.swi-prolog.org`

download and unzip {*log*} in any directory

`www.clpset.unipr.it`

# install {*log*}

check if everything is fine

open a command-line terminal, go to the {*log*} folder

`~/setlog$ swipl` or **`swipl-win`**

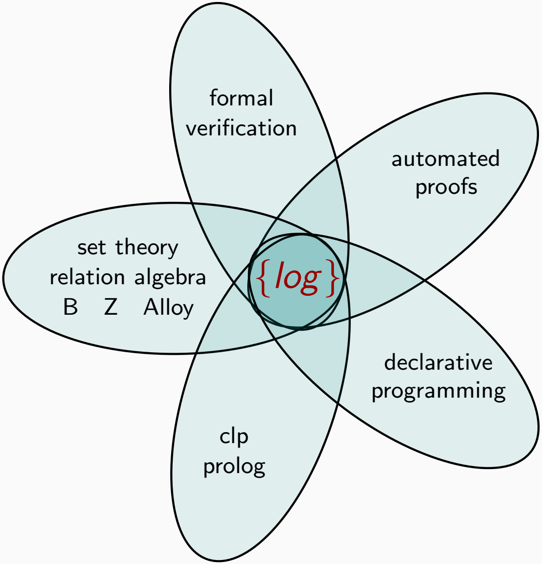
```
?- consult('setlog.pl').

?- setlog.

{log}=>
```

# introduction to {*log*}

# keywords


formal
verification
automated
proofs
set theory
relation algebra
B Z Alloy
{log}
declarative
programming
clp
prolog

programming, proof and counterexample generation in set theory

original development (1991)

a. dovier e. omodeo e. pontelli g. rossi

current development (2012)

m. cristiá g. rossi

# {*log*}

constraint logic programming (clp) language

first-class entities

- finite sets and set operators
- finite binary relations and relational operators
- restricted quantifiers
- linear integer arithmetic

finite set relation algebra + arithmetic + quantifiers

# {*log*}

syntactic unification + set unification

satisfiability solver

automated theorem prover

model finder, counterexample generator

prolog implementation

programs as formulas

formulas as programs

vs

programs as proofs

# programs as formulas, formulas as programs

**problem** compute the minimum of set $A$

# programs as formulas, formulas as programs

**problem** compute the minimum of set $A$

**specification** $min(A, m) \triangleq m \in A \wedge \forall x (x \in A \implies m \leq x)$

# programs as formulas, formulas as programs

**problem** compute the minimum of set $A$

**specification** $min(A, m) \triangleq m \in A \land \forall x(x \in A \implies m \leq x)$

**in** $\{log\}$

```
smin(A,M) :- M in A & foreach(X in A, M =< X).
```

# programs as formulas, formulas as programs

**problem** compute the minimum of set $A$

**specification** $min(A, m) \triangleq m \in A \land \forall x(x \in A \implies m \leq x)$

**in** $\{log\}$

```
smin(A,M) :- M in A & foreach(X in A, M =< X).
```

**why a formula?**

# programs as formulas, formulas as programs

**problem** compute the minimum of set $A$

**specification** $min(A, m) \triangleq m \in A \land \forall x(x \in A \implies m \leq x)$

**in $\{log\}$** `smin(A,M) :- M in A & foreach(X in A, M =< X).`

**why a formula?** `foreach(X in A, M =< X) & M in A`

# programs as formulas, formulas as programs

**problem** compute the minimum of set $A$

**specification** $min(A, m) \triangleq m \in A \land \forall x(x \in A \implies m \leq x)$

**in** $\{log\}$ `smin(A,M) :- M in A & foreach(X in A, M =< X).`

**why a formula?** `foreach(X in A, M =< X) & M in A`

**why a program?**

# programs as formulas, formulas as programs

**problem** compute the minimum of set $A$

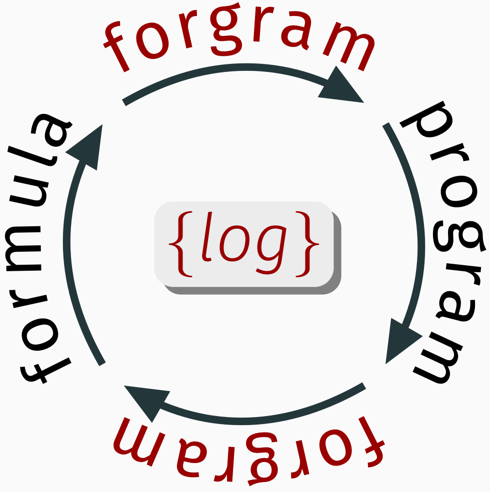
**specification** $min(A, m) \triangleq m \in A \land \forall x(x \in A \implies m \leq x)$

**in** $\{log\}$ `smin(A,M) :- M in A & foreach(X in A, M =< X).`

**why a formula?** `foreach(X in A, M =< X) & M in A`

**why a program?** `smin({3,6,1},Min)` $\rightarrow$ `Min = 1`

formula }
program } forgram




the formula-program duality

# `smin` is a forgram (program)

`smin({19,23,7},M)` $\rightarrow$ `M = 7`

# `smin` is a forgram (program)

`smin({19,23,7},M)` $\rightarrow$ `M = 7`

`smin({19,X,7},M)` `X` variable

$\rightarrow$ `M = X, X =< 19, X =< 7 ; M = 7, 7 =< X`

## `smin` is a forgram (program)

`smin({19,23,7},M)` $\rightarrow$ `M = 7`

`smin({19,X,7},M)` — `X` variable

$\rightarrow$ `M = X, X =< 19, X =< 7 ; M = 7, 7 =< X`

`smin({19,23,X},3)` $\rightarrow$ `X = 3`

# `smin` is a forgram (formula)

the following property is true of the specification

$$min(A, m) \wedge min(B, n) \wedge A \subseteq B \implies n \leq m$$

# `smin` is a forgram (formula)

the following property is true of the specification

$$min(A, m) \land min(B, n) \land A \subseteq B \implies n \leq m$$

does the $\{log\}$ forgram verifies the same property?

# `smin` is a forgram (formula)

the following property is true of the specification

$$min(A, m) \land min(B, n) \land A \subseteq B \implies n \leq m$$

does the $\{log\}$ forgram verifies the same property?

run this on $\{log\}$

# `smin` is a forgram (formula)

the following property is true of the specification

$min(A, m) \land min(B, n) \land A \subseteq B \implies n \leq m$

does the $\{log\}$ forgram verifies the same property?

**run this on $\{log\}$**

```
neg(                                                                          )
```

# `smin` is a forgram (formula)

the following property is true of the specification

$$min(A, m) \wedge min(B, n) \wedge A \subseteq B \implies n \leq m$$

does the $\{log\}$ forgram verifies the same property?

**run this on $\{log\}$**

```
neg( smin(A,M)                                                    )
```

# `smin` is a forgram (formula)

the following property is true of the specification

$$min(A, m) \wedge min(B, n) \wedge A \subseteq B \implies n \leq m$$

does the $\{log\}$ forgram verifies the same property?

**run this on $\{log\}$**

```
neg( smin(A,M) & smin(B,N)                                      )
```

# `smin` is a forgram (formula)

the following property is true of the specification

$$min(A, m) \land min(B, n) \land A \subseteq B \implies n \leq m$$

does the $\{log\}$ forgram verifies the same property?

run this on $\{log\}$

```
neg( smin(A,M) & smin(B,N) & subset(A,B)                  )
```

# `smin` is a forgram (formula)

the following property is true of the specification

$$min(A, m) \wedge min(B, n) \wedge A \subseteq B \implies n \leq m$$

does the $\{log\}$ forgram verifies the same property?

**run this on $\{log\}$**

```
neg( smin(A,M) & smin(B,N) & subset(A,B) implies N =< M )
```

# `smin` is a forgram (formula)

the following property is true of the specification

$$min(A, m) \wedge min(B, n) \wedge A \subseteq B \implies n \leq m$$

does the $\{log\}$ forgram verifies the same property?

run this on $\{log\}$

```
neg( smin(A,M) & smin(B,N) & subset(A,B) implies N =< M )
```

if the forgram verifies the property, the answer is `no`
otherwise $\{log\}$ produces a counterexample

```
{log}=> neg(smin(A,M) & smin(B,N) & subset(A,B)
             implies N =< M
        ).

no
```

i.e. unsatisfiable (unsat)

for **every** finite set and **every** integer number

# the sets of {*log*}

*set* ::=

*variable*

| `{}` — empty

| `{`*element* `/` *set*`}` — extensional

| `int(`*int*`,`*int*`)` — integer interval

| `cp(`*set*`,`*set*`)` — cartesian product

| `ris(`*term* `in` *set*`,`*formula*`)` — intensional

finite, untyped/typed, unbounded, nested, hybrid

unbounded → `{x / A}`, `A` can be of any finite cardinality

hybrid → non-set objects can be set elements `{1,a,[x,y]}`

nested → `{{1},{1,{2}}}`

# the operators of {*log*}

**base or primitives**

sets → `=` `in` `un` `disj` `size`

relations → `id` `inv` `comp`

**defined**

sets → `inters` `diff` `cmpt` `subset`

relations → `ran` `pfun` `dom` `rimg` `dres` `rres` `dares` ...

**restricted quantifiers**

ruq → `foreach(X in A,` *formula*`)`

req → `exists(X in A,` *formula*`)`

linear integer arithmetic

# set unification

finds out when two sets are equal

`{x / A}` is interpreted as $\{x\} \cup A$

sets may contain variables

`{X,1,a,[Y,q]}` `X`, `Y`: variables; `[_,_]`: ordered pair

sets may be partially specified

`{1,X / A}` `A`: variable

```
{log}=> {[a,1],[a,2],[b,1]} = {[a,X] / A}.
```

```
{log}=> {[a,1],[a,2],[b,1]} = {[a,X] / A}.

X = 1,  A = {[a,2],[b,1]}
```

```
{log}=> {[a,1],[a,2],[b,1]} = {[a,X] / A}.

X = 1,  A = {[a,2],[b,1]}
```

{*log*} **finds models**

```
{log}=> {[a,1],[a,2],[b,1]} = {[a,X] / A}.

X = 1,  A = {[a,2],[b,1]}

Another solution?  (y/n)
```

**{*log*} finds models**

```
{log}=> {[a,1],[a,2],[b,1]} = {[a,X] / A}.

X = 1,  A = {[a,2],[b,1]}

Another solution?  (y/n)

X = 1,  A = {[a,1],[a,2],[b,1]}
```

{*log*} finds models

```
{log}=> {[a,1],[a,2],[b,1]} = {[a,X] / A}.

X = 1,  A = {[a,2],[b,1]}

Another solution?  (y/n)

X = 1,  A = {[a,1],[a,2],[b,1]}

Another solution?  (y/n)
X = 2,  A = {[a,1],[b,1]}

Another solution?  (y/n)
X = 2,  A = {[a,1],[a,2],[b,1]}

Another solution?  (y/n)
no
```

{*log*} finds models

{*log*} returns a finite representation of all models

## self test

explore all the solutions of the following

- `{X,Y} = {Z}`
- `{[X,Y]} = {Z}`
- `{X,{X,Y}} = {W,{W,Z}}`
- `{X,{X,Y}} = {W,{W,Z}} & [X,Y] neq [W,Z]`

## recall

to write a dot at the end of the query

`t neq s` $\longrightarrow t \neq s$

# restricted intensional sets (ris)

$$\{x \in A \mid 0 \leq x\} \quad \to \quad \{x \mid x \in A \wedge 0 \leq x\}$$

$\{x \in A \mid 0 \leq x\} \quad \to$ `ris(X in A, 0 =< X)`

```
ris([X,Y] in R, X neq Y & X in B)
```

$$\to \quad \{(x, y) \in R \mid x \neq y \wedge x \in B\}$$

`ris(X in A, 0 =< X, [X,0])` $\quad \to \quad \{(x, 0) \mid x \in A \wedge 0 \leq x\}$

```
{log}=> ris(X in A, 0 =< X) = {3,M}.
```

```
{log}=> ris(X in A, 0 =< X) = {3,M}.

A = {3,M / _N1}
```

`_N1` → new existential variable

```
{log}=> ris(X in A, 0 =< X) = {3,M}.

A = {3,M / _N1}

Constraint: ris(X in _N1, [], 0 =< X, X, true) = {},
            0 =< M
```

`_N1` → new existential variable

**`Constraint`** → conjunction of constraints, always satisfiable

solution → substitute `_N1` by `{}` and `M` by `0`

```
{log}=> ris(X in A, 0 =< X) = {3,M}.

A = {3,M / _N1}

Constraint: ris(X in _N1, [], 0 =< X, X, true) = {},
            0 =< M

Another solution?  (y/n)
no
```

- `_N1` → new existential variable
- **`Constraint`** → conjunction of constraints, always satisfiable
- solution → substitute `_N1` by `{}` and `M` by `0`

# groundsol

generates ground solutions

solutions without variables

## groundsol

generates ground solutions

solutions without variables

```
{log}=> groundsol.
{log}=> ris(X in A, 0 =< X) = {3,M}.
```

## groundsol

generates ground solutions

solutions without variables

```
{log}=> groundsol.
{log}=> ris(X in A, 0 =< X) = {3,M}.

A = {3,0},  M = 0

Another solution?  (y/n)
no
```

# groundsol

generates ground solutions solutions without variables

```
{log}=> groundsol.
{log}=> ris(X in A, 0 =< X) = {3,M}.

A = {3,0},  M = 0

Another solution?  (y/n)
no
```

can't be used in proofs!!!

## self test

- write a ris such that for all the elements of a binary relation *R* the sum of the first and second component is equal to zero
- test your definition with equality and set membership
- prove your definition is correct by asserting that exists a pair in the ris such the sum of its component isn't zero

## recall

**`ris`**(expr **`in`** set, [vars], predicate, elements, functional predicate)

# set operators

(set) operators are given as constraints (predicates)

`un(A,B,C)` $\rightarrow$ $A \cup B = C$

`nun(A,B,C)` $\rightarrow$ $A \cup B \neq C$

negative constraints $\rightarrow$ add prefix `n` to the positive name

`size(A,N)` $\rightarrow$ $|A| = N$

```
{log}=> un({X},B,{1}).
```

```
{log}=> un({X},B,{1}).

X = 1,  B = {}

X = 1,  B = {1}
```

```
{log}=> un({X},B,{1}).

X = 1,  B = {}

X = 1,  B = {1}

{log}=> nun(A,B,C).
```

```
{log}=> un({X},B,{1}).

X = 1,  B = {}

X = 1,  B = {1}

{log}=> nun(A,B,C).

C = {_N2/_N1}
Constraint: _N2 nin A, _N2 nin B

A = {_N2/_N1}
Constraint: _N2 nin C

B = {_N2/_N1}
Constraint: _N2 nin C
```

# relational operators

`comp(R,S,T)` $\rightarrow$ $R \mathbin{;} S = T$

`dres(A,R,S)` $\rightarrow$ $A \lhd R = S$

`dares(A,R,S)` $\rightarrow$ $A \unlhd R = S$

`oplus(R,S,T)` $\rightarrow$ $(\mathrm{dom}\, S \unlhd R) \cup S = T$ B $\unlhd\!\!\!+$ Z $\oplus$

$(A \lhd R) \cup (A \ntriangleleft R) = R$

$(A \lhd R) \cup (A \ntriangleleft R) = R$

```
{log}=> neg(
          dres(A,R,R1) & dares(A,R,R2) implies un(R1,R2,R)
        ).
```

$(A \lhd R) \cup (A \ntriangleleft R) = R$

```
{log}=> neg(
          dres(A,R,R1) & dares(A,R,R2) implies un(R1,R2,R)
        ).

no
```

$(A \lhd R) \cup (A \ntriangleleft R) = R$

```
{log}=> neg(
          dres(A,R,R1) & dares(A,R,R2) implies un(R1,R2,R)
        ).

no

{log}=> dres(A,R,R1) & dares(A,R,R2) & nun(R1,R2,R).

no
```

actually, the first formula is rewritten into the second one

```
{log}=> dom({[a,1],[a,2],[b,3]},D).
```

```
{log}=> dom({[a,1],[a,2],[b,3]},D).

D = {a,b}
```

```
{log}=> dom({[a,1],[a,2],[b,3]},D).

D = {a,b}

{log}=> dom({[a,1],[A,2],[b,3]},D).
```

```
{log}=> dom({[a,1],[a,2],[b,3]},D).

D = {a,b}

{log}=> dom({[a,1],[A,2],[b,3]},D).

D = {a,A,b}
```

```
{log}=> dom({[a,1],[a,2],[b,3]},D).

D = {a,b}

{log}=> dom({[a,1],[A,2],[b,3]},D).

D = {a,A,b}

{log}=> dom(R,{1,2,3}).
```

```
{log}=> dom({[a,1],[a,2],[b,3]},D).

D = {a,b}

{log}=> dom({[a,1],[A,2],[b,3]},D).

D = {a,A,b}

{log}=> dom(R,{1,2,3}).

R = {[1,_N4],[2,_N3],[3,_N2]/_N1}
Constraint:
  [2,_N3] nin _N7, [1,_N4] nin _N8, [3,_N2] nin _N6,
  comp({[1,1]},_N8,_N8), comp({[2,2]},_N7,_N7),
  comp({[3,3]},_N6,_N6),
  un(_N7,_N6,_N5), un(_N8,_N5,_N1)
```

$R = \{(1, n_4), (2, n_3), (3, n_2)\} \cup (\{1\} \times R_8) \cup (\{2\} \times R_7) \cup (\{3\} \times R_6)$

```
{log}=> groundsol.
{log}=> dom(R,{1,2,3}).
```

```
{log}=> groundsol.
{log}=> dom(R,{1,2,3}).

R = {[1,n0],[2,n1],[3,n2]}
```

when groundsol is active basic constants are of the form
`n`$\langle$*number*$\rangle$

## self test

try the following:

- `dom(R,1,2,3) & pfun(R)`
- `ran(R,1,2,3) & pfun(R)`
- `ran(R,1,2,3) & inv(R,S) & pfun(S)`
- `ran(R,1,2,3) & pfun(R) & inv(R,S) & pfun(S)`

# function application

`apply(F,X,Y)` $\rightarrow$ `[X,Y] in F & pfun(F)`

**`pfun(F)`** $\rightarrow$ `F` is a function

`applyTo(F,X,Y)` $\rightarrow$ `comp({[X,X]},F,{[X,Y]})`

`F` is *locally* a function on `X`

```
{log}=> applyTo({[1,a],[2,a],[2,b]},1,Y).
```

```
{log}=> applyTo({[1,a],[2,a],[2,b]},1,Y).

Y = a
```

```
{log}=> applyTo({[1,a],[2,a],[2,b]},1,Y).

Y = a

{log}=> applyTo({[1,a],[2,a],[2,b]},2,Y).
```

```
{log}=> applyTo({[1,a],[2,a],[2,b]},1,Y).

Y = a

{log}=> applyTo({[1,a],[2,a],[2,b]},2,Y).

no
```

```
{log}=> applyTo({[1,a],[2,a],[2,b]},1,Y).

Y = a

{log}=> applyTo({[1,a],[2,a],[2,b]},2,Y).

no

{log}=> apply({[1,a],[2,a],[2,b]},1,Y).
```

```
{log}=> applyTo({[1,a],[2,a],[2,b]},1,Y).

Y = a

{log}=> applyTo({[1,a],[2,a],[2,b]},2,Y).

no

{log}=> apply({[1,a],[2,a],[2,b]},1,Y).

no
```

```
{log}=> applyTo({[1,a],[2,a],[2,b]},1,Y).

Y = a

{log}=> applyTo({[1,a],[2,a],[2,b]},2,Y).

no

{log}=> apply({[1,a],[2,a],[2,b]},1,Y).

no

{log}=> applyTo({[1,a],[2,a],[2,b]},X,Y).
```

```
{log}=> applyTo({[1,a],[2,a],[2,b]},1,Y).

Y = a

{log}=> applyTo({[1,a],[2,a],[2,b]},2,Y).

no

{log}=> apply({[1,a],[2,a],[2,b]},1,Y).

no

{log}=> applyTo({[1,a],[2,a],[2,b]},X,Y).

X = 1,  Y = a
```

```
{log}=> applyTo({[1,a],[2,a],[Q,b]},X,Y).
```

```
{log}=> applyTo({[1,a],[2,a],[Q,b]},X,Y).

X = 1,
Y = a
Constraint: Q neq 1

X = 2,
Y = a
Constraint: Q neq 2

X = Q,
Y = b
Constraint: Q neq 1, Q neq 2
```

# foplus

`oplus(R,S,T)` $\rightarrow$ $(\operatorname{dom} S \lhd R) \cup S = T$ B $\lhd\!\!\!+$ Z $\oplus$

# foplus

`oplus(R,S,T)` $\quad\rightarrow\quad (\operatorname{dom} S \lhd R) \cup S = T$ $\quad\overset{\text{B}}{\lhd\!\!+}\ \overset{\text{Z}}{\oplus}$

`oplus` makes {*log*} to incur in lengthy computations

`oplus` is mostly used as

```
oplus(F,{[X,Y]},G)
```

when `F` is a function

{*log*} provides `foplus` to compute `oplus` in those cases

```
foplus(F,X,Y,G) :-
    F = {[X,Z] / H} & [X,Z] nin H &
    comp({[X,X]},H,{}) & G = {[X,Y] / H}
  or
    comp({[X,X]},F,{}) & G = {[X,Y] / F}.
```

`F` is locally a function on `X`

```
?- time(
     setlog(
       oplus({[1,a],[2,b],[3,c],[4,d],[5,e],[6,f]},{[1,3]},G)
     )
   ).
% 136,231 inferences, 0.014 CPU in 0.014 seconds
```

```
?- time(
     setlog(
       foplus({[1,a],[2,b],[3,c],[4,d],[5,e],[6,f]},1,3,G)
     )
   ).
% 2,615 inferences, 0.001 CPU in 0.001 seconds
```

```
{log}=> foplus({[1,a],[2,b],[3,c]},1,3,G).
```

```
{log}=> foplus({[1,a],[2,b],[3,c]},1,3,G).

G = {[1,3],[2,b],[3,c]}
```

```
{log}=> foplus({[1,a],[2,b],[3,c]},1,3,G).

G = {[1,3],[2,b],[3,c]}

{log}=> oplus({[1,a],[2,b],[3,c]},{[1,3]},G).
```

```
{log}=> foplus({[1,a],[2,b],[3,c]},1,3,G).

G = {[1,3],[2,b],[3,c]}

{log}=> oplus({[1,a],[2,b],[3,c]},{[1,3]},G).

G = {[2,b],[3,c],[1,3]}
```

```
{log}=> foplus({[1,a],[2,b],[3,c]},1,3,G).

G = {[1,3],[2,b],[3,c]}

{log}=> oplus({[1,a],[2,b],[3,c]},{[1,3]},G).

G = {[2,b],[3,c],[1,3]}

{log}=> foplus({[1,a],[1,b],[3,c]},1,3,G).
```

```
{log}=> foplus({[1,a],[2,b],[3,c]},1,3,G).

G = {[1,3],[2,b],[3,c]}

{log}=> oplus({[1,a],[2,b],[3,c]},{[1,3]},G).

G = {[2,b],[3,c],[1,3]}

{log}=> foplus({[1,a],[1,b],[3,c]},1,3,G).

no
```

```
{log}=> foplus({[1,a],[2,b],[3,c]},1,3,G).

G = {[1,3],[2,b],[3,c]}

{log}=> oplus({[1,a],[2,b],[3,c]},{[1,3]},G).

G = {[2,b],[3,c],[1,3]}

{log}=> foplus({[1,a],[1,b],[3,c]},1,3,G).

no

{log}=> oplus({[1,a],[1,b],[3,c]},{[1,3]},G).
```

```
{log}=> foplus({[1,a],[2,b],[3,c]},1,3,G).

G = {[1,3],[2,b],[3,c]}

{log}=> oplus({[1,a],[2,b],[3,c]},{[1,3]},G).

G = {[2,b],[3,c],[1,3]}

{log}=> foplus({[1,a],[1,b],[3,c]},1,3,G).

no

{log}=> oplus({[1,a],[1,b],[3,c]},{[1,3]},G).

G = {[3,c],[1,3]}
```

```
{log}=> foplus({[1,a],[2,b],[3,c]},1,3,G).

G = {[1,3],[2,b],[3,c]}

{log}=> oplus({[1,a],[2,b],[3,c]},{[1,3]},G).

G = {[2,b],[3,c],[1,3]}

{log}=> foplus({[1,a],[1,b],[3,c]},1,3,G).

no

{log}=> oplus({[1,a],[1,b],[3,c]},{[1,3]},G).

G = {[3,c],[1,3]}

{log}=> foplus({[1,a],[1,b],[3,c]},3,3,G).
```

```
{log}=> foplus({[1,a],[2,b],[3,c]},1,3,G).

G = {[1,3],[2,b],[3,c]}

{log}=> oplus({[1,a],[2,b],[3,c]},{[1,3]},G).

G = {[2,b],[3,c],[1,3]}

{log}=> foplus({[1,a],[1,b],[3,c]},1,3,G).

no

{log}=> oplus({[1,a],[1,b],[3,c]},{[1,3]},G).

G = {[3,c],[1,3]}

{log}=> foplus({[1,a],[1,b],[3,c]},3,3,G).

G = {[3,3],[1,a],[1,b]}
```

# even `foplus` can be slow

in that case, if `F` is a function, use set unification

don't even use `applyTo`

instead of

```
applyTo(F,X,Y) & Y >= 0 & foplus(F,X,W,G)
```

use

```
F = {[X,Y]/H} & [X,Y] nin H & Y >= 0 & G = {[X,W]/H}
```

## self test

- prove that the result of `foplus` is a function
- prove that composition distributes over union

## recall

`foplus` assumes that `F` is a function

`comp(R,S,T)` $\rightarrow T = R \fatsemi S$

`un(A,B,C)` $\rightarrow C = A \cup B$

# why preferring foplus over oplus?

$\{log\}$ **computes** on its constraints

in a sense, constraints are like subroutines

it's different from interactive provers such as Coq

when proving unsatisfiability, lighter constraints will tend to increase the number of automatic proofs

**specify to leverage automation**

# $p \wedge \neg p$

in {*log*}, $p \wedge \neg p$ is proved to be unsatisfiable

{*log*} doesn't rewrite $p \wedge \neg p$, as a block, into *false*

```
?- time(setlog(oplus(R,S,T) & noplus(R,S,T))).
% 37,311 inferences, 0.006 seconds

?- time(setlog(oplus({X/R},S,T) & noplus({X/R},S,T))).
% 697,060 inferences, 0.060 seconds

?- time(setlog(oplus({X,Y/R},S,T) & noplus({X,Y/R},S,T))).
% 161,451,095 inferences, in 11.806 seconds
```

# execution options

{*log*} can be configured with execution options

they sensibly alter the solving algorithm

```
?- time(setlog(oplus({X,Y/R},S,T) & noplus({X,Y/R},S,T),[oplus_fe])).
% 842,753 inferences, 0.082 CPU in 0.082 seconds
```

they're particularly useful when proving unsatisfiability

# execution options — concurrency

```
{log}=> p_t_solve(G).
?- setlog(G,tryp(prover_all))
```

solve *G* by running *G* in several different threads, each configured with a different set of execution options

all possible combinations are attempted at once

the fastest one terminates the whole computation

# restricted universal quantifiers (ruq)

$$\forall x \in A : \phi(x) \quad \rightarrow \quad \forall x (x \in A \implies \phi(x))$$

$$\forall x \in A : \phi(x) \quad \rightarrow \quad \texttt{foreach(X in A, } \phi\texttt{(X))}$$

# restricted universal quantifiers (ruq)

```
foreach([X,Y] in A, φ)
```

```
foreach([X in A,[Y,Z] in B], φ)   →
   foreach(X in A, foreach([Y,Z] in B, φ))
```

```
foreach([X in A,Y in X], φ)
```

## self test

- state set union in terms of ruq
- state oplus in terms of ruq
- test your predicates

# cardinality

`size(set,card)` $\rightarrow$ $|set| = card$

`set` can't be cartesian product nor ris

`card` can only be a variable or a numeric constant

`card` can be part of LIA constraints

# cardinality

$$A \cap B = \emptyset \implies |A \cup B| = |A| + |B|$$

# cardinality

$$A \cap B = \emptyset \implies |A \cup B| = |A| + |B|$$

```
{log}=> neg(
          un(A,B,C) & size(C,K) & size(A,M) & size(B,N) &
          disj(A,B) implies K is M + N
        ).
```

`a = b + c` $\rightarrow$ sum is uninterpreted

`a is b + c` $\rightarrow$ sum is interpreted

Prolog stuff

# cardinality

$$A \cap B = \emptyset \implies |A \cup B| = |A| + |B|$$

```
{log}=> neg(
          un(A,B,C) & size(C,K) & size(A,M) & size(B,N) &
          disj(A,B) implies K is M + N
        ).

no
```

`a = b + c` $\rightarrow$ sum is uninterpreted

`a is b + c` $\rightarrow$ sum is interpreted

Prolog stuff

```
{log}=> un(A,B,C) & disj(A,B) & size(A,N) & size(B,M) &
        M is 2*N + 3.
```

```
{log}=> un(A,B,C) & disj(A,B) & size(A,N) & size(B,M) &
        M is 2*N + 3.

true                         returns same formula ==> satisfiable
Constraint: un(A,B,C), disj(A,B), size(A,N), N >= 0,
            size(B,M), M >= 0, M is 2*N + 3
```

```
{log}=> un(A,B,C) & disj(A,B) & size(A,N) & size(B,M) &
        M is 2*N + 3.

true                          returns same formula ==> satisfiable
Constraint: un(A,B,C), disj(A,B), size(A,N), N >= 0,
            size(B,M), M >= 0, M is 2*N + 3

{log}=> fix_size.             computes minimum solution; can't be used in proofs
```

```
{log}=> un(A,B,C) & disj(A,B) & size(A,N) & size(B,M) &
        M is 2*N + 3.

true                              returns same formula ==> satisfiable
Constraint: un(A,B,C), disj(A,B), size(A,N), N >= 0,
            size(B,M), M >= 0, M is 2*N + 3

{log}=> fix_size.                 computes minimum solution; can't be used in proofs

{log}=> un(A,B,C) & disj(A,B) & size(A,N) & size(B,M) &
        M is 2*N + 3.

A = {},  B = {_N3,_N2,_N1},  C = {_N3,_N2,_N1},
N = 0,  M = 3
Constraint: _N3 neq _N2, _N3 neq _N1, _N2 neq _N1
```

```
{log}=> groundsol.
```

can't be used in proofs

```
{log}=> groundsol.
```

can't be used in proofs

```
{log}=> un(A,B,C) & disj(A,B) & size(A,N) & size(B,M) &
        M is 2*N + 3.
```

```
{log}=> groundsol.
```

can't be used in proofs

```
{log}=> un(A,B,C) & disj(A,B) & size(A,N) & size(B,M) &
        M is 2*N + 3.

A = {},
B = {n0,n1,n2},
C = {n0,n1,n2},
N = 0,
M = 3
```

## self test

- two agents must process all the jobs in a pool in such a way that each of them must process a number of jobs whose difference can't be more than one
- test your specification

## recall

use `is` to force the evaluation of integer constraints

# integer intervals

`int(m,n)` $\rightarrow$ $[m, n] \cap \mathbb{Z}$

`m` and `n` can only be variables or integer constants

`m` and `n` can be part of LIA constraints

only some constraints support intervals

```
{log}=> {2,4,7 / A} = int(M,N).
```

```
{log}=> {2,4,7 / A} = int(M,N).

true
Constraint:
  subset(A,int(M,N)), size(A,_N4),
  2 nin A, 4 nin A, 7 nin A,
  M =< 2, 2 =< N, M =< 4, 4 =< N, M =< 7, 7 =< N,
  _N4 >= 0, _N3 >= 1, _N2 >= 1, _N1 >= 1,
  _N4 is _N3-1, _N3 is _N2-1, _N2 is _N1-1, _N1 is N-M+1

........................
```

```
2,4,7 in int(M,N) \ A
_N4 = N - M - 2
|int(M,N)| = N - M + 1
==> |A| = |int(M,N)| - 3 & subset(A,int(M,N))
```

```
{log}=> fix_size.

{log}=> {2,4,7 / A} = int(M,N).
```

```
{log}=> fix_size.

{log}=> {2,4,7 / A} = int(M,N).

A = {3,5,6},
M = 2,
N = 7

A = {7,3,5,6},
M = 2,
N = 7

........................
```

## self test

- write the definition of maximum of a set using intervals (without quantifiers)
- use intervals to write a predicate that given $x \in S \subset \mathbb{Z}$ finds, if any, $y \in S$ such that $x < y$ and there's no other element in $S$ between $x$ and $y$
  that is, the predicate finds the successor of $x$ in $S$
- test your predicates
- can successor be used to find the predecessor?

## recall

in `int(m,n)` both limits can only be constants or variables

# arrays

arrays modeled as sets of ordered pairs

```
arr(A,N) :- 0 < N & pfun(A) & dom(A,int(1,N)).
```

`size` can't be used on arrays or anything related

`arr(A,N) & ran(A,R) & size(R,M)` $\rightarrow$ don't!!

```
{log}=> arr({[1,a],[2,X]},N).

N = 2
```

```
{log}=> arr({[1,a],[2,X]},N).

N = 2

{log}=> arr({[1,a],[5,X]},N).

no
```

```
{log}=> arr({[1,a],[2,X]},N).

N = 2

{log}=> arr({[1,a],[5,X]},N).

no

{log}=> arr({[1,a],[I,X]},N).

I = 2,  N = 2

I = 1,  X = a,  N = 1
```

```
{log}=> arr({[1,a],[2,X]},N).

N = 2

{log}=> arr({[1,a],[5,X]},N).

no

{log}=> arr({[1,a],[I,X]},N).

I = 2,  N = 2

I = 1,  X = a,  N = 1

{log}=> arr(A,N) & [I,X] in A & [I,Y] in A & X neq Y.

no
```

```
{log}=> arr({[1,a],[2,X]},N).

N = 2

{log}=> arr({[1,a],[5,X]},N).

no

{log}=> arr({[1,a],[I,X]},N).

I = 2,  N = 2

I = 1,  X = a,  N = 1

{log}=> arr(A,N) & [I,X] in A & [I,Y] in A & X neq Y.

no

{log}=> neg(arr(A,N) & arr(A,M) implies N = M).

no
```

# sum of an array

assuming `arr(A,N)`, sum the first `K` elements of `A`

```
sum(A,N,K,S) :-
  arr(T,K) &                                    T: program trace
  applyTo(A,1,X) &
  T = {[1,X],[K,S] / U} & [1,X] nin U & [K,S] nin U &
  foreach(I in int(2,K),
    let([J,Y,Z,Si],
         applyTo(A,I,Y) & J is I - 1 & applyTo(T,J,Z)
           Si is Y + Z,
        [I,Si] in T,                            T(i) = T(i-1) + A(i)
    )
  ).
```

{*log*} tends to produce *repeated solutions*

```
{log}=> {X/R} = {Y/S} & R = {a} & un({a},{a},S).

R = {a},  Y = X,  S = {a}
X = a,  R = {a},  Y = a,  S = {a}
X = a,  R = {a},  Y = a,  S = {a}
... two more identical solutions ...
```

set unification can’t foresee the values variables will take

this is in general unavoidable

# negation

negation in $\{log\}$ is provided by:

- the `neg` predicate, and
- negative constraints (those beginning with `n`, e.g. `nun`)

`neg` not always works well due to existential variables

negation is an issue in logic programming

in those cases the negated formula would introduce an (unrestricted) universal quantification

$\{log\}$ can't deal with those formulas compute $A \cap (B \cup C)$

```
p(A,B,C,R) :- un(B,C,U) & inters(A,U,R).
```

`U` existential variable

$\{log\}$ can't compute `neg(p)`

# the `let/3` predicate

```
p(A,B,C,R) :- let([U], un(B,C,U), inters(A,U,R)).
```

now $\{log\}$ can compute **`neg(p)`**

`neg(p)` $\rightarrow$ `un(B,C,U) & ninters(A,U,R)`

# the `let/3` predicate

`let` helps in avoiding existential variables

`let` helps with negation

`let(`[vars], functional predicate, predicate`)`

$p(x, y)$ functional predicate iff $p(x) = y$ — $y$ is the 'result' of $p$

`un`, `applyTo`, `dom`, etc. are functional predicates

```
p(S) :- X in S & X >= 0.
```

`X` existential variable

{*log*} can't compute `neg(p)`

`let` can't be used

no functional predicate

```
p(S) :- exists(X in S, X >= 0).
```

`X` bound variable

now {*log*} can compute `neg(p)`

`neg(p)` $\rightarrow$ `foreach(X in S, X < 0)`

# restricted existential quantifiers (req)

$\exists x \in A : \phi(x) \quad \rightarrow \quad \exists x (x \in A \land \phi(x))$

$\exists x \in A : \phi(x) \quad \rightarrow \quad$ `exists(X in A,` $\phi$`(X))`

`exists([X,Y] in A,` $\phi$`)`

`exists([X in A,[Y,Z] in B],` $\phi$`)`

`foreach(X in A,exists(Y in B,` $\phi$`))`

`s(A) :- A = {Y}.` *A* is a singleton set

`Y` existential variable

{*log*} can't compute `neg(s)`

can't use `let` and req

```
s(A) :- size(A,1).
```

```
s(A) :- A neq {} & foreach([X in A, Y in A], X = Y).
```

now {*log*} can compute **neg(s)**

**neg(s)** $\rightarrow$ `nsize(A,1)` $\rightarrow$ `size(A,N) & (N < 1 or N > 1)`

**neg(s)** $\rightarrow$ `exists([X in A, Y in A], X neq Y).`

# negation — recap

use `neg` with care

check for existential variables

use `let` and req whenever possible

express the formula in some other way

otherwise, manually compute the negation

$$\forall x (x \in \operatorname{dom} f \implies f(x) + 1 \in S)$$

```
dom(F,D) &
foreach(X in D,applyTo(F,X,Y) & Y + 1 in S)
```

`Y` existential variable inside ruq, not allowed

`Y + 1 in S` $\rightarrow$ `5 + 1 in {6}` is false!

# ruq and `let`

```
dom(F,D) &
foreach(X in D,applyTo(F,X,Y) & Z is Y + 1 & Z in S)
```

`Y`, `Z` existential variables inside ruq

```
foreach([X,Y] in F, Z is Y + 1 & Z in S)
```

`Z` existential variable inside ruq

```
foreach([X,Y] in F, let([Z], Z is Y + 1, Z in S))
```

## self test

- state that the sum of the components of any pair in the binary relation `R` is equal to zero
- state that a function is monotonic
- run and analyze `neg(comp({[X,X]},F,{[X,Y]}))`
- write `neg(comp({[X,X]},F,{[X,Y]}))` in a simpler way
- test your predicates

# types in {*log*}

coming from Prolog, {*log*} is an untyped language

recently, an optional type system has been added

the type system is similar to B's or Z's

in typechecking mode all variables and predicates must be declared to be of some type

```
dec_p_type(smin(set(int),int)).
smin(S,M) :-
    M in S & foreach(X in S, M =< X & dec(X,int)).
```

```
dec_p_type(smin(set(int),int)).
smin(S,M) :-
    M in S & foreach(X in S, M =< X & dec(X,int)).

{log}=> type_check.
```

```
dec_p_type(smin(set(int),int)).
smin(S,M) :-
    M in S & foreach(X in S, M =< X & dec(X,int)).

{log}=> type_check.

{log}=> smin({3,6,1,8},M).

type error: variable M has no type declaration
```

```
dec_p_type(smin(set(int),int)).
smin(S,M) :-
    M in S & foreach(X in S, M =< X & dec(X,int)).

{log}=> type_check.

{log}=> smin({3,6,1,8},M).

type error: variable M has no type declaration

{log}=> smin({3,6,1,8},M) & dec(M,int).

M = 1
```

## self test

- run `ncomp(R,S,T)`; analyze its solutions paying attention to the last three
- activate the type checker, and run and analyze
  `ncomp(R,S,T) & dec([R,S,T],rel(t,t))`
  what happened to the last three cases? why?

# specifying state machines in {*log*}

# structure

parameters $\rightarrow$ `parameters([A,B])`

state variables $\rightarrow$ `variables([X,Y,Z])`

axioms $\rightarrow$ `axiom(name)`

invariants $\rightarrow$ `invariant(name)`

initial condition $\rightarrow$ `initial(name)`

operations $\rightarrow$ `operation(name)`

# state variables — after state

$x, x' \quad \rightarrow \quad$ `X, X_`

$$\left.\begin{array}{ll} \text{Z:} & A' = A \cup \{x\} \\ \\ \text{B:} & A := A \cup \{x\} \end{array}\right\}$$ `un(A,{X},A_)` or `A_ = {X / A}`

# variable declaration

```
variables([Usr,Addr]).
```

- `Usr` $\rightarrow$ set of users of the system
- `Addr` $\rightarrow$ function holding users' addresses

# invariants — separate declarations

```
invariant(inv1).

dec_p_type(inv1(rel(usr,addr))).
```

optional declaration

```
inv1(Addr) :- pfun(Addr).
```

# invariants — separate declarations

```
invariant(inv1).
dec_p_type(inv1(rel(usr,addr))).        optional declaration
inv1(Addr) :- pfun(Addr).

invariant(inv2).
dec_p_type(inv2(set(usr),rel(usr,addr))).
inv2(Usr,Addr) :- dom(Addr,Usr).
```

use same names for state variables

write type declarations right before clause

# invariants — single declaration

```
invariant(inv3).

inv3(Usr,Addr) :- pfun(Addr) & dom(Addr,Usr).
```

each strategy produces different proof obligations

# initial conditions

```
initial(init).

init(Usr,Addr) :- Usr = {} & Addr = {}.
```

# operations

```
operation(addUser).
addUser(Usr,Addr,U,A,Usr_,Addr_) :-
  U nin Usr &
  Usr_ = {U / Usr} &
  Addr_ = {[U,A] / Addr}.
```

primed state variables indicate the new state

# operations

```
operation(changeAddr).
changeAddr(Usr,Addr,U,Na,Addr_) :-
  U in Usr &
  oplus(Addr,{[U,Na]},Addr_).
```

`Usr_` doesn't appear in the head → unchanged variable

# operations

unchanged variables can be made more explicit

```
operation(changePass).

changeAddr( Usr ,Addr,U,Na, Usr ,Addr_) :- ........
```

# operations — state queries

get a user's address

```
operation(usrAddr).
usrAddr(Usr,Addr,U,A) :-
  U in Usr & applyTo(Addr,U,A).
```

`U` $\rightarrow$ input parameter

`A` $\rightarrow$ output parameter

distinction between inputs and outputs is conventional

## operations — more than one clause

in `changeAddr` show an error message when $\mathtt{U} \notin \mathtt{Usr}$

# operations — more than one clause

in `changeAddr` show an error message when $\mathtt{U} \notin \mathtt{Usr}$

```
changeAddrOk(Usr,Addr,U,Na,Addr_,M) :-
  U in Usr & oplus(Addr,{[U,Na]},Addr_) & M = ok.
```

# operations — more than one clause

in `changeAddr` show an error message when $\mathtt{U} \notin \mathtt{Usr}$

```
changeAddrOk(Usr,Addr,U,Na,Addr_,M) :-
  U in Usr & oplus(Addr,{[U,Na]},Addr_) & M = ok.

notAUsr(Usr,U,M) :- U nin Usr & M = error.
```

# operations — more than one clause

in `changeAddr` show an error message when `U` $\notin$ `Usr`

```
changeAddrOk(Usr,Addr,U,Na,Addr_,M) :-
  U in Usr & oplus(Addr,{[U,Na]},Addr_) & M = ok.

notAUsr(Usr,U,M) :- U nin Usr & M = error.

operation(changeAddr).
changeAddr(Usr,Addr,U,Na,Addr_,M) :-
  changeAddrOk(Usr,Addr,U,Na,Addr_,M)
  or notAUsr(Usr,U,M) & Addr_ = Addr.
```

# running state machines

# the NEXT environment

NEXT simplifies the execution of state machines

only fully instantiated executions are allowed

assuming the specification is in `usr.slog`

```
{log}=> consult('usr.slog').
{log}=> vcg('usr.slog').
{log}=> consult('usr-vc.slog').
```

```
{log}=> initial >> addUser(U:u,A:p) >> addUser(U:v,A:q).

Final result is:
Usr = {v,u},  Addr = {[v,q],[u,p]}

{log}=> initial >> addUser(U:[[u,v]],A:[[p,q]]).

Final result is:
Usr = {v,u},  Addr = {[v,q],[u,p]}
```

```
{log}=> [initial] >> addUser(U:[[u,v,u]],A:[[p,q,r]]).

Execution trace is:
Usr = {},
Addr = {}
  ----> addUser(U:u,A:p)
Usr = {u},
Addr = {[u,p]}
  ----> addUser(U:v,A:q)
Usr = {v,u},
Addr = {[v,q],[u,p]}
  ----> addUser(U:u,A:r) failed, execution aborted
```

```
init1(Usr,Addr) :- Usr = {v} & Addr = {[u,p],[v,q]}.

{log}=> [init1]:[inv1] >> addUser(U:u,A:r) >>
        changeAddr(U:u,Na:w) >> usrAddr(U:u,A).

Execution trace is:
Usr = {v},  Addr = {[u,p],[v,q]}
  ----> addUser(U:u,A:r)
Usr = {u,v},  Addr = {[u,r],[u,p],[v,q]}
  inv1 check failed
  ----> changeAddr(U:u,Na:w)
Usr = {u,v},  Addr = {[v,q],[u,w]}
  ----> usrAddr(U:u,A)
Usr = {u,v},  Addr = {[v,q],[u,w]},
A = w
```

# operations are subroutines

take each operation as a callable subroutine

(unprimed) state variables and some arguments are inputs

primed state variables and other arguments are outputs

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        usrAddr(Usr,Addr,v,A).
```

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        usrAddr(Usr,Addr,v,A).

A = q
```

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        usrAddr(Usr,Addr,v,A).

A = q

{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        inv3(Usr,Addr).
```

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        usrAddr(Usr,Addr,v,A).

A = q

{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        inv3(Usr,Addr).

yes
```

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        usrAddr(Usr,Addr,v,A).

A = q

{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        inv3(Usr,Addr).

yes

{log}=> Usr = {u,v,w} & Addr = {[u,p],[ v ,q],[w,r]} &
        addUser(Usr,Addr, v ,a,Usr_,Addr_).
```

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        usrAddr(Usr,Addr,v,A).

A = q

{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        inv3(Usr,Addr).

yes

{log}=> Usr = {u,v,w} & Addr = {[u,p],[ v ,q],[w,r]} &
        addUser(Usr,Addr, v ,a,Usr_,Addr_).

no
```

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        addUser(Usr,Addr,z,a,Usr_,Addr_).
```

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        addUser(Usr,Addr,z,a,Usr_,Addr_).

Usr_ = {z,u,v,w},  Addr_ = {[z,a],[u,p],[v,q],[w,r]}
```

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        addUser(Usr,Addr,z,a,Usr_,Addr_).

Usr_ = {z,u,v,w},  Addr_ = {[z,a],[u,p],[v,q],[w,r]}

{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        addUser(Usr,Addr,z,a, U1 , A1 ) &
        changeAddr( U1 , A1 ,v,k,Usr_,Addr_).
```

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        addUser(Usr,Addr,z,a,Usr_,Addr_).

Usr_ = {z,u,v,w},  Addr_ = {[z,a],[u,p],[v,q],[w,r]}

{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        addUser(Usr,Addr,z,a,U1,A1) &
        changeAddr(U1,A1,v,k,Usr_,Addr_).

U1 = {z,u,v,w},
A1 = {[z,a],[u,p],[v,q],[w,r]},
Usr_ = {z,u,v,w},
Addr_ = {[z,a],[u,p],[w,r],[v,k]}
```

## symbolic executions

```
{log}=> Usr = {u,v,w} & Addr = {[u,p],[v,q],[w,r]} &
        usrAddr(Usr,Addr,U,A).

U = u,  A = p

U = v,  A = q

U = w,  A = r
```

# verification of state machines

assuming the specification in in `usr.slog`

```
{log}=> consult('usr.slog').
{log}=> vcg('usr.slog').
```

generates the file `usr-vc.slog` containing verification conditions

# run the verification conditions

```
{log}=> consult('usr-vc.slog').

{log}=> check_vcs_usr.

  Checking init_sat_inv1 ... OK
  Checking init_sat_inv2 ... OK
  Checking init_sat_inv3 ... OK
  Checking addUser_is_sat ... OK
  Checking changeAddr_is_sat ... OK
  Checking usrAddr_is_sat ... OK
  Checking addUser_pi_inv1 ... ERROR
  Checking addUser_pi_inv2 ... OK
  ... all other VC's are OK ...
```

# verification conditions

the conjunction of all axioms is satisfiable

the initial state satisfies each invariant

if there's no initial state $\rightarrow$ checks that each invariant is satisfiable

each operation is satisfiable and can change the state

if it contains primed variables

each operation preservers each invariant

invariance lemma

some well-definedness conditions

# why `addUser_pi_inv1` failed?

```
inv1(Addr) :- pfun(Addr).

addUser(Usr,Addr,U,A,Usr_,Addr_) :-
  U nin Usr &
  Usr_ = {U / Usr} & Addr_ = {[U,A] / Addr}.

addUser_pi_inv1(Usr,Addr,U,A,Usr_,Addr_) :-
  neg(
    inv1(Addr) &
    addUser(Usr,Addr,U,A,Usr_,Addr_) implies
    inv1(Addr_)
  ).
```

# why `addUser_pi_inv1` failed?

use {*log*} to find the problem

# why `addUser_pi_inv1` failed?

use {*log*} to find the problem

```
{log}=> vcace(addUser_pi_inv1).
```

# why `addUser_pi_inv1` failed?

use {*log*} to find the problem

```
{log}=> vcace(addUser_pi_inv1).

Addr = {[U,_N2] / _N1},
Usr_ = {U / Usr},
Addr_ = {[U,A],[U,_N2] / _N1}
Constraint: U nin Usr, A neq _N2, ...
```

`U` is in the domain of `Addr` but it isn't in `Usr`

this violates `inv2`

```
inv2(Usr,Addr) :- dom(Addr,Usr).
```

# why addUser_pi_inv1 failed?

```
{log}=> vcgce(addUser_pi_inv1).

Addr = {[n2,n1]}
Sec = {}
Usr = {}
U = n2
A = n0
Usr_ = {n2}
Addr_ = {[n2,n0],[n2,n1]}
```

`n2` is in the domain of `Addr` but `Usr` is empty

# fix addUser_pi_inv1

add inv2 as an hypothesis

```
addUser_pi_inv1(Usr,Addr,U,A,Usr_,Addr_) :-
  neg(
    inv2(Usr,Addr) &
    inv1(Addr) &
    addUser(Usr,Addr,U,A,Usr_,Addr_) implies
    inv1(Addr_)
  ).
```

# `findh`

{*log*} provides `findh` to find missing axioms/invariants

when `check_vcs_*` is run, solutions are saved

`findh` retrieve those solutions and tries to satisfy ax/inv

when one is unsat it means that's a missing hypothesis

# findh

```
{log}=> check_vcs_usr.

  Checking init_sat_inv1 ... OK
  ...............................
  Checking addUser_is_sat ... OK
  Checking changeAddr_is_sat ... OK
  Checking usrAddr_is_sat ... OK
  Checking addUser_pi_inv1 ... ERROR
  Checking addUser_pi_inv2 ... OK
  ... all other VC's are OK ...


{log}=> findh.
Missing hypotheses for addUser_pi_inv1: [inv2]
```

## `vcace`, `vcgce` and `findh`

use `vcace` and `vcace` first

`findh` can only find missing hypotheses

verification conditions may fail for other reasons

missing precondition, invariant too strong, etc.

`findh` can be too slow

there are a few variants of `findh` that might help

# invariance lemmas

let $\mathcal{I}_1, \ldots, \mathcal{I}_n$ be all the invariants

invariance lemma: $\mathcal{I}_1 \wedge \cdots \wedge \mathcal{I}_n \wedge Op \implies \mathcal{I}'_j$ for all $j$

instead vcg generates: $\mathcal{I}_j \wedge Op \implies \mathcal{I}'_j$ for all $j$

why?

# hypothesis in automated proof

in an **automated** proof of

$$\mathcal{I}_1 \wedge \cdots \wedge \mathcal{I}_n \wedge Op \implies \mathcal{I}'_j$$

$\bigwedge_{k=1}^{n} \mathcal{I}_k$ makes the prover to go over many proof paths

this can take a very long time

an interactive proof could be the only way left

# minimize the number of hypothesis

$$\mathcal{I}_1 \wedge \cdots \wedge \mathcal{I}_n \wedge Op \implies \mathcal{I}'_j$$

most of the times only a few $\mathcal{I}_k$ are necessary

try $\quad \mathcal{I}_j \wedge Op \implies \mathcal{I}'_j \quad$ if it fails

1. use $\{log\}$ to find the right $\mathcal{I}_k$'s
2. add these $\mathcal{I}_k$'s as hypothesis
3. try again

this is (much?) easier than interactive proofs

## self test

- `oplus` slows down some proofs. solve it.
- delete `U in Usr` from `addUsr`. now the invariance lemma can't be proved. is the returned solution of any help?

{*log*} implements the Test Template Framework (TTF)

TTF applies *testing tactics* to each operation

tactics partition the VIS

testing tree

each tactic captures a testing heuristic

# test case generation

`ttf(`$atom$`)`

`applydnf(`$op(l_1, ..., l_n)$`)`

`applysp(`$cons(l_1, ..., l_n)$`)`

`writett`

`prunett`

`gentc`

`exportttt`

see section 13.6 of user's manual

# how {*log*} works

$\{log\}$ is a (highly) non-deterministic rewriting system

one atom is rewritten at each processing step

over 100 rewrite rules

$\phi \to \Phi_1 \vee \cdots \vee \Phi_n$

$$
\begin{aligned}
&\{x \sqcup A\} = \{y \sqcup B\} \longrightarrow \\
&\quad x = y \wedge A = B \\
&\quad \vee\, x = y \wedge \{x \sqcup A\} = B \\
&\quad \vee\, x = y \wedge A = \{y \sqcup B\} \\
&\quad \vee\, A = \{y \sqcup N\} \wedge \{x \sqcup N\} = B
\end{aligned}
$$

$$
\begin{aligned}
&un(\{x \sqcup C\}, A, \dot{B}) \longrightarrow \\
&\quad \{x \sqcup C\} = \{x \sqcup N_1\} \wedge x \notin N_1 \wedge B = \{x \sqcup N\} \\
&\quad \wedge (x \notin A \wedge un(N_1, A, N) \vee A = \{x \sqcup N_2\} \wedge x \notin N_2 \wedge un(N_1, N_2, N))
\end{aligned}
$$

# some rewrite rules

$$
\begin{aligned}
&oplus(R, S, T) \longrightarrow \\
&\quad dom(S, N_5) \wedge un(N_4, N_3, R) \wedge dom(N_4, N_2) \wedge dom(N_3, N_1) \\
&\quad \wedge subset(N_1, N_5) \wedge disj(N_5, N_2) \wedge un(S, N_4, T)
\end{aligned}
$$

$$
\begin{aligned}
&comp(\{(x, u) \sqcup R\}, \{(t, z) \sqcup S\}, \emptyset) \longrightarrow \\
&\quad u \neq t \\
&\quad \wedge comp(\{(x, u)\}, S, \emptyset) \wedge comp(R, \{(t, z)\}, \emptyset) \wedge comp(R, S, \emptyset)
\end{aligned}
$$

# cardinality and intervals are treated differently

rewrite rules are applied to *size* constraints

$$
\begin{aligned}
&size(\{x \sqcup A\}, m) \longrightarrow \\
&\quad m = n + 1 \\
&\quad \wedge (x \notin A \wedge size(A, n) \vee A = \{x \sqcup N\} \wedge x \notin N \wedge size(N, n))
\end{aligned}
$$

only $size(A, N)$, $A$ and $N$ variables, remain

zarba's algorithm is called in Calogero Zarba, 2002

# zarba's algorithm

turns the cardinality problem into a $(\mathbb{B}, \mathbb{Z})$ problem

- $un(A, B, C) \rightarrow (\neg C \lor B \lor A) \land (\neg A \lor C) \land (\neg B \lor C)$
- $\mathbb{Z}$ problem $\rightarrow$ linear integer problem

$\mathbb{B}$ problem $\rightarrow$ howe and king's sat solver (prolog)

$\mathbb{Z}$ problem $\rightarrow$ swi-prolog's clp(q) library

- computes the minimum of the problem
- very important for integer intervals

# integer intervals

rewrite rules are applied to remove intervals

$$\{x \sqcup A\} = [m, n] \longrightarrow$$
$$\quad subset(\{x \sqcup A\}, [m, n]) \wedge size(\{x \sqcup A\}, n - m + 1)$$

$$subset(\{x \sqcup A\}, [m, n]) \longrightarrow m \leq x \leq n \wedge subset(A, [m, n])$$

only $subset(A, [m, n])$, $A$ and $m$ or $n$ variables, remain

a cardinality problem must be solved

# integer intervals

$subset(A, [m, n])$ aren't passed to zarba

$A$ and $m$ or $n$ variables

$$\Phi \equiv \Phi_{Za} \wedge \Phi_{\subseteq[]}$$

$zarba_{min}(\Phi_{Za}) \rightarrow$ computes the minimal solution

if the minimal solution is a solution of $\Phi \rightarrow$ sat

if not, no larger solution of $\Phi_{Za}$ is a solution of $\Phi \rightarrow$ unsat

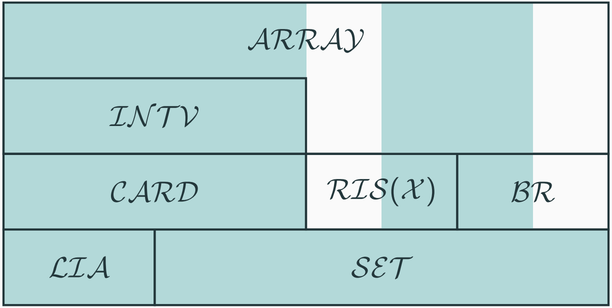
# how {*log*} computes

rewriting system + syntactic unification + linear integer solver

```
smin({12,3,Y,8},M) ⟶
M in {12,3,Y,8} & foreach(X in {12,3,Y,8}, M =< X) ⟶(4)
M in {12,3,Y,8} & M =< 12 & M =< 3 & M =< Y & M =< 8 ⟶(4)
(M = 12 & M =< 12 & M =< 3 & M =< Y & M =< 8
 or M = 3 & M =< 12 & M =< 3 & M =< Y & M =< 8
 or M = Y & M =< 12 & M =< 3 & M =< Y & M =< 8
 or M = 12 & M =< 12 & M =< 3 & M =< Y & M =< 8)
```

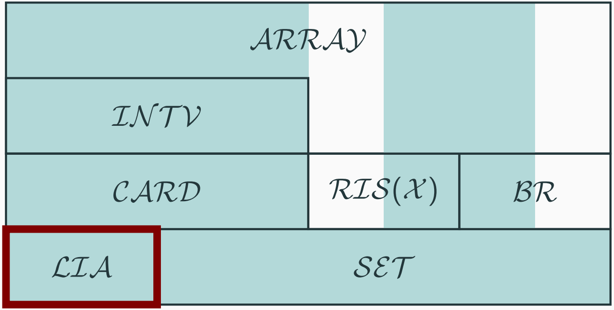
the integer solver solves each disjunct

# decision procedures implemented in {log}




a green block denotes a decidable theory

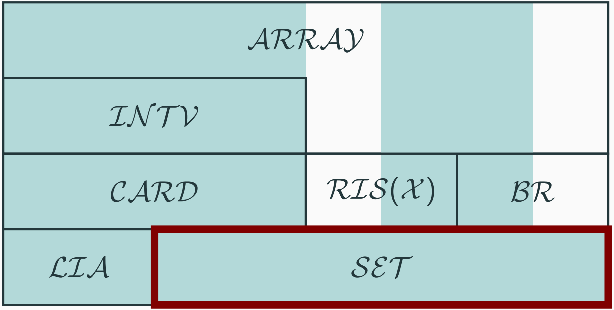
# decision procedures implemented in $\{log\}$




linear integer algebra

a green block denotes a decidable theory

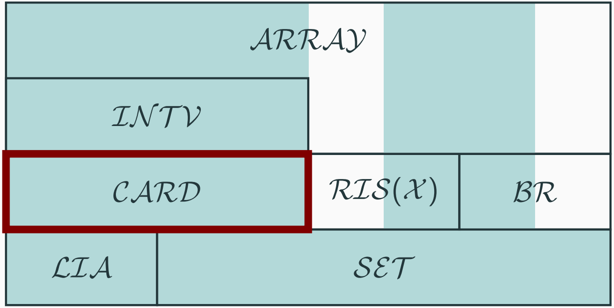
# decision procedures implemented in {log}




boolean algebra of finite sets

a green block denotes a decidable theory

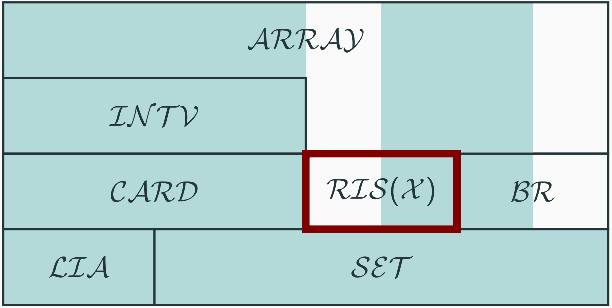
# decision procedures implemented in $\{log\}$




$\mathcal{SET}$ extended with cardinality

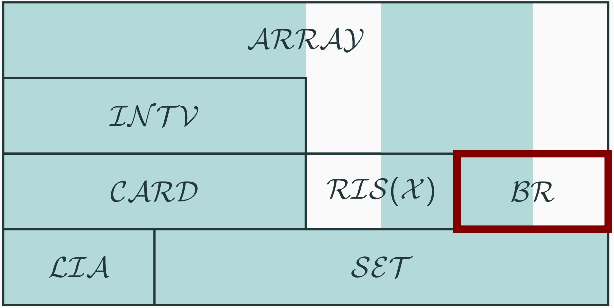
a green block denotes a decidable theory

# decision procedures implemented in $\{log\}$




$\mathcal{SET}$ extended with intensional sets (includes RUQ)

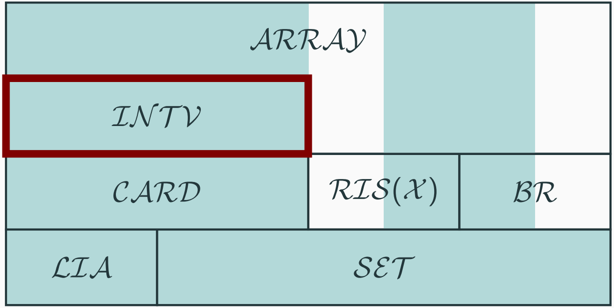
a green block denotes a decidable theory




$\mathcal{SET}$ extended with set relation algebra

a green block denotes a decidable theory

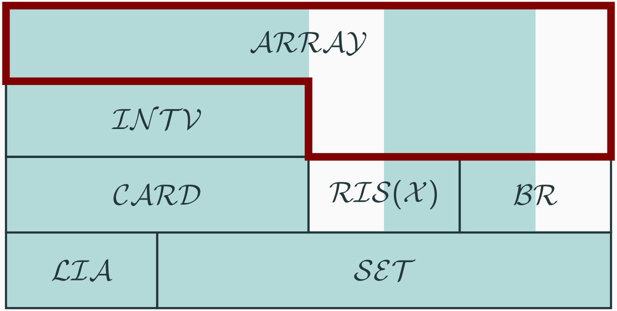
# decision procedures implemented in $\{log\}$




$\mathcal{CARD}$ extended with integer intervals

a green block denotes a decidable theory

# decision procedures implemented in $\{log\}$




combines all the theories (work in progress)

a green block denotes a decidable theory

# undecidability in set relation algebra ($\mathcal{BR}$)

$\{log\}$ can't decide the satisfiability of **most** formulas including

$comp(\mathcal{T}_1(R), \mathcal{T}_2(S), \mathcal{T}_3(R))$

$\mathcal{T}_i$ dependent term

$R, S$ variables

example: `comp({W/R},S,T) & un(R,Q,T)`

or

# undecidability in set relation algebra ($\mathcal{BR}$)

$\{log\}$ can't decide the satisfiability of **most** formulas including

$comp(\mathcal{T}_1(R), \mathcal{T}_2(S), \mathcal{T}_3(R))$

$\mathcal{T}_i$ dependent term

$R, S$ variables

example: `comp({W/R},S,T) & un(R,Q,T)`

or

$comp(\mathcal{T}_1(S), \mathcal{T}_2(R), \mathcal{T}_3(R))$

```
{log}=> comp(R,{X/S},R).

X = [_N2,_N1]
Constraint: comp(R,{[_N2,_N1]/S},R)
```

```
{log}=> comp(R,{X/S},R).

X = [_N2,_N1]
Constraint: comp(R,{[_N2,_N1]/S},R)

{log}=> comp({X/R},S,{Y/R}).
```

*returns an infinite number of solutions…*

```
{log}=> comp(R,{X/S},R).

X = [_N2,_N1]
Constraint: comp(R,{[_N2,_N1]/S},R)

{log}=> comp({X/R},S,{Y/R}).
```

*returns an infinite number of solutions…*

```
{log}=> comp({X/R},S,{Y/R}) & un(A,B,C) & nun(B,A,C).
```

*loops forever…*

```
{log}=> comp(R,{X/S},R).

X = [_N2,_N1]
Constraint: comp(R,{[_N2,_N1]/S},R)

{log}=> comp({X/R},S,{Y/R}).
```

*returns an infinite number of solutions…*

```
{log}=> comp({X/R},S,{Y/R}) & un(A,B,C) & nun(B,A,C).
```

*loops forever…*

```
{log}=> comp({X/R},S,{X/R}) & id(A,R).
```

*returns four solutions…*

# undecidability in set relation algebra ($\mathcal{BR}$)

other constraints hide the dangerous *comp* constraints

# undecidability in set relation algebra ($\mathcal{BR}$)

other constraints hide the dangerous *comp* constraints

```
{log}=>  dom({P / R},A) & ran(R,A).
```

*loops forever…*

# undecidability in set relation algebra ($\mathcal{BR}$)

other constraints hide the dangerous *comp* constraints

```
{log}=>  dom({P / R }, A ) & ran( R , A ).
```

*loops forever…*

$$
\begin{aligned}
& dom(R, A) \mathrel{\widehat{=}} \\
& \quad id(A, N_1) \land comp(N_1, R, N_2) \land R \subseteq N_2 \\
& \quad \land inv(R, N_3) \land comp(R, N_3, N_4) \land N_1 \subseteq N_4 \\
& ran(R, A) \mathrel{\widehat{=}} inv(R, N) \land dom(N, A)
\end{aligned}
$$

# undecidability of quantified formulas (ruq&req)

$$\forall x (x \in A \implies \phi(x)) \quad \rightsquigarrow \quad \forall x \in A : \phi(x) \quad \rightsquigarrow \quad \forall x \in A : \phi$$

$$\exists x (x \in A \wedge \phi(x)) \quad \rightsquigarrow \quad \exists x \in A : \phi(x) \quad \rightsquigarrow \quad \exists x \in A : \phi$$

$\phi$ is a quantifier-free formula

# undecidability of quantified formulas (ruq&req)

$\{log\}$ can't decide the satisfiability of **most** formulas including

$\forall x \in \mathcal{T}_1(A) : \exists y \in \mathcal{T}_2(A) : \phi$

$\mathcal{T}_i$ dependent term

$A$ variable

or

$(\forall x \in \mathcal{T}_1(A) : \exists y \in \mathcal{T}_2(B) : \phi) \quad \land \quad (\forall x \in \mathcal{T}_1(B) : \exists y \in \mathcal{T}_2(A) : \beta)$

# undecidability of quantified formulas (ruq&req)

$\{log\}$ can't decide the satisfiability of **most** formulas including

a req **after** a ruq sharing the same domain variable

# undecidability: set relation algebra and quantified formulas

# undecidability: set relation algebra and quantified formulas

$$T \subseteq R \mathbin{;} S \iff$$
$$\forall (x, z) \in T : \exists (a, b) \in R, (c, d) \in S : a = x \land b = c \land d = z$$

# undecidability: set relation algebra and quantified formulas

$$T \subseteq R \fatsemi S \iff$$
$$\forall (x, z) \in T : \exists (a, b) \in R, (c, d) \in S : a = x \land b = c \land d = z$$

then

$$\mathrm{R} \subseteq \mathrm{R} \fatsemi S \iff$$
$$\forall (x, z) \in \mathrm{R} : \exists (a, b) \in \mathrm{R}, (c, d) \in S : a = x \land b = c \land d = z$$

# undecidability: set relation algebra and quantified formulas

$T \subseteq R \fatsemi S \iff$
$\forall (x,z) \in T : \exists (a,b) \in R, (c,d) \in S : a = x \land b = c \land d = z$

then

$\mathrm{R} \subseteq \mathrm{R} \fatsemi S \iff$
$\forall (x,z) \in \mathrm{R} : \exists (a,b) \in \mathrm{R}, (c,d) \in S : a = x \land b = c \land d = z$

$comp(\mathcal{T}_1(R), \mathcal{T}_2(S), \mathcal{T}_3(R))$ is equivalent to having a req after a ruq sharing the same domain variable

# {*log*} in practice

we've assessed {*log*} with a few case studies and benchmarks

so far it performed well on all of them

more demanding problems would take {*log*} beyond its limits

new techniques and optimizations are being investigated

# bell-lapadula security model (blp)

first model for the confidentiality problem 1972

models system calls of a unix-like operating system

state machine described in terms of set theory and fol

two state invariants
→ security condition
→ *-property

# blp in {*log*}

10 operations

6 invariants — security condition, *-property + 4 type invariants

60 invariance lemmas

less than 2 seconds

# tokeneer project

commissioned by nsa to altran uk as a demonstration project

formal methods can be applied in an industrial setting

biometric verification of the user

common criteria eal5

2011 microsoft research verified software milestone award

# tokeneer project

altran uk → correctness by construction process

Z specification of user requirements

altran uk didn’t machine-checked the specification

# tokeneer in {*log*}

Z specification (2 kloc) → {*log*} program (2.6 kloc)

{*log*} → 523 verification conditions → 14 minutes

19 verification conditions involving security properties

Z specification (2 kloc) → {*log*} program (2.6 kloc)

{*log*} → 523 verification conditions → 14 minutes

19 verification conditions involving security properties

{*log*} program becomes a **certified prototype**

# android permissions system

coq model developed by betarte, luna and others

`https://github.com/g-deluca/android-coq-model`

udelar

36 properties were proven true of the model

minimum privilege, eavesdropping, intent spoofing, ...

coq functional specification

refinement proofs → certified prototype

# android in {*log*}

coq model (150 kb) → {*log*} program (11 kb)

33 out of 36 proofs automated in {*log*} 90%

- 800 satisfiability queries
- from 18 kloc of manual proofs to 500 loc 3%

13 minutes

# android in {*log*}

coq model (150 kb) → {*log*} program (11 kb)

33 out of 36 proofs automated in {*log*} 90%

800 satisfiability queries

from 18 kloc of manual proofs to 500 loc 3%

13 minutes

{*log*} program becomes a **certified prototype**

# landing gear system (lgs)

abz 2014 case study

event-b specification developed by mammar and laleau

4.8 kloc (213 kb) of Latex code

11 models (refinement)

285 proof obligations, 72% automatically discharged

# lgs in $\{log\}$

7.8 kloc (216 kb) of $\{log\}$ code

100% of proof obligations automatically discharged

290 seconds

$\{log\}$ program becomes a **certified prototype**

{*log*}

`www.clpset.unipr.it`

Maxi Cristiá

`cristia@cifasis-conicet.gov.ar`