{*log*}: programs as formulas (not as proofs)

maximiliano cristiá

universidad nacional de rosario

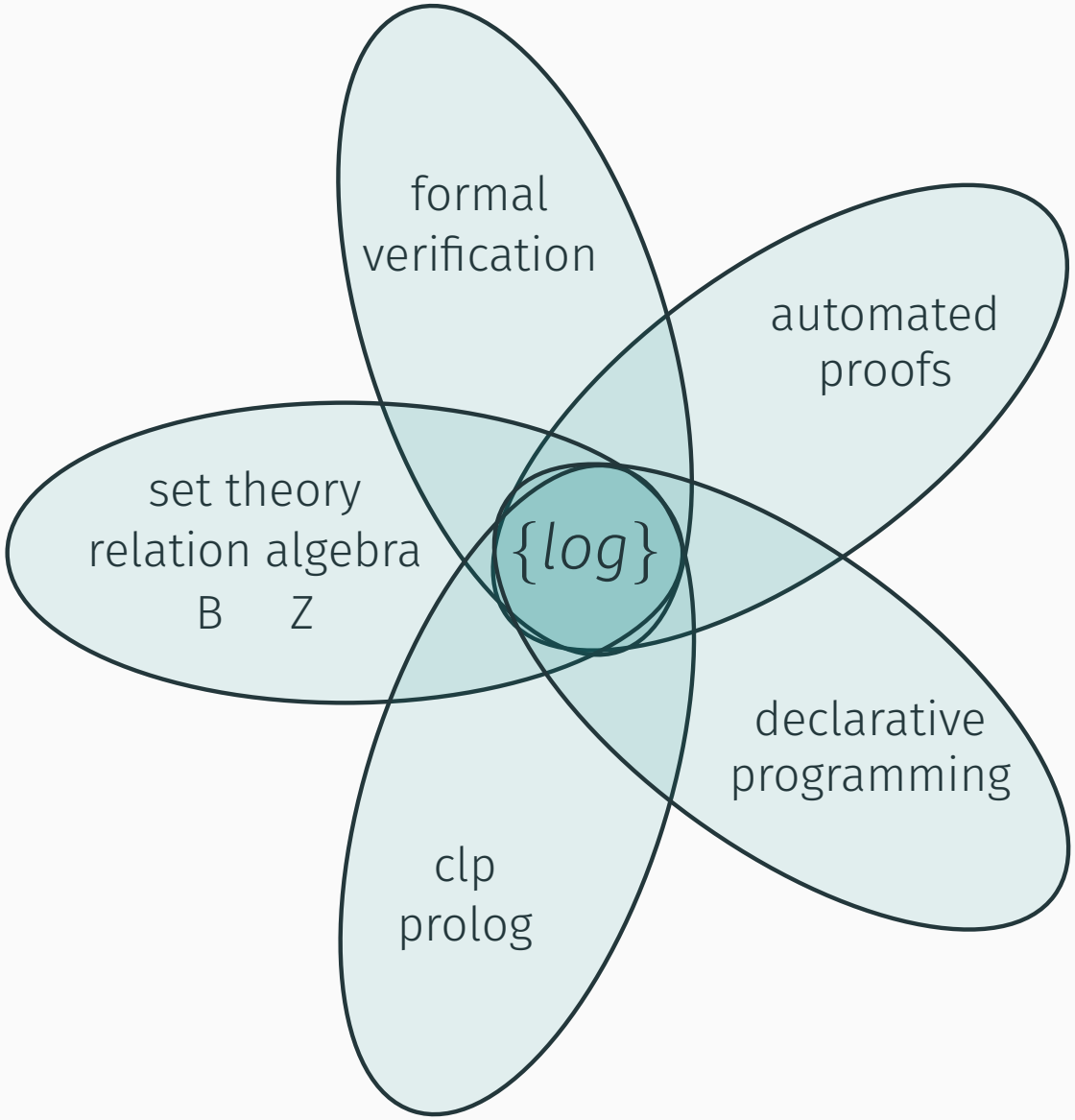

CIFASIS

argentina

joint work with **gianfranco rossi**

agh university of science and technology – kraków – march, 2022

keywords


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

programs as proofs

proofs, programs, types

proof P of theorem T can be transformed into a program of type T

Curry-Howard correspondence

(dependent) type theory

functional languages

software verification systems, e.g. Coq

programs as proofs: not that easy

it’s not easy to make a proof to get a program

it’s very rare to find programmers making proofs

programmers seldom write formal specifications

these programs are prototypes

programs as formulas

let's start from the beginning

↓

write a formal specification

can we **simulate** the specification without proving anything?

can we **automatically** prove properties of the specification?

programming, proof and counterexample generation

implemented in Prolog

language based on the theory of finite sets

- includes binary relations, relation algebra
- set unification
- constraint logic programming

developed from the ’90 by Gianfranco Rossi

- and by both of us since 2012

{*log*}: programs as formulas, formulas as programs

program to find if x is different from z and belongs to A

{*log*}: programs as formulas, formulas as programs

program to find if x is different from z and belongs to A

specification

$x \neq z \wedge x \in A$

{*log*}: programs as formulas, formulas as programs

program to find if x is different from z and belongs to A

specification

$x \neq z \wedge x \in A$

program

X neq Z & X in A

{*log*}: programs as formulas, formulas as programs

program to find if x is different from z and belongs to A

specification

$x \neq z \wedge x \in A$

program

X neq Z & X in A

why a formula?

{*log*}: programs as formulas, formulas as programs

program to find if x is different from z and belongs to A

specification

$x \neq z \wedge x \in A$

program

X neq Z & X in A

why a formula?

X neq Z & X in A $\iff$ X in A & X neq Z

{*log*}: programs as formulas, formulas as programs

program to find if x is different from z and belongs to A

specification

$x \neq z \wedge x \in A$

program

X neq Z & X in A

why a formula?

X neq Z & X in $A \iff X$ in A & X neq Z

why a program?

{*log*}: programs as formulas, formulas as programs

program to find if x is different from z and belongs to A

specification	**program**
$x \neq z \wedge x \in A$	X neq Z & X in A
why a formula?	X neq Z & X in $A \iff X$ in A & X neq Z
why a program?	it can be executed

{*log*}: programs as formulas, formulas as programs

program to find if x is different from z and belongs to A

specification

$x \neq z \wedge x \in A$

program

X neq Z & X in A

why a formula?

X neq Z & X in $A \iff X$ in A & X neq Z

why a program?

it can be executed

$X = hello, Z = 4, A = \{b, \{c\}, 4, hello, [a, 3]\} \quad \rightarrow \quad$ true

{*log*}: formulas as abstract programs

elements and sets can be variables

{*log*}: formulas as abstract programs

elements and sets can be variables

in this case {*log*} computes **abstract** solutions

solution: biding values to variables

{*log*}: formulas as abstract programs

elements and sets can be variables

in this case {*log*} computes **abstract** solutions

solution: biding values to variables

X neq *Z* & *X* in *A* & program

{*log*}: formulas as abstract programs

elements and sets can be variables

in this case {*log*} computes **abstract** solutions

solution: biding values to variables

$X \text{ neq } Z \;\&\; X \text{ in } A \;\&$ program

$Z = 4 \;\&\; A = \{b, c, 4, M, [a, 3]\}$ ~~$\;\&\; X = hello$~~ $\rightarrow$ inputs

{*log*}: formulas as abstract programs

elements and sets can be variables

in this case {*log*} computes **abstract** solutions

solution: biding values to variables

$X \text{ neq } Z \;\&\; X \text{ in } A \;\&$ program

$Z = 4 \;\&\; A = \{b, c, 4, M, [a, 3]\}$ ~~$\;\&\; X = hello$~~ $\rightarrow$ inputs

$X = b, A = \{b, \{c\}, 4, M, [a, 3]\}$

{*log*}: formulas as abstract programs

elements and sets can be variables

in this case {*log*} computes **abstract** solutions

solution: biding values to variables

$X \text{ neq } Z \;\&\; X \text{ in } A \;\&$ program

$Z = 4 \;\&\; A = \{b, c, 4, M, [a, 3]\}$ ~~$\;\&\; X = hello$~~ $\rightarrow$ inputs

$X = b, A = \{b, \{c\}, 4, M, [a, 3]\}$

$X = b, A = \{b, \{c\}, 4, [a, 3]\}, M = b$

{*log*}: formulas as abstract programs

elements and sets can be variables

in this case {*log*} computes **abstract** solutions

solution: biding values to variables

$X \text{ neq } Z \;\&\; X \text{ in } A \;\&$ program

$Z = 4 \;\&\; A = \{b, c, 4, M, [a, 3]\}$ ~~$\;\&\; X = hello$~~ $\rightarrow$ inputs

$X = b, A = \{b, \{c\}, 4, M, [a, 3]\}$

$X = b, A = \{b, \{c\}, 4, [a, 3]\}, M = b$

$A = \{X, b, \{c\}, 4, [a, 3]\}, M = X, X \text{ neq } 4$

{*log*}: formulas as abstract programs

elements and sets can be variables

in this case {*log*} computes **abstract** solutions

solution: biding values to variables

$X \text{ neq } Z \;\&\; X \text{ in } A \;\&$ program

$Z = 4 \;\&\; A = \{b, c, 4, M, [a, 3]\}$ ~~$\;\&\; X = hello$~~ $\rightarrow$ inputs

$X = b, A = \{b, \{c\}, 4, M, [a, 3]\}$

$X = b, A = \{b, \{c\}, 4, [a, 3]\}, M = b$

$A = \{X, b, \{c\}, 4, [a, 3]\}, M = X, X \text{ neq } 4$

and 11 more solutions

{*log*}: formulas as invertible programs

{*log*} doesn't make a difference between inputs and outputs

{*log*}: formulas as invertible programs

{*log*} doesn't make a difference between inputs and outputs

program adding an element to a set

{*log*}: formulas as invertible programs

{*log*} doesn't make a difference between inputs and outputs

program adding an element to a set

specification	program
$A' = A \cup \{x\}$	$\mathsf{un}(A, \{X\}, A\_)$

{*log*}: formulas as invertible programs

{*log*} doesn't make a difference between inputs and outputs

program adding an element to a set

specification	**program**
$A' = A \cup \{x\}$	$\mathsf{un}(A, \{X\}, A\_)$

from inputs to outputs

$A\_$ is free

{*log*}: formulas as invertible programs

{*log*} doesn't make a difference between inputs and outputs

program adding an element to a set

specification **program**

$A' = A \cup \{x\}$ $\quad$ $\mathsf{un}(A, \{X\}, A\_)$

from inputs to outputs $A\_$ is free

$A = \{1, 2\}$ & $X = 3$ & $\mathsf{un}(A, \{X\}, A\_) \quad \rightarrow A\_ = \{3, 1, 2\}$

{*log*}: formulas as invertible programs

{*log*} doesn't make a difference between inputs and outputs

program adding an element to a set

specification **program**

$A' = A \cup \{x\}$ $\quad$ $\mathrm{un}(A, \{X\}, A\_)$

from inputs to outputs $A\_$ is free

$A = \{1, 2\}$ & $X = 3$ & $\mathrm{un}(A, \{X\}, A\_) \quad \rightarrow A\_ = \{3, 1, 2\}$

from outputs to inputs A is free

{*log*}: formulas as invertible programs

{*log*} doesn't make a difference between inputs and outputs

program adding an element to a set

specification **program**

$A' = A \cup \{x\}$ $\quad$ $\mathrm{un}(A, \{X\}, A\_)$

from inputs to outputs $A\_$ is free

$A = \{1, 2\}$ & $X = 3$ & $\mathrm{un}(A, \{X\}, A\_) \quad \rightarrow A\_ = \{3, 1, 2\}$

from outputs to inputs A is free

$A\_ = \{1, 2\}$ & $X = 3$ & $\mathrm{un}(A, \{X\}, A\_) \quad \rightarrow$ no

{*log*}: formulas as invertible programs

{*log*} doesn't make a difference between inputs and outputs

program adding an element to a set

specification **program**

$A' = A \cup \{x\}$ $\quad \mathsf{un}(A, \{X\}, A\_)$

from inputs to outputs $A\_$ is free

$A = \{1, 2\}\ \&\ X = 3\ \&\ \mathsf{un}(A, \{X\}, A\_) \quad \rightarrow A\_ = \{3, 1, 2\}$

from outputs to inputs A is free

$A\_ = \{1, 2\}\ \&\ X = 3\ \&\ \mathsf{un}(A, \{X\}, A\_) \quad \rightarrow \mathsf{no}$

$A\_ = \{1, 2, 3\}\ \&\ X = 3\ \&\ \mathsf{un}(A, \{X\}, A\_) \rightarrow A = \{1, 2\}; A = \{1, 2, 3\}$

{*log*}: formulas as formulas

the specification $A' = A \cup \{x\}$ verifies the property

$A' = A \cup \{x\} \implies A \subseteq A'$

does the program verifies the same property?

{*log*}: formulas as formulas

the specification $A' = A \cup \{x\}$ verifies the property

$A' = A \cup \{x\} \implies A \subseteq A'$

does the program verifies the same property?

{*log*} is a satisfiability solver

to prove $p \implies q$ you have to prove $p \wedge \neg q \equiv \text{false}$

{*log*}: formulas as formulas

the specification $A' = A \cup \{x\}$ verifies the property

$A' = A \cup \{x\} \implies A \subseteq A'$

does the program verifies the same property?

{*log*} is a satisfiability solver

to prove $p \implies q$ you have to prove $p \wedge \neg q \equiv \text{false}$

$\text{un}(A, \{X\}, A\_) \;\&\; \text{nsubset}(A, A\_) \quad \rightarrow \text{no}$


formula
program
forgram

the sets of $\{log\}$

$$
\begin{array}{lr}
Set ::= & \\
\quad variable & \text{context or set}(X) \\
\quad \mid \{\} & \text{empty} \\
\quad \mid \{element \,/\, Set\} & \text{extensional} \\
\quad \mid \text{int}(Int, Int) & \text{integer interval} \\
\quad \mid \text{cp}(Set, Set) & \text{Cartesian product} \\
\quad \mid \text{ris}(termn \text{ in } Set, formula, term) & \text{intensional}
\end{array}
$$

finite, typed/untyped, unbounded, nested, hybrid

- unbounded $\rightarrow \{x \,/\, A\}$
- nested $\rightarrow \{\{1\}, \{1, \{2\}\}\}$
- hybrid $\rightarrow$ non-set elements are allowed

the operators of $\{log\}$

base or primitives

sets $\rightarrow$ $=$ in un disj

relations $\rightarrow$ pfun id inv comp

defined

sets $\rightarrow$ inters diff cmpt subset

relations $\rightarrow$ ran dom rimg dres rres dares rares apply oplus

restricted quantifiers

ruq $\rightarrow$ foreach(X in A, ϕ)

req $\rightarrow$ exists(X in A, ϕ)

set unification

set unification is the at the base of {*log*}

set unification

set unification is the at the base of $\{log\}$

$\{x \mid A\}$ is interpreted as $\{x\} \cup A$

set unification

set unification is the at the base of $\{log\}$

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

$\{[a, 1], [a, 2], [b, 1], [b, 2]\} = \{[a, X] \,/\, A\} \rightarrow$

set unification

set unification is the at the base of $\{log\}$

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

$$\{[a,1],[a,2],[b,1],[b,2]\} = \{[a,X] \,/\, A\} \rightarrow$$
$$X = 1, A = \{[a,2],[b,1],[b,2]\}$$

set unification

set unification is the at the base of $\{log\}$

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

$$\{[a,1],[a,2],[b,1],[b,2]\} = \{[a,X] \,/\, A\} \rightarrow$$
$$X = 1, A = \{[a,2],[b,1],[b,2]\}$$
$$X = 1, A = \{[a,1],[a,2],[b,1],[b,2]\} \qquad [a,X] \in A$$

set unification

set unification is the at the base of $\{log\}$

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

$$\{[a,1],[a,2],[b,1],[b,2]\} = \{[a,X] \,/\, A\} \rightarrow$$
$$X = 1, A = \{[a,2],[b,1],[b,2]\}$$
$$X = 1, A = \{[a,1],[a,2],[b,1],[b,2]\} \qquad [a,X] \in A$$
$$X = 2, A = \{[a,1],[b,1],[b,2]\}$$

set unification

set unification is the at the base of $\{log\}$

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

$$\{[a,1],[a,2],[b,1],[b,2]\} = \{[a,X] \,/\, A\} \rightarrow$$

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

$$X = 1, A = \{[a,1],[a,2],[b,1],[b,2]\} \qquad [a,X] \in A$$

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

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

set unification

set unification is the at the base of $\{log\}$

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

$$\{[a,1],[a,2],[b,1],[b,2]\} = \{[a,X] \,/\, A\} \rightarrow$$

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

$$X = 1, A = \{[a,1],[a,2],[b,1],[b,2]\} \qquad [a,X] \in A$$

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

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

unification extracts elements and subsets by their properties

example: a bank account

a bank has many savings accounts

each account is identified by an ID

the bank keeps the balance of each account

customers can deposit and withdraw money

banks as a state machines

we model the bank as a state machine

the state is the balance of each savings account

state transitions: make a deposit and open account

savings accounts: function from account ID's onto balances

- functions are sets of ordered pairs

deposit

```
deposit(SA,A,D,SA_) :-

  SA = {[A,B] / Rest}    &

  0 < D    &

  NB is B + D    &

  SA_ = {[A,NB] / Rest}.
```

simulation

deposit is a forgram $\longrightarrow$ can be executed

simulation

deposit is a forgram $\longrightarrow$ can be executed

$$\overbrace{\texttt{SA = \{\} \& A = a \& D = 100}}^{\text{initial state + inputs}} \quad \& \quad \overbrace{\texttt{deposit(SA,A,D,SA\_)}}^{\text{simulate this}} \; .$$

simulation

deposit is a forgram $\longrightarrow$ can be executed

$$\overbrace{\texttt{SA = \{\} \& A = a \& D = 100}}^{\text{initial state + inputs}} \quad \& \quad \overbrace{\texttt{deposit(SA,A,D,SA\_)}}^{\text{simulate this}} \; .$$

no

because there are no accounts in the bank

simulation

deposit is a forgram $\longrightarrow$ can be executed

$$\overbrace{\texttt{SA = \{\} \& A = a \& D = 100}}^{\text{initial state + inputs}} \;\&\; \overbrace{\texttt{deposit(SA,A,D,SA\_)}}^{\text{simulate this}} .$$

no

because there are no accounts in the bank

```
SA = {[y,50]} & A = x & D = 100 & deposit(SA,A,D,SA_).
```

simulation

deposit is a forgram $\longrightarrow$ can be executed

$$\overbrace{\texttt{SA = \{\} \& A = a \& D = 100}}^{\text{initial state + inputs}} \;\&\; \overbrace{\texttt{deposit(SA,A,D,SA\_)}}^{\text{simulate this}} .$$

```
no
```

because there are no accounts in the bank

```
SA = {[y,50]} & A = x & D = 100 & deposit(SA,A,D,SA_).
no
```

because account **x** doesn't exist in the bank

simulation

deposit is a forgram $\longrightarrow$ can be executed

$$\overbrace{\texttt{SA = \{\} \& A = a \& D = 100}}^{\text{initial state + inputs}} \quad \& \quad \overbrace{\texttt{deposit(SA,A,D,SA\_)}}^{\text{simulate this}} \texttt{.}$$

no

because there are no accounts in the bank

```
SA = {[y,50]} & A = x & D = 100 & deposit(SA,A,D,SA_).
```

no

because account x doesn't exist in the bank

```
SA = {[y,50],[x,10]} & A = x & D = 100 & deposit(SA,A,D,SA_).
```

simulation

deposit is a forgram $\longrightarrow$ can be executed

initial state + inputs: `SA = {} & A = a & D = 100`   simulate this: `deposit(SA,A,D,SA_).`

```
SA = {} & A = a & D = 100 & deposit(SA,A,D,SA_).
```

no

because there are no accounts in the bank

```
SA = {[y,50]} & A = x & D = 100 & deposit(SA,A,D,SA_).
```

no

because account **x** doesn't exist in the bank

```
SA = {[y,50],[x,10]} & A = x & D = 100 & deposit(SA,A,D,SA_).
```

```
SA_ = {[x,110],[y,50]}
```

open an account

```
openacc(SA,ID,SA_) :- SA_ = {[ID,0] / SA}.
```

is it correct?

is openacc correct?

openacc is a forgram $\longrightarrow$ can be executed

is openacc correct?

openacc is a forgram $\longrightarrow$ can be executed

```
SA = {} & openacc(SA,x,S1) & openacc(S1,y,S2).
```

is openacc correct?

openacc is a forgram $\longrightarrow$ can be executed

```
SA = {} & openacc(SA,x,S1) & openacc(S1,y,S2).
      S2 = {[y,0],[x,0]}
```

is openacc correct?

openacc is a forgram $\longrightarrow$ can be executed

```
SA = {} & openacc(SA,x,S1) & openacc(S1,y,S2).
     S2 = {[y,0],[x,0]}
```

```
SA = {} & openacc(SA,x,S1) & deposit(S1,x,5,S2) & openacc(S2,x,S3).
```

is openacc correct?

openacc is a forgram $\longrightarrow$ can be executed

```
SA = {} & openacc(SA,x,S1) & openacc(S1,y,S2).
       S2 = {[y,0],[x,0]}
```

```
SA = {} & openacc(SA,x,S1) & deposit(S1,x,5,S2) & openacc(S2,x,S3).
       S3 = {[x,0],[x,5]}
```

two balances for the same account !

is openacc correct?

openacc is a forgram $\longrightarrow$ prove it correct ! !

is openacc correct?

openacc is a forgram $\longrightarrow$ prove it correct ! !

the set of accounts must be a function: $SA \in A \nrightarrow B$

this property must hold of every state $\longrightarrow$ state invariant

$$SA \in A \nrightarrow B \wedge openacc(SA, ID, SA') \implies SA' \in A \nrightarrow B$$

is openacc correct?

openacc is a forgram $\longrightarrow$ prove it correct ! !

the set of accounts must be a function: $SA \in A \nrightarrow B$

this property must hold of every state $\longrightarrow$ state invariant

$SA \in A \nrightarrow B \wedge openacc(SA, ID, SA') \implies SA' \in A \nrightarrow B$

in $\{log\} \longrightarrow$ prove if the negation is false

$SA \in A \nrightarrow B \wedge openacc(SA, ID, SA') \wedge SA' \notin A \nrightarrow B$

is openacc correct?

openacc is a forgram $\longrightarrow$ prove it correct ! !

the set of accounts must be a function: $SA \in A \nrightarrow B$

this property must hold of every state $\longrightarrow$ state invariant

$SA \in A \nrightarrow B \wedge openacc(SA, ID, SA') \implies SA' \in A \nrightarrow B$

in $\{log\} \longrightarrow$ prove if the negation is false

$SA \in A \nrightarrow B \wedge openacc(SA, ID, SA') \wedge SA' \notin A \nrightarrow B$

```
pfun(SA) & openacc(SA,ID,SA_) & npfun(SA_).
```

is openacc correct?

openacc is a forgram $\longrightarrow$ prove it correct ! !

the set of accounts must be a function: $SA \in A \nrightarrow B$

this property must hold of every state $\longrightarrow$ state invariant

$SA \in A \nrightarrow B \wedge openacc(SA, ID, SA') \implies SA' \in A \nrightarrow B$

in $\{log\}$ $\longrightarrow$ prove if the negation is false

$SA \in A \nrightarrow B \wedge openacc(SA, ID, SA') \wedge SA' \notin A \nrightarrow B$

```
pfun(SA)  &  openacc(SA,ID,SA_)  &  npfun(SA_).
```

SA = {[ID,_N2]/_N1}, SA_ = {[ID,0],[ID,_N2] / _N1}

the answer should be **no**

$\{log\}$ provides a counterexample

openacc, corrected

the problem is that we don't check that ID is a new id

openacc, corrected

the problem is that we don't check that ID is a new id

new definition

```
openacc(SA,ID,SA_) :- dom(SA,D) & ID nin D & SA_ = {[ID,0] / SA}.
```

openacc, corrected

the problem is that we don't check that ID is a new id

new definition

```
openacc(SA,ID,SA_) :- dom(SA,D) & ID nin D & SA_ = {[ID,0] / SA}.
```

let's try again

```
SA = {} & openacc(SA,x,S1) & deposit(S1,x,5,S2) & openacc(S2,x,S3).
```

openacc, corrected

the problem is that we don't check that ID is a new id

new definition

```
openacc(SA,ID,SA_) :- dom(SA,D) & ID nin D & SA_ = {[ID,0] / SA}.
```

let's try again

```
SA = {} & openacc(SA,x,S1) & deposit(S1,x,5,S2) & openacc(S2,x,S3).
        no
```

openacc, corrected

the problem is that we don't check that ID is a new id

new definition

```
openacc(SA,ID,SA_) :- dom(SA,D) & ID nin D & SA_ = {[ID,0] / SA}.
```

let's try again

```
SA = {} & openacc(SA,x,S1) & deposit(S1,x,5,S2) & openacc(S2,x,S3).
```

no

```
pfun(SA) & openacc(SA,ID,SA_) & npfun(SA_).
```

openacc, corrected

the problem is that we don't check that ID is a new id

new definition

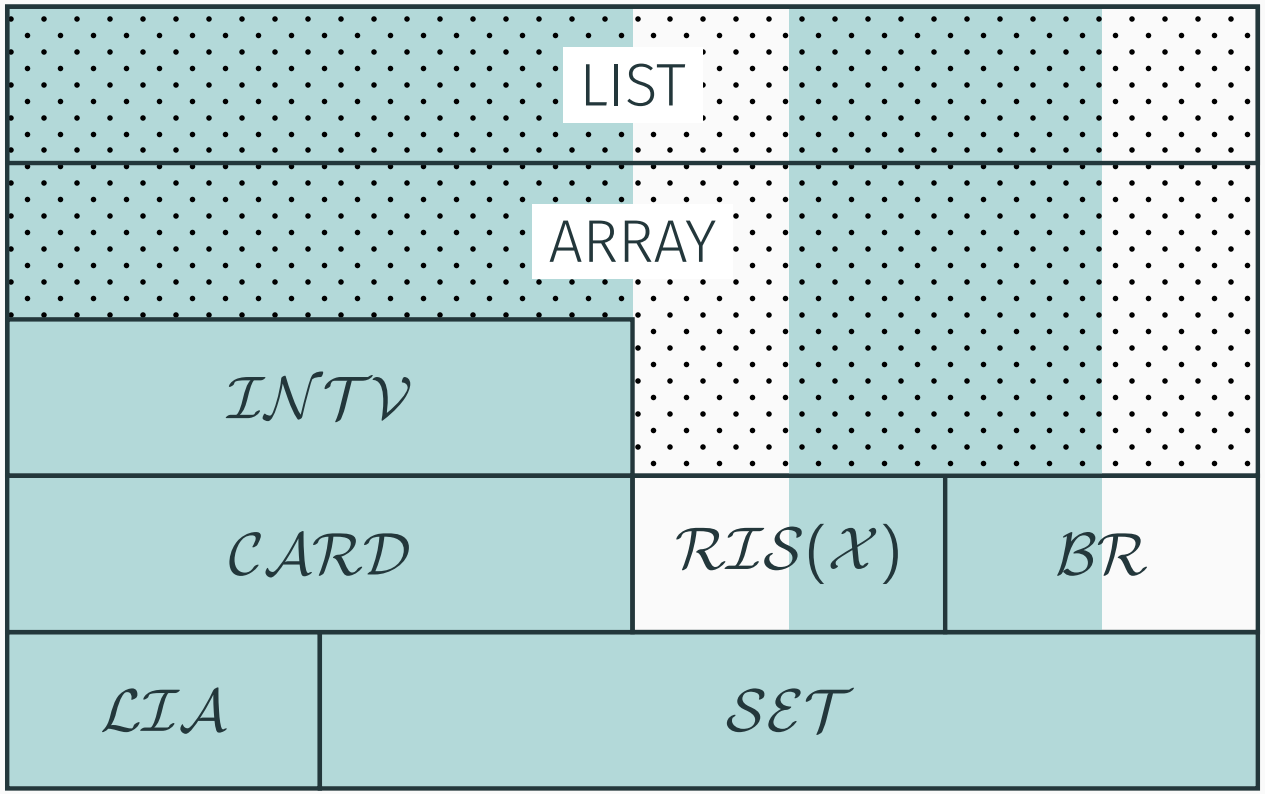
```
openacc(SA,ID,SA_) :- dom(SA,D) & ID nin D & SA_ = {[ID,0] / SA}.
```

let's try again

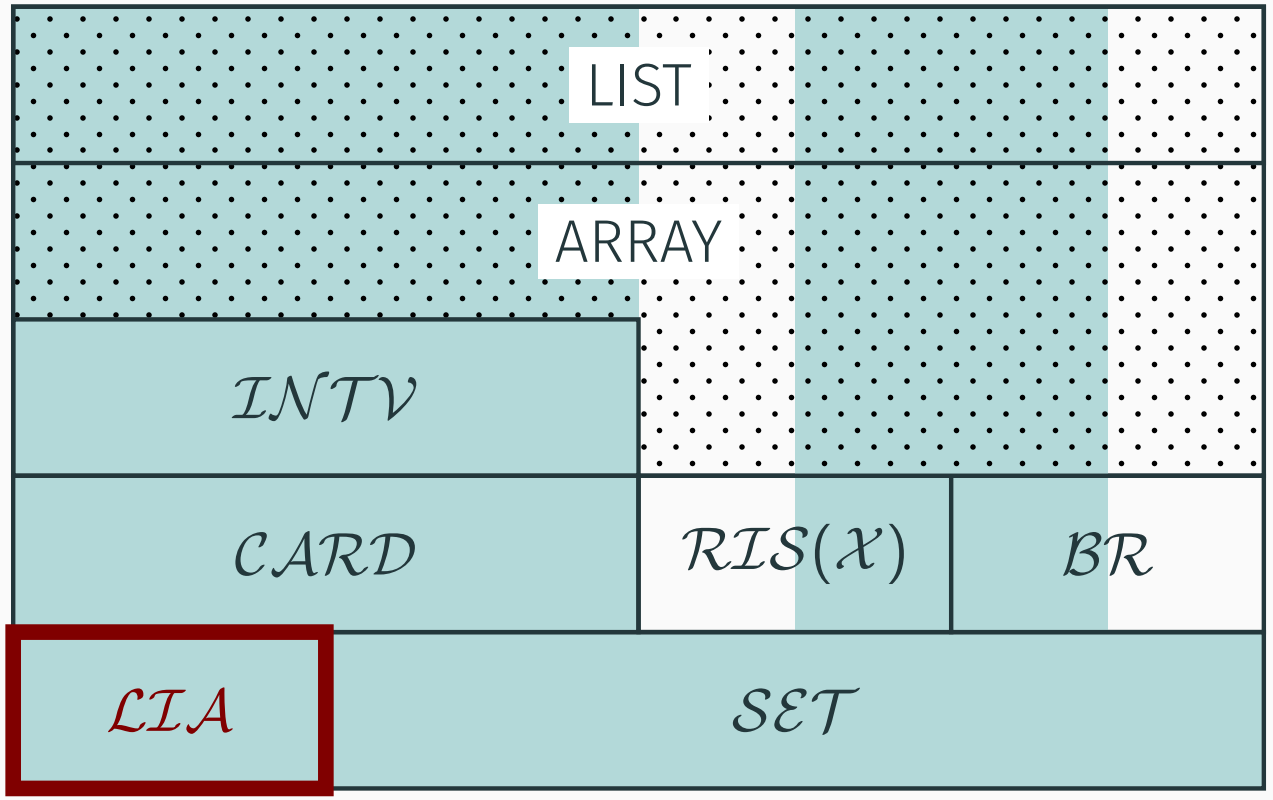
```
SA = {} & openacc(SA,x,S1) & deposit(S1,x,5,S2) & openacc(S2,x,S3).
        no

pfun(SA) & openacc(SA,ID,SA_) & npfun(SA_).
        no
```

what is {*log*} good for? (theory)

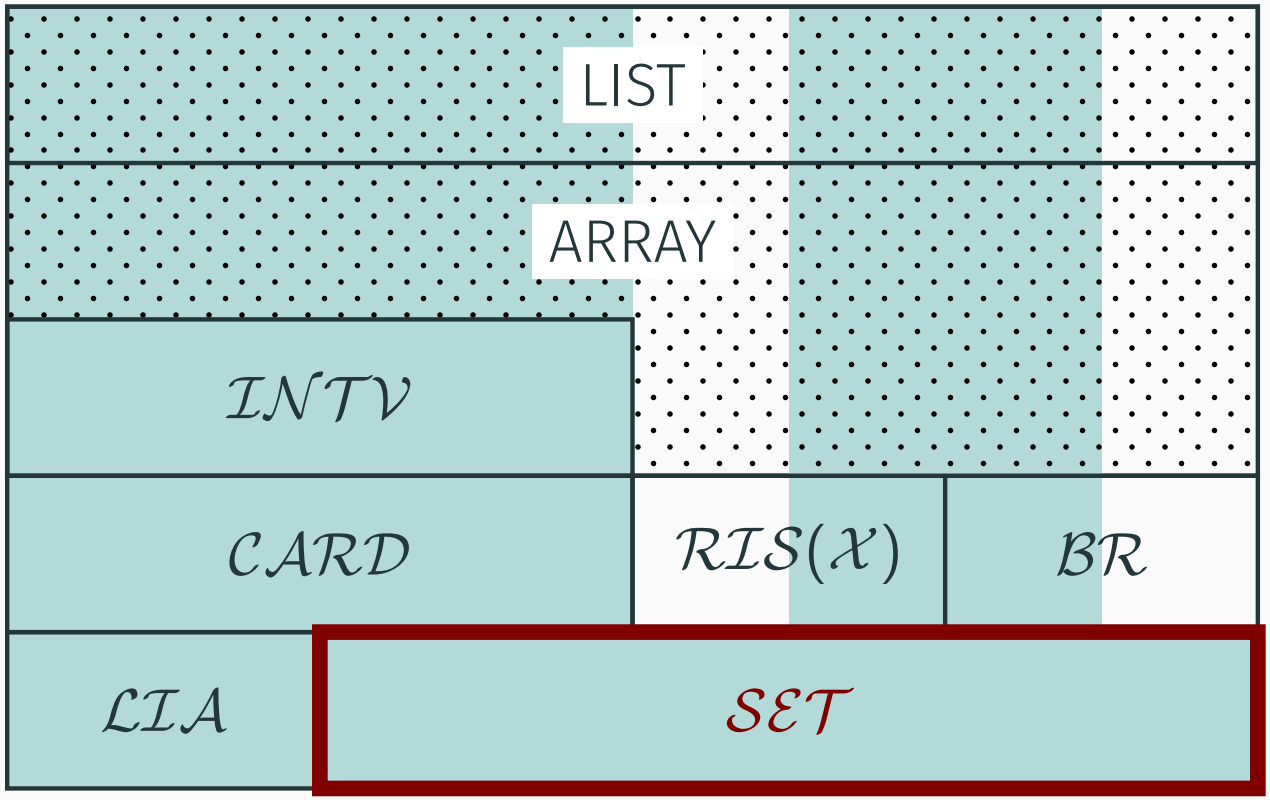


what is {*log*} good for? (theory)



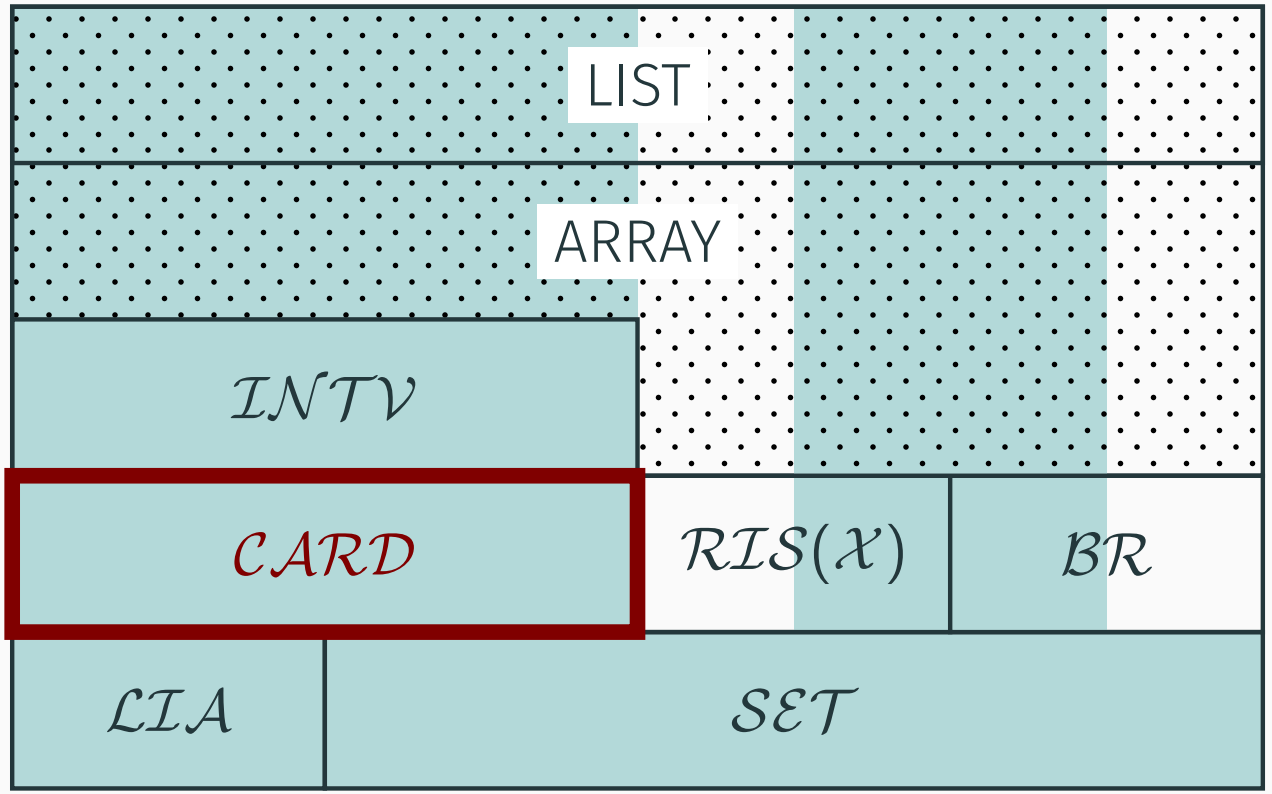
linear integer algebra

what is {*log*} good for? (theory)



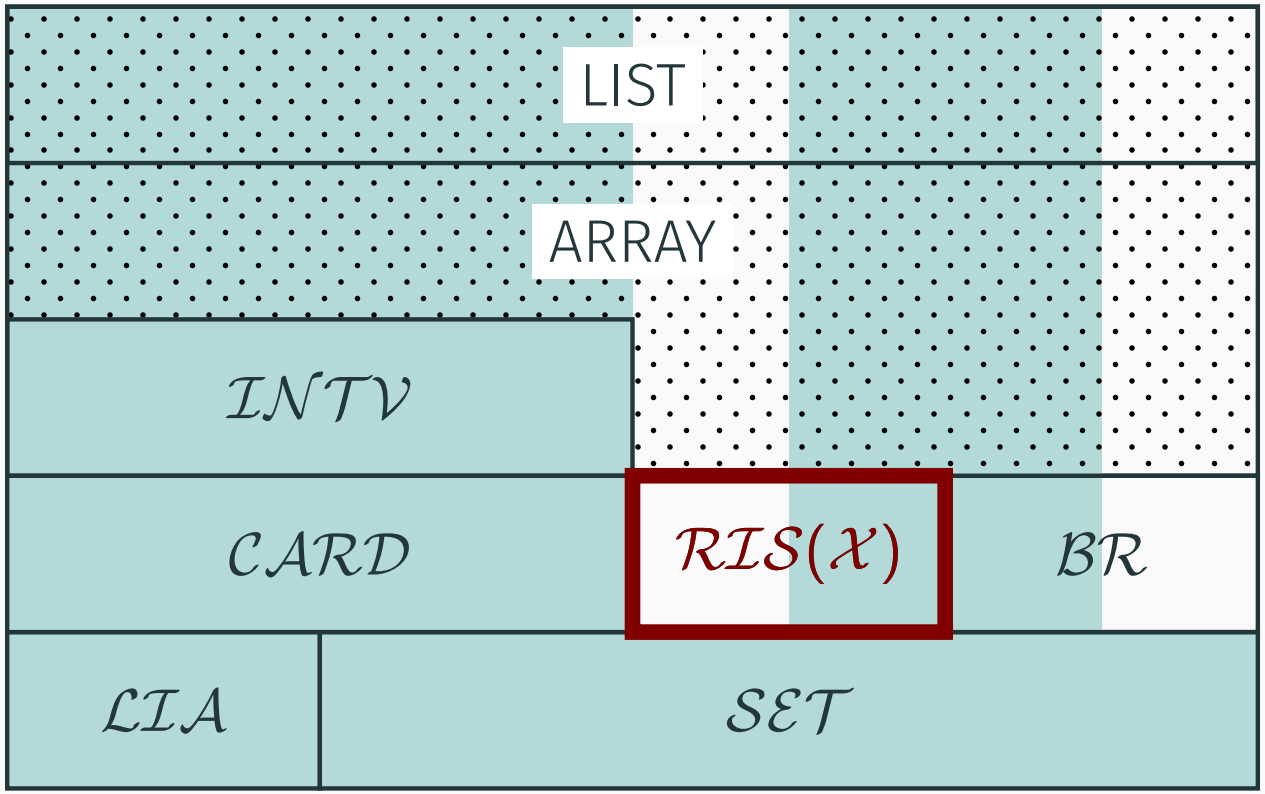
boolean algebra of finite sets

what is {*log*} good for? (theory)



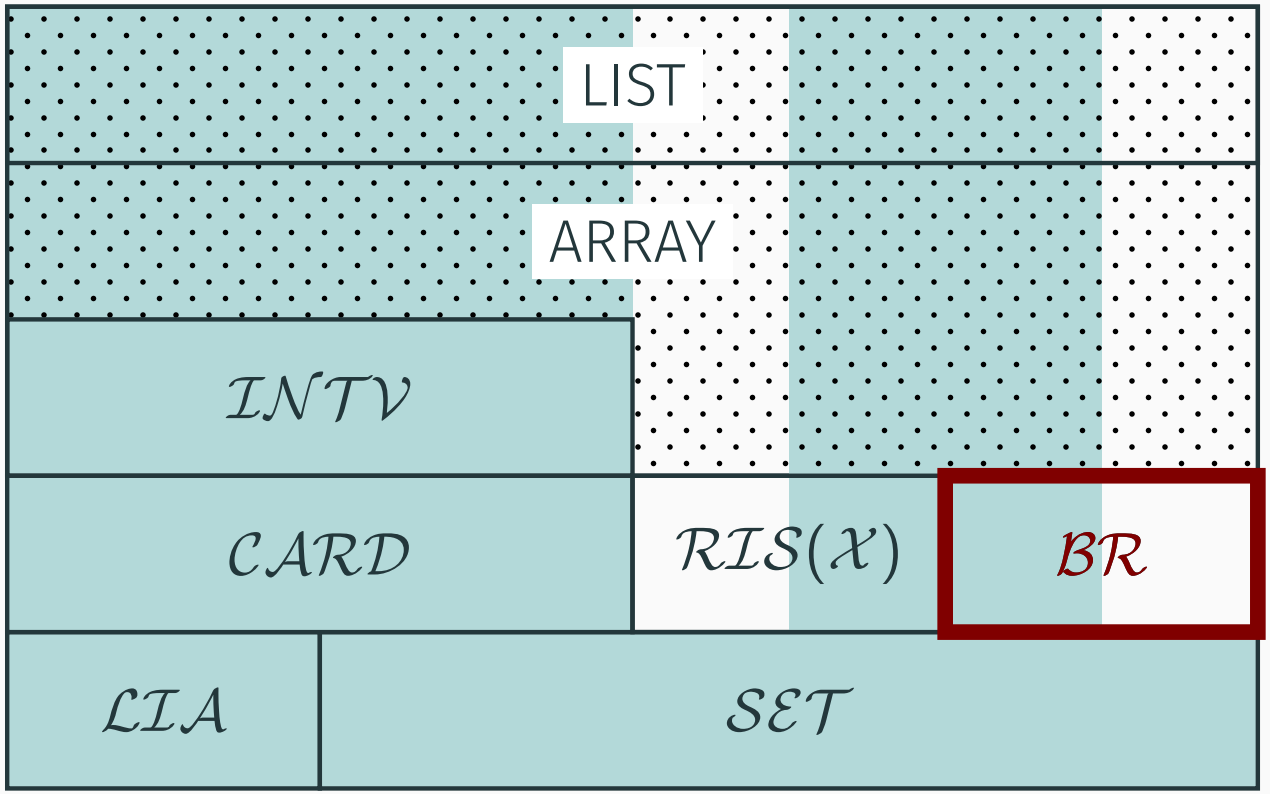
$\mathcal{SET}$ extended with cardinality

what is {*log*} good for? (theory)



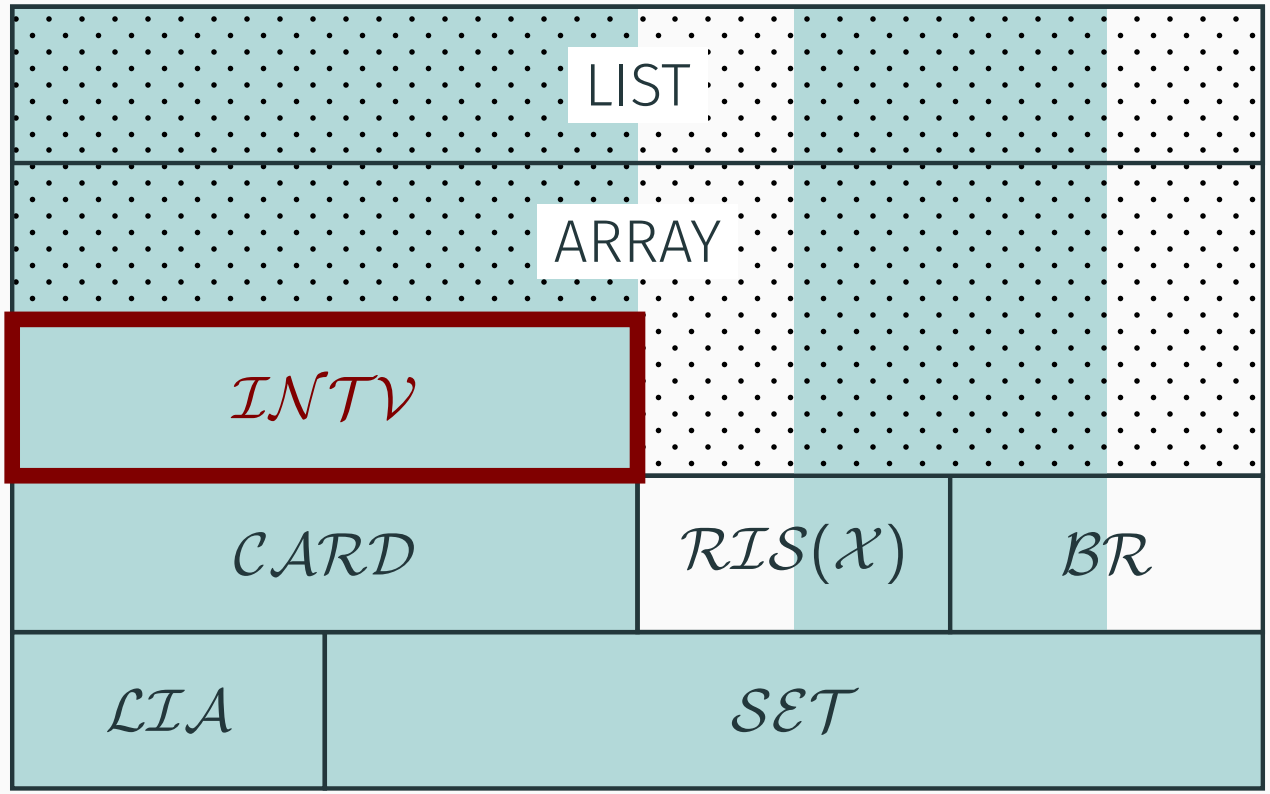
$\mathcal{SET}$ extended with intensional sets

what is {*log*} good for? (theory)



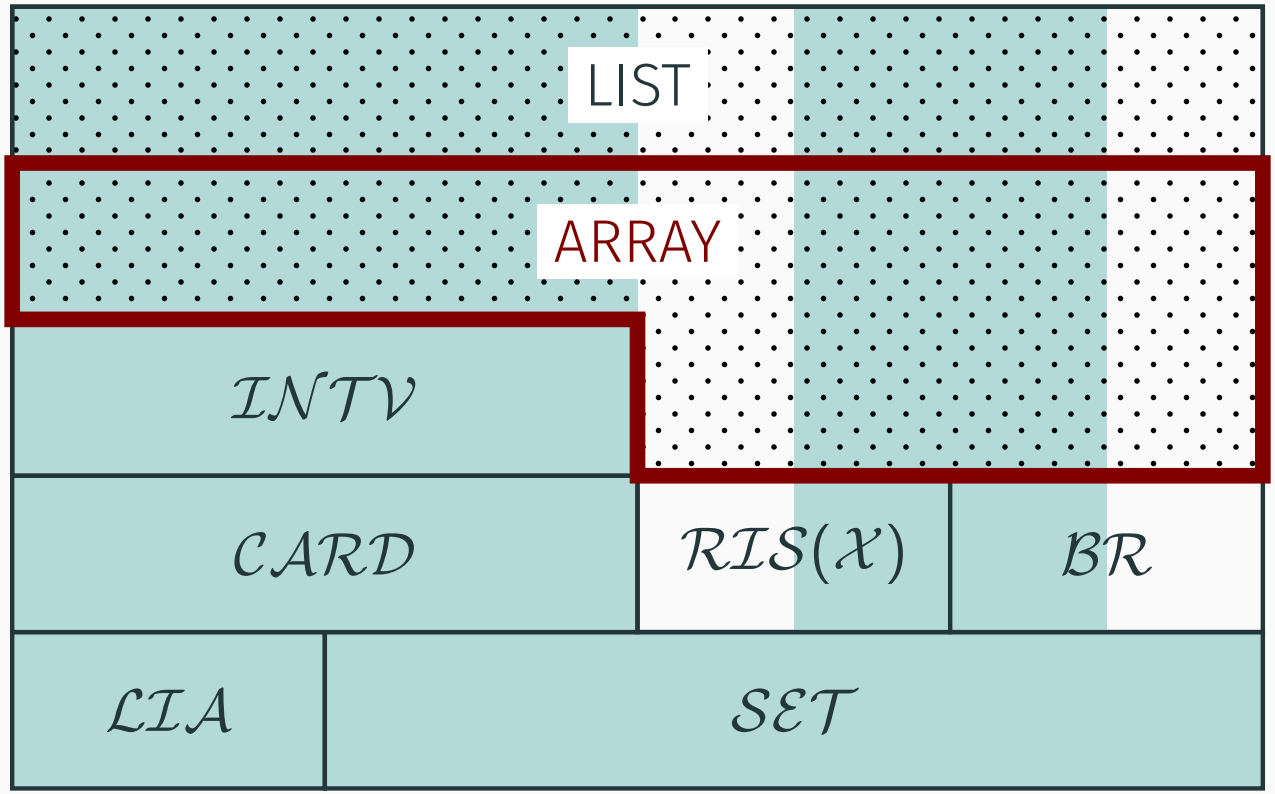
$\mathcal{SET}$ extended with set relation algebra

what is {*log*} good for? (theory)

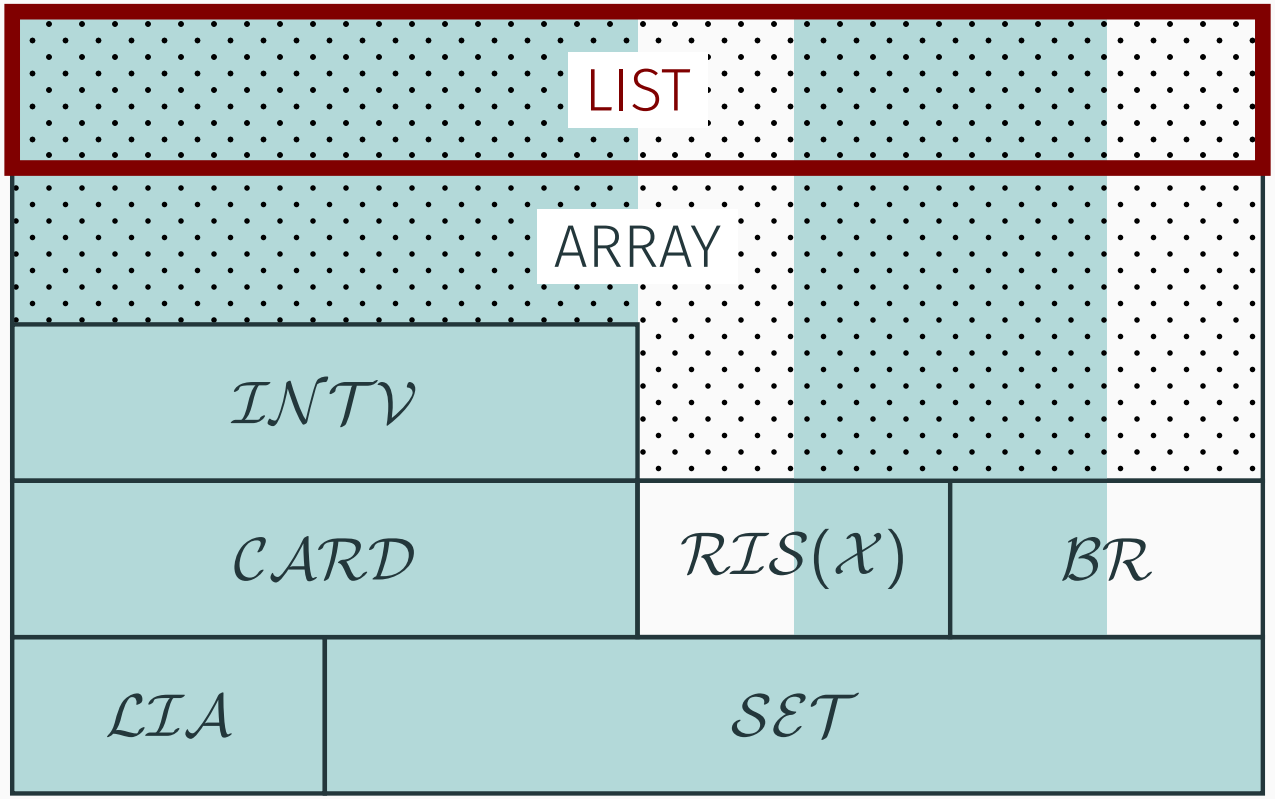


$\mathcal{CARD}$ extended with integer intervals

what is {*log*} good for? (theory)



what is {*log*} good for? (theory)



what is {*log*} good for? (practice)

automatically verified prototypes

Bell-LaPadula security model	
Tokeneer ID Station	Z
landing gear system	Event-B
android's permissions system	Coq

find {*log*} here

`www.clpset.unipr.it/setlog.Home.html`

or google

`gianfranco rossi setlog`

programs as formulas: conclusions

it is also difficult

programmers **must** write mathematics

programs are still prototypes

can we transform these prototypes into efficient programs?

can we prove that the new program is a correct implementation of the prototype?