Using a Set Constraint Solver for Program Verification

Maximiliano Cristiá — Universidad Nacional de Rosario – Argentina
Gianfranco Rossi — Università degli Studi di Parma – Italy
Claudia Frydman — Aix-Marseille Université – France

4th HCVS August 2017 Gothenburg - Sweden

{*log*}: a constraint solver for set theory

{*log*} is a complete solver for a fragment of set theory

- Prolog program based on set unification and CLP
- Rossi et al. 1991; Rossi & Cristiá since 2013

satisfiability solver

- returns a finite representation of all solutions of a given formula
- solution $\rightarrow$ assignment of values to the free variables of the formula

declarative programming language

sets in {*log*} are

- first-class entities
- finite, unbounded, untyped, nested, partially specified

$\{log\}$: some examples

set equality

$\{1, 2 \sqcup A\} = \{1, x, 3\}$

- $\{1, 2 \sqcup A\}$ is interpreted as $\{1, 2\} \cup A$
- $\{log\}$ returns four solutions

$$x = 2 \land A = \{3\} \qquad x = 2 \land A = \{2, 3\}$$
$$x = 2 \land A = \{1, 3\} \qquad x = 2 \land A = \{1, 2, 3\}$$

$\{log\}$: some examples

set equality

$\{1, 2 \sqcup A\} = \{1, x, 3\}$

- $\{1, 2 \sqcup A\}$ is interpreted as $\{1, 2\} \cup A$
- $\{log\}$ returns four solutions

$$x = 2 \wedge A = \{3\} \qquad x = 2 \wedge A = \{2, 3\}$$
$$x = 2 \wedge A = \{1, 3\} \qquad x = 2 \wedge A = \{1, 2, 3\}$$

set equality (unsatisfiable)

$\{1, 2 \sqcup A\} = \{1, x, 3\} \wedge x \neq 2$ — $\{log\}$ returns *false*

$\{log\}$: some examples

union is commutative (mathematics)

$A \cup B = B \cup A$

$\{log\}$: some examples

union is commutative (mathematics)

$A \cup B = B \cup A$

to prove it with $\{log\}$ enter the negation

union is commutative (negation in $\{log\}$)

$un(A, B, C) \land nun(B, A, C)$ $\{log\}$ returns *false*

$\{log\}$: some examples

union is commutative (mathematics)
$A \cup B = B \cup A$

to prove it with $\{log\}$ enter the negation

union is commutative (negation in $\{log\}$)	
$un(A, B, C) \land nun(B, A, C)$	$\{log\}$ returns *false*

- set operators become constraints

$\{log\}$: some examples

union is commutative (mathematics)

$A \cup B = B \cup A$

to prove it with $\{log\}$ enter the negation

union is commutative (negation in $\{log\}$)

$un(A, B, C) \wedge nun(B, A, C)$ $\{log\}$ returns *false*

- set operators become constraints
- the last formula can also be written as:
  $nun(A, B, C) \wedge un(B, A, C)$ or
  $un(A, B, C) \wedge un(B, A, XX) \wedge C \neq XX$

$\{log\}$: some examples

binary relations theorem (mathematics)

$(A \lhd R)[B] = R[A \cap B]$

A, B sets; R binary relation

$\lhd$ domain restriction; $\cdot[\cdot]$ relational image

$\{log\}$: some examples

binary relations theorem (mathematics)

$(A \lhd R)[B] = R[A \cap B]$

A, B sets; R binary relation

$\lhd$ domain restriction; $\cdot[\cdot]$ relational image

binary relations theorem (negation in $\{log\}$)

$dres(A, R, N_1) \land rimg(N_1, B, N_2) \land inters(A, B, N_3) \land nrimg(R, N_3, N_2)$

$\{log\}$: some examples

binary relations theorem (mathematics)

$(A \lhd R)[B] = R[A \cap B]$

A, *B* sets; *R* binary relation

$\lhd$ domain restriction; $\cdot[\cdot]$ relational image

binary relations theorem (negation in $\{log\}$)

$dres(A, R, N_1) \land rimg(N_1, B, N_2) \land inters(A, B, N_3) \land nrimg(R, N_3, N_2)$

- relational operators become constraints
- set and relations can be freely combined
- $\{log\}$ works as an automated theorem prover

{*log*}: functional partial program verification

first question

is {*log*} useful for *functional partial* program verification?

second question

will it automatically discharge verification conditions of a Hoare framework?

third question

if so, of what classes of programs?

specifications & programs

set theory is used as the specification language

much as in B and Z notations

programs are written in an abstract imperative language

abstract data types are also available

pre-conditions, loop invariants and post-conditions are given

Hoare rules apply

programs dealing with lists

an ADT named List is defined

an ADT for lists

```
adt List(T)
    public
        List()                    ▷ constructor
        add(T e)                  ▷ appends e to the list
        fst()
        T next()                  ▷ fst, next, more → abstract iterator
        Bool more()
        rpl(T e)                  ▷ replaces last iterated element with e
        del()                     ▷ empties the list
    end public
end adt
```

list subroutines

with the List ADT we can write list subroutines

list equality

```
function Bool listEq(List s, t)
    s.fst(); t.fst()
    while s.more() ∧ t.more() ∧ s.next() = t.next() do
        skip
    end while
    return ¬s.more() ∧ ¬t.more()
end function
```

and we can annotate subroutines with specifications

list equality

PRE-CONDITION $true$

```
function Bool listEq(List s, t)
    s.fst(); t.fst()
    INVARIANT s ∈ _ ⇸ _ ∧ s = s_p ∪ s_r ∧ s_p ∥ s_r
              ∧ t ∈ _ ⇸ _ ∧ t = t_p ∪ t_r ∧ t_p ∥ t_r
              ∧ s_p = t_p
    while s.more() ∧ t.more() ∧ s.next() = t.next() do
        skip
    end while
    return ¬s.more() ∧ ¬t.more()
end function
```

INVARIANT $s \in \_ \nrightarrow \_ \wedge s = s_p \cup s_r \wedge s_p \parallel s_r \wedge t \in \_ \nrightarrow \_ \wedge t = t_p \cup t_r \wedge t_p \parallel t_r \wedge s_p = t_p$

POST-CONDITION $ret \iff s = t$

specifications

annotations are formulas in our specification language

set theory + binary relations $\approx$ as in Z and B

$$
\begin{aligned}
\textsc{invariant}\quad & s \in \_ \nrightarrow \_ \wedge s = s_p \cup s_r \wedge s_p \parallel s_r \\
& \wedge\, t \in \_ \nrightarrow \_ \wedge t = t_p \cup t_r \wedge t_p \parallel t_r \\
& \wedge\, s_p = t_p
\end{aligned}
$$

- s program variable $\longrightarrow$ s specification variable
- s' $\longrightarrow$ value of s in the after state

specifications

INVARIANT $s \in \_ \nrightarrow \_ \wedge s = s_p \cup s_r \wedge s_p \parallel s_r$
$\wedge\, t \in \_ \nrightarrow \_ \wedge t = t_p \cup t_r \wedge t_p \parallel t_r$
$\wedge\, s_p = t_p$

if s is a List, then s enjoys List's interface properties:

- s is a set of ordered pairs $\langle \mathrm{m}, \mathrm{g}, \mathrm{b} \rangle \longrightarrow \{(1, m), (2, g), (3, b)\}$
- s is a partial function $s \in \_ \nrightarrow \_$
- s is partitioned by the iterator $s = s_p \cup s_r \wedge s_p \parallel s_r$

s_p processed part - s_r remaining part

all these properties are provable from List's specification

specifications

INVARIANT $s \in \_ \nrightarrow \_ \land s = s_p \cup s_r \land s_p \parallel s_r$
$\land\, t \in \_ \nrightarrow \_ \land t = t_p \cup t_r \land t_p \parallel t_r$
$\land\, s_p = t_p$

if s is a List, then s enjoys List's interface properties:

- s is a set of ordered pairs $\langle \mathrm{m}, \mathrm{g}, \mathrm{b} \rangle \longrightarrow \{(1, m), (2, g), (3, b)\}$
- s is a partial function $s \in \_ \nrightarrow \_$
- s is partitioned by the iterator $s = s_p \cup s_r \land s_p \parallel s_r$

  s_p processed part - s_r remaining part

  all these properties are provable from List's specification

- then processed parts are equal inside the loop $s_p = t_p$

verification conditions

Hoare rules are applied to generate verification conditions

the most complex verification conditions are

- if the loop condition holds, then the loop invariant is preserved after each iteration

  $\text{loop condition} \wedge \text{invariant} \wedge \text{iteration} \implies \text{invariant'}$

- upon termination of the loop its invariant implies the post-condition

  $\neg\, \text{loop condition} \wedge \text{invariant} \implies \text{post-condition}$

$\{log\}$ is used to automatically discharge vc's

verification condition: an example

an example from listEq

$$
\begin{aligned}
&(s_r = \emptyset && [\neg \text{ loop condition}]\\
&\quad \vee t_r = \emptyset\\
&\quad \vee s_r = \{(x, y_1) \sqcup s_r^1\} \wedge t_r = \{(x, y_2) \sqcup t_r^1\} \wedge y_1 \neq y_2)\\
&\wedge s \in \_ \nrightarrow \_ \wedge s = s_p \cup s_r \wedge s_p \parallel s_r && [\text{loop invariant}]\\
&\wedge t \in \_ \nrightarrow \_ \wedge t = t_p \cup t_r \wedge t_p \parallel t_r\\
&\wedge s_p = t_p\\
&\implies ((s_r = \emptyset \wedge t_r = \emptyset) \iff s = t) && [\text{postcondition}]
\end{aligned}
$$

verification conditions in $\{log\}$

the negation of vc's have to be translated into $\{log\}$

this translation is straightforward

$$
\begin{aligned}
&(s_r = \emptyset \\
&\quad \vee t_r = \emptyset \\
&\quad \vee s_r = \{(x, y_1) \sqcup s_r^1\} \wedge t_r = \{(x, y_2) \sqcup t_r^1\} \wedge y_1 \neq y_2) \\
&\wedge pfun(s) \wedge un(s_p, s_r, s) \wedge disj(s_p, s_r) \\
&\wedge pfun(t) \wedge un(t_p, t_r, t) \wedge disj(t_p, t_r) \\
&\wedge s_p = t_p \\
&\wedge (s_r = \emptyset \wedge t_r = \emptyset \wedge s \neq t \vee s = t \wedge (s_r \neq \emptyset \vee t_r \neq \emptyset))
\end{aligned}
$$

List's specification

List is implemented as a singly-linked list

each node of the list is of type Node

- a simple ADT with two fields: next and elem
- methods: setNext, getNext, setElem, getElem

instances of Node are modeled as ordered pairs: (n, e)

instances of List are modeled as partial functions:

$$\{c_1 \mapsto (c_2, e_1), c_2 \mapsto (c_3, e_2), \ldots, c_n \mapsto (null, e_n)\}$$

representing the list $\langle e_1, e_2, \ldots, e_n \rangle$

List's specification

in the specification of List we use three state variables

- $s \rightarrow$ representing the heap

  $\{c_1 \mapsto (c_2, e_1), c_2 \mapsto (c_3, e_2), \ldots, c_n \mapsto (null, e_n)\}$

- $a \rightarrow$ representing a stack-allocated variable store

  $\{\mathrm{v}_1 \mapsto c_1, \ldots, \mathrm{v}_\mathrm{m} \mapsto c_m\}$

- $s_m \rightarrow$ representing the memory locations of s whose nodes have already been iterated over

  $\{c_1, \ldots, c_k\}$

then $s_p \triangleq s_m \lhd s$ and $s_r \triangleq s \setminus s_p$

List's implementation

internally List maintains these member variables of type Node

- s $\rightarrow$ holding the first node of the list
- f $\rightarrow$ holding the last node of the list
- c $\rightarrow$ holding the current position of the iterator
- p $\rightarrow$ holding the previous position of the iterator

List's verification

more() → **returns** true **iff there are more elements**

PRE-CONDITION *true*

function more()

  return c $\neq$ null

end function

POST-CONDITION $ret \iff a(\mathrm{c}) \neq null$

List's verification

more() $\rightarrow$ **returns** true **iff there are more elements**

PRE-CONDITION *true*

function more()

return c $\neq$ null

end function

POST-CONDITION $ret \iff a(\mathrm{c}) \neq null$

rpl() $\rightarrow$ **replaces** p **with** e

PRE-CONDITION $a(\mathrm{p}) \neq null$

procedure rpl(T e)

p.setElem(e)

end procedure

POST-CONDITION $s' = s \oplus \{a(\mathrm{p}) \mapsto (y, e)\} \wedge s(a(\mathrm{p})) = (y, \_)$

specification invariants

$s \in \_ \nrightarrow \_$

$s = s_p \cup s_r$

$s_p \parallel s_r$ are state invariants of List's specification

$\{log\}$ can prove that List's interface preserves them

- if the invariant holds and a subroutine is executed, then the invariant must hold in the after state

  invariant $\wedge$ subroutine $\implies$ invariant'

specification invariants: an example

add() preserves $s \in \_ \rightarrow\!\!\!\!\!+ \_$

$$
\begin{array}{lr}
s \in \_ \rightarrow\!\!\!\!\!+ \_ & \text{[spec invariant]} \\
\wedge\, (s = \emptyset \wedge c \neq \mathit{null} \wedge s' = \{c \mapsto (\mathit{null}, e)\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} & \text{[add() spec]} \\
\quad \vee\, s = \{a(\mathsf{f}) \mapsto (y, z) \sqcup s_1\} \wedge c \notin \operatorname{dom} s \wedge c \neq \mathit{null} & \\
\qquad \wedge\, s' = \{c \mapsto (\mathit{null}, e), a(\mathsf{f}) \mapsto (c, z) \sqcup s_1\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\}) & \\
\implies s' \in \_ \rightarrow\!\!\!\!\!+ \_ & \text{[spec invariant']}
\end{array}
$$

$$
\begin{array}{lr}
\mathit{pfun}(s) & \text{[negation in } \{\mathit{log}\}\text{]} \\
\wedge\, (s = \emptyset \wedge c \neq \mathit{null} \wedge s' = \{(c, (\mathit{null}, e))\} \wedge \mathit{oplus}(a, \{(\mathsf{f}, c)\}, a') & \\
\quad \vee\, \mathit{apply}(a, \mathsf{f}, m_2) \wedge s = \{(m_2, (y, z)) \sqcup s_1\} \wedge \mathit{dom}(s, m_1) \wedge c \notin m_1 \wedge c \neq \mathit{null} & \\
\qquad \wedge\, \mathit{apply}(a, \mathsf{f}, m_2) \wedge s' = \{(c, (\mathit{null}, e)), (m_2, (c, z)) \sqcup s_1\} & \\
\qquad \wedge\, \mathit{oplus}(a, \{(\mathsf{f}, c)\}, a')) & \\
\wedge\, \mathit{npfun}(s') &
\end{array}
$$

case study

$\{log\}$ is used to prove the functional partial correctness of six subroutines based on List plus many specification invariants

GROUP	VC	TIME	AVG
loop invariant	6	1,639 ms	271 ms
post-condition	6	37 ms	6 ms
specification invariant	22	1,170 ms	53 ms
other properties	3	6 ms	2 ms
TOTALS	37	2,843 ms	76 ms

$\{log\}$ proves all the 37 vc's in less than 0.1s each

summary

$\{log\}$ is a CLP solver for a fragment of set theory

it can automatically prove theorems of set theory

set-based specifications are good for many programs

$\{log\}$ is able to automatically discharge vc's generated in a Hoare framework whose assertions are set formulas