Abstract Interpreter in Rust

Brugnera Luca

Content

- requirements
- code parsing
- interpreter implementation
  - propagation algorithm
- $\text{Int}_{m,n}$ domain
  - partial order
  - widening
  - narrowing

Requirements

- develop an interpreter based on abstract denotational semantic for a generic non-relational numerical abstract domain
- numerical abstract domain is a complete lattice
- compute loops and program invariants as output
- instantiate the interpreter to the $\mathrm{Int}_{m,n}$ domain

Code parsing

- lexer: *logos*
- parser: *lalrpop*
- language:
  - Aexp $\ni$ e ::= x | n | e_1 op e_2, where n$\in$Z, op $\in${+,-,*,/}
  - Bexp $\ni$ b ::= true | false | e_1 = e_2 | e_1 < e_2 | b_1 $\wedge$ b_2 | !b
  - While $\ni$ S ::= x := e | skip | S_1; S_2 | if b then S_1 else S_2 | while b do S
- boolean expression simplifications:
  - ! operator
  - expression $\bowtie$ 0, $\bowtie \in$ {<, ≥, =, ≠}
- initial state (optional):
  - 1st program line: *assume* (*var* := [*low*, *upper*];)* , where *low* := -inf | n, *upper* := n | inf, n $\in$Z
  - bad casting or no initial state: $\top$

Code parsing - example

```
assume x := [10, 10]

#will be simplified to x ⩾ 1
while ! x < 1 do {
  x := x - 2
}
```

```
Program: While {
    pos: Position {
        line: 3,
        clm: 0,
    },
    guard: ArithmeticCondition(
        ArithmeticCondition {
            lhs: BinaryOperation {
                lhs: Variable(
                    "x",
                ),
                operator: Sub,
                rhs: Integer(
                    1,
                ),
            },
            operator: GreaterOrEqual,
        },
    ),
    body: Assignment(
        Assignment {
            var: "x",
            value: BinaryOperation {
                lhs: Variable(
                    "x",
                ),
                operator: Sub,
                rhs: Integer(
                    2,
                ),
            },
        },
    ),
}
```

Interpreter<D>

- based on abstract denotational semantics
- uses generic non-relational numerical abstract domain: AbstractDomain trait
- its initialization requires:
    - non-relational abstract numerical domain
    - program AST
    - initial state (optional)
- stores invariants discovered throughout the analysis
- state<D> abstraction:
    - implemented as an hashmap
    - implements some AbstractDomain methods trait var-wise

AbstractDomain trait

- represents the generic non-relational abstract domain
- methods:
  - top
  - bottom
  - union
  - intersection
  - partial ordering
  - arithmetic operators (forward and backward)
  - widening with thresholds (optional)
  - narrowing (intersection as default)
  - constant and interval abstraction

Loops abstract denotational semantic

- optional: widening with thresholds
  - depending on the satisfiability of ACC
  - set of thresholds = $\{c \in Z \mid c \text{ appears in the code}\} \cup \{ 0 \}$
- narrowing:
  - until fixpoint and for a finite amount of steps (*NARROWING_STEPS* env. variable)

Boolean expressions abstract denotational semantic

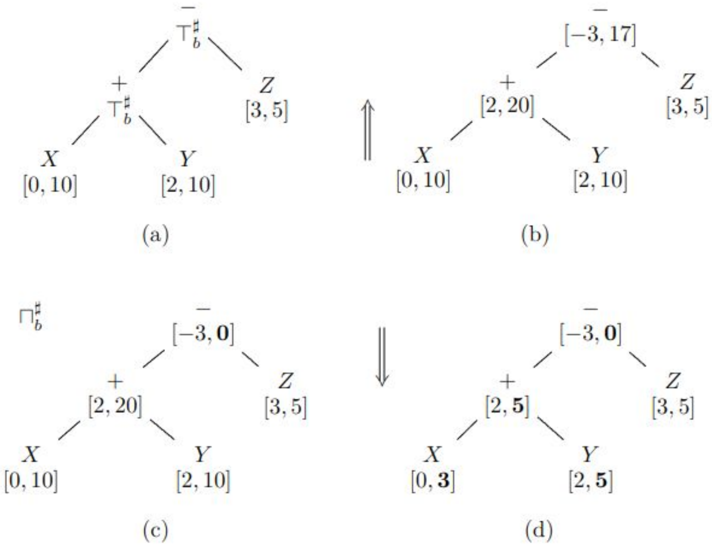
- trivial cases for *true* and *false*
- propagation algorithm for expression of the form $\bowtie 0$
- fixpoint for & and |, i.e. given a state $s^\#$, enforce $s^\# = cond[B_1 \& B_2]s^\#$ (same for |)
  - & and | aren't idempotent

Propagation algorithm

- expression $\bowtie$ 0 as binary tree
  - internal nodes: arithmetic operators
  - leaves: constant and variables
  - variable nodes are unique: multiple refinements are aggregated together using $\cap^{\#}$
- algorithm:
  - forward analysis: bottom-up tree traversal evaluates the expression
  - root node value *V* intersected with the test condition
    - $\bowtie = \{ = \} \rightarrow [0,0]^{\#}$
    - $\bowtie = \{ < \} \rightarrow [-inf,-1]^{\#}$
    - $\bowtie = \{ \geq \} \rightarrow [0,inf]^{\#}$
    - $\bowtie = \{ \neq \} \rightarrow V < 0 \ \cup^{\#} V \cap^{\#} [1,inf]^{\#}$
  - backward analysis: top-down tree traversal pushes the refinement to leaves
    - checking if the leaf refinement is $\perp$

Propagation algorithm example - 1

- $C^{\#}[(x+y)-z \leq 0]R^{\#}$, $R^{\#}$= { x = [0,10], y = [2,10], z = [3,5] }



Propagation algorithm example - 2

- (x+y)-z ≤0 ≡ ! (! (x+y)-z < 0 & (x+y)-z = 0) using *While* syntax
- ! (! (x+y)-z < 0 & (x+y)-z = 0) becomes (x+y)-z < 0 | (x+y)-z = 0
- focus on lhs: (x+y)-z < 0

```
After forward analysis
- [-3,17]
|  + [2,20]
|  |  Var [0,10]
|  |  Var [2,10]
|  Var [3,5]
After backward analysis
- [-3,-1]
|  + [2,4]
|  |  Var [0,2]
|  |  Var [2,4]
|  Var [3,5]
```

```
After forward analysis
- [-3,3]
|  + [2,6]
|  |  Var [0,2]
|  |  Var [2,4]
|  Var [3,5]
After backward analysis
- [-3,-1]
|  + [2,4]
|  |  Var [0,2]
|  |  Var [2,4]
|  Var [3,5]
```

Propagation algorithm example - 3

- focus on rhs: $(x+y)-z = 0$

```
After forward analysis
- [-3,17]
    + [2,20]
    |  Var [0,10]
    |  Var [2,10]
    Var [3,5]
After backward analysis
- [0,0]
    + [3,5]
    |  Var [0,3]
    |  Var [2,5]
    Var [3,5]
```

```
After forward analysis
- [-3,5]
    + [2,8]
    |  Var [0,3]
    |  Var [2,5]
    Var [3,5]
After backward analysis
- [0,0]
    + [3,5]
    |  Var [0,3]
    |  Var [2,5]
    Var [3,5]
```

- $C^\#[(x+y)-z < 0]R^\# \cup^\# C^\#[(x+y)-z = 0]R^\# = Q^\#$ after one iteration
  $Q^\# = \{ x = [0,3], y = [2,5], z = [3,5] \}$.
  recall that I want a fixpoint, but $C^\#[(x+y)-z \leq 0]Q^\# = Q^\#$, so $C^\#[(x+y)-z \leq 0]R^\# = Q^\#$

$Int_{m,n}$ domain

- $Int_{m,n}$ =
  - $\{\varnothing, Z\} \cup \{[k,k] \mid k \in Z\} \cup$
  - $\{[a,b] \mid a,b \in Z,\ a < b,\ [a,b] \subseteq [m,n]\} \cup$
  - $\{[-inf,k] \mid k \in Z,\ k \in [m,n]\} \cup$
  - $\{[k,inf] \mid k \in Z,\ k \in [m,n]\}$
- regard *m,n* (M and N env. var respectively) values:
  - constant domain
  - interval integer domain
  - *restricted* interval integer domain
- struct *Interval* {low: *Int*, upper: *Int*}, enum *Int* {*NegInf*, *Num*(i64), *PosInf*}
- most of the domain operations are implemented as in the interval domain
- multiple representations of the same element require the definition of equivalence

$Int_{m,n}$ domain - equivalence operator - 1

- = must be defined properly
- regardless the domain bounds
  - $\top = \top$
  - [-inf, inf] $= \top$
  - $\bot = \bot$
  - [a,b], a > b = $\bot$
  - a = c, b = d ➜ [a,b] = [c,d]
- constant domain extension
  - [a,b], a < b = $\top$

$Int_{m,n}$ domain - equivalence operator - 2

- *restricted* interval integer domain extension
  - things get slightly more complicated, given the abstract element [a,b]:
  - $b \leq m \rightarrow [a,b] = [-\inf, m]$
  - $a \geq n \rightarrow [a,b] = [n, \inf]$
  - $a \leq m,\ b \geq n \rightarrow \top$
  - otherwise [low,upper], $low \in [m,n]$ | $upper \in [m,n]$. Given [a,b] and [c,d]
    - $a,c < m \rightarrow [a,b] = [c,d] \Longleftrightarrow b = d$
    - $b,d > n \rightarrow [a,b] = [c,d] \Longleftrightarrow a = c$

$Int_{m,n}$ domain - partial order

- partial order relation for $Int_{m,n}$ ➜ $\leq$
- regardless domain bounds
- $[a,b] = [c,d]$ ➜ aforementioned equivalence
- $[a,b] < [c,d]$ ➜
  - lhs = $\bot$ & rhs $\neq \bot$
  - lhs $\neq \top$ & rhs = $\top$
  - $c < a$ & $b < d$

$Int_{m,n}$ domain - widening

- constant domain and *restricted* interval domain (both ACC): no widening
- interval domain:

$$[a,b] \nabla_b^T [c,d] \stackrel{\text{def}}{=} \left[\begin{cases} a & \text{if } a \leq c \\ \max\{\, x \in T \mid x \leq c \,\} & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ \min\{\, x \in T \mid x \geq d \,\} & \text{otherwise} \end{cases} \right]$$

$T = \{c \in \mathbb{Z} \mid c \text{ appears in the code}\} \cup \{\, 0 \,\}$

$Int_{m,n}$ domain - widening example

```
x := 10;
while ! x < 0 do {
  x := x - 1
}
```

- m = -1, n = 10

```
Seeking loop invariant
x → [10,10]  [9,10]  [8,10]  [7,10]  [6,10]  [5,10]  [4,10]  [3,10]  [2,10]  [1,10]  [0,10]  [-1,10]
```

- interval domain

```
Seeking loop invariant
x → [10,10]  [1,10]  [0,10]  [-inf,10]
```

$\text{Int}_{m,n}$ domain - narrowing

$$[a,b]\,\triangle_b\,[c,d] \stackrel{\text{def}}{=} \left[\begin{cases} c & \text{if } a=-\infty \\ a & \text{otherwise}\end{cases}, \begin{cases} d & \text{if } b=+\infty \\ b & \text{otherwise}\end{cases}\right]$$

- just 1 step required
- example:

```
#source code

x := 1;
while  x < 2 do {
  x := x + 4
}
```

```
#analysis with NARROWING_STEPS = 0

x := 1;
# LOOP INVARIANT: { x := [1,inf] }
while  x < 2 do {
  x := x + 4
}

# { x := [2,inf] }
```

```
#analysis with NARROWING_STEPS = 1

x := 1;
# LOOP INVARIANT: { x := [1,5] }
while  x < 2 do {
  x := x + 4
}

# { x := [2,5] }
```

$[1,\text{inf}] \triangle ([1,1] \cup [5,5]) = [1,5]$
$[1,5] \triangle ([1,1] \cup [5,5]) = [1,5]$