The Undefined Domain: Precise Relational Information for Entities that Do Not Exist

Holger Siegel, Bogdan Mihaila and Axel Simon

Technische Universität München
Institut für Informatik II

November 20, 2013

Static Analysis in the Presence of Dynamic Memory

Abstract Interpretation

- *domains* approximate sets of program states
- numeric domains hold numeric valuations for variables (state $s \subseteq \chi(s) \to \mathbb{Z}$ where $\chi(s)$ is the support set of s)
- e.g. *Intervals* domain gives non-relational approximation (state $s : \chi(s) \to \mathcal{I}$)
- *relational* domains infer relations between variables, e.g.

  Polyhedra (linear inequalities)

  TVPI (two variables per inequality)

Challenge: Joining states with different support sets

dynamic memory

how to represent regions that only exist in some traces

uninitialized values

how to avoid precision loss in presence of uninitialized ($\top$) values

Scenario 1: Non-existing fields

```
int * p;
if(rnd()) {
  p = malloc(4);
  *p = 3;
  // [p = 1 · &x, x = 3]
} else {
  p = NULL;
  // [p = 0 · &x]
}
// [p = [0, 1] · &x, x = ?]
```

Scenario 1: Non-existing fields

```
int * p;
if(rnd()) {
   p = malloc(4);
   *p = 3;
   // [p = 1 · &x, x = 3]
} else {
   p = NULL;
   // [p = 0 · &x, x = ⊥]
}
// [p = [0, 1] · &x, x = 3]
```

Attempt 1: add $\bot$ value

- $\bot \equiv \emptyset$
- most precise
- can be done in *Intervals* domain

Scenario 1: Non-existing fields

```
int * p;
if(rnd()) {
   p = malloc(4);
   *p = 3;
   // [p = 1 · &x, x = 3]
} else {
   p = NULL;
   // [p = 0 · &x, x = ⊥]
}
// [p = [0, 1] · &x, x = 3]
```

Attempt 1: add $\bot$ value

- $\bot \equiv \emptyset$
- most precise
- can be done in *Intervals* domain

But:

- not possible when domain models *smash product*:
  no "individual" $\bot$ value for x
- relational domains usually model smash products:
  rules out Polyhedra, TVPI, Octagon, ...

Problem 1: Non-existing fields

```
int * p;
if(rnd()) {
  p = malloc(4);
  *p = 3;
  // [p = 1 · &x, x = 3]
} else {
  p = NULL;
  // [p = 0 · &x, x = ⊤]
}
```

// $[p = [0,1] \cdot \&x,\ x = \begin{cases} \top & \text{if } p = 0 \\ 3, & \text{otherwise} \end{cases}]$

Attempt 2: add $\top$ value

- here, $\top \equiv \mathbb{Z}$
- can be represented in relational domains

Problem 1: Non-existing fields

```
int * p;
if(rnd()) {
  p = malloc(4);
  *p = 3;
  // [p = 1 · &x, x = 3]
} else {
  p = NULL;
  // [p = 0 · &x, x = ⊤]
}
```

// $[p = [0,1] \cdot \&x,\ x = \begin{cases} \top & \text{if } p = 0 \\ 3, & \text{otherwise} \end{cases}]$

Attempt 2: add $\top$ value

- here, $\top \equiv \mathbb{Z}$
- can be represented in relational domains

But:

- no linear inequality can express relation between p and x
- Approximating state in Polyhedra domain gives

$$[p \in [0,1] \cdot \&x,\ x = \top]$$

Scenario 1: Non-existing fields

```
int * p;
if(rnd()) {
  p = malloc(4);
  *p = 3;
  // [p = 1 · &x, x = 3]
} else {
  p = NULL;
  // [p = 0 · &x, x = 3]
}
// [p = [0, 1] · &x, x = 3]
```

Attempt 3: *copy and paste* value

- copy value from other branch
- paste value to missing field

Scenario 1: Non-existing fields

```
int * p;
if(rnd()) {
  p = malloc(4);
  *p = 3;
  // [p = 1 · &x, x = 3]
} else {
  p = NULL;
  // [p = 0 · &x, x = 3]
}
// [p = [0, 1] · &x, x = 3]
```

Attempt 3: *copy and paste* value

- copy value from other branch
- paste value to missing field

Success

- can be used in relational domains
- copied value "fits into" joined state
- avoids precision loss
- but: only sound when pasted value is not reachable

Scenario 2: Non-initialized fields

```
int x, y;
  // [x = ⊤, y = ⊤]
  //
if(rnd()) {
  x = 1;
  y = 2;
  // [x = 1, y = 2]
  //
} else {
  x = 0;
  // [x = 0, y = ⊤]
  //
}
// [x ∈ {0,1}, y = { ⊤  if x = 0 ; 2, otherwise }]
//
```

Precision loss

- final state approximated in Polyhedra domain:

$$[x \in \{0,1\},\ y = \top]$$

- and we cannot copy and paste...

Scenario 2: Non-initialized fields

```
int x, y;
  // [x = ⊤, y = ⊤]
  // []
if(rnd()) {
  x = 1;
  y = 2;
  // [x = 1, y = 2]
  //
} else {
  x = 0;
  // [x = 0, y = ⊤]
  //
}
// [x ∈ {0,1}, y = { ⊤ if x = 0 ; 2, otherwise }]
//
```

Precision loss

- final state approximated in Polyhedra domain:

$$[x \in \{0,1\},\ y = \top]$$

- and we cannot copy and paste...

Solution: Avoid $\top$ as value

- do not initialize variables with $\top$
- copy and paste missing variables when joining
- flags indicate definedness
  $1 \approx$ value given by domain state
  $0 \approx$ value is unrestricted

Scenario 2: Non-initialized fields

```
int x, y;
  // [x = ⊤, y = ⊤]
  // []
if(rnd()) {
  x = 1;
  y = 2;
  // [x = 1, y = 2]
  // [x = 1, f_x = 1, y = 2, f_y = 1]
} else {
  x = 0;
  // [x = 0, y = ⊤]
  //
}
// [x ∈ {0,1}, y = { ⊤  if x = 0 ; 2, otherwise }]
//
```

Precision loss

- final state approximated in Polyhedra domain:

$$[x \in \{0,1\},\ y = \top]$$

- and we cannot copy and paste...

Solution: Avoid $\top$ as value

- do not initialize variables with $\top$
- copy and paste missing variables when joining
- flags indicate definedness
  $1 \approx$ value given by domain state
  $0 \approx$ value is unrestricted

Scenario 2: Non-initialized fields

```
int x, y;
  // [x = ⊤, y = ⊤]
  // []
if(rnd()) {
  x = 1;
  y = 2;
  // [x = 1, y = 2]
  // [x = 1, f_x = 1, y = 2, f_y = 1]
} else {
  x = 0;
  // [x = 0, y = ⊤]
  // [x = 0, f_x = 1, y = 2, f_y = 0]
}
// [x ∈ {0,1}, y = { ⊤  if x = 0 ; 2  otherwise }]
//
```

Precision loss

- final state approximated in Polyhedra domain:

$$[x \in \{0, 1\}, y = \top]$$

- and we cannot copy and paste...

Solution: Avoid $\top$ as value

- do not initialize variables with $\top$
- copy and paste missing variables when joining
- flags indicate definedness
  $1 \approx$ value given by domain state
  $0 \approx$ value is unrestricted

Scenario 2: Non-initialized fields

```
int x, y;
  // [x = ⊤, y = ⊤]
  // []
if(rnd()) {
  x = 1;
  y = 2;
  // [x = 1, y = 2]
  // [x = 1, f_x = 1, y = 2, f_y = 1]
} else {
  x = 0;
  // [x = 0, y = ⊤]
  // [x = 0, f_x = 1, y = 2, f_y = 0]
}
// [x ∈ {0,1}, y = { ⊤ if x = 0 ; 2, otherwise }]
// [x ∈ {0,1}, f_x = 1, y = 2, f_y = x]
```

Precision loss

- final state approximated in Polyhedra domain:

$$[x \in \{0,1\},\ y = \top]$$

- and we cannot copy and paste...

Solution: Avoid $\top$ as value

- do not initialize variables with $\top$
- copy and paste missing variables when joining
- flags indicate definedness
  $1 \approx$ value given by domain state
  $0 \approx$ value is unrestricted

Putting things together: The *Undefined* Domain

Functor Domain

denoted by $U(D)$ for some child domain D

state $u \rhd s \in U(D)$ where $s \in D$

Support set χ

$x \in \chi(u \rhd s) \Longleftrightarrow x,\ f_x \in \chi(s)$

Semantics

if $f_x = 1$ use value of x

if $f_x = 0$ treat x as undefined

Putting things together: The *Undefined* Domain

Functor Domain

denoted by $U(D)$ for some child domain D

state $u \triangleright s \in U(D)$ where $s \in D$

Support set χ

$x \in \chi(u \triangleright s) \iff x,\ f_x \in \chi(s)$

Semantics

if $f_x = 1$ use value of x

if $f_x = 0$ treat x as undefined

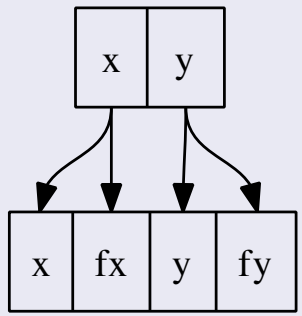
From the last example

represented state $u \triangleright s$

$$u \triangleright s \equiv [x \in \{0,1\},\ y = \begin{cases} \top & \text{if } x = 0 \\ 2, & \text{otherwise} \end{cases}]$$

child state s

$s = [x \in \{0,1\},\ f_x = 1,\ y = 2,\ f_y = x]$



Lattice Operations: Join and Compare

- make *compatible*: adjust support sets before joining and comparing
- make a "good guess" for missing values:
  we *copy and paste* values from other state
- copying values én bloc retains relational information

Comparing States: $u \rhd s \sqsubseteq u \rhd t$

- make states compatible
- then perform comparison $s \sqsubseteq t$ on child domain

Joining States: $u \rhd s \sqcup u \rhd t$

- make states compatible
- then perform join $s \sqcup t$ on child domain

Lattice Operations: Copy and Paste

Making states $u \rhd s$ and $u \rhd t$ compatible

first add variables missing in right hand side $u \rhd t$:

1. compute set $\chi_{\text{common}} := \chi(s) \cap \chi(t)$ of common variables
2. copy values missing in t from s:
   $L := drop_{\chi_{\text{common}}}(u \rhd s)$
3. paste values to t:
   $t := t \times L$
4. set flags to "undefined" for added variables:
   for each non-flag variable $x \in L$: $t := t[\![f_x := 0]\!]$

then same for left hand side

Lattice Operations: Correctness

Co-fibered Domain

- Pre-order $\sqsubseteq$ defined by Grothendieck construction:
  for each set χ of variables:

  projection $drop_\chi$

  $$u \rhd s \sqsubseteq drop_\chi(u \rhd s)$$

  family of introduction operations add_χ

  $$u \rhd s \sqsubseteq add_\chi(u \rhd s)$$

- ensures soundness of making states compatible

Lattice Operations: Correctness

Co-fibered Domain

- Pre-order $\sqsubseteq$ defined by Grothendieck construction:
  for each set χ of variables:

  projection $drop_\chi$

  $$u \rhd s \sqsubseteq drop_\chi(u \rhd s)$$

  family of introduction operations add_χ

  $$u \rhd s \sqsubseteq add_\chi(u \rhd s)$$

- ensures soundness of making states compatible

Transfer Functions 1: Assignments

assignment operation

for example $x := y + z$

(note that $+$ is a total function)

perform operation on values

on child domain: $x := y + z$

sound, because operation $+$ is total

adjust flags

on child domain: $f_x := f_y \wedge f_z$

result is undefined if an operand is undefined

Transfer Functions 1: Assignments

Caution: non-total functions (e.g. division)

```
if(rnd()) {
   x=1; y=0;
} else {
   x=0;
}
// [x = [0,1], f_x = 1, y = 0, f_y = x]
if(x==0) {

   y=3/y;

}
```

Transfer Functions 1: Assignments

Caution: non-total functions (e.g. division)

```
if(rnd()) {
   x=1; y=0;
} else {
   x=0;
}
// [x = [0,1], f_x = 1, y = 0, f_y = x]
if(x==0) {
   // [x = 0, f_x = 1, y = 0, f_y = 0]
   y=3/y;
   // [⊥]
}
```

Transfer Functions 2: Tests

Tests, for example $x + y < 0$

split child state

defined arguments

$$s_d = s[\![f_x = 1 \wedge f_y = 1]\!]$$

undefined argument

$$s_u = s[\![f_x = 0 \vee f_y = 0]\!]$$

perform test only where values are defined

$$s := s_u \sqcup s_d[\![x + y < 0]\!]$$

Efficient Implementation

Sources of Infficiency

one flag per variable, although

- many flags have constant value
- join operation introduces flags of equal value

How to avoid them

- do not store constant flags in child domain
- store only one flag for groups of equal-valued flags

Achieved improvement

ratio between number of variables and flags becomes better than 10 : 1

Experimental Results

	U	steps	time	memory	vars	add. flags	warnings
heap 1	✓	12	19	19.0	14	1	0
	✗	12	18	17.8	13	–	1
heap 2	✓	24	35	23.2	23	2	0
	✗	24	32	21.7	21	–	1
call stack 1	✓	114	450	42.0	50	2	0
	✗	76	377	41.9	48	–	7
call stack 2	✓	254	641	42.0	74	2	0
	✗	178	416	42.6	72	–	7
call stack 3	✓	153	718	42.4	66	4	0
	✗	76	422	41.9	48	–	7
call stack 4	✓	128	702	42.2	54	2	0
	✗	88	920	42.5	52	–	8
call stack 5	✓	173	1455	47.3	75	4	0
	✗	90	709	42.0	54	–	8

- the *Undefined* domains improves precision
- only a small number of flags is required

Finally...

the Undefined domain

provides a simple solution to a common problem

- enhances expressiveness of existing domains
- combination of two simple techniques:
  - mask "undefined" values by flags
  - copy and paste operation that respects relational information
- clean theoretic presentation as a co-fibered domain

related work

- recency abstraction [Balakrishnan et. al. 2006]
  most recent instance of object is kept apart from summary
- decision tree domain (Astrée [Cousot et. al., 2003])
  tree of domains, selected by flag value

future work

explore operations that retain even more relations than *copy and paste*