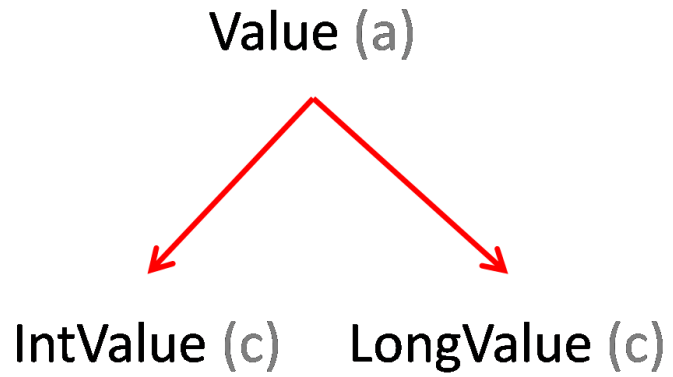
Decision Procedure for Partial Orders

Elena Sherman

September 12, 2009

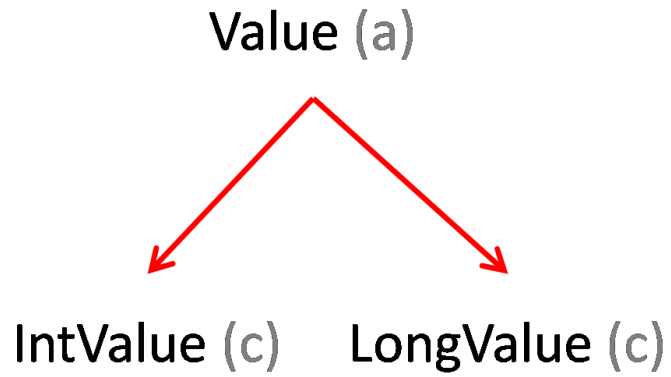
Subtyping in Kiasan

```
Value convert(Value v) {
1: long result;
2: if (v instanceof IntValue) {
3:       result = ((IntValue)v).value;
       }  else  {
4:       result = ((LongValue)v).value;
       }
5:     return new LongValue(result);
}
```



Subtyping in Kiasan

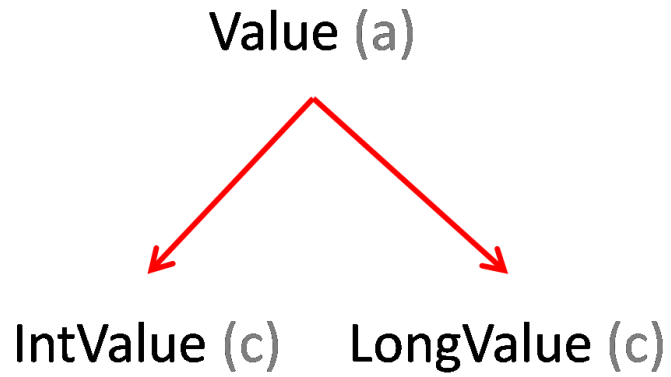
```
Value convert(Value v) {
1: long result;
2: if (v instanceof IntValue) {
3:       result = ((IntValue)v).value;
       }  else  {
4:       result = ((LongValue)v).value;
       }
5:     return new LongValue(result);
}
```



- DP = modeling of subtyping as an uninterpreted function → weak modeling → lots of false positives.

Subtyping in Kiasan

```
Value convert(Value v) {
1: long result;
2: if (v instanceof IntValue) {
3:       result = ((IntValue)v).value;
       }  else  {
4:       result = ((LongValue)v).value;
       }
5:     return new LongValue(result);
}
```

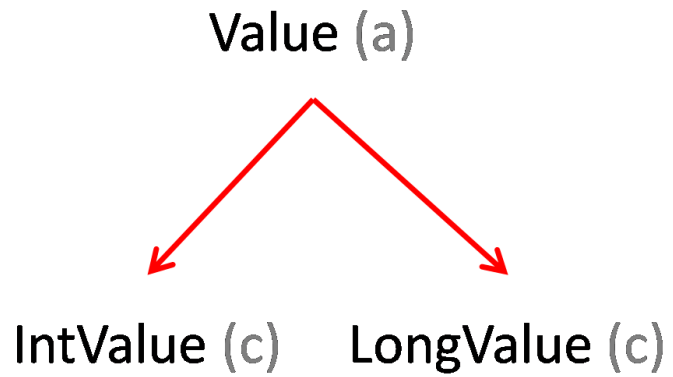


- DP = modeling of subtyping as an uninterpreted function → weak modeling → lots of false positives.
- Optimization: type is refined to the lowest successfully casted subclass → avoiding some calls to DP → few false positives.

Example of Subtyping in Kiasan

```
Value convert(Value v) {
1:   ▶long result;
2:    if (v instanceof IntValue) {
3:        result = ((IntValue)v).value;
      }  else  {
4:      result = ((LongValue)v).value;
      }
5:      return new LongValue(result);
}
```

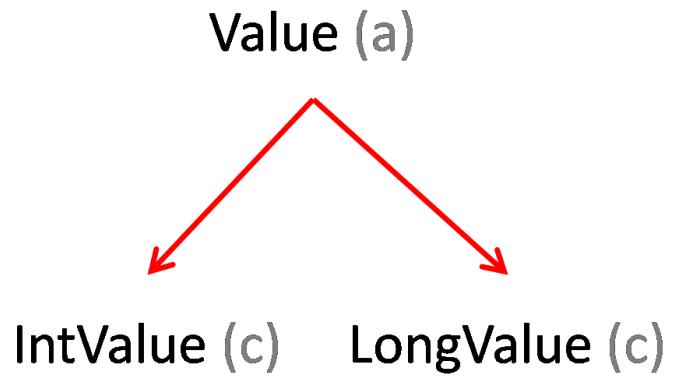
1. TypeOf(v) = Value



Example of Subtyping in Kiasan

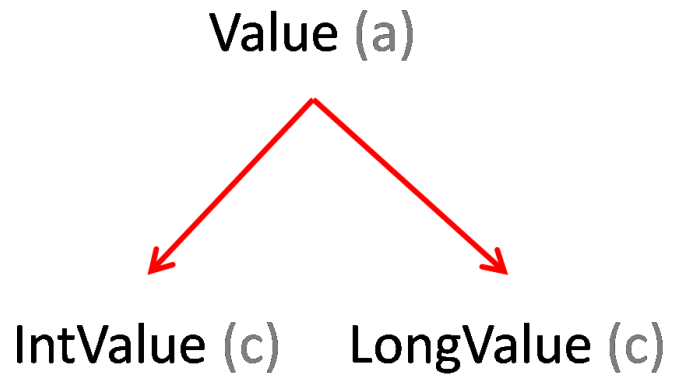
```
Value convert(Value v) {
1:    long result;
2:  ▶if (v instanceof IntValue) {  T
3:        result = ((IntValue)v).value;
      } else {
4:      result = ((LongValue)v).value;
      }
5:      return new LongValue(result);
}
```

1. TypeOf(v) = Value, TypeOf(v) = IntValue



Example of Subtyping in Kiasan

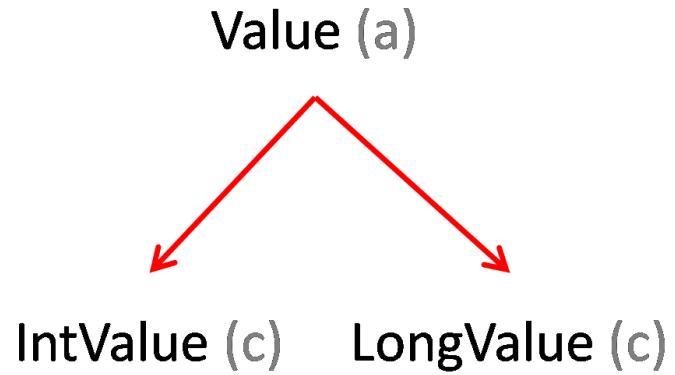
```
Value convert(Value v) {
1:    long result;
2:    if (v instanceof IntValue) {  T
3:       ►result = ((IntValue)v).value;
      }  else  {
4:      result = ((LongValue)v).value;
      }
5:      return new LongValue(result);
}
```



1. TypeOf(v) = `Value`, TypeOf(v) = `IntValue`, casting OK

Example of Subtyping in Kiasan

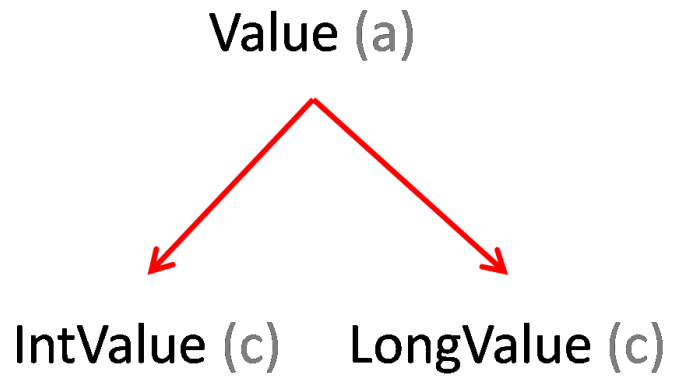
```
Value convert(Value v) {
1:    long result;
2:    if (v instanceof IntValue) {  T
3:        result = ((IntValue)v).value;
      }  else  {
4:      result = ((LongValue)v).value;
      }
5:    ►return new LongValue(result);
}
```



1. TypeOf(v) = `Value`, TypeOf(v) = `IntValue`, casting OK

Example of Subtyping in Kiasan

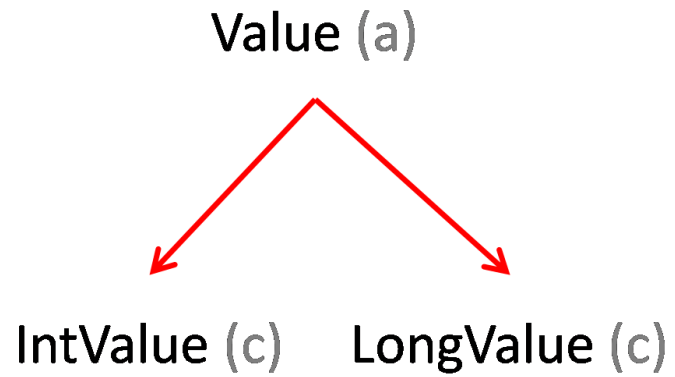
```
Value convert(Value v) {
1:    long result;
2:    if (v instanceof IntValue) {  T
3:       ►result = ((IntValue)v).value;
      }  else  {
4:      result = ((LongValue)v).value;
      }
5:      return new LongValue(result);
}
```



1. TypeOf(v) = `Value`, TypeOf(v) = `IntValue`, casting OK

Example of Subtyping in Kiasan

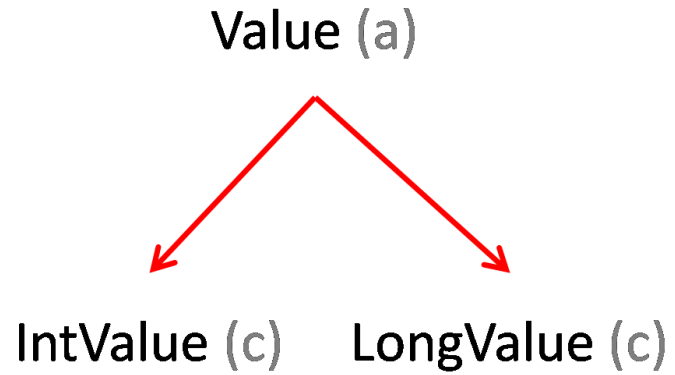
```
Value convert(Value v) {
1:    long result;
2:  ▶if (v instanceof IntValue) {  T  F
3:        result = ((IntValue)v).value;
      } else {
4:      result = ((LongValue)v).value;
      }
5:      return new LongValue(result);
}
```



1. TypeOf(v) = Value, TypeOf(v) = IntValue, casting OK
2. TypeOf(v) = Value, TypeOf(v) = Value

Example of Subtyping in Kiasan

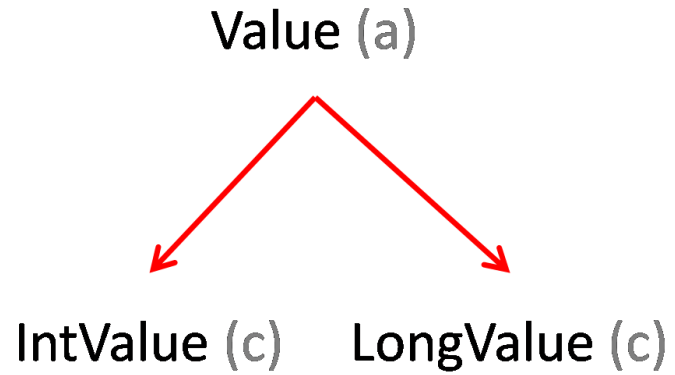
```
Value convert(Value v) {
1:    long result;
2:    if (v instanceof IntValue) {  T  F
3:        result = ((IntValue)v).value;
      } else {
4:    ►result = ((LongValue)v).value;
      }
5:      return new LongValue(result);
}
```



1. TypeOf(v) = Value, TypeOf(v) = IntValue, casting OK
2. TypeOf(v) = Value, TypeOf(v) = Value, casting EXCEPTION

Example of Subtyping in Kiasan

```
Value convert(Value v) {
1:    long result;
2:    if (v instanceof IntValue) {  T  F
3:        result = ((IntValue)v).value;
      }  else  {
4:      result = ((LongValue)v).value;
      }
5:      return new LongValue(result);
}
```



1. TypeOf(v) = `Value`, TypeOf(v) = `IntValue`, casting OK
2. TypeOf(v) = `Value`, TypeOf(v) = `Value`, casting EXCEPTION

Trace 2 produces false positive → need better DP → DP-PO.

ESQuaReD

University of Nebraska Lincoln

Theory of Partial Order (PO)

1. *Variables*: a set of variables X.
2. *Logical symbols*: $\{\wedge, \neg\}$.
3. *Nonlogical symbols*: $\{<:, E\}$, where E is the set of poset elements (constants) and $<:$ is defined as :
   1. *Reflexive*: $e <: e$ where $e \in E$
   2. *Antisymmetric* : $e_1 <: e_2$ and $e_2 <: e_1 \rightarrow e_1 = e_2$ where $e_1, e_2 \in E$.
   3. *Transitive*: $e_1 <: e_2$ and $e_2 <: e_3 \rightarrow e_1 <: e_3$ where $e_1, e_2, e_3 \in E$.
4. *Syntax*:

$$formula: \quad formula \wedge formula \quad | \quad atom \quad | \quad \neg \quad atom$$

$$atom: \quad x <: e \quad | \quad e <: x \quad | \quad e_1 <: e_2$$

where $x \in X$ and $e, e_1, e_2 \in E$.

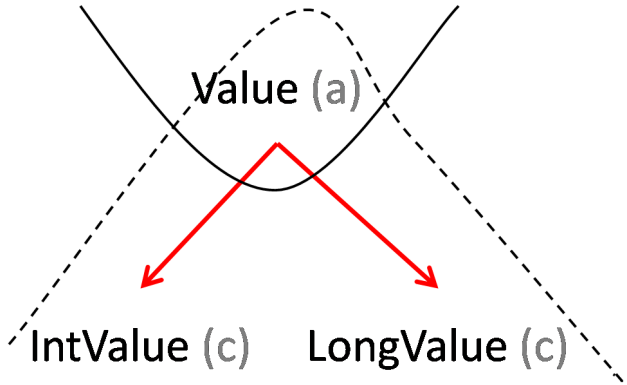
ESQuaReD

University of Nebraska Lincoln

Example of Subtyping using DP-PO

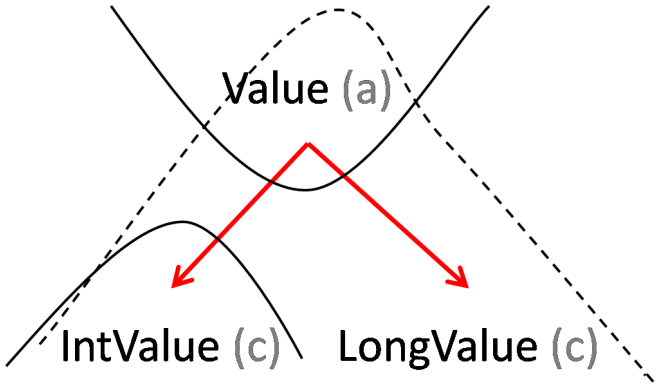
```
Value convert(Value v) {
1:   ▶long result;
2:    if (v instanceof IntValue) {
3:       result = ((IntValue)v).value;
      }  else  {
4:      result = ((LongValue)v).value;
      }
5:      return new LongValue(result);
}
```

1. $v <: \mathit{Value}, \neg(\mathit{Value} <: v) \rightarrow$ SAT



Example of Subtyping using DP-PO

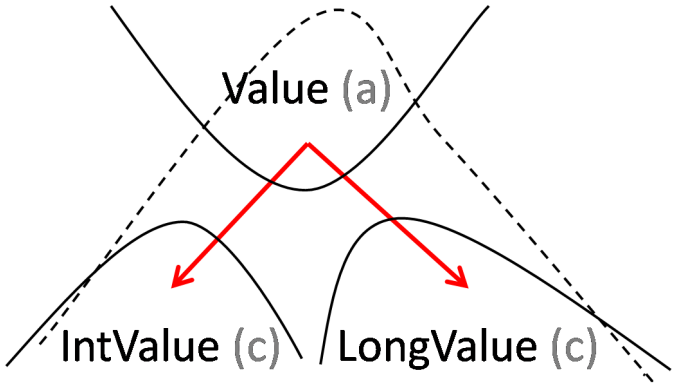
```
Value convert(Value v) {
1:    long result;
2:   ▶if (v instanceof IntValue) {   F
3:       result = ((IntValue)v).value;
       }  else  {
4:       result = ((LongValue)v).value;
       }
5:       return new LongValue(result);
}
```



1. $v <: \mathit{Value}, \neg(\mathit{Value} <: v) \rightarrow$ SAT
2. $\neg(v <: \mathit{IntValue}) \rightarrow$ SAT

Example of Subtyping using DP-PO

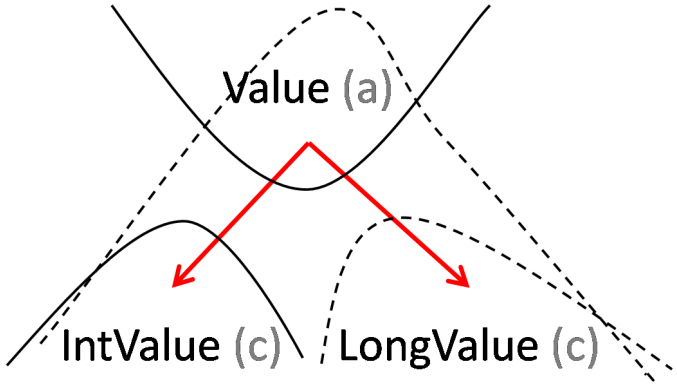
```
Value convert(Value v) {
1:    long result;
2:    if (v instanceof IntValue) {  F
3:       result = ((IntValue)v).value;
      }  else  {
4:    ▶result = ((LongValue)v).value;
      }
5:      return new LongValue(result);
}
```



1. $v <: \mathit{Value}, \neg(\mathit{Value} <: v) \rightarrow$ SAT
2. $\neg(v <: \mathit{IntValue}) \rightarrow$ SAT
3. $\neg(v <: \mathit{LongValue}) \rightarrow$ UNSAT

Example of Subtyping using DP-PO

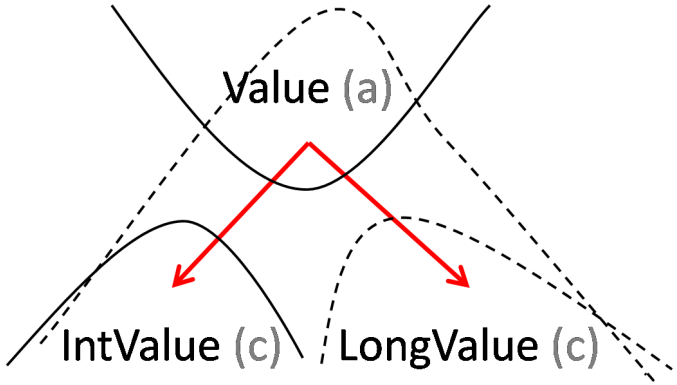
```
Value convert(Value v) {
1:    long result;
2:    if (v instanceof IntValue) {  F
3:       result = ((IntValue)v).value;
      }  else  {
4:    ▶result = ((LongValue)v).value;
      }
5:      return new LongValue(result);
}
```



1. $v <: \mathit{Value}, \neg(\mathit{Value} <: v) \rightarrow$ SAT
2. $\neg(v <: \mathit{IntValue}) \rightarrow$ SAT
3. $\neg(v <: \mathit{LongValue}) \rightarrow$ UNSAT
4. $v <: \mathit{LongValue} \rightarrow$ SAT

Example of Subtyping using DP-PO

```
Value convert(Value v) {
1:    long result;
2:    if (v instanceof IntValue) {  F
3:       result = ((IntValue)v).value;
      }  else  {
4:      result = ((LongValue)v).value;
      }
5:    ▶return new LongValue(result);
}
```

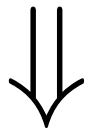


1. $v <: \mathit{Value}, \neg(\mathit{Value} <: v) \rightarrow$ SAT
2. $\neg(v <: \mathit{IntValue}) \rightarrow$ SAT
3. $\neg(v <: \mathit{LongValue}) \rightarrow$ UNSAT
4. $v <: \mathit{LongValue} \rightarrow$ SAT

The false positive exception is eliminated.

Design

- No $x_1 <: x_2$ constraint type $\Rightarrow$ variables are independent.
- Preserve clarity.
- Keep only necessary information for each variable.



- Each variable x_i has its own set of elements satisfying constraints associated with x_i.
- Create separate sets of elements for each of 4 constraint types: $x <: e$, $e <: x$, $\neg \quad (x <: e)$, $\neg \quad (e <: x)$.
- Keep only "front line" poset elements in a corresponding constraint type set - take advantage of the transitivity property.

sup Set

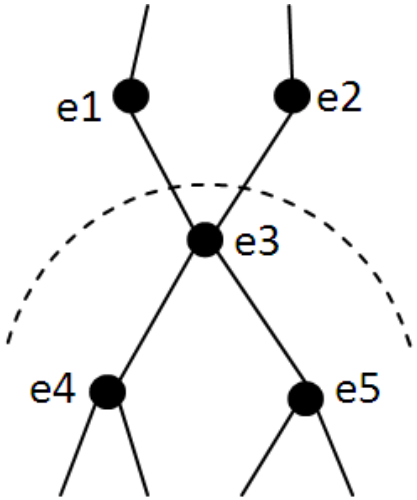
sup

$$sup(x) = \{e | \forall e_c, \forall e_d \in D(e) : e_d <: e_c \\ \wedge \quad \exists e_c, \forall e_a \in A(e) \setminus e : e_a \not<: e_c\}$$

where e_c is a poset element in a $x <: e_c$ constraint, $D(e)$ - descendants of e and $A(e)$ - ancestors of e (include itself).

Suppose we have:
$x <: e_1, x <: e_2, x <: e_3$,
so that $e_c \in \{e_1, e_2, e_3\}$

$$\Rightarrow sup(x) = \{e_3\}$$



ESQuaReD

University of Nebraska Lincoln

sup_not Set

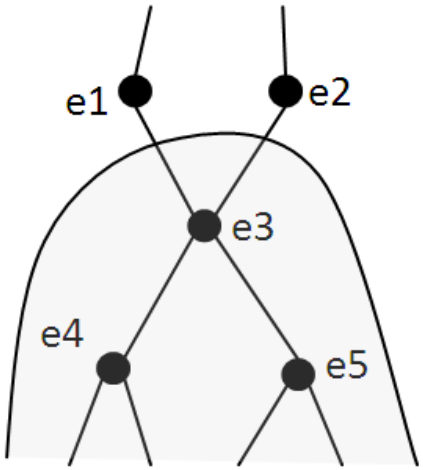
sup_not

$$sup\_not(x) = \{e |\ \forall e_c, \forall e_a \in A(e) \setminus e : \neg(e_a <: e_c) \\ \wedge \quad \exists e_c, \forall e_d \in D(e) : \neg(e_d \not<: e_c)\}$$

where e_c is a poset element in a $\neg(x <: e_c)$ constraint.

Suppose we have:
$\neg(x <: e_3), \neg(x <: e_4)$,
so that $e_c \in \{e_3, e_4\}$

$$\Rightarrow sup\_not(x) = \{e_3\}$$



sub Set

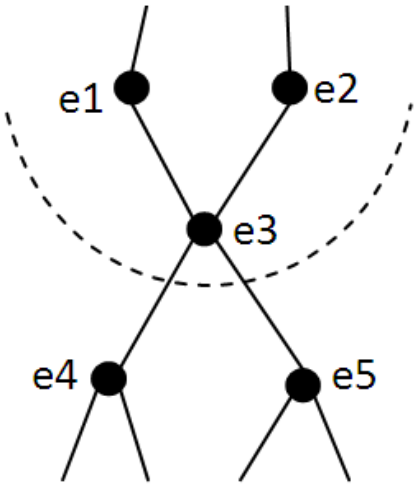
sub

$$sub(x) = \{e | \forall e_c, \forall e_a \in A(e) : e_c <: e_a \quad \wedge \quad \exists e_c, \forall e_d \in D(e) \backslash e : e_d \not<: e_c\}$$

where e_c is a poset element in a $e_c <: x$ constraint.

Suppose previously we have:
$e_4 <: x, e_5 <: x, e_3 <: x$,
so that $e_c \in \{e_3, e_4, e_5\}$

$$\Rightarrow sub(x) = \{e_3\}$$



sub_not Set

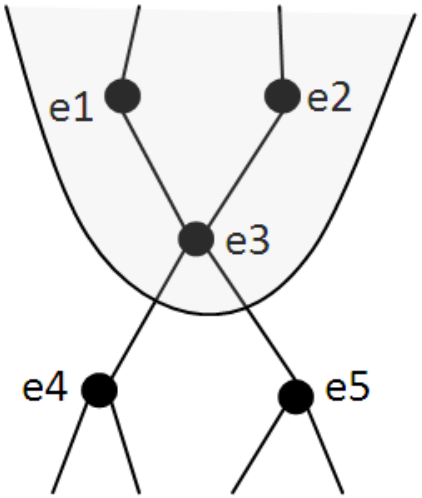
sub_not

$$sub\_not(x) = \{e|\ \forall e_c, \forall e_d \in D(e) \setminus e : \neg(e_c <: e_d)\}$$
$$\wedge \quad \exists e_c, \forall e_a \in A(e) : \neg(e_c \not<: e_a)$$

where e_c is a poset element in a $\neg(e_c <: x)$ constraint.

Suppose previously we have:
$\neg(e_1 <: x), \neg(e_3 <: x)$,
so that $e_c \in \{e_1, e_3\}$

$$\Rightarrow sub\_not(x) = \{e_3\}$$



SAT / UNSAT

Using these 4 sets we can determine if a *formula* is SAT or UNSAT:

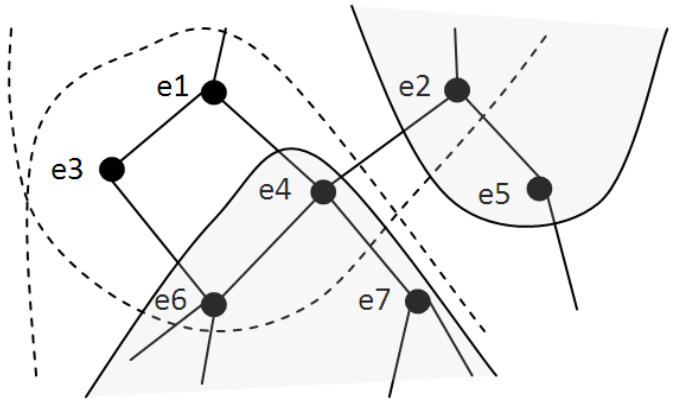
SAT / UNSAT

A *formula* is SAT when $\forall x_i \in X \exists$ a path p from $e_{sup} \in sup(x_i)$ to $e_{sub} \in sub(x_i)$ and $\exists e_p \in p$ such that $e_p \notin D(e_{sup\_not}) \wedge e_p \notin A(e_{sub\_not})$.

Example

$$formula = x <: e_1 \wedge \neg(x <: e_4) \wedge e_6 <: x \wedge \neg(e_5 <: x)$$

1. $x <: e_1 \Rightarrow sup(x) = e_1$
2. $\neg(x <: e_4) \Rightarrow sup\_not(x) = e_4$
3. $e_6 <: x \Rightarrow sub(x) = e_6$
4. $\neg(e_5 <: x) \Rightarrow sub\_not(x) = e_5$

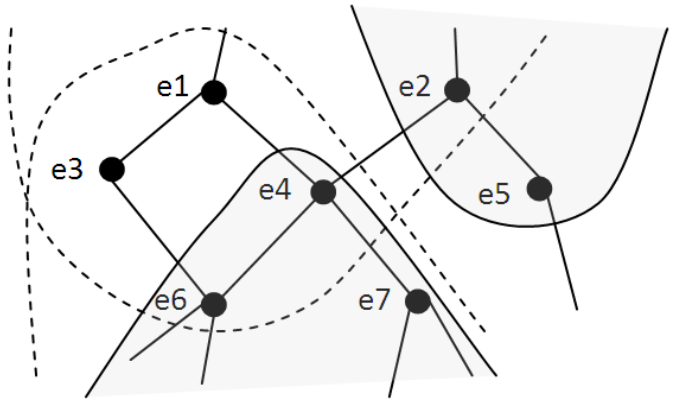


Example

$$formula = x <: e_1 \wedge \neg(x <: e_4) \wedge e_6 <: x \wedge \neg(e_5 <: x)$$

1. $x <: e_1 \Rightarrow sup(x) = e_1$
2. $\neg(x <: e_4) \Rightarrow sup\_not(x) = e_4$
3. $e_6 <: x \Rightarrow sub(x) = e_6$
4. $\neg(e_5 <: x) \Rightarrow sub\_not(x) = e_5$

- Find a path from e_1 to e_6
  $p = (e_1, e_3, e_6)$

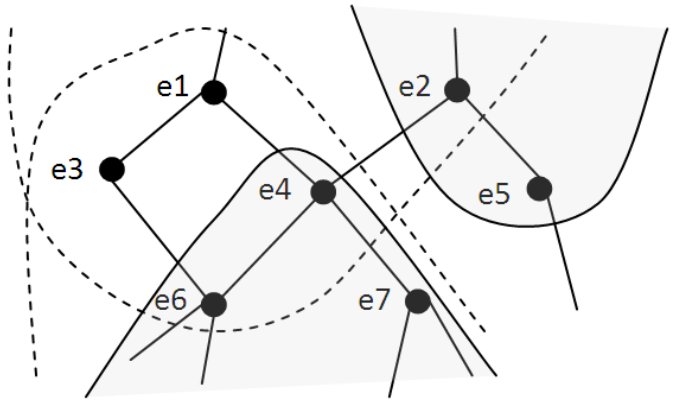


Example

$formula = x <: e_1 \wedge \neg(x <: e_4) \wedge e_6 <: x \wedge \neg(e_5 <: x)$

1. $x <: e_1 \Rightarrow sup(x) = e_1$
2. $\neg(x <: e_4) \Rightarrow sup\_not(x) = e_4$
3. $e_6 <: x \Rightarrow sub(x) = e_6$
4. $\neg(e_5 <: x) \Rightarrow sub\_not(x) = e_5$

- Find a path from e_1 to e_6
  $p = (e_1, e_3, e_6)$
- Is $e_6 \in D(e_4)$?

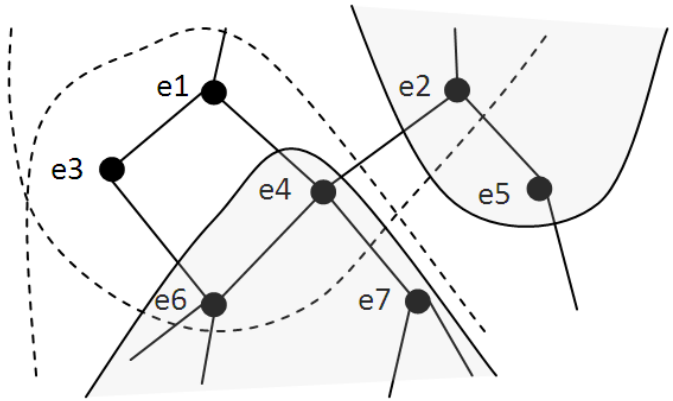


Example

$$formula = x <: e_1 \wedge \neg(x <: e_4) \wedge e_6 <: x \wedge \neg(e_5 <: x)$$

1. $x <: e_1 \Rightarrow sup(x) = e_1$
2. $\neg(x <: e_4) \Rightarrow sup\_not(x) = e_4$
3. $e_6 <: x \Rightarrow sub(x) = e_6$
4. $\neg(e_5 <: x) \Rightarrow sub\_not(x) = e_5$

- Find a path from e_1 to e_6
  $p = (e_1, e_3, e_6)$
- Is $e_6 \in D(e_4)$? YES

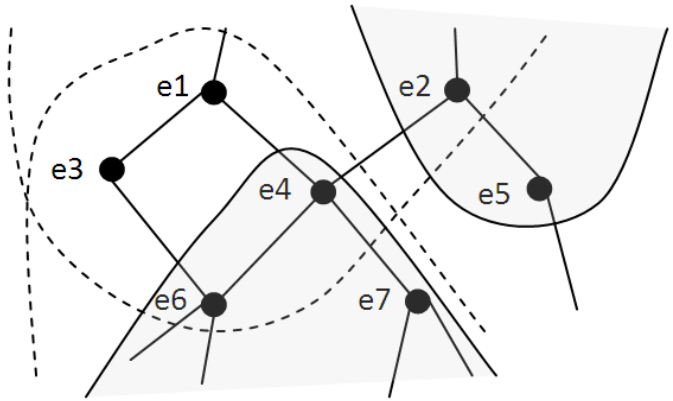


Example

$formula = x <: e_1 \wedge \neg(x <: e_4) \wedge e_6 <: x \wedge \neg(e_5 <: x)$

1. $x <: e_1 \Rightarrow sup(x) = e_1$
2. $\neg(x <: e_4) \Rightarrow sup\_not(x) = e_4$
3. $e_6 <: x \Rightarrow sub(x) = e_6$
4. $\neg(e_5 <: x) \Rightarrow sub\_not(x) = e_5$

- Find a path from e_1 to e_6
  $p = (e_1, e_3, e_6)$
- Is $e_6 \in D(e_4)$? YES
- Is $e_3 \in D(e_4)$?

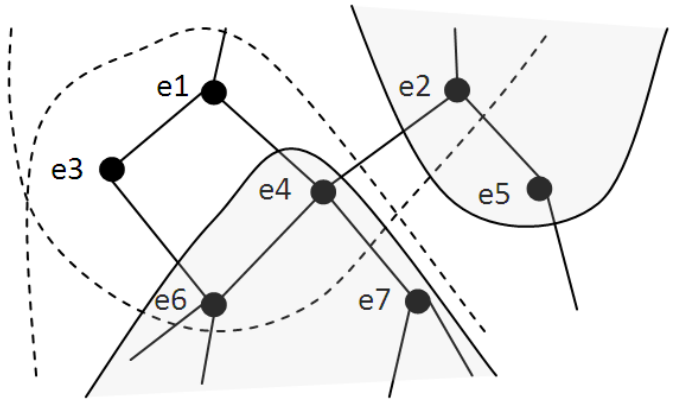


Example

$formula = x <: e_1 \wedge \neg(x <: e_4) \wedge e_6 <: x \wedge \neg(e_5 <: x)$

1. $x <: e_1 \Rightarrow sup(x) = e_1$
2. $\neg(x <: e_4) \Rightarrow sup\_not(x) = e_4$
3. $e_6 <: x \Rightarrow sub(x) = e_6$
4. $\neg(e_5 <: x) \Rightarrow sub\_not(x) = e_5$

- Find a path from e_1 to e_6
  $p = (e_1, e_3, e_6)$
- Is $e_6 \in D(e_4)$? YES
- Is $e_3 \in D(e_4)$? NO

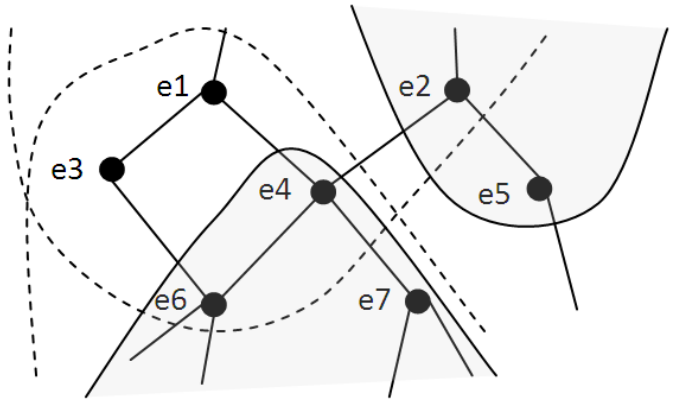


Example

$formula = x <: e_1 \wedge \neg(x <: e_4) \wedge e_6 <: x \wedge \neg(e_5 <: x)$

1. $x <: e_1 \Rightarrow sup(x) = e_1$
2. $\neg(x <: e_4) \Rightarrow sup\_not(x) = e_4$
3. $e_6 <: x \Rightarrow sub(x) = e_6$
4. $\neg(e_5 <: x) \Rightarrow sub\_not(x) = e_5$

- Find a path from e_1 to e_6
  $p = (e_1, e_3, e_6)$
- Is $e_6 \in D(e_4)$? YES
- Is $e_3 \in D(e_4)$? NO
- Is $e_3 \in A(e_5)$? NO $\rightarrow$ SAT



Future Work

- Better/other incremental reasoning.
- Better data structure for efficient descendant/ascendant querying.
- Considering a fragment for Java type hierarchy.
- Separating special cases, e.g. some of the sets are empty.