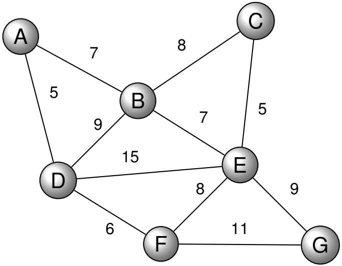
# Preventing Arithmetic Overflows in Alloy

**Aleksandar Milicevic** (aleks@csail.mit.edu)
Daniel Jackson (dnj@csail.mit.edu)

Software Design Group
Massachusetts Institute of Technology
Cambridge, MA

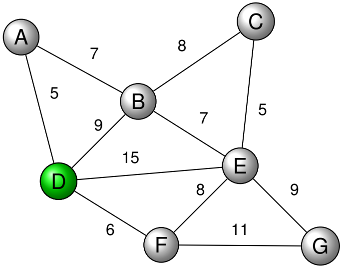
International Conference of Alloy, ASM, B, VDM, and Z Users
Pisa, Italy, June 2012

# Checking Prim's Algorithm




**Prim's algorithm for finding minimum spanning tree in a graph**

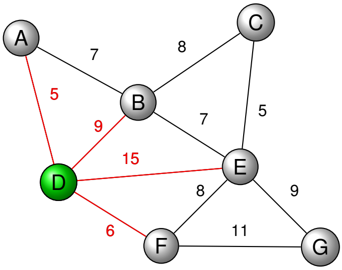
# Checking Prim's Algorithm




**Prim's algorithm for finding minimum spanning tree in a graph**

- select an arbitrary node to start with

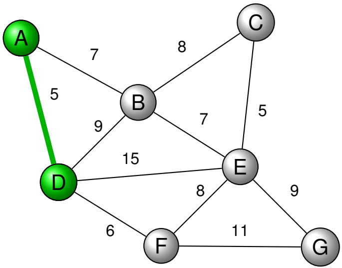
# Checking Prim's Algorithm




**Prim's algorithm for finding minimum spanning tree in a graph**

- select an arbitrary node to start with
- find edges from selected to unselected nodes

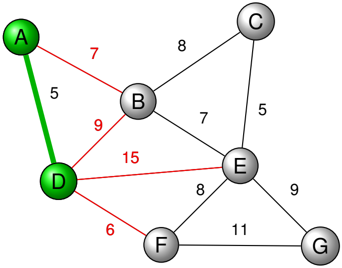
# Checking Prim's Algorithm




**Prim's algorithm for finding minimum spanning tree in a graph**

- select an arbitrary node to start with
- find edges from selected to unselected nodes
- select the edge with the smallest weight

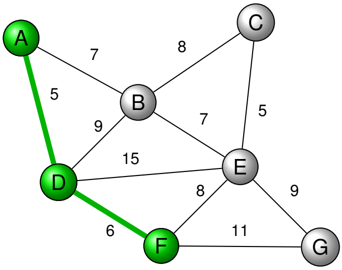
# Checking Prim's Algorithm




## Prim's algorithm for finding minimum spanning tree in a graph

- select an arbitrary node to start with
- find edges from selected to unselected nodes
- select the edge with the smallest weight
- repeat until all nodes have been selected

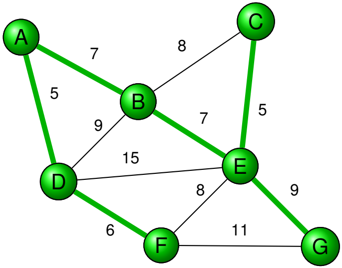
# Checking Prim's Algorithm




**Prim's algorithm for finding minimum spanning tree in a graph**

- select an arbitrary node to start with
- find edges from selected to unselected nodes
- select the edge with the smallest weight
- repeat until all nodes have been selected

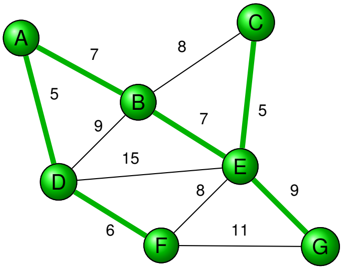
# Checking Prim's Algorithm




## Prim's algorithm for finding **minimum spanning tree** in a graph

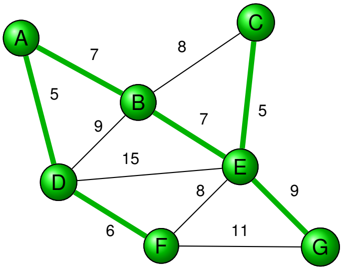
- select an arbitrary node to start with
- find edges from selected to unselected nodes
- select the edge with the smallest weight
- repeat until all nodes have been selected

# Checking Prim's Algorithm



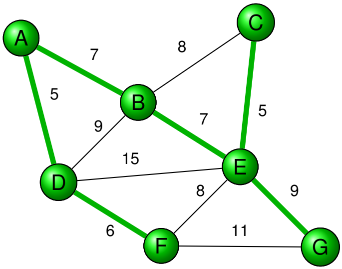

```
sig Node {}
sig Edge {
  weight: Int,
  nodes: set Node,

} {
  weight >= 0 && #nodes = 2
}
```

# Checking Prim's Algorithm




```
open util/ordering[Time]
sig Time {}
sig Node {}
sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 && #nodes = 2
}
```

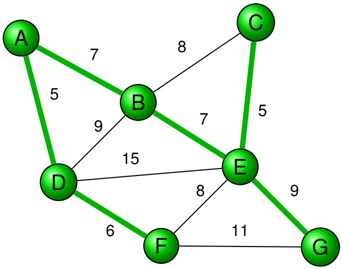
# Checking Prim's Algorithm




```
open util/ordering[Time]
sig Time {}
sig Node {}
sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 && #nodes = 2
}
```

```
fact prim { /* model of execution of Prim's algorithm */ }
```

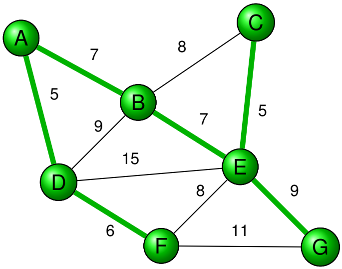
# Checking Prim's Algorithm




```
open util/ordering[Time]
sig Time {}
sig Node {}
sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 && #nodes = 2
}
```

```
fact prim { /* model of execution of Prim's algorithm */ }
pred spanningTree(edges: set Edges) { /* checks whether a given set of
                                         edges forms a spanning tree */ }
```

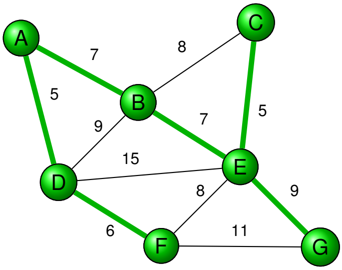
# Checking Prim's Algorithm




```
open util/ordering[Time]
sig Time {}
sig Node {}
sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 && #nodes = 2
}
```

```
fact prim { /* model of execution of Prim's algorithm */ }

pred spanningTree(edges: set Edges) { /* checks whether a given set of
                                          edges forms a spanning tree */ }

/* no set of edges is a spanning tree with a smaller total weight
   than the one returned by Prim's algorithm */
smallest: check {
  no edges: set Edge {
    spanningTree[edges]
    (sum e: edges | e.weight) < (sum e: chosen.last | e.weight)}}
```

# Checking Prim's Algorithm

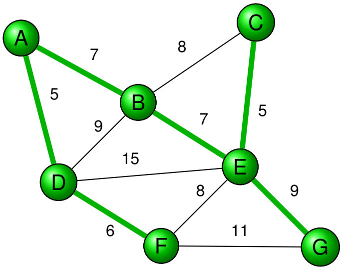



```
open util/ordering[Time]
sig Time {}
sig Node {}
sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 && #nodes = 2
}
```

```
fact prim { /* model of execution of Prim's algorithm */ }

pred spanningTree(edges: set Edges) { /* checks whether a given set of
                                         edges forms a spanning tree */ }

/* no set of edges is a spanning tree with a smaller total weight
   than the one returned by Prim's algorithm */
smallest: check {
  no edges: set Edge {
    spanningTree[edges]
    (sum e: edges | e.weight) < (sum e: chosen.last | e.weight)}}
```

counterexample: `leftSum = -5; rightSum = 24`

# Checking Prim's Algorithm

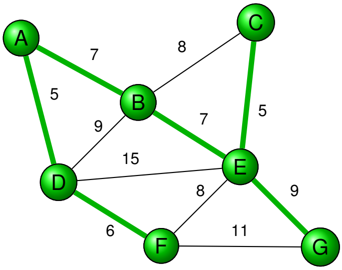



```
open util/ordering[Time]
sig Time {}
sig Node {}
sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 && #nodes = 2
}
```

```
fact prim { /∗ model of execution of Prim's algorithm ∗/ }

pred spanningTree(edges: set Edges) { /∗ checks whether a given set of
                                          edges forms a spanning tree ∗/ }

/∗ no set of edges is a spanning tree with a smaller total weight
   than the one returned by Prim's algorithm ∗/
smallest: check {
  no edges: set Edge {
    spanningTree[edges] and (sum e: edges | e.weight) > 0
    (sum e: edges | e.weight) < (sum e: chosen.last | e.weight)}}
```

counterexample: `leftSum = -5; rightSum = 24`

# Checking Prim’s Algorithm

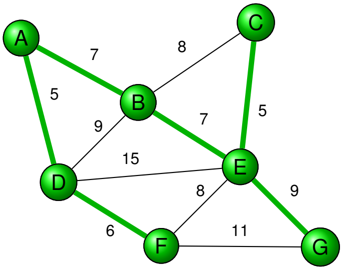



```
open util/ordering[Time]
sig Time {}
sig Node {}
sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 && #nodes = 2
}
```

```
fact prim { /∗ model of execution of Prim’s algorithm ∗/ }

pred spanningTree(edges: set Edges) { /∗ checks whether a given set of
                                          edges forms a spanning tree ∗/ }

/∗ no set of edges is a spanning tree with a smaller total weight
   than the one returned by Prim’s algorithm ∗/
smallest: check {
  no edges: set Edge {
    spanningTree[edges] and (sum e: edges | e.weight) > 0
    (sum e: edges | e.weight) < (sum e: chosen.last | e.weight)}}
```

counterexample: `leftSum = 2; rightSum = 28`

# Checking Prim's Algorithm




```
open util/ordering[Time]
sig Time {}
sig Node {}
sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 && #nodes = 2
}
```

```
fact prim { /* model of execution of Prim's algorithm */ }

pred spanningTree(edges: set Edges) { /* checks whether a given set of
                                          edges forms a spanning tree */ }

/* no set of edges is a spanning tree with a smaller total weight
   than the one returned by Prim's algorithm */
smallest: check {
  no edges: set Edge {
    spanningTree[edges] and (sum e: edges | e.weight) > 0
    (sum e: edges | e.weight) < (sum e: chosen.last | e.weight)}}
```

causes arithmetic overflows!

# Soundness of Alloy

**reason for overflows**

- wraparound semantics for arithmetic operations

  `Int = {-4, -3, ..., 2, 3}` $\Longrightarrow$ `3 + 1 = -4`

# Soundness of Alloy

**reason for overflows**

- wraparound semantics for arithmetic operations

  `Int = {-4, -3, ..., 2, 3}` $\Longrightarrow$ `3 + 1 = -4`

**alloy**

- first order relational modeling language

**the alloy analyzer**

- fully automated, bounded model finder for alloy

# Soundness of Alloy

## reason for overflows

- wraparound semantics for arithmetic operations

  `Int = {-4, -3, ..., 2, 3}` $\Longrightarrow$ `3 + 1 = -4`

## alloy

- first order relational modeling language

## the alloy analyzer

- fully automated, bounded model finder for alloy

## consequences of the bounded analysis

- not sound with respect to proof
  - → if no counterexample is found, one may still exist in a larger scope

# Soundness of Alloy

## reason for overflows

- wraparound semantics for arithmetic operations

  `Int = {-4, -3, ..., 2, 3}` $\Longrightarrow$ `3 + 1 = -4`

## alloy

- first order relational modeling language

## the alloy analyzer

- fully automated, bounded model finder for alloy

## consequences of the bounded analysis

- not sound with respect to proof
  - → if no counterexample is found, one may still exist in a larger scope
- not sound w.r.t. counterexamples when integers are used
  - → arithmetic operations can overflow ⇒ spurious counterexamples

# Soundness of Alloy

## reason for overflows

- wraparound semantics for arithmetic operations

  `Int = {-4, -3, ..., 2, 3}` $\Longrightarrow$ `3 + 1 = -4`

## alloy

- first order relational modeling language

## the alloy analyzer

- fully automated, bounded model finder for alloy

## consequences of the bounded analysis

- not sound with respect to proof
  - → if no counterexample is found, one may still exist in a larger scope
- not sound w.r.t. counterexamples when integers are used
  - → arithmetic operations can overflow ⇒ spurious counterexamples
- sound w.r.t. counterexamples if no integers are used
  - → i.e., if a counterexample is found, the property does not hold
  - → **reason**: relational operators are closed under finite universe

# Goal & Approach

## goal

- eliminate spurious counterexamples caused by overflows

  → makes the analyzer sound w.r.t. to counterexamples

# Goal & Approach

## goal

- eliminate spurious counterexamples caused by overflows

  → makes the analyzer sound w.r.t. to counterexamples

## idea

- treat arithmetic operations that overflow as undefined ($\perp$)
- use a standard 3-valued logic for boolean propositions [VDM]

$$\texttt{true} \wedge \perp = \perp, \quad \texttt{false} \wedge \perp = \texttt{false}, \quad \ldots$$

- change the semantics of quantifiers

$$[\![\texttt{all}\ \ x\texttt{:}\ \texttt{Int}\ \texttt{|}\ p(x)]\!] \ = \forall x \in \texttt{Int} \bullet (p(x) = \perp) \vee p(x)$$
$$[\![\texttt{some}\ x\texttt{:}\ \texttt{Int}\ \texttt{|}\ p(x)]\!] \ = \exists x \in \texttt{Int} \bullet (p(x) \neq \perp) \wedge p(x)$$

# Goal & Approach

## goal

- eliminate spurious counterexamples caused by overflows

  → makes the analyzer sound w.r.t. to counterexamples

## idea

- treat arithmetic operations that overflow as undefined ($\perp$)
- use a standard 3-valued logic for boolean propositions [VDM]

$$\texttt{true} \wedge \perp = \perp, \quad \texttt{false} \wedge \perp = \texttt{false}, \quad \ldots$$

- change the semantics of quantifiers

$$\begin{aligned} [\![\texttt{all}\ \ x\texttt{: Int | } p(x)]\!] &= \forall x \in \texttt{Int} \bullet (p(x) = \perp) \vee p(x) \\ [\![\texttt{some}\ x\texttt{: Int | } p(x)]\!] &= \exists x \in \texttt{Int} \bullet (p(x) \neq \perp) \wedge p(x) \end{aligned}$$

- **result**: returned models are always defined

# Goal & Approach

## goal

- eliminate spurious counterexamples caused by overflows

  → makes the analyzer sound w.r.t. to counterexamples

## idea

- treat arithmetic operations that overflow as undefined ($\perp$)
- use a standard 3-valued logic for boolean propositions [VDM]

$$\texttt{true} \land \perp = \perp, \quad \texttt{false} \land \perp = \texttt{false}, \quad \ldots$$

- change the semantics of quantifiers

$$\begin{aligned} [\![\texttt{all}\ \ x\texttt{: Int | } p(x)]\!] &= \forall x \in \texttt{Int} \bullet (p(x) = \perp) \lor p(x) \\ [\![\texttt{some}\ x\texttt{: Int | } p(x)]\!] &= \exists x \in \texttt{Int} \bullet (p(x) \neq \perp) \land p(x) \end{aligned}$$

- **result**: returned models are always defined

**challenge**: translation to existing SAT-based engine

# Example Using the New Semantics

## semantics of quantifiers

$$[\![\texttt{all}\ \ x\texttt{: Int | } p(x)]\!] = \forall x \in \texttt{Int} \bullet (p(x) = \bot) \vee p(x)$$
$$[\![\texttt{some}\ x\texttt{: Int | } p(x)]\!] = \exists x \in \texttt{Int} \bullet (p(x) \neq \bot) \wedge p(x)$$

## example

```
pred p[x, y: Int] {
  x > 0 && y > 0 => x.plus[y] > 0 }

check { all x, y: Int | p[x, y] }
for 3 Int
```

**scope**

`Int = {-4, -3, ..., 2, 3}`

# Example Using the New Semantics

## semantics of quantifiers

$$[\![\texttt{all}\ \ x\texttt{: Int | } p(x)]\!] = \forall x \in \texttt{Int} \bullet (p(x) = \perp) \vee p(x)$$
$$[\![\texttt{some}\ x\texttt{: Int | } p(x)]\!] = \exists x \in \texttt{Int} \bullet (p(x) \neq \perp) \wedge p(x)$$

## example

```
pred p[x, y: Int] {
  x > 0 && y > 0 => x.plus[y] > 0 }

check { all x, y: Int | p[x, y] }
for 3 Int
```

**scope**

`Int = {-4, -3, ..., 2, 3}`

## interpretation

$$p(-4,-4) \quad \cdots \quad p(1,1) \quad \cdots \quad p(3,3)$$

# Example Using the New Semantics

## semantics of quantifiers

$$[\![\texttt{all}\ \ x\texttt{: Int}\ |\ p(x)]\!] \;=\; \forall x \in \texttt{Int} \bullet (p(x) = \perp) \vee p(x)$$
$$[\![\texttt{some}\ x\texttt{: Int}\ |\ p(x)]\!] \;=\; \exists x \in \texttt{Int} \bullet (p(x) \neq \perp) \wedge p(x)$$

## example

```
pred p[x, y: Int] {
  x > 0 && y > 0 => x.plus[y] > 0 }

check { all x, y: Int | p[x, y] }
for 3 Int
```

**scope**

`Int = {-4, -3, ..., 2, 3}`

## interpretation

$$p(-4,-4) \quad \cdots \quad p(1,1) \quad \cdots \quad p(3,3)$$

$$\downarrow$$

$p(x,y) = \perp$ : ✗

$p(x,y)$ : ✔

# Example Using the New Semantics

## semantics of quantifiers

$$[\![\texttt{all}\ \ x\texttt{: Int | } p(x)]\!] \ = \forall x \in \texttt{Int} \bullet (p(x) = \perp) \vee p(x)$$
$$[\![\texttt{some}\ x\texttt{: Int | } p(x)]\!] \ = \exists x \in \texttt{Int} \bullet (p(x) \neq \perp) \wedge p(x)$$

## example

```
pred p[x, y: Int] {
  x > 0 && y > 0 => x.plus[y] > 0 }

check { all x, y: Int | p[x, y] }
for 3 Int
```

**scope**

`Int = {-4, -3, ..., 2, 3}`

## interpretation

$$\begin{array}{rccccc} & p(-4,-4) & \cdots & p(1,1) & \cdots & p(3,3) \\ & \downarrow & & \downarrow & & \\ p(x,y) = \perp : & ✗ & & ✗ & & \\ & & \wedge \cdots \wedge & & & \\ p(x,y) : & ✔ & & ✔ & & \end{array}$$

# Example Using the New Semantics

## semantics of quantifiers

$$[\![\texttt{all}\ \ x\texttt{: Int | } p(x)]\!] \ = \forall x \in \texttt{Int} \bullet (p(x) = \bot) \vee p(x)$$
$$[\![\texttt{some}\ x\texttt{: Int | } p(x)]\!] \ = \exists x \in \texttt{Int} \bullet (p(x) \neq \bot) \wedge p(x)$$

## example

```
pred p[x, y: Int] {
  x > 0 && y > 0 => x.plus[y] > 0 }

check { all x, y: Int | p[x, y] }
for 3 Int
```

**scope**

`Int = {-4, -3, ..., 2, 3}`

## interpretation

| | $p(-4,-4)$ | $\cdots$ | $p(1,1)$ | $\cdots$ | $p(3,3)$ |
|---|---|---|---|---|---|
| | $\downarrow$ | | $\downarrow$ | | $\downarrow$ |
| $p(x,y) = \bot$ : | ✗ | $\wedge \cdots \wedge$ | ✗ | $\wedge \cdots \wedge$ | ✔ |
| $p(x,y)$ : | ✔ | | ✔ | | ✗ |

# Example Using the New Semantics

## semantics of quantifiers

$$[\![\texttt{all}\ \ x\texttt{: Int | }p(x)]\!] = \forall x \in \texttt{Int} \bullet (p(x) = \perp) \vee p(x)$$

$$[\![\texttt{some}\ x\texttt{: Int | }p(x)]\!] = \exists x \in \texttt{Int} \bullet (p(x) \neq \perp) \wedge p(x)$$

## example

```
pred p[x, y: Int] {
  x > 0 && y > 0 => x.plus[y] > 0 }

check { all x, y: Int | p[x, y] }
for 3 Int
```

**scope**

`Int = {-4, -3, ..., 2, 3}`

## interpretation

| | $p(-4,-4)$ | | $p(1,1)$ | | $p(3,3)$ | |
|---|---|---|---|---|---|---|
| | $\downarrow$ | | $\downarrow$ | | $\downarrow$ | |
| $p(x,y) = \perp$ : | ✗ | $\wedge \cdots \wedge$ | ✗ | $\wedge \cdots \wedge$ | ✔ | = true |
| $p(x,y)$ : | ✔ | | ✔ | | ✗ | |

# Implementation Challenges

## implementation options

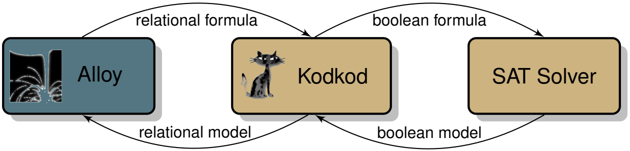
- enumerate values of bound variables and evaluate quantifiers
  - → extremely inefficient

# Implementation Challenges

## implementation options

- ~~enumerate values of bound variables and evaluate quantifiers~~
  - → ~~extremely inefficient~~
- directly encode to SAT
  - → 3-valued logic must be used throughout
  - → 2 bits required to represent 1 boolean variable
  - → likely to adversely affect models that don't involve integers

## alloy architecture

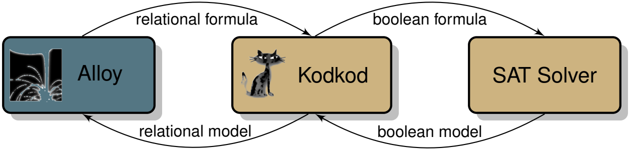


# Implementation Challenges

## implementation options

- ~~enumerate values of bound variables and evaluate quantifiers~~
  - → ~~extremely inefficient~~
- ~~directly encode to SAT~~
  - → ~~3-valued logic must be used throughout~~
  - → ~~2 bits required to represent 1 boolean variable~~
  - → ~~likely to adversely affect models that don't involve integers~~
- translate to classical logic and existing SAT-based back-end
  - → models without integers remain unaffected

## alloy architecture

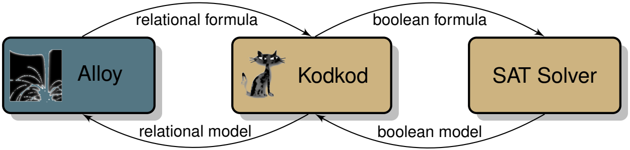


# Implementation Challenges

## implementation options

- ~~enumerate values of bound variables and evaluate quantifiers~~
  - → ~~extremely inefficient~~
- ~~directly encode to SAT~~
  - → ~~3-valued logic must be used throughout~~
  - → ~~2 bits required to represent 1 boolean variable~~
  - → ~~likely to adversely affect models that don't involve integers~~
- translate to classical logic and existing SAT-based back-end
  - → models without integers remain unaffected

## alloy architecture



# Translation to Classical Logic (1)

## key requirement

- every boolean formula must denote (evaluate to `true` or `false`)

## consequence

- a truth value must be assigned to predicates involving undefined terms [Farmer'95]

# Translation to Classical Logic (1)

## key requirement

- every boolean formula must denote (evaluate to `true` or `false`)

## consequence

- a truth value must be assigned to predicates involving undefined terms [Farmer'95]

```
all x: Int |
  x > 0 => x.plus[x] > x
```

$\longrightarrow$ x=3:

# Translation to Classical Logic (1)

## key requirement

- every boolean formula must denote (evaluate to `true` or `false`)

## consequence

- a truth value must be assigned to predicates involving undefined terms [Farmer'95]

```
all x: Int |
  x > 0 => x.plus[x] > x
```

$\longrightarrow$ x=3: $[\![$ x.plus[x] > x $]\!]$ = true

# Translation to Classical Logic (1)

## key requirement

- every boolean formula must denote (evaluate to `true` or `false`)

## consequence

- a truth value must be assigned to predicates involving undefined terms [Farmer'95]

```
all x: Int |
  x > 0 => x.plus[x] > x
```

$\longrightarrow$ x=3: $[\![$ x.plus[x] > x $]\!]$ = true

```
some x: Int |
  x > 0 && x.plus[x] < x
```

$\longrightarrow$ x=3: $[\![$ x.plus[x] < x $]\!]$ = false

# Translation to Classical Logic (1)

## key requirement

- every boolean formula must denote (evaluate to `true` or `false`)

## consequence

- a truth value must be assigned to predicates involving undefined terms [Farmer'95]

```
all x: Int |
  x > 0 => x.plus[x] > x
```

$\longrightarrow$ x=3: $[\![$ x.plus[x] > x $]\!]$ = true

```
some x: Int |
  x > 0 && x.plus[x] < x
```

$\longrightarrow$ x=3: $[\![$ x.plus[x] < x $]\!]$ = false

## approach

- **only** integer functions can result in an undefined integer value ($\perp$)
  - $\rightarrow$ use textbook overflow circuits to detect such cases

# Translation to Classical Logic (1)

## key requirement

- every boolean formula must denote (evaluate to `true` or `false`)

## consequence

- a truth value must be assigned to predicates involving undefined terms [Farmer'95]

```
all x: Int |
  x > 0 => x.plus[x] > x
```

```
some x: Int |
  x > 0 && x.plus[x] < x
```

$\longrightarrow$ x=3: $[\![$ x.plus[x] > x $]\!]$ = true

$\longrightarrow$ x=3: $[\![$ x.plus[x] < x $]\!]$ = false

## approach

- **only** integer functions can result in an undefined integer value ($\perp$)
  - $\rightarrow$ use textbook overflow circuits to detect such cases
- **single link** from integers to boolean formulas: comparison predicates
  - $\rightarrow$ adjust the semantics of integer comparison predicates
  - $\rightarrow$ when either term is $\perp$, evaluate to make the outer binding irrelevant

# Translation to Classical Logic (2)

## definition

$$[\![x < y]\!]\sigma = \begin{cases} x < y \ \wedge \ x \neq \bot \ \wedge \ y \neq \bot, & \textit{if } \sigma{=}\sigma_\exists \quad \text{(in existential context)} \\ x < y \ \vee \ x = \bot \ \vee \ y = \bot, & \textit{if } \sigma{=}\sigma_\forall \quad \text{(in universal context)} \end{cases}$$

# Translation to Classical Logic (2)

## definition

$$[\![x < y]\!]\sigma = \begin{cases} x < y \;\wedge\; x \neq \bot \;\wedge\; y \neq \bot, & \textit{if}\ \sigma{=}\sigma_\exists \quad (\textit{in existential context}) \\ x < y \;\vee\; x = \bot \;\vee\; y = \bot, & \textit{if}\ \sigma{=}\sigma_\forall \quad (\textit{in universal context}) \end{cases}$$

**what about negation:** $[\![\neg(x < y)]\!]\sigma_\exists = ?$

# Translation to Classical Logic (2)

## definition

$$[\![x<y]\!]\sigma = \begin{cases} x<y \;\wedge\; x\neq\bot \;\wedge\; y\neq\bot, & \textit{if}\;\; \sigma=\sigma_\exists \quad \text{(}\textit{in existential context}\text{)} \\ x<y \;\vee\; x=\bot \;\vee\; y=\bot, & \textit{if}\;\; \sigma=\sigma_\forall \quad \text{(}\textit{in universal context}\text{)} \end{cases}$$

**what about negation:** $[\![\neg(x<y)]\!]\sigma_\exists = ?$

compositional

$$[\![\neg(x<y)]\!]\sigma_\exists = \neg[\![x<y]\!]\sigma_\exists$$

$$= \neg(x<y \;\wedge\; x\neq\bot \;\wedge\; y\neq\bot)$$

$$= x\geq y \;\vee\; x=\bot \;\vee\; y=\bot$$

$$\neq [\![x\geq y]\!]\sigma_\exists$$

# Translation to Classical Logic (2)

## definition

$$[\![x<y]\!]\sigma = \begin{cases} x<y \ \wedge \ x \neq \bot \ \wedge \ y \neq \bot, & \textit{if } \sigma=\sigma_\exists \quad (\textit{in existential context}) \\ x<y \ \vee \ x = \bot \ \vee \ y = \bot, & \textit{if } \sigma=\sigma_\forall \quad (\textit{in universal context}) \end{cases}$$

## what about negation: $[\![\neg(x<y)]\!]\sigma_\exists = ?$

~~compositional~~

$$\cancel{\begin{aligned} [\![\neg(x<y)]\!]\sigma_\exists &= \neg[\![x<y]\!]\sigma_\exists \\ &= \neg(x<y \ \wedge \ x \neq \bot \ \wedge \ y \neq \bot) \\ &= x \geq y \ \vee \ x = \bot \ \vee \ y = \bot \\ &\neq [\![x \geq y]\!]\sigma_\exists \end{aligned}}$$

semantics preserving

$$\begin{aligned} [\![\neg(x<y)]\!]\sigma_\exists &= [\![x \geq y]\!]\sigma_\exists \\ &= x \geq y \ \wedge \ x \neq \bot \ \wedge \ y \neq \bot \end{aligned}$$

# Translation to Classical Logic (2)

## definition

$$[\![x<y]\!]\sigma = \begin{cases} x<y \;\wedge\; x\neq\bot \;\wedge\; y\neq\bot, & \textit{if } \sigma=\sigma_\exists \quad \text{(in existential context)} \\ x<y \;\vee\; x=\bot \;\vee\; y=\bot, & \textit{if } \sigma=\sigma_\forall \quad \text{(in universal context)} \end{cases}$$

**what about negation:** $[\![\neg(x<y)]\!]\sigma_\exists = ?$

~~compositional~~

$$\cancel{\begin{aligned} [\![\neg(x<y)]\!]\sigma_\exists &= \neg[\![x<y]\!]\sigma_\exists \\ &= \neg(x<y \;\wedge\; x\neq\bot \;\wedge\; y\neq\bot) \\ &= x\geq y \;\vee\; x=\bot \;\vee\; y=\bot \\ &\neq [\![x\geq y]\!]\sigma_\exists \end{aligned}}$$

semantics preserving

$$\begin{aligned} [\![\neg(x<y)]\!]\sigma_\exists &= [\![x\geq y]\!]\sigma_\exists \\ &= x\geq y \;\wedge\; x\neq\bot \;\wedge\; y\neq\bot \\ &= \neg(x<y \;\vee\; x=\bot \;\vee\; y=\bot) \\ &= \neg([\![x<y]\!]\sigma_\forall) \end{aligned}$$

# Translation to Classical Logic (2)

**definition**

$$[\![\rho(x,y)]\!]\sigma = \begin{cases} \rho(x,y) \;\wedge\; x \neq \bot \;\wedge\; y \neq \bot, & \text{if } \sigma = \sigma_\exists \quad \text{(in existential context)} \\ \rho(x,y) \;\vee\; x = \bot \;\vee\; y = \bot, & \text{if } \sigma = \sigma_\forall \quad \text{(in universal context)} \end{cases}$$

$$\rho \in \{<, \leq, =, \neq, >, \geq\}$$

**what about negation:** $[\![\neg(x<y)]\!]\sigma_\exists = ?$

compositional

$$\begin{aligned} [\![\neg(x<y)]\!]\sigma_\exists &= \neg[\![x<y]\!]\sigma_\exists \\ &= \neg(x<y \;\wedge\; x \neq \bot \;\wedge\; y \neq \bot) \\ &= x \geq y \;\vee\; x = \bot \;\vee\; y = \bot \\ &\neq [\![x \geq y]\!]\sigma_\exists \end{aligned}$$

semantics preserving

$$\begin{aligned} [\![\neg(x<y)]\!]\sigma_\exists &= [\![x \geq y]\!]\sigma_\exists \\ &= x \geq y \;\wedge\; x \neq \bot \;\wedge\; y \neq \bot \\ &= \neg(x<y \;\vee\; x = \bot \;\vee\; y = \bot) \\ &= \neg([\![x<y]\!]\sigma_\forall) \end{aligned}$$

**rule for negation:**

$$\begin{aligned} [\![\neg p]\!]\sigma_\exists &= \neg[\![p]\!]\sigma_\forall \\ [\![\neg p]\!]\sigma_\forall &= \neg[\![p]\!]\sigma_\exists \end{aligned}$$

# Evaluation

**how does the new encoding affect performance?**

- extra clauses are generated to detect and prevent overflows
  - → (only when arithmetic operations are used)
- no extra primary variables are used

# Evaluation

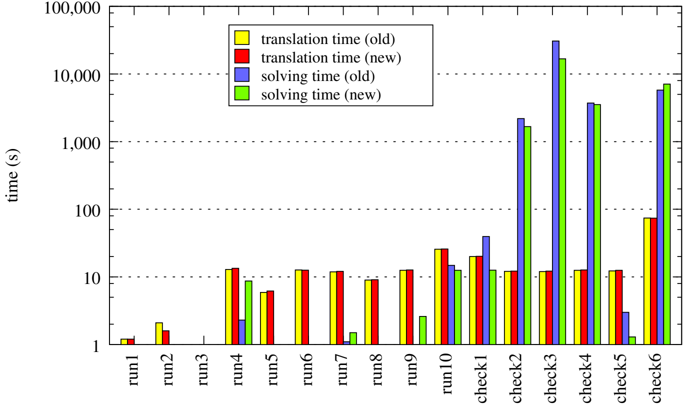
## how does the new encoding affect performance?

- extra clauses are generated to detect and prevent overflows
  - → (only when arithmetic operations are used)
- no extra primary variables are used
- possible effects of extra clauses on solving time:
  - → **speedup**: because search space smaller (more constrained)
  - → **slowdown**: SAT solver can get stuck more easily

# Experiment

## flash filesystem [Kang, ABZ'08]

- heavy use of arithmetic (for computing memory addresses)
- we ran 10 simulations and 6 checks
- **total time** decreased from 12 hours to 8 hours
- **this result is not meant to be conclusive!**



# Summary

## summary

- alloy made sound with respect to counterexamples

## applications that can benefit

- program verifications
- test case generation
- specification execution

## ideas for future work

- user-defined partial functions

CSAIL

# Thank You!

http://alloy.mit.edu

ALLOY

# Spurious Counterexamples due to Overflows

**reason for overflows**

- wraparound semantics for arithmetic operations
  - → `Int = {-4, -3, ..., 2, 3}` $\Longrightarrow$ `3 + 1 = -4`

# Spurious Counterexamples due to Overflows

## reason for overflows

- wraparound semantics for arithmetic operations
  → `Int = {-4, -3, ..., 2, 3}` ⟹ `3 + 1 = -4`

## prototypical anomalies

- sum of two positive integers is not necessarily positive!

```
check {
  all x, y: Int |
    x > 0 && y > 0 => x.plus[y] > 0
} for 3 Int
```

**counterexample**

```
Int = {-4, -3, ..., 2, 3}
a = 3; b = 1;
a.plus[b] = -4
```

# Spurious Counterexamples due to Overflows

## reason for overflows

- wraparound semantics for arithmetic operations

  → `Int = {-4, -3, ..., 2, 3}` ⟹ `3 + 1 = -4`

## prototypical anomalies

- sum of two positive integers is not necessarily positive!

```
check {
  all x, y: Int |
    x > 0 && y > 0 => x.plus[y] > 0
} for 3 Int
```

**counterexample**

```
Int = {-4, -3, ..., 2, 3}
a = 3; b = 1;
a.plus[b] = -4
```

- cardinality of a non-empty set is not necessarily positive!

```
check {
  all s: set univ |
    some s iff #s > 0
} for 4 but 3 Int
```

**counterexample**

```
Int = {-4, -3, ..., 2, 3}
s = {S0, S1, S2, S3}
#s = -4
```

# Example

## rules

$$[\![\rho(x,y)]\!]\sigma \quad = \quad \begin{cases} \rho(x,y) \ \wedge \ x \neq \bot \ \wedge \ y \neq \bot, & \textit{if } \ \sigma{=}\sigma_\exists \\ \rho(x,y) \ \vee \ x = \bot \ \vee \ y = \bot, & \textit{if } \ \sigma{=}\sigma_\forall \end{cases}$$

$\rho \in \{<, \leq, =, \neq, >, \geq\}$

$$[\![\neg p]\!]\sigma_\exists \quad = \quad \neg[\![p]\!]\sigma_\forall$$
$$[\![\neg p]\!]\sigma_\forall \quad = \quad \neg[\![p]\!]\sigma_\exists$$

## example

```
all x: Int |
  x > 0 => x.plus[x] > x
```

$\longrightarrow x = 3:$

```
some x: Int |
  x > 0 && !(x.plus[x] > x)
```

$\longrightarrow x = 3:$

# Example

## rules

$$[\![\rho(x,y)]\!]\sigma \quad = \quad \begin{cases} \rho(x,y) \;\wedge\; x \neq \bot \;\wedge\; y \neq \bot, & \textit{if } \sigma = \sigma_\exists \\ \rho(x,y) \;\vee\; x = \bot \;\vee\; y = \bot, & \textit{if } \sigma = \sigma_\forall \end{cases}$$

$$\rho \in \{<, \leq, =, \neq, >, \geq\}$$

$$[\![\neg p]\!]\sigma_\exists \quad = \quad \neg[\![p]\!]\sigma_\forall$$
$$[\![\neg p]\!]\sigma_\forall \quad = \quad \neg[\![p]\!]\sigma_\exists$$

## example

```
all x: Int |
  x > 0 => x.plus[x] > x
```

$\longrightarrow x = 3:$

$$[\![x + x > x]\!]\sigma_\forall$$
$$= 3 + 3 > 0 \;\vee\; 3 + 3 = \bot \;\vee\; 0 = \bot$$
$$= \texttt{false} \;\vee\; \texttt{true} \;\vee\; \texttt{false}$$
$$= \texttt{true}$$

```
some x: Int |
  x > 0 && !(x.plus[x] > x)
```

$\longrightarrow x = 3:$

# Example

## rules

$$\llbracket \rho(x,y) \rrbracket \sigma \quad = \quad \begin{cases} \rho(x,y) \ \wedge \ x \neq \bot \ \wedge \ y \neq \bot, & \textit{if } \ \sigma = \sigma_\exists \\ \rho(x,y) \ \vee \ x = \bot \ \vee \ y = \bot, & \textit{if } \ \sigma = \sigma_\forall \end{cases}$$

$$\rho \in \{<, \leq, =, \neq, >, \geq\}$$

$$\llbracket \neg p \rrbracket \sigma_\exists \quad = \quad \neg \llbracket p \rrbracket \sigma_\forall$$
$$\llbracket \neg p \rrbracket \sigma_\forall \quad = \quad \neg \llbracket p \rrbracket \sigma_\exists$$

## example

```
all x: Int |
  x > 0 => x.plus[x] > x
```

$\longrightarrow x = 3:$

$$\llbracket x + x > x \rrbracket \sigma_\forall$$
$$= 3 + 3 > 0 \ \vee \ 3 + 3 = \bot \ \vee \ 0 = \bot$$
$$= \texttt{false} \ \vee \ \texttt{true} \ \vee \ \texttt{false}$$
$$= \texttt{true}$$

```
some x: Int |
  x > 0 && !(x.plus[x] > x)
```

$\longrightarrow x = 3:$

$$\llbracket !(x + x > x) \rrbracket \sigma_\exists$$
$$= \neg \llbracket x + x > x \rrbracket \sigma_\forall$$
$$= \neg \texttt{true}$$
$$= \texttt{false}$$

# Law of the Excluded Middle

**is law of the excluded middle still preserved?**

- the non-compositional rule for negation suggests it's not

# Law of the Excluded Middle

## is law of the excluded middle still preserved?

- the non-compositional rule for negation suggests it's not
- in a bounded setting of **alloy**, that is usually not a problem
  - → all integers when multiplied by 2 are either negative or non-negative?

```
check {
  all x: Int | x.mul[2] < 0 or !(x.mul[2] < 0)
} for 4 Int
```

# Law of the Excluded Middle

## is law of the excluded middle still preserved?

- the non-compositional rule for negation suggests it's not
- in a bounded setting of **alloy**, that is usually not a problem
  - → all integers when multiplied by 2 are either negative or non-negative?

```
check {
  all x: Int | x.mul[2] < 0 or !(x.mul[2] < 0)
} for 4 Int
```

*all integers x such that x times 2 does not overflow,*
*x times 2 is either negative or non-negative*

# Law of the Excluded Middle

## is law of the excluded middle still preserved?

- the non-compositional rule for negation suggests it's not
- in a bounded setting of **alloy**, that is usually not a problem
  - → all integers when multiplied by 2 are either negative or non-negative?

```
check {
  all x: Int | x.mul[2] < 0 or !(x.mul[2] < 0)
} for 4 Int
```

*all integers x such that x times 2 does not overflow,*
*x times 2 is either negative or non-negative*

- violation of the law is still observable

```
check { 4.plus[5]  = 6.plus[3] } for 4 Int
```

# Law of the Excluded Middle

## is law of the excluded middle still preserved?

- the non-compositional rule for negation suggests it's not
- in a bounded setting of **alloy**, that is usually not a problem
  - → all integers when multiplied by 2 are either negative or non-negative?

```
check {
  all x: Int | x.mul[2] < 0 or !(x.mul[2] < 0)
} for 4 Int
```

*all integers x such that x times 2 does not overflow,*
*x times 2 is either negative or non-negative*

- violation of the law is still observable

```
check { 4.plus[5]  = 6.plus[3] } for 4 Int
```

```
check { 4.plus[5] != 6.plus[3] } for 4 Int
```

# Law of the Excluded Middle

## is law of the excluded middle still preserved?

- the non-compositional rule for negation suggests it's not
- in a bounded setting of **alloy**, that is usually not a problem
  - → all integers when multiplied by 2 are either negative or non-negative?

```
check {
  all x: Int | x.mul[2] < 0 or !(x.mul[2] < 0)
} for 4 Int
```

✓

*all integers x <u>such that x times 2 does not overflow</u>,*
*x times 2 is either negative or non-negative*

- violation of the law is still observable

```
check { 4.plus[5]  = 6.plus[3] } for 4 Int
```

✓

```
check { 4.plus[5] != 6.plus[3] } for 4 Int
```

✓

  - → the violation is visible if truth is associated with a check yields a counterexample at all

# Partial Functions in Logic

## overflows in alloy

- instance of a more general problem: handling partial functions in logic

## existing solutions/approaches

- logic of partial functions (LPF) [C. B. Jones]
  - → both integer functions and boolean formulas may be undefined
  - → uses a 3-valued logic
- traditional approach [Farmer'95]
  - → functions may be partial, but formulas must be denoting
  - → if any term is undefined, formula evaluates to `false`
  - → leaves open whether $\neg(a = a) \equiv a \neq a$ given that $a$ is undefined
- totalize all functions
  - → wraparound semantics for integer arithmetic in old alloy
  - → out-of-bounds applications result in unknown (but determined) value [B, Z]

## differentiating characteristics of our approach

- customized for the bounded setting
- masking quantifier bindings that produce undefinedness