Metasketches

James Bornholt
bornholt@cs.washington.edu
Today

Last lecture

• Program synthesis
• Solver-aided languages

Today

• Metasketches: building effective synthesis-aided tools
• MemSynth: an example of a metasketch-based tool

Reminders

• Course feedback form is open
• Project presentations on Friday
• Project reports and prototypes due Friday at 11:00pm
Metasketches

Program synthesis
Specification \rightarrow \text{Program synthesis}
Specification → Program synthesis → Program
Program synthesis

Specification

\[ f(x) = 4x \]

Program
Program synthesis

Specification

\[ f(x) = 4x \]

Program

\[ x + x + x + x \]
Program synthesis

Specification

$$f(x) = 4x$$

Compilation
[PLDI’14]

Data Structures
[PLDI’15]

End-user Programming
[POPL’11]

Program

$$x+x+x+x$$

Executable Biology
[POPL’13]

Browser Layout
[PPoPP’13]

Cache Protocols
[PLDI’13]
Program synthesis

Often looking for an optimal solution, not just any correct program
Often looking for an *optimal* solution, not just any correct program.

There are *many* programs, so tools must control search strategy.

**Program synthesis**

- **Compilation** [PLDI’14]
- **Data Structures** [PLDI’15]
- **End-user Programming** [POPL’11]
- **Executable Biology** [POPL’13]
- **Browser Layout** [PPoPP’13]
- **Cache Protocols** [PLDI’13]
Often looking for an optimal solution, not just any correct program

There are many programs, so tools must control search strategy

Compilation [PLDI'14]
Data Structures [PLDI'15]
End-user Programming [POPL'11]

Specification

Program synthesis

Executable Biology [POPL'13]
Browser Layout [PPoPP'13]
Cache Protocols [PLDI'13]
Program specification → Program synthesis → Program
Metasketches

Specification

Program synthesis

Program
Metasketches

A framework that makes search strategy and optimality part of the problem definition

Specification → Program synthesis → Program
Syntax-guided synthesis

Specification → Program synthesis → Program
Syntax-guided synthesis

Specification → Program synthesis → Program

Sketch
Syntax-guided synthesis

Specifying  

Program synthesis  

Sketch  

Program

```python
def f(x):
    return Expr

Expr := x | ?? | Expr op Expr
op := + | * | - | >> | <<
?? := integer constant
```
Syntax-guided synthesis: guess, check, learn

```python
def f(x):
    return Expr

Expr := x | ?? | Expr op Expr
    op := + | * | - | >> | <<
    ?? := integer constant
```
Syntax-guided synthesis: guess, check, learn

```python
def f(x):
    return Expr

Expr := x | ?? | Expr op Expr
op := + | * | - | >> | <<
?? := integer constant
```

Syntax
Syntax-guided synthesis: guess, check, learn
Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

$f(x) = 4x$
Syntax-guided synthesis: guess, check, learn
Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]
Syntax-guided synthesis: guess, check, learn
Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

Semantics

Syntax

\[ f(x) = 4x \]
Syntax-guided synthesis: guess, check, learn
Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

Semantics

Syntax

\[ f(x) = 4x \]

All programs \( p \) for which \( p(x_0) \neq f(x_0) \).
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

All programs $p$ for which $p(x_0) \neq f(x_0)$. 

Semantics

$\mathit{f(x)} = 4x$

Syntax
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

Semantics

\[ f(x) = 4x \]

Syntax

\[ x + x + x + x + x \]

All programs \( p \) for which \( p(x_0) \neq f(x_0) \).
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

1. Search order is critical
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

1. Search order is critical
2. Desire optimal solutions
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

1. Search order is critical
2. Desire optimal solutions
Metasketches express structure and strategy

1. Search order is critical
2. Desire optimal solutions

Syntax
Metasketches express structure and strategy

1. Search order is critical
2. Desire optimal solutions

A metasketch contains:

1. structured candidate space ($\mathcal{S}$, $\preceq$)
Metasketches express structure and strategy

1. Search order is critical
2. Desire optimal solutions

A metasketch contains:

1. structured candidate space ($\mathcal{S}$, $\preceq$)
2. cost function ($\kappa$)
Metasketches express structure and strategy

1. Search order is critical
2. Desire optimal solutions

A metasketch contains:

1. structured candidate space \((\mathcal{S}, \preceq)\)
2. cost function \((\kappa)\)
3. gradient function \((g)\)
Metasketches express structure and strategy

1. Search order is critical
2. Desire optimal solutions

A metasketch contains:
1. structured candidate space ($\mathcal{S}$, $\preceq$)
2. cost function ($\kappa$)
3. gradient function ($g$)
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)

2. cost function ($\kappa$)

3. gradient function ($g$)
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)

A fragmentation of the candidate space, and an ordering on those fragments.

2. cost function ($\kappa$)

3. gradient function ($g$)
Metasketches express structure and strategy

1. structured candidate space \((\mathcal{S}, \preceq)\)
   - A fragmentation of the candidate space, and an ordering on those fragments.
   - \(\mathcal{S} = \text{set of all SSA programs}\)

2. cost function \((\kappa)\)

3. gradient function \((g)\)
Metasketches express structure and strategy

1. **structured candidate space** \( (\mathcal{S}, \preceq) \)
   - A fragmentation of the candidate space, and an ordering on those fragments.

2. **cost function** \( (\kappa) \)

3. **gradient function** \( (g) \)

\[ \mathcal{S} = \text{set of all SSA programs} \]
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)
   - a countable set $\mathcal{S}$ of sketches
   - a total order $\preceq$ on $\mathcal{S}$

$\mathcal{S}$ = set of all SSA programs

A fragmentation of the candidate space, and an ordering on those fragments.

2. cost function ($\kappa$)

3. gradient function ($g$)
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)
   - a countable set $\mathcal{S}$ of sketches
   - a total order $\preceq$ on $\mathcal{S}$

   A fragmentation of the candidate space, and an ordering on those fragments.

   $\mathcal{S}$ = set of all SSA programs

   $\mathcal{S}_3$ (SSA programs of length 3)

   ```python
   def f(x):
       r1 = ??_op(??{x})
       r2 = ??_op(??{x,r1})
       r3 = ??_op(??{x,r1,r2})
       return r3
   ```

2. cost function ($K$)

3. gradient function ($g$)
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)
   - a countable set $\mathcal{S}$ of sketches
   - a total order $\preceq$ on $\mathcal{S}$

A fragmentation of the candidate space, and an ordering on those fragments.

$\mathcal{S}$ = set of all SSA programs

2. cost function ($\kappa$)

3. gradient function ($g$)
Metasketches express structure and strategy

1. structured candidate space \((\mathcal{S}, \preceq)\)
   - a countable set \(\mathcal{S}\) of sketches
   - a total order \(\preceq\) on \(\mathcal{S}\)

\[ S_3 \quad \text{(SSA programs of length 3)} \]

\[
\begin{align*}
\text{def } f(x): & \\
& r_1 = \text{??}_\text{op}(\text{??}\{x\}) \\
& r_2 = \text{??}_\text{op}(\text{??}\{x, r_1\}) \\
& r_3 = \text{??}_\text{op}(\text{??}\{x, r_1, r_2\}) \\
& \text{return } r_3
\end{align*}
\]

A fragmentation of the candidate space, and an ordering on those fragments.

\(\mathcal{S}\) = set of all SSA programs

2. cost function \((\kappa)\)

3. gradient function \((g)\)
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)
   - a countable set $\mathcal{S}$ of sketches
   - a total order $\preceq$ on $\mathcal{S}$

   A fragmentation of the candidate space, and an ordering on those fragments.

   $\mathcal{S}$ = set of all SSA programs

   S1
   S2
   S3
   S4
   S5
   ...

2. cost function ($\kappa$)

3. gradient function ($g$)
Metasketches express structure and strategy

1. structured candidate space \((\mathcal{S}, \preceq)\)
   - a countable set \(\mathcal{S}\) of sketches
   - a total order \(\preceq\) on \(\mathcal{S}\)

2. cost function \((\kappa)\):

3. gradient function \((g)\):

\[\mathcal{S} = \text{set of all SSA programs}\]

A fragmentation of the candidate space, and an ordering on those fragments.

**S_3** (SSA programs of length 3)

\[
\begin{align*}
def \ f(x): \\
& r_1 = \text{op}(\{x\}) \\
& r_2 = \text{op}(\{x, r_1\}) \\
& r_3 = \text{op}(\{x, r_1, r_2\}) \\
& \text{return } r_3
\end{align*}
\]

Ordering expresses high-level search strategy.
Metasketches express structure and strategy

1. structured candidate space \((\mathcal{S}, \preceq)\)
   - a countable set \(\mathcal{S}\) of sketches
   - a total order \(\preceq\) on \(\mathcal{S}\)

2. cost function \((\kappa)\)

3. gradient function \((g)\)

\(\mathcal{S}\) = set of all SSA programs

A fragmentation of the candidate space, and an ordering on those fragments.

Ordering expresses high-level search strategy.

Here, \(\preceq\) expresses iterative deepening.

\[
\begin{align*}
\mathcal{S}_3 \quad &\text{(SSA programs of length 3)} \\
\text{def } f(x): &\quad r_1 = \text{op}(\{x\}) \\
&\quad r_2 = \text{op}(\{x, r_1\}) \\
&\quad r_3 = \text{op}(\{x, r_1, r_2\}) \\
&\quad \text{return } r_3
\end{align*}
\]
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)
   - a countable set $\mathcal{S}$ of sketches
   - a total order $\preceq$ on $\mathcal{S}$

A fragmentation of the candidate space, and an ordering on those fragments.

$\mathcal{S}$ = set of all SSA programs

2. cost function ($\kappa$)

3. gradient function ($g$)

Implemented as a generator that returns the next sketch in the space
Metasketches express structure and strategy

1. structured candidate space \((\mathcal{S}, \preceq)\)
   - a countable set \(\mathcal{S}\) of sketches
   - a total order \(\preceq\) on \(\mathcal{S}\)

\[\mathcal{S} = \text{set of all SSA programs}\]

2. cost function \((\kappa)\)
3. gradient function \((g)\)
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)
   - a countable set $\mathcal{S}$ of sketches
   - a total order $\preceq$ on $\mathcal{S}$

2. cost function ($\kappa$)

3. gradient function ($g$)

$\mathcal{S} = \text{set of all SSA programs}$
Metasketches express structure and strategy

1. structured candidate space \((\mathcal{S}, \preceq)\)
   - a countable set \(\mathcal{S}\) of sketches
   - a total order \(\preceq\) on \(\mathcal{S}\)

   \(\mathcal{S}\) = set of all SSA programs

2. cost function \((\kappa)\)

3. gradient function \((g)\)
Metasketches express structure and strategy

1. structured candidate space \((\mathcal{S}, \preceq)\)
   - a countable set \(\mathcal{S}\) of sketches
   - a total order \(\preceq\) on \(\mathcal{S}\)

\(\mathcal{S}\) = set of all SSA programs

Semantic redundancy in the search space.

2. cost function \((\kappa)\)
3. gradient function \((g)\)
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)
   - a countable set $\mathcal{S}$ of sketches
   - a total order $\preceq$ on $\mathcal{S}$

$\mathcal{S}$ = set of all SSA programs

Semantic redundancy in the search space.

Structure constraints eliminate some overlap between sketches

2. cost function ($\kappa$)

3. gradient function ($g$)
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)
   - a countable set $\mathcal{S}$ of sketches
   - a total order $\preceq$ on $\mathcal{S}$

2. cost function ($\kappa$)

3. gradient function ($g$)

$\mathcal{S} = \text{set of all SSA programs}$

Semantics

Semantically redundant in the search space.

Structure constraints eliminate some overlap between sketches

$\mathcal{S}_2$ (SSA programs of length 3)

```python
def f(x):
    r_1 = op({x})
    r_2 = op({x, r_1})
    r_3 = op({x, r_1, r_2})
    return r_3
```
Metasketches express structure and strategy

1. structured candidate space ($\mathcal{S}$, $\preceq$)
   - a countable set $\mathcal{S}$ of sketches
   - a total order $\preceq$ on $\mathcal{S}$

2. cost function ($\kappa$)

3. gradient function ($g$)

\[ \mathcal{S} = \text{set of all SSA programs} \]

\[ S_1, S_2, S_3, S_4 \]

$S_3$ (SSA programs of length 3)

\[ \text{def } f(x): \]
\[ r_1 = ??_{op}(??_{\{x\}}) \]
\[ r_2 = ??_{op}(??_{\{x,r_1\}}) \]
\[ r_3 = ??_{op}(??_{\{x,r_1,r_2\}}) \]

return $r_3$
Cost functions rank candidate programs

1. structured candidate space $(\mathcal{S}, \preceq)$

2. cost function $(\kappa)$
   \[ \kappa : \mathcal{L} \rightarrow \mathbb{R} \]
   assigns a numeric cost to each program in the language $\mathcal{L}$

3. gradient function $(g)$

$\mathcal{S} = \text{set of all SSA programs}$
Cost functions rank candidate programs

1. structured candidate space \((\mathcal{S}, \preceq)\)

2. cost function \((\kappa)\)
   \[
   \kappa : \mathcal{L} \rightarrow \mathbb{R}
   \]
   assigns a numeric cost to each program in the language \(\mathcal{L}\)

Cost functions can be based on both syntax and semantics (dynamic behavior)

3. gradient function \((g)\)

\(\mathcal{S} = \text{set of all SSA programs}\)
Cost functions rank candidate programs

1. structured candidate space ($\mathcal{S}$, $\preceq$)

2. cost function ($\kappa$)

$$\kappa : \mathcal{L} \rightarrow \mathbb{R}$$

assigns a numeric cost to each program in the language $\mathcal{L}$

Cost functions can be based on both syntax and semantics (dynamic behavior)

3. gradient function ($g$)

$$\mathcal{S} = \text{set of all SSA programs}$$

$\kappa(P) = i$ for $P \in S_i \in \mathcal{S}$

The number of variables defined in $P$
Gradient functions provide cost structure

1. structured candidate space \((\mathcal{S}, \preceq)\)

2. cost function \((\kappa)\)

3. gradient function \((g)\)

\[ g : \mathbb{R} \rightarrow 2^\mathcal{S} \]

\( g(c) \) is the set of sketches in \( \mathcal{S} \) that \textit{may} contain a solution \( P \) with \( \kappa(P) < c \)

\[ \mathcal{S} = \text{set of all SSA programs} \]

\( \kappa(P) = i \) for \( P \in S_i \in \mathcal{S} \)
Gradient functions provide cost structure

1. structured candidate space \((\mathcal{S}, \preceq)\)
2. cost function \((\kappa)\)
3. gradient function \((g)\)

\[
g : \mathbb{R} \rightarrow 2^\mathcal{S}
\]

\(g(c)\) is the set of sketches in \(\mathcal{S}\) that may contain a solution \(P\) with \(\kappa(P) < c\)

\(\mathcal{S}\) = set of all SSA programs

\(\kappa(P) = i\) for \(P \in S_i \in \mathcal{S}\)
Gradient functions provide cost structure

1. structured candidate space ($\mathcal{S}$, $\preceq$)

2. cost function ($\kappa$)

3. gradient function ($g$)

$$g : \mathbb{R} \to 2^\mathcal{S}$$

$g(c)$ is the set of sketches in $\mathcal{S}$ that may contain a solution $P$ with $\kappa(P) < c$

$\mathcal{S} = \text{set of all SSA programs}$

$\kappa(P) = i$ for $P \in S_i \in \mathcal{S}$

$$g(c) = \{ S_i \in \mathcal{S} \mid i < c \}$$
Gradient functions provide cost structure

1. structured candidate space $(\mathcal{S}, \preceq)$
2. cost function $(\kappa)$
3. gradient function $(g)$

$$g : \mathbb{R} \rightarrow 2^{\mathcal{S}}$$

$g(c)$ is the set of sketches in $\mathcal{S}$ that 
*may* contain a solution $P$ with $\kappa(P) < c$

The gradient function overapproximates the behavior of $\kappa$ on $\mathcal{S}$

$\mathcal{S} = \text{set of all SSA programs}$

$g(4) = \{S_1, S_2, S_3\}$

$\kappa(P) = i$ for $P \in S_i \in \mathcal{S}$

$g(c) = \{S_i \in \mathcal{S} \mid i < c\}$
Gradient functions provide cost structure

1. structured candidate space ($\mathcal{S}$, $\preceq$)

2. cost function ($\mathbb{K}$)

3. gradient function ($g$)

$$g : \mathbb{R} \rightarrow 2^\mathcal{S}$$

$g(c)$ is the set of sketches in $\mathcal{S}$ that may contain a solution $P$ with $\mathbb{K}(P) < c$

$\mathcal{S} = \text{set of all SSA programs}$

$\mathbb{K}(P) = i$ for $P \in S_i \in \mathcal{S}$

$g(c) = \{ S_i \in \mathcal{S} | i < c \}$
Gradient functions provide cost structure

1. structured candidate space \((\mathcal{S}, \preceq)\)

2. cost function \((\kappa)\)

3. gradient function \((g)\)

\[ g : \mathbb{R} \rightarrow 2^\mathcal{S} \]

\(g(c)\) is the set of sketches in \(\mathcal{S}\) that may contain a solution \(P\) with \(\kappa(P) < c\)

\(\mathcal{S} = \text{set of all SSA programs}\)

**The gradient function overapproximates the behavior of \(\kappa\) on \(\mathcal{S}\)**

Always sound for \(g\) to return all of \(\mathcal{S}\) if a tighter bound is unavailable.

\(g(c)\) always being finite is sufficient (not necessary) to guarantee termination.

\[ \kappa(P) = i \quad \text{for} \quad P \in S_i \in \mathcal{S} \]

\[ g(c) = \{ S_i \in \mathcal{S} \mid i < c \} \]
Metasketches express structure and strategy

A metasketch contains:

1. structured candidate space ($\mathcal{S}$, $\preceq$)
2. cost function ($\kappa$)
3. gradient function ($g$)
Solving with two cooperative searches

\[ \langle \mathcal{S}, \preceq, K, g \rangle \]

- Coordinates the search for an optimal solution, offloading work to parallel local searches
Solving with two cooperative searches

\[ \langle S, \leq, K, g \rangle \]

Coordinates the search for an optimal solution, offloading work to parallel local searches

Global search

Local search

Local search

Local search

An incremental form of CEGIS that can accept new information from the global search
Solving with two cooperative searches

\[ \langle S, \preceq, \kappa, g \rangle \]
Solving with two cooperative searches

\[ \langle S, \leq, K, g \rangle \]

Global search

Local search

Local search

Local search

S1

S2

S3

S4

S5

S6

S7
Solving with two cooperative searches

\( \langle S, \leq, K, g \rangle \)

Global search

Local search

Local search

Local search

S1

S2

S3

S4

S5

S6

S7
Solving with two cooperative searches

\[ \langle \mathcal{S}, \preceq, K, g \rangle \]

Global search

Local search

Local search

Local search

S1

S2

S3

S4

S5

S6

S7
Solving with two cooperative searches

$\langle S, \leq, K, g \rangle$
Solving with two cooperative searches

\[ \langle S, \lesssim, K, g \rangle \]

Global search

Local search

Local search

Local search

SAT (P)

S1

S4

S3

S5

S6

S7
Solving with two cooperative searches

\[ \langle S, \preceq, K, g \rangle \]

Global search

Local search

Local search

Local search

SAT (P)

S1

S4

S3

S5

S6

S7
Solving with two cooperative searches

\( \langle S, \preceq, K, g \rangle \)

Global search

Local search

Local search

Local search

S1

S4

S3

S6

S5

S7
Solving with two cooperative searches

\[ \langle S, \preceq, K, g \rangle \]

Global search

- \( \kappa(P) \)
  - Local search
  - \( \kappa(P) \)
  - Local search
  - \( \kappa(P) \)
  - Local search

Prune local search spaces using \( \kappa(P) \)
Solving with two cooperative searches

\[ \langle S, \leq, K, g \rangle \]

Global search

Local search

Local search

Local search

Prune local search spaces using \( K(P) \)
Solving with two cooperative searches

\[ \langle S, \preceq, K, g \rangle \]

Global search

- Local search
- Local search
- Local search

Prune local search spaces using \( \kappa(P) \)

Prune global search space using \( g(\kappa(P)) \)
Solving with two cooperative searches

\[ \langle S, \preceq, \kappa, g \rangle \]

Global search

- Prune local search spaces using \( \kappa(P) \)
- Prune global search space using \( g(\kappa(P)) \)
Solving with two cooperative searches

\[ \langle S, \preceq, K, g \rangle \]

Continues until all search spaces exhausted, yielding an optimal solution.

Prune global search space using \( g(K(P)) \)

Prune local search spaces using \( K(P) \)
Synapse implementation

$$\langle \mathcal{S}, \leq, \kappa, g \rangle$$

Implemented in Rosette, a solver-aided extension of Racket
Synapse implementation

\[ \langle \mathcal{S}, \preceq, \kappa, g \rangle \]

Implemented in Rosette, a solver-aided extension of Racket

Local CEGIS searches can share counterexamples
Synapse implementation

\[ \langle \mathcal{S}, \leq, K, g \rangle \]

Implemented in Rosette, a solver-aided extension of Racket

Local CEGIS searches can share counterexamples

Local searches can time out, which weakens optimality
Synapse solves previously-intractable problems

Parrot benchmarks from approximate computing [Esmaelizadeh et al., 2012]

Find the most efficient approximate program within an error bound
Synapse solves previously-intractable problems

Parrot benchmarks from approximate computing [Esmaelizadeh et al., 2012]

Find the most efficient approximate program within an error bound

![Parrot Benchmarks Chart]

- All intractable to Sketch and Stoke
Synapse solves previously-intractable problems

Parrot benchmarks from approximate computing [Esmaelizadeh et al., 2012]

Find the most efficient approximate program within an error bound

```python
def inversek2j(float x, float y):
    th2 = acos(((x*x) + (y*y) - 0.5) / 0.5)
    th1 = asin((y * (0.5 + 0.5*cos(th2)) - 0.5*x*sin(th2)) / (x*x + y*y))
    return th1
```
Synapse solves standard benchmarks optimally

**Array Search** benchmarks from the syntax-guided synthesis (SyGuS) competition [Alur et al., 2015]

arraysearch-$n$: find program that searches lists of length $n$
Synapse solves standard benchmarks optimally

Array Search benchmarks from the syntax-guided synthesis (SyGuS) competition [Alur et al., 2015]

arraysearch-\(n\): find program that searches lists of length \(n\)

![Array Search diagram](image)

Synapse: 349 bytes
SyGuS: 7.1 MB
Is this a cat?
Synapse reasons about complex costs
Synapse reasons about complex costs

\[ \kappa(P) = \sum_{i} |P(x_i) - y_i| \]

*Classification error* executes the program for each point in the training set.
Metasketches express structure and strategy

A metasketch contains:

1. structured candidate space ($\mathcal{S}$, $\preceq$)
2. cost function ($\kappa$)
3. gradient function ($g$)

MemSynth

James Bornholt and Emina Torlak.
A memory model defines the reordering behaviors of a multiprocessor

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>print $Y$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
A memory model defines the reordering behaviors of a multiprocessor

<table>
<thead>
<tr>
<th></th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td>$X = Y = 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X = 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>print $Y$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$Y = 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>print $X$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
A memory model defines the reordering behaviors of a multiprocessor

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>print $Y$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
A memory model defines the reordering behaviors of a multiprocessor

<table>
<thead>
<tr>
<th>Initially $X = Y = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 1</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$X = 1$</td>
</tr>
<tr>
<td>$Y = 1$</td>
</tr>
<tr>
<td>print $X$</td>
</tr>
<tr>
<td>print $Y$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
A memory model defines the reordering behaviors of a multiprocessor

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y = 1$</td>
<td></td>
</tr>
<tr>
<td>$X = 1$</td>
<td></td>
</tr>
<tr>
<td>print $X$</td>
<td>print $Y$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
A memory model defines the reordering behaviors of a multiprocessor

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$Y = 1$</td>
<td></td>
</tr>
<tr>
<td>print $X$</td>
<td></td>
</tr>
<tr>
<td>$X = 1$</td>
<td></td>
</tr>
<tr>
<td>print $Y$</td>
<td></td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
A memory model defines the reordering behaviors of a multiprocessor

Thread 1

Initially $X = Y = 0$

Thread 2

Can this program print two zeroes?

print $Y$
A memory model defines the reordering behaviors of a multiprocessor. Initially $X = Y = 0$. Thread 1 can this program print two zeroes?

```
print Y
```
A memory model defines the reordering behaviors of a multiprocessor.

Thread 1

Initially $X = Y = 0$

Thread 2

Can this program print two zeroes?

print $X$

print $Y$

---

**Example 8-3. Loads May be Reordered with Older Stores**

<table>
<thead>
<tr>
<th>Processor 0</th>
<th></th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov $_x$, 1</td>
<td>mov $_y$, 1</td>
<td></td>
</tr>
<tr>
<td>mov r1, $_y$</td>
<td>mov r2, $_x$</td>
<td></td>
</tr>
</tbody>
</table>

Initially $x = y = 0$

$r1 = 0$ and $r2 = 0$ is allowed
x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors

By Peter Sewell, Sandor Szabo, Scott Owens, Francesco Pavya Nardell, and Magnus Olsson

Abstract

Exploiting the multiprocessors that have recently become ubiquitous requires high-performance and reliable concurrent systems code, for concurrent data structures, operating system kernels, synchronization libraries, compilers, and so on. However, concurrent programming, which is always challenging, is made much more so by two problems. First, real multiprocessors typically do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only locally consistent views of a shared memory. Second, the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for these specifications), leading to widespread confusion.

In this paper we focus on x86 processors. We revisit several recent Intel and AMD specifications, showing that all contain serious ambiguities, some arguably too weak to program on, and some are simply unanswerable with respect to actual hardware. We present a new x86-64 programmer's model that is, to the best of our knowledge, unambiguous in these problems. It is mathematically precise and rigorously defined in [10], but can be presented as an intuitive abstract machine which should be widely accessible to working programmers. We illustrate how this can be used to reason about the correctness of a Linux spool file implementation and describe a generic theory of data-queue semantics for x86-64. This should put all multiprocessor system building on a more solid foundation. It should also provide a basis for future work on verification of such systems.

1. Introduction

Multiprocessor machines, with many processes acting on a shared memory, have now become ubiquitous. However, the difficulty of programming concurrent systems has motivated extensive research on programming language design, semantics, and verification, from semaphores and monitors to program logics, software model checking, and so forth. This work has almost always assumed that concurrent systems are a large, sequentially consistent memory, with their reads and writes interleaved in some order. In fact, however, real multiprocessors use sophisticated techniques to achieve high-performance, high-quality, and on-chip caches, speculative execution, etc. These optimizations are not observable by sequential code, but in multithreaded programs different threads may see entirely different views of memory, with machines exhibiting relaxed, or weak, memory models. For a simple example, consider the following assembly language program (in /386 for modern Intel or AMD x86 multiprocessors given two distinct memory locations x and y initially holding 0): when processors repeatedly write to x and read from y, a x-interregister EAX on processor 0 and EDX on processor 1, it is possible for both to read 0 in the same execution. It is easy to construct this result even when none of the reads or writes of the two processors conflict with each other. These do not have a sequentially consistent execution.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor 0</td>
<td>x = 0</td>
</tr>
<tr>
<td>Processor 1</td>
<td>y = 0</td>
</tr>
</tbody>
</table>

Microarchitecturally, one can view this particular example as a visible consequence ofatom buffering if each processor effectively has a FIFO buffer of pending memory writes (to model the need to block while a write completes), then the writes from y and x occur before the writes have propagated from the buffers to memory. Other families of multiprocessors, when back at least to the IBM 370, and including ARM, Itanium, POWER, and others, exhibit relaxed-memory behavior. However, there are major and subtle differences between different processor families existing from their different internal design choices; in the details of exactly what non-sequentially-consistent executions they permit, and of how memory barrier and synchronization mechanisms they provide to let the programmer regain control.

For any of these processors, relaxed memory behavior resembles the difficulty of writing concurrent software, as system programmers cannot reason, at the level of abstraction of memory reads and writes, in terms of an intuitive model of global time.

This paper is based on work that first appeared in the Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), 2009, and in the Proceedings of the 23rd International Conference on Tools and Proof in Higher Order Logic (TYPHOON), 2009.
x86-ISO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors

By Peter Sewell, Samuli Seppänen, Scott Owens, Francesco Piazza, and Magura Zmeyeva

Abstract

Exploring the multiprocessors that have recently become ubiquitous requires high-performance and reliable concurrent system designs. This paper describes the x86-ISO model, a rigorous model for x86 multiprocessors that includes support for processors with shared memory. The model is built on the x86 processor architecture and includes a detailed description of the memory system, including support for shared memory. The model is intended to be used by system designers to explore the behavior of multiprocessors with shared memory.

1 Introduction

Multiprocessors with shared memory are becoming more ubiquitous. Meanwhile, the complexity of programming concurrent systems has increased, making verification more challenging. This paper presents a model of x86 processors with shared memory that is designed to be used for exploration and verification of concurrent systems.
x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors

1. Introduction

The x86-TSO (Translation State Option) is a state machine that provides a rigorous and usable model for x86 multiprocessors. It is designed to be both useful for hardware designers and easy to implement in software. The model is based on a combination of ideas from the Mach virtual memory system and the x86 architecture.

The x86-TSO model is based on a state machine that models the behavior of the x86 multiprocessor. The state machine is divided into two parts: the kernel state machine and the user state machine. The kernel state machine models the behavior of the kernel, while the user state machine models the behavior of user programs.

The x86-TSO model is designed to be easy to implement in software. It is based on a combination of ideas from the Mach virtual memory system and the x86 architecture. The model is designed to be both useful for hardware designers and easy to implement in software.

The x86-TSO model is based on a state machine that models the behavior of the x86 multiprocessor. The state machine is divided into two parts: the kernel state machine and the user state machine. The kernel state machine models the behavior of the kernel, while the user state machine models the behavior of user programs.

The x86-TSO model is designed to be easy to implement in software. It is based on a combination of ideas from the Mach virtual memory system and the x86 architecture. The model is designed to be both useful for hardware designers and easy to implement in software.

The x86-TSO model is based on a state machine that models the behavior of the x86 multiprocessor. The state machine is divided into two parts: the kernel state machine and the user state machine. The kernel state machine models the behavior of the kernel, while the user state machine models the behavior of user programs.

The x86-TSO model is designed to be easy to implement in software. It is based on a combination of ideas from the Mach virtual memory system and the x86 architecture. The model is designed to be both useful for hardware designers and easy to implement in software.

The x86-TSO model is based on a state machine that models the behavior of the x86 multiprocessor. The state machine is divided into two parts: the kernel state machine and the user state machine. The kernel state machine models the behavior of the kernel, while the user state machine models the behavior of user programs.

The x86-TSO model is designed to be easy to implement in software. It is based on a combination of ideas from the Mach virtual memory system and the x86 architecture. The model is designed to be both useful for hardware designers and easy to implement in software.
x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors

ARM

RISC-V

Trippel et al.
ASPLOS’17

...
ARM
Trippel et al.
ASPLOS’17

GPUs
Alglave et al.
ASPLOS’15

RISC-V
Accelerators

...
MemSynth: automated reasoning for memory models
MemSynth: automated reasoning for memory models

Verification
Check that model M allows litmus test T
MemSynth: automated reasoning for memory models

**Verification**
Check that model $M$ allows litmus test $T$

**Synthesis**
Complete a *framework sketch* $M$ to be correct for all tests $T$
MemSynth: automated reasoning for memory models

Verification
Check that model M allows litmus test T

Synthesis
Complete a framework sketch M to be correct for all tests T

Equivalence
Synthesize a test T on which two models differ
**MemSynth:**
**automated reasoning for memory models**

**Verification**
Check that model $M$ allows litmus test $T$

**Synthesis**
Complete a framework sketch $M$ to be correct for all tests $T$

**Equivalence**
Synthesize a test $T$ on which two models differ

**Ambiguity**
Determine if a model uniqueness explains a set of tests $T$
Ocelot: bounded relational logic in Rosette

- **formula** in relational logic (FOL, relations, transitive closure)
- **bounds** (partial model and types)
- **finite universe**
Ocelot: bounded relational logic in Rosette

**formula** in relational logic (FOL, relations, transitive closure)

**bounds** (partial model and types)

**finite universe**

**translator**

**ROSETTE**

**model**
Memory models as relational logic

<table>
<thead>
<tr>
<th>Initially $X = Y = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>$X = 1$</td>
</tr>
<tr>
<td>print $Y$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
Memory models as relational logic

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>print $Y$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
Memory models as relational logic

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>print $Y$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
Memory models as relational logic

Initially $X = Y = 0$

Thread 1

$X = 1$

print $Y$

Thread 2

$Y = 1$

print $X$

Can this program print two zeroes?
Memory models as relational logic

Initially $X = Y = 0$

Thread 1

$X = 1$

print $Y$

Thread 2

$Y = 1$

print $X$

Can this program print two zeroes?

No: there is a cycle in the happens-before graph
Memory models as relational logic

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>print $Y$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
Memory models as relational logic

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>print $Y$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

x86 allows store-load reordering

Can this program print two zeroes?
Memory models as relational logic

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>print $Y$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

x86 allows store-load reordering

Can this program print two zeroes?
### Memory models as relational logic

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td><code>print Y</code></td>
<td><code>print X</code></td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
Memory models as relational logic

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$ $X = 1$</td>
<td>$E_3$ $Y = 1$</td>
</tr>
<tr>
<td>$E_2$ print $Y$</td>
<td>$E_4$ print $X$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?
Memory models as relational logic

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$</td>
<td>$X = 1$</td>
</tr>
<tr>
<td>$E_2$</td>
<td>print $Y$</td>
</tr>
</tbody>
</table>

$\{\langle E_1, E_3 \rangle \} \subseteq \text{Write} \subseteq \{\langle E_1, E_3 \rangle \}$

$\{\langle E_2, E_4 \rangle \} \subseteq \text{Read} \subseteq \{\langle E_2, E_4 \rangle \}$

$\{\langle E_1, 1 \rangle, \langle E_2, 1 \rangle, \ldots \} \subseteq \text{thd} \subseteq \{\langle E_1, 1 \rangle, \langle E_2, 1 \rangle, \ldots \}$

$\ldots$

Can this program print two zeroes?
Memory models as relational logic

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$</td>
<td>$E_2$</td>
</tr>
<tr>
<td>$X = 1$</td>
<td>print $Y$</td>
</tr>
<tr>
<td>$E_3$</td>
<td>$E_4$</td>
</tr>
<tr>
<td>$Y = 1$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

$\{\langle E_1 \rangle, \langle E_3 \rangle \} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle \}$

$\{\langle E_2 \rangle, \langle E_4 \rangle \} \subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle \}$

$\{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots \} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots \}$

$\{} \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$

Can this program print two zeroes?
Memory models as relational logic

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$</td>
<td>$E_3$</td>
</tr>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>$E_2$</td>
<td>$E_4$</td>
</tr>
<tr>
<td>print $Y$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

$\forall e_i, e_j : \text{Event} |$
$\quad i < j \land e_i.thd = e_j.thd \Rightarrow \langle e_i, e_j \rangle \in \text{hb}$

Can this program print two zeroes?

$\{\langle E_1, E_3 \rangle \subseteq \text{Write} \subseteq \{\langle E_1, E_3 \rangle\}$
$\{\langle E_2, E_4 \rangle \subseteq \text{Read} \subseteq \{\langle E_2, E_4 \rangle\}$
$\{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots \} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots \}$
$\ldots$
$\{\} \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$
Memory models as relational logic

Initially \( X = Y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 ): ( X = 1 )</td>
<td>( E_3 ): ( Y = 1 )</td>
</tr>
<tr>
<td>( E_2 ): print ( Y )</td>
<td>( E_4 ): print ( X )</td>
</tr>
</tbody>
</table>

\[ \{\langle E_1 \rangle, \langle E_3 \rangle\} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle\} \]

\[ \{\langle E_2 \rangle, \langle E_4 \rangle\} \subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle\} \]

\[ \{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots \} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots \} \]

\[ \{\} \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\} \]

\[ \forall e_i, e_j : \text{Event} \mid i < j \land e_i.\text{thd} = e_j.\text{thd} \Rightarrow \langle e_i, e_j \rangle \in \text{hb} \]

\[ \ldots \]

\[ \text{no} \ ^{\text{\text{^hb}}} \cap \text{idem} \]
Memory models as relational logic

Initially \( X = Y = 0 \)

\[
\begin{array}{c|c}
\text{Thread 1} & \text{Thread 2} \\
\hline
E_1 & X = 1 \quad E_3 & Y = 1 \\
E_2 & \text{print } Y \quad E_4 & \text{print } X \\
\end{array}
\]

\[\{\langle E_1 \rangle, \langle E_3 \rangle \} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle \}\]
\[\{\langle E_2 \rangle, \langle E_4 \rangle \} \subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle \}\]
\[\{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots \} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots \}\]
\[\ldots\]
\[\emptyset \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}\]

\[\forall e_i, e_j : \text{Event} \mid i < j \land e_i.\text{thd} = e_j.\text{thd} \Rightarrow \langle e_i, e_j \rangle \in \text{hb}\]

\[\ldots\]

\[\neg \emptyset \cap \text{idem}\]
Memory models as relational logic

Initially \( X = Y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 )</td>
<td>( E_3 )</td>
</tr>
<tr>
<td>( X = 1 )</td>
<td>( Y = 1 )</td>
</tr>
<tr>
<td>( E_2 )</td>
<td>( E_4 )</td>
</tr>
<tr>
<td>print ( Y )</td>
<td>print ( X )</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?

\[
\{\langle E_1 \rangle, \langle E_3 \rangle\} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle\}
\]

\[
\{\langle E_2 \rangle, \langle E_4 \rangle\} \subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle\}
\]

\[
\{\langle E_1, 1 \rangle, \langle E_2, 1 \rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle E_1, 1 \rangle, \langle E_2, 1 \rangle, \ldots\}
\]

\[
\emptyset \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}
\]

\[
\forall e_i, e_j : \text{Event} \mid \quad i < j \land e_i.\text{thd} = e_j.\text{thd} \land \neg(e_i \in \text{Write} \land e_j \in \text{Read})
\]

\[
\Rightarrow \langle e_i, e_j \rangle \in \text{hb}
\]

\[
\ldots
\]

\[
\no \ ^{\text{hb}} \cap \text{idem}
\]

\( x_{86} \) ("total store order")
Initially $X = Y = 0$

Thread 1

$E_1$ $X = 1$

$E_2$ print $Y$

Thread 2

$E_3$ $Y = 1$

$E_4$ print $X$

Can this program print two zeroes?

$\{\langle E_1 \rangle, \langle E_3 \rangle \} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle \}$

$\{\langle E_2 \rangle, \langle E_4 \rangle \} \subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle \}$

$\{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots \} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots \}$

$\ldots$

$\} \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$

$\forall e_i, e_j : \text{Event} |$

$i < j \land e_i.\text{thd} = e_j.\text{thd} \land \neg(e_i \in \text{Write} \land e_j \in \text{Read})$

$\Rightarrow \langle e_i, e_j \rangle \in \text{hb}$

$\ldots$

$\text{no} \ ^{\wedge}\text{hb} \cap \text{idem}$
Initially \( X = Y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 ) ( X = 1 )</td>
<td>( E_3 ) ( Y = 1 )</td>
</tr>
<tr>
<td>( E_2 ) \text{print } Y</td>
<td>( E_4 ) \text{print } X</td>
</tr>
</tbody>
</table>

\( \forall e_i, e_j : \text{Event} \mid i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge ?? \Rightarrow \langle e_i, e_j \rangle \in \text{hb} \)

\( \ldots \)

\( \text{no } ^\text{hb} \cap \text{idem} \)
Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$</td>
<td>$E_3$</td>
</tr>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>$E_2$</td>
<td>$E_4$</td>
</tr>
<tr>
<td><code>print Y</code></td>
<td><code>print X</code></td>
</tr>
</tbody>
</table>

Can this program print two zeroes?

\[
\forall e_i, e_j : \text{Event} \mid
\quad i < j \land e_i\.thd = e_j\.thd \land ??
\quad \Rightarrow \langle e_i, e_j \rangle \in hb
\]

... 

\[\text{no } \quad \wedge \text{hb } \cap \text{idem} \]
Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$</td>
<td>$E_3$</td>
</tr>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>$E_2$</td>
<td>$E_4$</td>
</tr>
<tr>
<td>print $Y$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?

\[
\{\langle E_1 \rangle, \langle E_3 \rangle\} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle\} \\
\{\langle E_2 \rangle, \langle E_4 \rangle\} \subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle\} \\
\{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \ldots\} \\
\ldots \\
\{\} \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}
\]

$\forall e_i, e_j : \text{Event} | \\
i < j \land e_i.\text{thd} = e_j.\text{thd} \land \text{??} \\
\Rightarrow \langle e_i, e_j \rangle \in \text{hb}$

\ldots

no $^\wedge \text{hb} \cap \text{iden}$

**Spec:** model gives expected outcomes (allowed/not) on a set of litmus tests, from documentation or elsewhere
Can this program print two zeroes?

Initially \( X = Y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 )</td>
<td>( E_1 )</td>
</tr>
<tr>
<td>( X = 1 )</td>
<td>( Y = 1 )</td>
</tr>
<tr>
<td>( E_2 )</td>
<td>( E_3 )</td>
</tr>
<tr>
<td>\text{print } Y</td>
<td>\text{print } X</td>
</tr>
</tbody>
</table>

\[ \forall e_i, e_j : 	ext{Event} \mid i < j \land e_i.\text{thd} = e_j.\text{thd} \land \neg (e_i \in \text{Write} \land e_j \in \text{Read}) \Rightarrow \langle e_i, e_j \rangle \in \text{hb}_1 \]

\[ \ldots \]

\[ \neg \exists \text{hb}_1 \cap \text{idem} \]
Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$</td>
<td>$E_3$</td>
</tr>
<tr>
<td>$X = 1$</td>
<td>$Y = 1$</td>
</tr>
<tr>
<td>$E_2$</td>
<td>$E_4$</td>
</tr>
<tr>
<td>print $Y$</td>
<td>print $X$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?

\[ \forall e_i, e_j \text{ : Event } | \]
\[ i < j \land e_i.\text{thd} = e_j.\text{thd} \land \neg (e_i \in \text{Write} \land e_j \in \text{Read}) \]
\[ \Rightarrow \langle e_i, e_j \rangle \in \text{hb}_1 \]

...
Initially \( X = Y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 )</td>
<td>( E_3 )</td>
</tr>
<tr>
<td>( E_2 )</td>
<td>( E_4 )</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?

\[
\forall e_i, e_j : \text{Event} \mid i < j \land e_i.\text{thd} = e_j.\text{thd} \land \neg(e_i \in \text{Write} \land e_j \in \text{Read}) \land (e_i, e_j) \in h_{b1} \Rightarrow (e_i, e_j) \in h_{b1}
\]

... no \( h_{b1} \cap \text{idem} \)
Metasketches for memory models

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$</td>
<td>$E_3$</td>
</tr>
<tr>
<td>$E_2$</td>
<td>$E_4$</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?

\[\{\langle E_1, E_3 \rangle \} \subseteq \text{Write} \subseteq \{\langle E_1, E_2, E_3, E_4 \rangle \}\]
\[\emptyset \subseteq \text{Read} \subseteq \{\langle E_2, E_4 \rangle \}\]
\[\{\langle E_1, 1 \rangle, \langle E_2, 1 \rangle, \ldots \} \subseteq \text{thd} \subseteq \{\langle E_1, 1 \rangle, \langle E_2, 1 \rangle, \ldots \}\]
\[\emptyset \subseteq \text{hb}_1 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}\]
\[\emptyset \subseteq \text{hb}_2 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}\]

$\forall e_i, e_j : \text{Event} |$

\[i < j \land e_i.\text{thd} = e_j.\text{thd} \land \neg(e_i \in \text{Write} \land e_j \in \text{Read})\]
\[\Rightarrow \langle e_i, e_j \rangle \in \text{hb}_1\]

...
Metasketches for memory models

<table>
<thead>
<tr>
<th>Initially X = Y = 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$ ?? = ??</td>
<td>$E_3$ ?? = ??</td>
<td>$E_4$ ??</td>
</tr>
</tbody>
</table>

Can this program print two zeroes?

\[
\forall e_i, e_j : \text{Event} \mid \\
i < j \land e_i.\text{thd} = e_j.\text{thd} \land \neg (e_i \in \text{Write} \land e_j \in \text{Read}) \\
\Rightarrow \langle e_i, e_j \rangle \in \text{hb}_1
\]

...  
\[\text{no} \wedge \text{hb}_1 \cap \text{idem}\]
Metasketches for memory models

![Graph showing the comparison between MemSynth and Alloy in terms of problems solved over time per problem. The x-axis represents time per problem (s), ranging from 0 to 1000 seconds. The y-axis represents the number of problems solved, ranging from 0 to 40. The red line represents MemSynth, and the blue line represents Alloy. MemSynth solves more problems than Alloy at all time per problem values shown.]
Metasketches for memory models

Slowest problem \(~30\times\) faster than Alloy

Problems solved

Time per problem (s)
MemSynth can synthesize real models

x86

10 tests (Intel manual)
2 seconds

PowerPC

768 tests (existing work)
16 seconds
MemSynth can synthesize real models

**PowerPC**

```
(define ppo
  (& po (& (- (- (+ po dep)
    (& (join loc (~ loc))
      (-> Write Write)))
    (& (- po dep)
      (-> Read Write)))
  (- (- po (-> Write Event))
    (- (-> Read Read)
      (& po dep))))

(define grf
  (-> none none))

(define fences
  (+
    (^ (let ([poFpo (join (:> po Fence) po)])
          (+ (+ rf poFpo) (join poFpo poFpo))
          (+ (join poFpo Write) (join Write poFpo)))
    (+ (join (+ rf poFpo) (+ rf poFpo))
      (join (-> Read Write) (+ Read poFpo))))
    (^ (let ([poLWFpo (join (:> po Lwsyncs) po)]
        [RE+WW (+ (-> Read Event) (-> Write Write)]
        (+
          (+ (:: (RE+WW poLWFpo) Write)
              (join (& RE+WW poLWFpo) Write)
              (join Write (& RE+WW poLWFpo)))
          (join (+ Read Write)
            (+ (<: Write rf)
                (<: Write (& RE+WW poLWFpo)))))

10 tests (Intel manual)
2 seconds

768 tests (existing work)
16 seconds
MemSynth can find ambiguities in real models

**x86**

10 tests (Intel manual)
2 seconds

3 missing litmus tests
(Intel manual identifies at least 4 different models!)

**PowerPC**

768 tests (existing work)
16 seconds

10 missing litmus tests
(existing testing identifies at least 11 different models!)
Summary

Today

• Metasketches: scalable program synthesis
• MemSynth: a metasketch-based synthesis tool

Next lecture

• Project demos!
  • 7 groups, 11 minutes per group
• Please be on time!