Software Verification (Fall 2013)
Lecture 7: Graph-Based Reasoning and Verification

Chris Poskitt
Programs we’ve reasoned about so far: (1) store-manipulating programs

```
a := 5;
b := a*2;
c := a;
```
x := cons(3,3);
y := cons(4,4);
[x+1] := y;
[y+1] := x;

Programs we’ve reasoned about so far: (2) store+heap-manipulating programs
Programs we’ll reason about today:
graph-manipulating programs (!)
Programs we’ll reason about today: graph-manipulating programs (!)

- What is a “graph manipulation”? 
- Why reason about graphs? 
- How do we reason about them?
Manipulating graphs?

- creating a new graph out of another algorithmically
  - relabelling (including “marking” nodes/edges)
  - creation/deletion of structure
Manipulating graphs?

- creating a new graph out of another algorithmically
  - relabelling (including “marking” nodes/edges)
  - creation/deletion of structure

how?
One way to manipulate a graph

• graph as an abstract data type:
  
  \[ \text{adjacent}(G, v, w) \]
  \[ \text{addEdge}(G, v, w) \]
  \[ \text{deleteEdge}(G, v, w) \]
  etc.

with the graph data structure represented as
  e.g. an adjacency matrix or adjacency list

• implement graph algorithms
  e.g. Dijkstra’s shortest path

• reason about and verify them using separation logic
So, that’s all ... ?
(not entirely!)

• can use separation logic in reasoning, but:
  - significant sharing possible in graphs
    => proofs can become complicated

• the beauty of graphs is in their simplicity
  - lose some of this when worrying about representation

• efficiently implementing graph algorithms is not our only aim
  - abstraction facilitates high-level reasoning about conceptually difficult problems
Raise the abstraction!

• we will use graph transformation as a computational abstraction

• program states are graphs (in the mathematical sense)

• computational steps are applications of rules
  - akin to Chomsky string-rewriting rules, but for graphs
Example

linkNodes:

\[
\text{where not edge}(1,3)
\]
Example

linkNodes:

\[
\begin{array}{c}
\text{1} \rightarrow \text{2} \rightarrow \text{3} \\
\end{array}
\]

\Rightarrow

\[
\begin{array}{c}
\text{1} \rightarrow \text{2} \rightarrow \text{3} \\
\end{array}
\]

where not edge(1,3)
Example

linkNodes:

\[ \text{where not edge}(1,3) \]

\[ \Rightarrow \]

\[ \Rightarrow \text{linkNodes} \]
Example

linkNodes:

where not edge(1,3)

==>

nondeterministic!
Example

linkNodes:

where not edge(1,3)

nondeterministic!
Example

linkNodes:

```
1 -> 2 -> 3  =>  1 -> 2 -> 3
```

WHERE not edge(1,3)

```
1 2 3
```

=>

```
1 2 3
```

```
  
  
  
```

=⇒

```
  
  
  
```

=⇒

```
  
  
  
```

```
  
  
  
```

=⇒

```
  
  
  
```

```
  
  
  
```

nondeterministic!
A few of the application areas of graph transformation in CS:

- graph reduction in functional programming languages
- model-driven software development; semantics of UML
- checking shape safety of pointer manipulations
- visual modelling of structure/attribute-changing systems - e.g. a rule in a “mobile” system (Pennemann 09)
A few of the application areas of graph transformation in CS:

- graph reduction in functional programming languages
- model-driven software development; semantics of UML
- checking shape safety of pointer manipulations
- visual modelling of structure/attribute-changing systems - e.g. a rule in a “mobile” system (Pennemann 09)
Modelling is only half the story

• modelling problems as graphs and graph transformation rules is only “half the story”

• such visualisations aid in our understanding
  - intuitively express the relations between entities

but the use of such techniques alone does not guarantee correctness
Modelling is only half the story

• modelling problems as graphs and graph transformation rules is only “half the story”

• such visualisations aid in our understanding - intuitively express the relations between entities

but the use of such techniques alone does not guarantee correctness

\{ \text{pre ?} \} \begin{array}{c}
1 \xrightarrow{} 2 \\
2 \xrightarrow{} 3
\end{array} \Rightarrow \begin{array}{c}
1 \xrightarrow{} 2 \\
2 \xrightarrow{} 3
\end{array} \{ \text{post ?} \}
Verifying graph transformations

• a comprehensive theory for graph transformation has been developed since the 1970s
  - based on notions from category theory
  - (only an informal presentation today)

• a basis for sound formal reasoning and verification

• but verification research in the community only gained momentum in the last decade
  - model checking approaches (Rensink, Varró, König, ...)
  - weakest preconditions (Habel & Pennemann)
  - Hoare logic and attributes (Poskitt & Plump)
Verifying graph transformations

• a comprehensive theory for graph transformation has been developed since the 1970s
  - based on notions from category theory
  - (only an informal presentation today)

• a basis for sound formal reasoning and verification

• but verification research in the community only gained momentum in the last decade
  - model checking approaches (Rensink, Varró, König, ...)
  - weakest preconditions (Habel & Pennemann)
  - Hoare logic and attributes (Poskitt & Plump)
Next on the agenda

(1) a programming language for graphs

(2) an assertion language for graphs

(3) Hoare-style reasoning about graph transformation

(4) program proofs
A program state is a graph (and only a graph)

- not a store, not a heap, not any kind of mapping from variables to data

- just a graph

- label alphabet: (sequences of) integers and strings

- parallel edges and loops allowed
A program state is a graph
(and only a graph)

• not a store, not a heap, not any kind of mapping from variables to data

• just a graph

• label alphabet: (sequences of) integers and strings

• parallel edges and loops allowed

what about variables, counters, ... ?
A programming language based on graph transformation

• we will follow the syntax and semantics of GP 2
  - for “Graph Programs”
  - other languages: AGG, Fujaba, and GrGen

• graph programs comprise two components:

  (1) a set of graph transformation rules

  (2) a command sequence informing their application
Graph transformation rules

• rules comprise two graphs:

(1) a left-hand side $L$, describing what is to be matched

(2) a right-hand side $R$, describing what to replace the match with

- $L$ and $R$ are labelled over expressions

• rule can be equipped with a textual condition
  - expressing relations between labels, nodes
Example

\[ \text{bridge}(a, b, x, y, z: \text{list}) \]

\[ \xrightarrow{28} \]

where not edge(1, 3)
Example

**rule name**

bridge(a, b, x, y, z: list)

where not edge(1, 3)
Example

“variables”, instantiated during graph matching
(type list = sequences of ints and strings)

bridge(a, b, x, y, z: list)

where not edge(1, 3)
Example

“variables”, instantiated during graph matching
(type list = sequences of ints and strings)

bridge(a, b, x, y, z: list)

where not edge(1, 3)

Warning: program states (graphs) do not have variables
...but rules do!

Here they are placeholders, not references to a store
Example

bridge(a, b, x, y, z: list)

where not edge(1, 3)

new edge takes the label x’: z’
  where x |-> x’ and z |-> z’
Example

bridge(a, b, x, y, z: list)

where not edge(1, 3)

numbers indicate that the nodes are the same
Example

bridge\((a, b, x, y, z: \text{list})\)

where not edge\((1, 3)\)

\(\text{a condition: rule cannot be applied if there is an edge from matched node 1 to matched node 3}\)
Example rule application

bridge(a, b, x, y, z: list)

where not edge(1, 3)

let us apply bridge to this graph
Example rule application

bridge(a, b, x, y, z: list)

where not edge(1, 3)

let us apply bridge to this graph

find mapping \( \alpha : \text{Vars} \to \text{Data} \) such that:

1. there is a “match” for \( L^\alpha \) in the graph
2. the condition holds

(1) there is a “match” for \( L^\alpha \) in the graph
(2) the condition holds
Example rule application

bridge(a, b, x, y, z: list)

\[ L \]

\[ \begin{array}{ccc}
    & a & \\
    x & \rightarrow & y \\
    1 & & 2 \\
    b & \rightarrow & z \\
    & \rightarrow & 3 \\
\end{array} \]

\[ \Rightarrow \]

\[ \begin{array}{ccc}
    & a & \\
    x & \rightarrow & y \\
    1 & & 2 \\
    b & \rightarrow & z \\
    & \rightarrow & 3 \\
\end{array} \]

where not edge(1, 3)

\[ \begin{align*}
    \alpha & : & \\
    x & \rightarrow & 0:1:2 \\
    y & \rightarrow & 3 \\
    z & \rightarrow & 4 \\
\end{align*} \]

\[ \begin{array}{ccc}
    0:1:2 & \rightarrow & 3 \\
    & \rightarrow & 4 \\
    & \rightarrow & 2 \\
\end{array} \]
Example rule application

bridge(a, b, x, y, z: list)

where not edge(1, 3)

$L$

$x \rightarrow a \rightarrow y \rightarrow b \rightarrow z$

$\Rightarrow$

$x \rightarrow a \rightarrow y \rightarrow b \rightarrow z$

$L^\alpha$

$0:1:2 \rightarrow 3 \rightarrow 4$

$x \rightarrow 0:1:2 \quad a \rightarrow \text{blank}$

$y \rightarrow 3 \quad b \rightarrow \text{blank}$

$z \rightarrow 4$
Example rule application

bridge(a, b, x, y, z: list)

where not edge(1, 3)
Example rule application

bridge(a, b, x, y, z: list)

\[ L \]

\[ \begin{array}{ccc}
  & a & \\
 x & \rightarrow & y \\
 1 & & 2 \\
 & b & \\
 y & \rightarrow & z \\
 2 & & 3 \\
\end{array} \]

\[ \Rightarrow \]

\[ \begin{array}{ccc}
  & a & \\
 x & \rightarrow & y \\
 1 & & 2 \\
 & b & \\
 y & \rightarrow & z \\
 2 & & 3 \\
\end{array} \]

where not edge(1, 3)

\[ L^{\alpha} \]

\[ \begin{array}{ccc}
  & 0:1:2 & \\
 0:1:2 & \rightarrow & 3 \\
 1 & & 2 \\
 & 3 & \\
 3 & \rightarrow & 4 \\
 2 & & 3 \\
\end{array} \]

\[ \Rightarrow \]

\[ \begin{array}{ccc}
  & 0:1:2 & \\
 0:1:2 & \rightarrow & 3 \\
 1 & & 2 \\
 & 3 & \\
 3 & \rightarrow & 4 \\
 2 & & 3 \\
\end{array} \]

\[ \begin{array}{ccc}
  & x \triangleleft z \\
 0:1:2 & \rightarrow & 3 \\
 1 & & 2 \\
 & 3 & \\
 3 & \rightarrow & 4 \\
 2 & & 3 \\
\end{array} \]
Example rule application

bridge(a, b, x, y, z: list)

where not edge(1, 3)

\[ L \]

\[ L^\alpha \]
Example rule application

bridge(a, b, x, y, z: list)

\[ L \]

\[ x \rightarrow y \rightarrow z \]

\[ \Rightarrow \]

\[ x \rightarrow y \rightarrow z \]

where not edge(1, 3)

\[ L^\alpha \]

\[ 0:1:2 \rightarrow 3 \rightarrow 4 \]

\[ \Rightarrow \]

\[ 0:1:2 \rightarrow 3 \rightarrow 4 \]

\[ \Rightarrow \]

\[ 0:1:2 \rightarrow 3 \rightarrow 4 \]

\[ \Rightarrow \]

\[ 0:1:2 \rightarrow 3 \rightarrow 4 \]
Example rule application

bridge(a, b, x, y, z: list)

where not edge(1, 3)
Rule application is nondeterministic

bridge \( (a, b, x, y, z: \text{list}) \)

where not edge \((1, 3)\)

\[
\begin{array}{ccc}
\text{x} & \text{a} & \text{y} & \text{b} & \text{z} \\
1 & 2 & 3 \\
\end{array}
\Rightarrow
\begin{array}{ccc}
\text{x} & \text{a} & \text{y} & \text{b} & \text{z} \\
1 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{ccc}
0:1:2 & 3 & 4 \\
\end{array}
\Rightarrow
\begin{array}{ccc}
\text{2} \\
\end{array}
\]

\[\text{rule application is nondeterministic} \]
\[\text{- what other graph could result?}\]
Deleting nodes

• we can create nodes/edges and relabel without issue

• we can even delete edges without issue

• can we arbitrarily delete nodes?
Deleting nodes

clip (x, y: int)

\[ \text{clip} \ (x, y: \text{int}) \]

\[ \begin{array}{c}
\text{x} \\
| \\
\rightarrow \\
| \\
\text{y} \\
\end{array} \Rightarrow \begin{array}{c}
\text{x} \\
| \\
\end{array} \]
Deleting nodes

clip \((x,y: \text{int})\)

\[
\begin{array}{c}
\text{x} \\
\downarrow \\
\text{y}
\end{array}
\quad \Rightarrow 
\begin{array}{c}
\text{x}
\end{array}
\]

\[
\begin{array}{c}
\text{7} \\
\downarrow \\
\text{5}
\end{array}
\quad \Rightarrow_{\text{clip}} 
\begin{array}{c}
\text{6}
\end{array}
\]
Deleting nodes

clip (x,y: int)

\[
\begin{array}{c}
\text{x} \\
\text{y}
\end{array}
\]

=>

\[
\begin{array}{c}
\text{x}
\end{array}
\]

not a graph!
Deleting nodes: a solution

- only allow rule applications that do not leave edges dangling

- satisfy the “dangling condition”

- called the double-pushout approach (DPO) to graph transformation

  - key property: rule applications are side-effect free
Deleting nodes: notation alert!

trickyRule (x: int)

\[
\begin{array}{c}
\text{x} \\
\Rightarrow \\
\text{x}
\end{array}
\]
Deleting nodes: notation alert!

trickyRule \((x: \text{int})\)

no number implies nodes are not the same

\(\text{i.e. match of } L^\alpha \text{ is deleted, then recreated with the same label}\)
No matches => failure

trickyRule (x: int)

\[
\begin{array}{c}
\text{x} \\
\Rightarrow \\
\text{x}
\end{array}
\]

\[
\begin{array}{c}
7 \\
5 \\
\Rightarrow \text{trickyRule}
\end{array}
\]
No matches => failure

trickyRule (x: int)

\[
\begin{array}{c}
\text{x} \\
\Rightarrow \\
\text{x}
\end{array}
\]

\[
\begin{array}{c}
7 \\
\Rightarrow \text{trickyRule} \\
5 \\
\Rightarrow 6
\end{array}
\]

fail
No matches => failure

trickyRule (x: int)

x => x

no match since an edge would be left dangling - program fails and terminates

7 => trickyRule

5 => 6

fail
Graph programs: control constructs

- simple core of control constructs

\[
\begin{array}{ll}
{\{r_0, \ldots, r_n\}} & \text{single rule application} \\
{r} & \text{nondeterministic rule choice} \\
{P; Q} & \text{sequential composition} \\
{\{r_0, \ldots, r_n\}!} & \text{as-long-as-possible iteration} \\
\end{array}
\]

- if \{r_0, ..., r_n\} then P else Q
- try \{r_0, ..., r_n\} then P else Q
Graph programs: control constructs

- simple core of control constructs

\[
\begin{align*}
& r \\
& \{r_0, \ldots, r_n\} \\
& \{r_0, \ldots, r_n\}! \\
& P; Q \\
& \text{if } \{r_0, \ldots, r_n\} \text{ then } P \text{ else } Q \\
& \text{try } \{r_0, \ldots, r_n\} \text{ then } P \text{ else } Q \\
\end{align*}
\]

- single rule application
- nondeterministic rule choice
- sequential composition
- as-long-as-possible iteration

branch decided on whether guard fails or not
- in “if”, the effects of guard not retained
- in “try”, the effects of guard are retained
Example graph program: compute a colouring

\[\text{colouring} = \text{init!}; \text{inc!}\]

\[\text{init}(x: \text{atom}) \quad \Rightarrow \quad \text{x:0}\]

\[\text{inc}(i: \text{int}; k: \text{list}; x, y: \text{atom}) \quad \Rightarrow \quad \text{x:i} \xrightarrow{k} \text{y:i+1}\]

**NB:** type “atom” denotes an int or string; type “list” denotes an arbitrary sequence of ints/strings
colouring = init!; inc!

init(x: atom)

\[
\begin{align*}
\text{x} & \quad \Rightarrow \quad \text{x:0} \\
1 & \quad 1
\end{align*}
\]

inc(i: int; k: list; x, y: atom)

\[
\begin{align*}
x:i \quad k & \quad y:i \\
1 & \quad 2 & \Rightarrow & \quad x:i \quad k \quad y:i+1 \\
1 & \quad 2
\end{align*}
\]

Example 2.43

The program initially colours each node with 0 by applying the `init` rule for as long as possible, using the iteration operator `!`. It then produces a colouring (an assignment of integers) for every input node. Figure 2.19 produces a colouring (an assignment of integers) for every input graph that is connected; and the third reduces an input graph to check whether it was a tree or not.

In this subsection, we present some example programs in order to compute a graph colouring; the second marks nodes in order to give some intuition into graph programming, before later introducing a formal operational semantics. The first program manipulates the list components of nodes, that adjacent nodes have different colours, and increments the colour of the target node by 1 otherwise. That the iteration of `inc` would be applied again and the iteration would continue.

Example 2.43 (Computing a graph colouring)

The program was a tree or not.
colouring = init!; inc!

\[ \text{init}(x : \text{atom}) \]

\[ x \quad \Rightarrow \quad x : 0 \]

\[ \text{inc}(i : \text{int}; k : \text{list}; x, y : \text{atom}) \]

\[ x : i \xrightarrow{k} y : i \quad \Rightarrow \quad x : i \xrightarrow{k} y : i + 1 \]

Example 2.43

In this subsection, we present some example programs in order to compute a graph colouring; the second marks nodes in order to check connectedness; and the third reduces an input graph to check whether it was a tree or not.

The program initially colours each node with 0 by applying the rule

\[ x \xrightarrow{\text{init}} x : 0 \]

for as long as possible, which matches adjacent nodes with the same colour, and increments the colour of the target node by 1 if adjacent nodes have different colours) for every input.

\[ \begin{align*}
1 & \quad 3 \\
3 & \quad 1 \\
3 & \quad 1 \\
3 & \quad 0
\end{align*} \]

The program

\[ \text{inc}(i : \text{int}; k : \text{list}; x, y : \text{atom}) \]

\[ x : i \xrightarrow{k} y : i \quad \Rightarrow \quad x : i \xrightarrow{k} y : i + 1 \]

would be applied again and the iteration would continue.
colouring = init!; inc!

init(x: atom)

inc(i: int; k: list; x, y: atom)
Next on the agenda

(1) a programming language for graphs

(2) an assertion language for graphs

(3) Hoare-style reasoning about graph transformation

(4) program proofs
An assertion language for graphs

- we could use some first-order logic interpreted over graphs

\[ \exists v. \text{node}(v) \land l(v) = "party" \]

- but program states are graphs, and computational steps are graph transformation rules
  - both at a high-level of abstraction

- define an assertion language at the same level of abstraction
  - facilitate visual specifications
  - integrate the theory of graph transformation into assertional reasoning
E-conditions: graphical assertions about the program state

- **E-conditions** - a logical assertion language embedding pictures of graphs

- **idea**: express the existence of some subgraph with particular structural features and particular relations between labels

- combine with Boolean negation and connectives for a visual logic equivalent to first-order logic over graphs

- **historical note**: a generalisation of nested conditions (Habel & Pennemann)
E-conditions: graphical assertions about the program state

\[ \exists (C \mid \gamma, c') \]

true
E-conditions: graphical assertions about the program state

\[ \exists (C \mid \gamma, c') \]

true

a constant symbol
E-conditions: graphical assertions about the program state

true

$\exists(C \mid \gamma, c')$

$C$ is a graph labelled over expressions
E-conditions: graphical assertions about the program state

true

$\exists (C \mid \gamma, c')$

$\gamma$ is a constraint over the labels of $C$
E-conditions: graphical assertions about the program state

$c'$ is an E-condition, i.e. true or $\exists (C' \mid \gamma', c'')$
Semantics by example

true

satisfied by all graphs
Semantics by example

\[ \exists (x) \]

“There exists a node incident to a loop”
Semantics by example

“there exists a node incident to a loop”
Semantics by example

"there exists a node incident to a loop"
Boolean expressions over E-conditions are also E-conditions

\[ \lnot \exists (x) \]

“there does not exist a node incident to a loop”
Expressions as labels are important, too!

\[ \neg \exists (x \xrightarrow{k} x) \]
Expressions as labels are important, too!


\[ \neg \exists (x \xrightarrow{k} x) \]

“there does not exist a pair of adjacent nodes with the same label”
Expressions as labels are important, too!

"there does not exist a pair of adjacent nodes with the same label"

\[ \neg \exists (k(x, x)) \]

“there does not exist a pair of adjacent nodes with the same label”
Expressions as labels are important, too!

\[ \neg \exists ( k \xrightarrow{k} x \mid k = x \ast x ) \]

now \( k \) is constrained

read aloud as “such that”
Expressions as labels are important, too!

¬∃((x \xrightarrow{k} x) \mid k = x \times x)

“there does not exist a pair of adjacent nodes such that:
(1) the nodes have the same label; and
(2) the edge is the square of that label”
Chapter 4. Verification with E-Conditions

This E-condition is satisfied by any graph in which every node \(i\) is incident to an outgoing edge, but not incident to more than one. Numbers are used to indicate which nodes are the same down the levels of nesting; here, the node labelled \(x\) (which is universally quantified) appears in the nested E-condition where information about its required context is given. Again, the \(k\)s and \(y\)s in the nested E-condition are not bound together, but this time, the \(x\)s are bound. Once a variable is used within an E-condition, every occurrence of it down the levels of nesting evaluates to the same value under an assignment. It can help to think of this nesting in E-conditions as a tree of (injective) graph morphisms equipped with Boolean symbols, as in Figure 4.1.

\[
\exists(x_1 \overset{k}{\rightarrow} y \mid x > y) \land \neg \exists(z \mid z < 0)
\]
Constraints enforce relations between labels

\[
\exists( \exists_k (x_1 \rightarrow y \mid x > y) \land \neg \exists (\exists z \mid z < 0))
\]

“there is a pair of adjacent nodes with source label greater than target label AND no node is labelled with a negative number”
Nesting allows assertions about specific contexts

\[\forall (x_1 \xrightarrow{k} y_2, \exists (x_1 \xrightarrow{k} y_2))\]
Nesting allows assertions about specific contexts

\[ \forall(x_1 \xrightarrow{k} y_1, \exists(x_1 \xrightarrow{j} y_2)) \]

these nodes and the k-labelled edge are the same

- nesting allows us to express properties about particular contexts
  i.e. express something about the universally quantified graph
Nesting allows assertions about specific contexts

\[ \forall (x_1 \rightarrow^k y_2, \exists (x_1 \rightarrow^k y_2)) \]

“if there is an edge from \( v \) to \( w \), then there is also an edge from \( w \) to \( v \) (graph is undirected)”
View as a “tree” that is building up structural and relational information

\[ \forall (x_1, \exists (x_1 \to y) \land \exists (x_1 \to y)) \]

\[ \forall (x_1, \exists (x_1 \to y) \land \exists (x_1 \to y)) \]

\[ \forall (x_1, \exists (x_1 \to y) \land \exists (x_1 \to y)) \]

\[ \forall (x_1, \exists (x_1 \to y) \land \exists (x_1 \to y)) \]
View as a “tree” that is building up structural and relational information

\[ \forall (x_1, \exists (x_1 \xrightarrow{k} y)) \land \neg \exists (x_1 \xrightarrow{j} y) \]

\[ \emptyset \]

\[ \forall \]

\[ x_1 \]

\[ \exists \]

\[ \neg \exists \]

\[ x_1 \xrightarrow{k} y \]

\[ x_1 \xrightarrow{j} y \]
View as a “tree” that is building up structural and relational information

\[ \forall (x_1, \exists (x_1 \rightarrow k y) \land \neg \exists (x_1 \rightarrow j y)) \]

for every node,
(1) there exists an outgoing edge to another node; and
(2) there does not exist a pair of outgoing edges to another node
Satisfaction of E-conditions
(an incomplete definition)

• a graph $G$ satisfies an E-condition $c$, written $G \models c$, if:

  (1) $c = \text{true}$; or
Satisfaction of E-conditions
(an incomplete definition)

• a graph $G$ satisfies an E-condition $c$, written $G \models c$, if:

  (1) $c = \text{true}$; or

  (2) $c = \exists (C \mid \gamma, c')$

and there is a mapping $\alpha: \text{Vars} \rightarrow \text{Data}$ such that

  (a) $|\gamma|\alpha = \text{true}$;

  (b) $C^\alpha$ is a subgraph of $G$;

  (c) the context of $C^\alpha$ in $G$ “satisfies” $c'$
Satisfaction of E-conditions
(an incomplete definition)

• a graph $G$ satisfies an E-condition $c$, written $G \models c$, if:

  (1) $c = \text{true}$; or

  (2) $c = \exists (C \mid \gamma, c')$

  and there is a mapping $\alpha : \text{Vars} \to \text{Data}$ such that

  (a) $[\gamma] \alpha = \text{true}$;

  (b) $C^\alpha$ is a subgraph of $G$;

  (c) the context of $C^\alpha$ in $G$ “satisfies” $c'$

the complete formal definition needs the notion of “morphism” for the inductive part (c)
Satisfaction by example

\[
\models \exists (x \mid \text{int}(x))
\]
Satisfaction by example

\[ \models \exists (x \mid \text{int}(x)) \]

by assignment \( x \mapsto 6 \), \( k \mapsto 2 \)
What can E-conditions not specify?

- E-conditions are expressively equivalent to a first-order logic interpreted over graphs.

- For relations between labels, this is great...

- But for graphs, first-order logic is quite weak for expressing structure:
  - Only “local” properties
  - Need more than FO for path properties, connectedness...

What prevents us from simply adding predicates for these properties?
Next on the agenda

(1) a programming language for graphs

(2) an assertion language for graphs

(3) Hoare-style reasoning about graph transformation

(4) program proofs
Partial correctness specifications

- **partial correctness**: if program $P$ is executed on a graph $G$ such that $G \models \text{pre}$, then if a graph $H$ results, $H \models \text{post}$

- differs to the partial correctness definition in lecture 2:
  - **nondeterminism**: many graphs $H$ could result, but all guaranteed to satisfy $\text{post}$
  - $P$ might fail on $G$
Proof rules

[ruleset] \[ \{c\} \ r \ \{d\} \quad \text{for each } r \in \mathcal{R} \]

\[ \{c\} \ \mathcal{R} \ \{d\} \]

where: \( \mathcal{R} = \{r_0, \ldots, r_n\} \)
Proof rules

\[ [\text{comp}] \quad \frac{\{c\} \; P \; \{e\} \; \{e\} \; Q \; \{d\} }{\{c\} \; P; \; Q \; \{d\} } \]
Proof rules

\[
[!] \quad \frac{\{inv\} \mathcal{R} \{inv\}}{\{inv\} \mathcal{R}! \{inv \land \neg \text{App}(\mathcal{R})\}}
\]

where \(\text{App}(\mathcal{R})\) constructs an E-condition expressing that \(\mathcal{R}\) will not fail on the graph
Proof rules

[if] \[ \left\{ c \land \text{App}(\mathcal{R}) \right\} P \{ d \} \quad \left\{ c \land \neg\text{App}(\mathcal{R}) \right\} Q \{ d \} \]
\[ \{ c \} \text{ if } \mathcal{R} \text{ then } P \text{ else } Q \{ d \} \]

[try] \[ \left\{ c \land \text{App}(\mathcal{R}) \right\} \mathcal{R}; P \{ d \} \quad \left\{ c \land \neg\text{App}(\mathcal{R}) \right\} Q \{ d \} \]
\[ \{ c \} \text{ try } \mathcal{R} \text{ then } P \text{ else } Q \{ d \} \]
Proof rules

[cons] \[ \begin{array}{c} c \Rightarrow c' \\
\{c'\} P \{d'\} \\
\{c\} P \{d\} \\
d' \Rightarrow d \end{array} \]

how to show the validity of an E-condition?
Axioms

\[
[\text{nonapp}] \quad \frac{\{\neg \text{App} \{\{r\}\}\} \ r \ \{\text{false}\}}{}
\]
Axioms

\[
[ruleapp] \frac{\{\text{Pre}(r, c)\} \quad r \quad \{c\}}{
}\text{where Pre}(r,c) \text{ constructs an E-condition expressing the weakest precondition that must hold for } r \text{ to establish } c}
\]
What does $\text{App}(R)$ look like? (see Poskitt 13 for the construction)

reduce($a,b,c: \text{int}$)

\[ \begin{array}{c}
0 & \longrightarrow & c \\
\circ & \quad & \bullet \\
a & \longrightarrow & b
\end{array} \quad \Rightarrow \quad \begin{array}{c}
0 & \longrightarrow & \\
\circ & \quad & \bullet \\
a & \longrightarrow & \\
\end{array} \]

$a < b$ and $b < c$

an E-condition equivalent to the following is constructed by $\text{App}($reduce$)$:
What does \text{App}(R) look like?  
(see Poskitt 13 for the construction)

\begin{align*}
\text{reduce}(a,b,c: \text{int}) \\
\begin{tikzpicture}
\node (a) at (0,0) [circle,draw] {$a$};
\node (b) at (1.5,0) [circle,draw] {$b$};
\node (c) at (3,0) [circle,draw] {$c$};
\draw (a) -- node[below] {$1$} (b);
\draw (b) -- node[below] {$2$} (c);
\end{tikzpicture} \\
a < b \text{ and } b < c
\end{align*}

an E-condition equivalent to the following is constructed by \text{App}(\text{reduce}):

\begin{align*}
&\exists (\begin{tikzpicture}
\node (a) at (0,0) [circle,draw] {$a$};
\node (b) at (1.5,0) [circle,draw] {$b$};
\node (c) at (3,0) [circle,draw] {$c$};
\draw (a) -- node[below] {$1$} (b);
\draw (b) -- node[below] {$2$} (c);
\end{tikzpicture} | a < b \text{ and } b < c, \\
&\neg \exists (\begin{tikzpicture}
\node (a) at (0,0) [circle,draw] {$a$};
\node (b) at (1.5,0) [circle,draw] {$b$};
\node (c) at (3,0) [circle,draw] {$b$};
\node (x) at (3,0.5) [circle,draw] {$x$};
\draw (a) -- (b);
\draw (b) -- (x);
\end{tikzpicture}) \land \neg \exists (\begin{tikzpicture}
\node (a) at (0,0) [circle,draw] {$a$};
\node (b) at (1.5,0) [circle,draw] {$b$};
\node (c) at (3,0) [circle,draw] {$c$};
\node (x) at (3,0.5) [circle,draw] {$x$};
\draw (a) -- (b);
\draw (b) -- (x);
\end{tikzpicture}) \land \neg \exists (\begin{tikzpicture}
\node (a) at (0,0) [circle,draw] {$a$};
\node (b) at (1.5,0) [circle,draw] {$b$};
\node (c) at (3,0) [circle,draw] {$b$};
\node (x) at (3,0.5) [circle,draw] {$x$};
\draw (a) -- (b);
\draw (b) -- (x);
\end{tikzpicture}) \\
&\land \neg \exists (\begin{tikzpicture}
\node (a) at (0,0) [circle,draw] {$a$};
\node (b) at (1.5,0) [circle,draw] {$b$};
\node (c) at (3,0) [circle,draw] {$b$};
\node (x) at (3,0.5) [circle,draw] {$x$};
\draw (a) -- (b);
\draw (b) -- (x);
\end{tikzpicture}) \land \neg \exists (\begin{tikzpicture}
\node (a) at (0,0) [circle,draw] {$a$};
\node (b) at (1.5,0) [circle,draw] {$b$};
\node (c) at (3,0) [circle,draw] {$b$};
\node (x) at (3,0.5) [circle,draw] {$x$};
\draw (a) -- (b);
\draw (b) -- (x);
\end{tikzpicture})
\end{align*}
What does App(R) look like? (see Poskitt 13 for the construction)

reduce(a,b,c: int)

\[ \Gamma \equiv \exists \{ a \rightarrow b \} \land \exists \{ b \rightarrow c \} \]

a < b and b < c

an E-condition equivalent to the following is constructed by App(reduce):

\[ \exists( a_1 \rightarrow b_2 ) \mid a < b \text{ and } b < c, \]

\[ \neg \exists( a_1 \rightarrow b_2 ) \land \neg \exists( a_1 \rightarrow b_2 ) \land \neg \exists( a_1 \rightarrow b_2 ), \]

\[ \land \neg \exists( a_1 \rightarrow b_2 ) \land \neg \exists( a_1 \rightarrow b_2 ) \land \neg \exists( a_1 \rightarrow b_2 ) \]

context satisfies dangling condition!
What does $\text{Pre}(r,c)$ look like?
(see Poskitt 13 for the construction)

\[
\begin{align*}
\text{init}(x: \text{atom}) \\
\text{take init as } r
\end{align*}
\]

\[
\forall (a_1, \exists (a_1 \mid \text{atom}(a)) \lor \exists (a_1 \mid a = b:c \text{ and } \text{atom}(b) \text{ and } c \geq 0))
\]

\[
\text{take as postcondition c}
\]
“every node is either labelled by (1) an atom; or (2) a sequence $b:c$ with $b$ an atom, $c$ a natural
What does \( \text{Pre}(r,c) \) look like?
(see Poskitt 13 for the construction)

\[
\forall( \exists_1 | \text{atom}(x), \\
\forall( \exists_1 \exists_2, \exists( \exists_1 \exists_2 | \text{atom}(a)) \\
\forall \exists( \exists_1 \exists_2 | a = b \cdot c \text{ and } \text{atom}(b) \text{ and } c \geq 0)) \\
\land \forall( \exists_1 , \exists( \exists_1 | \text{atom}(x:0)) \\
\forall \exists( \exists_1 | x:0 = b \cdot c \text{ and } \text{atom}(b) \text{ and } c \geq 0)))
\]

take home point:
embeds the left-hand graph of \( r \) and the postcondition \( c \) together in a precondition
Instance of [ruleapp] axiom for init, c

\[
\forall (\mathbf{\text{x}}_1 \mid \text{atom}(x)), \quad \\
\forall (\mathbf{\text{x}}_1 \mathbf{\text{a}}_2, \exists (\mathbf{\text{x}}_1 \mathbf{\text{a}}_2 \mid \text{atom}(a))) \\
\vee \exists (\mathbf{\text{x}}_1 \mathbf{\text{a}}_2 \mid a = b:c \text{ and } \text{atom}(b) \text{ and } c \geq 0) \\
\wedge \forall (\mathbf{\text{x}}_1, \exists (\mathbf{\text{x}}_1 \mid \text{atom}(x:0))) \\
\vee \exists (\mathbf{\text{x}}_1 \mid x:0 = b:c \text{ and } \text{atom}(b) \text{ and } c \geq 0))
\]

init

\[
\forall (\mathbf{\text{a}}_1, \exists (\mathbf{\text{a}}_1 \mid \text{atom}(a))) \vee \exists (\mathbf{\text{a}}_1 \mid a = b:c \text{ and } \text{atom}(b) \text{ and } c \geq 0))
\]
Classic Hoare logic vs. graph program

Hoare logic

• for the most part, classic and graph-based Hoare logic are very similar

• but interestingly, in “raising the abstraction” of programs and assertions, we make the core axiom of the Hoare logic very technical / complicated

  - compare to the simplicity of the assignment axiom

• motivates tool support, especially for generating $\text{App}(R)$, $\text{Pre}(r,c)$, and for deciding implications that have them as consequences
Next on the agenda

(1) a programming language for graphs

(2) an assertion language for graphs

(3) Hoare-style reasoning about graph transformation

(4) program proofs
Partial correctness of colouring

colouring = init!; inc!

init(x: atom)

init(x: atom)

inc(i: int; k: list; x, y: atom)

inc(i: int; k: list; x, y: atom)
Partial correctness of colouring

\[
\text{colouring} = \text{init!; inc!}
\]

\[
\begin{align*}
\text{init}(x: \text{atom}) & \quad \Rightarrow \\
\begin{array}{c}
x \\
1
\end{array} & \quad \begin{array}{c}
x:0 \\
1
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{inc}(i: \text{int}; k: \text{list}; x, y: \text{atom}) & \quad \Rightarrow \\
\begin{array}{c}
x:i \quad k \quad y:i \\
1 \quad 2
\end{array} & \quad \begin{array}{c}
x:i \quad k \quad y:i+1 \\
1 \quad 2
\end{array}
\end{align*}
\]

\[
\begin{align*}
\{ & \forall( a_1, \exists( a_1 \mid \text{atom}(a))) \\
\& \text{init!; inc!} & \\
\{ & \forall( a_1, \exists( a_1 \mid a = b:c \text{ and } \text{atom}(b) \text{ and } c \geq 0)) \\
& \land \neg \exists( x:i \quad k \quad y:i \mid \text{atom}(x,y) \text{ and } \text{int}(i))
\}
\end{align*}
\]
5.1. Computing a Graph Colouring

\[ \text{Pre}(\text{init}, e) \vdash \text{init} \{ e \} \]

\[ \{ e \} \vdash \text{init} \{ e \} \]

\[ \{ e \} \vdash \text{init}! \{ e \land \lnot \text{App}(\{\text{init}\}) \} \]

\[ \{ c \} \vdash \text{init}! \{ d \} \]

\[ \vdash \text{par} \{ c \} \vdash \text{init}!; \text{inc}! \{ d \land \lnot \text{App}(\{\text{inc}\}) \} \]

\[ \{ \text{Pre}(\text{inc}, d) \} \vdash \text{inc} \{ d \} \]

\[ \{ d \} \vdash \text{inc} \{ d \} \]

\[ \{ d \} \vdash \text{inc}! \{ d \land \lnot \text{App}(\{\text{inc}\}) \} \]

\[ \vdash \text{par} \{ c \} \vdash \text{init}!; \text{inc}! \{ d \land \lnot \text{App}(\{\text{inc}\}) \} \]
\[
\begin{align*}
\text{[ruleapp]} & \quad \frac{\text{[Pre}(\text{init}, e)\} \text{ init } \{e\}} {\{e\} \text{ init } \{e\}} \\
\text{[cons]} & \quad \frac{\{e\} \text{ init } \{e\}} {\{c\} \text{ init } \{d\}} \\
\text{[!] } & \quad \frac{\{e\} \text{ init! } \{e \land \neg \text{App}([\text{init}])\}} {\{d\} \text{ inc! } \{d \land \neg \text{App}([\text{inc}])\}} \\
\text{[comp]} & \quad \frac{\text{par } \{c\} \text{ init!}; \ \text{inc! } \{d \land \neg \text{App}([\text{inc}])\}} \\

\end{align*}
\]

\[
c = \forall( a_1, \exists( a_1 \mid \text{atom}(a))
\]

\[
d = \forall( a_1, \exists( a_1 \mid a = b : c \ \text{and} \ \text{atom}(b) \ \text{and} \ c \geq 0))
\]

\[
\neg \text{App}([\text{inc}]) = \neg \exists( x : i \xrightarrow{k} y : i \mid \text{atom}(x, y) \ \text{and} \ \text{int}(i))
\]


\[ \text{Pre}(	ext{init}, e) \] \[
\{ \text{init} \} \text{ init} \{ e \} \]

\[ \{ e \} \text{ init!} \{ e \wedge \neg \text{App}([\text{init}]) \} \]

\[ \{ c \} \text{ init!} \{ d \} \]

\[ \{ d \} \text{ inc!} \{ d \wedge \neg \text{App}([\text{inc}]) \} \]

\[ \vdash \text{par} \{ c \} \text{ init!}; \text{ inc!} \{ d \wedge \neg \text{App}([\text{inc}]) \} \]

\[
c = \forall (\exists_i, \exists(\exists_i | \text{atom}(a)))
\]

\[
d = \forall (\exists_i, \exists(\exists_i | a = b : c \text{ and } \text{atom}(b) \text{ and } c \geq 0))
\]

\[
e = \forall (\exists_i, \exists(\exists_i | \text{atom}(a))
\]

\[\vee \exists(\exists_i | a = b : c \text{ and } \text{atom}(b) \text{ and } c \geq 0)\]

\[
\neg \text{App}([\text{init}]) = \neg \exists(\exists_i | \text{atom}(x))
\]

\[
\neg \text{App}([\text{inc}]) = \neg \exists(\xrightarrow{i} y | \text{atom}(x, y) \text{ and } \text{int}(i))
\]
Next on the agenda

(1) a programming language for graphs

(2) an assertion language for graphs

(3) Hoare-style reasoning about graph transformation

(4) program proofs
The full picture

• many technical details hidden “under the carpet”
  - impossible to cover everything in the assigned time

• the full picture is quite interesting (I think!)
  - references will be added to the course webpage
  - but these are of course \textit{optional readings}
  - the exercises on Wednesday will make clear the level of understanding I aimed for
Summary

• motivated the study of graph-manipulating programs and discussed some applications

• introduced the notion of graph transformation: program states as graphs; steps as rules

• considered a programming language for modelling problems as graph transformations

• presented an overview of an assertion language and Hoare logic for proving properties about graph structure and relations between labels
Ongoing work

- reasoning about arbitrary-length path properties
- graph-based semantics for concurrency models
Thank you! Questions?

Next lecture:

• data flow analysis (with Sebastian Nanz)