Rule-Based Graph Programs

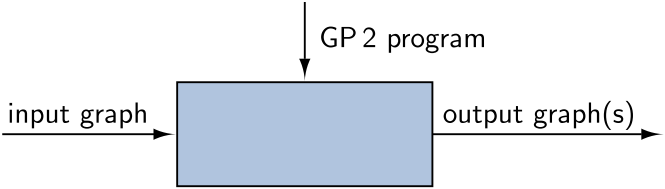
Detlef Plump

University of York

in cooperation with Tim Atkinson, Chris Bak, Graham Campbell, Brian Courtehoute, Mike Dodds, Ivaylo Hristakiev, Chris Poskitt, Sandra Steinert and Gia Wulandari

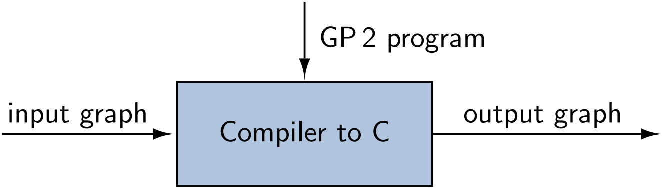
Overview

Graph Programming Language GP 2



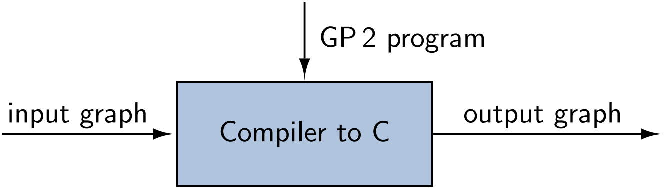
- Experimental DSL for graphs
- Based on graph-transformation rules
- Abstracts from low-level data structures
- Non-deterministic
- Computationally complete

Graph Programming Language GP 2



- Experimental DSL for graphs
- Based on graph-transformation rules
- Abstracts from low-level data structures
- Non-deterministic
- Computationally complete

Graph Programming Language GP 2



- Experimental DSL for graphs
- Based on graph-transformation rules
- Abstracts from low-level data structures
- Non-deterministic
- Computationally complete

Aim: facilitating formal reasoning while supporting practical problem solving

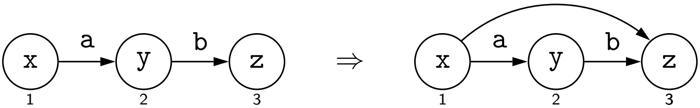
Example program: transitive closure

A graph is *transitive* if for every directed path $v \rightsquigarrow v'$ with $v \neq v'$, there is an edge $v \to v'$.

Program for computing a *transitive closure* of the input graph (smallest transitive extension):

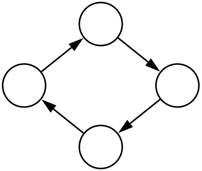
$\texttt{Main} = \texttt{link}!$

$\texttt{link}(\mathrm{a}, \mathrm{b}, \mathrm{x}, \mathrm{y}, \mathrm{z} \colon \texttt{list})$

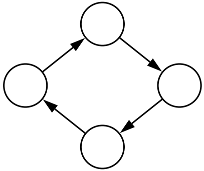


$\texttt{where not edge}(1, 3)$

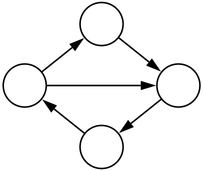
Example program: transitive closure (cont'd)



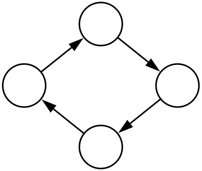
Example program: transitive closure (cont'd)



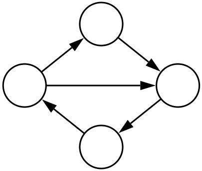
$\Rightarrow$



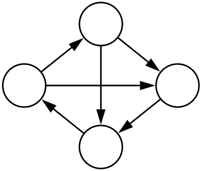
Example program: transitive closure (cont'd)



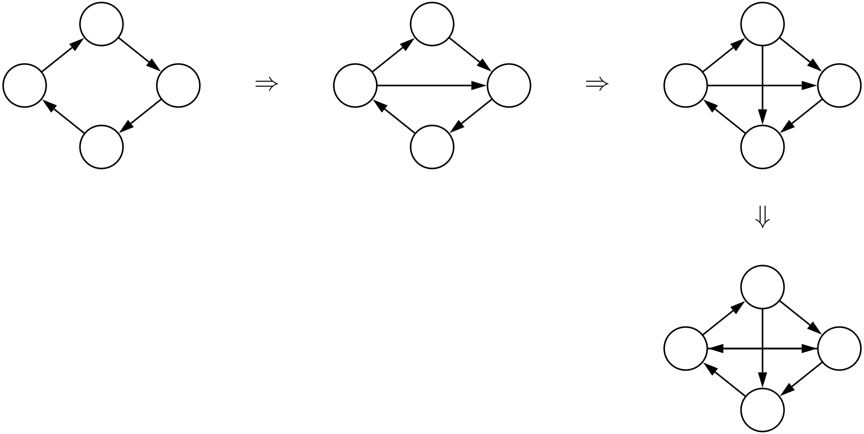
$\Rightarrow$



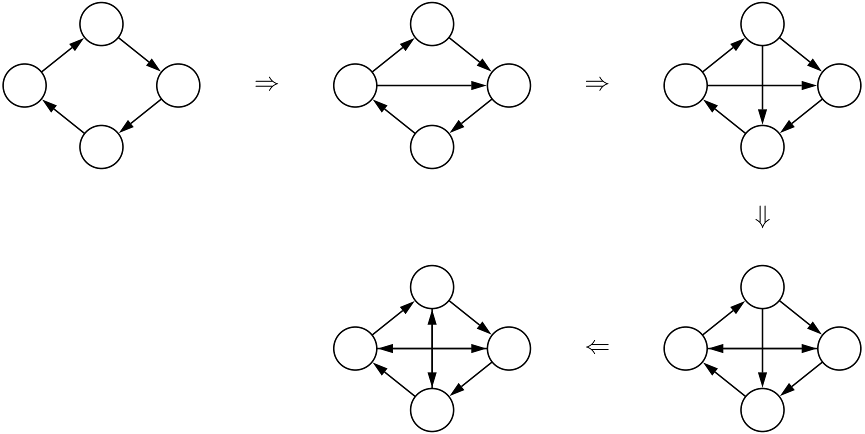
$\Rightarrow$



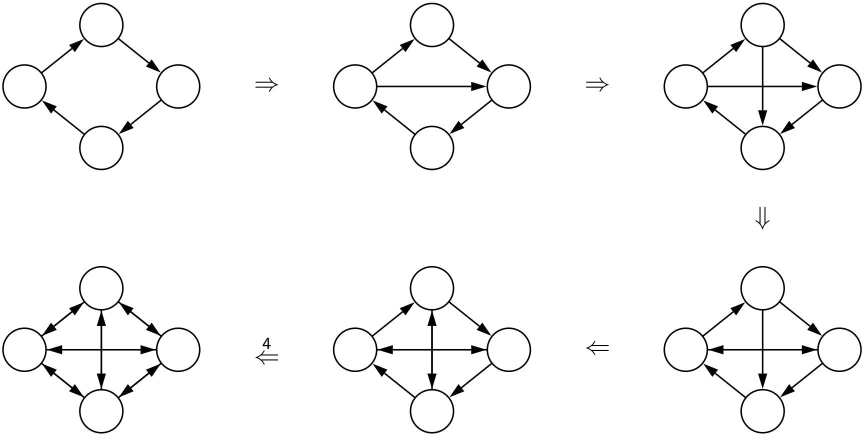
Example program: transitive closure (cont'd)



Example program: transitive closure (cont'd)



Example program: transitive closure (cont'd)



DPO graph transformation with relabelling

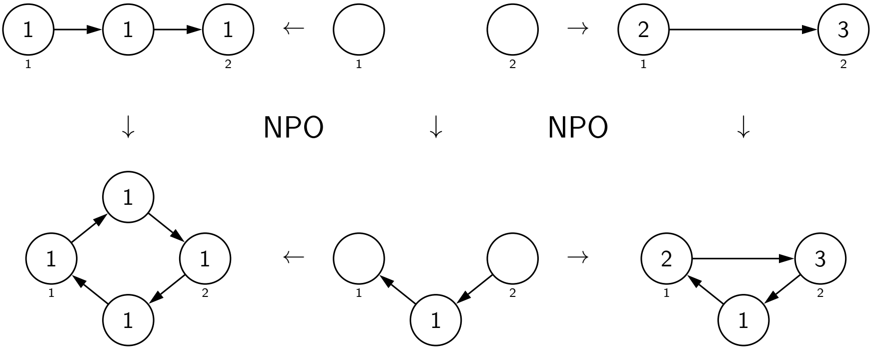
- A *rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ is a pair of graph inclusions; L, R are totally labelled and K is partially labelled
- Graph morphisms preserve graph structure and labels; unlabelled items can be mapped to arbitrary items
- Given an injective morphism $g\colon L \to G$, a *direct derivation* $G \Rightarrow_{r,g} H$ consists of two natural pushouts[1] of the form

$$
\begin{tikzcd}
L \arrow[d, "g"'] \arrow[dr, phantom, "\text{NPO}"] & K \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "\text{NPO}"] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}
$$

- NPOs exist if and only if g satisfies the *dangling condition*
- D and H are determined uniquely up to isomorphism

[1] a pushout is *natural* if it is also a pullback

Example: direct derivation



Construction of direct derivations

Given $r = \langle L \leftarrow K \rightarrow R \rangle$ and injective $g\colon L \rightarrow G$ satisfying the dangling condition:

<u>Construct D from G</u>

1. Remove all items in $g(L) - g(K)$
2. For each unlabelled item x in K, make $g(x)$ unlabelled

<u>Construct H from D</u>

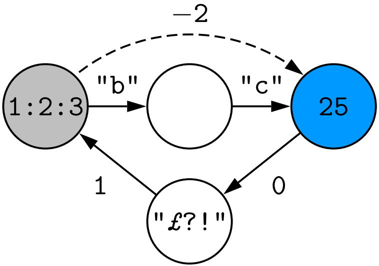
3. Add disjointly all items from $R - K$, retaining labels
4. For $e \in E_R - E_K$, $s_H(e)$ is $s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $g_V(s_R(e))$; analogously for targets
5. For each unlabelled item x in K, label $g(x)$ with $l_R(x)$

GP 2 host graph labels and type hierarchy

Label	::=	List [Mark]
List	::=	`empty` | Atom | List ':' List
Atom	::=	Integer | String
Integer	::=	['-'] Digit {Digit}
String	::=	' " '{Character}' " '
Mark	::=	`red` | `green` | `blue` | `grey` | `dashed`

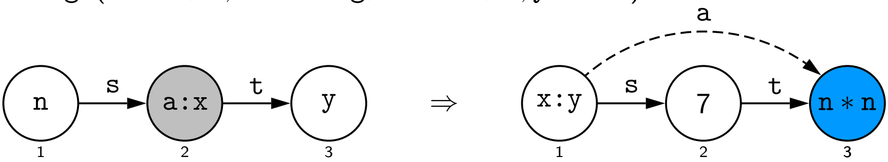
$$
\begin{tikzcd}[column sep=small]
 & \mathtt{list} \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] & \\
 & \mathtt{atom} \arrow[dl, phantom, "\rotatebox{45}{$\subseteq$}"] \arrow[dr, phantom, "\rotatebox{135}{$\subseteq$}"] & \\
\mathtt{int} & & \mathtt{string} \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] \\
 & & \mathtt{char}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}[column sep=small]
 & (\mathbb{Z} \cup \mathrm{Char}^*)^* \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] & \\
 & \mathbb{Z} \cup \mathrm{Char}^* \arrow[dl, phantom, "\rotatebox{45}{$\subseteq$}"] \arrow[dr, phantom, "\rotatebox{135}{$\subseteq$}"] & \\
\mathbb{Z} & & \mathrm{Char}^* \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] \\
 & & \mathrm{Char}
\end{tikzcd}
$$

Example: GP 2 host graph



Rule schemata for attributed graph transformation

$\texttt{bridge(n: int; s, t: string; a: atom; x, y: list)}$



$\texttt{where n} < 0 \texttt{ and not edge}(1, 3)$

- Variables in RHS and condition must occur in LHS
- LHS labels are *simple*:
  - no operators except ':' and unary '−'
  - at most one occurrence of a list variable
  - at most one occurrence of a string variable in each string expression

Rule-schema application

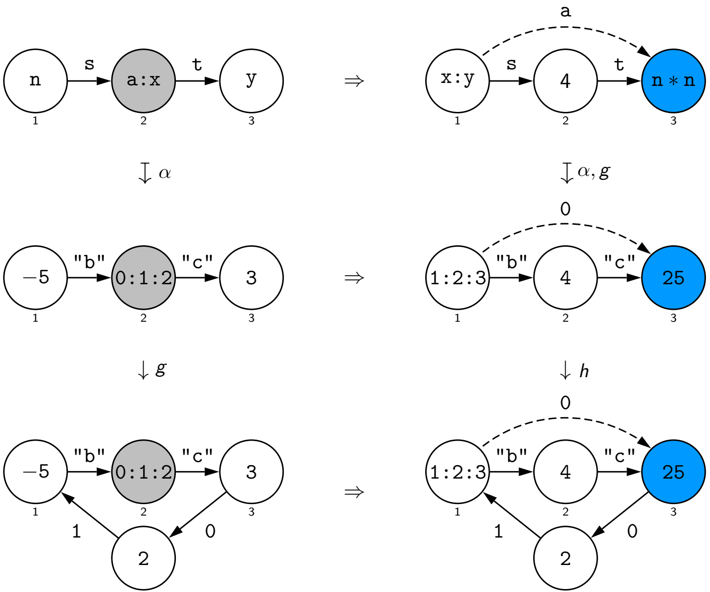
Applying $\langle L \Rightarrow R,\, c\rangle$ to a host graph G:

1. Find injective premorphism $g\colon L \to G$ (ignoring labels)
2. Check if g induces variable assignment α such that $g\colon L^{\alpha} \to G$ is label-preserving
3. Check whether $c^{\alpha,g} = \mathrm{true}$
4. Apply *rule instance* $L^{\alpha} \Rightarrow R^{\alpha,g}$ with match g

where L^{α}, $R^{\alpha,g}$ and $c^{\alpha,g}$ result from

- replacing variables x with $\alpha(x)$,
- replacing node identifiers v with $g(v)$, and
- evaluating the resulting expressions.

Example: rule-schema application



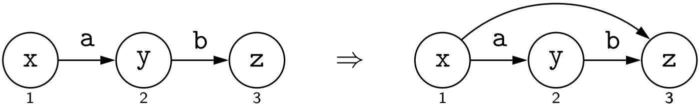
Abstract syntax of programs

$$
\begin{array}{lcl}
\text{Program} & ::= & \text{Decl } \{\text{Decl}\} \\
\text{Decl} & ::= & \text{RuleDecl} \mid \text{ProcDecl} \mid \text{MainDecl} \\
\text{ProcDecl} & ::= & \text{ProcId } `=\text{'} \ [\text{LocalDecl}] \text{ ComSeq} \\
\text{MainDecl} & ::= & \texttt{Main } `=\text{'} \text{ ComSeq} \\
\text{ComSeq} & ::= & \text{Com } \{`;\text{'} \text{ Com}\} \\
\text{Com} & ::= & \text{RuleSetCall} \mid \text{ProcCall} \\
 & & \mid \texttt{if} \text{ ComSeq } \texttt{then} \text{ ComSeq } [\texttt{else} \text{ ComSeq}] \\
 & & \mid \texttt{try} \text{ ComSeq } [\texttt{then} \text{ ComSeq } [\texttt{else} \text{ ComSeq}]] \\
 & & \mid \text{ComSeq } `!\text{'} \\
 & & \mid \text{ComSeq } \texttt{or} \text{ ComSeq} \\
 & & \mid `(\text{'} \text{ ComSeq } `)\text{'} \\
 & & \mid \texttt{break} \mid \texttt{skip} \mid \texttt{fail} \\
\text{RuleSetCall} & ::= & \text{RuleId} \mid `\{\text{'} \ [\text{RuleId } \{`,\text{'} \text{ RuleId}\}] \ `\}\text{'} \\
\text{ProcCall} & ::= & \text{ProcId}
\end{array}
$$

Case study: transitive closure

$\texttt{Main} = \texttt{link}!$

$\texttt{link}(\mathrm{a}, \mathrm{b}, \mathrm{x}, \mathrm{y}, \mathrm{z}\colon \texttt{list})$



$\texttt{where not}\,\mathrm{edge}(1, 3)$

Program: transitive closure (cont'd)

Proposition (Termination)

On every input graph G, the program terminates after at most $|V_G|^2$ rule schema applications.

Proof

Given any graph X, let

$$\#X = |\{\langle v, w\rangle \mid v, w \in \mathrm{V}_X \text{ and there is no edge } v \to w\}|.$$

Note that $\#X \leq |V_X|^2$. Moreover, for every step $G \Rightarrow_{\mathtt{link}} H$, $\#H = \#G - 1$. Hence `link!` terminates after at most $|V_G|^2$ rule schema applications. □

Case study: transitive closure (cont'd)

Proposition (Correctness)

The program returns a transitive closure of the input graph.

Proof

Let G be the input graph and T the resulting graph. For every step $X \Rightarrow_{\texttt{link}} Y$, there is an injective morphism $X \to Y$ because `link` does not delete or relabel any items. Hence T is an extension of G.

Transitivity of T is shown by induction on the length of directed paths. Consider a path $v_0, v_1, \ldots, v_n$ with $n > 1$ and $v_0 \neq v_n$. By induction hypothesis, there is an edge $v_0 \to v_{n-1}$. Thus there are edges $v_0 \to v_{n-1} \to v_n$. Then there must be an edge $v_0 \to v_n$ because `link` has been applied as long as possible.

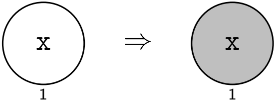
T is a smallest transitive extension of G because whenever `link` creates an edge $v \to v'$, by the declaration of `link` there is no such edge but a path $v \rightsquigarrow v'$. □

Case study: vertex colouring

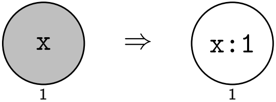
A *vertex colouring* is an assignment of colours to nodes such that adjacent nodes get different colours

Main = mark!; init!; inc!

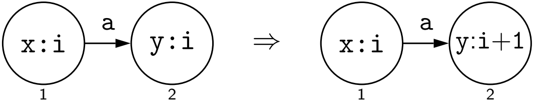
mark(x: list)



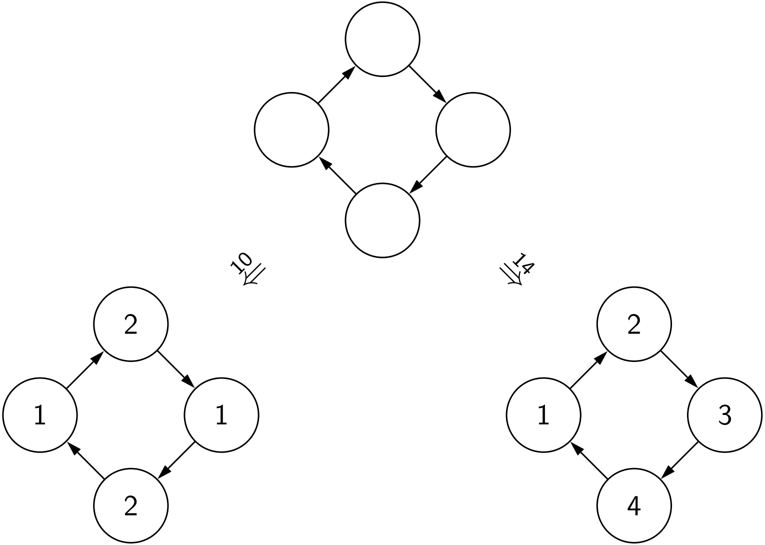
init(x: list)



inc(a, x, y: list; i: int)



Case study: vertex colouring (cont'd)



Partial correctness of vertex colouring

For a node v labelled $x{:}i$ with $i \in \mathbb{Z}$, let $\mathrm{colour}(v) = i$. A graph is *coloured* if any adjacent nodes v, v' satisfy $\mathrm{colour}(v) \neq \mathrm{colour}(v')$.

Proposition (Partial correctness)

If the program terminates with a graph M, then M is coloured.

Proof

Given a terminating program run

$$G \underset{\mathtt{mark}}{\overset{*}{\Rightarrow}} G' \underset{\mathtt{init}}{\overset{*}{\Rightarrow}} H \underset{\mathtt{inc}}{\overset{*}{\Rightarrow}} M,$$

H is obtained from G by replacing each node label x with $x{:}1$. If M were not correctly coloured, there were two adjacent nodes with the same colour. But then `inc` would be applicable to M, contradicting the fact that `inc` has been applied as long as possible. $\square$

Termination of vertex colouring

Lemma (Invariant)

If $G \Rightarrow^*_{\mathtt{inc}} H$ *with* $\mathrm{colour}(v) = 1$ *for all* $v \in V_G$*, then*

$$\{\mathrm{colour}(v) \mid v \in V_H\} = \{i \mid 1 \leq i \leq n\} \text{ for some } 1 \leq n \leq |V_H|.$$

Proposition (Termination)

Given a host graph G*, the program terminates after* $\mathrm{O}(|V_G|^2)$ *rule applications.*

Proof

Both `mark!` and `init!` terminate after $|V_G|$ steps. Suppose there was an infinite derivation

$$G \underset{\mathtt{mark}}{\overset{*}{\Rightarrow}} G' \underset{\mathtt{init}}{\overset{*}{\Rightarrow}} H_0 \Rightarrow_{\mathtt{inc}} H_1 \Rightarrow_{\mathtt{inc}} H_2 \Rightarrow_{\mathtt{inc}} \ldots$$

Termination of vertex colouring (cont'd)

Define $\#H_i = \sum_{v \in V_{H_i}} \mathrm{colour}(v)$. By the invariant,

$$\#H_i \leq \sum_{j=1}^{|V_{H_i}|} j \text{ and hence } \#H_i \leq \sum_{j=1}^{|V_G|} j.$$

But the labelling of `inc` implies

$$\#H_i < \#H_{i+1} \text{ for every } i \geq 0,$$

a contradiction. Thus the infinite derivation cannot exist.

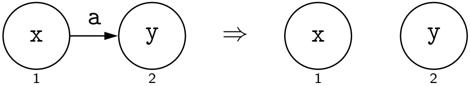
Also, any sequence of `inc` applications starting from G has at most a quadratic length because

$$\sum_{j=1}^{|V_G|} j = \frac{|V_G| \times (|V_G| + 1)}{2}.$$

Case study: recognising cyclic graphs

Main = if Cyclic then P else Q
Cyclic = delete!; {edge, loop}

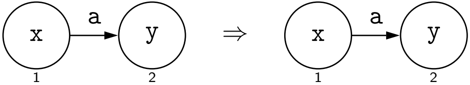
delete(a, x, y: list)



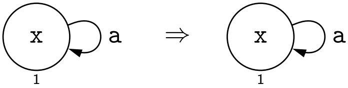
where indeg(1) = 0

/∗ *preserves cycles and cycle-freeness* ∗/

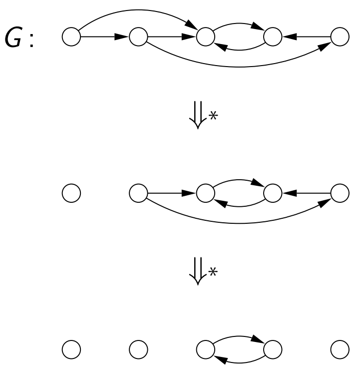
edge(a, x, y: list)



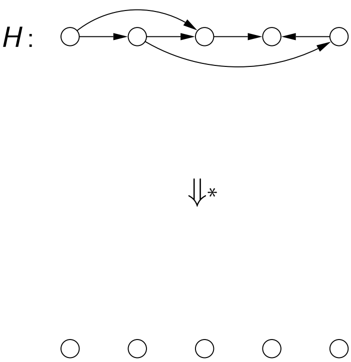
loop(a, x: list)



Case study: recognising cyclic graphs (cont'd)



- edge succeeds
  $\Rightarrow$ G is cyclic



- {`edge`, `loop`} fails
  $\Rightarrow$ H is acyclic

Time Complexity

- Bottleneck for efficient graph transformation: graph matching
- Matching a rule's LHS L in a host graph G requires time $|G|^{|L|}$
- Polynomial since program is fixed (only G is input)

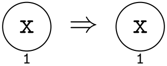
- *Consequence: linear-time graph algorithms usually require polynomial-time when recast in (unrooted) GP 2*

Example: Complexity of Tree Recognition

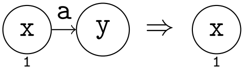
A graph is a *tree* if it contains a node from which there is a unique directed path to each node

```
Main = nonempty; prune!; if Invalid then fail
Invalid = {two_nodes, loop}
```

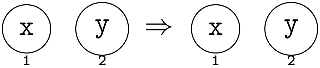
`nonempty(x:list)`



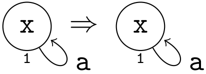
`prune(a,x,y:list)`



`two_nodes(x,y:list)`



`loop(a,x:list)`

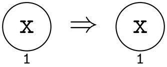


Example: Complexity of Tree Recognition

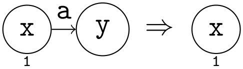
A graph is a *tree* if it contains a node from which there is a unique directed path to each node

```
Main = nonempty; prune!; if Invalid then fail
Invalid = {two_nodes, loop}
```

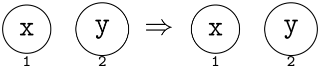
`nonempty(x:list)`



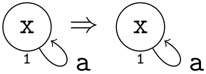
`prune(a,x,y:list)`



`two_nodes(x,y:list)`



`loop(a,x:list)`



Proposition (Correctness)

The program fails on a graph G if and only if G is not a tree.

Example: Complexity of Tree Recognition

Proposition (Cost of matching)

Finding a match for `prune` *requires time*

- $\mathcal{O}(|V_G||E_G|)$ *on arbitrary graphs, and*
- $\mathcal{O}(|V_G|)$ *on graphs of bounded node degree.*

Corollary (Complexity of tree recognition)

The program's time complexity is

- $\mathcal{O}(|V_G|^2|E_G|)$ *on arbitrary graphs, and*
- $\mathcal{O}(|V_G|^2)$ *on graphs of bounded node degree.*

Proof

Rule `prune` is applied at most $|V_G| - 1$ times. The cost of each application is dominated by the matching time. $\square$

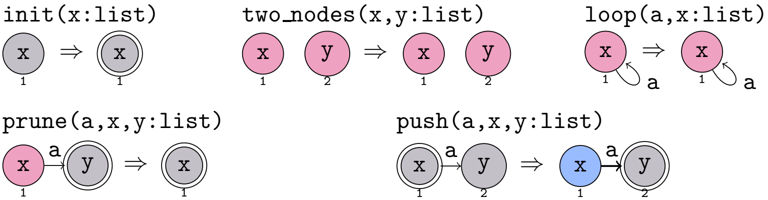
Rooted Graph Transformation

- Idea goes back to [Dörr 1995]: distinguish certain nodes as *roots* and match roots in rules with roots in host graphs
- Only the neighbourhood of host graph roots needs to be searched for matches
- Allows constant-time matching under mild conditions
- Adapted to DPO setting in [Dodds-P 2006] and to graph programs in [Bak-P 2012]

- *Price to pay: programs get more complicated and less declarative*

Example: Complexity of Rooted Tree Recognition

```
Main = init; {prune, push}!; if Invalid then fail
Invalid = {two_nodes, loop}
```

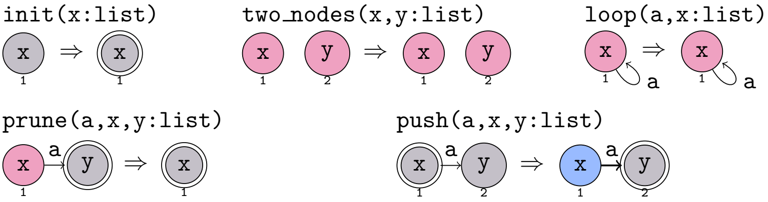


(pink is a wild card)

Assumption: input graphs have grey nodes

Example: Complexity of Rooted Tree Recognition

```
Main = init; {prune, push}!; if Invalid then fail
Invalid = {two_nodes, loop}
```



Proposition (Correctness and Complexity)

(1) *The program fails on a graph G if and only if G is not a tree.*

(2) *On graphs of bounded node degree, the program terminates in time $\mathcal{O}(|V_G|)$.*

Fast Rules

A conditional rule $\langle L \Rightarrow R,\ c\rangle$ is *fast* if

(1) each node in L is undirectedly reachable from some root,

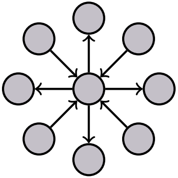
(2) neither L nor R contain repeated list, string or atom variables,

(3) the condition c contains neither an `edge` predicate nor a test $e_1 \!=\! e_2$ or $e_1 ! \!=\! e_2$ where both e_1 and e_2 contain a list, string or atom variable.

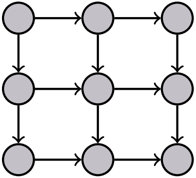
Theorem (Complexity of matching fast rules [Bak-P 2012])

Rooted graph matching can be implemented to run in constant time for fast rules, provided there are upper bounds on the maximal node degree and the number of roots in host graphs.

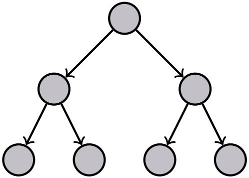
Graph Classes for Benchmarking



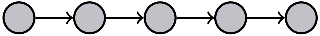
Star graphs



Square grids

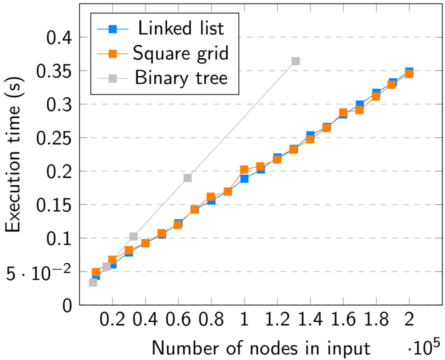


Balanced binary trees

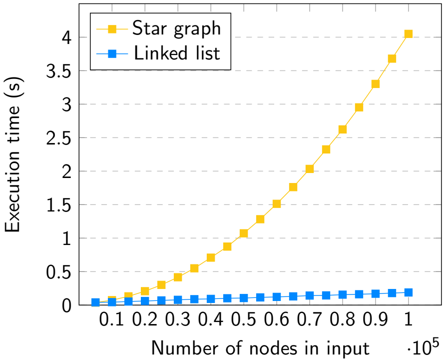


Linked lists

Measurements for Rooted Tree Recognition



Bounded-degree graphs



Star graphs and linked lists

More Linear-Time Algorithms

- Recognition of acyclic graphs and binary acyclic graphs
- Recognition of connected graphs
- Computing a 2-colouring
- Topological sorting of acyclic graphs

All programs use *depth-first search strategies* and expect bounded-degree graphs

Other Topics in Graph Programs

- Hoare-style program verification [Poskitt-P 10,12a,12b,14; Poskitt 13; P 16; Wulandari-P 18]
- Checking confluence by critical-pair analysis [Hristakiev-P 15,17,18; Hristakiev 17]
- Computational completeness [P 17]
- Structural operational semantics [Steinert-P 10, P 11]
- Compiling GP 2 to C [Bak 15; Bak-P 16]
- Probabilistic graph programs for randomised and evolutionary algorithms [Atkinson-P-Stepney 18a,18b,19]