Rule-based Graph Programming

Detlef Plump

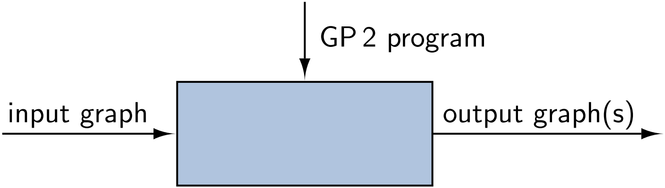
University of York, UK

Overview

Part I

Introduction

Graph Programming Language GP 2

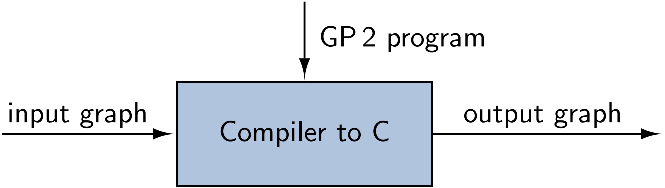

GP 2 program

input graph

output graph(s)

- Experimental domain-specific language for graphs
- Based on graph transformation rules
- Commands to control rule applications
- Non-deterministic
- Simple syntax and semantics to facilitate formal reasoning

Graph Programming Language GP 2



GP 2 program

input graph

Compiler to C

output graph

- Experimental domain-specific language for graphs
- Based on graph transformation rules
- Commands to control rule applications
- Non-deterministic
- Simple syntax and semantics to facilitate formal reasoning

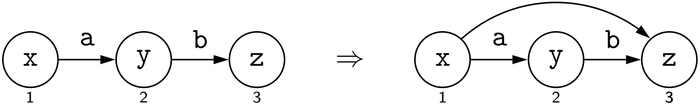
Example program: transitive closure

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):

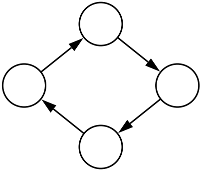
$\mathtt{Main} = \mathtt{link}!$

$\mathtt{link}(\mathrm{a}, \mathrm{b}, \mathrm{x}, \mathrm{y}, \mathrm{z} : \mathtt{list})$

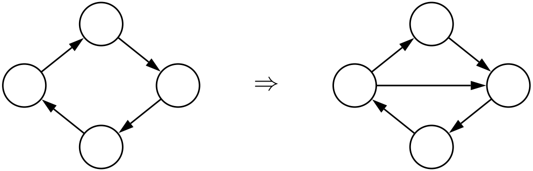


$\mathtt{where\ not\ edge}(1, 3)$

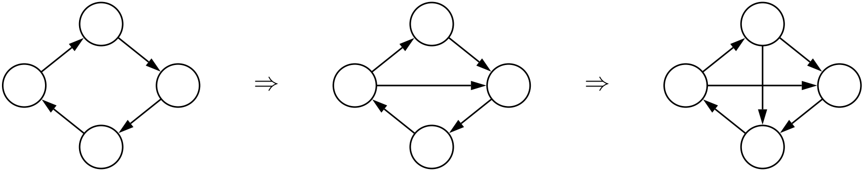
Example program: transitive closure (cont'd)



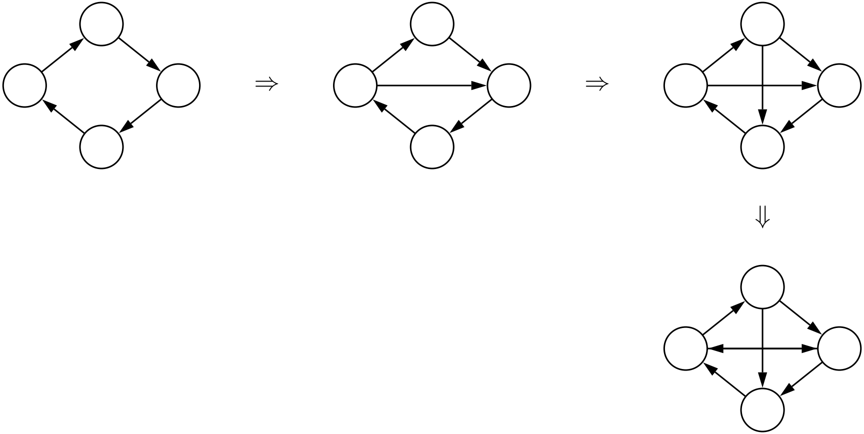
Example program: transitive closure (cont'd)



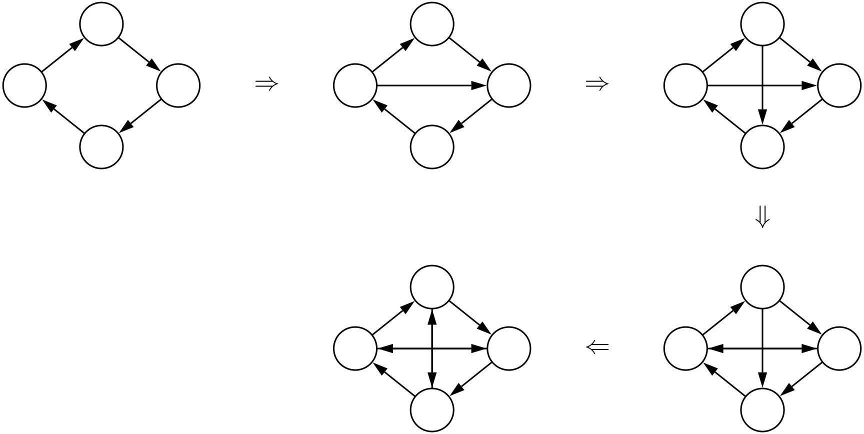
Example program: transitive closure (cont'd)



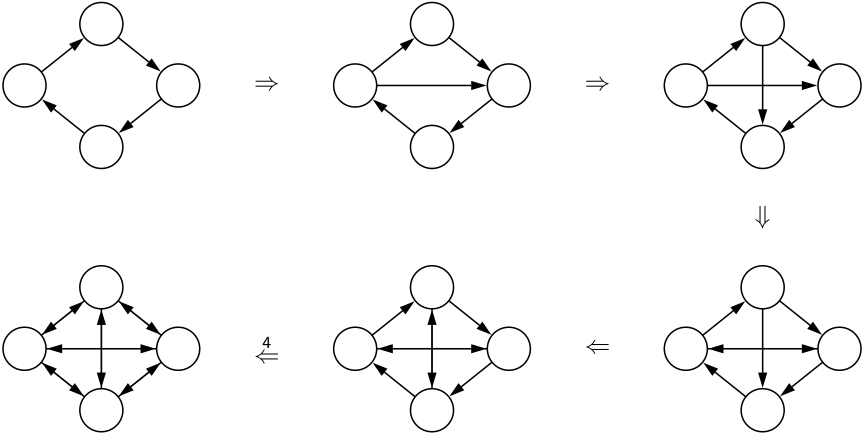
Example program: transitive closure (cont'd)



Example program: transitive closure (cont'd)



Example program: transitive closure (cont'd)



Part II

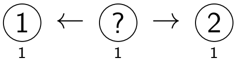
GP 2 Foundations

Graph transformation with relabelling

- How to write a program with double-pushout rules that replaces each node label 1 with 2 ?

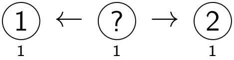
Graph transformation with relabelling

- How to write a program with double-pushout rules that replaces each node label 1 with 2 ?
- Graph morphisms are *label-preserving*, hence conventional rules cannot relabel nodes:



Graph transformation with relabelling

- How to write a program with double-pushout rules that replaces each node label 1 with 2 ?
- Graph morphisms are *label-preserving*, hence conventional rules cannot relabel nodes:

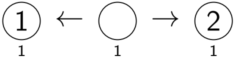


- Rule

works only for isolated nodes because of the dangling condition

Graph transformation with relabelling

Solution: *unlabelled* nodes in the interface:

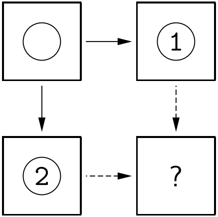


- *Partially labelled graphs* are defined as usual, except that the node labelling function is a partial function
- *Graph morphisms* between partially labelled graphs are defined as usual, except that unlabelled nodes can be mapped to arbitrary nodes
- *Pushouts* are defined as usual, except that their graphs are partially labelled

Graph transformation with relabelling

There is a price to pay:

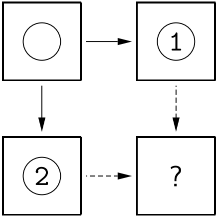
1. Pushouts need not exist:



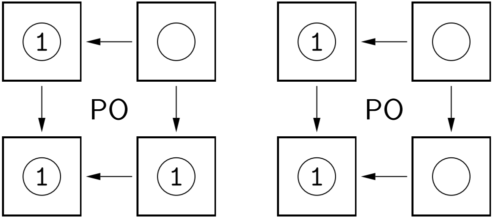
Graph transformation with relabelling

There is a price to pay:

1. Pushouts need not exist:



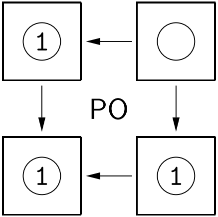
2. Pushout complements need not be unique:



Graph transformation with relabelling

Solution: *natural* pushouts

- A pushout is *natural* if it is also a pullback.
- A non-natural pushout:



- Note: For totally labelled graphs, pushouts with injective morphisms are always natural.

Graph transformation with relabelling

Lemma (Habel-P 2002)

A pushout of partially labelled graphs

$$\begin{tikzcd} A \arrow[r, "b"] \arrow[d, "c"'] & B \arrow[d] \\ C \arrow[r] & D \end{tikzcd}$$

is natural if and only if for each unlabelled item x in A, $b(x)$ or $c(x)$ is unlabelled.

Direct derivations with relabelling

- Consider rules $r = \langle L \leftarrow K \rightarrow R \rangle$ where K is partially labelled and L, R are totally labelled.
- Given an injective morphism $g \colon L \rightarrow G$, a *direct derivation* $G \Rightarrow_{r,g} H$ consists of two natural pushouts of the form

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

Proposition (Habel-P 2002)

Given r and g as above, there exists a direct derivation $G \Rightarrow_{r,g} H$ if and only if g satisfies the dangling condition. Moreover, in this case D and H are determined uniquely up to isomorphism.

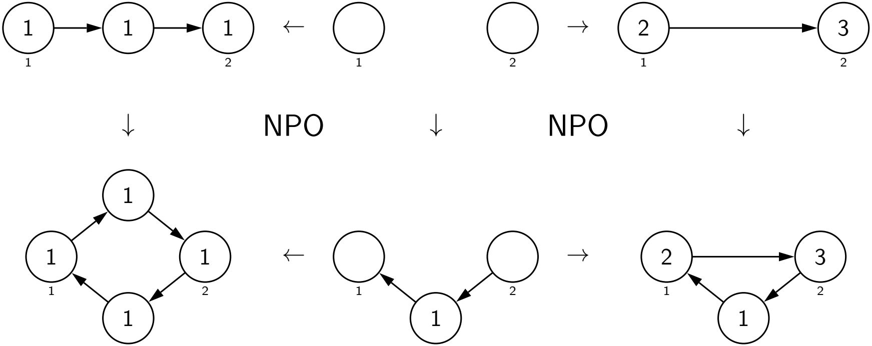
Direct derivations with relabelling

Proposition (Habel-P 2002)

Given a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ and an injective morphism $g\colon L \rightarrow G$ satisfying the dangling condition, graphs D and H as above can be constructed as follows:

1. *Constructing D from G:*
    - *Remove all items in $g(L) - g(K)$.*
    - *For each unlabelled node v in K, make $g_V(v)$ unlabelled.*
2. *Constructing H from D:*
    - *Add disjointly all items from $R - K$ while keeping their labels.*
    - *For $e \in E_R - E_K$, $s_H(e)$ is $s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $g_V(s_R(e))$. Targets are defined analogously.*
    - *For each unlabelled node v in K, label $g_V(v)$ with $l_R(v)$.*

Example: direct derivation with relabelling



NPO

NPO

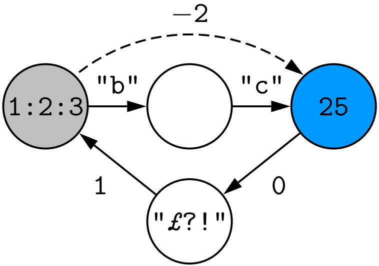
GP 2 host graph labels and type hierarchy

Label ::= List [Mark]
List ::= `empty` | Atom | List ‘:’ List
Atom ::= Integer | String
Integer ::= [‘-’] Digit {Digit}
String ::= ‘ “ ’{Character}‘ ” ’
Mark ::= `red` | `green` | `blue` | `grey` | `dashed`

$$\begin{tikzcd}
& \texttt{list} & \\
& \texttt{atom} \arrow[u, phantom, sloped, "\subseteq"] & \\
\texttt{int} \arrow[ur, phantom, sloped, "\subseteq"] & & \texttt{string} \arrow[ul, phantom, sloped, "\subseteq"] \\
& & \texttt{char} \arrow[u, phantom, sloped, "\subseteq"]
\end{tikzcd}$$

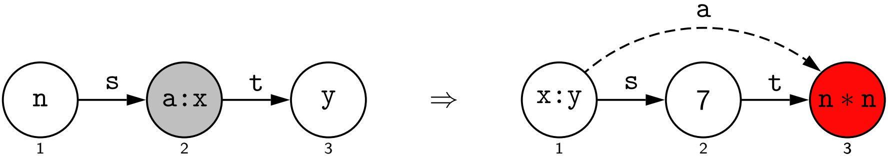
$$\begin{tikzcd}
& (\mathbb{Z} \cup \mathrm{Char}^*)^* & \\
& \mathbb{Z} \cup \mathrm{Char}^* \arrow[u, phantom, sloped, "\subseteq"] & \\
\mathbb{Z} \arrow[ur, phantom, sloped, "\subseteq"] & & \mathrm{Char}^* \arrow[ul, phantom, sloped, "\subseteq"] \\
& & \mathrm{Char} \arrow[u, phantom, sloped, "\subseteq"]
\end{tikzcd}$$

Example: GP 2 host graph



Rule schemata for attributed graph transformation

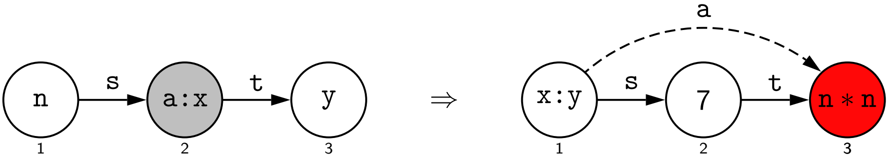
`bridge(n: int; s, t: string; a: atom; x, y: list)`



`where n < 0 and not edge(1, 3)`

Rule schemata for attributed graph transformation

bridge(n: int; s, t: string; a: atom; x, y: list)



where $n < 0$ and not edge(1, 3)

- ':' is list concatenation
- LHS expressions are *simple* (e.g. no operators except ':' and '−')
- Variables in RHS and condition must occur in LHS

Rule-schema application (sketched)

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

Rule-schema application (sketched)

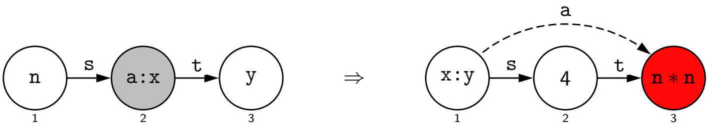
Applying $\langle L \Rightarrow R,\, c\rangle$ to a host graph G:

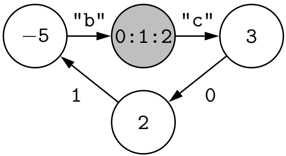
1. Find injective premorphism $g\colon L \to G$ (ignoring labels)
2. Check if g induces variable assignment α such that $g\colon L^\alpha \to G$ is label-preserving
3. Check whether $c^{\alpha,g} = \mathrm{true}$
4. Apply rule instance $L^\alpha \Rightarrow R^{\alpha,g}$ with match g

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

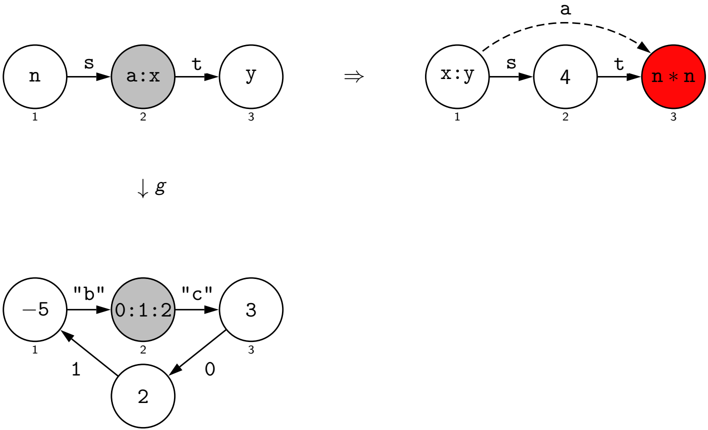
- replacing variables x with $\alpha(x)$,
- replacing node identifiers v with $g(v)$, and
- evaluating the resulting expressions.

Example: rule-schema application

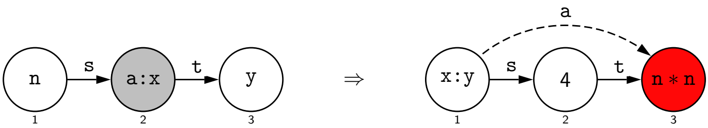


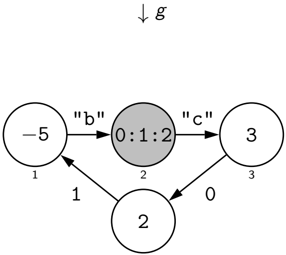


Example: rule-schema application



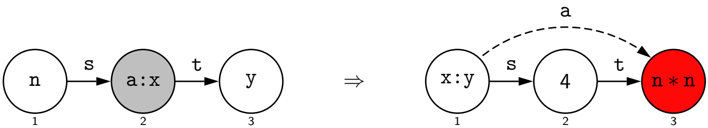
Example: rule-schema application

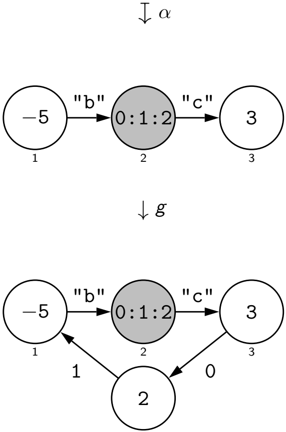




$$\alpha \left\{ \begin{array}{lcr} \mathtt{n} & \mapsto & -5 \\ \mathtt{a} & \mapsto & 0 \\ \mathtt{x} & \mapsto & 1{:}2 \\ \mathtt{y} & \mapsto & 3 \\ \mathtt{s} & \mapsto & \mathtt{"b"} \\ \mathtt{t} & \mapsto & \mathtt{"c"} \end{array} \right.$$

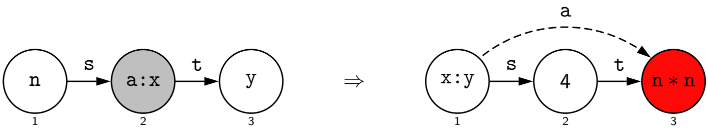
Example: rule-schema application





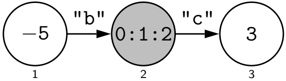
$$\alpha \left\{ \begin{array}{lcc} \mathtt{n} & \mapsto & -5 \\ \mathtt{a} & \mapsto & 0 \\ \mathtt{x} & \mapsto & 1{:}2 \\ \mathtt{y} & \mapsto & 3 \\ \mathtt{s} & \mapsto & \mathtt{"b"} \\ \mathtt{t} & \mapsto & \mathtt{"c"} \end{array} \right.$$

Example: rule-schema application

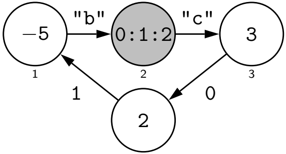


$\downarrow\!\!\!\!\overline{}\ \alpha$

$$\alpha \left\{ \begin{array}{lcl} \mathtt{n} & \mapsto & -5 \\ \mathtt{a} & \mapsto & 0 \\ \mathtt{x} & \mapsto & 1{:}2 \\ \mathtt{y} & \mapsto & 3 \\ \mathtt{s} & \mapsto & \texttt{"b"} \\ \mathtt{t} & \mapsto & \texttt{"c"} \end{array} \right.$$

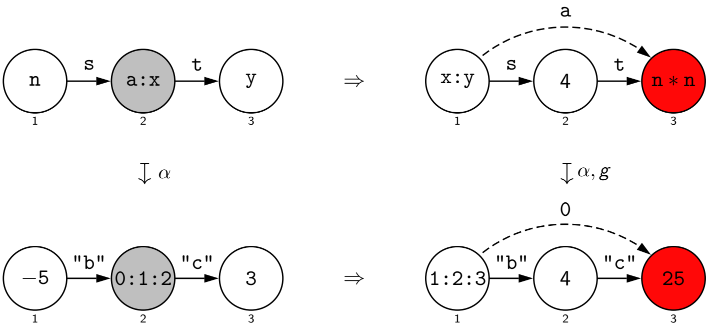


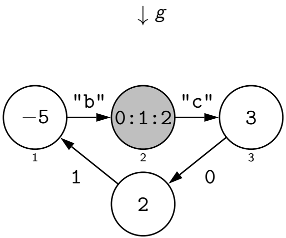
$\downarrow g$



$$\begin{array}{rl} & (\mathtt{n} < 0 \text{ and not edge}(1,\,3))^{\alpha,g} \\ \equiv & -5 < 0 \text{ and not edge}(1,\,3) \\ \equiv & \text{true} \end{array}$$

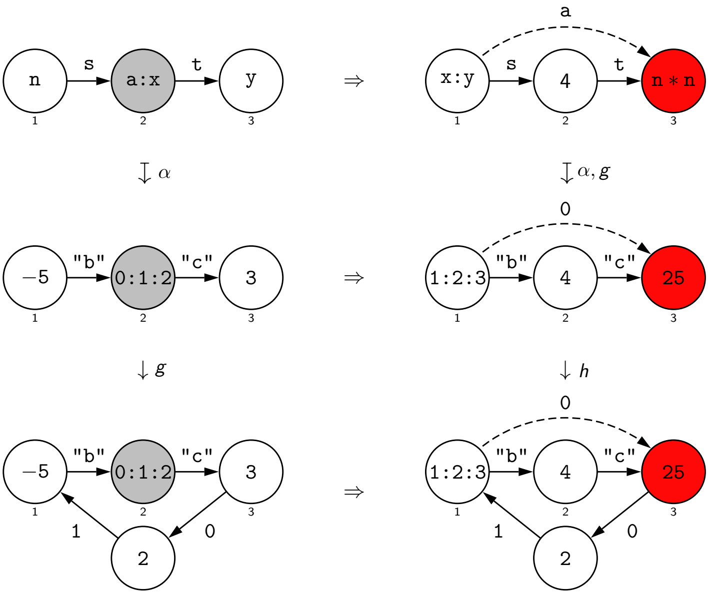
Example: rule-schema application





$$\alpha \begin{cases} \mathtt{n} \mapsto -5 \\ \mathtt{a} \mapsto 0 \\ \mathtt{x} \mapsto 1{:}2 \\ \mathtt{y} \mapsto 3 \\ \mathtt{s} \mapsto \texttt{"b"} \\ \mathtt{t} \mapsto \texttt{"c"} \end{cases}$$

Example: rule-schema application



Part III

Graph Programs

Syntax of rule-schema labels

Label ::= ListExp [Mark]

ListExp ::= `empty` | ListVar | AtomExp
| ListExp ‘:’ ListExp

AtomExp ::= AtomVar | IntExp | StringExp

IntExp ::= IntVar | Number
| (`indeg` | `outdeg`) ‘(’ NodeId ‘)’
| `length` ‘(’ (AtomVar | StringVar | ListVar) ‘)’
| ‘-’ IntExp
| ‘(’ IntExp ‘)’
| IntExp (‘+’ | ‘−’ | ‘∗’ | ‘/’) IntExp

StringExp ::= StringVar | CharVar | ‘ " ’ {Character} ‘ " ’
| StringExp ‘`.`’ StringExp

Mark ::= `red` | `green` | `blue` | `grey` | `dashed` | `any`

Syntax of rule-schema conditions

```
Condition     ::=  edge '(' NodeID ',' NodeID [ ',' Label ] ')'
                   | SubtypeExp
                   | ListExp ( '=' | '!=' ) ListExp
                   | IntExp ('>' | '>=' | '<' | '<=') IntExp
                   | not Condition
                   | Condition (and | or) Condition
                   | '(' Condition ')'
SubtypeExp    ::=  (atom | int | string | char) '(' Var ')'
```

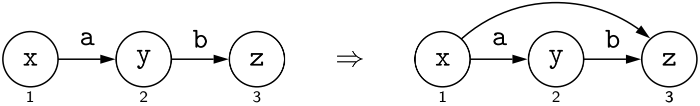
Syntax of commands

```
Program      ::=  Decl {Decl}
Decl         ::=  RuleDecl | ProcDecl | MainDecl
ProcDecl     ::=  ProcId ‘=’ [LocalDecl] ComSeq
MainDecl     ::=  Main ‘=’ ComSeq
ComSeq       ::=  Com {‘;’ Com}
Com          ::=  RuleSetCall | ProcCall
                  | if ComSeq then ComSeq [else ComSeq]
                  | try ComSeq [then ComSeq [else ComSeq]]
                  | ComSeq ‘!’
                  | ComSeq or ComSeq
                  | ‘(’ ComSeq ‘)’
                  | break | skip | fail
RuleSetCall  ::=  RuleId | ‘{’ [RuleId {‘,’ RuleId}] ‘}’
ProcCall     ::=  ProcId
```

Example program: transitive closure

`Main = link!`

`link(a, b, x, y, z: list)`



`where not edge(1, 3)`

Example 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. ☐

Example program: transitive closure (cont'd)

Proposition (Correctness)

The program returns a transitive closure of the input graph.

Proof

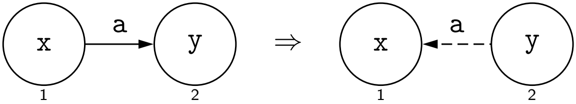
Let G be the input graph and T the resulting graph. For every step $X \Rightarrow_{\texttt{link}} Y$, there is an injective graph morphism $X \to Y$ because `link` does not delete or relabel any items. It follows that T is an extension of G (up to isomorphism).
We show that T is transitive by induction on the length of paths in T. Consider a directed path $v_0, v_1, \ldots, v_n$ with $v_0 \neq v_n$. We can assume w.l.o.g. that $v_0, \ldots, v_n$ are distinct. If $n = 1$, there is an edge $v_0 \to v_n$. If $n > 1$, there is an edge $v_0 \to v_{n-1}$ by induction hypothesis. Thus there are edges $v_0 \to v_{n-1} \to v_n$. As `link` has been applied as long as possible, there must be an edge $v_0 \to v_n$.
Finally, 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 $v \rightsquigarrow v'$. □

Example program: graph inverse

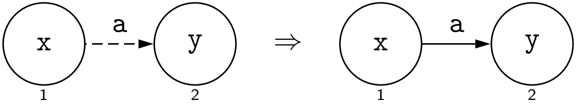
The *inverse* of a graph is obtained by reversing all edges.

```
Main = reverse!; unmark!
```

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



```
unmark(a, x, y : int)
```

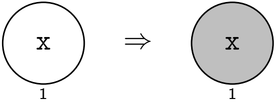


Example program: vertex colouring

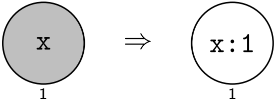
A *vertex colouring* is an assignment of colours to nodes such that each non-loop edge has end points with distinct colours.

`Main = mark!; init!; inc!`

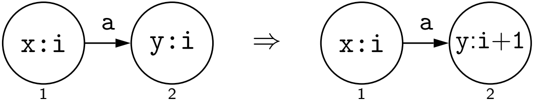
`mark(x: list)`



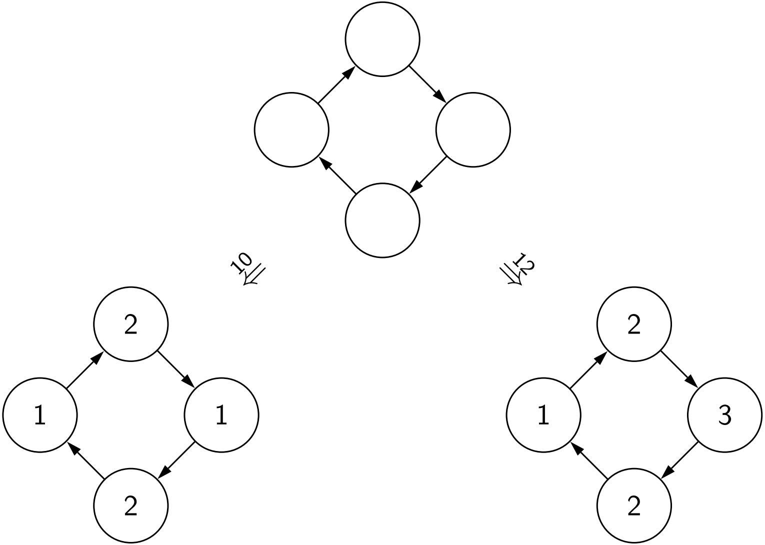
`init(x: list)`



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



Example program: vertex colouring (cont'd)



Partial correctness of vertex colouring

A graph is *correctly coloured* if the end points of each non-loop edge have labels of the form $x{:}i$ and $y{:}j$, for integers i, j with $i \neq j$

Proposition (Partial correctness)

If the program terminates on an input graph G, then it returns G correctly coloured.

Proof

Consider a terminating program run

$$G \overset{*}{\underset{\texttt{mark}}{\Rightarrow}} G' \overset{*}{\underset{\texttt{init}}{\Rightarrow}} H \overset{*}{\underset{\texttt{inc}}{\Rightarrow}} M.$$

Then H is obtained from G by replacing each node label x with $x{:}1$. Suppose that M is not correctly coloured. Then it contains a non-loop edge whose end points have the same colour. Hence `inc` is applicable to M, contradicting the fact that M results from applying `inc` as long as possible. ☐

Termination of vertex colouring

For a coloured graph G, let $\mathrm{Colours}(G) = \{\mathrm{colour}(v) \mid v \in V_G\}$ where for any node v with label $x{:}i$, $\mathrm{colour}(v) = i$

Lemma (Invariant)

Given any derivation $G \Rightarrow^*_{\mathtt{inc}} H$ *with* $\mathrm{Colours}(G) = \{1\}$,

$$\mathrm{Colours}(H) = \{i \mid 1 \leq i \leq n\} \textit{ for some } 1 \leq n \leq |V_H|.$$

Proof

Every step $X \Rightarrow_{\mathtt{inc}} Y$ satisfies $\mathrm{Colours}(Y) = \mathrm{Colours}(X) \cup \Delta$, where $\Delta = \emptyset$ or $\Delta = \{\max(\mathrm{Colours}(X)) + 1\}$. The invariant then follows by induction on the length of $G \Rightarrow^*_{\mathtt{inc}} H$. □

Termination of vertex colouring (cont'd)

Proposition (Termination)

On every input graph G, the program terminates after $\mathrm{O}(|V_G|^2)$ rule applications.

Proof

Both `mark!` and `init!` terminate after $|V_G|$ steps.

Let G be a coloured host graph with $\mathrm{Colours}(G) = \{1\}$ and suppose there was an infinite derivation

$$G = G_0 \Rightarrow_{\mathtt{inc}} G_1 \Rightarrow_{\mathtt{inc}} G_2 \Rightarrow_{\mathtt{inc}} \cdots$$

For $i \geq 0$, let $\#G_i = \sum_{v \in V_{G_i}} \mathrm{colour}(v)$. Then

$$\#G_i < \#G_{i+1} \text{ for every } i \geq 0$$

by the labelling of `inc`. (continued)

Termination of vertex colouring (cont'd)

But the invariant shows that for all $i \geq 0$,

$$\#G_i \leq \sum_{j=1}^{|V_{G_i}|} j = \sum_{j=1}^{|V_G|} j$$

where $|V_{G_i}| = |V_G|$ because `inc` preserves the number of nodes. Thus the infinite sequence $G = G_0 \Rightarrow_{\texttt{inc}} G_1 \Rightarrow_{\texttt{inc}} \ldots$ cannot exist.

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

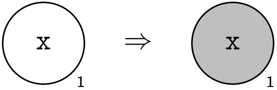
$$\sum_{j=1}^{|V_G|} j = \frac{|V_G| \times (|V_G| + 1)}{2}.$$

□

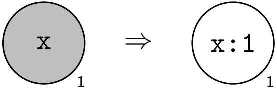
Example program: 2-colouring

Main = try (mark!; init; colour!; if illegal then fail)

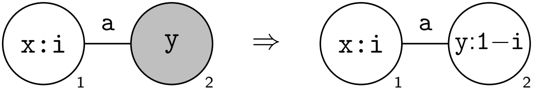
mark(x: list)



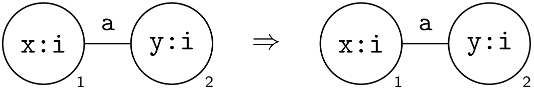
init(x: list)



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



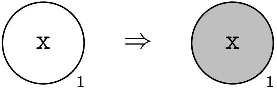
illegal(a, x, y: list; i: int)



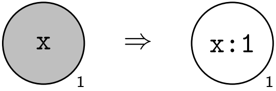
Example program: 2-colouring

`Main = try (mark!; init; colour!; if illegal then fail)`

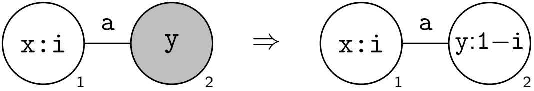
`mark(x: list)`



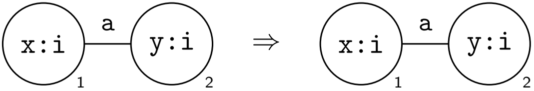
`init(x: list)`



`colour(a, x, y: list; i: int)`

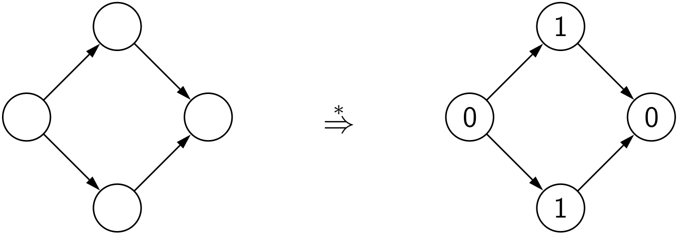


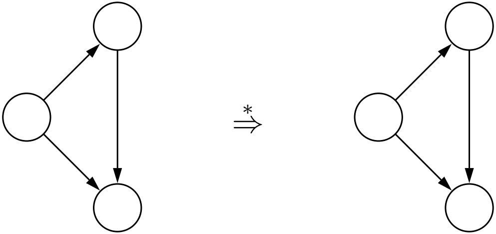
`illegal(a, x, y: list; i: int)`



Assumption: input graph is connected

Example program: 2-colouring (cont'd)





Example program: 2-colouring (cont'd)

Lemma (2-colourability)

A graph is 2-colourable if and only if it does not contain an undirected cycle of odd length ≥ 3.

Proof

See any textbook on graph theory.

Proposition (Termination)

On every input graph G*, the program terminates after* $\mathrm{O}(|V_G|)$ *rule schema applications.*

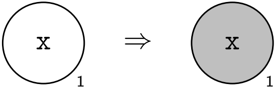
Proposition (Correctness)

Given a connected input graph G*, the program either returns* G *2-coloured or, if* G *is not 2-colourable, returns* G *unmodified.*

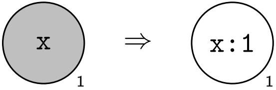
Example: 2-colouring for disconnected graphs (Version 1)

Main = try (mark!; (init; colour!)!; if illegal then fail)

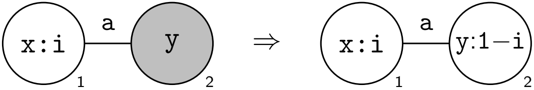
mark(x: list)



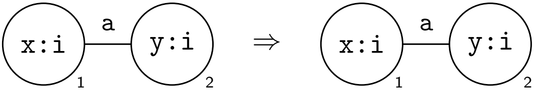
init(x: list)



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



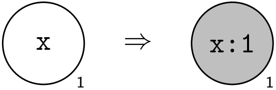
illegal(a, x, y: list; i: int)



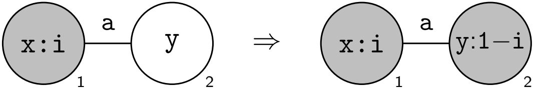
Example: 2-colouring for disconnected graphs (Version 2)

```
Main = (init; colour!)!; if illegal then undo! else unmark!
```

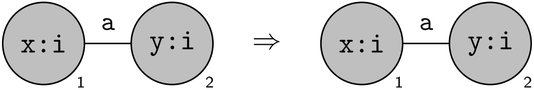
`init(x: list)`



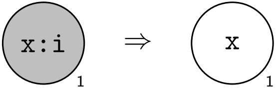
`colour(a, x, y: list; i: int)`



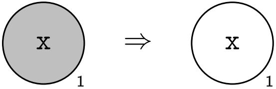
`illegal(a, x, y: list; i: int)`



`undo(x: list; i: int)`



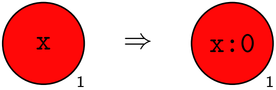
`unmark(x: list)`



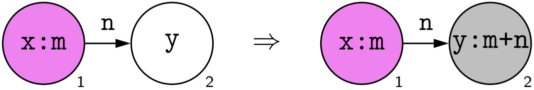
Example program: shortest distances

Main = init; {add, reduce}!

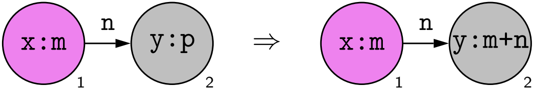
init(x: list)



add(x, y: list; m, n: int)



reduce(x, y: list; m, n, p: int)

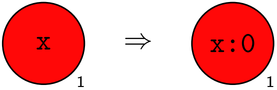


where $m + n < p$

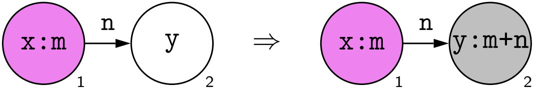
Example program: shortest distances

```
Main = init; {add, reduce}!
```

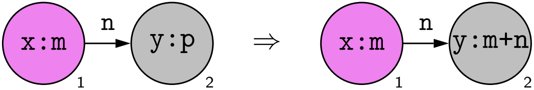
```
init(x: list)
```



```
add(x, y: list; m, n: int)
```



```
reduce(x, y: list; m, n, p: int)
```



```
where m + n < p
```

Assumption: input graph has a single red node; edge labels are non-negative integers

Example program: shortest distances (cont'd)

Proposition (Termination)

The program terminates on every input graph whose edge labels are non-negative integers.

Proof

Suppose that the program does not terminate on some input graph. Then there is an infinite sequence $G_0 \Rightarrow_{r_0} G_1 \Rightarrow_{r_1} G_2 \Rightarrow \dots$ with $r_i \in \{\texttt{add}, \texttt{reduce}\}$ for all $i \geq 0$. Because `add` decreases the number of unmarked nodes, and `reduce` preserves this number, there must be some $k \geq 0$ such that $r_i = \texttt{reduce}$ for all $i \geq k$. Given a graph G_i in the sequence, let $\#G_i$ be the sum of all distances in G_i's marked nodes. Then $\#G_i > \#G_{i+1}$ for each step $G_i \Rightarrow_{\texttt{reduce}} G_{i+1}$. Since all edge labels in G_0 are non-negative, all distances in the marked graphs of the sequence are non-negative (an invariant easily proved by induction). But this contradicts the infinity of $G_k \Rightarrow G_{k+1} \Rightarrow G_{k+2} \Rightarrow \dots$ □

Example program: shortest distances (cont'd)

Lemma (Invariant)

If $G \Rightarrow_{\texttt{init}} G' \Rightarrow^{*}_{\{\texttt{add},\,\texttt{reduce}\}} H$*, then each marked node* v *in* H *has a label* $x{:}n$ *where* n *is a distance from the red node to* v*.*

Proof

By induction on the length of $G' \Rightarrow^{*}_{\{\texttt{add},\,\texttt{reduce}\}} H$. $\square$

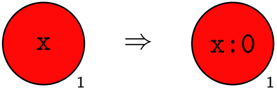
Proposition (Correctness)

Consider an input graph whose edge labels are non-negative integers and suppose that there is a single red node v*. Then the program marks each node* w *that is reachable from* v *and adds to* w*'s label the shortest distance from* v *to* w*.*

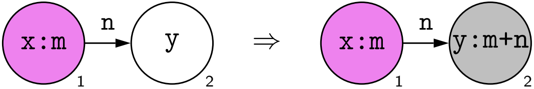
Equivalent program for shortest distances

Main = init; add!; reduce!

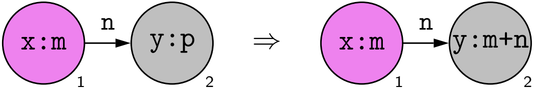
init(x: list)



add(x, y: list; m, n: int)



reduce(x, y: list; m, n, p: int)



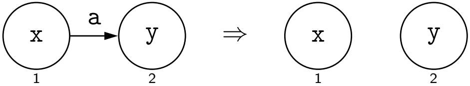
where $m + n < p$

Note: {add, reduce} can be shown to be confluent by a (non-trivial) critical-pair analysis [Hristakiev 17]

Example program: recognising cyclic graphs

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

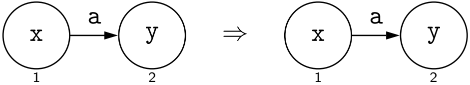
delete(a, x, y: list)



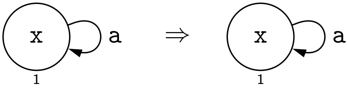
where indeg(1) = 0

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

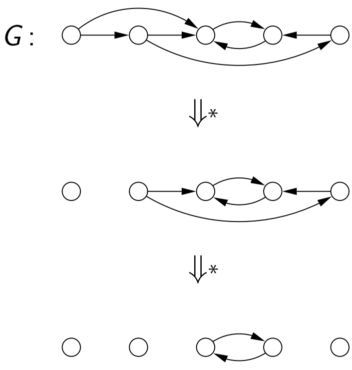
edge(a, x, y: list)



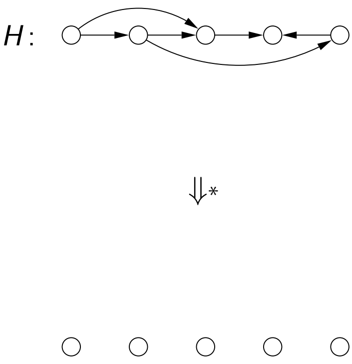
loop(a, x: list)



Example program: recognising cyclic graphs (cont'd)



- `edge` succeeds,
  P is executed on G

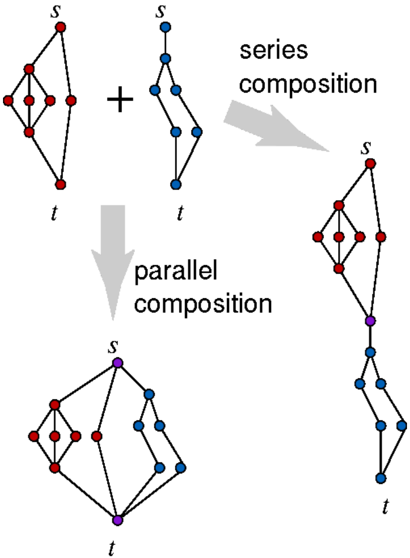


- {`edge`, `loop`} fails,
  Q is executed on H

Example program: recognising series-parallel graphs

Series-parallel graphs are inductively defined:

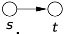
1. $\underset{s}{\circ}\!\longrightarrow\!\underset{t}{\circ}$ is series-parallel, where s is the *source* and t is the *sink*.
2. The class of series-parallel graphs is closed under *parallel composition* and *sequential composition*.

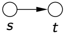


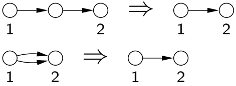
(Wikipedia)

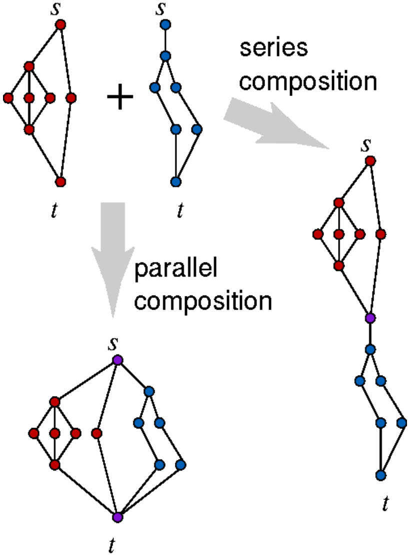
Example program: recognising series-parallel graphs

Series-parallel graphs are inductively defined:

1.  is series-parallel, where *s* is the *source* and *t* is the *sink*.
2. The class of series-parallel graphs is closed under *parallel composition* and *sequential composition*.

Equivalently, a graph is series-parallel if it can be reduced to  by the following rules:



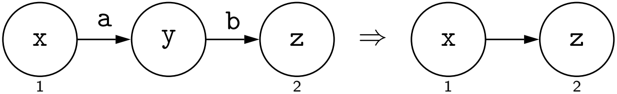


(Wikipedia)

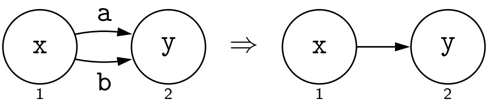
Example program: recognising series-parallel graphs

```
Series-parallel = Reduce!; delete; if nonempty then fail
Reduce = {series, parallel}
```

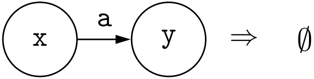
series(a, b, x, y, z : list)



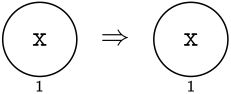
parallel(a, b, x, y : list)



delete(a, x, y : list)



nonempty(x : list)



Termination of `Series-parallel`

Proposition (Termination)

On every input graph G, `Series-parallel` *terminates after at most* $|E_G| + 1$ *rule schema applications.*

Proof

The maximum number of rule schema applications is given if `Reduce!` deletes all but one edges, `delete` removes the last edge, and `nonempty` is applicable to some remaining node. □

Correctness of `Series-parallel`

Proposition (Correctness)

Given any input graph G, `Series-parallel` *returns the empty graph if G is series-parallel, and fails otherwise.*

Proof

If G is series-parallel, every execution of `Reduce!` on G results in the graph `TwoNodes` consisting of two nodes and one non-loop edge. This is because `Reduce` is confluent on series-parallel graphs:

$$\texttt{TwoNodes} \overset{*}{\underset{\texttt{Reduce}}{\Leftarrow}} G \underset{\texttt{Reduce}}{\Rightarrow} H$$

implies $H \Rightarrow^{*}_{\texttt{Reduce}}$ `TwoNodes`. This can be shown by critical-pair analysis [Hristakiev 17].

The application of `delete` to `TwoNodes` will then return $\emptyset$ and the `if`-command has no effect.

If G is not series-parallel, either `delete` fails or it is applicable but produces a non-empty graph on which the `if`-command fails. □

Part IV

Operational Semantics

Structural operational semantics (SOS)

- *Configurations* are given by $(\mathsf{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\mathsf{fail}\}$, where $\mathcal{G}$ consists of all host graphs
- *Results* are configurations in $\mathcal{G} \cup \{\mathsf{fail}\}$
- *Transition relation*

$$\rightarrow \;\subseteq\; (\mathsf{ComSeq} \times \mathcal{G}) \times ((\mathsf{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\mathsf{fail}\})$$

is defined by structural induction on command sequences

SOS: rule set-call $\mathcal{R}$ (set of rule schemata)

$$[\mathrm{call}_1] \qquad \frac{G \Rightarrow_{\mathcal{R}} H}{\langle \mathcal{R},\ G \rangle \to H}$$

$$[\mathrm{call}_2] \qquad \frac{G \not\Rightarrow_{\mathcal{R}}}{\langle \mathcal{R},\ G \rangle \to \mathsf{fail}}$$

SOS: sequential composition

$$[\mathrm{seq}_1] \qquad \frac{\langle P,\ G\rangle \rightarrow \langle P',\ H\rangle}{\langle P; Q,\ G\rangle \rightarrow \langle P'; Q,\ H\rangle}$$

$$[\mathrm{seq}_2] \qquad \frac{\langle P,\ G\rangle \rightarrow H}{\langle P; Q,\ G\rangle \rightarrow \langle Q,\ H\rangle}$$

$$[\mathrm{seq}_3] \qquad \frac{\langle P,\ G\rangle \rightarrow \mathsf{fail}}{\langle P; Q,\ G\rangle \rightarrow \mathsf{fail}}$$

Example: transition relation

$$\boxed{\begin{array}{l} \texttt{Main} = \{\texttt{r1}, \texttt{r2}\}; \{\texttt{r1}, \texttt{r2}\}; \texttt{r1!} \\[2ex] \texttt{r1}: \textcircled{1} \Rightarrow \textcircled{1} \qquad \texttt{r2}: \textcircled{1} \Rightarrow \textcircled{2} \end{array}}$$

$$\begin{tikzcd}
\langle \texttt{Main}, \textcircled{1} \rangle \arrow[r] \arrow[d] & \langle P, \textcircled{2} \rangle \arrow[r] & \text{fail} & \\
\langle P, \textcircled{1} \rangle \arrow[r] \arrow[d] & \langle \texttt{r1!}, \textcircled{1} \rangle \arrow[r] & \langle \texttt{r1!}, \textcircled{1} \rangle \arrow[r] & \ldots \\
\langle \texttt{r1!}, \textcircled{2} \rangle \arrow[d] & & & \\
\textcircled{2} & & &
\end{tikzcd}$$

where $P = \{\texttt{r1}, \texttt{r2}\}; \texttt{r1!}$

SOS: if-then-else and try-then-else

$$[\mathrm{if}_1] \qquad \frac{\langle C,\ G\rangle \to^{+} H}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\ G\rangle \to \langle P,\ G\rangle}$$

$$[\mathrm{if}_2] \qquad \frac{\langle C,\ G\rangle \to^{+} \mathrm{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\ G\rangle \to \langle Q,\ G\rangle}$$

$$[\mathrm{try}_1] \qquad \frac{\langle C,\ G\rangle \to^{+} H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\ G\rangle \to \langle P,\ H\rangle}$$

$$[\mathrm{try}_2] \qquad \frac{\langle C,\ G\rangle \to^{+} \mathrm{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\ G\rangle \to \langle Q,\ G\rangle}$$

SOS: as-long-as-possible iteration

$$[\mathrm{alap}_1] \qquad \frac{\langle P,\ G\rangle \rightarrow^{+} H}{\langle P!,\ G\rangle \rightarrow \langle P!,\ H\rangle}$$

$$[\mathrm{alap}_2] \qquad \frac{\langle P,\ G\rangle \rightarrow^{+} \mathrm{fail}}{\langle P!,\ G\rangle \rightarrow G}$$

SOS: as-long-as-possible iteration

$$[\mathrm{alap}_1] \qquad \frac{\langle P,\ G\rangle \to^{+} H}{\langle P!,\ G\rangle \to \langle P!,\ H\rangle}$$

$$[\mathrm{alap}_2] \qquad \frac{\langle P,\ G\rangle \to^{+} \mathrm{fail}}{\langle P!,\ G\rangle \to G}$$

Note: $P!$ may be a nested loop

SOS: break

$$[\text{break}] \qquad \frac{\langle \texttt{break}; P,\ G\rangle}{\langle \texttt{break},\ G\rangle}$$

$$[\text{alap}_3] \qquad \frac{\langle P,\ G\rangle \rightarrow^* \langle \texttt{break}, H\rangle}{\langle P!,\ G\rangle \rightarrow H}$$

SOS: derived commands

$$[\mathrm{or}_1]\ \langle P\ \texttt{or}\ Q,\ G\rangle \rightarrow \langle P,\ G\rangle \qquad [\mathrm{or}_2]\ \langle P\ \texttt{or}\ Q,\ G\rangle \rightarrow \langle Q,\ G\rangle$$

$$[\mathrm{skip}]\ \langle \texttt{skip},\ G\rangle \rightarrow G \qquad [\mathrm{fail}]\ \langle \texttt{fail},\ G\rangle \rightarrow \mathrm{fail}$$

$$[\mathrm{if}_3]\ \frac{\langle C,\ G\rangle \rightarrow^{+} H}{\langle \texttt{if}\ C\ \texttt{then}\ P,\ G\rangle \rightarrow \langle P,\ G\rangle} \qquad [\mathrm{if}_4]\ \frac{\langle C,\ G\rangle \rightarrow^{+} \mathrm{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P,\ G\rangle \rightarrow G}$$

$$[\mathrm{try}_3]\ \frac{\langle C,\ G\rangle \rightarrow^{+} H}{\langle \texttt{try}\ C\ \texttt{then}\ P,\ G\rangle \rightarrow \langle P,\ H\rangle} \qquad [\mathrm{try}_4]\ \frac{\langle C,\ G\rangle \rightarrow^{+} \mathrm{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P,\ G\rangle \rightarrow G}$$

$$[\mathrm{try}_5]\ \frac{\langle C,\ G\rangle \rightarrow^{+} H}{\langle \texttt{try}\ C,\ G\rangle \rightarrow H} \qquad [\mathrm{try}_6]\ \frac{\langle C,\ G\rangle \rightarrow^{+} \mathrm{fail}}{\langle \texttt{try}\ C,\ G\rangle \rightarrow G}$$

Semantic function $[\![\_]\!]$

$[\![\_]\!] : \mathrm{ComSeq} \to (\mathcal{G} \to 2^{\mathcal{G} \cup \{\mathrm{fail},\, \bot\}})$ is defined by

$$
\begin{aligned}
[\![P]\!]G \quad = \quad & \{X \in \mathcal{G} \cup \{\mathrm{fail}\} \mid \langle P,\, G\rangle \xrightarrow{+} X\} \cup \\
& \{\bot \mid P \text{ can diverge or get stuck from } G\}
\end{aligned}
$$

where

- P *can diverge from* G if there is an infinite sequence $\langle P,\, G\rangle \to \langle P_1,\, G_1\rangle \to \langle P_2,\, G_2\rangle \to \dots$
- P *can get stuck from* G if there is a terminal configuration $\langle Q,\, H\rangle$ such that $\langle P,\, G\rangle \to^* \langle Q,\, H\rangle$ (where Q cannot be executed because no inference rule is applicable)

Semantic equivalence

Programs P and Q are *semantically equivalent* if $[\![P]\!] = [\![Q]\!]$

Examples

Semantic equivalence

Programs P and Q are *semantically equivalent* if $[\![P]\!] = [\![Q]\!]$

Examples

- $\texttt{skip} \equiv \texttt{null}$

  with `null`: $\emptyset \Rightarrow \emptyset$

Semantic equivalence

Programs P and Q are *semantically equivalent* if $[\![P]\!] = [\![Q]\!]$

Examples

- $\texttt{skip} \equiv \texttt{null}$

  with `null`: $\emptyset \Rightarrow \emptyset$
- $\texttt{fail} \equiv \{\}$

Semantic equivalence

Programs P and Q are *semantically equivalent* if $[\![P]\!] = [\![Q]\!]$

Examples

- $\texttt{skip} \equiv \texttt{null}$

  with `null`: $\emptyset \Rightarrow \emptyset$
- $\texttt{fail} \equiv \{\}$
- $\texttt{if}\ C\ \texttt{then}\ P \equiv \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{null}$

Semantic equivalence

Programs P and Q are *semantically equivalent* if $[\![P]\!] = [\![Q]\!]$

Examples

- $\texttt{skip} \equiv \texttt{null}$
  with $\texttt{null}$: $\emptyset \Rightarrow \emptyset$
- $\texttt{fail} \equiv \{\}$
- $\texttt{if } C \texttt{ then } P \equiv \texttt{if } C \texttt{ then } P \texttt{ else null}$
- $P \texttt{ or } Q \equiv \texttt{if } (\texttt{Delete!};\ \{\texttt{create}, \texttt{null}\};\ \texttt{node}) \texttt{ then } P \texttt{ else } Q$
  with $\texttt{Delete}$ removing edges and isolated nodes, $\texttt{create}$: $\emptyset \Rightarrow \bigcirc$ and $\texttt{node}$: $\underset{1}{\bigcirc} \Rightarrow \underset{1}{\bigcirc}$

Semantic equivalence

Programs P and Q are *semantically equivalent* if $[\![P]\!] = [\![Q]\!]$

Examples

- $\texttt{skip} \equiv \texttt{null}$
  with $\texttt{null}$: $\emptyset \Rightarrow \emptyset$
- $\texttt{fail} \equiv \{\}$
- $\texttt{if}\ C\ \texttt{then}\ P \equiv \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{null}$
- $P\ \texttt{or}\ Q \equiv \texttt{if}\ (\texttt{Delete!};\ \{\texttt{create}, \texttt{null}\};\ \texttt{node})\ \texttt{then}\ P\ \texttt{else}\ Q$
  with $\texttt{Delete}$ removing edges and isolated nodes, $\texttt{create}$: $\emptyset \Rightarrow \bigcirc$ and $\texttt{node}$: $\underset{1}{\bigcirc} \Rightarrow \underset{1}{\bigcirc}$
- $\texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q \not\equiv \texttt{if}\ C\ \texttt{then}\ (C; P)\ \texttt{else}\ Q$

Semantic equivalence

Programs P and Q are *semantically equivalent* if $[\![P]\!] = [\![Q]\!]$

Examples

- $\texttt{skip} \equiv \texttt{null}$
  with $\texttt{null}$: $\emptyset \Rightarrow \emptyset$
- $\texttt{fail} \equiv \{\}$
- $\texttt{if}\ C\ \texttt{then}\ P \equiv \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{null}$
- $P\ \texttt{or}\ Q \equiv \texttt{if}\ (\texttt{Delete!};\ \{\texttt{create}, \texttt{null}\};\ \texttt{node})\ \texttt{then}\ P\ \texttt{else}\ Q$
  with $\texttt{Delete}$ removing edges and isolated nodes, $\texttt{create}$: $\emptyset \Rightarrow \bigcirc$ and $\texttt{node}$: $\underset{1}{\bigcirc} \Rightarrow \underset{1}{\bigcirc}$
- $\texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q \not\equiv \texttt{if}\ C\ \texttt{then}\ (C; P)\ \texttt{else}\ Q$
  (choose $C = \texttt{skip}\ \texttt{or}\ \texttt{fail}$ and $P, Q = \texttt{skip}$)

Part V

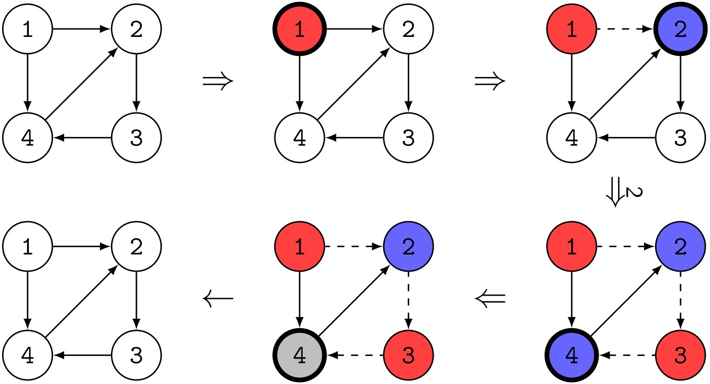
Case Study in Verification

Part VI

Miscellanea

Rooted programs

Example: rooted 2-colouring

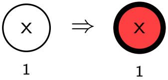


- Program implements a depth-first search.

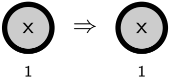
Example: rooted 2-colouring

```
Main = try (init; Colour!; if grey_root then fail)
Colour = (ColourNode; try Invalid then break)!; try Back else break
ColourNode = {colour_blue, colour_red} Invalid = {joined_blues, joined_reds}
Back = {back_blue, back_red}
```

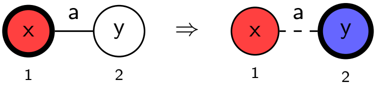
init(x:list)



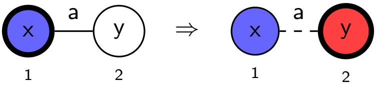
grey_root(x:list)



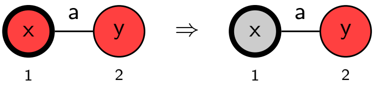
colour_blue(a,x,y:list)



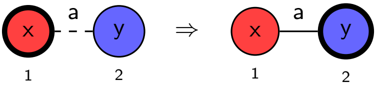
colour_red(a,x,y:list)



joined_reds(a,x,y:list)

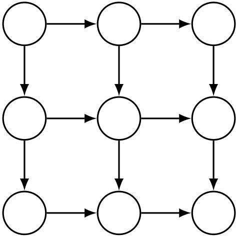


back_red(a,x,y:list)

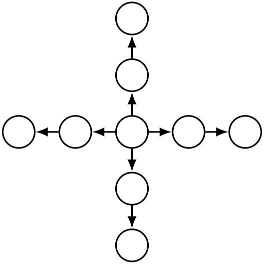


Runtime of rooted 2-colouring: GP 2 vs C

Comparison with Sedgewick's tailor-made program from *Algorithms in C* (Addison-Wesley, 2002)

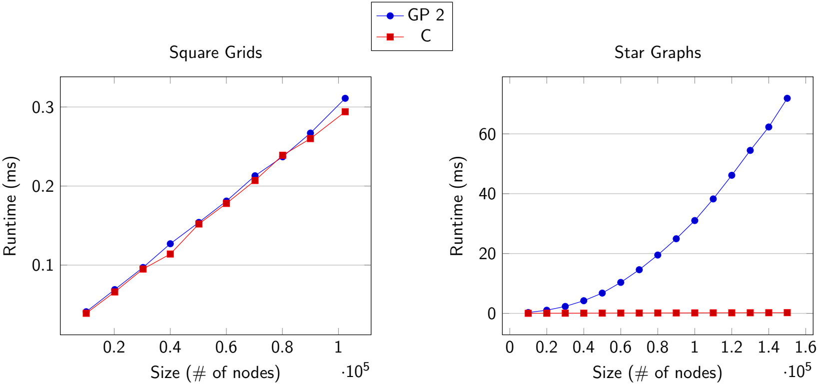


3 × 3 square grid

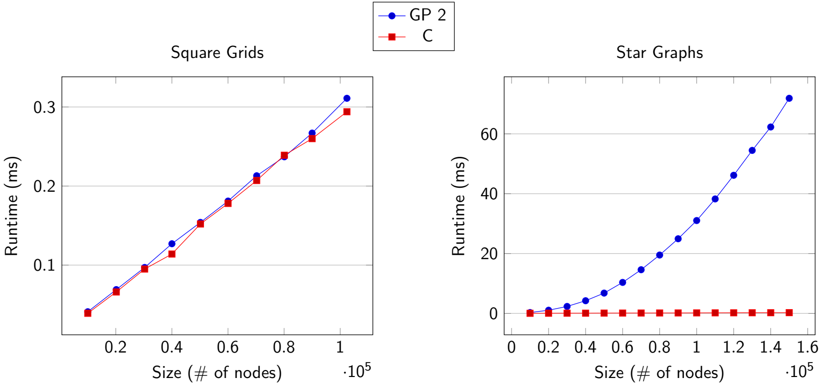


Star graph of degree 4

Runtime of rooted 2-colouring: GP 2 vs C



Runtime of rooted 2-colouring: GP 2 vs C



- In general, the GP 2 program runs in linear time on bounded-degree graphs.

Other topics in graph programs

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

Part VII

References

Language design, semantics and implementation

- From Imperative to Rule-based Graph Programs. *JLAMP 88*, 2017
- Compiling Graph Programs to C. *ICGT 2016*, LNCS 9761 (with C. Bak)
- A Reference Interpreter for the Graph Programming Language GP 2. *GaM 2015*, EPTCS 181 (with C. Bak, G. Faulkner and C. Runciman)
- Rooted Graph Programs. *GraBaTs 2012*, ECEASST 54 (with C. Bak)
- The Design of GP 2. *WRS 2011*, EPTCS 82
- The Semantics of Graph Programs. *RULE 2009*, EPTCS 21 (with S. Steinert)
- The Graph Programming Language GP. *CAI 2009*, LNCS 5725
- The GP Programming System. *GT-VMT 2008*, ECEASST 10 (with G. Manning)
- The York Abstract Machine. *GT-VMT 2006*, ENTCS 211 (with G. Manning)
- Towards Graph Programs for Graph Algorithms. *ICGT 2004*, LNCS 3256 (with S. Steinert)

Program verification

- Verifying a Copying Garbage Collector in GP 2. *GCM 2018*, to appear (with G. Wulandari)
- Reasoning about Graph Programs. *TERMGRAPH 2016*, EPTCS 225
- Verifying Monadic Second-Order Properties of Graph Programs. *ICGT 2014*, LNCS 8571 (with C. Poskitt)
- Verifying Total Correctness of Graph Programs. *GCM 2012*, ECEASST 61 (with C. Poskitt)
- Hoare-Style Verification of Graph Programs. *Fundamenta Informaticae 118*, 2012 (with C. Poskitt)
- A Hoare Calculus for Graph Programs. *ICGT 2010*, LNCS 6372 (with C. Poskitt)

Checking graph programs for confluence

- Checking Graph Programs for Confluence. *STAF 2017 Workshops*, LNCS 10748 (with I. Hristakiev)
- Towards Critical Pair Analysis for the Graph Programming Language GP 2. *WADT 2016*, LNCS 10644 (with I. Hristakiev)
- Attributed Graph Transformation via Rule Schemata: Church-Rosser Theorem. *STAF 2016 Workshops*, LNCS 9946 (with I. Hristakiev)
- A Unification Algorithm for GP 2. *GCM 2014*, ECEASST 71 (with I. Hristakiev)

Probabilistic GP 2

- Probabilistic Graph Programs for Randomised and Evolutionary Algorithms. *ICGT 2018*, LNCS 10887 (with T. Atkinson and S. Stepney)
- Evolving Graphs by Graph Programming. *EuroGP 2018*, LNCS 10781 (with T. Atkinson and S. Stepney)
- Probabilistic Graph Programs. *GCM 2017* (with T. Atkinson and S. Stepney)

Case studies in graph programming

- Minimizing Finite Automata with Graph Programs. *GCM 2010*, ECEASST 39 (with R. Suri and A. Singh)
- Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. *AGTIVE 2007*, LNCS 5088 (with G. Taentzer, E. Biermann, D. Bisztray, B. Bohnet, I. Boneva, A. Boronat, L. Geiger, R. Geiss, A. Horvath, O. Kniemeyer, T. Mens, B. Ness and T. Vajk)

PhD theses at York

- Ivaylo Hristakiev: *Confluence Analysis for a Graph Programming Language*, 2017
- Chris Bak: *GP 2: Efficient Implementation of a Graph Programming Language*, 2015
- Chris Poskitt: *Verification of Graph Programs*, 2013
- Sandra Steinert: *The Graph Programming Language GP*, 2007