The Graph Programming Language GP

Greg Manning, Detlef Plump, Sandra Steinert

The University of York
GP (Graph Programs)
GP (Graph Programs)

- Based on graph transformation rules
- Commands to control rule applications
- Non-deterministic
- Computationally complete
- Formal semantics
GP (Graph Programs)

- Based on graph transformation rules
- Commands to control rule applications
- Non-deterministic
- Computationally complete
- Formal semantics
Part I
The GP Language
Graph transformation rule
Graph transformation rule
Graph transformation rule: application
Graph transformation rule: application

1 2 ⇒ 1 2

↓

Diagram showing the transformation of a graph from one configuration to another.
Graph transformation rule: application

1 2
⇒
-Sep-20

1 2

-Sep-20

-Sep-20
Graph transformation rule: application

$1 \xrightarrow{2} \Rightarrow 1 \xrightarrow{2}$

$\downarrow \quad \downarrow$

$\supseteq \quad \subseteq$

$1 \xrightarrow{2}$
Graph transformation rule: application
Rule schema

reduce (a,b,x,y,z:int)

\[
x_1 \xrightarrow{a} y \xrightarrow{b} z \Rightarrow x_1 \xrightarrow{a+b} z_2
\]
reduce \((a, b, x, y, z:\text{int})\)

\[
\begin{array}{c}
x \\
1
\end{array} \quad \begin{array}{c}
y \\
2
\end{array} \quad \begin{array}{c}
z \\
1
\end{array} \quad \Rightarrow \quad \begin{array}{c}
x \\
1
\end{array} \quad \begin{array}{c}
y \\
2
\end{array} \quad \begin{array}{c}
z \\
1
\end{array}
\]

where \(x + a + b = z\) and not edge\((2, 1)\)
Example: Transitive closure

```
main = link!

link

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

\[ \text{main} = \text{reverse!}; \text{unmark!} \]

**reverse**

\[
\begin{array}{c}
\text{x} & \text{a} & \text{y} \\
1 & & 2
\end{array}
\quad \Rightarrow 
\begin{array}{c}
\text{x} & \text{a}_0 & \text{y} \\
1 & & 2
\end{array}
\]

**unmark**

\[
\begin{array}{c}
\text{x} & \text{a}_0 & \text{y} \\
1 & & 2
\end{array}
\quad \Rightarrow 
\begin{array}{c}
\text{x} & \text{a} & \text{y} \\
1 & & 2
\end{array}
\]
Example: Vertex colouring

\[
\text{main} = \text{init!}; \{\text{inc1, inc2}\}!
\]

\textbf{init}

\[
\begin{array}{c}
\text{x}_i \quad \Rightarrow \quad \text{x}_{i+1} \\
1 & & 1
\end{array}
\]

\textbf{inc1}

\[
\begin{array}{c}
\text{x}_i \quad \rightarrow \quad \text{y}_i \quad \Rightarrow \quad \text{x}_i \quad \rightarrow \quad \text{y}_{i+1} \\
1 & \text{a} & 2 & \text{a} & 2
\end{array}
\]

\textbf{inc2}

\[
\begin{array}{c}
\text{x}_i \quad \rightarrow \quad \text{y}_i \quad \Rightarrow \quad \text{x}_{i+1} \quad \rightarrow \quad \text{y}_i \\
1 & \text{a} & 2 & \text{a} & 2
\end{array}
\]
Example: 2-colouring

main = choose; \{\text{colour1, colour2}\}!; if illegal then undo!

choose
\[x \Rightarrow x_0\]

colour1
\[x_i \xrightarrow{a} y \Rightarrow x_i \xrightarrow{a} y_{1-i}\]

illegal
\[x_i \xrightarrow{a} y_i \Rightarrow x_i \xrightarrow{a} y_i\]

undo
\[x_i \Rightarrow x\]

colour2
\[x_i \xleftarrow{a} y \Rightarrow x_i \xleftarrow{a} y_{1-i}\]
Example: Sierpinski triangles

main = init; (inc; expand!)

init

\[
x \quad \Rightarrow \quad x_0
\]

\[
\begin{array}{c}
0 \\
1
\end{array}
\]

\[
0 \quad 1
\]

\[
2
\]

\[
x \quad \Rightarrow \quad x + 1
\]

\[
\begin{array}{c}
0 \\
1
\end{array}
\]

\[
\begin{array}{c}
y \\
y + 1
\end{array}
\]

where \( x > y \)

expand

\[
\begin{array}{c}
u \\
3
\end{array}
\]

\[
\begin{array}{c}
v \\
4
\end{array}
\]

\[
\begin{array}{c}
x_0 \\
1
\end{array}
\]

\[
\begin{array}{c}
x + 1 \\
y + 1
\end{array}
\]

\[
\begin{array}{c}
y + 1 \\
y + 1
\end{array}
\]

\[
\begin{array}{c}
u \\
3
\end{array}
\]

\[
\begin{array}{c}
0 \\
1
\end{array}
\]

\[
\begin{array}{c}
2 \\
1
\end{array}
\]
Sierpinski triangle (3rd generation)
GP abstract syntax

Prog ::= Decl {Decl}
Decl ::= RuleDecl | MacroDecl | MainDecl
MacroDecl ::= MacroId '==' ComSeq
MainDecl ::= main '==' ComSeq

ComSeq ::= Com {';' Com}
Com ::= RuleSetCall | MacroCall
     | if ComSeq then ComSeq [else ComSeq]
     | ComSeq '!' | skip | fail
RuleSetCall ::= RuleId | '{' [RuleId {',' RuleId}] '}'
MacroCall ::= MacroId
Structural operational semantics

Small-step transition relation

\[ \rightarrow \subseteq (\text{ComSeq} \times \mathcal{G}) \times ((\text{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\text{fail}\}) \]

defined by inference rules

Notation:

- \( P, P', Q, C \): command sequences (programs)
- \( \mathcal{R} \): rule-set call
- \( G, H, M \): graphs
Rule application

[Call₁] \[ \frac{G \Rightarrow_{\mathcal{R}} H}{\langle \mathcal{R}, G \rangle \rightarrow H} \]

[Call₂] \[ \frac{G \notin \text{Dom}(\Rightarrow_{\mathcal{R}})}{\langle \mathcal{R}, G \rangle \rightarrow \text{fail}} \]
Sequential composition

[Seq1] \[
\langle P, G \rangle \rightarrow \langle P', H \rangle
\]
\[
\langle P; Q, G \rangle \rightarrow \langle P'; Q, H \rangle
\]

[Seq2] \[
\langle P, G \rangle \rightarrow H
\]
\[
\langle P; Q, G \rangle \rightarrow \langle Q, H \rangle
\]

[Seq3] \[
\langle P, G \rangle \rightarrow \text{fail}
\]
\[
\langle P; Q, G \rangle \rightarrow \text{fail}
\]
Branching

[If\textsubscript{1}] \[
\langle C, G \rangle \rightarrow^+ H \\
\langle \text{if } C \text{ then } P \text{ else } Q, G \rangle \rightarrow \langle P, G \rangle
\]

[If\textsubscript{2}] \[
\text{C finitely fails on } G \\
\langle \text{if } C \text{ then } P \text{ else } Q, G \rangle \rightarrow \langle Q, G \rangle
\]
Finite failure

Program $C$ **finitely fails** on graph $G$:

1. **Termination:** there is no infinite transition sequence
   
   \[(C, G) \rightarrow (C_1, G_1) \rightarrow (C_2, G_2) \rightarrow \ldots\]

2. **Failure:** for every terminal configuration $\gamma$,
   
   \[\langle C, G \rangle \rightarrow^* \gamma \text{ implies } \gamma = \text{fail}\]

(Used in logic programming to define “negation as failure”)

Iteration

\[ [\text{Alap}_1] \quad \frac{\langle P, G \rangle \rightarrow^+ H}{\langle P!, G \rangle \rightarrow \langle P!, H \rangle} \]

\[ [\text{Alap}_2] \quad \frac{P \text{ finitely fails on } G}{\langle P!, G \rangle \rightarrow G} \]
Derived constructs

[Skip] \langle \text{skip}, G \rangle \rightarrow \langle r, G \rangle

where \( r \) is an identifier for \( \emptyset \Rightarrow \emptyset \)

[Fail] \langle \text{fail}, G \rangle \rightarrow \langle \{\}, G \rangle

[If_3] \langle \text{if } C \text{ then } P, G \rangle \rightarrow \langle \text{if } C \text{ then } P \text{ else skip, } G \rangle
Semantic function $S$

- $S : \text{ComSeq} \rightarrow (\mathcal{G} \rightarrow 2^{\mathcal{G} \cup \{\text{fail}\} \cup \{\bot\}})$ is defined by

$$S[P]G = \{X \in \mathcal{G} \cup \{\text{fail}\} \mid \langle P, G \rangle \xrightarrow{\dagger} X\} \cup \{\bot \mid P \text{ can diverge or get stuck from } G\}$$
Semantic function $S$

- $S : \text{ComSeq} \rightarrow (\mathcal{G} \rightarrow 2^{\mathcal{G} \cup \{\text{fail}\} \cup \{\bot\}})$ is defined by

$$S[P]G = \{X \in \mathcal{G} \cup \{\text{fail}\} \mid \langle P, G \rangle \Downarrow X\} \cup \{\bot \mid P \text{ can diverge or get stuck from } G\}$$

- $P$ and $Q$ are semantically equivalent if $S[P] = S[Q]$
Bounded nondeterminism

Lemma

For every program $P$ and graph $G$, if $S[P]G$ is infinite then $P$ can diverge from $G$. 
Bounded nondeterminism

Lemma

For every program $P$ and graph $G$, if $S[P]G$ is infinite then $P$ can diverge from $G$.

Example

main = \{stop, continue\}!

\begin{align*}
\text{stop} & \quad \text{continue} \\
1 & \Rightarrow \emptyset & 1 & \Rightarrow 1 2
\end{align*}

$S[\text{main}] 1 = \{\bot, \emptyset, 2, 2 2, 2 2 2, \ldots\}$
Part II
The GP Programming System
Overview

Graphical Editor

GP (textual)

Compiler

Bytecode

GXL

GXL to YAMG

YAM

GXL

YAMG
York Abstract Machine (YAM)

- Written in C
  - Input: YAM bytecode and graph files
  - Output: (sets of) graphs
- Executes low-level operations on graphs needed for graph matching and transformation
- Stack-based
- Maintains a current graph and allows fast queries
- Provides operations and mechanisms for backtracking
Edges and nodes are identified by integers

Graph structure contains ordered lists of integers representing nodes/edges with certain properties

These are intersected to generate complex searches (intersecting ordered lists is easy)
YAM: Backtracking

- A fail frame stores an enclosing function, a PC to jump to on fail, and the current age of the graph
- Bytecodes Fail, Assert and OnFail provide main functionality
- Also ClearFail, UpdateFail, Cut

<table>
<thead>
<tr>
<th></th>
<th>fail to</th>
<th>PC</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>topmost frame</td>
<td>as fail frame</td>
<td>rewind</td>
</tr>
<tr>
<td>ClearFail</td>
<td>given frame</td>
<td>increment</td>
<td>rewind</td>
</tr>
<tr>
<td>Cut</td>
<td>given frame</td>
<td>increment</td>
<td>leave</td>
</tr>
</tbody>
</table>
GP compiler

- GP to YAM bytecode compiler written in Haskell
- Generates searchplans for graph rules (search for each graph element in sequence, given what has already been found)
- Generates code implementing the effect of rules
- Generates code for backtracking
Compiler: Backtracking

- All queries return intersected lists
- Making the “correct” choice from such a list solves the nondeterminism problem
- Use a helper function to nondeterministically pick from a list by trying each member in turn:
  1. \( \text{answer} = 0 \)
  2. \( \text{answer} = \) next element in list greater than \( \text{answer} \)
  3. Fail if there are no more answers
  4. On failure, go to step 2
  5. Return \( \text{answer} \)
Editor

Graph Programs

Program Editor

Rule Name: expand

4: x - y

2: y

1: a

3: b

4: x - y

2: y + 1

y + 1

0 1 0 1

1: a

0 1

2: y + 1

0 1

1: a

3: b

Program Text

main = int (inc; expand());
Conclusion

- Solving graph problems at a high level of abstraction
- Rule-based, visual language
- Simple syntax and semantics to facilitate formal reasoning
- Reasonably fast implementation, faithfully matching the semantics: sound and, for terminating systems, complete
Future work

- Procedures (Sandra’s thesis)
- Typing: restricting the shape of graphs via labels; *graph types* defined by graph-reduction rules
- Static analysis: termination (techniques from term rewriting); determinism (critical-pair analysis)
- Hoare calculus for program verification
- Replacing graph transformation with other rule-based formalisms (string rewriting, term rewriting, . . . )
Appendix
Example: Destructive testing

```
make_even = if odd then add else skip
odd = delete!; check
```

**delete:**

\[
\begin{align*}
\text{x} & \rightarrow \text{y} \\
1 & \rightarrow 2
\end{align*}
\]

⇒

\[
\begin{align*}
\text{x} & \rightarrow \text{y} \\
1 & \rightarrow 2
\end{align*}
\]

**check:**

\[
\begin{align*}
\text{x} & \rightarrow \text{y} \\
1 & \rightarrow 2
\end{align*}
\]

⇒

\[
\begin{align*}
\text{x} & \rightarrow \text{y} \\
1 & \rightarrow 2
\end{align*}
\]

add:

\[
\begin{align*}
\emptyset & \rightarrow 1
\end{align*}
\]

⇒

\[
\begin{align*}
\emptyset & \rightarrow 1
\end{align*}
\]
**Example: Destructive testing**

\[
\text{make_even} = \begin{cases} 
\text{if odd then add else skip} \\
\text{odd} = \text{delete!}; \text{check}
\end{cases}
\]

\[
S[\text{make_even}]G = \begin{cases} 
\{ G + \{1\} \} & \text{if } |V_G| \text{ is odd,} \\
\{ G \} & \text{otherwise}
\end{cases}
\]
Many ways to find a subgraph (possible searchplans)
At compile time, nothing is known about the host graph
Guess which element it is best to start with, or decide at runtime
Runtime searchplan generation is hard
Compromise: Make a few possible searchplans at compile time, and choose one at runtime
YAM: Bytecode example

0  PushI(1)               9  PushI(2)
1  LNodesWithLabelSize    10  PushI(0)
2  PushI(1)               11  Load
3  PushI(0)               12  PushI(0)
4  LNodesWithLabel        13  PushI(2)
5  Func "PickFromList2"   14  Roll
6  Call                   15  Relabel_node
7  PushI(0)               16  Succ
8  Store