A Foundation for Functional Graph Programs

The Graph Transformation control Algebra (GTA)

Jens H. Weber

Department of Computer Science
University of Victoria
Canada, B.C.

13th Int. Workshop on Graph Computation Models (GCM 2022), July 2022

Outline

Graph Transformation Control Structures

Many practical applications of Graph Transformations (GT) require **control structures** to restrict or direct the application of GT rules.

Typical approaches:[1]

- Non-terminals
- Control expressions (alap, atomic, choice, conditional, ...)
- Integrity constraints
- Procedural abstractions

[1] R. Heckel & G. Taentzer (2020): Beyond Individual Rules: Usage Scenarios and Control Structures. In: Graph Transformation for Software Engineers, Springer

Implementing Control Structures is non-trivial

Most existing GT tools follow a **stateful** computational model
$\longrightarrow$ *The* Graph is destructively modified when GT rules are applied.

Non-determinism during rule application (matching) and rule selection must be dealt with, typically by using backtracking.

Transactional behaviour may be required, i.e., composite units of rule applications may need to be performed atomically and in isolation (ACID).

Implementing Control Structures is non-trivial

Most existing GT tools follow a **stateful** computational model
$\longrightarrow$ *The* Graph is destructively modified when GT rules are applied.

Non-determinism during rule application (matching) and rule selection must be dealt with, typically by using backtracking.

Transactional behaviour may be required, i.e., composite units of rule applications may need to be performed atomically and in isolation (ACID).

University of Victoria

Implementing Control Structures is non-trivial

Most existing GT tools follow a **stateful** computational model
$\longrightarrow$ *The* Graph is destructively modified when GT rules are applied.

Non-determinism during rule application (matching) and rule selection must be dealt with, typically by using backtracking.

Transactional behaviour may be required, i.e., composite units of rule applications may need to be performed atomically and in isolation (ACID).

GT Tools that Support Backtracking

Progres [2]

- sophisticated graph programming language with deterministic and non-deterministic operators
- semantic definition: approx. 300 pages
- Graph reconstruction (at choice points) based on non-standard graph database system GRAS "undo/redo" mechanism
- platform no longer maintained and abandoned

[2] Schurr et al. (1999): The PROGRES approach: Language and environment. In: Handbook Of Graph Grammars And Computing By Graph Transformation: Vol 2: Applications, Languages and Tools, World Scientific

GT Tools that Support Backtracking

Progres [2]

- sophisticated graph programming language with deterministic and non-deterministic operators
- semantic definition: approx. 300 pages
- Graph reconstruction (at choice points) based on non-standard graph database system GRAS “undo/redo” mechanism
- platform no longer maintained and abandoned

[2] Schurr et al. (1999): The PROGRES approach: Language and environment. In: Handbook Of Graph Grammars And Computing By Graph Transformation: Vol 2: Applications, Languages and Tools, World Scientific

GT Tools that Support Backtracking

Progres [2]

- sophisticated graph programming language with deterministic and non-deterministic operators
- semantic definition: approx. 300 pages
- Graph reconstruction (at choice points) based on non-standard graph database system GRAS "undo/redo" mechanism
- platform no longer maintained and abandoned

[2] Schurr et al. (1999): The PROGRES approach: Language and environment. In: Handbook Of Graph Grammars And Computing By Graph Transformation: Vol 2: Applications, Languages and Tools, World Scientific

GT Tools that Support Backtracking

Progres [2]

- sophisticated graph programming language with deterministic and non-deterministic operators
- semantic definition: approx. 300 pages
- Graph reconstruction (at choice points) based on non-standard graph database system GRAS "undo/redo" mechanism
- platform no longer maintained and abandoned

[2] Schurr et al. (1999): The PROGRES approach: Language and environment. In: Handbook Of Graph Grammars And Computing By Graph Transformation: Vol 2: Applications, Languages and Tools, World Scientific

GT Tools that Support Backtracking

GP / GP2 [3]

- GP has only 4 operators (n.d. rule application, sequence, branching, and iteration).
- Simple semantics: GP programs theoretically relate to a given input graph all possible output graphs
- Non-determinism left for execution mechanism to resolve
- Execution handled by York Abstract Machine (YAM), which can reconstruct graphs
- In practice: GP programs may diverge and not terminate
- GP2 adds operators and changes semantics of existing ones to disable backtracking.

[3] Plump (2009): The Graph Programming Language GP. In: Algebraic Informatics, LNCS 5725, Springer. Plump (2012): The Design of GP 2. EPTCS 82, 2012, pp. 1-16, doi:10.4204/EPTCS.82.1

GT Tools that Support Backtracking

GP / GP2 [3]

- GP has only 4 operators (n.d. rule application, sequence, branching, and iteration).
- Simple semantics: GP programs theoretically relate to a given input graph all possible output graphs
- Non-determinism left for execution mechanism to resolve
- Execution handled by York Abstract Machine (YAM), which can reconstruct graphs
- In practice: GP programs may diverge and not terminate
- GP2 adds operators and changes semantics of existing ones to disable backtracking.

[3] Plump (2009): The Graph Programming Language GP. In: Algebraic Informatics, LNCS 5725, Springer. Plump (2012): The Design of GP 2. EPTCS 82, 2012, pp. 1-16, doi:10.4204/EPTCS.82.1

GT Tools that Support Backtracking

GP / GP2 [3]

- GP has only 4 operators (n.d. rule application, sequence, branching, and iteration).
- Simple semantics: GP programs theoretically relate to a given input graph all possible output graphs
- Non-determinism left for execution mechanism to resolve
- Execution handled by York Abstract Machine (YAM), which can reconstruct graphs
- In practice: GP programs may diverge and not terminate
- GP2 adds operators and changes semantics of existing ones to disable backtracking.

---

[3] Plump (2009): The Graph Programming Language GP. In: Algebraic Informatics, LNCS 5725, Springer.
Plump (2012): The Design of GP 2. EPTCS 82, 2012, pp. 1-16, doi:10.4204/EPTCS.82.1

GT Tools that Support Backtracking

GP / GP2 [3]

- GP has only 4 operators (n.d. rule application, sequence, branching, and iteration).
- Simple semantics: GP programs theoretically relate to a given input graph all possible output graphs
- Non-determinism left for execution mechanism to resolve
- Execution handled by York Abstract Machine (YAM), which can reconstruct graphs
- In practice: GP programs may diverge and not terminate
- GP2 adds operators and changes semantics of existing ones to disable backtracking.

[3] Plump (2009): The Graph Programming Language GP. In: Algebraic Informatics, LNCS 5725, Springer. Plump (2012): The Design of GP 2. EPTCS 82, 2012, pp. 1-16, doi:10.4204/EPTCS.82.1

GT Tools that Support Backtracking

GP / GP2 [3]

- GP has only 4 operators (n.d. rule application, sequence, branching, and iteration).
- Simple semantics: GP programs theoretically relate to a given input graph all possible output graphs
- Non-determinism left for execution mechanism to resolve
- Execution handled by York Abstract Machine (YAM), which can reconstruct graphs
- In practice: GP programs may diverge and not terminate
- GP2 adds operators and changes semantics of existing ones to disable backtracking.

---

[3] Plump (2009): The Graph Programming Language GP. In: Algebraic Informatics, LNCS 5725, Springer. Plump (2012): The Design of GP 2. EPTCS 82, 2012, pp. 1-16, doi:10.4204/EPTCS.82.1

GT Tools that Support Backtracking

GP / GP2 [3]

- GP has only 4 operators (n.d. rule application, sequence, branching, and iteration).
- Simple semantics: GP programs theoretically relate to a given input graph all possible output graphs
- Non-determinism left for execution mechanism to resolve
- Execution handled by York Abstract Machine (YAM), which can reconstruct graphs
- In practice: GP programs may diverge and not terminate
- GP2 adds operators and changes semantics of existing ones to disable backtracking.

[3] Plump (2009): The Graph Programming Language GP. In: Algebraic Informatics, LNCS 5725, Springer. Plump (2012): The Design of GP 2. EPTCS 82, 2012, pp. 1-16, doi:10.4204/EPTCS.82.1

GT Tools that Support Backtracking

Grape/ GrapePress [4]

- Internal DSL to Clojure.
- 4 dedicated control structures (transact, n.d. rule matching, n.d. rule choice, conditional iteration)
- Backtracking implemented in Clojure run-time and based on Neo4J ACID transactions
- Available on Github and as Docker package
- Limitation similar to GP (programs may diverge)
- GrapePress adds computational notebook UI.

[4] Weber (2017): GRAPE – A Graph Rewriting and Persistence Engine. ICGT 2017, LNCS 10373
Weber (2021): GrapePress - A Computational Notebook for Graph Transformations. ICGT 2021, LNCS 12741

GT Tools that Support Backtracking

Grape/ GrapePress [4]

- Internal DSL to Clojure.
- 4 dedicated control structures (transact, n.d. rule matching, n.d. rule choice, conditional iteration)
- Backtracking implemented in Clojure run-time and based on Neo4J ACID transactions
- Available on Github and as Docker package
- Limitation similar to GP (programs may diverge)
- GrapePress adds computational notebook UI.

[4] Weber (2017): GRAPE – A Graph Rewriting and Persistence Engine. ICGT 2017, LNCS 10373
Weber (2021): GrapePress - A Computational Notebook for Graph Transformations. ICGT 2021, LNCS 12741

GT Tools that Support Backtracking

Grape/ GrapePress [4]

- Internal DSL to Clojure.
- 4 dedicated control structures (transact, n.d. rule matching, n.d. rule choice, conditional iteration)
- Backtracking implemented in Clojure run-time and based on Neo4J ACID transactions
- Available on Github and as Docker package
- Limitation similar to GP (programs may diverge)
- GrapePress adds computational notebook UI.

[4] Weber (2017): GRAPE – A Graph Rewriting and Persistence Engine. ICGT 2017, LNCS 10373
Weber (2021): GrapePress - A Computational Notebook for Graph Transformations. ICGT 2021, LNCS 12741

GT Tools that Support Backtracking

Grape/ GrapePress [4]

- Internal DSL to Clojure.
- 4 dedicated control structures (transact, n.d. rule matching, n.d. rule choice, conditional iteration)
- Backtracking implemented in Clojure run-time and based on Neo4J ACID transactions
- Available on Github and as Docker package
- Limitation similar to GP (programs may diverge)
- GrapePress adds computational notebook UI.

[4] Weber (2017): GRAPE – A Graph Rewriting and Persistence Engine. ICGT 2017, LNCS 10373
Weber (2021): GrapePress - A Computational Notebook for Graph Transformations. ICGT 2021, LNCS 12741

GT Tools that Support Backtracking

Grape/ GrapePress [4]

- Internal DSL to Clojure.
- 4 dedicated control structures (transact, n.d. rule matching, n.d. rule choice, conditional iteration)
- Backtracking implemented in Clojure run-time and based on Neo4J ACID transactions
- Available on Github and as Docker package
- Limitation similar to GP (programs may diverge)
- GrapePress adds computational notebook UI.

[4] Weber (2017): GRAPE – A Graph Rewriting and Persistence Engine. ICGT 2017, LNCS 10373
Weber (2021): GrapePress - A Computational Notebook for Graph Transformations. ICGT 2021, LNCS 12741

GT Tools that Support Backtracking

Grape/ GrapePress [4]

- Internal DSL to Clojure.
- 4 dedicated control structures (transact, n.d. rule matching, n.d. rule choice, conditional iteration)
- Backtracking implemented in Clojure run-time and based on Neo4J ACID transactions
- Available on Github and as Docker package
- Limitation similar to GP (programs may diverge)
- GrapePress adds computational notebook UI.

[4] Weber (2017): GRAPE – A Graph Rewriting and Persistence Engine. ICGT 2017, LNCS 10373
Weber (2021): GrapePress - A Computational Notebook for Graph Transformations. ICGT 2021, LNCS 12741

Other GT Tools - w/o backtracking

Other tools avoid backtracking altogether, for example:

- AGG (Taentzer 2004): “random” choice, CP analysis to avoid non-determinism
- FUJABA (Nickel et al., 2000): activity / story diagrams
- GrGen (Geiß et al. 2006): textual language similar to regular expressions
- GReAT (Agrawal et al. 2006): textual model trafo language + OCL constraints, multiple graphs
- GROOVE (Rensink 2004): rule priorities, breadth-first exploration with collision detection

Other GT Tools - w/o backtracking

Other tools avoid backtracking altogether, for example:

- AGG (Taentzer 2004): “random” choice, CP analysis to avoid non-determinism
- FUJABA (Nickel et al., 2000): activity / story diagrams
- GrGen (Geiß et al. 2006): textual language similar to regular expressions
- GReAT (Agrawal et al. 2006): textual model trafo language + OCL constraints, multiple graphs
- GROOVE (Rensink 2004): rule priorities, breadth-first exploration with collision detection

Other GT Tools - w/o backtracking

Other tools avoid backtracking altogether, for example:

- AGG (Taentzer 2004): “random” choice, CP analysis to avoid non-determinism
- FUJABA (Nickel et al., 2000): activity / story diagrams
- GrGen (Geiß et al. 2006): textual language similar to regular expressions
- GReAT (Agrawal et al. 2006): textual model trafo language + OCL constraints, multiple graphs
- GROOVE (Rensink 2004): rule priorities, breadth-first exploration with collision detection

Other GT Tools - w/o backtracking

Other tools avoid backtracking altogether, for example:

- AGG (Taentzer 2004): “random” choice, CP analysis to avoid non-determinism
- FUJABA (Nickel et al., 2000): activity / story diagrams
- GrGen (Geiß et al. 2006): textual language similar to regular expressions
- GReAT (Agrawal et al. 2006): textual model trafo language + OCL constraints, multiple graphs
- GROOVE (Rensink 2004): rule priorities, breadth-first exploration with collision detection

Other GT Tools - w/o backtracking

Other tools avoid backtracking altogether, for example:

- AGG (Taentzer 2004): “random” choice, CP analysis to avoid non-determinism
- FUJABA (Nickel et al., 2000): activity / story diagrams
- GrGen (Geiß et al. 2006): textual language similar to regular expressions
- GReAT (Agrawal et al. 2006): textual model trafo language + OCL constraints, multiple graphs
- GROOVE (Rensink 2004): rule priorities, breadth-first exploration with collision detection

Observations

Non-determinism is theoretically appealing (simple, but ...

Simple control structures, but leave complexity to interpretation mechanism.

Interpreters have limitations

Programs may diverge, execution may be inefficient, choices may be ignored

Program assurance?

Theoretically computable solutions for graph programs may not be computable in practice, given current tools. Problematic for Engineering applications that need assurance.

Functional Graph Rewriting

- Graphs are immutable
- Explicit I/O parameter (rather than implicit global variable)
- Non-deterministic operators replaced by deterministic operators that produce *sets* of graphs
- Simple realization of ACID transactions based on the notion of *Graph Processes*, i.e., unsuccessful executions can simply be "forgotten" (Baldan, ICGT 2006).
- Tool: *GrapeVine*[5]

[5] Weber (2022): Tool support for Fully-Persistent Graph Rewriting - GrapeVine. ICGT 2022, LNCS 13349

Functional Graph Rewriting

- Graphs are immutable
- Explicit I/O parameter (rather than implicit global variable)
- Non-deterministic operators replaced by deterministic operators that produce *sets* of graphs
- Simple realization of ACID transactions based on the notion of *Graph Processes*, i.e., unsuccessful executions can simply be “forgotten” (Baldan, ICGT 2006).
- Tool: *GrapeVine*[5]

[5] Weber (2022): Tool support for Fully-Persistent Graph Rewriting - GrapeVine. ICGT 2022, LNCS 13349

Functional Graph Rewriting

- Graphs are immutable
- Explicit I/O parameter (rather than implicit global variable)
- Non-deterministic operators replaced by deterministic operators that produce *sets* of graphs
- Simple realization of ACID transactions based on the notion of *Graph Processes*, i.e., unsuccessful executions can simply be “forgotten” (Baldan, ICGT 2006).
- Tool: *GrapeVine*[5]

[5] Weber (2022): Tool support for Fully-Persistent Graph Rewriting - GrapeVine. ICGT 2022, LNCS 13349

Functional Graph Rewriting

- Graphs are immutable
- Explicit I/O parameter (rather than implicit global variable)
- Non-deterministic operators replaced by deterministic operators that produce *sets* of graphs
- Simple realization of ACID transactions based on the notion of *Graph Processes*, i.e., unsuccessful executions can simply be “forgotten” (Baldan, ICGT 2006).
- Tool: *GrapeVine*[5]

[5] Weber (2022): Tool support for Fully-Persistent Graph Rewriting - GrapeVine. ICGT 2022, LNCS 13349

Functional Graph Rewriting

- Graphs are immutable
- Explicit I/O parameter (rather than implicit global variable)
- Non-deterministic operators replaced by deterministic operators that produce *sets* of graphs
- Simple realization of ACID transactions based on the notion of *Graph Processes*, i.e., unsuccessful executions can simply be “forgotten” (Baldan, ICGT 2006).
- Tool: *GrapeVine*[5]

[5] Weber (2022): Tool support for Fully-Persistent Graph Rewriting - GrapeVine. ICGT 2022, LNCS 13349

Graph Transformations

Graph

A tuple $G : (N, E, s, t)$ where N is a finite set of *nodes*, E is a finite set of *edges*, and $s, t : E \to N$ are total *source* and *target* functions, respectively.

Rule

A (GT) *rule* is a pair of graph morphisms $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$

Transformation

An application of a rule $r : L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ to a given host graph G requires the existence of a graph morphism $L \xrightarrow{m} G$ (a *match*) . The application deletes all elements $m(L - I)$ and creates co-matched elements $m'(R - I)$, while embedding them in the context $m(I)$. The transformation of a graph G into a graph G' with rule r at a match m is denoted as $G \overset{r,m}{\rightsquigarrow} G'$.

Constrained Graph Transformations

Constraints are an important means of limiting (and controlling) the application of (graph transformation) rules. Our notion of graph constraints implements Orejas et al's "logic of graph constraints" (2008).

Constrained Graph

A tuple (G, K) where G is a graph and K is a finite set of *graph constraints* satisfied by G, i.e, $\forall \kappa \in K : G \models \kappa$.

Transformation (of a constrained graph)

A transformation of a constrained graph $(G, K) \overset{r,m}{\rightsquigarrow} (G', K)$ exists, if there exists a corresponding transformation for the unconstraint graph $G \overset{r,m}{\rightsquigarrow} G'$, where the resulting graph satisfies all constraints, i.e., $\forall \kappa \in K : G' \models \kappa$.

(In the following, we use "graph" to refer to "constrained graph".)

(Programmed) Graph Transformation System

(Programmed) Graph Transformation System

A (programmed) *Graph Transformation System* (GTS) as a tuple (R, C, P), where R is a set of *rules*, C is a set of *constraints*, and P is a set of graph *programs*.

The Graph Transformation control Algebra (GTA)

Algebra with a single data type: *grape*

Graph set enumeration (grape)

A non-empty sequence $\dddot{G} : \langle \bar{G}_1, \bar{G}_2, .., \bar{G}_n \rangle$ where each $\bar{G}_i$ is a finite set of graphs. Let $\dddot{\mathbb{G}}$ denote the domain (data type) of *grapes*.

12 GTA operators:

- 10 deterministic (functional) operators with signature $\dddot{\mathbb{G}} \rightarrow \dddot{\mathbb{G}}$
- 2 non-deterministic (choice) operators $\dddot{\mathbb{G}} \rightsquigarrow \dddot{\mathbb{G}}$

The choice operators are added for efficiency reasons only.

Graph Programs - *Syntax*

Given a GTS (R, C, P), $c \in C$ and $r \in R$, each program $p \in P$ is a GTA expression, which is defined as one of the following:

- $\triangle(c)$ and $\boxtimes(c)$ are GTA expressions, called *constrain* and *unconstrain*, respectively ;
- $\twoheadrightarrow(r)$ and $\rightsquigarrow(r)$ are GTA expressions, called *derive* and *derive-choice*, respectively;
- $\circledcirc(n, \lesssim)$ with $n \in \mathbb{N}$ and a total order $\lesssim$ on graphs is a GTA expression, called *select*;
- $\cdot|\cdot(e_1, e_2)$, $\div(e_1, e_2)$ and $\dot{\div}(e_1, e_2)$ are GTA expressions, if e_1 and e_2 are GTA expressions; They are called *sequence*, *alternative*, and *alternative-choice* respectively;
- $\circlearrowright(e)$ is a GTA expression called *loop* if e is a GTA expression;
- $\looparrowright(c, e)$ is a GTA expression called *search* if e is a GTA expression;
- $)($ and $\not\equiv$ are GTA expressions called *new* and *distinct*, respectively.

Graph Programs - *Semantics*

A graph program that does not use any of the two non-deterministic operators are called *deterministic*. Its semantic is given by a function $\dddot{\mathbb{G}} \rightarrow \dddot{\mathbb{G}}$, based on the semantics definition of the individual GTA operators.

Constrain ($\triangle$)

$\triangle(c)$ declares constraint c on all the graphs in the last element of a given *grape* that satisfy c. All other graphs are removed.

$[\![\triangle(c)]\!](\langle .., \bar{G}_n \rangle) = \langle .., \bar{G}'_n \rangle$ with $\bar{G}'_n = \{(G, K + \{c\}) \,|\, (G, K) \in \bar{G}_n \wedge G \models c\}$

Unconstrain ($\boxtimes$)

$\boxtimes(c)$ removes constraint c from the graphs in the last element of a grape:

$[\![\boxtimes(c)]\!](\langle .., \bar{G}_n \rangle) = \langle .., \bar{G}'_n \rangle$ and $\bar{G}'_n = \{(G, K - \{c\}) \,|\, (G, K) \in \bar{G}_n\}$

University of Victoria

Derive ($\twoheadrightarrow$)

Computes all direct linear derivation of each graph in the last element of a given *grape* and extends the given input *grape* with an element that contains *all* resulting graphs, i.e., $[\![\twoheadrightarrow (r)]\!](\langle .., \bar{G}_n \rangle) = \langle .., \bar{G}_n, \bar{G}_{n+1} \rangle$ where $\bar{G}_{n+1} = \{G' | \exists G \in \bar{G}_n : G \overset{r}{\rightsquigarrow} G'\}$

Select (⊚)

Function *Select* ($\circledcirc(k, \lesssim)$) reduces the last element of a given *grape* to at most k elements. The selection is determined by a total order on graphs $\lesssim$. Formally,
$[\![\circledcirc(k, \lesssim)]\!](\langle .., \bar{G}_n \rangle) = \langle .., \bar{G}'_n \rangle$, with $\bar{G}'_n \subseteq \bar{G}_n \wedge |\bar{G}'_n| \leq k \wedge |\bar{G}'_n| \leq |\bar{G}_n| \wedge \nexists G \in \bar{G}_n - \bar{G}'_n, G' \in \bar{G}'_n : G' \lesssim G$

Sequence (·|·)

$\cdot|\cdot(a, b)$ composes two GTA expressions sequentially by using relational composition, i.e., $[\![\cdot|\cdot(a, b)]\!] = \{(\dddot{G}, \dddot{K}) \in \mathbb{G} | (\dddot{G}, \dddot{H}) \in [\![a]\!] \wedge (\dddot{H}, \dddot{K}) \in [\![b]\!]\}$.

Alternative (÷)

$\div(a, b)$ composes two GTA expressions (a and b) as alternatives by extending a given *grape* with a new element that is the union of the last elements of the *grapes* produced by interpreting the two expressions, i.e., $[\![\div(a, b)]\!](\dddot{G} : \langle ..x \rangle) = \langle ..x, \bar{O}_1 \cup \bar{O}_2 \rangle$ with $[\![a]\!](\dddot{G}) = \langle ..y, \bar{O}_1 \rangle$ and $[\![b]\!](\dddot{G}) = \langle ..z, \bar{O}_2 \rangle$.

Distinct (≢)

Graph exploration may produce graphs that are identical (up to isomorphism). The *distinct* operator (≢) removes all graphs from the last element of a given *grape*, if they are identical to any other graph in the *grape*.

$$[\![\not\equiv]\!](\langle \bar{G}_1, .., \bar{G}_n \rangle) = \begin{cases} \langle \bar{G}_1, .., [\![\not\equiv]\!](\bar{G}_n - \{D\}) \rangle, & \text{if } \exists D, J \in \bigcup_{1 \leq i \leq n} \bar{G}_i : D \neq J \land D \cong J \\ \langle \bar{G}_1, .., \bar{G}_n \rangle, & \text{otherwise} \end{cases}$$

New ()()

The *new* operator is used to start a new *grape*. It takes a *grape* as input but “forgets” all but the last element in the sequence.
$[\![ )( ]\!](\langle .., \bar{G}_n \rangle) = \langle \bar{G}_n \rangle$.

Loop ($\circlearrowright$)

$\circlearrowright(e)$ recursively interprets GTA expression e on the most recently computed *grape* while the last element is not empty, i.e,

$$[\![\circlearrowright(e)]\!](\dddot{G}) = \begin{cases} \dddot{G}, & \text{if } [\![e]\!](\dddot{G}) = \langle .., \varnothing \rangle \\ [\![\circlearrowright(e)]\!] \circ [\![e]\!](\dddot{G}) & \text{otherwise} \end{cases}$$

Search ($\looparrowright$)

$\looparrowright (c, o)$ recursively interprets a GTA expression o on the most recently computed *grape* while none of the graphs in the last element of the current *grape* satisfy constraint c and the last element is not empty, i.e,

$$[\![\looparrowright (c, o)]\!](\dddot{G} : \langle .., \bar{G}_n \rangle) = \begin{cases} \dddot{G}, & \text{if } \bar{G}_n = \varnothing \vee \exists G \in \bar{G}_n : G \models c \\ [\![\looparrowright (c, o)]\!] \circ [\![o]\!](\ddot{G}) & \text{otherwise} \end{cases}$$

University of Victoria

Semantics of the two non-deterministic operators

Derive-choice ($\rightsquigarrow$)

$\rightsquigarrow$ is interpreted as a relation that extends the classical notion of non-deterministic rule application to the data type of *grapes*.

$[\![\rightsquigarrow(r)]\!] = \{(\langle .., \bar{G}_n \rangle, \langle .., \bar{G}_n, \bar{G}_{n+1} \rangle) \in \dddot{\mathbb{G}} \times \dddot{\mathbb{G}} | \forall G \in \bar{G}_n : ((\exists! X \in \bar{G}_{n+1} : G \overset{r}{\rightsquigarrow} X) \vee (\nexists Y \in \mathbb{G} : G \overset{r}{\rightsquigarrow} Y)) \wedge |\bar{G}_{n+1}| \leq |\bar{G}_n|\}$.

Alternative-choice ($\overset{.}{\underset{.}{\sim}}$)

$\overset{.}{\underset{.}{\sim}}$ is interpreted as a relation that makes a non-deterministic choice between the relations implied by the two GTA expressions, i.e., $[\![\overset{.}{\underset{.}{\sim}}(a, b)]\!] = [\![a]\!] \vee [\![\overset{.}{\underset{.}{\sim}}(a, b)]\!] = [\![b]\!]$.

Properties of the GTA and Rationale

- Computationally complete, but not minimal (n.d. rule application, iteration and sequential composition sufficient and minimal [6])
- non-deterministic operators not needed but included for “performance” reasons
- Operators *Constrain* (△) and *Unconstrain* (⨂) help limit exploration by using graph constraints.
- Operator *Select* (⊚) is included to limit exploration by allowing for heuristic search.
- Operator *Distinct* (≢) is included to avoid state collision during solution exploration.

[6] Habel Plump (2001): Computational Completeness of Programming Languages Based on Graph Transformation. Foundations of Software Science and Computation Structures, Springer

University of Victoria

Properties of the GTA and Rationale

- Computationally complete, but not minimal (n.d. rule application, iteration and sequential composition sufficient and minimal [6])
- non-deterministic operators not needed but included for "performance" reasons
- Operators *Constrain* (△) and *Unconstrain* (⨻) help limit exploration by using graph constraints.
- Operator *Select* (◎) is included to limit exploration by allowing for heuristic search.
- Operator *Distinct* (≢) is included to avoid state collision during solution exploration.

[6] Habel Plump (2001): Computational Completeness of Programming Languages Based on Graph Transformation. Foundations of Software Science and Computation Structures, Springer

Properties of the GTA and Rationale

- Computationally complete, but not minimal (n.d. rule application, iteration and sequential composition sufficient and minimal [6])
- non-deterministic operators not needed but included for “performance” reasons
- Operators *Constrain* (△) and *Unconstrain* (⨻) help limit exploration by using graph constraints.
- Operator *Select* (⊚) is included to limit exploration by allowing for heuristic search.
- Operator *Distinct* (≢) is included to avoid state collision during solution exploration.

[6] Habel Plump (2001): Computational Completeness of Programming Languages Based on Graph Transformation. Foundations of Software Science and Computation Structures, Springer

Properties of the GTA and Rationale

- Computationally complete, but not minimal (n.d. rule application, iteration and sequential composition sufficient and minimal [6])
- non-deterministic operators not needed but included for “performance” reasons
- Operators *Constrain* (△) and *Unconstrain* (⨻) help limit exploration by using graph constraints.
- Operator *Select* (⊚) is included to limit exploration by allowing for heuristic search.
- Operator *Distinct* (≢) is included to avoid state collision during solution exploration.

---

[6] Habel Plump (2001): Computational Completeness of Programming Languages Based on Graph Transformation. Foundations of Software Science and Computation Structures, Springer

Properties of the GTA and Rationale

- Computationally complete, but not minimal (n.d. rule application, iteration and sequential composition sufficient and minimal [6])
- non-deterministic operators not needed but included for "performance" reasons
- Operators *Constrain* (△) and *Unconstrain* (⨂) help limit exploration by using graph constraints.
- Operator *Select* (⊚) is included to limit exploration by allowing for heuristic search.
- Operator *Distinct* (≢) is included to avoid state collision during solution exploration.

[6] Habel Plump (2001): Computational Completeness of Programming Languages Based on Graph Transformation. Foundations of Software Science and Computation Structures, Springer

University of Victoria

GrapeVine overview

- major new (functional) revision of its ancestor tool *Grape/GrapePress*
- internal DSL to Clojure (JVM)
- based on Neo4J graph database
- Computational notebook front-end (optional)
- Graphs are stored in fully-persistent data structure[7]

[7] Weber (2022): Tool support for Fully-Persistent Graph Rewriting - GrapeVine. In: ICGT 2022, LNCS 13349, Springer

GrapeVine overview

- major new (functional) revision of its ancestor tool *Grape*/*GrapePress*
- internal DSL to Clojure (JVM)
- based on Neo4J graph database
- Computational notebook front-end (optional)
- Graphs are stored in fully-persistent data structure[7]

[7] Weber (2022): Tool support for Fully-Persistent Graph Rewriting - GrapeVine. In: ICGT 2022, LNCS 13349, Springer

GrapeVine overview

- major new (functional) revision of its ancestor tool *Grape/GrapePress*
- internal DSL to Clojure (JVM)
- based on Neo4J graph database
- Computational notebook front-end (optional)
- Graphs are stored in fully-persistent data structure[7]

[7] Weber (2022): Tool support for Fully-Persistent Graph Rewriting - GrapeVine. In: ICGT 2022, LNCS 13349, Springer

GrapeVine overview

- major new (functional) revision of its ancestor tool *Grape/GrapePress*
- internal DSL to Clojure (JVM)
- based on Neo4J graph database
- Computational notebook front-end (optional)
- Graphs are stored in fully-persistent data structure[7]

[7] Weber (2022): Tool support for Fully-Persistent Graph Rewriting - GrapeVine. In: ICGT 2022, LNCS 13349, Springer

GrapeVine overview

- major new (functional) revision of its ancestor tool *Grape/GrapePress*
- internal DSL to Clojure (JVM)
- based on Neo4J graph database
- Computational notebook front-end (optional)
- Graphs are stored in fully-persistent data structure[7]

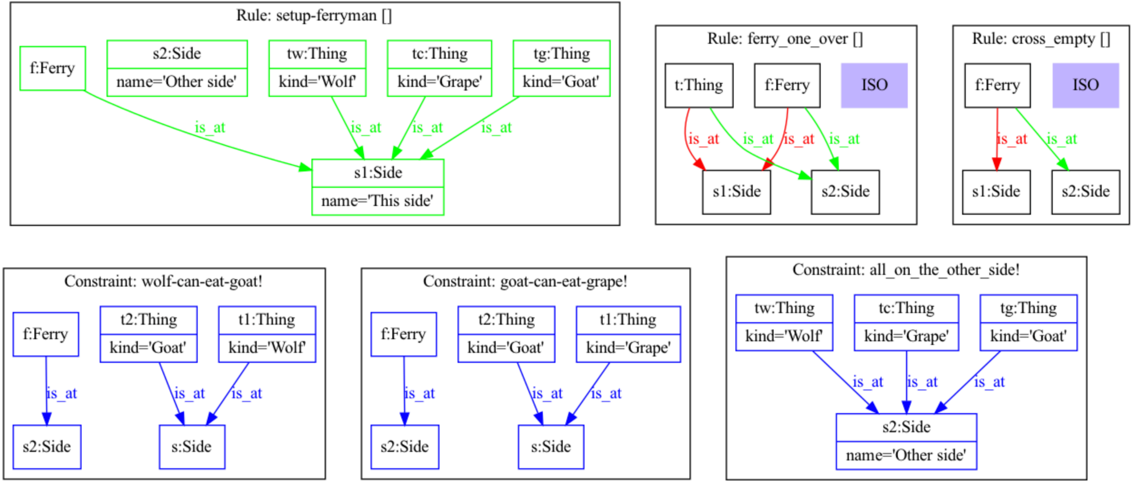
[7] Weber (2022): Tool support for Fully-Persistent Graph Rewriting - GrapeVine. In: ICGT 2022, LNCS 13349, Springer

GrapeVine Control Structures and their GTA semantics:

Description	GTA expression	*GrapeVine* Syntax
Rule application (deterministic)	$\twoheadrightarrow(r)$	`r`
Rule application (non-deterministic)	$\rightsquigarrow(r)$	`r~`
Add constraint *c*	$\triangle(c)$	`(schema c ..)`
Add constraint negated *c*	$\triangle(\neg c)$	`(schema c- ..)`
Remove constraint *c*	$\not\triangle_c$	`(schema-drop c ..)`
Remove negated constraint *c*	$\not\triangle(\neg c)$	`(schema-drop c- ..)`
Check constraint *c*	$\cdot\!\vert\!\cdot(\triangle(c), \not\triangle(c))$	`c`
Check negated constraint *c*	$\cdot\!\vert\!\cdot(\triangle(\neg c), \not\triangle(\neg c))$	`c-`
Sequence	$\cdot\!\vert\!\cdot(e_1, e_2)$	`(-> e_1 e_2 ..)`
Alternative (deterministic)	$\div(e_1, e_2)$	`(|| e_1 e_2 ..)`
Alternative (non-deterministic)	$\tilde{\div}(e_1, e_2..)$	`(||~ e_1 e_2 ..)`
Loop (while possible)	$\circlearrowright(e)$	`(->* e ..)`
Until (without collisions check)	$\looparrowright(c, e)$	`(->?* c e ..)`
Until (with collision check)	$\looparrowright(c, \cdot\!\vert\!\cdot(e, \not\equiv)) \circ)($	`(->?+ c e ..)`
New	$)($	`newgrape`
creates a *grape* with a single element containing an empty graph	$)(\; (\langle(\varnothing, \varnothing)\rangle)$	`(newgrape)`
Distinct	$\not\equiv$	`dist`
Select	$\circledcirc(k, v)$	`(select k v)`

A Simple Example: Ferryman

Visual representation of rules and constraints:



GrapeVine Program

```
(-> (newgrape) setup-ferryman
    (->?+ all_on_the_other_side!
        (|| ferry_one_over cross_empty)
            wolf-can-eat-goat!-
            goat-can-eat-grape!-)))
```

or, alternatively,

```
(-> (newgrape)
    (schema wolf-can-eat-goat!- goat-can-eat-grape!-)
    setup-ferryman
    (->?+ all_on_the_other_side
        (|| ferry_one_over cross_empty)))
```

Efficiency considerations

Key prerequisite: fully-persistent data structure for graphs.

The *Distinct* operator may appear expensive, however graphs are compared based on hashed fingerprints that are indexed in the database (O(log n)).

Run-time experiment:

- Program takes approx. 7 sec. (creating 27 graphs).
- Running the program 1,000 times creates 27,000 graphs
- The next program run still takes about 7 seconds.
- Running a modified program that uses the *Until* operator *without* collision check (->?*) creates 216 graphs and takes 52 secs.
- Similar to backtracking-based solution using the former *Grape*, which takes about 50 seconds. *Grape* (like Progres and GP) cannot find a solution without limiting the steps.

University of Victoria

Efficiency considerations

Key prerequisite: fully-persistent data structure for graphs.

The *Distinct* operator may appear expensive, however graphs are compared based on hashed fingerprints that are indexed in the database (O(log n)).

Run-time experiment:

- Program takes approx. 7 sec. (creating 27 graphs).
- Running the program 1,000 times creates 27,000 graphs
- The next program run still takes about 7 seconds.
- Running a modified program that uses the *Until* operator *without* collision check (->?*) creates 216 graphs and takes 52 secs.
- Similar to backtracking-based solution using the former *Grape*, which takes about 50 seconds. *Grape* (like Progres and GP) cannot find a solution without limiting the steps.

University of Victoria

Efficiency considerations

Key prerequisite: fully-persistent data structure for graphs.

The *Distinct* operator may appear expensive, however graphs are compared based on hashed fingerprints that are indexed in the database (O(log n)).

Run-time experiment:

- Program takes approx. 7 sec. (creating 27 graphs).
- Running the program 1,000 times creates 27,000 graphs
- The next program run still takes about 7 seconds.
- Running a modified program that uses the *Until* operator *without* collision check (`->?*`) creates 216 graphs and takes 52 secs.
- Similar to backtracking-based solution using the former *Grape*, which takes about 50 seconds. *Grape* (like Progres and GP) cannot find a solution without limiting the steps.

Conclusions and Future Work

- The abstraction provided by non-deterministic operators is appealing due to its theoretical simplicity. However, it shifts complexity to the interpretation mechanism and limits the assurances provided by programs in practice.
- Functional graph rewriting seeks to avoid non-determinism by computing *sets* of graphs (all possible results)
- Graph state collision detection, heuristic selection, and graph constraints help limit the number of graphs explored in programs.
- Our current work is on performing more rigorous performance and scalability analyses, using standard benchmarks.

Conclusions and Future Work

- The abstraction provided by non-deterministic operators is appealing due to its theoretical simplicity. However, it shifts complexity to the interpretation mechanism and limits the assurances provided by programs in practice.
- Functional graph rewriting seeks to avoid non-determinism by computing *sets* of graphs (all possible results)
- Graph state collision detection, heuristic selection, and graph constraints help limit the number of graphs explored in programs.
- Our current work is on performing more rigorous performance and scalability analyses, using standard benchmarks.

Conclusions and Future Work

- The abstraction provided by non-deterministic operators is appealing due to its theoretical simplicity. However, it shifts complexity to the interpretation mechanism and limits the assurances provided by programs in practice.
- Functional graph rewriting seeks to avoid non-determinism by computing *sets* of graphs (all possible results)
- Graph state collision detection, heuristic selection, and graph constraints help limit the number of graphs explored in programs.
- Our current work is on performing more rigorous performance and scalability analyses, using standard benchmarks.

Conclusions and Future Work

- The abstraction provided by non-deterministic operators is appealing due to its theoretical simplicity. However, it shifts complexity to the interpretation mechanism and limits the assurances provided by programs in practice.
- Functional graph rewriting seeks to avoid non-determinism by computing *sets* of graphs (all possible results)
- Graph state collision detection, heuristic selection, and graph constraints help limit the number of graphs explored in programs.
- Our current work is on performing more rigorous performance and scalability analyses, using standard benchmarks.

Thank you!

Available in Docker: github.com/jenshweber/grape

Tool demo on Friday @noon (ICGT)