Types, terms and proofs in categorical attributed graph transformation

Bertrand Boisvert, Louis Féraud, Sergei Soloviev[1]

IRIT - Université de Toulouse - ACADIE/MACAO

SPIIRAN, St.-Petersburg, Decemeber 9, 2011

[1]Part of this research has been supported by the Climt project, ANR-11-BS02-016-02

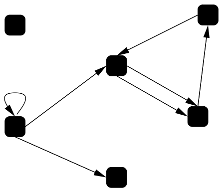
Plan

1. Attributed graphs
2. Categorical graph rewriting
3. Our approach
4. Examples
5. From terms to proofs

Attributed graphs

Attributed graph =

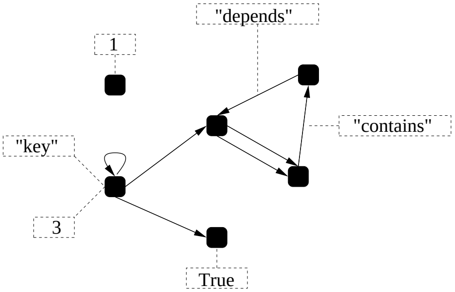
- Structural part



Attributed graphs

Attributed graph =

- Structural part
- Attributes



What we would like to do with graphs:

- Build graphs (example: by using graph grammars)
- Study properties of graphs (examples: presence of cycles?)
- Transform graphs (example: by using graph grammars)
- Study properties of graph transformations (example: does a transformation preserve connexity?)

What we would like to do with graphs:

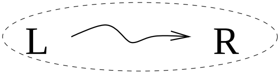
- Build graphs (example: by using graph grammars)
- Study properties of graphs (examples: presence of cycles?)
- Transform graphs (example: by using graph grammars)
- Study properties of graph transformations (example: does a transformation preserve connexity?)

Graph rewriting systems

- G: host graph

G

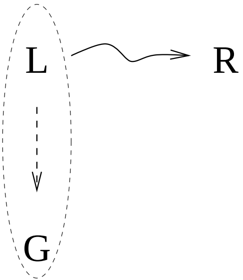
Graph rewriting systems



G

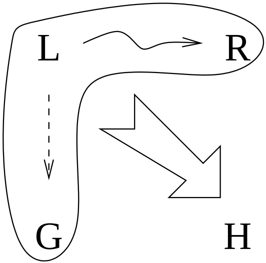
- G: host graph
- $L \rightsquigarrow R$: transformation rule
  - L: Pattern to modify
  - $\rightsquigarrow R$: transformation instructions

Graph rewriting systems

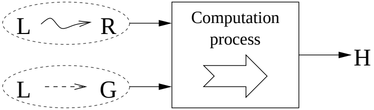


- G: host graph
- $L \rightsquigarrow R$: transformation rule
  - L: Pattern to modify
  - $\rightsquigarrow R$: transformation instructions
- $L \dashrightarrow G$: matching

Graph rewriting systems

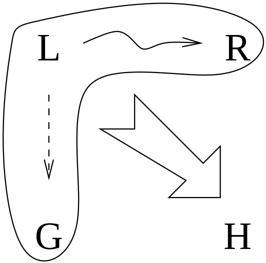


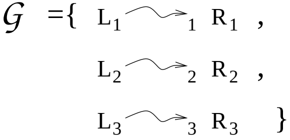
- G: host graph
- $L \rightsquigarrow R$: transformation rule
  - L: Pattern to modify
  - $\rightsquigarrow R$: transformation instructions
- $L \dashrightarrow G$: matching
- $\Rightarrow$: computation process



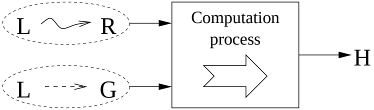
- H: result graph

Graph rewriting systems





- G: host graph
- $L \rightsquigarrow R$: transformation rule
  - L: Pattern to modify
  - $\rightsquigarrow R$: transformation instructions
- $L \dashrightarrow G$: matching
- $\Rightarrow$: computation process



- H: result graph
- a graph grammar

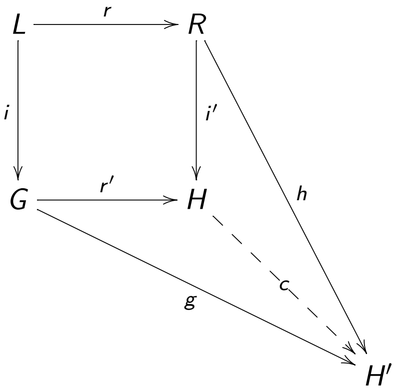
Existing graph rewriting systems

- Node remplacement approaches
  - NLC
  - NCE
  - edNCE
  - ...
- Edge remplacement approaches
  - ...
- Categorical approaches
  - Double Pushout
  - Simple Pushout
  - ...

Plan

Why do we use category theory?

- why not?
- formal and abstract language
- some categorical constructions represent gluing and deletion (Pushout)



How to create a categorical graph rewriting system?

1/ Define a category

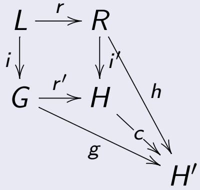
- Objects: attributed graphs
- Arrows: attributed graph morphisms

2/ Define graph Transformation rules

- described by one or more attributed graph morphisms

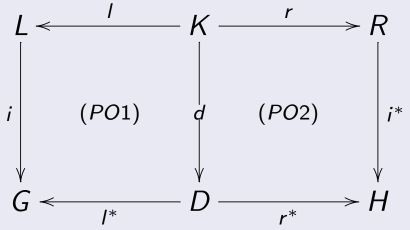
3/ Describe how to do the computation of a rule application

- by computation of canonical constructions (Pushouts, ...)



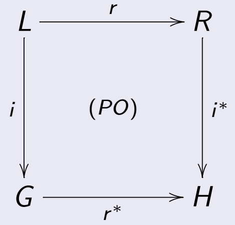
Two main approaches: Double pushout & Single pushout

Double Pushout



- total morphisms
- pushout complement & pushout
- application conditions

Single pushout



- partial morphisms
- one pushout
- application conditions not necessary

Categorical attributed graph rewriting systems

Classical approaches

- representation of attributes with Σ-algebras
- same representation for structural part and attribute part

Limitations of approaches based on Σ-algebras

- no functional attributes
- combinatorial explosion for non trivial computations
- difficulties for implementation

Plan

Our goal

Pragmatic approach

- reuse the well developed SPo approach on structural part
- improve attribute part
- use different theoretical frameworks for structure and attributes
- unify the two parts in category theory

Typed λ-calculus with inductive types

λ-calculus

- simply typed λ-calculus
- with inductive types
- pairing
- terminal object

Inductive types examples

- $Nat = Ind\alpha\{0 : \alpha,$
  $Succ : \alpha \to \alpha\}$
- $T_2 = Ind\alpha\{Leaf : \alpha,$
  $Node : \alpha \to \alpha \to \alpha\}$
- $T_\omega = Ind\alpha\{Leaf : \alpha,$
  $Succ_\omega : \alpha \to \alpha,$
  $Lim : (Nat \to \alpha) \to \alpha\}$

Inductive types

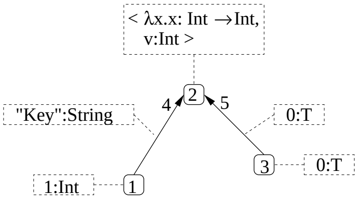
Benefits of using inductive types

- more expressive than Σ-algebras
- recursion operators
- good reduction properties:
  - strong normalization
  - local confluence

1/ Define a category: objects = finite attributed graphs

Struture of a graph G

- finite sets of vertices and edges
- source and target functions
- total order on vertices $\cup$ edges



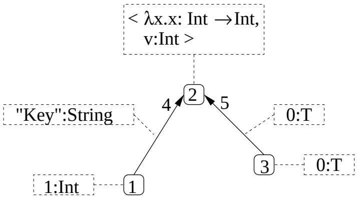
1/ Define a category: objects = finite attributed graphs

Struture of a graph G

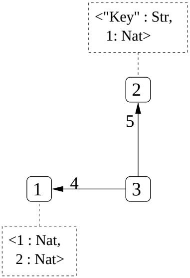
- finite sets of vertices and edges
- source and target functions
- total order on vertices ∪ edges

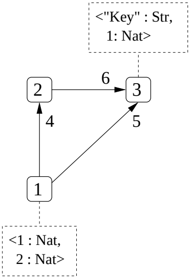
Attributes of a graph G

- one typed λ-term for each element

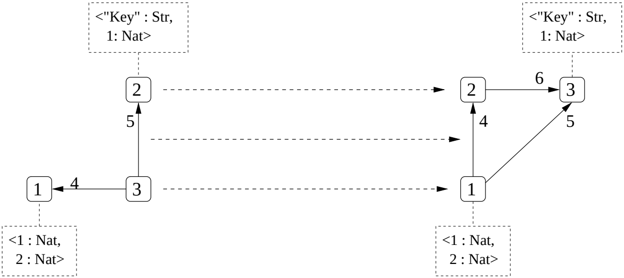


1/ Define a category: arrows = attributed graph morphisms



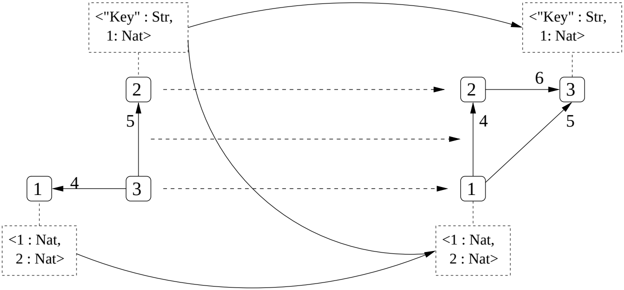


1/ Define a category: arrows = attributed graph morphisms



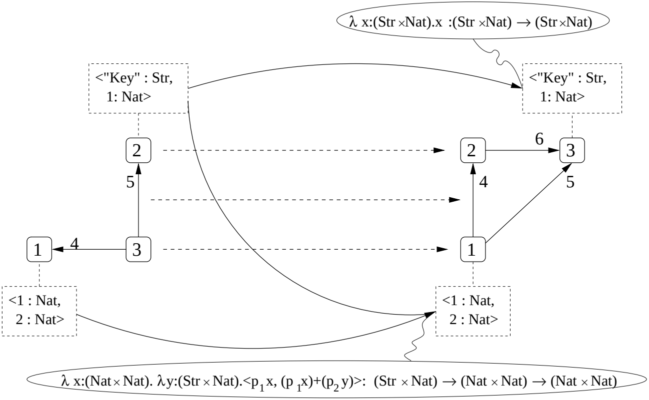
- partial graph homomorphism;

1/ Define a category: arrows = attributed graph morphisms



- partial graph homomorphism;
- attribute dependency relation;

1/ Define a category: arrows = attributed graph morphisms



- partial graph homomorphism;
- attribute dependency relation;
- λ-terms defining computations on attributes;

2/ Define graph transformation rules

Transformation rules

- transformation rule given by one morphism $L \xrightarrow{r} R$
- embedding given by one morphism $L \xrightarrow{i} G$

3/ Describe how to do the computation of a rule application

Theorem

Weak pushouts exist in the category Gr^T

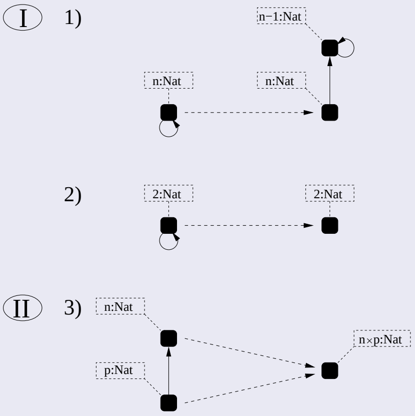
Construction

Straightforward

Plan

Computation of n!

Σ-algebras

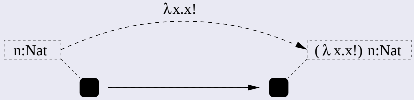


λ-terms

$$add = \lambda x.Rec^{Nat \rightarrow Nat}(x)(\lambda u \lambda v.succ(v))$$

$$mult = \lambda x.Rec^{Nat \rightarrow Nat}(0)(\lambda u \lambda v.(Add\ x\ v))$$

$$x! = Rec^{Nat \rightarrow Nat}(succ(0))(\lambda u \lambda v.(Mult\ u\ v))x$$

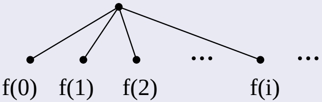


Managing infinity with functional attributes

ω-Trees

$$T_\omega = Ind\alpha\{Leaf : \alpha,\quad Succ_\omega : \alpha \to \alpha,\quad Lim : (Nat \to \alpha) \to \alpha\}$$

Example: one ω-tree defined by Lim(f)



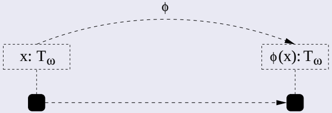
Managing infinity with functional attributes

Rule to select pair branches

$$d = Rec^{Nat \to Nat}(0)(\lambda x.\lambda y.Succ(Succ(y)))$$

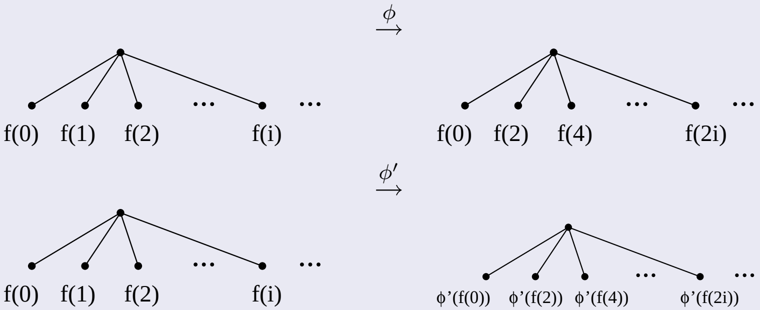
$$\phi = Rec^{T_\omega \to T_\omega}(Leaf)(\lambda x^{T_\omega}.Succ_\omega)(\lambda u.\lambda v.(ud))$$

$$\phi' = Rec^{T_\omega \to T_\omega}(Leaf)(\lambda x^{T_\omega}.Succ_\omega)(\lambda u.\lambda v.(vd))$$



Managing infinity with functional attributes

Computation on an attribute: selecting pair branches



Information balance between attributes and structure

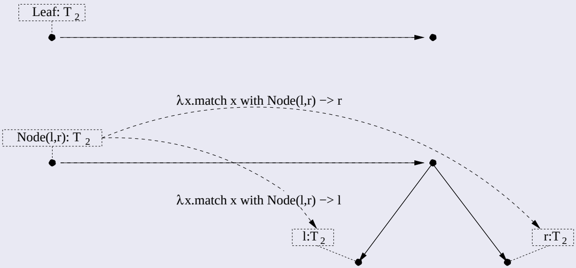
Inductive type for binary trees

$$T_2 = Ind\alpha\{Leaf : \alpha, \\ Node : \alpha \to \alpha \to \alpha\}$$

Graph grammar

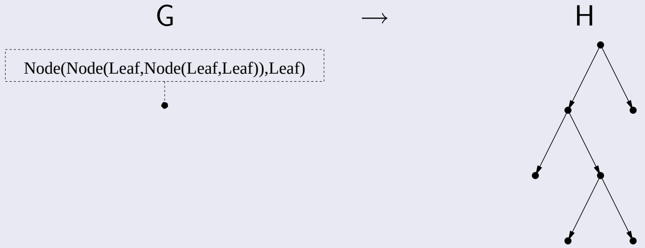
Rule 1:

Rule 2:



Information balance between attributes and structure

Example: application of R2;R2;R2;R1;R1;R1;R1 on G



Differences between our approach and other approaches

Structural part

- same that single pushout

Attribute part

- more complex attributes (functional attributes)
- more complex computation functions
- better expressivity
- guaranteed strong normalization and confluence
- flexibility

Plan

More on attribute part

Our framework will hold if:

- instead of using λ-terms only (as computation functions)
- we shall use proof-schemas
- and combine computation and proof

More on attribute part

A partial proof is a tree with the following properties:

1. Each node is labelled with a sequent and the rule of inference which is applied to this sequent (backwards) to produce the node's children (and the children of course must be the premises of this rule).
2. The final sequent (or the goal) is the sequent at the root of the three.
3. If the leaf is labelled by an axiom, then it is called *complete*.
4. If no rule of inference is specified for a leaf, then the leaf is *open*.
5. A proof is a partial proof with no open leaves.

More on attribute part

The notion of proof-schema is obtained when we permit to use **meta-level sequents** instead of sequents in partial proofs.

Definition

A meta-level sequent is an abstraction of an object-level sequent which may contain meta-variables.

Remarque

Not all elements of a meta-level sequent need to be metavariables, There may be metavariables of different kinds, e.g., for terms, contexts (lists of typed variables), even for variables (as an the axiom schema above).

Definition

1. Each node is labelled with a meta-level sequent and the rule of inference which is applied to this sequent (backwards) to produce the node's children (and the children of course must be the meta-level sequents matching the premises of this rule).
2. The final meta-level sequent (or the goal) is the meta-level sequent at the root of the three.
3. If the leaf is labelled by an axiom schema, then it is called *complete*.
4. If no rule of inference is specified for a leaf, then the leaf is *open*.
5. A proof schema is a partial proof schema with no open leaves.

Example

Proof-schema in predicate calculus:

$$\dfrac{\dfrac{*}{\Gamma \vdash \exists y.A} \qquad \dfrac{\dfrac{\dfrac{*}{\Gamma, [t/y]A, \Delta \vdash B}}{\Gamma, [t/y]A, \Delta \vdash \forall x.B}(\vdash \forall)}{\Gamma, \exists y.A, \Delta \vdash \forall x.B}(\exists \vdash)}{\Gamma, \Delta \vdash \forall x.B}(cut)$$

Example

Proof-schema in simply typed λ-calculus:

$$\dfrac{\dfrac{*}{\Gamma \vdash s : A} \quad \dfrac{\dfrac{*}{\Gamma, x : A \vdash t : B}}{\Gamma \vdash \lambda x : A.t : A \to B}(abstr)}{\Gamma \vdash (x : A.t)s : B}(app)$$

- Now, instead of taking lambda-terms as attributes we may take judgements (sequents).
- They may include lambda-terms.
- Instead of computation functions, we may take proof-schemes.

More examples

Permutation of rules (Kleene-style). Let us consider two rules in propositional calculus:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}(\to -right)$$

and

$$\frac{\Gamma_1 \vdash C \quad \Gamma_1, D, \Gamma_2 \vdash E}{\Gamma_1, C \to D, \Gamma_2 \vdash E}(\to -left).$$

Let us consider first the schema where $(\to -right)$ is applied first. It is to notice that we have to consider the premise of $(\to -right)$ more “finely structured” than in case when each rule schema is taken separately:

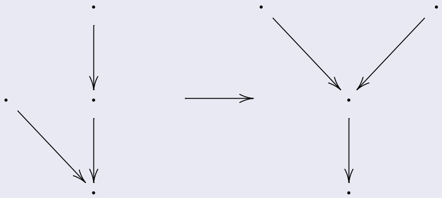
More examples

$$\dfrac{\dfrac{}{\Gamma_1 \vdash C} \quad \dfrac{\Gamma_1, D, \Gamma_2, A \vdash B}{\Gamma_1, D, \Gamma_2 \vdash A \to B}}{\Gamma_1, C \to D, \Gamma_2 \vdash A \to B}.$$

Permutation of two inferences gives:

$$\dfrac{\dfrac{\Gamma_1 \vdash C \quad \Gamma_1, D, \Gamma_2, A \vdash B}{\Gamma_1, C \to D, \Gamma_2, A \vdash B}}{\Gamma_1, C \to D, \Gamma_2 \vdash A \to B}.$$

On the level of graph structure (with sequents as attributes) this may be seen as a transformation



"Distant links" in derivations.

Let us consider (for simplicity) the derivation d of the following form:

$$\cfrac{\cfrac{\Gamma, [t/y]A, \Delta \vdash B}{\Gamma, \exists y A, \Delta \vdash B}}{\cfrac{\ldots}{\Gamma', \exists A, \Delta' \vdash B'}}$$

"Distant links" in derivations

If we keep a "long distance" link in the derivation, we may formalize the rule that permits to return to $[t/y]A$ from $\exists yA$ in one step.

Future work

Study classical Properties

- local confluence
- parallelism
- critical pairs

Implementation

- DPoPb implementation in haskell language
- implementation of our new approach

Representation of proofs

- proof schemes as attributes and computation functions

Questions

Questions?