Graphs and maps: rewriting techniques at work*

A GRAPHICAL TREATMENT OF DYADIC

FIRST ORDER SPECIFICATIONS

Andrea Formisano
Dipartimento di Matematica e Informatica, Università di Perugia
formis@dipmat.unipg.it

Marta Simeoni
Fachbereich 13 Informatik, Technische Universität Berlin
and
Dipartimento di Informatica, Università *Ca' Foscari* di Venezia
simeoni@dsi.unive.it

Introduction

Map reasoning is concerned with relations over an unspecified domain of discourse. Two limitations w.r.t. first-order reasoning are: only dyadic relations are taken into account; all map formulas are equations, having the same expressive power as first-order sentences in three variables. The map formalism inherits from the Peirce-Schröder tradition, through contributions of Tarski [45] and many others.

Direct algebraic manipulation of map expressions seems to be, for human beings, much less natural than developing inferences in first-order logic; it may in fact appear to be overly machine-oriented for direct hand-based exploitation.

The situation radically changes when one resorts to a convenient representation of map expressions based on labeled graphs. In this manner it is possible to abstracts w.r.t. inessential features of expressions. An approach of this kind have been proposed in [10, 8]. By exploiting AGG —an automated tool for graph transformation developed at TU Berlin— in this paper we implement in a graphical fashion, some conservative methods for performing the translation of first-order formulas into our equational framework. Then, we move the first step toward the implementation of an automated tool for the mechanization of graphical map-reasoning.

As mentioned, the kernel of our graphical inference-system is AGG. It has a formal foundation based on the single-pushout approach to graph transformation. This approach is briefly reviewed in the first section. In Sec.2 we describe the equational specification language of map calculus and provide a graphical representation for both expressions and equalities in terms of the features offered by AGG. Sections 3 and 4, describe, mainly by means of worked examples, the implementation of the translation techniques and of the inferenche machinery on top of AGG.

*This research was partially funded by the Italian CNR (coordinated project log(SETA)—unit of IASI, Rome); by MURST (PGR-2000—Automazione del ragionamento in teorie insiemistiche); by the EC TMR Network GETGRATS (GEneral Theory of GRAph Transformation Systems); and by Esprit Working Group APPLIGRAPH.

1 Graph transformation: the Single Pushout approach

The framework of graph transformation combines two basic concepts: graphs and rewriting rules. Graphs represent in a natural, intuitive and visual way objects and relationships between objects, while transformation or rewriting rules are useful wherever permitted actions and transitions must be described. In this paper, we consider the so-called Single Pushout (SPO) approach on graph transformation as introduced in [34]. We review here the basic concepts, and refer to [39] for a complete treatment.

The kind of graphs that we use are directed (multi-)graphs. A graph is given by a set of nodes, a set of edges and two functions assigning to each edge its source and target node, respectively. Moreover, if nodes and edges labels are considered, the definition of graph comprises two labeling functions, associating a label to each node and edge. More formally:

Definition 1 *Given two fixed alphabets $\mathcal{A}_E$, $\mathcal{A}_N$ for node and edge labels, respectively, a* (LABELED) GRAPH *(over ($\mathcal{A}_E$, $\mathcal{A}_N$)) is a tuple $G = \langle G_E, G_N, G_s, G_t, G_{le}, G_{ln} \rangle$ where G_E is a set of* EDGES, *G_N is a set of* NODES *$G_s, G_t : G_E \to G_N$ are the* SOURCE *and* TARGET *functions, and $G_{le} : G_E \to \mathcal{A}_E$, $G_{ln} : G_N \to \mathcal{A}_N$ are the* NODE *and the* EDGE LABELING FUNCTIONS.

Example 1 *Fig.1 shows a labeled graph. In this case labels are given in a visual fashion, by using square and rounded boxes for nodes, and solid and dashed arrows for edges, i.e: $\mathcal{A}_N = \{ ▢ , \square \}$ and $\mathcal{A}_E =_{Def} \{ \longrightarrow, \dashrightarrow \}$. Node and edge identifiers are not drawn.*

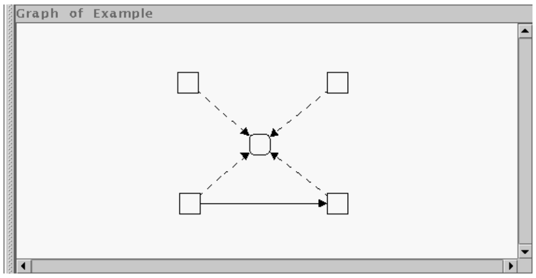


Fig. 1: A graph

A (partial) graph morphism is given by two (partial) mappings relating nodes and edges, respectively, which preserve the graph structure and labeling.

Definition 2 *Let $G = \langle G_E, G_N, G_s, G_t, G_{le}, G_{ln} \rangle$ and $H = \langle H_E, H_N, H_s, H_t, H_{le}, H_{ln} \rangle$ be graphs over $\mathcal{A}_E$, $\mathcal{A}_N$. A* (PARTIAL) GRAPH MORPHISM *$k = (k_E, k_N) : G \to H$ is a pair of (partial) functions $k_E : G_E \to H_E$ and $k_N : G_N \to H_N$ such that:*

$$H_s \circ k_E = k_N \circ G_s, \quad H_t \circ k_E = k_N \circ G_t, \quad H_{le} \circ k_E = G_{le}, \quad H_{ln} \circ k_N = G_{ln}.$$

Graphs over fixed labeling alphabets and partial graph morphisms define a category, called $\mathbf{Graph^P}$, which is closed under pushouts. This closure property is the basis for the definition of a derivation step (see definition 5 below).

A production in the SPO approach is a partial graph morphism $p : L \to R$ where L and R are the left and right hand–side graphs, respectively. Objects (nodes and edges) where p is undefined are deleted while objects preserved by p form the application context. The objects in R which are not in the preimage of p are the newly created objects. Thus, L expresses the objects to be deleted, R expresses the objects to be created and both of them express the context to be left unchanged.

Definition 3 *A* GRAPH PRODUCTION *$p : L \to R$ is a partial graph morphism between a left-hand side graph G and a right-hand side graph R.*

Example 2 *Consider the same alphabets of Example 1. Fig.2 shows a graph production which transform the left-hand side graph by adding a new solid edge and deleting the two dashed edges and the rounded box node.*

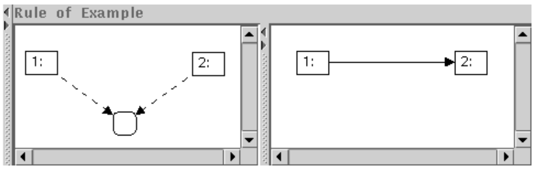


Fig. 2: A graph production

The partial morphism between the left- and the right-hand side graphs is visualized by associating the same number to objects which are related (see the numbers inside the two square nodes of figure Fig.2).

A graph grammar consists then of a start graph and a set of productions, describing the allowed transformation steps. Formally:

Definition 4 *A graph grammar $\mathcal{G}$ is a pair $\mathcal{G} = (G_0, P)$ where G_0 is the start graph and P is a set of graph productions.*

A production $p : L \to R$ can be applied to a given graph G if G contains all nodes and edges which are present in L. To ensure this, the notion of a match as a total graph morphism matching L with (a part of) G is introduced: totality of the match morphism ensures that all objects in L have a corresponding image in G. The actual transformation of G is performed by removing the part matched by L and then adding R.

Definition 5 *A* MATCH *for a production $p : L \to R$ in some graph G is a total morphism $m : L \to G$. A* DIRECT DERIVATION *from G to a graph H via p and m is denoted with $G \overset{p/m}{\Rightarrow} H$, and the resulting graph H is the pushout object of the following diagram in* $\mathbf{Graph^P}$.

$$\begin{array}{ccc} L & \xrightarrow{p} & R \\ {\scriptstyle m}\downarrow & (po) & \downarrow \\ G & \longrightarrow & H \end{array}$$

Note that the match morphism m is not required, in general, to be injective. Note moreover that in this approach the deletion of a node causes the implicit deletion of all its incident edges, the so-called *dangling edges*, even if they belong to the context which should normally be preserved.

The single pushout construction ensures that any effect specified by the rule is actually performed in the concrete derivation. Moreover, nothing more is done, than what is specified in the rule (up to the deletion of the possible dangling edges).

Example 3 *Consider again Examples 1 and 2. The result of applying the production of Fig.2 to the graph of Fig.1 is the graph represented in Fig.3.*

Note that, as a side effect of the deletion of the rounded box node, the two dashed edges on the bottom of the original graph have been deleted too (implicit deletion of dangling edges).

A sequence of direct derivations $G_0 \overset{p_1/m_1}{\Rightarrow} \ldots \overset{p_k/m_k}{\Rightarrow} G_k$ form a *derivation* from G_0 to G_k denoted by $G_0 \Rightarrow^* G_k$. The graph language generated by a graph grammar $\mathcal{G}$ is the set $L(G)$ of all graphs G_k such that there exists a derivation $G_0 \Rightarrow^* G_k$ using productions of $\mathcal{G}$.

The left-hand side of a production permits the expression a condition on the existence of nodes and edges in a given graph, in order to apply the rule, i.e. it expresses a *positive application*

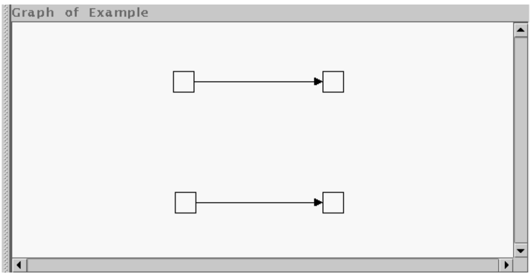


Fig. 3: The result graph of a direct derivation step

condition. With the SPO approach, it is also possible to require the non existence of some object in the given graphs, i.e to express *negative application conditions*.

Negative application conditions for a production $p : L \to R$ are specified via graph morphisms called *constraints*. More precisely, a constraint for L is a graph morphism $l : L \to \overline{L}$ and the elements in $\overline{L} - L$ represent the forbidden structure: a match $m : L \to G$ satisfies the constraint l, if the forbidden elements are not present in the context of the match. Hence, p can be applied to the given graph G if m satisfies all the constraints associated with L, i.e all the negative application conditions.

The SPO approach can be applied not only to simple labeled graphs, but also to attributed graphs. In this case, attributes are specified using algebraic specifications (see [16]). Thus, informally, an attributed graph consists of a (labeled) graph, an attribute algebra, and some attribute functions to connect the graphical and algebraic parts. An attributed graph morphism consists of a partial graph morphisms (to relate the graphical parts) and a total algebra morphism (to relate the algebraic parts) such that the graphical and algebraic parts are compatible.

A production for attributed graphs is given by an attributed graph morphism, again between a left-hand side graph L and a right-hand side graph R. A match morphism from L to an attributed graph G has to combine both the graphical and the algebraic aspects, i.e an assignment for the attributes compatible with the matching of the graphical part has to be found.

1.1 AGG: a tool for graph rewriting

The Attributed Graph Grammar system (AGG) provides a visual programming environment which implements the theoretical concepts of the SPO framework. In this section we sketch its basic characteristics and features; see [19] for a detailed description.

Graph transformation based applications are described by graph grammars: the AGG environment supports the visual handling of both the start graph and the productions. The start graph, as well as the productions may be attributed by Java objects (i.e. instances of Java classes, either imported from standard libraries or user defined) and expressions, which are evaluated during rule application. Notice that the attribution mechanism of AGG deviates from its theoretical counterpart; in particular, it is not realized in terms of explicit algebraic specifications, but by exploiting to the Java classes and methods.

The user interacts with the AGG environment through a graphical user interface which provides several visual editors, and an interpreter: the graph-transformation machine. Fig.4 shows the main window of the AGG user-interface. On the left, the current graph grammar is visualized: the user can browse and inspect the rules and the start graph. The selected graph or rule is shown in the corresponding graphical editor, on the right: the upper editor displays the left- and right-hand sides of the rule, while the lower one displays the graph. A special attribute editor pops up whenever an object is selected for attribution.

AGG allows the association of visual labels (or "types") to nodes and edges: shape, color and a textual description can be defined and the actual choice is displayed just below the menu bar (see

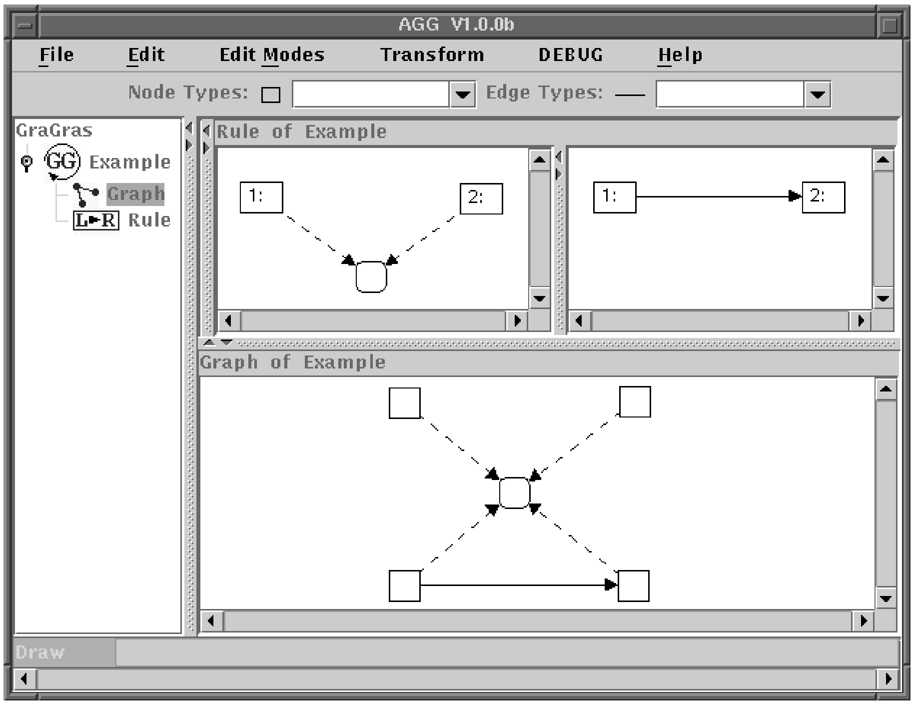


Fig. 4: The AGG user interface

Fig.4).

Once a graph grammar has been formalized, AGG allows one to apply the graph transformation rules by providing two basic mechanisms: a) the user can select and apply one-by-one the rules; or b) AGG itself can perform automatically a complete run of transformations. In the latter case, the rules are selected following their order in the grammar; each single rule is repeatedly applied, to the current graph, as much as possible before the next one is taken into consideration.

Finally, we would like to remark that AGG allows the setting of two main parameters regarding the rewriting technique to be applied. The first one concerns the match morphisms: it is possible to allow only the use of injective morhpisms, instead of general morphisms. The second one concerns the implicit deletion of dangling edges: it is possible to allow or to forbid the application of rules which cause (the implicit deletion of) dangling edges. The application we are going to describe in the next sections uses always injective matching morphisms, and dangling edges are forbidden.

2 A purely equational specification language

In what follows we introduce the basic concepts regarding *map calculus*, the arithmetic of dyadic relations, by slightly adapting notions borrowed from [45].

Map calculus is a ground equational formalism where one can state (and as we will see, derive) properties of dyadic relations over a fixed domain of discourse.[1] It can be seen as an evolution of the theory of relations, an algebraic approach to logic developed around the end of the nineteenth century by, among others, A. de Morgan, C. S. Peirce and E. Schröder [42, 35, 36]. More recently, the publication of [45] gave rise to new interest in map calculus; in parallel, a big amount of effort is being devoted to the contiguous field of relation algebras (see, e.g., [40, 41, 38, 14, 23]).

[1]Our usage of the word 'map' to mean dyadic relation (not necessarily a function!) complies with an old usage of this word still witnessed in some of today's literature (cf., e.g., [43]).

2.1 $\mathcal{L}^\times$ and $\mathcal{L}^+$: syntax and semantics

Here we give a characterization of the two languages, $\mathcal{L}^\times$ and $\mathcal{L}^+$, introduced in [45]. As we mentioned, map calculus is a fully-equational formalism based on an underlying language $\mathcal{L}^\times$, which is a ground equational language. In this formalism one can state properties of dyadic relations over an unspecified, yet fixed, domain $\mathcal{U}$ of discourse.

The basic ingredients of this language are:

- three *constants*: $\emptyset$, $\mathbb{1}$, ι;
- infinitely many *map letters*: $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \ldots$;
- dyadic constructs $\cap$, $\triangle$, **;** of map *intersection*, map *symmetric difference*, and map *composition*;
- the monadic construct $^\smile$ of map *conversion*.

Map expressions are obtained through repeated use of $\cap$, $\triangle$, **;**, and $^\smile$, starting from the map letters p_i, which can be freely interpreted as dyadic relations over $\mathcal{U}$, and from the mentioned constants.

More formally:

Definition 6 *A* MAP EXPRESSION *is any term of the following signature:*

symbol :	$\emptyset$	$\mathbb{1}$	ι	p_i	$\cap$	$\triangle$	**;**	$^\smile$	$\overline{}$	$-$	$\cup$
degree :	0	0	0	0	2	2	2	1	1	2	2
priority :					5	3	6	7		2	2

Of these, $\cap, \triangle,$ **;**$, -, \cup$ will be used as left-associative infix operators, $^\smile$ as a postfix operator, and $\overline{}$ as a line topping its argument.

Definition 7 *A* MAP EQUALITY *is a writing of the form $Q{=}R$, where Q and R are map expressions.*

Notice that our treatment slightly deviates from the one given in [45], in that

- we are adopting $\cap$, $\triangle$, $\emptyset$, and $\mathbb{1}$, instead of $\cup$ and $\overline{}$, as primitive Boolean constructs;
- we have an infinite supply of map letters available from the outset, whereas [45] starts with only one map letter.

Once a nonempty domain $\mathcal{U}$ has been fixed, the map constants $\emptyset$, $\mathbb{1}$, and ι are interpreted by putting: [2]

$$\emptyset^{\Im} =_{\mathrm{Def}} \emptyset, \qquad \mathbb{1}^{\Im} =_{\mathrm{Def}} \mathcal{U}^2 =_{\mathrm{Def}} \mathcal{U} \times \mathcal{U}, \qquad \iota^{\Im} =_{\mathrm{Def}} \{[a,a] : a \text{ in } \mathcal{U}\}.$$

Then, subsets $\mathsf{p}_1^{\Im}, \mathsf{p}_2^{\Im}, \mathsf{p}_3^{\Im}, \ldots$ of $\mathcal{U}^2$ are put in correspondence with the p_is, each map expression P comes to designate a specific map $P^{\Im}$, on the basis of the following evaluation rules:

$$\begin{aligned}
(Q\cap R)^{\Im} &=_{\mathrm{Def}} \{[a,b] \in Q^{\Im} : [a,b] \in R^{\Im}\};\\
(Q\triangle R)^{\Im} &=_{\mathrm{Def}} \{[a,b] \in \mathcal{U}^2 : [a,b] \in Q^{\Im} \text{ if and only if } [a,b] \notin R^{\Im}\};\\
(Q;R)^{\Im} &=_{\mathrm{Def}} \{[a,b] \in \mathcal{U}^2 : \text{there are } c\text{s in } \mathcal{U} \text{ for which } [a,c] \in Q^{\Im} \text{ and } [c,b] \in R^{\Im}\};\\
(Q^\smile)^{\Im} &=_{\mathrm{Def}} \{[b,a] : [a,b] \in Q^{\Im}\}.
\end{aligned}$$

[2] Our choice to restrict the treatment to homogeneous relations, i.e. we deal with only one universe $\mathcal{U}$, is much closer to the spirit of [9, 21, 22], where the ultimate goal consists in realizing an inference framework for an equational re-formulation of Zermelo-Skolem-Fraenkel Set Theory (cf. [32, 29]). In this context, the inference machinery for map calculus is ultimately tailored to support set reasoning, and since we are interested in modeling a universe of sets involving objects of only one sort, a single domain suffices.

Accordingly, an equality $Q{=}R$ turns out to be either true or false in each interpretation $\Im$.

The language $\mathcal{L}^+$ is a variant version of a first-order dyadic predicate language: an *atomic formula* (briefly, an *atom*) of $\mathcal{L}^+$ has either the form xQy or the form $Q{=}R$, where x, y stand for individual variables (ranging over $\mathcal{U}$) and Q, R stand for map expressions of $\mathcal{L}^\times$.

Here propositional connectives and existential/universal quantifiers are employed as usual. An ordering $\mathsf{v}_1, \mathsf{v}_2, \ldots$ of all individual variables is assumed.

To enrich $\mathcal{L}^\times$ and $\mathcal{L}^+$ we can introduce the following definition for derived constructs:

$P \cup Q \ \equiv_{\mathrm{Def}}\ P \bigtriangleup Q \bigtriangleup P \cap Q$	$P - Q \ \equiv_{\mathrm{Def}}\ P \bigtriangleup P \cap Q$
$\not P \ \equiv_{\mathrm{Def}}\ \overline{P} \ \equiv_{\mathrm{Def}}\ P \bigtriangleup \mathbb{1}$	$P \subseteq Q \ \equiv_{\mathrm{Def}}\ P \cap Q = P$
$\mathsf{funcPart}(P) \equiv_{\mathrm{Def}} P \cap \overline{P ; \overline{\iota}}$	$\mathsf{tot}(P) \equiv_{\mathrm{Def}} P \bigtriangleup \overline{P \,;\, \mathbb{1}}$
$\mathsf{Func}(P) \ \equiv_{\mathrm{Def}}\ \mathsf{funcPart}(P) = P$	$\mathsf{Total}(P) \ \equiv_{\mathrm{Def}}\ P;\mathbb{1}{=}\mathbb{1}$

The interpretation of $\mathcal{L}^\times$ and $\mathcal{L}^+$ obviously extends to the new constructs; e.g.,

$$\mathsf{funcPart}(P)^{\Im} \ =_{\mathrm{Def}}\ \{[a,b] \in P^{\Im} : [a,c] \notin P^{\Im} \text{ for any } c \neq b\},$$

so that $\mathsf{funcPart}(\cdot)$ renders the 'functional aspect' of a predicate: $\mathsf{funcPart}(P){=}P$ will mean 'P is a partial function'.

Several axiomatizations of map calculus are available (cf., e.g., [11] and [35, 41]); one possible choice is depicted in Fig.5. Notice that the last law in Fig.5, i.e. $P^{\smile} ; (R - P;Q) \cap Q = \emptyset$, is closely related to the law $P^{\smile} ; \overline{P;Q} \cup \overline{Q} = \overline{Q}$ known from the work of Peirce and Schröder (see, for instance, [37, 45]). Our adoption of a different set of primitive map-constructs motivated the introduction of this law.

The equalities in the fixed initial endowment of axioms, describing the full variety of dyadic relations over a generic domain $\mathcal{U}$ of discourse, are called *logical* axioms. Further axioms, added to the logical ones, lead to algebraic characterizations of specific domains and data structures: at varying degrees of mathematical abstraction, one has general classes and sets, hereditarily finite sets, trees, nested or flat lists, lines subjected to editing, etc.—cf. [1, 22]. These, which are called *proper* axioms, state the properties of context-specific maps, e.g., `car` and `cdr` in the case of lists, and membership, $\in$, in the case of sets. Typically, proper axioms are *ground*, i.e. devoid of the map variables that occur in the logical axioms;[3] usually they involve, in addition to the standard constructs of map algebra, the symbol(s) characteristic of the application, e.g. $\in$.

It is undoubted that map calculus is, at least at first sight, less attractive and less 'user-oriented' than first-order logic. This motivates not only the research effort in implementing automated systems for map reasoning, but also in studying the possibilities of automating the translation of sentences, or of entire theories, from the map formalism into first-order predicate logic, and *vice-versa*. These tasks constitute the main themes of the following exposition. In particular, we begin by introducing a graphical representation for map expressions and map equalities, by adopting the framework described in Sections 1; secondly, in the following section, we introduce a basic technique aimed at translating first-order formulas into the map formalism, together with a series of possible conservative refinements. Moreover, we will show how prototypes of all these algorithms can be straightforwardly implemented by exploiting AGG. Finally, in Sec.4 it is described a graphical approach to automated map-reasoning which exploits the features offered by AGG. Several examples are also offered.

[3] A non-ground axiom could be the algebraic rendering of what would be an axiom *scheme* in a specification based on first-order predicate logic.

$$\Lambda^{\times}$$

$$\begin{array}{rcl}
P \triangle P & = & \emptyset \\
P \triangle (Q \triangle P) & = & Q \\
R \cap Q \triangle R \cap P & = & (P \triangle Q) \cap R \\
P \cap P & = & P \\
(P \star_1 Q) \star_1 R & = & P \star_1 (Q \star_1 R) \\
\iota ; P & = & P \\
P^{\smile\smile} & = & P \\
(P \star_2 Q)^{\smile} & = & Q^{\smile} \star_2 P^{\smile} \\
(P \cup Q) ; R & = & Q ; R \cup P ; R \\
\mathbb{1} \cap P & = & P \\
P^{\smile} ; \big(R \cap (P ; Q \triangle R) \big) \cap Q & = & \emptyset
\end{array}$$

$\star_1 \in \{\triangle, \cap, ;\}$ and $\star_2 \in \{\cap, ;\}$

Fig. 5: Logical axioms for map algebra.

2.2 A graphical rendering for map specifications

It is at times useful (cf. [7]) to represent a map expression P or an identity $P = \mathbb{1}$ or, more generally, an existentially quantified conjunction φ of literals, by a (directed) graph whose edges are attributed by map expressions. By recalling the definitions introduced in Sec.1, we can describe such a graphic representation as follows.

Let us assume that φ is a conjunction $(\exists x_1) \cdots (\exists x_k)(L_1 \wedge \cdots \wedge L_n)$ of atoms, and that $vars(\varphi) \subseteq \{\mathsf{v}_1, \mathsf{v}_2, \ldots\}$ is the set of all the variables occurring in φ. Moreover, assume that $x_1, \ldots, x_k \in vars(\varphi)$ and that each L_i is of the form $x_{L_i} P_{L_i} y_{L_i}$, where $x_{L_i}, y_{L_i} \in vars(\varphi)$ are variables (assumed ranging over the domain $\mathcal{U}$ of discourse), and P_{L_i} is a map expression of $\mathcal{L}^{\times}$ (Equality atoms $Q = R$ have been rewritten already in the form $x \overline{\mathbb{1}; (Q \triangle R); \mathbb{1}} y$, negative literals in the form $x \overline{Q} y$; clearly, free variables may occur in φ intermixed with existentially quantified variables.) A graph G_{φ} representing φ is so defined: given the two alphabets

$$\mathcal{A}_N =_{\text{Def}} \{ \bigcirc \}$$

and

$$\mathcal{A}_E =_{\text{Def}} \{ \longrightarrow , \dashrightarrow \},$$

for nodes and edges (graphical) labels, we have $G_{\varphi} = \langle G_E, G_N, G_s, G_t, G_{le}, G_{ln} \rangle$ where

$G_E =_{\text{Def}} \{1, \ldots, n\}$,
$G_N =_{\text{Def}} vars(\varphi)$,
$G_s : G_E \to G_N$ maps each $i \in G_E$ into x_{L_i},
$G_t : G_E \to G_N$ maps each $i \in G_E$ into y_{L_i},
$G_{ln} : G_N \to \mathcal{A}_N$ maps each node to the unique available label,
$G_{le} : G_E \to \mathcal{A}_E$, maps each edge to $\longrightarrow$.

Note that, at the moment, we are using just one kind of node and one kind of arrow, i.e. we are actually using unlabeled graphs. Later on we will introduce the dotted arrow to express map complementation.

Our graphical representation exploits attribution in order to permit suitable manipulation of map expressions. In what seen so far, we described how to represent a conjunction of literals as a graph. The attribution mechanism allows us to decorate the graph as follows.

We define these attribution functions:

- $\mathtt{type} : G_N \to \{\mathtt{B}, \mathtt{F}\}$
- $\mathtt{Esp} : G_E \to \mathcal{L}^{\times}$
- $\mathtt{Sval}, \mathtt{lavS} : G_E \to \{0, 1\}$

where, for each node x, `type`(x) is `B` (resp. `F`) if x is a bound (resp. free) variable in φ. Accordingly, a node is said *bound* if the corresponding variable is bound in φ. Moreover, for each edge i, `Esp`(i) denotes the map expression P_{L_i}; and `Sval`(i) (resp. `lavS`(i)) is 1 only if P_{L_i} (resp. $P_{L_i}^{\smile}$) is a single-valued map expression.

A chain of transformations can then be applied to any graph obtained in this manner, by the following somehow intuitive graph-rewriting rules which manifestly preserve the meaning of the graph.

1. An edge attributed by the map expression $\mathbb{1}$ can be removed or created between any two nodes.
2. An edge whose attributing map expression is P, can be converted into an edge, associated to the same pair of nodes—but having opposite orientation, with attributing map expression Q, if either $P \equiv Q^{\smile}$ or $Q \equiv P^{\smile}$ or $P \equiv Q \equiv \iota$.
3. Two edges associated to the same pair of nodes and such that their attributing expressions are P and Q, respectively, can be replaced by a single edge (between the same nodes), attributed by $P \cap Q$, and conversely.
4. Let x be a node, and i,j be two (different) edges such that $G_t(i) = G_s(j) = x$, `type`$(x) =$ `B`, and such that x is not the target or the source node of any other edge. Then the edges i, j, can be replaced by a new edge h (having as source and target, the source of i and the target of j, respectively); the map expression attributing the new edge will be $P;Q$. Conversely, an edge h, attributed by the expression $P;Q$, can be replaced by two edges i, j (whose attributing expressions are P and Q, respectively) such that $G_s(i) = G_s(h)$, $G_t(j) = G_t(h)$, and $G_t(i) = G_s(j)$ is a new bound node.

The productions corresponding to (3) and (4) are displayed in Fig.6 by adopting our graphical formalism. We recall that the numbers inside the nodes visualize the partial graph morphism between the left- and right-hand side graphs. The computation of the `Esp` attribute is performed via Java methods, while the attributes `Sval` and `lavS` are calculated by using Java expressions.

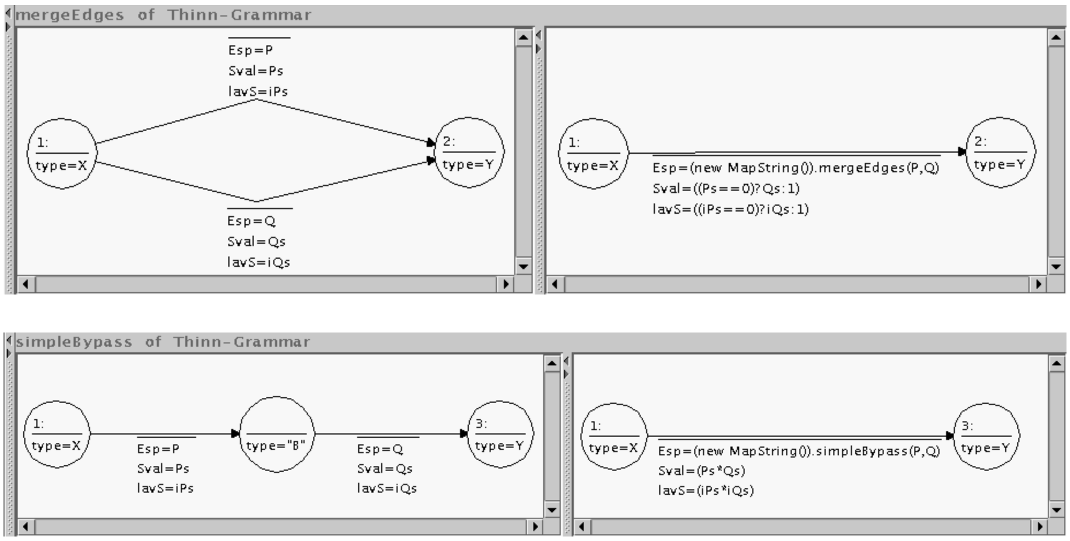


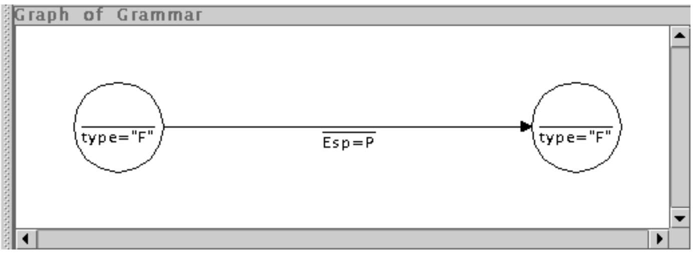
Fig. 6: Two simple rewriting productions

Let us consider Fig.6. Note that X, Y, P, Q, Ps, Qs, iPs and iQs are variables which are instantiated with concrete attribute values when the left-hand side graphs are matched with a certain graph. Given the map expressions denoted by P and Q, the Java methods `mergeEdges` and simpleBypass produce a new map expression corresponding to $P \cap Q$ and $P;Q \cap \mathbb{1}$, respectively (cf. Appendix A).

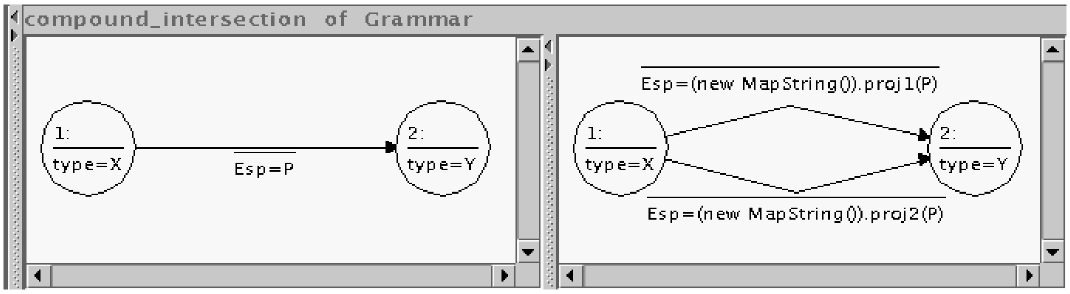
These basic actions can be packaged into relatively complex transformation rules, tactics, and even algorithms of some sophistication, which preserve the meaning of the representation.

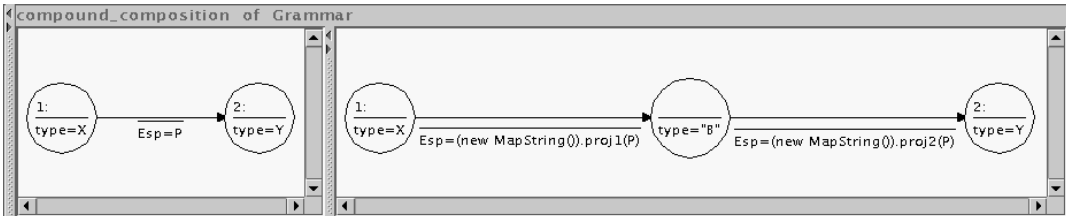
The above rules can be combined to constitute an algorithm for associating an 'unfolded' graph G to a given (compound) map expression P. Two designated nodes s_0 and s_1, named *source* and *sink*, will represent the two arguments of P, and every node distinct from these two will be regarded as being bound.

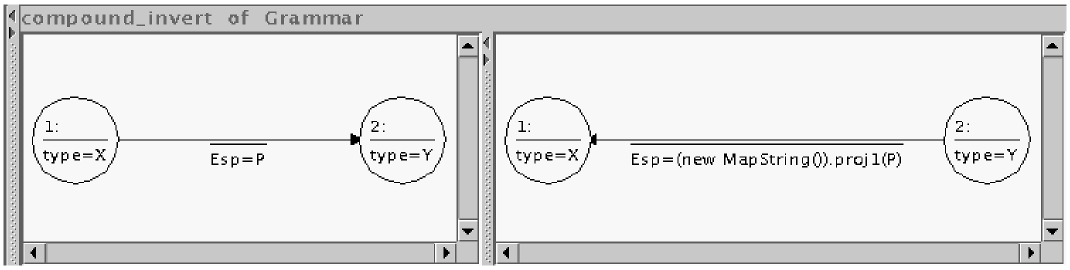
Algorithm. (Graph fattening) Let be given a single-edge graph, let x and y be the (free variables corresponding to the) two nodes, and P a map expression:



one proceeds non-deterministically by applying the following productions:







These productions are applicable when P is instantiated with $P_1 \cap P_2$ (the first production), $P_1;P_2$ (the second one) and $P_1^{\smile}$ (the last one), where P_1 and P_2, in turn, stand for general map expressions. The two projection methods proj1 and proj2 return the map expressions P_1 and P_2, respectively. Applicability of the above productions is checked via Java methods: Fig.7 shows the AGG window *Attribute Context*, which allows the specification of such methods. In this particular case, the right side of the window shows that the method test_appl is designated to check whether the map expression P is of the kind $P_1 \cap P_2$ (this condition refers to the rule `compound_intersection` above shown). $\square$

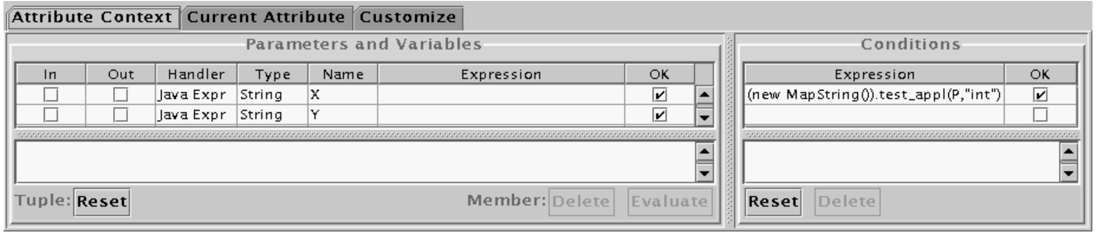


Fig. 7: AGG: the *Attribute Context* window

Notice that the output of the graph-fattening algorithm is a graph having two free nodes (namely, x and y) only, to be intended as the *source* and the *sink* of the resulting graph.

Representing map equalities

Let us now extend our approach in order to suitably treat some slightly more sophisticated situations. For instance, we are looking for a suitable representation of equalities such as:

1. $P = \mathbb{1}$;
2. $P ; \mathbb{1} = \mathbb{1}$, i.e., $\mathsf{Total}(\,P\,)$;
3. $P \neq \emptyset$, i.e., $\mathsf{Total}(\,\mathbb{1} ; P\,)$;
4. $P = \emptyset$.

Notice that these map equalities correspond to closed first-order formulas. For instance, $P = \mathbb{1}$ translates to $(\forall x)(\forall y)(xPy)$. In other words, we convert a graph representation of P (i.e., (xPy)) into the representation of $P = \mathbb{1}$, by bounding x, y by universal quantification.

In general, the four map equalities above listed correspond to these first-order sentences:

1. $(\forall x)(\forall y)(xPy)$
2. $(\forall x)(\exists y)(xPy)$
3. $(\exists x)(\exists y)(xPy)$
4. $\neg(\exists x)(\exists y)(xPy)$

In relation with the graph-fattening algorithm, to render this situations by our graph notation, we refine the attribute function `type` introduced above. Namely, we extend such a function as follows:

$$\mathtt{type} : G_N \rightarrow \{\mathtt{B}, \mathtt{F}, \mathtt{A}, \mathtt{E}, \mathtt{NE}\}$$

where: for each node x, $\mathtt{type}(x)$ is `B` if x is a bound node in the graph obtained by the fattening of P (notice that by the semantics of **;** (cf. Sec.2.1) and our convention for graphical representation of maps, these bound nodes correspond to existentially bound variables in the first-order translation of P).

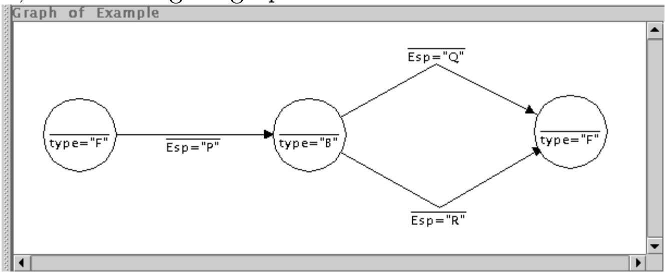
Depending on the equality we are dealing with, the source x and the sink y of the graph representing P will be so attributed:

1. $\mathtt{type}(x) = \mathtt{A}$ and $\mathtt{type}(y) = \mathtt{A}$;
2. $\mathtt{type}(x) = \mathtt{A}$ and $\mathtt{type}(y) = \mathtt{E}$;
3. $\mathtt{type}(x) = \mathtt{E}$ and $\mathtt{type}(y) = \mathtt{E}$;

4. $\texttt{type}(x) = \texttt{NE}$ and $\texttt{type}(y) = \texttt{E}$;

For the sake of simplicity, in what follows, we will refer to these four kind of graph by the writings $\forall\forall$, $\forall\exists$, $\exists\exists$, and $\neg\exists\exists$, respectively.

Thus, for example, the following $\forall\exists$-graph



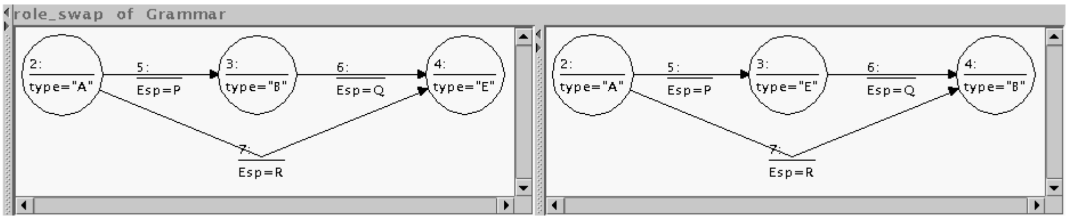
represents the map equality $P\,;(Q \cap R)\,;\mathbb{1} = \mathbb{1}$, i.e., it states that the map $P\,;(Q \cap R)$ is total (i.e., $\mathsf{Total}(\,P\,;(Q \cap R)\,)$, see Sec.2.1).

Remark 1 Obvious rules enabling one to displace source and sink are:

– In a graph of type $\exists\exists$ or $\neg\exists\exists$, the roles of source and sink can indifferently be played by any two distinct nodes;

– in a graph of type $\forall\forall$, source and sink can swap their roles;

– in a graph of type $\forall\exists$, the role of sink can be played by any node distinct from the source, e.g.,



□

Remark 2 Notice that we do not deal with closed formulas of the kind $(\exists x)(\forall y)(xPy)$. This kind of sentences do not cleanly correspond to map equalities. Consider that the equality $\mathbb{1}\,;P = \mathbb{1}$ corresponds to the first-order formula $(\forall x)(\exists y)(xP^{\smile}y)$ (i.e., $(\forall x)(\exists y)(yPx)$), and that exchanging the two quantifiers in $(\exists x)(\forall y)(xPy)$ would yield a semantically different sentence. □

Dealing with complementation

We introduce now a further extension of our graphical representation. In the examples we shown so far, we used exclusively the edge label $\longrightarrow$. In order to permit (partial) handling of complementation, we introduce also the edge label: $\dashrightarrow$. To denote that the associated attributed expression, say P, has to be intended as complemented. Simple graph-rewriting rules related to this convention are shown in Fig.8.

Double-negation rule (i.e., $\overline{\overline{P}} = P$) —as well as double-inversion rule— has been usually embedded in our Java methods. Hence, the method

```
public String complement (String)
```

which is exploited in the rules of Fig.8, applies map-complementation to a given map expression (P, in the figure), and, whenever it is possible, rewrites any expression of the kind $\overline{\overline{P}}$ as P.

The above discussion does not address issues related to the map operators $\triangle$ and $\cup$. Concerning these constructs, we only report in Fig.9 a rule regarding $\cup$ that will be exploited in the sequel.

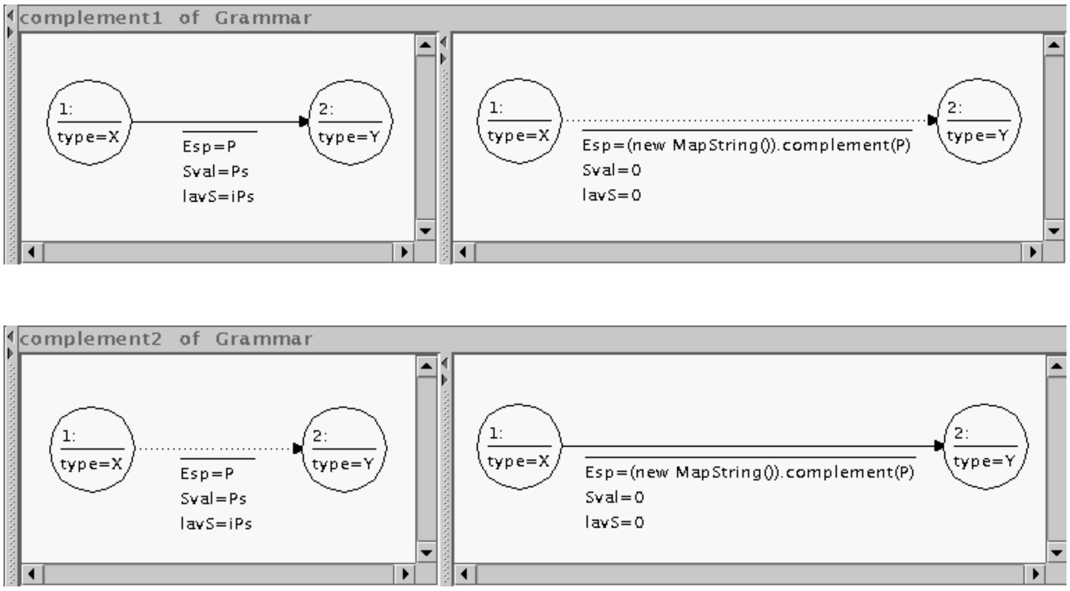


Fig. 8: Two rewriting rules involving map complementation

This rule is a slightly generalized version of the 'de Morgan' rule: $\overline{P\,;T}\cap\overline{Q\,;T}=\overline{(P\cup Q)\,;T}$. The rewriting rule is applicable only when the two variables R and S are instantiated by expressions of the form $P\,;T$ and $Q\,;T$, respectively, for suitable P, Q, T. In this case, the map expression attributing the new edge in the right-hand side of the rule will be $(P\cup Q)\,;T$ (as before, both the evaluation of the applicability condition, and the generation of the resulting map-expression, are realized through Java methods).

Further rules related to $\triangle$ and $\cup$ have been introduced, for instance, in [10].

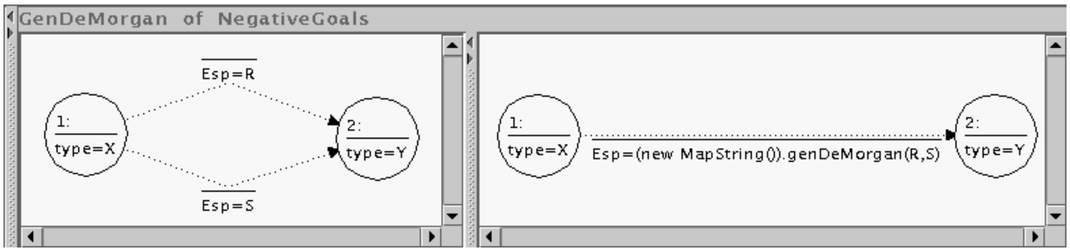


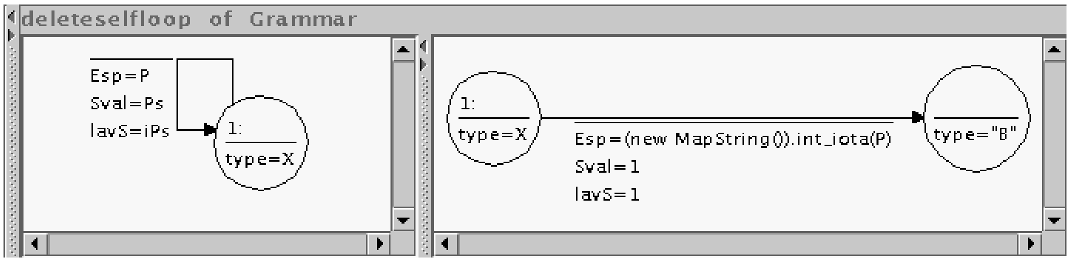
Fig. 9: A useful rewriting rule involving $\cup$

In Sec.4 we will see that our treatment of complementation, albeit partial, allows one to model/represent map inclusions of the form $P \subseteq Q$—this corresponds to represent map equalities such as $P \cap \overline{Q} = \emptyset$ by means of $\neg\exists\exists$-graphs. Moreover, we will describe a simple set of rewriting rules for inferring (new) map inclusions.

3 Automated translation from first-order logic into map calculus

As is well known (cf. [45]), a sentence α of dyadic first-order logic can be translated into a ground map equality if and only if α is logically equivalent to a sentence involving at most three distinct variables. This characterization of translatable sentences is, alas, not very useful in practice: establishing whether a given α belongs to this collection is in fact an undecidable problem (cf. [33]). Notwithstanding, conservative translation techniques can be devised to partially solve the problem. Let us introduce one of these techniques together with a number of its conservative refinements.

Algorithm. (Graph thinning) An existentially quantified conjunction φ of literals of the form xPy is given (cf. Sec.2.2). The goal is to find a quantifier-free conjunction —or simply an atom, if there are at most two free variables in φ— equivalent to φ. Initially, the graph G_φ is considered, then it is normalized by merging multiple edges associated to the same pair of nodes (cf. the first rule in Fig.6), and by eliminating loop-edges:



given a loop-edge attributed with a map expression P, a new bound node is introduced and the loop-edge is substituted with an edge to the bound node, attributed by $P\cap\iota$, which is the result of the method call `int_iota(P)`. Clearly, both this new map expression and its inverse map are functions.

This G_φ will be manipulated with the aim of eliminating as many bound nodes as possible. This elimination (which represents the elimination of existential quantifiers from φ) is performed by repeatedly applying the two graph-transformation rules displayed in Figures 10 and 11.

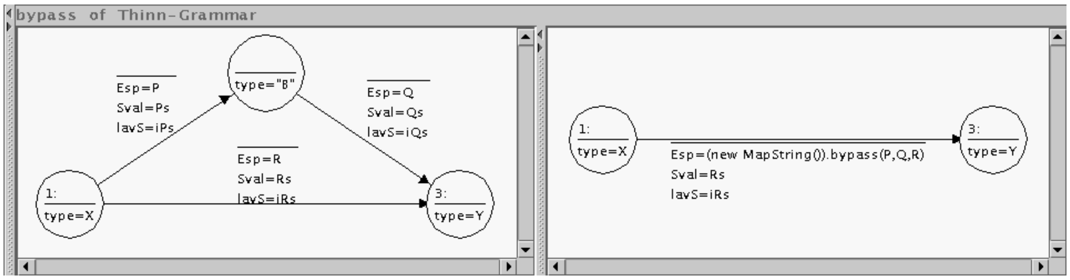


Fig. 10: The BYPASS rule

More precisely, given the map expressions P, Q and R, the method call `bypass(P,Q,R)` of Fig.10 returns the expression $P;Q\cap R$, while the method call `bigamy(Q,R)` of Fig.11 returns the expression $\mathbb{1};Q\cap R$.

The process ends when no further applications of the above rules can be carried out. If the resulting graph has no bound nodes of degree greater than 1, the sought conjunction can be directly read off the graph, else we have a failure. □

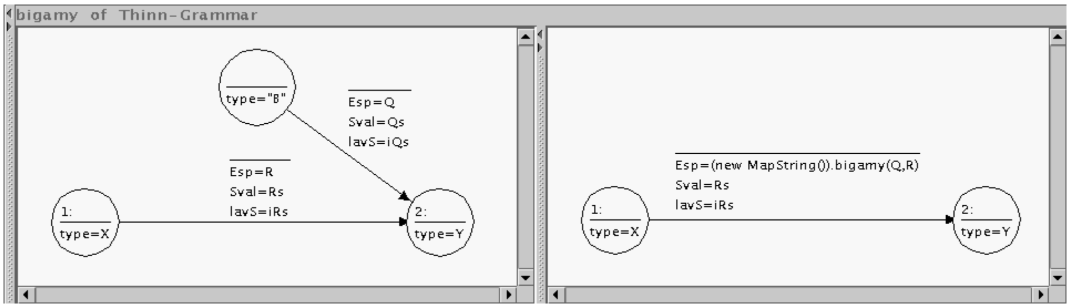
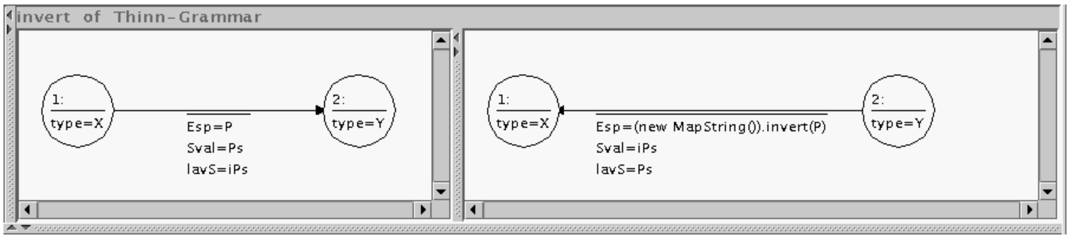


Fig. 11: The BIGAMY rule

Remark 3 Notice that in order to fully realize the thinning algorithm it should be necessary to add the rules of Figures 10 and 11, a further rule allowing AGG to invert any edge:



where the result of the method call `invert(P)` is the map expression $P^{\smile}$.

Clearly, this rule does not amalgamate well with a non-deterministic strategy for rule application. In order to ensure termination of the rewriting process, we need to embed implicit applications of the inversion rule within the other productions. Actually, we enriched the graph-grammar by introducing (in place of the inversion rule) suitable variants of the bypass and the bigamy rules. Examples of these additional rules are shown in Fig.12. ☐

Usually, it may be the case that a given map expression involved/generated in a step of the thinning algorithm could be simplified by applying simple algebraic manipulations. These transformations are justified by the properties that the map constructs must satisfy once we have assumed, for instance, the proper axioms in Fig.5. Some of these obvious properties are

$$P\,;\iota = P, \qquad \overline{\overline{P}} = P, \qquad \overline{P} \cup P = \mathbb{1},$$
$$\emptyset\,; P = \emptyset, \qquad P\,;\emptyset = \emptyset.$$

These laws, as well as many others, can be straightforwardly derived from the axioms of $\Lambda^{\times}$. Several properties of this kind have been automatically proved/verified by adopting a standard approach based on term rewriting techniques (cf., for instance, [22, 20]). In Sec.4 we will describe a (semi-)automated approach based on graph-rewriting methods, designed for solving this task.

As mentioned, in AGG any expression that attributes an object in the right-hand side of a production, is usually obtained by calling Java methods. This feature allows us to embed the above (among others) algebraic transformation in the Java code, and hence represents an initial step toward the combination of graphic and algebraic rewriting techniques.

In [8] it has been proved that the thinning algorithm has complexity $\mathcal{O}(|N_\varphi| \log |N_\varphi|)$, where N_φ is the set of all nodes of G_φ. Moreover, it has been proved that the following confluence property holds: the order in which bypass and bigamy actions (together with their variants needed to get rid of the invert rule) are performed during an execution of the thinning algorithm is immaterial.

It should be noticed that our initial implementation of the thinning algorithm exploited the default strategy supplied by AGG for firing the rewriting rules. As explained in Sec.1.1, this strategy selects the rules following their order in the grammar, and applies each rule as much as

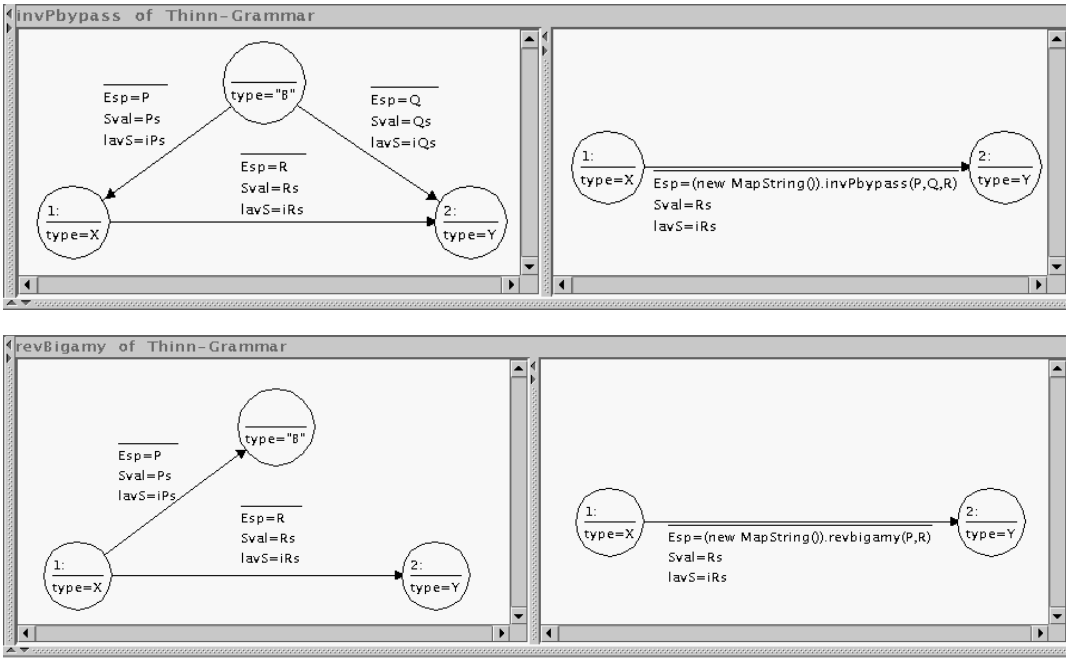


Fig. 12: Two variants of the bypass and the bigamy rules

possible. Unfortunately, this first solution is somehow unsatisfactory. Hence, the subsequent step consisted in exploiting the confluence properties of the thinning algorithm [8]. As mentioned, this result states that: if there exists a derivation of a graph G' from the start graph G that removes all of the bound nodes, then a strategy that applies the rewriting rules of Figures 10 and 11 (together with their variants, as mentioned above) in a non-deterministical fashion suffices.

A new strategy for production applications has been implemented. This allowed AGG to support the encoding of a fully non-deterministic version of the thinning algorithm. Actually, this is the strategy we employ in what follows.

Some worked examples

Let us see through a series of examples how the thinning algorithm works. Firstly, we recall a map-specification of nested lists built from two or more atoms, as described in [8]:

$$\begin{array}{|lcr|}
\hline
\mathsf{Const}(\,\mathsf{nl}\,) & & \mathsf{at} \cap \mathsf{nl} = \emptyset \\
\mathsf{Func}(\,\mathsf{hd}\,) & & \mathsf{Func}(\,\mathsf{tl}\,) \\
\hline
\mathsf{hd}\,;\mathbb{1} = \mathsf{tl}\,;\mathbb{1} & & \mathsf{at} \,\triangle (\,\mathsf{nl}\,;\mathbb{1}\,) = \overline{\mathsf{tl}\,;\mathbb{1}} \\
\overline{\mathsf{at}} \subseteq \mathsf{tl}^{\smile}\,;\mathsf{hd} & \mathsf{at}\,;\mathsf{at}^{\smile} - \iota \neq \emptyset & (\,\mathsf{hd}\,;\mathsf{hd}^{\smile}\,) \cap (\,\mathsf{tl}\,;\mathsf{tl}^{\smile}\,) \subseteq \iota \\
\hline
\end{array}$$

In this specification, the 'nil' predicate nl, represents the distinguished individual NL, not to be counted among atoms. By the map equalities on the second row, the 'head' and 'tail' predicates (hd and tl respectively) are partial functions, which, by the equality $\mathsf{hd}\,;\mathbb{1} = \mathsf{tl}\,;\mathbb{1}$, are defined for the same entities (to be regarded as the 'lists') of $\mathcal{U}$; moreover, the complement of their common domain consists of all atoms together with NL. Notice that, by $\mathsf{at} \,\triangle (\,\mathsf{nl}\,;\mathbb{1}\,) = \overline{\mathsf{tl}\,;\mathbb{1}}$, the 'is atom' predicate at does not truly depend on its second argument. The remaining axioms state that

- for any given pair a, b in $\mathcal{U}$ with b a list or $b = \mathsf{NL}$, one can find a list whose head is a and whose tail is b;
- there are at least two atoms;
- lists that differ from one another cannot have the same head and the same tail.

We will use this specification as a starting point for specifying the abstract data-type of a line-editor. This example is borrowed from [17], where the basic type *string* is exploited in place of lists.

For the sake of completeness, we start by giving a Prolog specification of the line-editor; the next step will consist in compiling this specification into map calculus by using our graphical implementation of the thinning algorithm.

```
rotate([A],[A]).
rotate([A,B|C],[B|D]) :- rotate([A|C],[D]).

concat([[]|A],A).
concat([[A|B]|C],[A|D]) :- concat([B|C],D).

char(A) :- at(A).

string([]).
string([A|B]) :- char(A), string(B).

line([A|B]) :- string(A), string(B).

clear(A,[[]]) :- line(A).

createLine(A) :- clear(A,A).

toBeginning(A,[[]|B]) :- line(A), concat(A,B).

toEnd(A,[B]) :- line(A), concat(A,B).

delChar([A],[A]) :- line([A]).
delChar([A|[B|C]],[A|C]) :- line([A|[B|C]]).

shift([A|[B|C]],[D|C]) :- line([A|[B|C]]), rotate([B|A],D).

moveRight([A],[A]) :- line([A]).
moveRight(A,B) :- shift(A,B).

moveLeft([[]|A],[[]|A]) :- line([[]|A]).
moveLeft(A,B) :- shift(B,A).

addChar([A|[B|C]],D) :- shift([B|[A|C]],D).
```

Predicates `rotate/2` and `concat/2` characterize two basic operations on lists (to be more precise, on lines): *rotation* and *concatenation*. Rotating a list means removing its head from the beginning to place it at the end of the list. Concatenating two lists means, as usual, inserting the elements of the second of them at the end of the first; notice that, in order to permit a more direct translation into the map formalism, we did not define concatenation by means of a ternary predicate: the binary predicate `concat/2` can be considered as characterizing a unary function which concatenates together head and tail of its argument.

Predicate `string/1` states that a *string* is either NL or a flat list of *characters* drawn from an alphabet consisting of atoms; while `line/1` defines a *line* to be a list whose head and tail both are strings; virtually, a *cursor* separates these two strings. Then there are defined operations on lines: `clear/2` empties both the string that precedes the cursor and the one that follows it, `createLine/1` initializes a line as a pair of empty strings; `toBeginning/2` and `toEnd/2` bring the cursor at the two extreme positions of a line; `delChar/2` erases the character following the cursor, if any, while `addChar/2` inserts a character after the cursor and shifts the cursor to the immediate right of the new character; `moveRight/2` and `moveLeft/2` shift the cursor to the right or to the left by one character, when this is possible.

$$
\begin{array}{l}
\mathsf{rotate}{=}\mathsf{tl}\,;\mathsf{nl}\,;\mathbb{1}\cap\iota\,\triangle \\
\qquad\qquad\big(\,\mathsf{hd}\,;\mathsf{hd}^{\smile}\cap\mathsf{tl}\,;\mathsf{tl}\,;\mathsf{tl}^{\smile}\,\big)\,;\mathsf{rotate}\,;\mathsf{tl}^{\smile}\cap\mathsf{tl}\,;\mathsf{hd}\,;\mathsf{hd}^{\smile} \\
\mathsf{concat}{=}\mathsf{hd}\,;\mathsf{nl}\,;\mathbb{1}\cap\mathsf{tl}\,\triangle \\
\qquad\qquad\big(\,\mathsf{tl}\,;\mathsf{tl}^{\smile}\cap\mathsf{hd}\,;\mathsf{tl}\,;\mathsf{hd}^{\smile}\,\big)\,;\mathsf{concat}\,;\mathsf{tl}^{\smile}\cap\mathsf{hd}\,;\mathsf{hd}\,;\mathsf{hd}^{\smile} \\
\mathsf{char}{\subseteq}\mathsf{at}\cap\iota \\
\mathsf{string}{=}\mathsf{nl}\,\triangle\,\mathsf{hd}\,;\mathsf{char}\,;\mathbb{1}\cap\mathsf{tl}\,;\mathsf{string}\,;\mathbb{1}\cap\iota \\
\mathsf{line}{=}\mathsf{hd}\,;\mathsf{string}\,;\mathbb{1}\cap\mathsf{tl}\,;\mathsf{string}\,;\mathbb{1}\cap\iota \\
\mathsf{clear}{=}\mathsf{line}\,;\mathbb{1}\,;(\,\mathsf{hd}\,;\mathsf{nl}\cap\mathsf{tl}\,)^{\smile} \\
\mathsf{createLine}{=}\mathsf{clear}\cap\iota \\
\mathsf{toBeginning}{=}\mathsf{line}\,;\mathsf{concat}\,;(\,\mathsf{hd}\,;\mathsf{nl}\,;\mathbb{1}\cap\mathsf{tl}\,)^{\smile} \\
\mathsf{toEnd}{=}\mathsf{line}\,;\mathsf{concat}\,;(\,\mathsf{tl}\,;\mathsf{nl}\,;\mathbb{1}\cap\mathsf{hd}\,)^{\smile} \\
\mathsf{delChar}{=}\mathsf{line}\,;\mathsf{tl}\,;\mathsf{nl}\,;\mathbb{1}\cap\iota\,\triangle\,\mathsf{line}\,;\mathsf{tl}\,;\mathsf{tl}\,;\mathsf{tl}^{\smile}\cap\mathsf{hd}\,;\mathsf{hd}^{\smile} \\
\mathsf{shift}{=}\mathsf{line}\,;\mathsf{tl}\,;\mathsf{tl}\,;\mathsf{tl}^{\smile}\cap(\,\mathsf{hd}\,;\mathsf{tl}^{\smile}\cap\mathsf{tl}\,;\mathsf{hd}\,;\mathsf{hd}^{\smile}\,)\,;\mathsf{rotate}\,;\mathsf{hd}^{\smile} \\
\mathsf{moveRight}{=}\mathsf{line}\,;\mathsf{tl}\,;\mathsf{nl}\,;\mathbb{1}\cap\iota\,\triangle\,\mathsf{shift} \\
\mathsf{moveLeft}{=}\mathsf{line}\,;\mathsf{hd}\,;\mathsf{nl}\,;\mathbb{1}\cap\iota\,\triangle\,\mathsf{shift}^{\smile} \\
\mathsf{addChar}{=}\big(\,(\,\mathsf{hd}\,;\mathsf{hd}^{\smile}\cap\mathsf{tl}\,;\mathsf{tl}\,;\mathsf{tl}^{\smile}\,)\,;\mathsf{tl}^{\smile}\cap\mathsf{tl}\,;\mathsf{hd}\,;\mathsf{hd}^{\smile}\,\big)\,;\mathsf{shift}
\end{array}
$$

Fig. 13: Specification of a simple line-editor

The map-formulation of the line-editor is displayed in Fig.13. The translations have been obtained by our implementation of graph-thinning realized on top of AGG. Notice that some of the predicate in the Prolog specification are unary (namely, `char/1`, `string/1`, `line/1`, and `createLine/1`). In order to re-formulate them into map calculus we will exploit maps that are (proper) subsets of ι. This approach is alternative to the choice adopted to characterize the 'is atom' predicate by means of a map at that is asbolute on its right argument.

Let us describe the automated translations of the following two clauses as produced by AGG.

(1) `delChar([A],[A]) :- line([A]).`

(2) `delChar([A|[B|C]],[A|C]) :- line([A|[B|C]]).`

A preliminary step for each translation ammounts to normalizing the given clause. This phase yields the clauses:

(1) `delChar(X,Y) :- line(X), X=[A], X=Y.`

(2) `delChar(X,Y) :- line(X), X=[A|[B|C]], Y=[A|C].`

which correspond to the following formulas:

(1) $(\exists z)(\exists v)(\exists w)(X\,\mathsf{line}\,z\ \wedge\ z\,\mathsf{tl}\,v\ \wedge\ v\,\mathsf{nl}\,w\ \wedge\ X\,\iota\,Y)$

(2) $(\exists z)(\exists u)(\exists v)(\exists w)(X\,\mathsf{line}\,z\ \wedge\ z\,\mathsf{tl}\,u\ \wedge\ u\,\mathsf{tl}\,v\ \wedge\ Y\,\mathsf{tl}\,v\ \wedge\ Y\,\mathsf{hd}\,w\ \wedge\ X\,\mathsf{hd}\,w)$

The start graphs relative to these formulas are shown on top of Figures 14 and 15, respectively. Each intermediate graph produced by the application of the rewriting productions during the AGG runs, are also shown for both clauses.

The complete translation of the predicate `delChar/2` consists in the sum of the two map expressions which can be read from the final graphs of Figures 14 and 15. Since the two clauses of `delChar/2` are to be intended to be mutually exclusive and the resulting maps are disjoint, we are enabled to employ symmetric difference in place of sum. Moreover, in view of [44], we resort to the closed-world assumption and replace map inclusion by map equality.

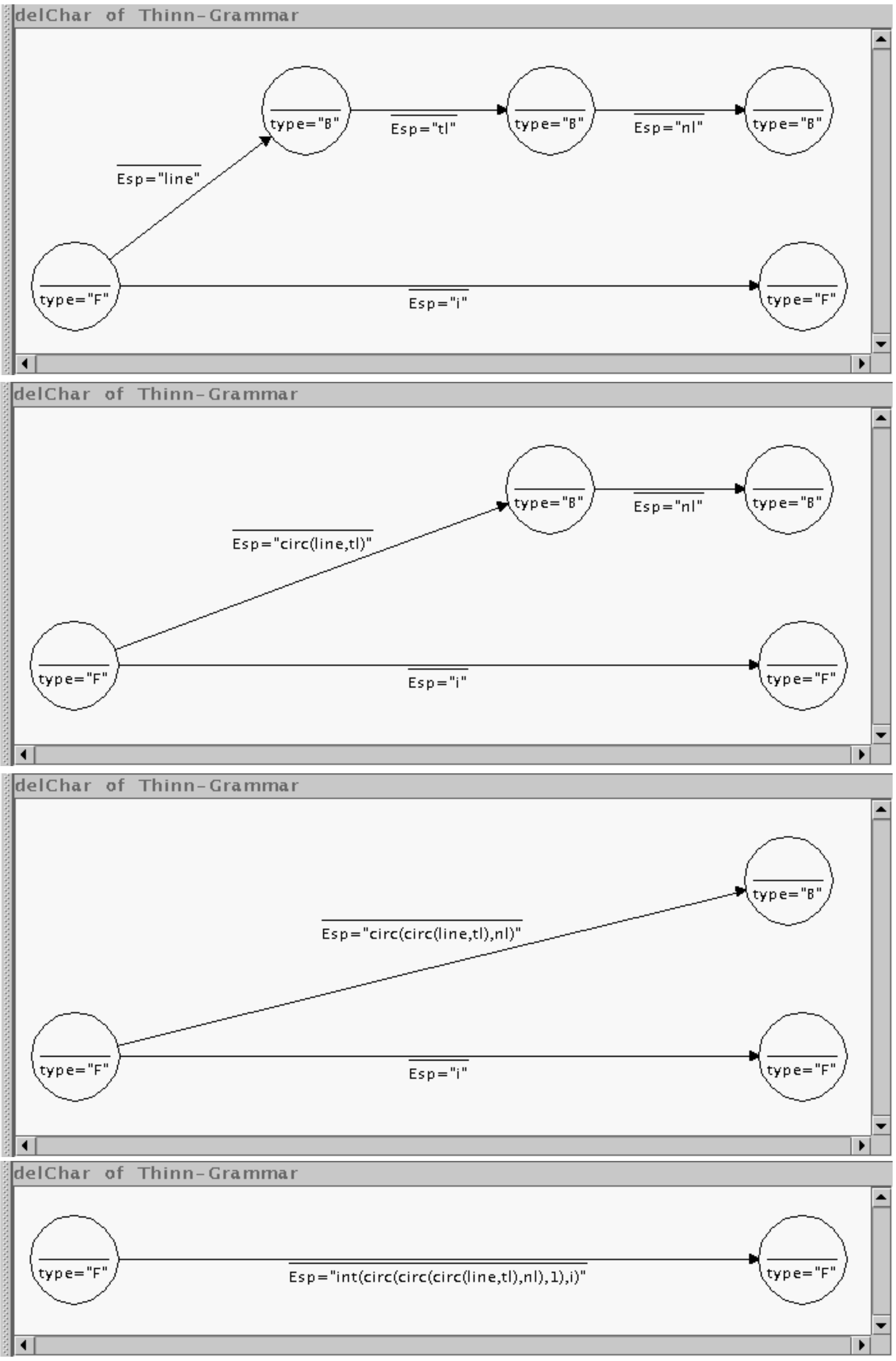


Fig. 14: Derivation steps for translating the specification of `delChar/2` (first clause)

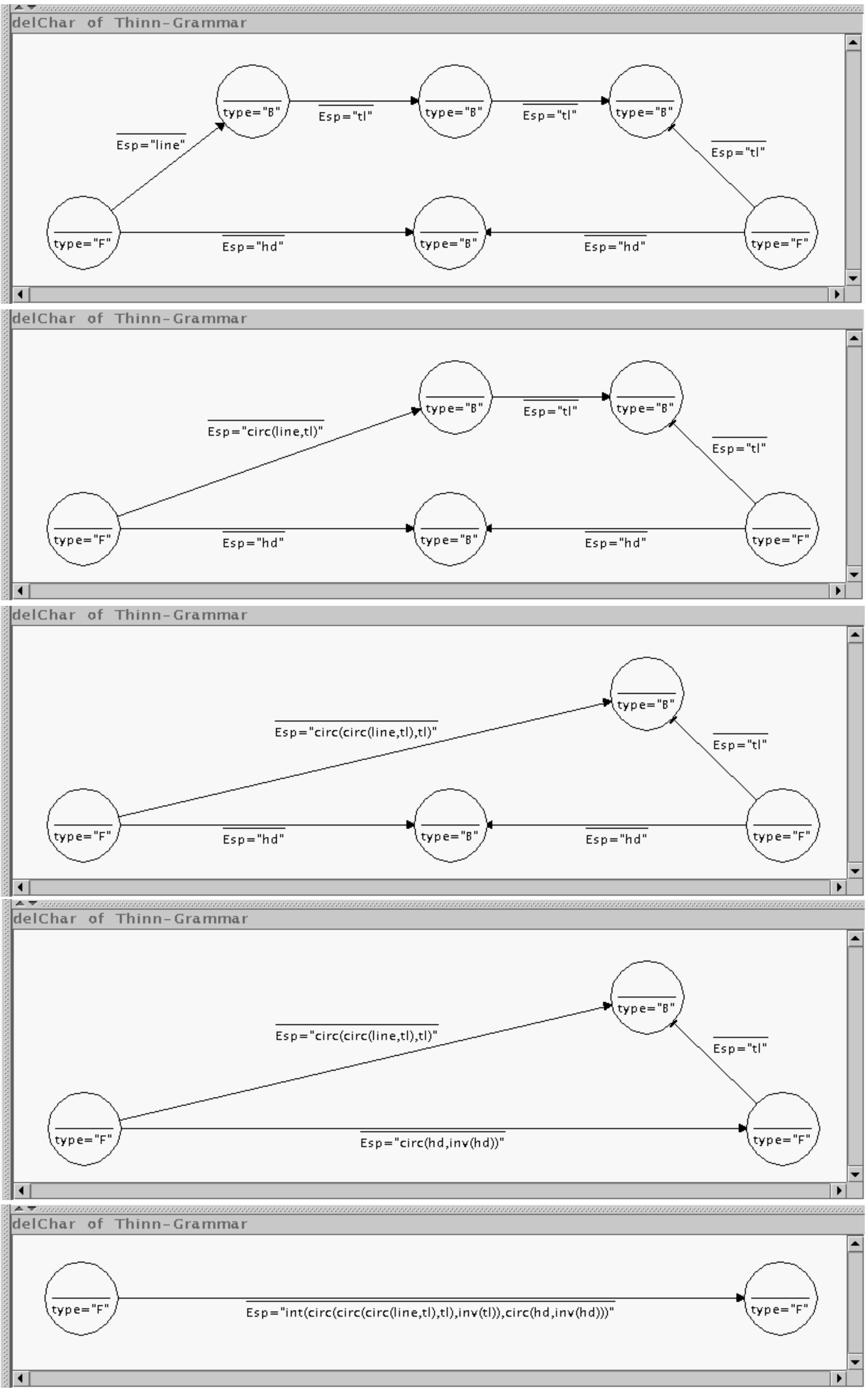


Fig. 15: Derivation steps for translating the specification of `delChar/2` (second clause)

Exploiting semantical knowledge in the translation process

As we will see in the sequel of this section, the thinning algorithm may fail in completing the translation of a particular formula even though it is translatable. This is consequence of the above-mentioned limitative results on decidability proved in [33].

We introduce some conservative extensions of the basic algorithm which exploit semantical knowledge on the formula to be translated.

These refinements overcome the limitations of the basic technique by means of the introduction of new rewriting rules. Nevertheless, it can be shown that in presence of this new rules the confluence property of the algorithm may be lost [8].

Exploiting functionality of maps

Let us attempt to translate the clause which defines the predicate `addChar/2`:

```
addChar([A|[B|C]],D) :- shift([B|[A|C]],D).
```

Once normalization is performed, we obtain:

```
addChar(X,Y) :- shift(Z,Y), X=[A|[B|C]], Z=[B|[A|C]].
```

From this clause we easily obtain the start graph displayed at the top of Fig.17. Then we run our implementation of the thinning algorithm. After few applications of rewriting productions the procedure is stuck. In fact, no further rewriting rules can be applied to the graph displayed at the bottom of Fig.17. In order to complete the translation we have to exploit semantic knowledge on the involved maps. Now the two attributes `Sval` and `lavS` enter into play, and we will refine the thinning algorithm by adding a new transformation rule which uses the fact that (some of) the maps actually are assumed to be functions rather than generic dyadic relations. Fig.16 shows the *functionality* rule. Intuitively speaking, suppose that a single-valued map-expression P (i.e., such that $(\forall x)(\forall y)(\forall z)\ (xPy \wedge xPz \rightarrow x\iota y))$ attributes an edge directed to a bound node x. Then, the node x and the edge itself can be duplicated and the edges exiting from x can be partitioned between x and its copy. Subsequently, the usual productions (i.e., bypass and bigamy) become applicable.

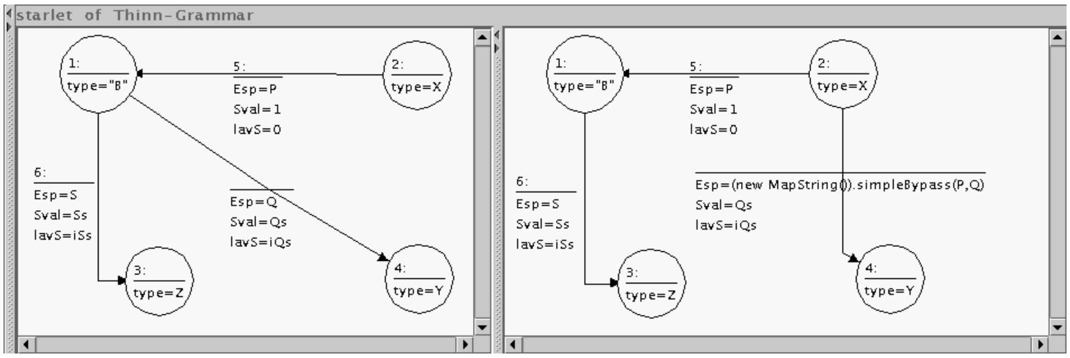
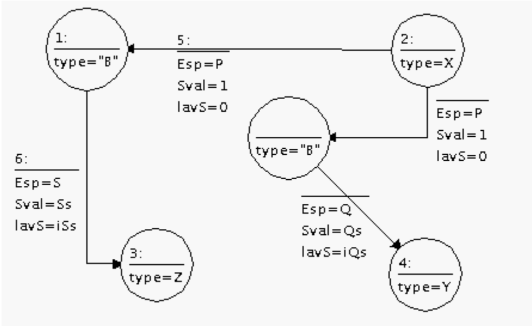


Fig. 16: The Functionality rule

In order to explain how the production of Fig.16 works, consider that it exploits the functionality of the expression `P` attributing the edge marked `5:`. The node marked `1:` is duplicated yielding this intermediate configuration:



then, the bypass rule is applied to bypass the newly created node (which never really appears in the transformation process).

A single application of the functionality rule would permit the algorithm to complete the translation as shown in Fig.18.

Several strategies could be adopted in extending our implementation in order to deal with single-valued maps. The easiest way consists in simply adding the functionality rule to the others and in running AGG in its non-deterministical fashion already adopted in the case of the original thinning-algorithm.

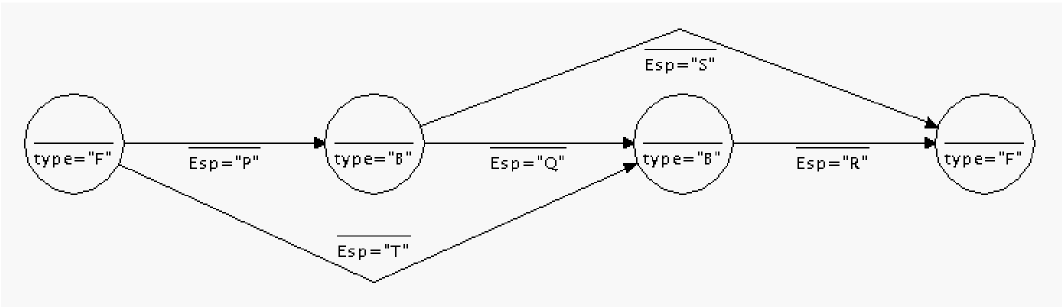
An alternative solution would consist in applying the functionality rule only when no other production is applicable. This strategy has been quickly implemented and embedded into AGG. It has been succesfully used to obtain the run shown in Figures 17 and 18, a well as in a number of other cases (actually, the specification of the line-editor of Fig.13 is an excerpt of them). The refined algorithm proceeds by repeating the following two phases, until the graph is no more reducible:

- keep selecting and applying (only once) the bypass rule or the bigamy rule (or anyone of their variants, cf. Fig.12), as much as possible;
- apply once the functionality rule.

Exploiting absoluteness of maps

A map T is said to be *absolute* w.r.t. its right argument, if it holds that $T\,;\mathbb{1}=T$.[4] In what follows we show how absoluteness can be profitably used during the translation process.

Consider the following graph

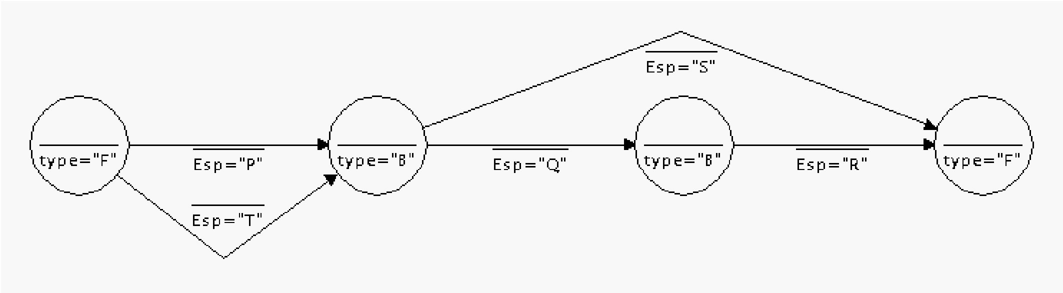


corresponding to the first-order formula

$$(\exists\, z)(\exists\, w)\ (XPz\ \wedge\ zQw\ \wedge\ wRY\ \wedge\ zSY\ \wedge\ XTw).$$

The graph cannot be reduced by means of any of the rules introduced until now. Nevertheless, we can exploit the fact that $T\,;\mathbb{1} = T$ to justify the transformation of the above graph into the following one:

[4] A relation satisfying this property is called *vector* in [40].



Consequently, by applying bypass and bigamy we obtain the map expression $(P \cap T);(Q;R \cap S)$.

Conjugated projections

As a further example, let us assume that two particular maps L, R, fulfill the following conditions:

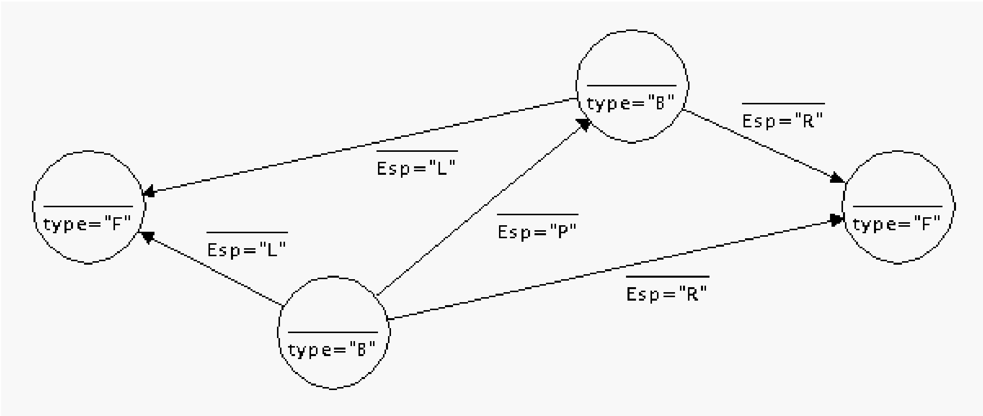
$$L^{\smile}; R = \mathbb{1}, \qquad \mathsf{Func}(L), \qquad \mathsf{Func}(R), \qquad L; L^{\smile} \cap R; R^{\smile} \subseteq \iota.$$

In this case we say that L and R are *conjugated projections* (cf. [45]).

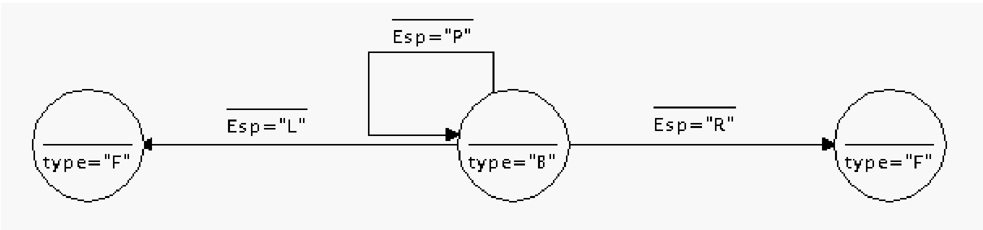
The fact that two maps are conjugated projections can be used to obtain the translation of formulas that otherwise could not be translated. For instance, consider the first-order formula:

$$(\exists z)(\exists w)\ (zLX \wedge zRY \wedge wLX \wedge wRY \wedge zPw).$$

The corresponding grapical representation is:



which cannot be reduced by any of the above-outlined methods. Nevertheless, the fact that L, R, taken together, define an injection, justifies the transformation of the preceeding graph into:



where the two bound nodes (corresponding to the bound variables z, w) have been merged because they must represent the same element. From this graph, by using the transformation rules described in the previous sections, we finally get the map expression sought for: $L^{\smile};(R \cap (P \cap \iota);\mathbb{1})$.

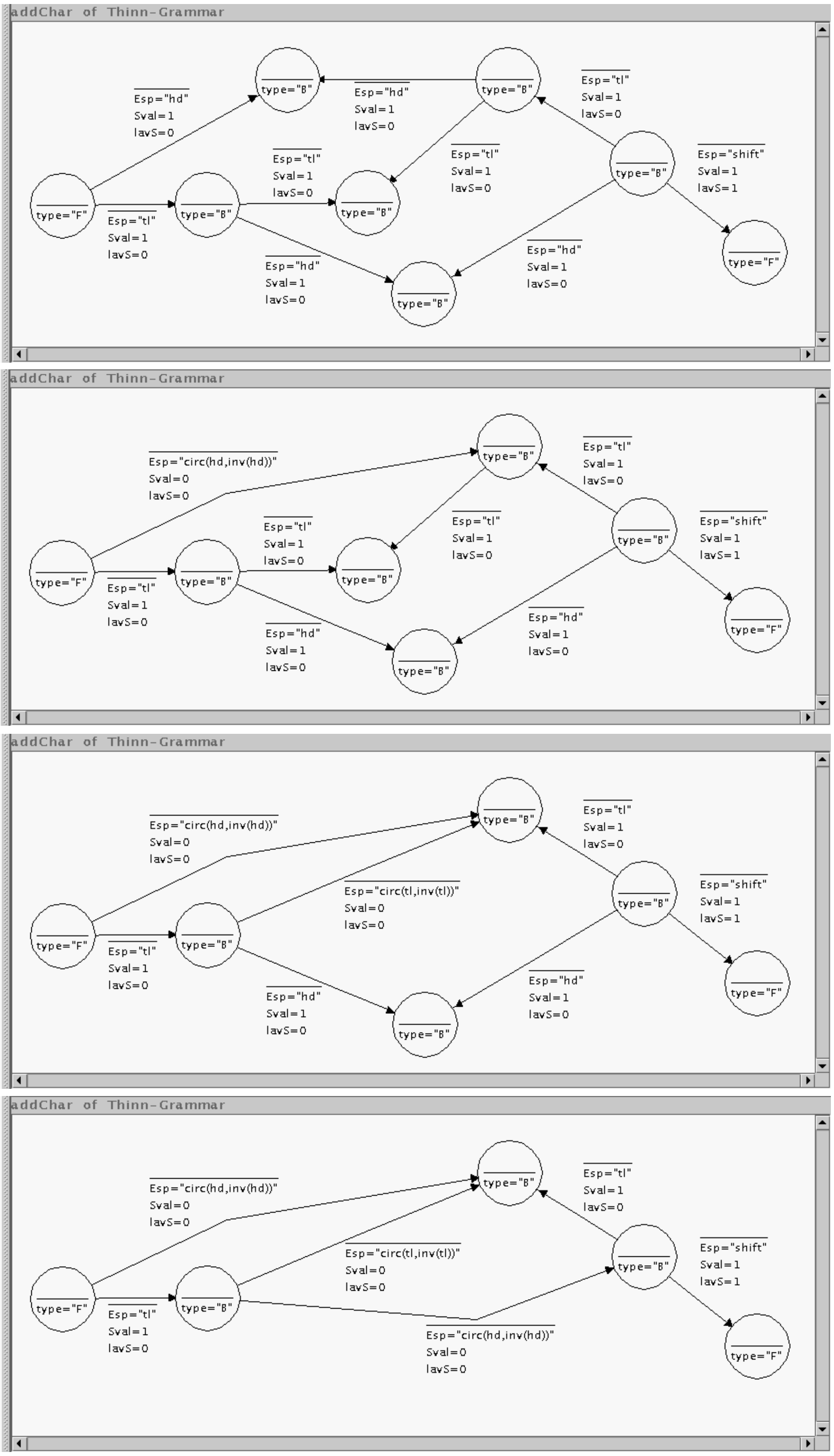


Fig. 17: Derivation steps for translating the specification of `addChar/2` (continued)

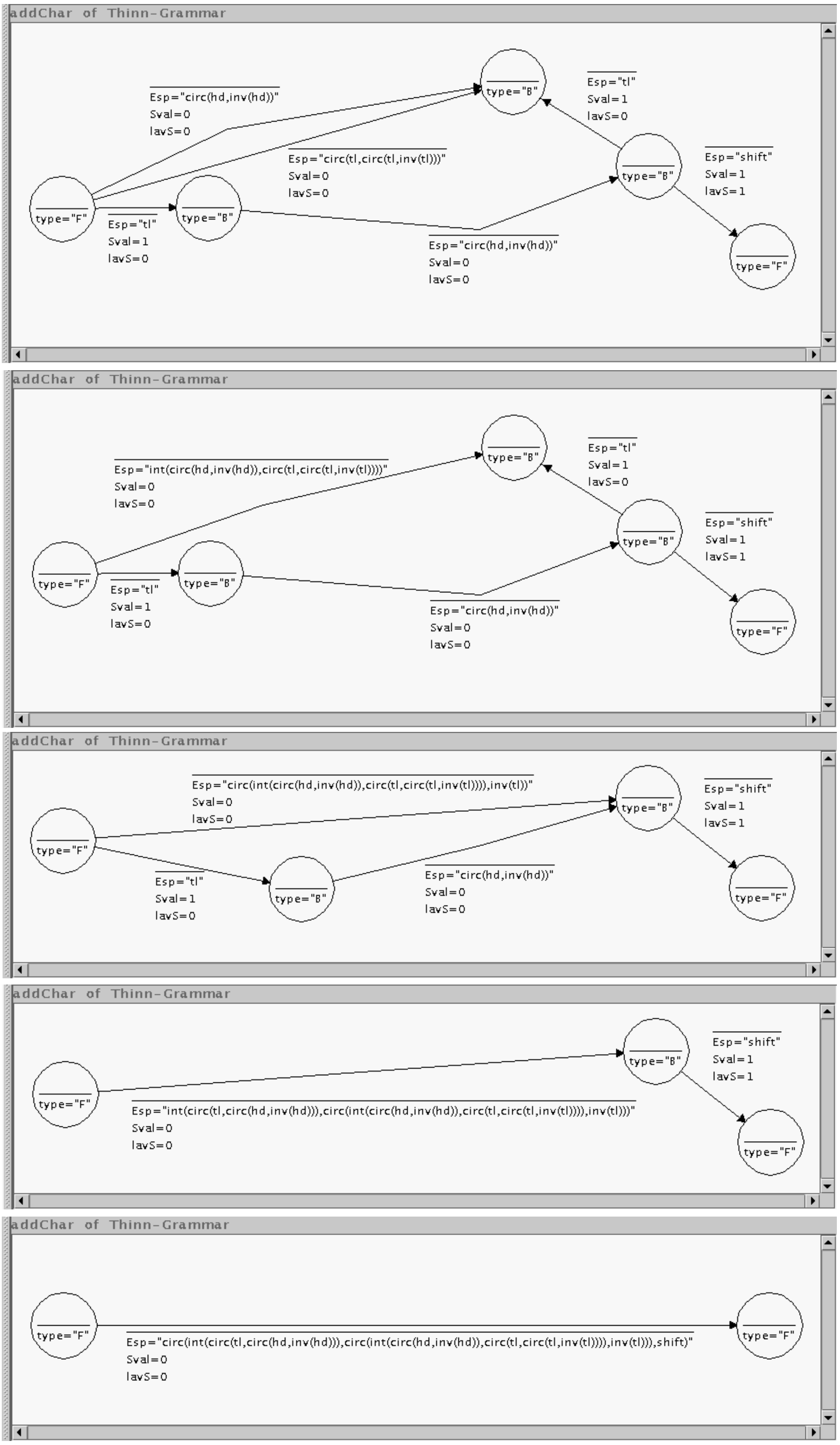


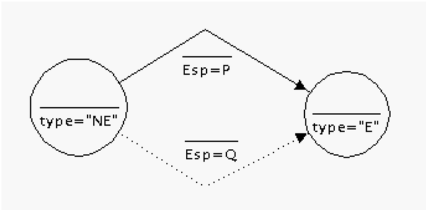
Fig. 18: Derivation steps for translating the specification of `addChar/2`

4 Implementation of a proof-assistant through graph-rewriting techniques

Deriving map equalities from proper axioms, and from laws already known, can to a large extent be viewed as a graph-rewriting activity. In this section we are interested in two particular classes of graphs, namely, the existential graphs of the two types $\forall\forall$ and $\neg\exists\exists$. We call them *positive* and *negative* graphs, respectively. These graphs are exploited to represent premises and conclusions, known laws, theses, etc.

From this perspective, inference mechanisms are seen as graph-rewriting rules and techniques: in forward reasoning, they are used to transform premises into conclusions; in backward reasoning, to reduce theses —'goals', as they are often called— into simpler goals and, ultimately, into known and perhaps obvious facts.

As a basic principle, it is legitimate to replace a positive goal by a more demanding one, and a negative goal by one less demanding. E.g., new attributed edges can be added at will to a positive goal, whereas edges can be removed from a negative goal. Solving the new goal, although not necessarily equivalent to solving the previous goal, will in fact suffice for the purpose. Quite often a negative premise represents an inclusion $P \subseteq Q$, i.e., $\neg(\exists x)(\exists y)(x\,P \cap \overline{Q}\,y)$:



If a subgraph of a positive goal matches the part of the premise which represents Q, then it can be replaced by the part representing P; in a negative goal, on the opposite, Q may replace P.

For the sake of simplicity, in what follows we refer to negative graphs only. An analogous treatment could be straightforwardly described for the case of positive graphs.

By considering negative goals, the above-outlined basic rule is formulated in the AGG graphical language as displayed in Fig.19, where the nodes marked `4:` and `5:` constitute (together with the edges marked `10:` and `12:`) the negative premise; while the nodes `6:` and `9:` identify the edge to be replaced during the firing of the rule. Notice that the negative premise $P \subseteq Q$ remains unchanged in the transformed graph. Moreover, to ensure the soundness of the rule, further conditions have to be imposed. In fact, during the application of the rule, the negative premise must match with a complete connected component of the graph.

The properties of inclusion (e.g., transitivity) easily determine simple inference rules, Fig.20 shows some of them.

As mentioned, proving a particular map equality in the framework we are going to describe, amounts to reducing the corresponding graph/goal into simplier goals, these goals into other goals, and so on until known facts are derived. Below we listed some of these obvious facts, or *axioms*, related to map-inclusion:

$$\begin{array}{rcc} P \subseteq \mathbb{1}, & P \subseteq P, & P \subseteq P\,;\iota, \\ P\,;\iota \subseteq \iota\,;P, & \mathbb{1} \subseteq P \cup \overline{P}, & \iota\,;P \subseteq P. \end{array}$$

Axioms correspond to AGG productions that remove obvious facts from the graph. The left-hand sides of these rules (which have empty right-hand sides) are displayed in Fig.21.

Notice that AGG imposes that only constants or variables can attribute the left-hand side of a production. Because of this restriction, the sintactical structure of the expressions attributing the edges of the axioms has to be inspected by exploiting suitable Java methods. For instance, consider (A_5) in Fig.21 which corresponds to the axiom $\mathbb{1} \subseteq P \cup \overline{P}$. This rule is applicable only when the variable Q is instantiable by a map expression of the form $P \cup \overline{P}$, for some P. This

(1)

(2)

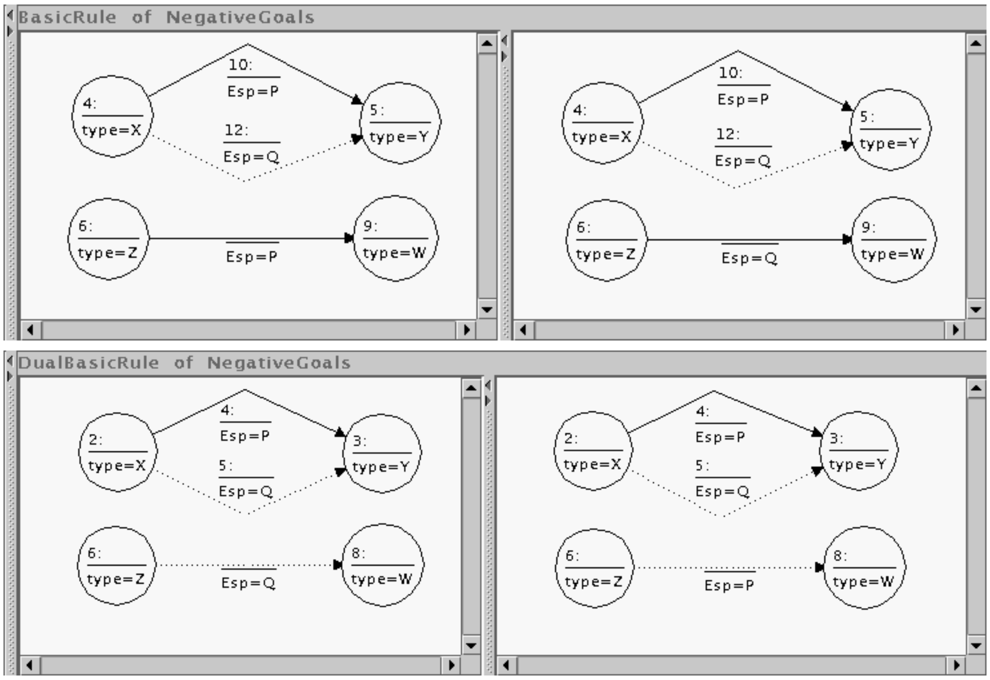


Fig. 19: Basic inference rules for ¬∃∃-goals

applicability condition is evaluated by exploiting a method having this prototype:

```
public boolean isPuniNotP(String)
```

The association of such a method to the rule is done, as mentioned, by means of the attribute context editor (cf. Fig.22).

A simple example

In this section we describe a simple example of assisted reasoning. More precisely, we prove that for any map expression N the following holds:

$$\mathsf{Func}(N^{\smile}) \;\rightarrow\; \mathbb{1}\,;N - N = \overline{\iota}\,;N.$$

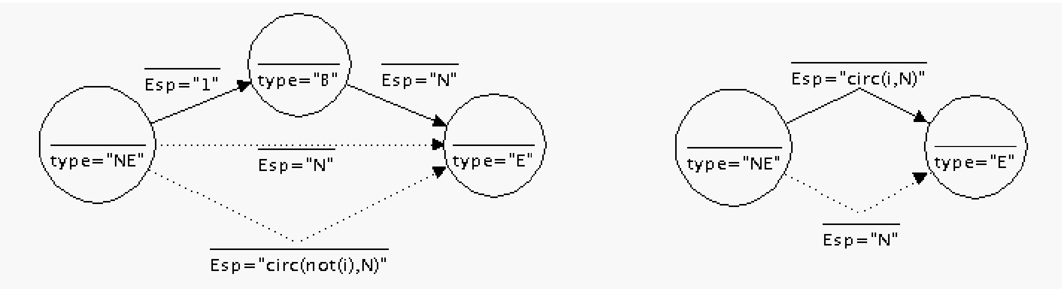
We proceed stepwise by reporting the inferences performed by AGG. We split our task into two parts:

(a) prove that $\mathbb{1}\,;N - N \subseteq \overline{\iota}\,;N$;

(b) prove that $\mathsf{Func}(N^{\smile}) \rightarrow \overline{\iota}\,;N \subseteq \mathbb{1}\,;N - N$.

Let us show the proof of both theses. Notice that the former thesis does not require $N^{\smile}$ to be a function.

(a) The representation of $\mathbb{1}\,;N - N \subseteq \overline{\iota}\,;N$ as a ¬∃∃-graph is:



(1)

(2)

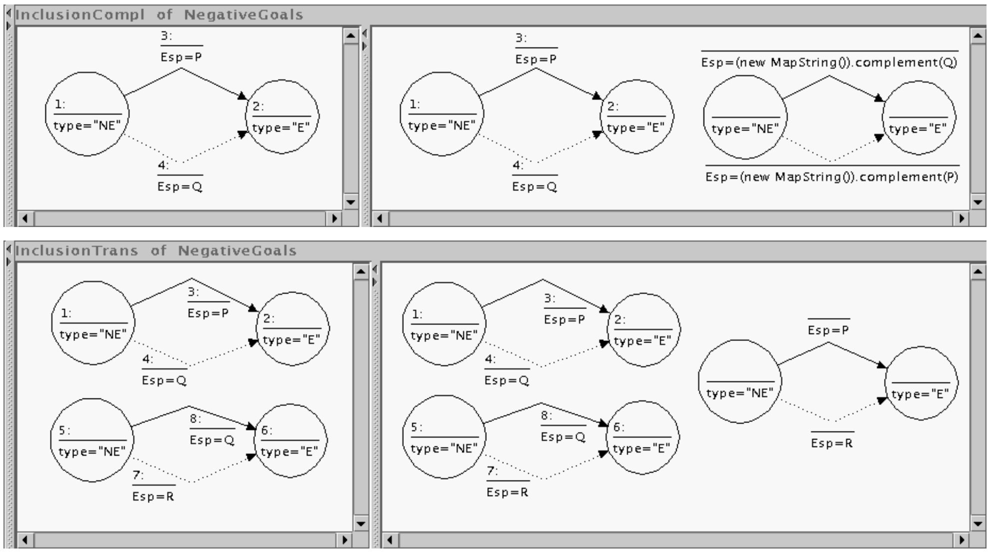
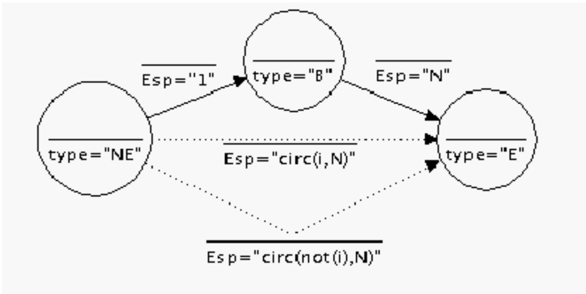
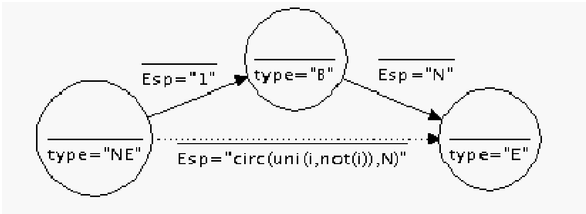


Fig. 20: Simple inference rules for map inclusion

On the right we have added an instance of the axiom (A_6) since the first step will consist in applying rule (2) of Fig.19. As a result we obtain the graph:



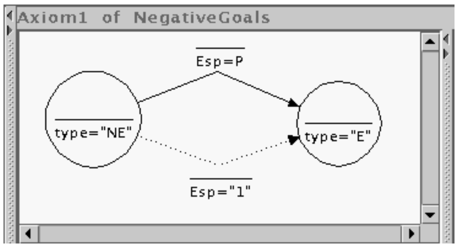
which can be rewritten as follows by applying the rule of Fig.9:



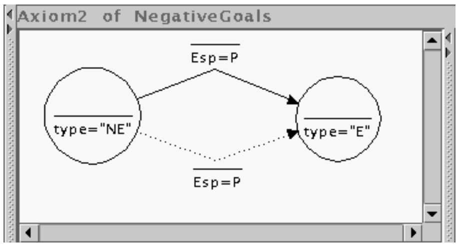
by (1) of Fig.23 AGG reduces this goal into the two goals:

these two goals are then immediately proved by using (A_5) and (A_2), respectively. This completes the first part of the proof.

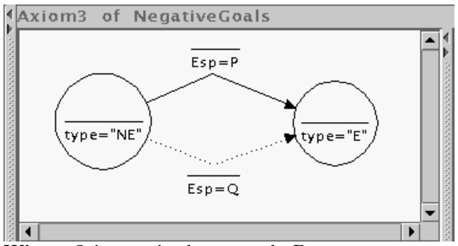
(A_1)



(A_2)

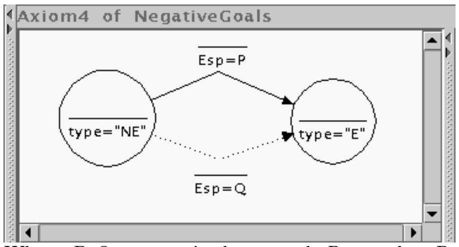


(A_3)



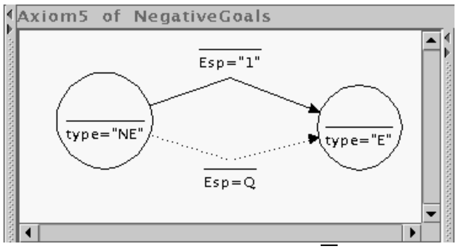
Where Q is required to match $P\,;\iota$.

(A_4)



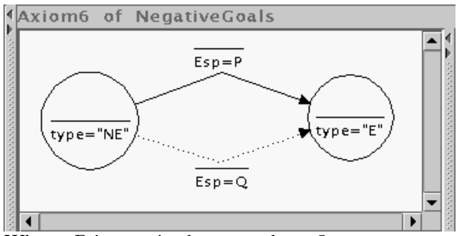
Where P, Q are required to match $R\,;\iota$ and $\iota\,;R$, respectively, for some R.

(A_5)



Where Q is required to match $P \cup \overline{P}$, for some P.

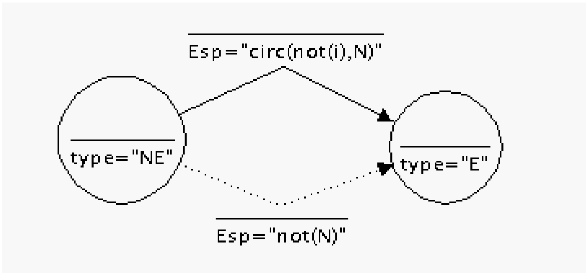
(A_6)



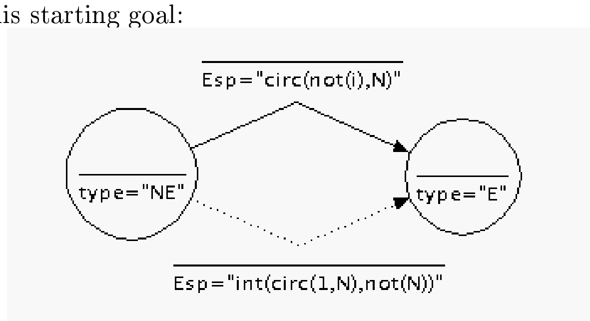
Where P is required to match $\iota\,;Q$.

Fig. 21: Axioms on map inclusion

(b) By definition, $\mathsf{Func}(\,N^{\smile}\,)$ can be rewriten as $\overline{\iota}\,;N \subseteq \overline{N}$, i.e.:



Our thesis originates this starting goal:



by applying an instance of rule (2) of Fig.23 —obtained identifying P and Q in (2)—, Agg reduces the thesis to two subgoals:

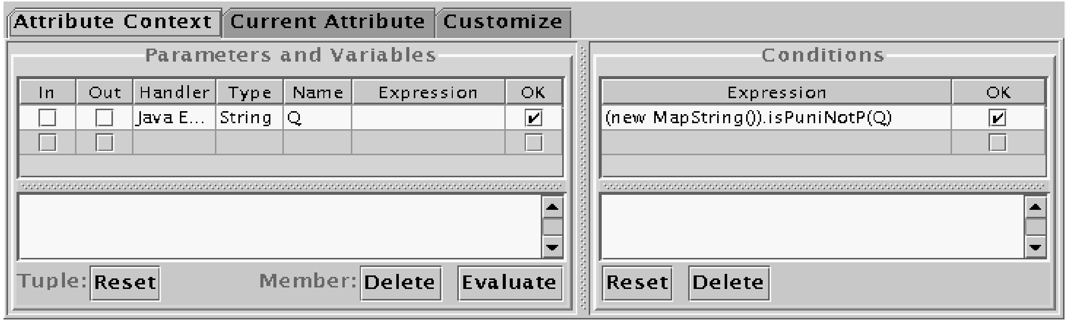
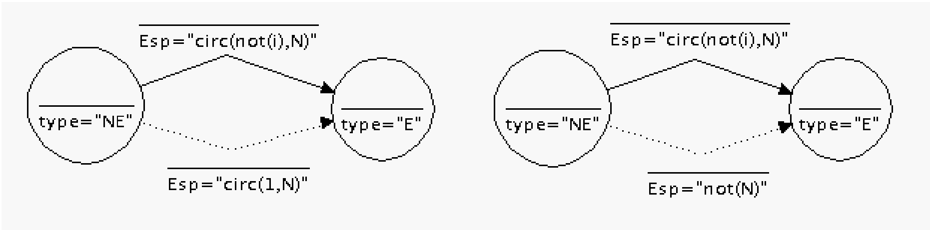
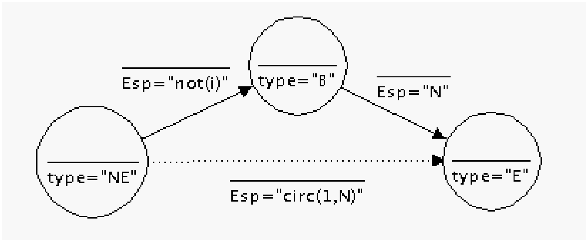


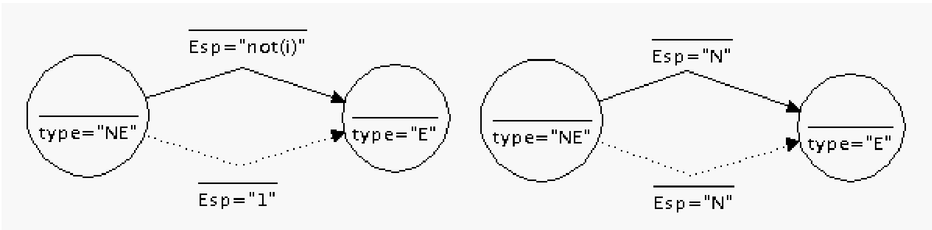
Fig. 22: The attribute context window for axiom (A_5)



The goal on the right corresponds to our hypothesis Func($N^{\smile}$); the other one can be further reduced as follows

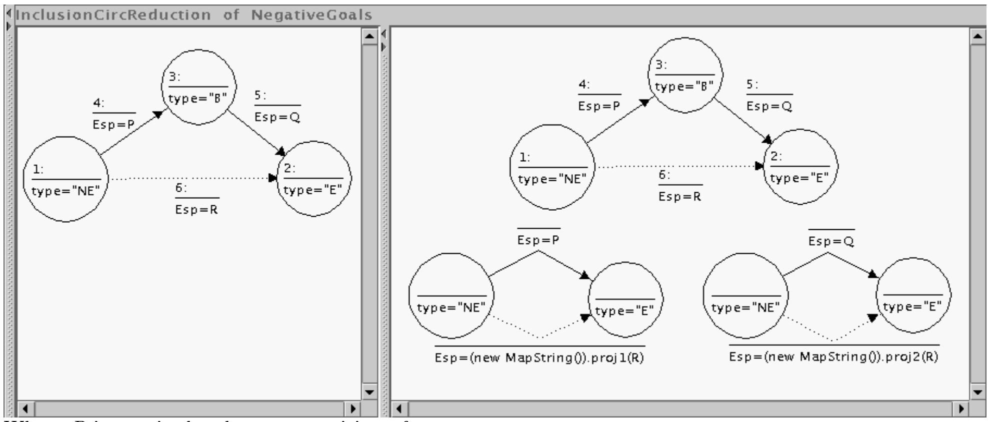


by applying the 'compound_composition' rule of page 10. A further step reduces the remaining goal by means of (1) of Fig.23, yielding the two goals:



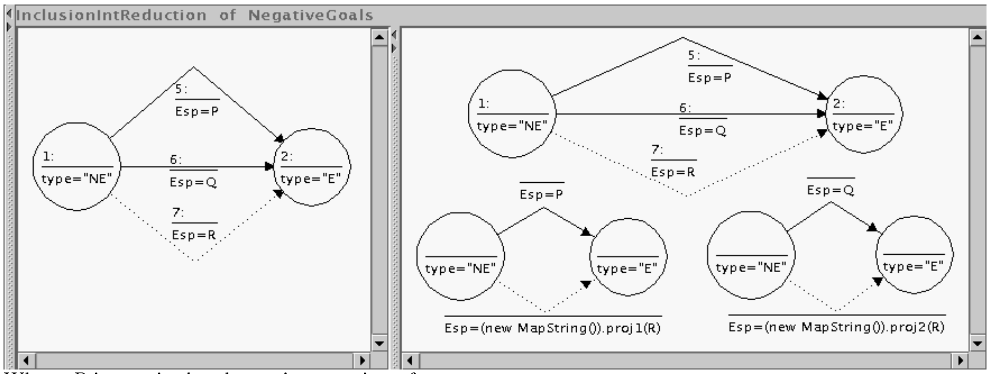
which are instances of (A_1) and (A_2), respectively. This completes the proof.

(1)



Where R is required to be a composition of maps.

(2)



Where R is required to be an intersection of maps.

Fig. 23: Simple reduction rules for $\neg\exists\exists$-goals involving map inclusion

5 Related work

Several approaches to the automation of map-reasoning have been proposed. Various tools supporting algebraic logic exist already. We would like to mention, at least, RALF, Libra, and RELVIEW. RALF is basically a graphical interactive proof assistant and proof checker: it allows the user to manipulate relation-algebraic formulas mainly by using substitution of equals for equals, weakening and strengthening (cf. [28]). The RELVIEW system (cf. [2]) offers a support for relational computation: assuming finiteness of domains and relations, it offers explicit and extensional representation of concrete relations and provides efficient implementation of the basic relational constructs. The Libra language (Lazy Interpreter of Binary Relational Algebra) is a general-purpose programming language based on the algebra of dyadic relations that offers immediate support to program specification. A portable implementation (in Prolog) of a lazy evaluator of Libra programs is illustrated is [15].

A few graphical approaches to relational calculus have been proposed too. For instance, relational methods are exploited in [5, 6] to tame the problem of circuit design. This goal is achieved by developing a pictorial representation of relational terms and by providing a (relational) semantics for pictures. High-level operations on pictures/circuits are rendered by transformation rules that ultimately correspond to the axioms/laws of the calculus.

In [26], a graphical representation by means of *diagrams* is proposed for all term-expressions of an equational theory of binary relations which does not involve complementation. A notion of reduction is given in terms of morphisms between diagrams. Normalization and decidability properties for this graphical framework are also provided.

Another graphical calculus for representation of and visual-reasoning on mathematical formulas is proposed in [13]. In this approach the treatment of $\cap$ and **;** essentially coincides with ours; moreover, [13] introduces a set of graphical transformation rules which turns out to have a large overlap with the rules exploited in our Graph-fattening algorithm; however, [13] does not deal explicitly with complementation and adopts a different treatment of map inclusion.

The work of Kahl (cf. [30, 31]) provides a more general approach to the graphical calculi of relations introduced by [5] and [13], by resorting to algebraic graph-rewriting techniques and concepts [12, 16, 34].

Conclusion

In this paper we reported on a first attempt in implementing graphical techniques for map transformation and map reasoning.

Some conservative extensions of a basic translator have been proposed, nevertheless, the design of further extensions which exploit semantical knowledge can be considered as an interesting stream of research. A challenging problem consists in giving a characterization of the classes of formulas that are translatable by these techniques. Another problem that has not been addressed here consists in the design and implementation of a more sophisticated proof-assistant for graphical map-reasoning: consider for instance the capability of performing backtracking or to suggesting the user the 'better' transformation rule to apply. The implementation of these features could be considered as a first step toward the realization of a tool for automated map-reasoning based on graph-transformation techniques.

Recently, several improvements and new features have been studied and developed for the AGG system. Among them, we mention the implementation of a parser for graph grammars. These new features can reasonably be considered as a suitable basis on which an automated theorem prover could be built up.

Acknowledgements

We are grateful to Olga Runge, Gabriele Taentzer, and Thorsten Schultzke for useful discussions and suggestions about the AGG system. We would like to thank Andrea Corradini, Wolfram Kahl, James Lipton, and Antonino Salibra for giving us many suggestions and useful references.

References

[1] F. Aureli, A. Formisano, E. G. Omodeo, and M. Temperini. Map calculus: Initial application scenarios and experiments based on Otter. Rep. 466, IASI-CNR, 1998.

[2] R. Behnke, R. Berghammer, and P. Schneider. Machine support of relational computations: The Kiel RELVIEW system. Technical Report Bericht Nr. 9711, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Kiel, Germany, 1997.

[3] J. G. F. Belinfante. On a modification of Gödel's algorithm for class formation. *AAR Newsletter*, 34:10–15, 1996.

[4] J. G. F. Belinfante. Computer proofs in Gödel's class theory with equational definitions for composite and cross. *J. of Automated Reasoning*, 22(3):311–339, 1999.

[5] C. Brown and G. Hutton *Categories, Allegories and Circuit Design*. In Proc. of LICS94, pp.372–381, IEEE Computer Society Press.

[6] C. Brown and A. Jeffrey *Allegories of Circuits*. Proc. Logic For Computer Science, Springer, 1994.

[7] D. Cantone, A. Cavarra, and E. G. Omodeo. On existentially quantified conjunctions of atomic formulae of $\mathcal{L}^+$. In M. P. Bonacina and U. Furbach, eds, *Proc. of the FTP97 Int. workshop on first-order theorem proving*, RISC-Linz Rep. No.97-50:45–52, 1997.

[8] D. Cantone, A. Formisano, E. G. Omodeo, and C. G. Zarba. Compiling dyadic first-order specifications. *Proceedings of the 16th Twente Workshop on Language Technology – 2nd AMAST Workshop Algebraic Methods in Language Processing (AMILP 2000).* Iowa City, Iowa, USA, May 2000.

[9] A. Chiacchiaretta, A. Formisano, and E. G. Omodeo. Benchmark #1 for equational set theory. *Giornata "Analisi Sperimentale di Algoritmi per l'Intelligenza Artificiale"* (Rome, 16, December 1999). (`http://www.dis.uniroma1.it/~rcra/roma99`).

[10] A. Chiacchiaretta, A. Formisano, and E. G. Omodeo. Map reasoning through existential multigraphs. Report 05/00, Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, April 2000.

[11] L. H. Chin and A. Tarski. Distributive and modular laws in the arithmetic of relation algebras. *University of California Publications in Mathematics*, 1(9):341–384, 1951.

[12] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In [39].

[13] S. Curtis and G. Lowe. Proofs with graphs. *Science of Computer Programming*, 26(1–3):197–216, 1996.

[14] I. Düntsch. A logic for rough sets. *Theoretical Computer Science*, 179(1–2):427–436, 1997.

[15] B. Dwyer. LIBRA: a Lazy Interpreter of Binary Relational Algebra. Technical Report 95-10, Department of Computer Science University of Adelaide, Australia, 1995.

[16] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformations II: Single pushout approach and comparison with double pushout approach. In [39].

[17] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1 – Equations and initial semantics.* Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1985.

[18] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[19] C. Ermel, M. Rudolf, and G. Taentzer. The AGG Approach: Language and Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, eds., *Handbook of Graph Grammars and Computing by Graph Transformations, vol. 2: Applications, Languages and Tools.* World Scientific, 1999.

[20] A. Formisano, E. G. Omodeo, and M. Temperini. Instructing equational set-reasoning with Otter. To appear.

[21] A. Formisano and E. G. Omodeo. An equational re-engineering of set theories. In Caferra and Salzer eds., *Automated Deduction in Classical and Non-Classical Logics*, LNCS 1761 (LNAI), pages 176–191. Springer, 2000.

[22] A. Formisano, E. G. Omodeo, and M. Temperini. Goals and benchmarks for automated map reasoning. In M.-P. Bonacina and U. Furbach, eds., *Advances in First-order Theorem Proving.* Special issue of *J. Symb. Comp.*, 29(2):259–297, 2000.

[23] M. F. Frias, A. M. Haeberer, and P. A. S. Veloso. A finite axiomatization for fork algebras. *Journal of the IGPL*, 5(3):311–319, 1997.

[24] K. Gödel. *The Consistency of the Axiom of Choice and of the Generalized Continuum-Hypothesis with the Axioms of Set Theory.* Princeton University Press, 1940. Also in [25].

[25] K. Gödel. *Collected Works. Volume II. Publications 1938-1974.* Oxford University Press, New York - Oxford, 1990. Solomon Feferman (Editor-in-chief).

[26] C. Gutierrez and D. Dougherty. Normal forms and reduction for theories of binary relations. In L. Bachmair ed., *Proc. of RTA2000*, LNCS 1833, pp. 95–109. Springer-Verlag, 2000.

[27] C. Hattensperger. RALF - An interactive proof system for relationalgebra. Fakultät für Informatik, Universität der Bundeswehr, München, Germany, 1998.

[28] C. Hattensperger, R. Berghammer, and G. Schmidt. Ralf - a relation-algebraic formula manipulation system and proof. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Proc. of AMAST93.* Springer-Verlag, 1993.

[29] T. J. Jech *Set theory*, Academic Press, New York, 1978.

[30] W. Kahl. Algebraic Graph Derivations for Graphical Calculi. In F. d'Amore, P. Franciosa, and A. Marchetti-Spaccamela, eds., Graph Theoretic Concepts in Computer Science, *Proc. of WG96*, LNCS 1197, pp.244–238, 1996.

[31] W. Kahl. Relational Matching for Graphical Calculi of Relations. *Information Sciences*, 119(3/4):253–273, 1999.

[32] J.-L. Krivine. *Introduction to axiomatic set theory.* Reidel, Dordrecht, 1971.

[33] M. K. Kwatinetz. *Problems of expressibility in finite languages.* PhD thesis, University of California, Berkeley, 1981.

[34] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.

[35] R. D. Maddux. The origin of relation algebras in the development and axiomatization of the calculus of relations. *Studia Logica*, 50(3/4):421–455, 1991.

[36] R. D. Maddux. A perspective on the theory of relation algebra. *Algebra Universalis*, 31:456–465, 1994.

[37] R. D. Maddux. Relation-algebraic semantics. *TCS*, 160:1–85, 1996.

[38] E. Orlowska. Relational semantics for nonclassical logics: Formulas are relations. In J. Wolenski, editor, *Philosophical Logic in Polland*, pages 167–186. , 1994.

[39] G. Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformations*, vol. 1: Foundations. World Scientific, 1998.

[40] G. Schmidt and T. Ströhlein. Relation algebras: Concept of points and representability. *Discrete Mathematics*, 54:83–92, 1985.

[41] G. Schmidt and T. Ströhlein. *Relations and graphs.* Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1993.

[42] E. Schröder. *Vorlesungen über die Algebra der Logik (exakte Logik).* B. Teubner, Leipzig, 1891-1895. [Reprinted by Chelsea Publishing Co., New York, 1966.]

[43] J. T. Schwartz, R. K. B. Dewar, E: Dubinsky, and E. Schonberg. *Programming with Sets: An introduction to SETL.* Texts and Monographs in Computer Science, Springer-Verlag, 1986.

[44] J. C. Shepherdson. Negation as failure: A comparison of Clark's completed database and Reiter's closed world assumption. *J. of Logic Programming*, 1(1):51–79, 1984.

[45] A. Tarski and S. Givant. *A formalization of set theory without variables.* Vol. 41 of *Colloquium Publications.* AMS, 1987.

A Java code

Class MapTree

```
import java.util.Enumeration;
import java.awt.*;
import java.io.*;

public class MapTree implements Serializable {

    private String str;
    private String exp;
    private MapTree father;
    private MapTree left;
    private MapTree right;


    public MapTree(String aStr) {
        father = null;
        exp = aStr;
        left = null;
        right = null;
    }

    public void add_sx(MapTree node) {
        node.set_father(this);
        this.left = node;
    }

    public void add_dx(MapTree node) {
        node.set_father(this);
        this.right = node;
    }

    public void set_father(MapTree newfather) {
        this.father = newfather;
    }

    public void set_exp(String newexp) {
        this.exp = newexp;
    }

    public MapTree get_father() {
        return father;
    }

    public MapTree get_sx() {
        return left;
    }

    public MapTree get_dx() {
        return right;
    }

    public String get_exp() {
        return exp;
    }

    public boolean is_leaf() {
        return (right == null) && (left == null);
    }
```

```
public MapTree mv_inv_inward_in_tree(int numinv) {
String s = "", w = "";
MapTree resul = null;
MapTree resulsx = null;
MapTree resuldx = null;

    w = this.get_exp();
    if (this.is_leaf()) {
       if ((numinv % 2) == 0) {
           resul = this;
       }
       else {
           resul = new MapTree("inv");
           resul.add_sx(this);
       }
    }
    else if (w.equals("uni") || w.equals("int") || w.equals("symm")) {
        resulsx =  (this.get_sx()).mv_inv_inward_in_tree(numinv);
        resuldx =  (this.get_dx()).mv_inv_inward_in_tree(numinv);
        this.add_sx(resulsx);
        this.add_dx(resuldx);
        resul = this;
    }
    else if (w.equals("circ")) {
        resulsx =  (this.get_sx()).mv_inv_inward_in_tree(numinv);
        resuldx =  (this.get_dx()).mv_inv_inward_in_tree(numinv);
        this.add_dx(resulsx);
        this.add_sx(resuldx);
        resul = this;
    }
    else if (w.equals("not")) {
        resulsx =  (this.get_sx()).mv_inv_inward_in_tree(numinv);
        this.add_sx(resulsx);
        resul = this;
    }
    else if  (w.equals("inv")) {
        resulsx =  (this.get_sx()).mv_inv_inward_in_tree(numinv+1);
        resul = resulsx;
    }
    return resul;
}



public String tree_to_string() {
String s = "", w = "";

    w = this.get_exp();
    if (this.is_leaf()) {
       s = w;
    }
    else if (w.equals("uni") || w.equals("int") || w.equals("circ") ||
        w.equals("symm")) {
        s = s + w + "(" + left.tree_to_string() +  "," +
        right.tree_to_string() + ")";
    }
    else if (w.equals("not") || w.equals("inv")) {
        s = s + w + "(" + left.tree_to_string() + ")";
    }
    return s;
}
```

```
    public MapTree string_to_tree(String s) {
    String s1 = "", s2 = "", s3 = "";
    int j1,z, begin1;

        s = s.trim();
        j1 = s.indexOf("(");
        if (j1 == -1)
            this.set_exp(s);
        else {
             s1 = s.substring(0, j1);
             begin1 = j1 + 1;
             if (s1.equals("uni") || s1.equals("int") || s1.equals("circ") ||
                 s1.equals("symm")) {
                 z = 0;
                 while ((s.charAt(j1) != ',') || (z != 0)) {
                    j1++;
                    if (s.charAt(j1) == '(') z++;
                    if (s.charAt(j1) == ')') z--;
                 }
                 s2 = s.substring(begin1, j1);
                 s3 = s.substring(j1 + 1, s.lastIndexOf(')'));
                 this.set_exp(s1);
                 MapTree x = new MapTree("");
                 MapTree y = new MapTree("");
                 x.string_to_tree(s2);
                 y.string_to_tree(s3);
                 this.add_sx(x);
                 this.add_dx(y);
            }
            else if (s1.equals("not") || s1.equals("inv")) {
                s2 = s.substring(begin1, s.lastIndexOf(')'));
                this.set_exp(s1);
                MapTree x = new MapTree("");
                x.string_to_tree(s2);
                this.add_sx(x);
            }
            else {
                System.out.println("opss..! " + s + " ---" + s1 + "--- "  );
                this.set_exp("ops");
            }
        }
        return this;
        }
}
```

Class MapString

```
import java.util.Enumeration;
import java.awt.*;
import java.io.*;
import MapTree;

public class MapString implements Serializable {
    private String str;

    public MapString() {
        this.str = "";
    }

    public String get_string() {
        return this.str;
    }

    public void set_string(String s) {
        this.str = s;
    }

    public String bypass(String sp, String sq, String sr) {

        MapTree root1 = (new MapTree("int"));
        MapTree root2 = (new MapTree("circ"));
        root2.add_sx(((new MapTree("")).string_to_tree(sp)));
        root2.add_dx(((new MapTree("")).string_to_tree(sq)));
        root1.add_sx(root2);
        root1.add_dx(((new MapTree("")).string_to_tree(sr)));
        this.str = root1.tree_to_string();
        return this.str;
    }

    public String invPbypass(String sp, String sq, String sr) {

        this.bypass(this.invert(sp),sq,sr);
        return this.str;
    }

    public String invQbypass(String sp, String sq, String sr) {

        this.bypass(sp,this.invert(sq),sr);
        return this.str;
    }

    public String invRbypass(String sp, String sq, String sr) {

        this.bypass(sp,sq,this.invert(sr));
        return this.str;
    }

    public String simpleBypass(String sp, String sq) {

        MapTree root2 = (new MapTree("circ"));
        root2.add_sx(((new MapTree("")).string_to_tree(sp)));
        root2.add_dx(((new MapTree("")).string_to_tree(sq)));
        this.str = root2.tree_to_string();
        return this.str;
    }

    public String invPsimpleBypass(String sp, String sq) {

        this.simpleBypass(this.invert(sp),sq);
        return this.str;
    }
```

```
public String invQsimpleBypass(String sp, String sq) {

    this.simpleBypass(sp, this.invert(sq));
    return this.str;
}

public String mergeEdges(String sp, String sq) {

    MapTree root2 = (new MapTree("int"));
    root2.add_sx(((new MapTree("")).string_to_tree(sp)));
    root2.add_dx(((new MapTree("")).string_to_tree(sq)));
    this.str = root2.tree_to_string();
    return this.str;
}

public String invPmergeEdges(String sp, String sq) {

    this.mergeEdges(this.invert(sp),sq);
    return this.str;
}

public String bigamy (String sq, String sr) {

    this.bypass("1",sq,sr);
    return this.str;
}

public String revbigamy (String sp, String sr) {

    this.bypass(sp,"1",sr);
    return this.str;
}

public String invPrevbigamy (String sp, String sr) {

    this.bypass(this.invert(sp),"1",sr);
    return this.str;
}

public String invRrevbigamy (String sp, String sr) {

    this.bypass(sp,"1",this.invert(sr));
    return this.str;
}

public String invertMap (String sp) {

    sp = sp.trim();
    MapTree root2 = (new MapTree("inv"));
    root2.add_sx(((new MapTree("")).string_to_tree(sp)));
    root2 = root2.mv_inv_inward_in_tree(0);
    this.str = root2.tree_to_string();
    return this.str;
}

public String invert (String sp) {

    sp = sp.trim();
    if (sp.startsWith("inv(")) {
        sp = sp.substring(4,(sp.length() - 1));
    } else {
        sp = "inv(" + sp + ")";
    }
    this.str =sp;
    return this.str;
}
```

```
public String complement (String sp) {

    sp = sp.trim();
    if (sp.startsWith("not(")) {
        sp = sp.substring(4,(sp.length() - 1));
    } else {
        sp = "not(" + sp + ")";
    }
    this.str =sp;
    return this.str;
}


public String int_iota (String sp) {

    sp = sp.trim();
    sp = "int(" + sp + ",i)";
    this.str = sp;
    return this.str;
}


public String genDeMorgan (String sr, String ss) {

    MapTree tt = (new MapTree("")).string_to_tree(sr);
    tt = tt.get_dx();
    MapTree tp = (new MapTree("")).string_to_tree(sr);
    tp = tp.get_sx();
    MapTree tq = (new MapTree("")).string_to_tree(ss);
    tq = tq.get_sx();
    MapTree rootuni = (new MapTree("uni"));
    rootuni.add_sx(tp);
    rootuni.add_dx(tq);
    MapTree rootcirc = (new MapTree("circ"));
    rootcirc.add_sx(rootuni);
    rootcirc.add_dx(tt);
    this.str=(rootcirc.tree_to_string());
    return this.str;
}



public boolean isPuniNotP(String sq) {
    String wt = "";
    String wr = "";
    String expr = "";
    String exprnot = "";

    sq = sq.trim();
    MapTree r = (new MapTree("")).string_to_tree(sq);
    expr = r.get_exp();
    r = r.get_sx();
    wr = r.tree_to_string();
    r = (new MapTree("")).string_to_tree(sq);
    r = r.get_dx();
    exprnot = r.get_exp();
    r = r.get_sx();
    wt = r.tree_to_string();
    if ((wr.equals(wt)) && (expr.equals("uni")) && (exprnot.equals("not")))
        return true;
    else
        return false;
 }
```

```
public boolean PcircIotaANDIotacircP(String sp, String sq) {
    String wt = "";
    String wr = "";
    String io1 = "";
    String io2 = "";

    sp = sp.trim();
    sq = sq.trim();
    MapTree t = (new MapTree("")).string_to_tree(sp);
    t = t.get_sx();
    MapTree r = (new MapTree("")).string_to_tree(sq);
    r = r.get_dx();
    wt = t.tree_to_string();
    wr = r.tree_to_string();
    t = (new MapTree("")).string_to_tree(sp);
    t = t.get_dx();
    r = (new MapTree("")).string_to_tree(sq);
    r = r.get_sx();
    io1 = t.tree_to_string();
    io2 = r.tree_to_string();
    if ((wr.equals(wt)) && (io1.equals("i")) && (io2.equals("i")))
        return true;
    else
        return false;
 }

public boolean QIsIotacircP(String sp, String sq) {

    sp = sp.trim();
    sq = sq.trim();
    if (sq.equals(("circ(i,"+sp)))
        return true;
    else
        return false;
 }

public boolean QIsPcircIota(String sp, String sq) {

    sp = sp.trim();
    sq = sq.trim();
    if (sq.equals(("circ("+sp+",i)")))
        return true;
    else
        return false;
 }


public boolean test_DeMorgan(String sr, String ss) {
    String wr = "";
    String ws = "";

    MapString w = (new MapString());
    if ((w.test_appl(sr,"circ")) && (w.test_appl(ss,"circ"))) {
        MapTree tr = (new MapTree("")).string_to_tree(sr);
        tr = tr.get_dx();
        wr = tr.tree_to_string();
        MapTree ts = (new MapTree("")).string_to_tree(ss);
        ts = ts.get_dx();
        ws = ts.tree_to_string();
        if (ws.equals(wr))
            return true;
        else
            return false;
    }
    else return false;
 }
```

```
    public boolean test_appl(String sp, String cond) {
        String w = "";

        sp = sp.trim();
        MapTree t = (new MapTree("")).string_to_tree(sp);
        w = t.get_exp();
        if (w.equals(cond))
            return true;
        else
            return false;
     }


    public String proj1 (String sp) {

        sp = sp.trim();
        MapTree t = (new MapTree("")).string_to_tree(sp);
        t = t.get_sx();
        this.str = t.tree_to_string();
        return this.str;

    }

    public String proj2 (String sp) {

        sp = sp.trim();
        MapTree t = (new MapTree("")).string_to_tree(sp);
        t = t.get_dx();
        this.str = t.tree_to_string();
        return this.str;

    }

}
```