

# A strategy language for graph rewriting systems\*

Maribel Fernández and Olivier Namet  
King's College London, Department of Computer Science  
Strand, London WC2R 2LS, U.K.  
Maribel.Fernandez,Olivier.Namet@kcl.ac.uk

## Abstract

We design a language that can be used to control the application of graph rewriting rules. The traditional operators found in strategy languages for term rewriting have been adapted to deal with the more general setting of graph rewriting, and some new constructs have been included to deal with graph traversal and management of rewriting positions in the graph. This language is part of a graph transformation environment that is currently being built within the PORGY project.

## 1 Introduction

Rewriting is a computational model used in computer science, algebra, logic and linguistics, amongst others. Its purpose is to transform syntactic objects (words, terms,  $\lambda$ -terms, programs, proofs, graphs, ...) by applying rules. Rewriting is typically used to simplify an algebraic expression (in computer algebra), to express the grammar of a programming language or a natural language (syntactic analysis), to define the operational semantics of a programming language, to study the structure of a groupe or a monid (combinatorics algebra), or to express the computational content of a mathematical proof (proof theory). Other practical applications can be mentioned, such as handling headers in the electronic mails in the *sendmail* program, program refactoring, code optimisation in compilers or the modelling of complex systems in biology.

Strategic rewriting has been studied for term rewriting systems, and there are languages that allow the user to specify a strategy controlling the use of rewrite rules and to apply it [4, 8]. In this paper we will define a language to define strategies for graph rewriting systems, where not only the strategy needs to take into account rules and sequences of rules but also location and propagation in a graph (the latter is complicated by the fact that in a graph there is no notion of a root, so standard term rewriting strategies based on top-down or bottom-up traversals do not make sense in this setting). Because of this, we develop a specific language to deal with strategies for graph rewriting systems, which can be also applied to port graph rules [1] and interaction net systems [6] as particular cases.

## 2 Background: Graph Rewriting

In a more formal way, a graph rewrite system is a set of rewrite rules of the form  $L \rightarrow R$  where  $L$  and  $R$  are graphs. Such a rule applies to a graph  $G$  if  $G$  contains at least one instance of the left-hand side  $L$ , i.e. a subgraph  $A$  isomorphic to  $L$ . Each rule specifies an interface that is used in order to rewrite  $A$  in  $G$ :  $G$  rewrites to a new graph  $G'$  obtained by replacing the instance  $A$  of  $L$  by an instance  $B$  of  $R$ , where edges that were connected to nodes in  $A$  are connected to  $B$  as specified by the rule's interface. This rewriting process induces a transitive relation on graphs.

---

\*Research partially supported by the PORGY project (INRIA Bordeaux-Sud-Ouest and King's College London).

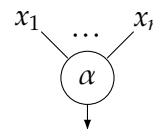
Each rule application is a rewriting step and a derivation is a sequence of rewriting steps, that we will sometimes call a computation, referring to application of rewriting to programming languages. A graph on which no rule is applicable is said to be in *normal form*.

There are several formal definitions of graph rewriting systems, depending on the kind of graph that is rewritten and the specific format of the rules that can be defined in the rewrite system (see, for instance, [2, 7, 3, 6]).

A graph rewriting system may have a potentially large set of rules to apply to a graph. The order in which rules are applied can greatly alter the end graph when general graph rewriting is considered. For this reason, as in the case of term rewriting, a strategy language is needed to control the way in which rules are applied in order to build derivations. Several strategy languages have been defined for term rewriting systems, see for instance ELAN [4] and Stratego [8].

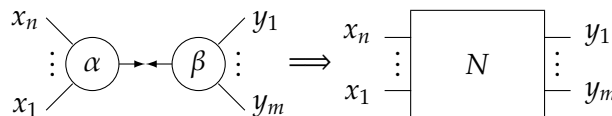
As a particular example of graph rewriting formalism of interest, in this paper we consider interaction nets, introduced by Lafont in [6].

A system of *interaction nets* is specified by a set  $\Sigma$  of symbols with fixed arities, and a set  $\mathcal{R}$  of interaction rules. An occurrence of a symbol  $\alpha \in \Sigma$  is called an *agent*. If the arity of  $\alpha$  is  $n$ , then the agent has  $n + 1$  ports: a *principal port* depicted by an arrow, and  $n$  *auxiliary ports*. Such an agent will be drawn in the following way:



Intuitively, a net  $N$  is a graph (not necessarily connected) with agents at the vertices and each edge connecting at most 2 ports. The ports that are not connected to another agent are *free*. There are two special instances of a net: a wiring (no agents) and the empty net; the extremes of wirings are also called free ports. The *interface* of a net is its set of free ports.

An interaction rule  $((\alpha, \beta) \Rightarrow N) \in \mathcal{R}$  replaces a pair of agents  $(\alpha, \beta) \in \Sigma \times \Sigma$  connected together on their principal ports (an *active pair* or *redex*, written  $\alpha \bowtie \beta$ ) by a net  $N$  with the same interface. Rules must satisfy two conditions: all free ports are preserved during reduction (reduction is local, i.e. only the part of the net corresponding to the redex is modified, no global modifications are required), and there is at most one rule for each pair of agents. Because of this last restriction, a rule is fully determined by its left hand-side; such a rule will thus be sometimes denoted by  $\alpha \bowtie \beta$  as well. The following diagram shows the format of interaction rules ( $N$  can be any net built from  $\Sigma$ ).



The key property of interaction nets, besides locality of reduction, is that reduction is strongly confluent. We refer the reader to [6] for more details and examples.

Although the strong confluence property of interaction nets ensures that all reduction sequences to full normal form are equivalent, this is not the case if we use a notion of reduction that does not reach a full normal form. For instance, reduction to interface normal form [5] requires to transform an interaction net into a net whose interface is stable, that is, no future application of a rule can change the agents in the interface. This notion is the analogous to a root-stable term or a weak-head normal form in the  $\lambda$ -calculus. Users may therefore want to not just blindly apply rules but to create a strategy around these rules to control the rewriting process and ensure that rules are applied in an efficient way.

### 3 Strategy language

We will use a strategy language where expressions will include operators to select rules and the position where the rules will be applied, and also operators to navigate in a given graph (i.e., to change the position where the rewriting rules will be applied). Given a set of graph rewriting rules and an initial graph, we will generate a derivation by using a strategy expression. Below we describe our strategy language and give examples of derivations.

#### 3.1 Syntax & Grammar

As discussed above, strategies for graph rewriting need to specify the way rules will be applied and the position (or location) in the graph where the rules will apply.

We will denote by  $G$  the graph to be rewritten, and by  $P$  a subgraph of  $G$  which specifies the location in  $G$  where rewrite rules will apply.

We define the following grammars  $S$ ,  $T$ ,  $A$  and  $E$  to generate strategy expressions, location transformations, rule applications, and base results, respectively:

- **Strategy:**

$$S := S;S \mid \text{while}(S = E)\text{do}(S)\text{min}(m)\text{max}(n) \mid S + S \mid \text{if}(S = E)\text{then}(S)\text{else}(S) \mid \text{pick}(S,S) \mid \text{PnotEmpty} \mid < S > \mid T \mid A$$

- **Transformation:**

$$T := \text{Id} \mid \text{current} \mid \text{Int} \mid \text{All} \mid \text{One} \mid \text{Next} \mid \text{Explicit}(P) \mid \text{property}(X,Y) \mid \cup \mid \cap \mid \bar{T} \mid T_1 - T_2$$

Applying  $T$  on  $G[P]$  gives us  $G[P']$ : a transformation only affects the subgraph  $P$  of  $G$ .

- **Application:**

$$A := \text{Id} \mid \text{Fail} \mid (L \rightarrow R)_M \mid A\|A \mid A\|\|A \mid A^{(m,n)}$$

Applying  $A$  on  $G[P]$  gives us  $G'[P']$ : an application may change the graph  $G$  and also its subgraph  $P$ .

- **Base Type:**

$$E := \text{Id} \mid \text{Fail}$$

$E$  is a subset of  $S$  and is useful for typing purposes in the semantic definitions of 3.2

**Definition 1** (Graph System). *A graph system is a graph with its associated position subgraph.*

**Definition 2** (Strategy System). *A strategy system is a pair of a strategy  $S$  and a graph system  $G[P]$ . We will write a strategy system as  $S, G[P]$ . A grammatically correct terminating strategy system will always rewrite to a base system  $E, G[P]$ , where  $E$  is a base type as defined in the grammar above.*

We describe below the purpose of the different constructs in the language:

- **Transformations:**

$\text{Id}$  is the identity transformation;  $\text{current}$  returns the current position of the graph;  $\text{Int}$  changes  $P$  to become the interface of  $P$  (for instance, in the case of Interaction Nets this is the set of free ports where edges could be attached);  $\text{All}$  returns immediate successors of (non - visited) nodes of  $P$ ;  $\text{One}$  will pick one immediate successor randomly;  $\text{Next}$  is assigned a specific port for each agent type to follow (in the specific case of Interaction

Nets, all agents types would assign their principal port to *Next* ; *Explicit(P')* changes P of G to P'; *property(X,Y)* updates P to only contain all nodes from Y that satisfy the property X (Y would generally be P or G);  $\cup$ ,  $\cap$ ,  $\bar{T}$  and  $T_1 - T_2$  represent the set theory operators *union*, *intersection*, *complement* and *subtraction*.

- **Applications:**

*Id* for applications never fails and leaves the graph unchanged; *Fail* is an application that always fails (it leaves the graph unchanged and returns failure);  $(L \rightarrow R)_M$  represents the application of the rule  $L \rightarrow R$  at the given position  $P$ . For this application to be successful  $L \cap P$  must not be  $\emptyset$ .  $M$  is the subgraph of  $R$  that is added to  $P$  (the rest of  $R$ ,  $\bar{M}$ , is only added to  $G$  and not  $P$ );  $A||A'$  represents simultaneous application of  $A$  and  $A'$  on disjoint redexes and only returns *Id* if both applications are possible;  $A|||A$  is a weaker version of  $A||A$  where as long as at least one of the applications is possible then it will not return *Fail*;  $A^{||m,n}$  applies  $A$  simultaneously a minimum of  $m$  and a maximum of  $n$  times. If the minimum isn't satisfied then *Fail* is returned or else *Id* is returned. To not have a maximum, give  $n$  a negative value.

- **Strategies:**

$S;S'$  represents sequential computations.  $while(S = E)do(S)min(m)max(n)$  keeps on sequentially applying while the expression  $S = E$  still holds. If the minimum isn't satisfied then it returns *Fail* or else *Id* is returned. Like  $A^{||m,n}$ , setting  $n$  to a negative eliminates the maximum.  $S + S'$  applies the strategy that doesn't fail. If both fail then *Fail* is returned and if both are possible then one is picked randomly.  $if(S = E)then(S')else(S'')$  checks if  $S$  returns  $E$  and if so then apply  $S'$  or else apply  $S''$ .  $pick(S,S)$  randomly picks one of the two strategies (a weaker version of  $+$ ).  $PnotEmpty$  returns *Fail* if  $P$  is empty and *True* if not (this is useful to put at the end of a strategy to then check in an *if* or *while* if that strategy makes  $P$  empty or not).  $\langle S \rangle$  applies  $S$  and considers  $S$  as one step in the trace.

$S = E$  will not directly compare  $S$  and  $E$  but will rewrite  $S$  until it reaches a base type (if it terminates at all) and then compare it to  $E$  (it can be read as  $S$  rewrites to  $E$ ).

### 3.2 Semantics

We define the semantics of the strategy constructs (defined by the grammars above) using rewriting rules on strategy systems, which will reduce a strategy system to a base system (if the strategy is terminating). During the reduction process, a subtree of the strategy can be promoted to a strategy system, in order to implement a subreduction on the subtree.

The full system can be found in the appendix.

### 3.3 Examples

- $repeat(S)min(m)max(n)$  is defined as  $while(Id = Id)do(S)min(m)max(n)$
- $repeat_m(S)$  is defined as  $while(Id = Id)do(S)min(m)max(m)$
- $repeat_*(S)$  is defined as  $while(S = Id)do(S)min(-1)max(-1)$
- $repeat_+(S)$  is defined as  $while(S = Id)do(S)min(1)max(-1)$
- $SorelseS'$  is defined as  $if(S = Id)then(S)else(S')$ .

- $Not(S) \rightarrow if(S = Id)then(Fail)else(Id)$
- $Try(S)$  is defined as  $if(S = Id)then(S)else(Id)$ .
- $Try(S_1, S_2, S_3, S_4)$  is defined as  $if(S_1)then(S_1)else(Try(S_2, S_3, S_4))$ . This will try to apply any of the listed strategies from left to right and only returns a *Fail* if none could be applied.
- $repeat_*(R_1; Int)orelseNext(G[Int])$  is a strategy that computes Interface Normal Form for Interaction Nets
- $R_1orelse(Explicit(P'); R_2)$  tries to apply  $R_1$ , if that fails change  $P$  to  $P'$  and apply  $R_2$ .
- $repeat_*((A \rightarrow B) || (C \rightarrow D))$  can be used if  $(A \rightarrow B)$  and  $(C \rightarrow D)$  are two chemical reactions that must happen together.
- $repeat_*(R_1orelseR_2)$  specifies that we apply two rules  $R_1$  and  $R_2$  as many times as possible but always give priority to  $R_1$ .
- $((Int; R_1)orelseR_1)$  Try at the interface of the graph and if and if not possible then try at the current position  $P$ .

## 4 Conclusion and Future Work

The strategy language defined in this paper is part of the PORGY system, which is an environment that allows users to define graphs and graph transformation rules. PORGY is implemented using the TULIP platform, for the visualisation of graphs and graph transformation rules. PORGY provides also tools to visualise traces of rewriting, and the strategy language is used in particular to guide the construction of the traces.

**Acknowledgements** We are grateful to the members of the PORGY team, and in particular to Hélène Kirchner, for many inspiring discussions on the topics of this paper.

## References

- [1] Oana Andrei and Hélène Kirchner. A Rewriting Calculus for Multigraphs with Ports. In *Proceedings of RULE'07*, volume 219 of *Electronic Notes in Theoretical Computer Science*, pages 67–82, 2008.
- [2] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J. R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe*, number 259-II in LNCS, pages 141–158, Eindhoven, The Netherlands, 1987. Springer-Verlag.
- [3] Klaus Barthelmann. How to construct a hyperedge replacement system for a context-free set of hypergraphs. Technical report, Universität Mainz, Institut für Informatik, 1996.
- [4] Peter Borovansky, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An Overview of ELAN. In Hélène Kirchner, Claude & Kirchner, editor, *Second Workshop on Rewriting Logic and its Applications - WRLA'98 Electronic Notes in Theoretical Computer Science*, volume 15 of *Electronic Notes in Theoretical Computer Science*, page 16 p, Pont-à-Mousson, France, 1998. Elsevier Science B. V.
- [5] Maribel Fernández and Ian Mackie. A calculus for interaction nets. In *Proceedings of PPDP'99, Paris*, number 1702 in *Lecture Notes in Computer Science*. Springer, 1999.
- [6] Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990.

- [7] Detlef Plump. Term graph rewriting. Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools, Chapter 1, pages 3-61, eds. H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg. World Scientific, 1998.
- [8] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

## A Semantics of the strategy language

We define the semantics of the strategy constructs in section 3.1 using rewriting rules which will reduce a strategy system to a base system (if the strategy is terminating). During the reduction process, a subtree of the strategy can be promoted to a strategy system, in order to implement a subreduction on the subtree.

Rules are applied outermost and leftmost first, and rules for each operator are used in top-down fashion, so the order in which they are written is important. This allows the use of variables as wildcards, to define *ELSE* cases. We can type variables by naming them as the initial symbol of the corresponding grammar with an index number if needed (for example:  $A_1$  denotes a variable of application type, that is, it represents a rule application or a parallel application of rules;  $S_3$  represents a strategy expression;  $T_2$  represents a transformation...). We assume that two variables of the same type  $V_n$  and  $V_m$  can only be equal if  $m = n$ , if not they represent two *different* variables.

### A.1 Base Cases

$(L \rightarrow R)_{M,G[P]} \rightarrow Id, ((G - L_i) \cup R) [(P - L_i) \cup M]$  (If the rule is applicable. In general, this means that  $L_i$  is a subgraph of  $G$  isomorphic to  $L$ , and  $L_i \cap P \neq \emptyset$ )

$(L \rightarrow R)_{M,G[P]} \rightarrow Fail, G[P]$  (if the rule is *not* applicable)

$T, G[P] \rightarrow Id, G[P']$  (Where  $P'$  is the graph resulting from applying the transformation  $T$  to  $P$  in  $G$ .)

### A.2 Sequence

$\begin{array}{c} ; \\ / \quad \backslash \\ S_1 \quad S_2 \end{array}, G[P] \rightarrow \begin{array}{c} ; \\ / \quad \backslash \\ S_1, G[P] \quad S_2 \end{array}, G[P]$

$\begin{array}{c} ; \\ / \quad \backslash \\ Id, G'[P'] \quad S_2 \end{array}, G[P] \rightarrow S_2, G'[P']$

$$\begin{array}{c} ; \\ \swarrow \quad \searrow \\ \text{Fail}, G'[P'] \quad S_2 \end{array} , G[P] \rightarrow \text{Fail}, G[P]$$

**A.3** *if*( $S_1 = E$ )*then*( $S_2$ )*else*( $S_3$ )

$$\begin{array}{c} \text{if} \\ \swarrow \quad \downarrow \quad \searrow \\ S_1 \quad E \quad S_2 \quad S_3 \end{array} , G[P] \rightarrow \begin{array}{c} \text{if} \\ \swarrow \quad \downarrow \quad \searrow \\ S_1, G[P] \quad E \quad S_2 \quad S_3 \end{array} , G[P]$$

$$\begin{array}{c} \text{if} \\ \swarrow \quad \downarrow \quad \searrow \\ E_1 G'[P'] \quad E_1 \quad S_2 \quad S_3 \end{array} , G[P] \rightarrow S_2, G[P]$$

$$\begin{array}{c} \text{if} \\ \swarrow \quad \downarrow \quad \searrow \\ E_1, G'[P'] \quad E_2 \quad S_2 \quad S_3 \end{array} , G[P] \rightarrow S_3, G[P]$$

**A.4 Non determinism**

**A.4.1** +

$$\begin{array}{c} + \\ \swarrow \quad \searrow \\ S_1 \quad S_2 \end{array} , G[P] \rightarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ S_1, G[P] \quad S_2, G[P] \end{array} , G[P]$$

$$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{Id}, G_1[P_1] \quad \text{Fail}, G_2[P_2] \end{array} , G[P] \rightarrow \text{Id}, G_1[P_1]$$

$$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{Fail}, G_1[P_1] \quad \text{Id}, G_2[P_2] \end{array} , G[P] \rightarrow \text{Id}, G_2[P_2]$$

$$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{Fail}, G_1[P_1] \quad \text{Fail}, G_2[P_2] \end{array} , G[P] \rightarrow \text{Fail}, G[P]$$

$$\begin{array}{c}
 + \quad ,G[P] \quad \rightarrow \quad \text{Id},G_1[P_1] \\
 \swarrow \quad \searrow \\
 \text{Id},G_1[P_1] \quad \text{Id},G_2[P_2]
 \end{array}$$

$$\begin{array}{c}
 + \quad ,G[P] \quad \rightarrow \quad \text{Id},G_2[P_2] \\
 \swarrow \quad \searrow \\
 \text{Id},G_1[P_1] \quad \text{Id},G_2[P_2]
 \end{array}$$

**A.4.2 pick**

$$\begin{array}{c}
 \text{pick } ,G[P] \quad \rightarrow \quad S_1,G[P] \\
 \swarrow \quad \searrow \\
 S_1 \quad S_2
 \end{array}$$

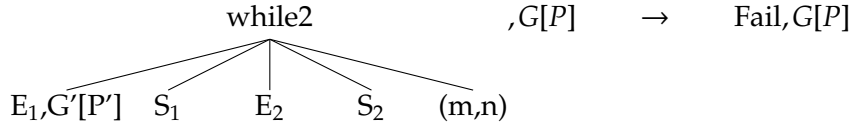
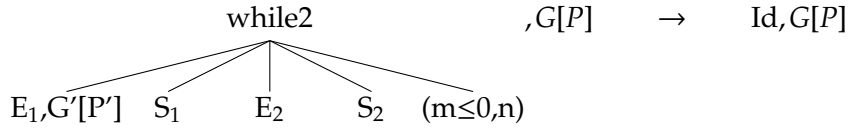
$$\begin{array}{c}
 \text{pick } ,G[P] \quad \rightarrow \quad S_2,G[P] \\
 \swarrow \quad \searrow \\
 S_1 \quad S_2
 \end{array}$$

**A.5** *while*( $S_1 = E$ )*do*( $S_2$ )*min*( $m$ )*max*( $n$ )

$$\begin{array}{c}
 \text{while} \quad ,G[P] \quad \rightarrow \quad \text{while2} \quad ,G[P] \\
 \swarrow \quad \downarrow \quad \searrow \quad \searrow \\
 S_1 \quad E \quad S_2 \quad (m,n) \\
 \swarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \searrow \\
 S_1,G[P] \quad S_1 \quad E \quad S_2 \quad (m,n)
 \end{array}$$

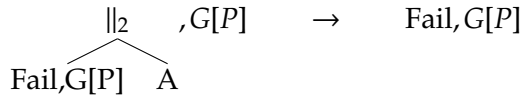
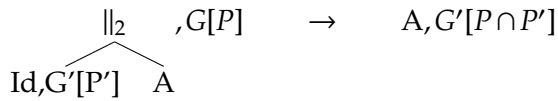
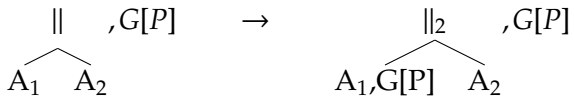
$$\begin{array}{c}
 \text{while2} \quad ,G[P] \quad \rightarrow \quad \text{Id},G[P] \\
 \swarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \searrow \\
 S_1,G[P] \quad S_1 \quad E \quad S_2 \quad (m,0)
 \end{array}$$

$$\begin{array}{c}
 \text{while2} \quad ,G[P] \quad \rightarrow \quad ; \quad ,G[P] \\
 \swarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \searrow \\
 E_1,G'[P'] \quad S_1 \quad E_1 \quad S_2 \quad (m,n) \\
 \swarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \searrow \\
 S_2,G[P] \quad ; \quad \text{while} \quad ,G[P] \\
 \swarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \searrow \\
 S_1 \quad E \quad S_2 \quad (m-1,n-1)
 \end{array}$$

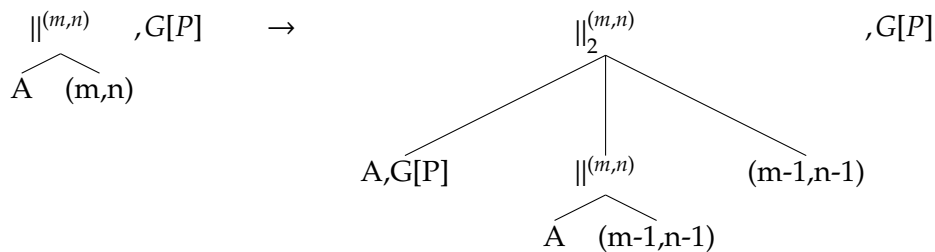
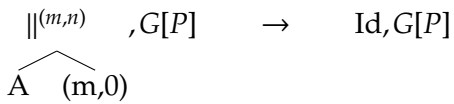


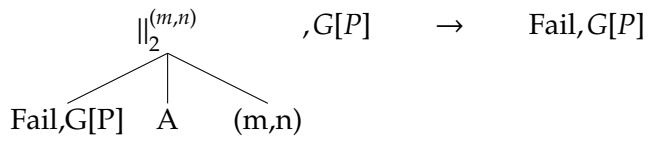
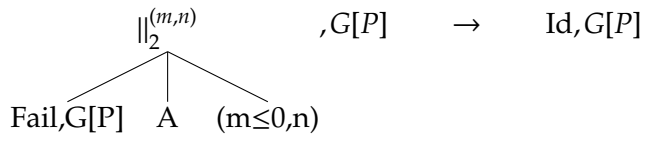
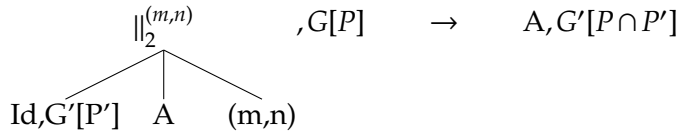
## A.6 Parallelism

### A.6.1 $\parallel$



### A.6.2 $\parallel^{(m,n)}$





**A.6.3**  $\parallel$

