Pre-proceedings

10th International Workshop on Rule-Based Programming

RULE 2009

Brasilia

28 June 2009

Federated Conference on Rewriting, Deduction and Programming

Editors:

Anamaria Martins Moreira and Ian Mackie

Preface

RULE 2009 is the tenth International Workshop on Rule-Based Programming, and takes place June 28th 2009, Brasília, Brazil, in conjunction with RDP 2009. The first Rule workshop was held in Montréal in 2000, and subsequent editions took place in Firenze, Pittsburgh, Valencia, Aachen, Nara, Seattle, Paris, and Hagenberg.

The fundamental concepts of rule-based programming are present in many areas of computer science, from theory to practical implementations. In programming languages, term rewriting is used in semantics as well as in implementations that use bottom-up rewriting for code generation. Rules are also used to perform computations in various systems; to describe logical inference in theorem provers; to specify and implement constraint-based algorithms and applications; and to describe and implement program transformations. Rule-based programming provides a common framework for viewing computation as a sequence of transformations on some shared structure such as a term, graph, proof, or constraint store. Rule selection and application is typically governed by a rich set of sophisticated mechanisms for recognizing and manipulating structures.

After the development of the principles of rewriting logic and of the rewriting calculus in the nineties, languages and systems such as ASF+SDF, BURG, CHRS, Claire, ELAN, Maude, and Stratego contributed to demonstrate the importance of rule-based programming. The area has since been experiencing a period of growth with the emergence of new concepts, systems, and applications domains, such as Domain Specific Languages, Generative and Aspect-Oriented Programming, and Software Engineering activities like maintenance, reverse engineering, and testing.

The goal of this workshop is to bring together researchers from the various communities working on rule-based programming to foster advances in the foundations and research on rule-based programming methods and systems; and to promote cross-fertilization between theory and practice, and the application of rule-based programming in various important domains.

Topics of interest include:

- Theory and Languages for rule-based programming: Advances in pattern and rewriting calculi, Advances in rewriting logic, Complexity results, Static analysis, Semantics, Type Systems, Implementation techniques, Domain-specific Languages
- Rule-based specification: Business rule systems, Policy specifications
- Applications: Software analysis and transformation, Software development and testing, Reengineering, Security
- Paradigm combinations of Rule-based programming: with Functional Programming, with Logic Programming, with Object-oriented programming, Language embedding and extensions

- Tool and System descriptions: Usability engineering for rule-based programming tools, Experience in building or using rule-based programming systems, Practical aspects of rule-based programming systems, Empirical evaluation of rule-based programming

The Programme Committee has selected ten papers for presentation at RULE 2009. In addition, the programme includes an invited talk by Hélène Kirchner. The Final Proceedings of RULE 2009 will appear as a volume of *Electronic Proceedings in Theoretical Computer Science*.

We would like to thank all those who contributed to RULE 2009. We are grateful to the programme committee members and the external referees for their careful and efficient work in the reviewing process.

Anamaria Martins Moreira and Ian Mackie

Verifying Temporal Regular properties of Abstractions of Term Rewriting Systems

Benoît Boyer
Université Rennes 1, France
Benoit.Boyer@irisa.fr

Thomas Genet
Université Rennes 1, France
Thomas.Genet@irisa.fr

The tree automaton completion is an algorithm used for proving safety properties of systems that can be modeled by a term rewriting system. This representation and verification technique works well for proving properties of infinite systems like cryptographic protocols or more recently on Java Bytecode programs. This algorithm computes a tree automaton which represents a (regular) over approximation of the set of reachable terms by rewriting initial terms. This approach is limited by the lack of information about rewriting relation between terms. Actually, terms in relation by rewriting are in the same equivalence class: there are recognized by the same state in the tree automaton.

Our objective is to produce an automaton embedding an abstraction of the rewriting relation sufficient to prove temporal properties of the term rewriting system.

We propose to extend the algorithm to produce an automaton having more equivalence classes to distinguish a term or a subterm from its successors w.r.t. rewriting. While ground transitions are used to recognize equivalence classes of terms, ε-transitions represent the rewriting relation between terms. From the completed automaton, it is possible to automatically build a Kripke structure abstracting the rewriting sequence. States of the Kripke structure are states of the tree automaton and the transition relation is given by the set of ε-transitions. States of the Kripke structure are labelled by the set of terms recognized using ground transitions. On this Kripke structure, we define the Regular Linear Temporal Logic (R-LTL) for expressing properties. Such properties can then be checked using standard model checking algorithms. The only difference between LTL and R-LTL is that predicates are replaced by regular sets of acceptable terms.

1 Introduction

Our main objective is to formally verify programs or systems modeled using Term Rewriting Systems. In a previous work [2], we have shown that it is possible to translate a Java bytecode program into a Term Rewriting System (TRS). In this case, terms model Java Virtual Machine (JVM) states and the execution of bytecode instructions is represented by rewriting, according to the small-step semantics of Java. An interesting point of this approach is the possibility to classify rewriting rules. More precisely, there is a strong relation between the position of rewriting in a term and the semantics of the executed transition on the corresponding state. For the case of Java bytecode, since a term represents a JVM state, rewriting at the top-most position corresponds to manipulations of the call stack, i.e. it simulates a method call or method return. On the other hand, since the left-most subterm represents the execution context of the current method (so called frame), rewriting at this position simulates the execution of the code of *this* method. Hence, by focusing on rewriting at a particular position, it is possible to analyse a Java program at the method call level (inter procedural control flow) or at the instruction level (local control flow).

The verification technique used in [2], called Tree Automata Completion [5], is able to finitely over-approximate the set of reachable terms, i.e. the set of all reachable states of the JVM. However, this technique lacks precision in the sense that it makes no difference between all those reachable terms. Due to the approximation algorithm, all reachable terms are considered as equivalent and the execution

Submitted to Electronic Proceedings
in Theoretical Computer Science.

© Boyer & Genet

ordering is lost. In particular, this prevents to prove temporal properties of such models. However, using approximations makes it possible to prove unreachability properties of infinite state systems.

In this preliminary work, we propose to improve the Tree Automata Completion method so as to prove temporal properties of a TRS representing a finite state system. The first step is to refine the algorithm so as to produce a tree automaton keeping an approximation of the rewriting relation between terms. Then, in a second step, we propose a way to check LTL-like formulas on this tree automaton.

2 Preliminaries

Comprehensive surveys can be found in [1] for rewriting, and in [4, 7] for tree automata and tree language theory.

Let $\mathscr{F}$ be a finite set of symbols, each associated with an arity function, and let $\mathscr{X}$ be a countable set of variables. $\mathscr{T}(\mathscr{F},\mathscr{X})$ denotes the set of terms, and $\mathscr{T}(\mathscr{F})$ denotes the set of ground terms (terms without variables). The set of variables of a term t is denoted by $\mathscr{V}ar(t)$. A substitution is a function σ from $\mathscr{X}$ into $\mathscr{T}(\mathscr{F},\mathscr{X})$, which can be uniquely extended to an endomorphism of $\mathscr{T}(\mathscr{F},\mathscr{X})$. A position p for a term t is a word over $\mathbb{N}$. The empty sequence λ denotes the top-most position. The set $\mathscr{P}os(t)$ of positions of a term t is inductively defined by:

- $\mathscr{P}os(t) = \{\lambda\}$ if $t \in \mathscr{X}$
- $\mathscr{P}os(f(t_1,\ldots,t_n)) = \{\lambda\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathscr{P}os(t_i)\}$

If $p \in \mathscr{P}os(t)$, then $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the term obtained by replacement of the subterm $t|_p$ at position p by the term s. A term rewriting system (TRS) $\mathscr{R}$ is a set of *rewrite rules* $l \to r$, where $l, r \in \mathscr{T}(\mathscr{F},\mathscr{X})$, $l \notin \mathscr{X}$, and $\mathscr{V}ar(l) \supseteq \mathscr{V}ar(r)$. The TRS $\mathscr{R}$ induces a rewriting relation $\to_{\mathscr{R}}$ on terms as follows. Let $s,t \in \mathscr{T}(\mathscr{F},\mathscr{X})$ and $l \to r \in \mathscr{R}$, $s \to^p_{\mathscr{R}} t$ denotes that there exists a position $p \in \mathscr{P}os(t)$ and a substitution σ such that $s|_p = l\sigma$ and $r = s[r\sigma]_p$. Note that the rewriting position p can generally be omitted, i.e. we generally write $s \to_{\mathscr{R}} t$. The reflexive transitive closure of $\to_{\mathscr{R}}$ is denoted by $\to^*_{\mathscr{R}}$. The set of $\mathscr{R}$-descendants of a set of ground terms E is $\mathscr{R}^*(E) = \{t \in \mathscr{T}(\mathscr{F}) \mid \exists s \in E \text{ s.t. } s \to^*_{\mathscr{R}} t\}$.

The *verification technique* defined in [6, 5] is based on the approximation of $\mathscr{R}^*(E)$. Note that $\mathscr{R}^*(E)$ is possibly infinite: $\mathscr{R}$ may not terminate and/or E may be infinite. The set $\mathscr{R}^*(E)$ is generally not computable [7]. However, it is possible to over-approximate it [6, 5, 9] using tree automata, i.e. a finite representation of infinite (regular) sets of terms. In this verification setting, the TRS $\mathscr{R}$ represents the system to verify, sets of terms E and *Bad* respectively represent the set of initial configurations and the set of "bad" configurations that should not be reached. Using tree automata completion, we construct a tree automaton B whose language $\mathscr{L}(B)$ is such that $\mathscr{L}(B) \supseteq \mathscr{R}^*(E)$. If $\mathscr{L}(B) \cap Bad = \emptyset$ then this proves that $\mathscr{R}^*(E) \cap Bad = \emptyset$, and thus that none of the "bad" configurations is reachable. We now define tree automata.

Let Q be a finite set of symbols, with arity 0, called *states* such that $Q \cap \mathscr{F} = \emptyset$. $\mathscr{T}(\mathscr{F} \cup Q)$ is called the set of *configurations*.

Definition 1 (Transition, normalized transition, ε-transition)**.** *A* transition *is a rewrite rule $c \to q$, where c is a configuration i.e. $c \in \mathscr{T}(\mathscr{F} \cup Q)$ and $q \in Q$. A* normalized transition *is a transition $c \to q$ where $c = f(q_1,\ldots,q_n)$, $f \in \mathscr{F}$ whose arity is n, and $q_1,\ldots,q_n \in Q$. An* ε-transition *is a transition of the form $q \to q'$ where q and q' are states.*

Definition 2 (Bottom-up nondeterministic finite tree automaton)**.** *A bottom-up nondeterministic finite tree automaton (tree automaton for short) is a quadruple $A = \langle \mathscr{F}, Q, Q_F, \Delta \cup \Delta_\varepsilon \rangle$, where $Q_F \subseteq Q$, Δ is a set of normalized transitions and Δ_ε is a set of ε-transitions.*

The *rewriting relation* on $\mathscr{T}(\mathscr{F} \cup Q)$ induced by the transitions of A (the set $\Delta \cup \Delta_\varepsilon$) is denoted by $\rightarrow_{\Delta \cup \Delta_\varepsilon}$. When Δ is clear from the context, $\rightarrow_{\Delta \cup \Delta_\varepsilon}$ will also be denoted by $\rightarrow_A$. We also introduce $\rightarrow_A^{\not\varepsilon}$ the *transitive relation* which is induced by the set Δ alone.

Definition 3 (Recognized language, canonical term)**.** *The tree language recognized by A in a state q is $\mathscr{L}(A,q) = \{t \in \mathscr{T}(\mathscr{F}) \mid t \rightarrow_A^* q\}$. The language recognized by A is $\mathscr{L}(A) = \bigcup_{q \in Q_F} \mathscr{L}(A,q)$. A tree language is regular if and only if it can be recognized by a tree automaton. A term t is a* canonical term *of the state q, if $t \rightarrow_A^{\not\varepsilon} q$.*

Example 1. *Let A be the tree automaton $\langle \mathscr{F}, Q, Q_F, \Delta \rangle$ such that $\mathscr{F} = \{f,g,a\}$, $Q = \{q_0,q_1,q_2\}$, $Q_F = \{q_0\}$, $\Delta = \{f(q_0) \rightarrow q_0, g(q_1) \rightarrow q_0, a \rightarrow q_1, b \rightarrow q_2\}$ and $\Delta_\varepsilon = \{q_2 \rightarrow q_1\}$. In Δ, transitions are* normalized*. A transition of the form $f(g(q_1)) \rightarrow q_0$ is not normalized. The term $g(a)$ is a term of $\mathscr{T}(\mathscr{F} \cup Q)$ (and of $\mathscr{T}(\mathscr{F})$) and can be rewritten by Δ in the following way: $g(a) \rightarrow_A^{\not\varepsilon} g(q_1) \rightarrow_A^{\not\varepsilon} q_0$. Hence $g(a)$ is a canonical term of q_1. Note also that $b \rightarrow_A q_2 \rightarrow_A q_1$. Hence, $\mathscr{L}(A,q_1) = \{a,b\}$ and $\mathscr{L}(A) = \mathscr{L}(A,q_0) = \{g(a), g(b), f(g(a)), f(f(g(b))), \ldots\} = \{f^*(g([a|b]))\}$.*

3 The Tree Automata Completion with ε-transitions

Given a tree automaton A and a TRS $\mathscr{R}$, the tree automata completion algorithm, proposed in [6, 5], computes a *tree complete automaton* $A^*_\mathscr{R}$ such that $\mathscr{L}(A^*_\mathscr{R}) = \mathscr{R}^*(\mathscr{L}(A))$ when it is possible (for some of the classes of TRSs where an exact computation is possible, see [5]), and such that $\mathscr{L}(A^*_\mathscr{R}) \supseteq \mathscr{R}^*(\mathscr{L}(A))$ otherwise. In this paper, we only consider the exact case.

The tree automata completion with ε-transtions works as follow. From $A = A^0_\mathscr{R}$ completion builds a sequence $A^0_\mathscr{R}.A^1_\mathscr{R} \ldots A^k_\mathscr{R}$ of automata such that if $s \in \mathscr{L}(A^i_\mathscr{R})$ and $s \rightarrow_\mathscr{R} t$ then $t \in \mathscr{L}(A^{i+1}_\mathscr{R})$. Transitions of $A^i_\mathscr{R}$ are denoted by the set $\Delta^i \cup \Delta^i_\varepsilon$. Since for every tree automaton, there exists a deterministic tree automaton recognizing the same language, we can assume that initially A has the following property:

Property 1. *If Δ contains two normalized transitions of the form $f(q_1,\ldots,q_n) \rightarrow q$ and $f(q_1,\ldots,q_n) \rightarrow q'$, it means $q = q'$. This ensures that the rewriting relation $\rightarrow^{\not\varepsilon}$ is deterministic.*

If we find a fixpoint automaton $A^k_\mathscr{R}$ such that $\mathscr{R}^*(\mathscr{L}(A^k_\mathscr{R})) = \mathscr{L}(A^k_\mathscr{R})$, then we note $A^*_\mathscr{R} = A^k_\mathscr{R}$ and we have $\mathscr{L}(A^*_\mathscr{R}) = \mathscr{R}^*(\mathscr{L}(A^0_\mathscr{R}))$ [5]. To build $A^{i+1}_\mathscr{R}$ from $A^i_\mathscr{R}$, we achieve a *completion step* which consists of finding *critical pairs* between $\rightarrow_\mathscr{R}$ and $\rightarrow_{A^i_\mathscr{R}}$. To define the notion of critical pair, we extend the definition of substitutions to the terms of $\mathscr{T}(\mathscr{F} \cup Q)$. For a substitution $\sigma : \mathscr{X} \mapsto Q$ and a rule $l \rightarrow r \in \mathscr{R}$, a critical pair is an instance $l\sigma$ of l such that there exists $q \in Q$ satisfying $l\sigma \rightarrow^*_{A^i_\mathscr{R}} q$ and $l\sigma \rightarrow_\mathscr{R} r\sigma$. Note that since $\mathscr{R}$, $A^i_\mathscr{R}$ and the set Q of states of $A^i_\mathscr{R}$ are finite, there is only a finite number of critical pairs. For every critical pair detected between $\mathscr{R}$ and $A^i_\mathscr{R}$ such that we do not have a state q' for which $r\sigma \rightarrow^{\not\varepsilon}_{A^i_\mathscr{R}} q'$ and $q' \rightarrow q \in \Delta^i_\varepsilon$, the tree automaton $A^{i+1}_\mathscr{R}$ is constructed by adding new transitions $r\sigma \rightarrow^{\not\varepsilon} q'$ to Δ^i and $q' \rightarrow q$ to Δ^i_ε such that $A^{i+1}_\mathscr{R}$ recognizes $r\sigma$ in q, i.e. $r\sigma \rightarrow^*_{A^{i+1}_\mathscr{R}} q$, see Figure 1. It is important to note that

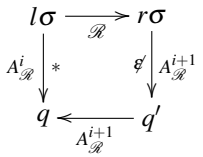


Figure 1: A critical pair solved

we consider the critical pair only if the last step of the reduction $l\sigma \rightarrow^*_{A^i_\mathscr{R}} q$, is the last step of rewriting is not a ε-transition. Without this condition, the completion computes the transitive closure of the expected

relation Δ_ε, and thus looses precision. The transition $r\sigma \to q'$ is not necessarily a normalized transition of the form $f(q_1,\ldots,q_n) \to q'$ and so it has to be normalized first. Instead of adding $r\sigma \to q'$ we add $\downarrow (r\sigma \to q')$ to transitions of Δ^i. Here is the $\downarrow$ function used to normalize transitions. Note that, in this function, transitions are normalized using new states of Q_{new}.

Definition 4 ($\downarrow$)**.** *Let $A = \langle \mathscr{F}, Q, Q_F, \Delta \cup \Delta_\varepsilon \rangle$ be a tree automaton, Q_{new} a set of* new *states such that $Q \cap Q_{new} = \emptyset$, $t \in \mathscr{T}(\mathscr{F} \cup Q)$ and $q \in Q$. The normalisation of the transition $s \to q'$ is done in two steps. We rewrite s by Δ until rewriting is impossible: we obtain a unique configuration t if Δ respects the property 1. The second step $\downarrow'$ is inductively defined by:*

- $\downarrow' (t \to q') = \emptyset$ *if* $t \in Q$,
- $\downarrow' (f(t_1,\ldots,t_n) \to q) = \bigcup_{i=1\ldots n} \downarrow' (t_i \to q_i) \cup \{f(q_1,\ldots,q_n) \to q\}$ *where* $\forall i = 1\ldots n : (t_i \in Q \Rightarrow q_i = t_i) \wedge (t_i \in \mathscr{T}(\mathscr{F} \cup Q) \setminus Q \Rightarrow q_i \in Q_{new})$.

It is very important to remark that the transition $q' \to q$ in Figure 1 creates an order between the language recognized by q and the one recognized by q'. Intuitively, we know that for all substitution $\sigma' : \mathscr{X} \to \mathscr{T}(\mathscr{F})$ such that $l\sigma'$ is a term recognized by q, it is rewritten by $\mathscr{R}$ into a canonical term $(r\sigma')$ of q'. By duality, the term $r\sigma'$ has a parent $(l\sigma')$ in the state q. Extending this reasoning we can define a relation between canonical terms of states related by Δ_ε : we consider only the rewriting at the top position as relevant and we forget rewriting at subterm.

Definition 5 ($\dashrightarrow$)**.** *Let $\mathscr{R}$ be a TRS. For all terms u v, we have $u \dashrightarrow_{\mathscr{R}} v$ iff there exists w such that $u \to^*_{\mathscr{R}} w$, $w \to^{\lambda}_{\mathscr{R}} v$ and there is not rewriting on top position λ on the sequence denoted by $u \to^*_{\mathscr{R}} w$.*

In the following, we show that the completion builds a tree automaton where the set Δ_ε is an *abstraction* $\dashrightarrow_{\mathscr{R}_i}$ of the rewriting relation $\to_{\mathscr{R}}$, for any relevant set $\mathscr{R}_i$.

Theorem 1 (Correctness)**.** *Let be $A^*_{\mathscr{R}}$ a complete tree automaton such that $q' \to q$ is a ε-transition of $A^*_{\mathscr{R}}$. Then, there exists two canonical terms u v such we have the following commutative diagram :*

$$\begin{array}{ccc} u & \overset{*}{\underset{\mathscr{R}}{\dashrightarrow}} & v \\ {\scriptstyle A^*_{\mathscr{R}}}\downarrow{\scriptstyle \varepsilon'} & & {\scriptstyle A^*_{\mathscr{R}}}\downarrow{\scriptstyle \varepsilon'} \\ q & \longleftarrow & q' \end{array}$$

To prove theorem 1, we need a stronger property:

Lemma 1. *Let be $A^*_{\mathscr{R}}$ a complete tree automaton, q a state of $A^*_{\mathscr{R}}$ and $v \in \mathscr{L}(A^*_{\mathscr{R}}, q)$. Then, there exists a canonical term u of q such we have $u \to^*_{\mathscr{R}} v$.*

Proof sketch. The proof is done by induction on the number of steps of completion to reach the post-fixpoint $A^*_{\mathscr{R}}$: we are going to show that if $A^i_{\mathscr{R}}$ respects the property of theorem 1, then $A^{i+1}_{\mathscr{R}}$ does also it.

First, we consider the normalization of a transition of the form $r\sigma \to^{\varepsilon'} q'$ and we show that the property is true for q'. For all substitution $\sigma' : \mathscr{X} \mapsto \mathscr{T}(\mathscr{F})$ such that $r\sigma' \in \mathscr{L}(A^i_{\mathscr{R}}, q')$, there exists a substitution $\sigma'' : \mathscr{X} \mapsto \mathscr{T}(\mathscr{F})$ such that $r\sigma'' \to^*_{\mathscr{R}} r\sigma'$ and $r\sigma''$ is a canonical term of q'. We consider the added transition $q' \to q$. For all canonical term $r\sigma''$ of q', there exists a term $l\sigma''' \in \mathscr{L}(A^i_{\mathscr{R}}, q)$ such that $l\sigma''' \to_{\mathscr{R}} r\sigma'''$ and $r\sigma''' = r\sigma''$. By hypothesis on $A^i_{\mathscr{R}}$, we know there exists a canonical term u of q such that $u \to^*_{\mathscr{R}} l\sigma'''$. By transitivity,we have $u \to^*_{\mathscr{R}} r\sigma'$. The last step consists in proving that for all terms of all states of $A^{i+1}_{\mathscr{R}}$, the property holds: this can be done by induction on the deepning of the recognized terms. □

The theorem 1 is proved by considering the introduction of the transition $q' \to q$: by construction, there exists a substitution $\sigma : \mathscr{X} \mapsto Q$ and a rule $l \to r \in \mathscr{R}$ such that we have $l\sigma \to^*_{A^*_\mathscr{R}} q$ and $r\sigma \to^{\varepsilon'}_{A^*_\mathscr{R}} q'$. We define a new substitution $\sigma' : \mathscr{X} \mapsto \mathscr{T}(\mathscr{F})$ such that for each variable $x \in \mathscr{V}(l)$, $\sigma'(x)$ is a canonical term of the state $\sigma(x)$. Obviously, using the result of the lemma 1, there exists a canonical term u of q such that $u \to^*_\mathscr{R} l\sigma'$. Since the last step of rewriting in the reduction $l\sigma \to^*_{A^*_\mathscr{R}} q$ is not a ε-transition, we also deduce that $l\sigma'$ is not produced by a rewriting at the top position of u whereas it is the case for $r\sigma'$ and we have $u \dashrightarrow_\mathscr{R} r\sigma'$.

Theorem 2 (Completeness)**.** *Let $A^*_\mathscr{R}$ be a complete tree automaton, q, q' states of $A^*_\mathscr{R}$ and $u, v \in \mathscr{T}(\mathscr{F})$ such that u is a canonical term of q and v is a canonical term of q'. If $u \dashrightarrow_\mathscr{R} v$ then there exists a ε-transition $q' \to q$ in $A^*_\mathscr{R}$.*

Proof sketch. By definition of $u \dashrightarrow_\mathscr{R} v$ there exists a term w such that $u \to^*_\mathscr{R} w$ and and there exists a rule $l \to r \in \mathscr{R}$ and a substitution $\sigma : \mathscr{X} \mapsto \mathscr{T}(\mathscr{F})$ such that $w = l\sigma$ and $v = r\sigma$. Since $A^*_\mathscr{R}$ is a complete tree automaton, it is closed by rewriting. It means that any term obtained by rewriting any term of $\mathscr{L}(A^*_\mathscr{R}, q)$ is also in $\mathscr{L}(A^*_\mathscr{R}, q)$. This property is true in particular for the terms u and w. Since w is rewritten in q by transitions of $A^*_\mathscr{R}$, we can define a second substitution $\sigma' : \mathscr{X} \mapsto Q$ such that $l\sigma \to^*_{A^*_\mathscr{R}} l\sigma' \to^*_{A^*_\mathscr{R}} q$. Using again the closure property of $A^*_\mathscr{R}$, we know that the critical pair $l\sigma' \to_\mathscr{R} r\sigma'$ and $l\sigma' \to^*_{A^*_\mathscr{R}} q$ is solved by adding the transitions $r\sigma' \to^{\varepsilon'}_{A^*_\mathscr{R}} q''$ and $q'' \to q$. Since the property 1 is preserved by completion steps, we can deduce that $q'' = q'$ which means $q' \to q$. □

Example 2. *To illustrate this result, we give a completed tree automaton for a small TRS. We define $\mathscr{R}$ as the union of the two sets of rules $\mathscr{R}_1 = \{a \to b,\ b \to c\}$ and $\mathscr{R}_2 = \{f(c) \to g(a),\ g(c) \to h(a),\ h(c) \to f(a)\}$. We define initial set $E = \{f(a)\}$. We obtain the following tree automaton fixpoint :*

$$A^*_\mathscr{R} = \left\langle Q_F = \{q_f\},\quad \Delta = \left\{ \begin{array}{rcl} a & \to & q_a \\ b & \to & q_b \\ c & \to & q_c \\ f(q_a) & \to & q_f \\ g(q_a) & \to & q_g \\ h(q_a) & \to & q_h \end{array} \right\} \Delta_\varepsilon = \left\{ \begin{array}{rcl} q_b & \to & q_a \\ q_c & \to & q_b \\ q_g & \to & q_f \\ q_h & \to & q_g \\ q_f & \to & q_h \end{array} \right\} \right\rangle$$

If we consider the transition $q_h \to q_g$, and its canonical terms $h(a)$ and $g(a)$ respectively, we can deduce $g(a) \dashrightarrow_\mathscr{R} h(a)$. This is obviously an abstraction since we have $g(a) \to^1_\mathscr{R} g(b) \to^1_\mathscr{R} g(c) \to^\lambda_\mathscr{R} h(a)$.

In the following, we use the notation $\dashrightarrow_{\mathscr{R}_i}$ to specify the relation for a relevant subset $\mathscr{R}_i$ of $\mathscr{R}$. For instance, $u \dashrightarrow_{\mathscr{R}_i} v$ denotes that there exists w such that $u \to^*_\mathscr{R} w$ with no rewriting at the λ position of u and $w \to^\lambda_{\mathscr{R}_i} v$. In example 2, we can say that $g(a) \dashrightarrow_{\mathscr{R}_2} h(a)$.

4 From Tree Automaton to Kripke Structure

Let $A^*_\mathscr{R} = \langle \mathscr{T}(\mathscr{F}), Q, Q_F, \Delta \cup \Delta_\varepsilon \rangle$ be a complete tree automaton, for a given TRS $\mathscr{R}$ and an initial language recognized by A. A Kripke structure is a four tuple $K = (S, S_0, R, L)$ where S is a set of states, $S_0 \subseteq S$ initial states, $R \subseteq S \times S$ a left-total transition relation and L a function that labels each state with a set of predicates which are true in that state. In our case, the set of true predicates is a regular set of terms.

Definition 6 (Labelling Function)**.** *Let $A_P = \langle \mathscr{T}(\mathscr{F}), Q, \Delta \rangle$ be the structure defined from $A^*_{\mathscr{R}}$ by removing ε-transitions and final states. We define the labelling function $L : q \mapsto \langle \mathscr{T}(\mathscr{F}), Q, \{q\}, \Delta \rangle$ as the function which associates to a state q the automaton A_P where q is the unique final state. We obviously have the property for all state state q :*

$$\forall t \in \mathscr{L}(L(q)), \quad t \rightarrow^{\varepsilon'}_{A^*_{\mathscr{R}}} q$$

Now, we can build the Kripke structure for the subset $\mathscr{R}_i$ of $\mathscr{R}$ on which we want to prove some temporal properties.

Definition 7 (Construction of a Kripke Structure)**.** *We build the 4-tuple (S, S_0, R, L) from a tree automaton such that we have $S = Q$, $S_0 \subseteq S$ is a set of initial states, $R(q, q')$ if $q' \rightarrow q \in \Delta_\varepsilon$ and the labelling function L as just defined previously.*

Kripke structures must have a complete relation R. For any state q whose have no successor by R, we had a loop such that $R(q,q)$ holds. Note that this is a classical transformation of Kripke structures [3]. A Kripke structure is parametrized by the set S_0. It defines which connected component of R we are interested to analyze. For instance, to analyze the abstract rewriting at the top position of terms in $\mathscr{L}(A^*_{\mathscr{R}})$, we define set $S_0 = Q_F$ (the set of final states of $A^*_{\mathscr{R}}$), since all canonical terms of final states are initial terms. For all abstract rewriting at a deeper position p, we need to define a set *Sub* of initial subterms considered as the beginning of the rewriting at the position p. Then the set S_0 will be defined as $S_0 = \{q \mid \exists t \in Sub,\ t \rightarrow^{\varepsilon'}_{A^*_{\mathscr{R}}} q\}$.

Kripke structure models exactly the abstract rewriting relation $\dashrightarrow^*_{\mathscr{R}_i}$ for the corresponding subset $\mathscr{R}_i \subseteq \mathscr{R}$.

Theorem 3. *Le be $K = (S, S_0, R, L)$ a Kripke structure built from $A^*_{\mathscr{R}}$. For any states s, s' such that $R(s,s')$ holds, there exists two terms $u \in L(s)$ and $v \in L(s')$ such that $u \dashrightarrow_{\mathscr{R}_i} v$.*

Proof. Here, the proof is quite trivial. It is a consequence of the theorem 1 which can be applied on the relation R of the Kripke structure. □

In Example 2, if we want to verify properties of $\mathscr{R}_1$ or $\mathscr{R}_2$, we need to consider a different subset of Δ_ε corresponding to the abstraction of the relation rewriting $\dashrightarrow_{\mathscr{R}_i}$. Figures 2 and 3 show the Kripke structures corresponding to those abstractions. Note that in figure 2, a loop is needed on state c to have a total relation for K_1.

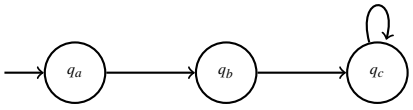


Figure 2: Kripke structure K_1 for $\dashrightarrow_{\mathscr{R}_1}$

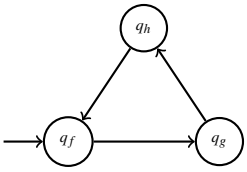


Figure 3: Kripke structure K_2 for $\dashrightarrow_{\mathscr{R}_2}$

The set S_0 of initial states depends of the abstract rewriting relation selected. For example, if we want to analyze $\dashrightarrow_{\mathscr{R}_2}$ (or $\dashrightarrow_{\mathscr{R}_1}$), we define $S_0 = \{q_f\}$ (resp. $S_0 = \{q_a\}$).

5 Verification of R-LTL properties

To express our properties, we propose to define the Regular Linear Temporal Logic (R-LTL). R-LTL is LTL where predicates are replaced by a tree automaton. The language of such a tree automaton

characterizes a set of admissible terms. A state q of a Kripke structure validates the atomic property P characterized by a tree automaton A_P if and only if one term recognized by $L(q)$ must be recognized by A_P to satisfy the property. More formally:

$$K(Q,\, Q_F,\, R,\, L),\, q \models P \quad \Longleftrightarrow \quad \mathscr{L}(L(q)) \cap \mathscr{L}(A_P) \neq \emptyset$$

We also add the operators ($\wedge$, $\vee$, $\neg$, **X**, **F**, **G**, **U**, **R**) with their standard semantics as in LTL to keep the expressiveness of the temporal logic. More information about these operators can be found in [3]. Note that temporal properties do not range over the rewriting relation $\rightarrow_{\mathscr{R}}$ but over its abstraction $\dashrightarrow_{\mathscr{R}}$. It means that the semantics of the temporal operators has to be interpreted w.r.t. this specific relation. For example, the formula $\mathbf{G}(\{f(a)\} \implies \mathbf{X}\{g(a)\})$ on K_2 (for more clarity, we note predicates as sets of terms): the formula has to be interpreted as : for all $q\ q'$, if $K_2,\ q \models \{f(a)\}$ and $R(q,q')$ then we have $K_2,\ q' \models \{g(a)\}$. In the rewriting interpretation the only term u such that $f(a) \dashrightarrow_{\mathscr{R}_2} u$ is $u = g(a)$.

We use the Büchi automata framework to perform model checking. A survey of this technique can be found in the chapter 9 of [3]. LTL (or R-LTL) formulas and Kripke structures can be translated into Büchi automata. We construct two Büchi automata : B_K obtained from the Kripke structure and B_L defined by the LTL formula. Since the set of behaviors of the Kripke structure is the language of the automaton B_K, the Kripke structure satisfies the R-LTL formula if all its behaviors are recognized by the automaton B_L. It means checking $\mathscr{L}(B_K) \subseteq \mathscr{L}(B_L)$. For this purpose, we construct the automaton $\overline{B_L}$ that recognizes the language $\overline{\mathscr{L}(B_L)}$ and we check the emptiness of the automaton $B_\cap$ that accepts the intersection of languages $\mathscr{L}(B_K)$ and $\overline{\mathscr{L}(B_L)}$. If this intersection is empty, the term rewriting system satisfies the property. This is the standard model-checking technique.

B_M and B_K are classically defined as 5-tuples: alphabet, states, initial states, final states and transition relation. Generally, the alphabet of Büchi automata is a set of predicates. Since we use here tree automata to define predicates, the alphabet of B_K and B_L is Σ the set of tree automata that can be defined over $\mathscr{T}(\mathscr{F})$. Actually, a set of behaviors is a word which describes a sequence of states: if $\pi = s_0 s_1 s_2 s_3 \ldots$ denotes a valid sequence of states in the Kripke structure, then the word $\pi' = L(s_0)L(s_1)L(s_2)\ldots$ is recognized by B_K. The algorithms used to build B_M and B_K can be found in [3].

The automaton intersection $B_\cap$ is obtained by computing the product of B_K by $\overline{B_L}$. By construction all states of B_K have to be final. Intuitively any infinite path over the Kripke structure must be recognized by B_K. This case allows to use a simpler version of the general Büchi automata product.

Definition 8 ($B_K \times \overline{B_L}$)**.** *The product of $B_K = \langle \Sigma,\, Q,\, Q_i,\, \Delta,\, Q \rangle$ by $\overline{B_L} = \langle \Sigma,\, Q',\, Q'_i,\, \Delta',\, F \rangle$ is defined as*

$$\langle \Sigma,\, Q \times Q',\, Q_i \times Q'_i,\, \Delta_\times,\, Q \times F \rangle$$

where $\Delta_\times$ is the set of transitions $(q_K, q_L) \xrightarrow{(A_K, A_L)} (q'_K, q'_L)$ such that $q_K \xrightarrow{A_K} q'_K$ is a transition of B_K and $q_L \xrightarrow{A_L} q'_L$ is a transition of $\overline{B_L}$. Moreover, the transition is only valid if the intersection between the languages of A_K and A_L is non empty as expected by the satisfiability of the R-LTL atomic formula.

Finally the emptiness of the language $\mathscr{L}(B_\cap)$ can be checked using the standard algorithm based on depth first search to check if final states are reachable.

Example 3. *To illustrate the approach, we propose to check the formula $P = \mathbf{G}(\{f(a)\} \implies \mathbf{X}\{g(a)\})$ on example 2. The automaton $\overline{B_L}$ (fig. 4) recognizes the negation of the formula P expressed as $\mathbf{F}(\{f(a)\} \wedge \mathbf{X}\neg\{g(a)\})$ and B_K (fig. 5) recognizes the all behaviors of the Kripke structure K_2 (fig. 3). The notation A_α denotes the tree automaton such that its language is described by α ($A_{\neg g(a)}$ recognizes the complement of the language $\mathscr{L}(A_{g(a)})$ and A_* recognizes all term in $\mathscr{T}(\mathscr{F})$). Figure 6 shows the result of*

intersection $B_\cap$ between B_K and $\overline{B_L}$. Only reachable states and valid transitions (labeled by non empty tree automata intersection) are showed. Since no reachable states of $B_\cap$ are final, its language is empty. It means that all behaviors of K_2 satisfy P : the only successor of $f(a)$ for the relation $\dashrightarrow_{\mathscr{R}_2}$ is $g(a)$.

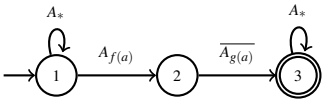


Figure 4: Automaton $\overline{B_L}$

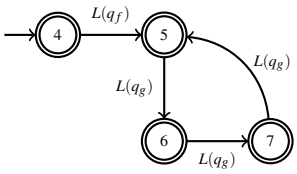


Figure 5: Automaton B_K

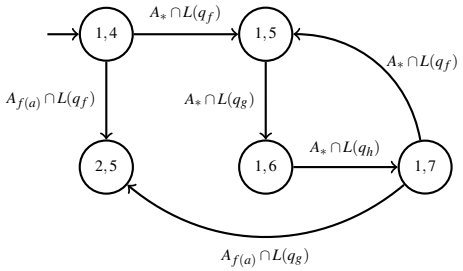


Figure 6: Automaton $B_\cap$

6 Conclusion, Discussion

In this paper, we show how to improve the tree automata completion mechanism to keep the ordering between reachable terms. This ordering was lost in the original algorithm [5]. Another contribution is the mechanism making it possible to prove LTL-like temporal properties on such abstractions of sets of reachable terms. In this paper, we only deal with finite state systems and exact tree automata completion results. Future plans are to extend this result so as to prove temporal properties on over-approximations of infinite state systems. A similar objective has already been tackled in [8]. However, this was done in a pure rewriting framework where abstractions are more heavily constrained than in tree automata completion [5]. Hence, by extending LTL formula checking on tree automata over-approximations, we hope to ease the verification of temporal formula on infinite state systems.

Acknowledgements

Many thanks to Axel Legay and Vlad Rusu for fruitful discussions on this work and to anonymous referees for their comments.

References

[1] F. Baader & T. Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press.

[2] Y. Boichut, T. Genet, T. Jensen & L. Leroux (2007): *Rewriting Approximations for Fast Prototyping of Static Analyzers*. In: *RTA*, *LNCS* 4533. Springer Verlag, pp. 48–62.

[3] Edmund M. Clarke, Orna Grumberg & Doron A. Peled (2000): *Model Checking*. MIT Press.

[4] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison & M. Tommasi (2008): *Tree Automata Techniques and Applications*. `http://tata.gforge.inria.fr`.

[5] G. Feuillade, T. Genet & V. Viet Triem Tong (2004): *Reachability Analysis over Term Rewriting Systems*. *Journal of Automated Reasonning* 33 (3-4), pp. 341–383. Available at `http://www.irisa.fr/lande/genet/publications.html`.

[6] T. Genet (1998): *Decidable Approximations of Sets of Descendants and Sets of Normal forms*. In: *Proc. 9th RTA Conf., Tsukuba (Japan)*, *LNCS* 1379. Springer-Verlag, pp. 151–165.

[7] R. Gilleron & S. Tison (1995): *Regular Tree Languages and Rewrite Systems*. *Fundamenta Informaticae* 24, pp. 157–175.

[8] J. Meseguer, M. Palomino & N. Martí-Oliet (2008): *Equational abstractions*. *TCS* 403(2-3), pp. 239–264.

[9] T. Takai (2004): *A Verification Technique Using Term Rewriting Systems and Abstract Interpretation*. In: *Proc. 15th RTA Conf., Aachen (Germany)*, *LNCS* 3091. Springer, pp. 119–133.

An Implementation of Nested Pattern Matching in Interaction Nets

Abubakar Hassan
Department of Computer Science
University of Sussex
Falmer, Brighton,
U.K.
abubakar.hassan@sussex.ac.uk

Eugen Jiresch
Theory and Logic Group
Insitute of Computer Languages
Vienna University of Technology
Vienna, Austria
jiresch@logic.at

Shinya Sato
Faculty of Econoinformatics
Himeji Dokkyo University
5-7-1 Kamiohno, Himeji-shi
Hyogo 670-8524, Japan
shinya@himeji-du.ac.jp

Reduction rules in interaction nets are constrained to pattern match exactly one argument at a time. Consequently, a programmer has to introduce auxiliary rules to perform more sophisticated matches. In this paper, we describe the design and implementation of a system for interaction nets which allows nested pattern matching on interaction rules. We achieve a system that provides convenient ways to express interaction net programs without defining auxiliary rules.

1 Introduction

Interaction nets [3] were introduced over 10 years ago as a new programming paradigm based on graph rewriting. Programs are expressed as graphs and computation is expressed as graph transformation. They enjoy nice properties such as locality of reduction, strong confluence and Turing completeness. The definition of interaction nets allows them to share computation: reducible expressions (active pairs) cannot be duplicated. For these reasons, optimal and efficient $\lambda-$calculus evaluators [1, 4, 5] based on interaction nets have evolved. Indeed, interaction nets have proved to be very useful for studying the dynamics of computation. However, they remain fruitful only for theoretical investigations.

Despite that we can already program in interaction nets, they still remain far from being used as a practical programming language. Drawing an analogy with functional programming, we only have the λ-calculus that is without high level constructs which provide programming comfort. Interaction nets have a very primitive notion of pattern matching since only two agents can interact at a time. Consequently, many auxiliary agents and rules are needed to implement more sophisticated matches. These auxiliaries are implementation details and should be generated automatically other than by the programmer.

In this paper we take a step towards developing a richer language for interaction nets which facilitates nested pattern matching. To illustrate what we are doing, consider the following definition of a function that computes the last element of a list:

```
lastElt (x:[]) = x
lastElt (x:xs) = lastElt xs
```

In this function `[]` is a nested pattern in `(x:[])`. We cannot represent functions with nested patterns in interaction nets. Hence, a programmer has to introduce auxiliary functions to pattern match the extra arguments.

```
lastElt (x:xs) = aux x xs
aux x [] = x
aux x (y:ys) = lastElt (y:ys)
```

In our previous work [2] we defined a conservative extension of interaction rules that allows nested pattern matching. The purpose of this paper is to bring these ideas into practise:

This is a preliminary version of a paper
that will appear in Electronic Proceedings
in Theoretical Computer Science.

- we define a programming language that captures the extended form of interaction rules;
- we describe the implementation of these extended rules.

In [6] we defined a textual language for interaction nets (PIN) and an abstract machine that executes PIN programs. We take PIN as our starting point and extend the PIN language to allow the representation of rules with nested patterns.

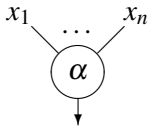
There has been several works that extend interaction nets in some way. Sinot and Mackie's Macros for interaction nets [7] are quite close to what we present in this paper. They allow pattern matching on more than one argument by relaxing the restriction of one principal port per agent. The main difference with our work is that their system does not allow nested pattern matching. Our system facilitates nested/deep pattern matching of agents.

The rest of this paper is organised as follows: In the Section 2 we give a brief introduction of interaction nets. In Section 3 we define a programming language that allows the definition of interaction rules with nested patterns. In Section 4 we give an overview of the implementation of nested pattern matching. A more detailed explanation of the algorithm is found in Section 5 (verification of well-formedness) and Section 6 (rule translation). Finally, we conclude the paper in Section 7.

2 Interaction Nets

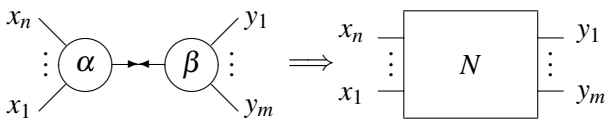
We review the basic notions of interaction nets. See [3] for a more detailed presentation. Interaction nets are specified by the following data:

- A set Σ of *symbols*. Elements of Σ serve as *agent* (node) labels. Each symbol has an associated arity *ar* that determines the number of its *auxiliary ports*. If $ar(\alpha) = n$ for $\alpha \in \Sigma$, then α has $n+1$ *ports: n* auxiliary ports and a distinguished one called the *principal port*. We represent an agent graphically as:



  and textually using the syntax: $x_0 \sim \alpha[x_1,\ldots,x_n]$ where x_0 is the principal port.

- A *net* built on Σ is an undirected graph with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge at every port. A port which is not connected is called a *free port*. A set of free ports is called an *interface*. A symbol denoting a free port is called a *free variable*.
- Two agents $(\alpha,\beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair* (analogous to a redex). An interaction rule $((\alpha,\beta) \Longrightarrow N) \in \mathscr{R}$ replaces the pair (α,β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.



  For present purposes, we represent this rule textually using $\langle \alpha[x_1,\ldots,x_n] \;><\; \beta[y_1,\ldots,y_n] \rangle \Longrightarrow N$.

We use the notation $N_1 \longrightarrow N_2$ for the one step reduction and $\longrightarrow^*$ for its transitive and reflexive closure. Interaction Nets have the following property [3]:

- **Strong Confluence**: Let N be a net. If $N \longrightarrow N_1$ and $N \longrightarrow N_2$ with $N_1 \neq N_2$, then there is a net N_3 such that $N_1 \longrightarrow N_3$ and $N_2 \longrightarrow N_3$.

In Figure 1 we give a simple example of an interaction net system that computes the last element of a list. We can represent lists using the agents Cons (:) of arity 2 and Nil of arity 0. The first port of Cons connects to an element of the list and the second port of Cons connects to the rest of the list. The agent Nil marks the end of the list. An active pair between Lst and Cons rewrites to an auxiliary agent Aux with it's principal port oriented towards the second auxiliary port of Cons. This means that during computation, Aux will interact with either a Cons agent or a Nil agent[1]. To avoid *blocking* the computation, we define rules for active pairs (Aux, Nil) and (Aux, Cons). An active pair between Aux and Nil rewrites to a single wire, which connects the agents at the auxiliary ports of Aux. When paired with Cons, Aux is replaced by Lst, analogous to the recursive call of the `lastElt` function. The list element which is connected to the first port of Aux is deleted, as it is not the last element of the list. This is modeled by the agent ε, which erases all other agents.

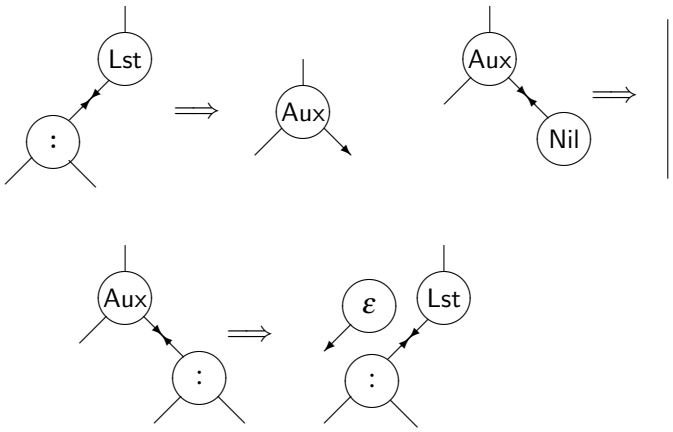


Figure 1: Rules to compute last element of a list

Figure 2 gives an example reduction sequence that computes the last element of a list that contains just one element: [1]. The second port of Lst is free and thus acts as the interface of the net. First, the

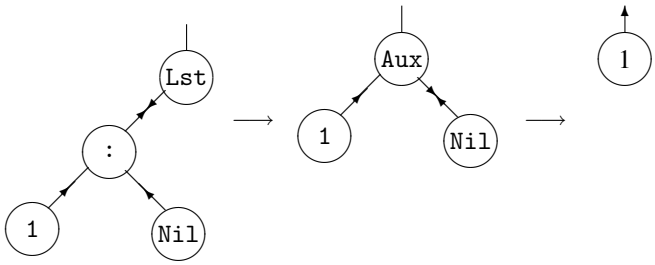


Figure 2: Example reduction sequence

active pair of Lst and Cons is rewritten, introducing an Aux agent. Now the second rule is applied to the

[1] The second auxiliary port of a Cons agent will be connected to either a Cons agent or a Nil agent.

pair $(\mathsf{Aux}, \mathsf{Nil})$, removing both agents and connecting 1 to the interface of the net. As expected, this final net contains only the agent 1, which is equivalent to the result of `LastElt (1:[])`.

2.1 Interaction rules with nested patterns - INP

The above definition of interaction nets constraints pattern matching to exactly one argument at a time. Consequently, we have to introduce auxiliary agents and rules to perform deep pattern matching (as exemplified in Figure 1). Following [2], an interaction rule may contain a *nested active pair* with more than two agents on it's left-hand side (lhs). A nested active pair is defined inductively as follows:

- Every active pair in ordinary interaction rules (ORN) is a nested active pair e.g. $P = \langle \alpha[x_1,\ldots,x_n] \ >< \ \beta[y_1,\ldots,y_m] \rangle$
- A net obtained as a result of connecting the principal port of some agent γ to a free port y_j in a nested active pair P is also a nested active pair e.g. $\langle P, y_j \sim \gamma[z_1,\ldots,z_l] \rangle$

As an example, Figure 3 gives a set of INP rules that will compute the last element of a list. In this Figure both rules contain a nested active pair on the lhs. The (non interacting) agents `Nil` and Cons on the lhs of the rules are nested agents. These rules are compiled into the set of ORN rules given in Figure 1 (See [2] for details of the compilation).

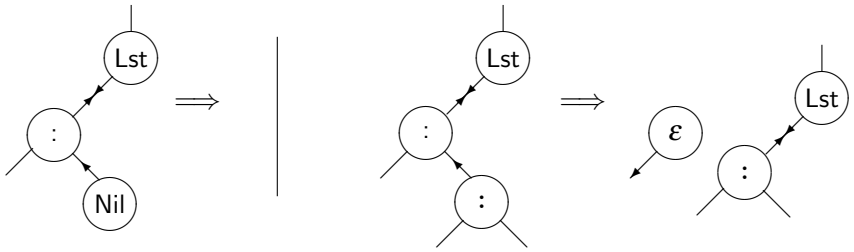


Figure 3: example INP rules to compute the last element of a list

To ensure that INP preserves the Strong Confluence property of interaction nets, rules in INP must satisfy the following constraints [2]:

Definition 2.1 (Sequentiality)**.** *Let P be a nested active pair. The set of nested active pairs $\mathscr{P}$ is sequential iff when $\langle P, y_j \sim \beta[x_1,\ldots,x_n] \rangle \in \mathscr{P}$ then*

1. *for the nested pair P, $P \in \mathscr{P}$ and,*
2. *for all the free ports y in P except the y_j and for all agents α, $\langle P, y \sim \alpha[w_1,\ldots,w_n] \rangle \notin \mathscr{P}$*

Definition 2.2 (Well-formedness)**.** *A set of INP rules $\mathscr{R}$ is well-formed iff*

1. *there is a sequential set of nested active pairs which contains every lhs of rules in $\mathscr{R}$,*
2. *for every rule $P \Longrightarrow N$ in $\mathscr{R}$, there is no interaction rule $P' \Longrightarrow N'$ in $\mathscr{R}$ such that P' is a subnet of P.*

Intuitively, the Sequentiality property avoids overlaps between rules: a set of INP rules containing nested patterns that violate condition 2 of definition 2.1 can give rise to critical pairs, which potentially destroys the Strong Confluence property of interaction nets. Note that the definition of Sequentiality allows a nested active pair to be a subnet of another nested active pair (in the same sequential set) which may also give rise to critical pairs. Definition 2.2 ensures that there is at most one nested active pair in any given set of rules.

3 The Language

We represent nets in the usual way as a comma separated list of agents. This just corresponds to a flattening of the net, and there are many different (equivalent) ways to do this depending on the order in which the agents are enumerated. Using the net in Figure 2 as an example we write:

```
p∼Lst[r],p∼Cons[x,xs],x∼1,xs∼Nil
```

The symbol '∼' denotes the principal port of the agent. The variables `p`, `x` and `xs` are used to model the connection between two ports. All variable names occur at most twice: this limitation corresponds to the requirement that it is not possible to have two edges connected to the same port in the graphical setting. If a name occurs once, then it corresponds to the free ports of the net (r is free in the above). If a name occurs twice, then it represents an edge between two ports. In this latter case, we say that a variable is *bound*.

The syntax above can be simplified by replacing equals for equals:

```
Lst[r]∼Cons[1,Nil]
```

In this notation the general form of an active pair is $\alpha[\ldots] \sim \beta[\ldots]$.

We represent rules by writing $l \Longrightarrow r$, where l is the net on the left of the rule, and r is the resulting net. In particular, we note that l will always consist of two agents connected at their principal ports. As an example, the rules in Figure 1 are written as:

```
Lst[r] >< Cons[x,xs] ⟹ xs∼ Aux[x,r]
Aux[x,r] >< Nil ⟹ x∼r
Aux[x,r] >< Cons[y,ys] ⟹ Lst[r]∼Cons[y,ys], ε ∼ x
```

The names of the bound variables in the two nets must be disjoint, and the free variables must coincide, which corresponds to the condition that the free variables must be preserved under reduction. Note that we use the symbol '><' for the active pair of the rule so that we can distinguish between an active pair and a rule.

We can simplify only *optimised* ORN rules i.e. rules that do not contain active pairs on their right-hand side (rhs). For example we can write the set of rules above in a simplified form:

```
Lst[r] >< Cons[x,Aux[x,r]]
Aux[x,x] >< Nil
Aux[x,r] >< Cons[y,ys] ⟹ Lst[r]∼Cons[y,ys], ε ∼ x
```

Note that once an optimised ORN rule is simplified, it's rhs becomes empty and therefore we omit the '⟹'.

We represent INP rules using a similar mechanism and allow the lhs of a rule to contain more than two agents. We restrict the simplification of INP rules such that the lhs and rhs are simplified independently of one another. As an example, the set of INP rules in Figure 3 can be written as:

```
Lst[r] >< Cons[x,xs], xs∼Nil ⟹ r∼x
Lst[r] >< Cons[x,xs],xs∼Cons[y,ys] ⟹ x∼ε,p∼Lst[r],p∼Cons[y,ys]
```

or in a simplified form:

```
Lst[r] >< Cons[x,Nil]⟹ r∼x
Lst[r] >< Cons[x,Cons[y,ys]]⟹ x∼ε,Lst[r]∼Cons[y,ys]
```

Other language constructs The PIN system provides other language constructs – modules, built-in operations on agent values, input/output e.t.c. These constructs remain unaffected by our extension and are out of scope in this paper. See [6] for a detailed description of the additional language features.

4 Translation

In this Section we give an overview of our translation. In general, the PIN compiler reads programs in our source language and builds the corresponding abstract syntax tree (AST). On the basis of the AST, the PIN compiler generates some byte codes which can be executed by an abstract machine or be further compiled into C source code. See [6] for a more detailed presentation of the PIN system.

Our translation function rewrites ASTs that represent INP rules into ASTs that represent ordinary interaction rules. Therefore, the back end of the PIN system remains unaffected by the translation. Overall, our translation function is similar to the compilation schemes defined in the original paper [2]. We summarise the translation algorithm in the following steps:

1. A rule is found in the AST. This rule can be either ORN or INP. All other nodes of the AST that are not rules (imports, variable declarations,...) are ignored.
2. Check if the lhs is not a subnet of a previously translated lhs (and vice versa if the rule is INP). This is the first part of verifying the well-formedness property (Definition 2.2). We discuss this verification in Section 5.1.
3. If the rule is not INP, return.
4. If the rule is INP: check if the current and all previous nested active pairs can be added to a sequential set. This is the second part of verifying the well-formedness property (Section 5.2).
5. If both checks are passed, translate the rule (else, exit with an error message):
   (a) Resolve the first nested agent of the rule's active pair.
   (b) Add an auxiliary rule to the AST.
   (c) The remaining nested agents are not (yet) translated. They are resolved by translating the auxiliary rule.

   We describe the translation algorithm in Section 6.
6. traverse the AST until the next (unprocessed) rule is found.

This algorithm allows for an arbitrary number of nested patterns (i.e., the number of nested agents in the lhs of an INP rule) and an arbitrary pattern depth.

5 Verifying the Well-Formedness Property

Our verification algorithm (see below) consists of two parts which correspond to the two constraints of the well-formedness property. We verify that the set of nested active pairs in a given PIN program are both disjoint and sequential.

We use the notation `[]` for the empty list, $[1,\ldots,n]$ for a list of n elements and $ps_1 @ ps_2$ to append two lists.

Definition 5.1 (Position Set). *Let $l \Longrightarrow r$ be a rule in* PIN. *We define the function* $\mathrm{PosSym}(l)$ *that given a nested active pair will return a set of pairs (ps,u) where ps is a list that represents the position of a symbol u in l.*

$$\begin{array}{lcl}
\mathrm{PosSym}(\alpha\,\texttt{[}t_1\,\texttt{,}\ldots\texttt{,}t_n\texttt{]}\ \texttt{><}\ \beta\,\texttt{[}s_1\,\texttt{,}\ldots\texttt{,}s_m\texttt{]}) & = & \mathrm{PosSym}_t(\texttt{[}1,1\texttt{]},\ t_1) \cup \ldots \cup \mathrm{PosSym}_t(\texttt{[}1,n\texttt{]},\ t_n) \cup \\
 & & \mathrm{PosSym}_t(\texttt{[}2,1\texttt{]},\ s_1) \cup \ldots \cup \mathrm{PosSym}_t(\texttt{[}2,m\texttt{]},\ s_m) \\
\mathrm{PosSym}_t(ps,\ x) & = & \emptyset \\
\mathrm{PosSym}_t(ps,\ \alpha\,\texttt{[}\bar{x}\texttt{]}) & = & \{(ps,\alpha)\} \\
\mathrm{PosSym}_t(ps,\ \alpha\,\texttt{[}t_1\,\texttt{,}\ldots\texttt{,}t_n\texttt{]}) & = & \mathrm{PosSym}_t(ps\,\texttt{@}\,\texttt{[}1\texttt{]},\ t_1) \cup \ldots \cup \mathrm{PosSym}_t(ps\,\texttt{@}\,\texttt{[}n\texttt{]},\ t_n)
\end{array}$$

where the sequence of terms $t_1,\ldots,t_n$ in $\mathrm{PosSym}_t(ps,\ \alpha\,\texttt{[}t_1\,\texttt{,}\ldots\texttt{,}t_n\texttt{]})$ *contain at least one term which is not a variable; and $\bar{x}$ is a sequence of zero or more variables.*
The function $\mathrm{Pos}(l)$ *returns a set of lists that represent the position of each nested agent in a nested active pair:*

$$\begin{array}{lcl}
\mathrm{Pos}(l) & = & \pi_1(\mathrm{PosSym}(l)) \\
\pi_1(\emptyset) & = & \emptyset, \\
\pi_1(\{(ps,s)\} \cup A) & = & \{ps\} \cup \pi_1(A - \{(ps,s)\}).
\end{array}$$

We extend these operations into the sequence $l_1,\ldots,l_k$ of lhs of rules as follows:

$$\begin{array}{lcl}
\mathrm{PosSym}(l_1,\ldots,l_k) & = & \mathrm{PosSym}(l_1) \cup \ldots \cup \mathrm{PosSym}(l_k), \\
\mathrm{Pos}(l_1,\ldots,l_k) & = & \mathrm{Pos}(l_1) \cup \ldots \cup \mathrm{Pos}(l_k)
\end{array}$$

Example 5.2. *For each rule in Figure 3, we can get sets of positions of nested agent pairs as follows:*

- $\mathrm{PosSym}(\texttt{Lst[r] >< Cons[x,Nil]})$
  $= \mathrm{PosSym}_t(\texttt{[1,1],r}) \cup \mathrm{PosSym}_t(\texttt{[2,1],x}) \cup \mathrm{PosSym}_t(\texttt{[2,2],Nil}) = \{(\texttt{[2,2],Nil})\}$
- $\mathrm{Pos}(\texttt{Lst[r] >< Cons[x,Nil]}) = \{\texttt{[2,2]}\}$
- $\mathrm{PosSym}(\texttt{Lst[r] >< Cons[x,Cons[y,ys]]})$
  $= \mathrm{PosSym}_t(\texttt{[1,1],r}) \cup \mathrm{PosSym}_t(\texttt{[2,1],y}) \cup \mathrm{PosSym}_t(\texttt{[2,2],Cons[y,ys]}) = \{(\texttt{[2,2],Cons})\}$
- $\mathrm{Pos}(\texttt{Lst[r] >< Cons[x,Cons[y,ys]]}) = \{\texttt{[2,2]}\}$

5.1 Subnet property

Verifying the subnet property is straightforward. Since rules are represented as trees (subtrees of the AST), it is easy to verify if one rule's lhs is a subtree of another. We compute the lhs subtree relation of the current rule P (to be translated) against all the rules Q which have already been translated. If P is in ORN, we verify the subtree relation in only one direction: the lhs of an INP rule cannot be a subnet of the lhs of an ORN rule. Otherwise we verify the subtree relation in both directions: P against Q and Q against P. The case of two ORN rules with the same active agents is handled by the compiler at an earlier stage. If the current rule's lhs is not a subnet of any previous rules' lhs's, we add it to the set of previous rules.

Note that we consider a tree to be a subtree of another tree up to alpha conversion, i.e., variable names are not considered.

5.2 Sequential set property

The check for the sequential set property is a bit more complicated than the subnet one. According to the definition of the well-formedness property, there must exist a sequential set that contains all nested active pairs in a given set of INP rules. Rather than attempting to construct such a sequential set, the algorithm tries to falsify this condition: it searches (exhaustively) for two nested patterns that cannot be in the same sequential set. This is done as follows.

For the current nested pattern P and all previously verified patterns Q with the same active agents:

1. Compare the sets $\mathrm{Pos}(P)$ and $\mathrm{Pos}(Q)$. We only consider the positions of agents at this point, not the agents themselves.
2. If one set is a subset of another, P and Q can be added to a sequential set [2]. P is added to the set of previous nested patterns.
3. Else, we compare the actual nested agents at the common positions $CP = \mathrm{Pos}(P) \cap \mathrm{Pos}(Q)$.
4. If for each element $p \in CP$, α and β are the same where $(p,\alpha) \in \mathrm{PosSym}(P)$ and $(p,\beta) \in \mathrm{PosSym}(Q)$, no sequential set can contain P and Q, as $P \equiv \langle M, x \sim \alpha[\ldots], \mathbf{a}\rangle, Q \equiv \langle M, y \sim \beta[\ldots], \mathbf{b}\rangle$ with $x \neq y$
5. Else, P and Q can be added to a sequential set. P is added to the set of previously verified nested patterns.

It is straightforward to see that after the full traversal of the AST, all possible pairs of nested patterns are considered. Hence, the search for a pair that violates the sequential set property is exhaustive.

Proposition 5.3. *Let R be a set of INP rules. R is well-formed $\Leftrightarrow$ R is correctly verified to be well-formed using our verification algorithm.*

Proof. $\Leftarrow$:

Assume R is not well-formed and passes the well-formedness checks. We proceed by a complete case distinction (according to the definition of well-formedness):

Case 1. There exist two rules $P \Longrightarrow N, Q \Longrightarrow M \in R$ where P is a subnet of Q. But then, the pair (P,Q) is tested for the subnet relation (Section 5.1). Hence, R does not pass the well-formedness check.

Case 2. There exist two rules $A \Longrightarrow N, B \Longrightarrow M \in R$ where $A \equiv \langle P, x \sim \alpha[\ldots]\rangle, B \equiv \langle P, y \sim \beta[\ldots]\rangle$ for $x \neq y$. But since all pairs of nested patterns are checked for the sequential set property (Section 5.2), (A,B) will be detected. Hence, R does not pass the check.

In both cases, we reach a contradiction to the assumption above, hence it cannot be true.

$\Rightarrow$:

Assume R does not pass the well-formedness check, but is well-formed. Again, there are only two cases:

Case 1. R does not pass the check because $\exists P \Longrightarrow N, Q \Longrightarrow N' \in R$ where P is a subnet of Q. But then, R is not well-formed (by the definition of well-formedness).

[2] Note that P and Q have already passed the subnet check at this point. This means that (some of) the nested agents at the common positions are different. Hence, P and Q cannot give rise to a critical pair.

Case 2. R does not pass because are two rules $A \Longrightarrow N, B \Longrightarrow M \in R$ where $A \equiv \langle P, x \sim \alpha[\ldots]\rangle, B \equiv \langle P, y \sim \beta[\ldots]\rangle$ for $x \neq y$. Then, there is no sequential set that contains both A and B. Hence, R is not well-formed.

Again, we reach a contradiction to the assumption above in either case.

□

6 Rule Translation

We now describe the translation algorithm in more detail. As mentioned earlier, we translate INP rules to ORN rules by rewriting the AST. We perform a pre-order traversal of the AST and identify nodes that represent INP rules. Once we find an INP rule, we replace its nested agents with a fresh (variable) node n and replace the subtree that represents the rhs of the rule with a new tree N_t. The nested agents and the rhs of the rule are stored for later processing. The tree N_t represents an active pair between n and a newly created auxiliary agent *Aux*. This auxiliary agent holds all the agents and attributes of the original active pair, with the exception of the former variable agent.

Now, we create an auxiliary rule with an active pair between *Aux* and the current nested agent (initially connected to the interacting agent). We set the rhs of the auxiliary rule to be the rhs of the original INP rule. Finally, we add this auxiliary rule to the system.

Note that the auxiliary agent in the new rule may still contain additional nested agents, i.e., the auxiliary rule may be INP. Hence, the translation algorithm recursively translates the generated rules until the lhs of each of the generated rules contains exactly two agents. The idea behind this is to resolve one nested agent per translation pass. Further nested agents are processed when the translation function reaches the respective auxiliary rule(s).

We can formalise the translation algorithm as a function $translate(\mathscr{R}, U, S)$, where $\mathscr{R}$ denotes the input set of interaction rules and U and S are *stores*. Intuitively, the components U and S are used to store previously processed rule patterns in order to verify the subnet and sequential set properties respectively. The function *translate* is defined as follows (in pseudo-code notation), where `FAIL` denotes termination of the program due to non well-formedness of $\mathscr{R}$:

```
translate([],U,S) = []
translate((P⟹N):ℛ,U,S) =
if (P is a subnet of any Q ∈ U or vice versa)
 FAIL
else if (P is ORN)
 (P⟹N):translate(ℛ,P:U,S)
else
 if (P cannot be added to a sequential set with
any Q ∈ S)
  FAIL
 else
 (P'⟹(PX∼p)):translate((PX >< A ⟹ N):ℛ,P:U,P:S)
  where
   p = position of the first nested agent of P
   A = the nested agents at position p
   P' = P with all nested agents replaced by variable
```

```
     ports
PX = auxiliary agent that contains all ports of P
     except p
```

Proposition 6.1. ***(Termination)*** *For a finite $\mathscr{R}$, translate terminates.*

Proof. Let n be the number of rules in $\mathscr{R}$ and p be the sum of all nested agents of these rules. By a complete case distinction, we show that with each recursive call, $(n+p)$ decreases:

Case 1 The current rule is ORN. At the recursive call, it is removed from $\mathscr{R}$, hence n decreases by 1.

Case 2 The current rule has i nested agents ($i > 0$). The rule is removed from $\mathscr{R}$ and an auxiliary rule with exactly $i-1$ nested agents is added to $\mathscr{R}$. Hence, with the recursive call p decreases by 1.

translate terminates if $n = 0$. n only decreases if we encounter an ORN rule. Yet, since the number of nested agents for each rule is finite, all rules in $\mathscr{R}$ will be ORN after finitely many decreases of p. □

Example 6.2. Consider the interaction rules from Figure 3. $\mathscr{R}$ consists of two rules:

```
1. Lst[r] >< Cons[x,Nil] ⟹ r∼x
2. Lst[r] >< Cons[x,Cons[y,ys]] ⟹ x∼ε,Lst[r]∼Cons[y,ys]
```

The translation works as follows:

- Rule 1 is INP (due to the `Nil` agent).
  Its lhs is checked for the subnet property. As there are no previous rules in U, the check is passed.
- As the rule is INP, it is checked for the sequential set property. Again, there are no previous rules in S, hence it passes the check.
- The rule is transformed, introducing the auxiliary agent `Lst_Cons`

  ```
  1. Lst[r] >< Cons[x,var0] ⟹ Lst_Cons[r,x]∼var0
  ```

- A new auxiliary rule is added to $\mathscr{R}$:

  ```
  3. Lst_Cons[r,x] >< Nil ⟹ r∼x
  ```

  The lhs pattern of rule 1 is added to U and S.
- Rule 2 is INP (due to the nested `Cons` agents).
  Its lhs is checked for the subnet property. The only lhs pattern in U is `Lst[r] >< Cons[x,Nil]`. Due to different agents at the second auxiliary port of `Cons`, the lhses cannot be subnets of one another. The check is passed.
- Rule 2 is checked for the sequential set property. First, the positions of nested agents of Rule 1 and 2 are compared. Since they are the same (both have their nested agent at the second auxiliary port of `Cons`), they can be added to a sequential set. There are no further rules in S, hence the check is passed.
- The rule is transformed and another auxiliary rule is added to $\mathscr{R}$:

  ```
  2. Lst[r] >< Cons[x,var1] ⟹ Lst_Cons[r,x]∼var1
  4. Lst_Cons[r,x] >< Cons[y,ys] ⟹  x∼ε,Lst[r]∼Cons[y,ys]
  ```

- Rule 3 is ORN.
  Its lhs is checked for the subnet property. As there are no with the same active agents in U, the check is passed. The lhs pattern of Rule 3 is added to U.
- Rule 4 is ORN.
  Its lhs is checked for the subnet property. Again, there are no rules with the same active agents in S. The check is passed and the lhs pattern is added to U.

This yields the translated set of rules

```
1. Lst[r] >< Cons[x,var0] ⟹ Lst_Cons[r,x]∼var0
2. Lst[r] >< Cons[x,var1] ⟹ Lst_Cons[r,x]∼var1
3. Lst_Cons[r,x] >< Nil ⟹ r∼x
4. Lst_Cons[r,x] >< Cons[y,ys] ⟹  x∼ε,Lst[r]∼Cons[y,ys]
```

Note that the rules 1 and 2 are identical (save variable names). Since only one of them is needed, rule 2 is discarded by PIN. As expected, we get the set of ORN rules given in Figure 1.

6.1 Additional language features

The PIN language offers some features that are not considered in the original definition of the nested pattern translation function. Some important examples are data values of agents (integers, floats, strings,...), side effects (declaration and manipulation of variables, I/O) and conditions. These features are not involved in the process of nested pattern matching. Therefore, they do not need to be processed or changed by the translation function:

- with regard to nested pattern matching, data values can be considered as variable ports (they do not contain nested agents). Hence, they are unaffected by the translation.
- conditionals and side effects only occur in the rhs of a rule. Since the original rhs of an INP rule is propagated to the final auxiliary rule without a change, these features are not affected either.
- all auxiliary rules but the last one are responsible for pattern matching only, they do not do the "actual work" of the original rule. All special language features are simply passed to the next auxiliary rule.

6.2 The Implementation

We have developed a prototype implementation which can be obtained from the project's web page `http://www.interaction-nets.org/`. We have thoroughly tested the prototype implementation and developed several example modules. These examples include rule systems with a large number and depth of nested patterns as well as heavy use of state, conditionals and I/O. Additionally, we have designed several non well-formed systems in order to improve error handling. Some of these examples can be found in the current PIN distribution at the project's web page.

7 Conclusion

We have presented an implementation for nested pattern matching of interaction rules. The implementation closely follows the definition of nested patterns and their translation to ordinary patterns introduced in [2]. We have shown nice properties of the algorithm such as its correctness and termination.

The resulting system allows programs to be expressed in a more convenient way rather than introducing auxiliary agents and rules to pattern match nested agents. We see this as a positive step for further extensions to interaction nets: future implementations of high-level language constructs can be built upon these more expressive rules.

References

[1] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, January 1992.

[2] Abubakar Hassan and Shinya Sato. Interaction nets with nested pattern matching. *Electronic Notes in Theoretical Computer Science*, 203, 2008.

[3] Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, January 1990.

[4] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 16–30. ACM Press, January 1990.

[5] Ian Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998.

[6] Ian Mackie, Abubakar Hassan, and Shinya Sato. Interaction nets: programming language design and implementation. *Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques*, 2008.

[7] François-Régis Sinot and Ian Mackie. Macros for interaction nets: A conservative extension of interaction nets. *Electronic Notes Theoretical Computer Science*, 127(5):153–169, 2005.

Graph creation, visualisation and transformation

Maribel Fernández and Olivier Namet
Kings College London, Department of Computer Science
Strand, London WC2R 2LS, U.K.
olivier.namet@kcl.ac.uk

We describe a tool to create, edit, visualise and compute with interaction nets, a form of graph rewriting systems. The editor, called GraphPaper, allows users to create and edit graphs and their transformation rules using an intuitive user interface. The editor uses the functionalities of the TULIP system, which gives us access to a wealth of visualisation algorithms. Interaction nets are not only a formalism for the specification of graphs, but also a rewrite-based computation model. We discuss graph rewriting strategies and a language to express them in order to perform strategic interaction net rewriting.

1 Introduction

Graph representation and graph transformations are important in Computer Science. It is well known that graphical formalisms have clear advantages as modelling tools, in particular in the earlier phases of system specification and development. Graphical formalisms are more intuitive and make it easier to visualise the system, whether in theoretical or practical domains. For example, consider the textual representation of proofs in the sequent calculus [5] versus proof nets [6], the entity-relationship diagrams [2] used to specify a relational database versus the tables, etc.

On the negative side, there are some well-known implementation problems when dealing with graphical formalisms (pattern-matching is not an easy problem, see for instance [12]), and graph rewriting can be inefficient in general. Graphical editors and graphical programming environments exist, but none of the available tools to manipulate graphs provides a unified framework to create, edit and visualise graphs, and to define dynamic transformations. Ideally, such a tool should allow users to create graphs, edit them and define different views, export to different formats (e.g., image files, encapsulated postscript, latex macros, etc.), and should also be able to model some notion of computation, allowing the user to evaluate parts of their graphs using transformation rules plugged into the tool.

In this paper, we describe the design of an editor to draw graphs and their transformation rules, and its integration into a tool that can be used to visualise graphs and their associated computations. The editor, called GraphPaper, is tailored for the design of interaction net [10] systems. These are graph rewriting systems that enjoy useful rewriting properties, such as confluence, by construction. The user interface of GraphPaper mimics the operations that users perform to draw graphs manually in paper.

Graph rewriting needs complex pattern matching, but in the case of interaction nets, the pattern-matching algorithm is simpler due to the restricted form of the left-hand sides of the rules. Still, there are several problems that have to be solved in a graphical editor, due to the changes in the layout that arise after each rewriting step. Furthermore, users of graph rewriting

Submitted to Electronic Proceedings
in Theoretical Computer Science.

© Fernández and Namet
This work is licensed under the
Creative Commons Attribution License.

systems often need to define specific strategies dictating how and when rules are applied, thus a formal language is needed to express these strategies.

Summarising: we describe the implementation of the GraphPaper editor (for graph and rule creation) and its interface with the TULIP[1] system for visualisation, strategic interaction net rewriting and tracing. TULIP is an environment for graph visuation which currently provides algorithms to display graphs in various formats, and to check static properties of graphs. In future work, we will also include a mechanism for tracing graphs during the rewriting process to allow greater control and facilitate the debugging of the rewriting system. This is work in progress within the PORGY collaboration between INRIA Bordeaux and King's College London, which focuses on graph visualisation and rewriting strategies for interaction nets and port graphs [9] in general.

Related Work There are many graphical editors available, but only a few of them allow the user to specify dynamic information in the form of graph rewriting rules. Below we discuss four systems that include this functionality and are directly related to our work.

The Interaction Net Laboratory (INL)[2] developed by De Falco is a graphical editor for interaction nets. It has a rich feature set with a Net editor and a Rule editor. Cells can be created and have properties such as "title" and "colour". Feature wise, INL is quite complete for creating and editing nets and rules but the user-interface has some limitations. Clicking on objects does not result in immediate visual feedback. To add cells to the net the user has to click on the cell in the list to the left and then click somewhere on the net. To add more than one of the same kind of cell, one has to keep on re-clicking on the cell that is in the list.

The Graphical Interpreter for Interaction Nets developed by Lippi [11] imports a text based representation of a net and a set of reduction rules and creates a graphical net. The user can then choose to reduce the net step by step or to perform all possible reductions in one go. While this program is useful for displaying a net and visualising its reduced form, the user still needs to input and edit the nets using a text-based language.

INblobs[3] developed by Vilaça et al. [1] defines itself as *an editor and interpreter for Interaction Nets*. Feature wise, INblobs is on par with INL but suffers from similar problems.

PROGRES is a programming environment based on graph grammars developed at the University of Technology Aachen[4]. It is built around an executable specification language based on a specific kind of graph rewriting rules. The environment provides a graphical editor for the specification language and a translator into C and Tcl/Tk-code. The tool seems to focus on the specification language, which is expressive enough to allow the user to model complex systems using graphs. Our goals are different: we focus on the graphical editor and in the principles behind the design of a graphical interface for the representation of graphs.

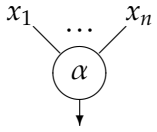
Overview of the paper Section 2 provides a concise introduction to interaction nets and rewriting strategies. In Section 3 we describe GraphPaper's novel "digital paper" user interface. Section 4 deals with visualisation and rewriting; we describe the architecture of the system and its implementation via TULIP. Section 5 contains conclusions and directions for future work.

[1] http://www.tulip-software.org
[2] http://inl.sourceforge.net/
[3] http://haskell.di.uminho.pt/jmvilaca/INblobs/
[4] See `http://www-i3.informatik.rwth-aachen.de/tikiwiki/tiki-index.php`

2 Preliminaries

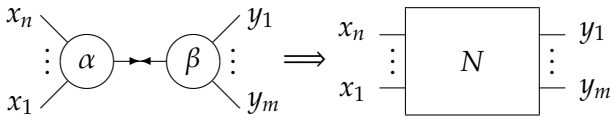
2.1 Interaction Nets

A system of interaction nets is specified by a set Σ 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 α 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 $((\alpha,\beta) \Longrightarrow N) \in \mathcal{R}$ 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 involved in the rewrite is modified), 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 Σ).



We use the notation $\Longrightarrow$ for the one-step reduction relation, or $\underset{\alpha\bowtie\beta}{\Longrightarrow}$ if we want to be explicit about the rule used, and $\Longrightarrow^*$ for its transitive and reflexive closure. If a net does not contain any active pairs then we say that it is in normal form. The key property of interaction nets, besides locality of reduction, is that reduction is strongly confluent. Indeed, all reduction sequences are permutation equivalent and standard results from rewriting theory tell us that weak and strong normalisation coincide (if one reduction sequence terminates, then all reduction sequences terminate). We refer the reader to [10] for more details and examples.

2.2 Rewriting Strategies

A graph rewriting system will 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. In the case of interaction nets, the strong confluence property ensures that all reduction sequences to full normal form are equivalent. However, 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 [4]). Also, for interaction nets, even if the end graph does not change, the size and layout of the graph during the rewriting process can differ depending on what rules and where

they are applied first. Users may therefore want to not just blindly apply rules but to create a strategy around these rules to direct the rewriting.

Strategic rewriting has been studied for term rewriting systems, and there are languages that allow the user to specify a strategy and to apply it [3, 13]. 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 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 interaction nets (which can be also applied to general graph rewriting systems).

3 GraphPaper

3.1 Digital Paper

In the past few years, new technology has shown that new Human Interface Devices (HID) can be an efficient way of dealing with digital information. There is the example of video games with Nintendo's Wii and DS consoles. One allows users to perform natural motions to convey commands (swinging your arm to swing a sword) while the other lets the user draw to the screen using a tablet. These methods of interaction have been hugely popular across all sorts of demographics. Also many new mobile devices now come with touch-screen and therefore software makers have had to rethink the way users interact with their phones. Much like the Nintendo consoles, touch screen phones (like the Apple iPhone) have also been very popular across all demographics.

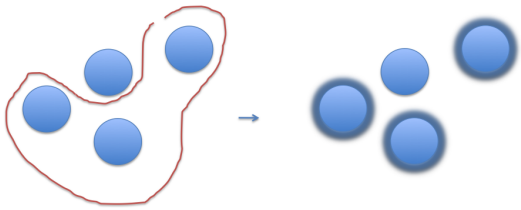
It is indeed more natural to push and pinch a map around on a touch screen to move it than it is using a mouse and icons. This is because these new HID mimic natural motions. The data that are represented in this case are graphs where the natural instinct is to use a pen and paper to draw them. We will take this intuition and try to apply it to the HCI ideology of GraphPaper. A user should feel like (s)he is using a pen and paper but with the dynamic advantages of a computer.

For graphs, agents and edges are the two types of data that the user will interact with. This allows the tool to have a simple interface. There is no need to have a toolbar to activate whether the user is trying to interact with agents or edges; it is possible for the tool to deduce what the user is interacting with based on the context of the interaction. For example, edges need to be created between ports so cannot be created independently. This means that if the user is trying to work on an empty part of the canvas, they are trying to create agents or a wiring. When the user tries to interact with ports of an agent, then we know that the goal is to create or edit edges to and from that agent. For more complex operations on graphs shape recognition is needed (see, for instance, [7]). Once a shape is recognised, the user's intention can be deduced by the context and location where the shape was drawn.

Here is an outline of shapes and some of their actions depending on context:

- Circular:

If created on an empty space then a new agent is created there.



If other agents are inside the bounds of the circle, the user's intention was to select those agents. The circle becomes a lasso selection tool.

- Line:

If drawn between two ports, an edge is created between them

If a line is drawn from inside an agent to just outside of it, then a port is created at that location on the agent.

GraphPaper has two *states* that the user can be in when it comes to the creation of graphs. A *view/edit* state and a *draw* state:

- The *view/edit* state will allow the user to modify already created agents and edges. For example, moving agents around, renaming agents and ports, etc. The user will also be able to modify the view by panning around and zooming in and out.
- The *draw* state is where the user can create and delete new objects by drawing to the screen. See the previous examples of creation.

3.2 Drawing Rules

Graph rewriting rules consist of a left hand side graph and a right hand side graph, and a mapping that defines the relation between the interfaces of the graphs. In the case of interaction nets, both sides must have the same interface, and the left hand side must be a graph consisting of two agents connected through their principal ports. To draw an interaction rule using GraphPaper, the user simply draws the two agents in the left hand side in the standard way, and then lasso selects them. In this way GraphPaper moves to the rule drawing mode, and the user can continue drawing the right hand side. The correspondence between ports of the

interface in the left hand side and in the right hand side is explicitely indicated by joining the ports (GraphPaper will show these links in a different colour). GraphPaper will then deduce this is a rule, and will isolate it to a part of the *paper* where users can still have access to it if they need to modify it.

4 Graph Rewriting

For the visualisation and transformation of graphs, GraphPaper uses the functionalities available through the TULIP[5] tool developed at INRIA Bordeaux.

- Pattern Matching: TULIP has a built in search function that can iterate through a graph of potentially very large size. Using this iteration function, we can then quite efficiently pattern match to find possible reductions in a graph. In the case of general graphs, the high complexity for pattern matching is greatly aided by the efficient iteration that TULIP provides. For Interaction Nets, it is simply a case of iterating through all the edges, finding the ones that connect two principal ports and adding them to a list. After every reduction, we then just need to iterate through that new sub-graph and its neighbours to find new active edges and add those to the list. There is no need to search the entire graph for new active edges since the reduction only affects the new sub-graph and its neighbours.
- TULIP is capable of displaying graphs with over 1,000,000 elements and can display graphs of that size in real time. GraphPaper inherits that power, allowing the user to work with graph rewriting systems where graphs grow after each rewrite step.
- Visualisation: TULIP provides dynamic visualisation of graphs, that is to say if a sub-graph is added or removed, the remaining graph dynamically changes its appearance to a chosen visual model (for example a cone tree, circular, planar). This is particularly useful in the case of rewriting graphs since once a rewrite occurs, the graph might need to be redrawn to accommodate for extra or lack of spacing.
- Strategies: As discussed in Section 2.2, strategies for graph rewriting need to specify the way rules will be applied and must also be aware of *location*. We propose to use expressions generated by the following grammar:

$$S := id \mid R \mid S,S \mid S \,||\, S \mid S^* \mid S \; or \; S$$

  where *id* is the identity (which never fails and always leaves the graph unchanged); R is an expression of the form $R_i(subgraph, depth)$ that denotes the application of rule R_i in the graph *subgraph* (which can be selected using the graphical interface) or its neighbours up to the given *depth* (as explained below); S_1, S_2 represents sequential application: apply S_1 and if successful then apply S_2 (if either of them was unsuccessful the result is $fail$); $S_1 \,||\, S_2$ represents simultaneous application (both strategies must be applied at the same time; S^* means "apply S as many times as possible in a row"; and S_1 *or* S_2 means apply S_1, if it fails then apply S_2 (not both). Note that when defining a graph rewriting strategy as a parallel composition (i.e., simultaneous application of two strategies), conflicts may arise. However, in the case of interaction nets, the constraints on interaction rules imply that

[5] http://www.tulip-software.org

each agent can only be involved in one interaction, hence all redexes can be simultaneously reduced without conflict.

The basic blocks to build a strategy are a rule and the identity. When we indicate that a rule will be applied, we must also provide the location where the rule should be applied. This is given by the arguments *subgraph* and *depth*, i.e., we specify a subgraph, and the *depth* represents how far one should look through the *subgraph*'s neighbours for a possible application of the rule. If we want it to be strict, i.e. apply the rule in the given subgraph only, we use 0. To look as far as possible starting from the given subgraph we use the value -1. The expression $R_i(subgr,1)$ indicates that we want to apply the rule R_i in the subgraph that includes *subgr* and all the neighbours at distance 1. To look n steps out of the *subgraph* then set *depth* to n. In the case of interaction nets, the *depth* search only follows principal ports of the nodes.

For convenience, when composing strategies we allow the user to factor out the common sub-expressions: If we compose strategies that have the same location (same subgraph and depth) we can write these parameters only once (e.g. $(R_1,R_2)(subgr,0)$ indicates that we want to apply R_1 and then R_2 to the subgraph *subgr*), and if we wish to apply the same strategy at several locations in the graph, we can write for example $(R_1,R_2)[(subgr_1,0),(subgr_2,0)]$, meaning that we need to apply R_1 followed by R_2 in $subgr_1$ and also in $subgr_2$.

We also define auxiliary functions $Interface(subgraph)$ which returns a graph containing the interface nodes of *subgraph*, and $Successors(subgraph)$. The first, used with $R_i()$, allows the user to easily write strategies that give priority to rewriting steps at the interface of the *subgraph*: for example, $R_1(Interface(subgraph),1)$ tries to apply R_1 on nodes of the interface and their neighbours. This is useful when computing interface normal forms of interaction nets.

- Trace: Each time a rewrite is performed, a new graph is created. To keep track of the rewriting history, we use a Trace that will store all the different graphs and if one graph is the result of a rewrite of another, an edge is created from the the latter to the former with the rule and location of the rewriting as its label. Since more than one rewrite is possible at any one time, the Trace will branch for each one, allowing the user to see all the possibilities. The Trace will therefore take on the shape of a tree. See Figure 1 on page 9 for an example and a schematic description of the architecture of the tool. Since TULIP is very efficient when it comes to storing graphs, we define and represent everything using a main graph (which we call root graph). The set of rules and strategies are stored as subgraphs of the root graph and each have a unique name. A base model M0 is also created as a subgraph of the root graph and holds the initial state of the graph the user will be rewriting on. The trace is also subgraph of the root. See Figure 2 on page 10 for an example. In this particular example, a graph $M0$ was created by the user (stored as $Me0$ in the trace) and rules $R1$ and $R2$ were applied to $M0$. The user then selected $Me1$ in the *Trace* to get a closer look at $M1$.

5 Conclusion & Future Work

GraphPaper functions as a stand-alone tool used specifically to create and edit graphs and their rules and strategies. The graph system created can then be exported into TULIP where

the rewriting will happen based on a selected strategy. The tool will generate a *Trace* of the rewriting as it happens and then allow the user to observe any point of the rewrite in more detail. The ease of use of GraphPaper combined with the power of TULIP and the detailed information provided by the Trace gives the user an environment to work on graph rewriting systems, and interaction nets in particular, efficiently and intuitively. In particular, GraphPaper can serve as an editor for visual programming languages based on interaction nets (see [8]).

In future and within the PORGY collaboration, we hope to develop the tools to support more general forms of graph rewriting by implementing a more complex and versatile pattern matching algorithm. We will also extend the interface for rule definition in GraphPaper, in order to represent more general kinds of rules, such as the ones used in PortGraph systems [9].

References

[1] José Bacelar Almeida, Jorge Sousa Pinto & Miguel Vilaça (2007): *A Tool for Programming with Interaction Nets*. In: Joost Visser & Victor Winter, editors: *Proceedings of the Eighth International Workshop on Rule-Based Programming*. Elsevier. To appear in Electronic Notes in Theoretical Computer Science.

[2] Richard Barker: *Case*Method: Entity Relationship Modelling*. Addison-Wesley.

[3] Peter Borovansky, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau & Christophe Ringeissen (1998): *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*, *Electronic Notes in Theoretical Computer Science* 15. Elsevier Science B. V., Pont-à-Mousson, France, p. 16 p. Available at `http://hal.inria.fr/inria-00098518/en/`. Colloque avec actes et comité de lecture.

[4] Maribel Fernández & Ian Mackie (1999): *A Calculus for Interaction Nets*. In: *Proceedings of PPDP'99, Paris*, number 1702 in Lecture Notes in Computer Science. Springer.

[5] G. Gentzen (1969): *Investigations into Logical Deduction*. In: M. E. Szabo, editor: *The Collected Papers of Gerhard Gentzen*. North-Holland.

[6] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50(1), pp. 1–102.

[7] T. Hammond (2007). *LADDER: A Perceptually-Based Language to Simplify Sketch Recognition User Interfaces Development*. MIT PhD Thesis.

[8] Abubakar Hassan, Ian Mackie & Jorge Sousa Pinto (2008): *Visual Programming with Interaction Nets*. In: Gem Stapleton, John Howse & John Lee, editors: *Diagrammatic Representation and Inference, 5th International Conference, Diagrams 2008, Herrsching, Germany, September 19-21, 2008. Proceedings*, *Lecture Notes in Computer Science* 5223. Springer, pp. 165–171.

[9] Hélène Kirchner & Oana Andrei (2007): *A Rewriting Calculus for Multigraphs with Ports*. *Proc.8th Int. Workshop on Rule-Based Programming (RULE07)* .

[10] Yves Lafont (1990): *Interaction Nets*. In: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*. ACM Press, pp. 95–108.

[11] Sylvain Lippi (2002): *in2: A Graphical Interpreter for Interaction Nets*. In: *RTA '02: Proceedings of the 13th International Conference on Rewriting Techniques and Applications*. Springer-Verlag, London, UK, pp. 380–386.

[12] J.R. Ullman (1976): *An Algorithm for Subgraph Isomorphism*. *Journal of the ACM* 23(1), pp. 31–42.

[13] Eelco Visser (2001): *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)*, *Lecture Notes in Computer Science* 2051. Springer-Verlag, pp. 357–361.

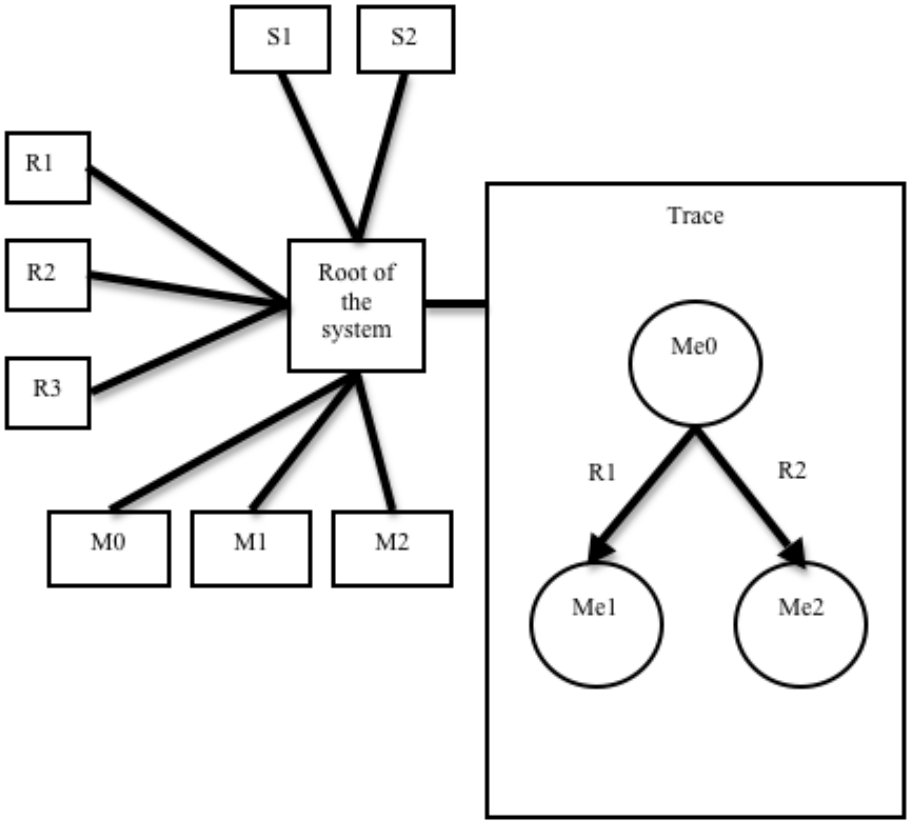


Figure 1: The architecture of the system.

Figures

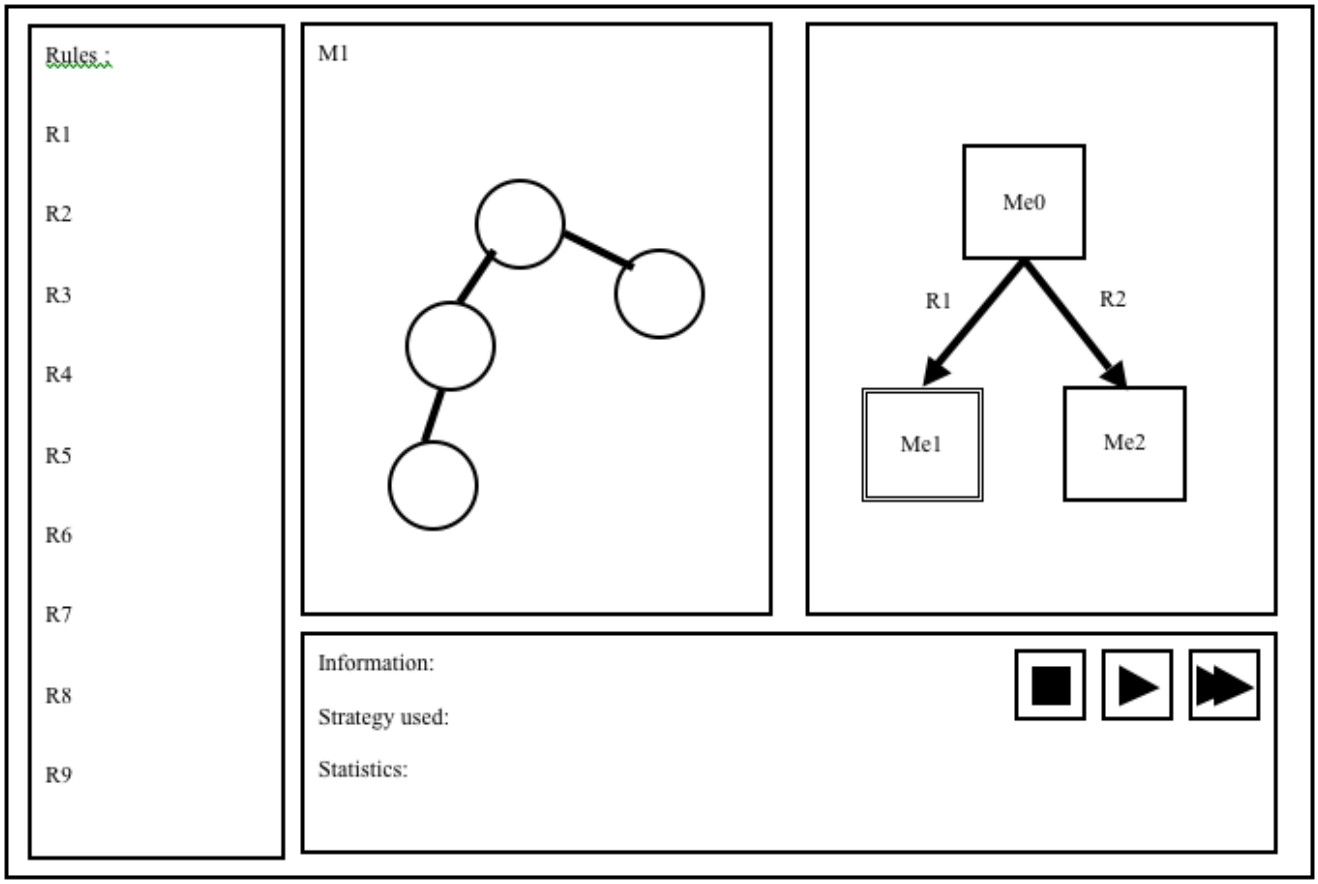


Figure 2: GUI concept for TULIP perspective.

A Type System for Tom

Claude Kirchner
INRIA Bordeaux - Sud Ouest
351, cours de la Libération, BP A29,
33405 Talence Cedex France
Claude.Kirchner@inria.fr

Pierre-Etienne Moreau
INRIA & LORIA
615 rue du Jardin Botanique, CS 20101
54603 Villers-lès-Nancy Cedex France
Pierre-Etienne.Moreau@loria.fr

Cláudia Tavares*
Claudia.Tavares@loria.fr

Extending a given language with new dedicated features is a general and quite used approach to make the programming language more adapted to problems. Being closer to the application, this leads to less programming flaws and easier maintenance. But of course one would still like to perform program analysis on these kinds of extended languages, in particular type checking and inference. But in this case one has to make the typing of the extended features compatible with the ones in the starting language.
The Tom programming language is a typical example of such a situation as it consists of an extension of Java that adds pattern matching, more particularly associative pattern matching, and reduction strategies. This paper presents a type system with subtyping for Tom, that is compatible with Java's type system, and that performs both type checking and type inference.
We propose an algorithm that checks if all patterns of a Tom program are well-typed. In addiction, we propose an algorithm based on equality and subtyping constraints that infers types of variables occurring in a pattern. Both algorithms are exemplified and the proposed type system is showed to be sound and complete.

1 Introduction of the problem: static typing in Tom

We consider here the Tom language, which is an extension of Java that provides rule based constructs. In particular, any Java program is a Tom program. We call this kind of extension *formal islands* [3, 2] where the *ocean* consists of Java code and the *island* of algebraic patterns. For simplicity, we consider only two new Tom constructs: a `%match` construct and a ‘ (backquote) construct. The semantics of `%match` is close to the *match* that exists in functional programming languages, but in an imperative context. A `%match` is parameterized by a list of subjects (*i.e.* expressions evaluated to ground terms) and contains a list of rules. The left-hand side of the rules are patterns built upon constructors and fresh variables, without any linearity restriction. The right-hand side is *not* a term, but a Java statement that is executed when the pattern matches the subject. However, thanks to the backquote construct (‘) a term can be easily built and returned. In a similar way to the standard `switch/case` construct, patterns are evaluated from top to bottom. In contrast to the functional *match*, several actions (*i.e.* right-hand sides) may be fired for a given subject as long as no `return` or `break` is executed. To implement a simple reduction step for each rule, it suffices to encode the left-hand side with a pattern and consider the Java statement that returns the right-hand side. For example, given the sort `Nat` and the function symbols `suc` and `zero`, addition and comparison of Peano integers may be encoded as follows:

*This work was partially supported by CAPES, BEX 4878-06-0, Brazil.

```
public Nat plus(Nat t1, Nat t2) {
 %match(t1,t2) {
   x,zero() -> { return `x; }
   x,suc(y) -> { return `suc(plus(x,y)); }
 }
}
```

```
public boolean greaterThan(Nat t1, Nat t2) {
 %match(t1, t2) {
   x,x           -> { return false; }
   suc(x),zero() -> { return true; }
   zero(),suc(y) -> { return false; }
   suc(x),suc(y) -> { return `greaterThan(x,y); }
 }
}
```

In this combination of an ocean language (in our case Java) and island features (in our case abstract data types and matching), it is still an open question to perform type checking and type inference.

Since we want to allow for type inclusion at the pattern level, the first purpose of this paper is to present an extension of the signature definition mechanism allowing for subtypes. In this context we propose an algorithm based on unification of equality constraints [7] and simplification of subtype constraints [4, 1, 9]. It infers the types of the variables that occur in a pattern (x and y in the previous example). Moreover, we also propose an algorithm that checks that the patterns occurring in a Tom program are correctly typed.

2 Type checking

Given a signature Σ_v, the (simplified) abstract syntax of a Tom program is as follows:

$$\begin{array}{lcl} \mathit{rule} & ::= & \mathit{cond} \longrightarrow \mathit{action} \\ \mathit{cond} & ::= & \mathit{term}_1 \prec\!\!\!\prec_{[s]} \mathit{term}_2 \mid \mathit{cond}_1 \wedge \mathit{cond}_2 \\ \mathit{term} & ::= & x \mid f(\mathit{term}_1, \ldots, \mathit{term}_n) \\ \mathit{action} & ::= & (x_1, \ldots, x_n) \end{array}$$

The left-hand side of a rule is a conjunction of matching conditions $\mathit{term}_1 \prec\!\!\!\prec_{[s]} \mathit{term}_2$ consisting of a pair of terms and where s denotes a sort. Since we allow for some symbols to be associative, we introduce two kinds of symbols. Fixed arity ones to denote free symbols and variadic symbols to denote associative ones. We denote these two kinds of symbol sets $\mathscr{F}$ and $\mathscr{F}_v$ respectively. Terms are many-sorted variadic terms composed of variables $x \in \mathscr{X}$ and function symbols $f \in \mathscr{F} \cup \mathscr{F}_v$. The set of terms is written $\mathscr{T}(\mathscr{F} \cup \mathscr{F}_v, \mathscr{X})$. In the following, we often write l a variadic operator and call it a *list*. In general, an *action* is a Java statement, but for our purpose we can consider an abstraction described by the variables $x_1, \ldots, x_n \in \mathscr{X}$ whose instantiations are described by the conditions, and used in the Java statement.

Example 2.1. *The last rule of the* `greaterThan` *function given above can be represented by the following rule expression:*

$$\mathit{suc}(x) \prec\!\!\!\prec_{[\mathbb{N}]} t_1 \wedge \mathit{suc}(y) \prec\!\!\!\prec_{[\mathbb{N}]} t_2 \longrightarrow (x, y)$$

In a first step, we consider that a *context* Γ is composed of a set of pairs (variable,sort), and (function symbol,signature):

$$\Gamma ::= \varnothing \mid \Gamma_1 \cup \Gamma_2 \mid x : s \mid f : s_1, \ldots, s_n \to s$$

We denote by $\Gamma(x : s)$ the fact that $x : s$ belongs to Γ. Similarly, $\Gamma(f : s_1, \ldots, s_n \to s)$ means that $f : s_1, \ldots, s_n \to s$ belongs to Γ. In Fig. 1 we give a classical type checking system defined by a set of inference rules. Starting from a context Γ and a rule expression π, we say that π is well-typed if $\pi : \mathit{wt}$ can be derived by applying the inference rules. *wt* is a special sort that corresponds to the well-typedness of a *rule* or a condition *cond*.

$$\frac{}{\Gamma(x:s)\vdash x:s}\ \text{T-VAR} \qquad \frac{\Gamma\vdash e_1:s_1 \quad \ldots \quad \Gamma\vdash e_n:s_n}{\Gamma(f:s_1,\ldots,s_n\to s)\vdash f(e_1,\ldots,e_n):s}\ \text{T-FUN}$$

$$\frac{\Gamma\vdash e_1:s \qquad \Gamma\vdash e_2:s}{\Gamma\vdash (e_1 \ll_{[s]} e_2):wt}\ \text{T-MATCH} \qquad \frac{\Gamma\vdash (cond_1):wt \quad \ldots \quad \Gamma\vdash (cond_n):wt}{\Gamma\vdash (cond_1\wedge\ldots\wedge cond_n):wt}\ \text{T-CONJ}$$

$$\frac{\Gamma\vdash (cond):wt \qquad \Gamma\vdash e_1:s_1 \quad \ldots \quad \Gamma\vdash e_n:s_n}{\Gamma\vdash (cond\longrightarrow (e_1,\ldots,e_n)):wt}\ \text{T-RULE}$$

Figure 1: Simple type checking system.

2.1 Subtypes and associative-matching

In order to introduce subtypes in Tom, we define $\mathscr{S}$ as the set of sorts, equipped with a partial order $<:$, called *subtyping*. It is a binary relation on $\mathscr{S}$ that satisfies reflexivity, transitivity and antisymmetry.

We extend matching over lists (*i.e.* variadic operators) to be associative. Therefore a pattern matches a subject considering equality relation modulo flattening. Lists can be denoted by function symbols $l\in\mathscr{F}_v$, as said previously, or by variables $x\in\mathscr{X}$ annotated by *. Such variables, which we write x^*, are called *star variables*. So we consider in the following many-sorted variadic terms composed of variables $x\in\mathscr{X}$, star variables x^* (where $x\in\mathscr{X}$) and function symbols $f\in\mathscr{F}\cup\mathscr{F}_v$. Moreover, we define that function symbols $l\in\mathscr{F}_v$ with variable domain (since they have a variable arity) of sort s_1 and codomain s are written $l:{s_1}^*\to s$ while star variables x^* are also sorted and written $x^*:s$.

Since terms built from syntactic and variadic operators can have the same codomain, we cannot distinguish one from the other only by theirs sorts. However, this is necessary to know which typing rule applies. For this purpose, we introduce a notion of sorts decorated with function symbols, called *types*, to classify terms. The special symbol ? is used as decoration when is not useful to know what the function symbol is. This leads to a new set of decorated sorts $\mathscr{D}$.

Given these notions, we define a context Γ by the following grammar:

$$\Gamma ::= \varnothing \mid \Gamma_1\cup\Gamma_2 \mid s_1<:s_2 \mid x:s^? \mid x^*:s^g \mid f:s_1^?,\ldots,s_n^?\to s^f \mid l:(s_1^?)^*\to s^l$$

and context access is defined by the function $\texttt{sortOf}(\Gamma,e):\Gamma\times\mathscr{T}(\mathscr{F}\cup\mathscr{F}_v,\mathscr{X})\to\mathscr{D}$ which returns the type of term e:

$$\begin{array}{rclcrcl}\texttt{sortOf}(\Gamma,x) & = & s^?, \text{ if } x:s^?\in\Gamma & \qquad & \texttt{sortOf}(\Gamma,f(e_1,\ldots,e_n)) & = & s^f, \text{ if } f:s_1^?,\ldots,s_n^?\to s^f\in\Gamma\\ \texttt{sortOf}(\Gamma,x^*) & = & s^g, \text{ if } x^*:s^g\in\Gamma & & \texttt{sortOf}(\Gamma,l(e_1,\ldots,e_n)) & = & s^l, \text{ if } l:(S^?)^*\to s^f\in\Gamma\end{array}$$

where $x\in\mathscr{X}$, $f\in\mathscr{F}$, $l\in\mathscr{F}_v$, $g\in\mathscr{F}\cup\mathscr{F}_V$ and $s^?,s_i^?,s^f,s^g,s^l\in\mathscr{D}$ for $i\in\{1,2,\ldots,n\}$.

The context has at most one declaration of type or signature per term since overloading is forbidden. This means that for $e\in\mathscr{T}(\mathscr{F}\cup\mathscr{F}_v,\mathscr{X})$ and $s_1^{g_1},s_2^{g_2}$ (where $g_1,g_2\in\mathscr{F}\cup\mathscr{F}_v\cup\{?\}$ and $s_1^{g_1},s_2^{g_2}\in\mathscr{D}$) if $e:s_1^{g_1}\in\Gamma$ and $e:s_2^{g_2}\in\Gamma$ then $s_1^{g_1}=s_2^{g_2}$.

2.2 Type checking algorithm

In Fig. 2 we give a type checking system to many-sorted variadic terms applying associative matching. The rules are standard except for the use of decorated types. The most interesting rules are those ones applying to lists. They are three: [T-EMPTY] checks if a empty list has the same type declared in Γ;

[T-ELEM] is similar to [T-FUN] but is applied to lists; and [T-MERGE] is applied to a concatenation of two lists of type s^l in Γ, resulting in a new list with same type s^l.

$$\frac{}{\Gamma(x : s^?) \vdash x : s^?}\ \text{T-VAR} \qquad \frac{}{\Gamma(x^* : s^l) \vdash x^* : s^l}\ \text{T-SVAR}$$

$$\frac{\Gamma \vdash e_1 : s_1^? \qquad \dots \qquad \Gamma \vdash e_n : s_n^?}{\Gamma(f : s_1^?, \dots, s_n^? \to s^f) \vdash f(e_1, \dots, e_n) : s^f}\ \text{T-FUN}$$

$$\frac{}{\Gamma(l : (s_1^?)^* \to s^l) \vdash l() : s^l}\ \text{T-EMPTY}$$

$$\frac{\Gamma \vdash l(e_1, \dots, e_n) : s^l \qquad \Gamma \vdash e : s_1^?}{\Gamma(l : (s_1^?)^* \to s^l) \vdash l(e_1, \dots, e_n, e) : s^l}\ \text{T-ELEM}$$
$$\text{if } \texttt{sortOf}(\Gamma, e) \neq s^l$$

$$\frac{\Gamma \vdash l(e_1, \dots, e_n) : s^l \qquad \Gamma \vdash e : s^l}{\Gamma(l : (s_1^?)^* \to s^l) \vdash l(e_1, \dots, e_n, e) : s^l}\ \text{T-MERGE}$$

$$\frac{\Gamma \vdash e : s_1^?}{\Gamma(s_1 <: s) \vdash e : s^?}\ \text{SUB} \qquad \frac{\Gamma \vdash e : s^h}{\Gamma \vdash e : s^?}\ \text{GEN}$$
$$\text{if } \texttt{sortOf}(\Gamma, e) = s^h, \text{ for } h \in \mathscr{F} \cup \mathscr{F}_v$$

$$\frac{\Gamma \vdash e_1 : s^? \qquad \Gamma \vdash e_2 : s^?}{\Gamma \vdash (e_1 \ \prec_{[s^?]}\ e_2) : wt}\ \text{T-MATCH}$$

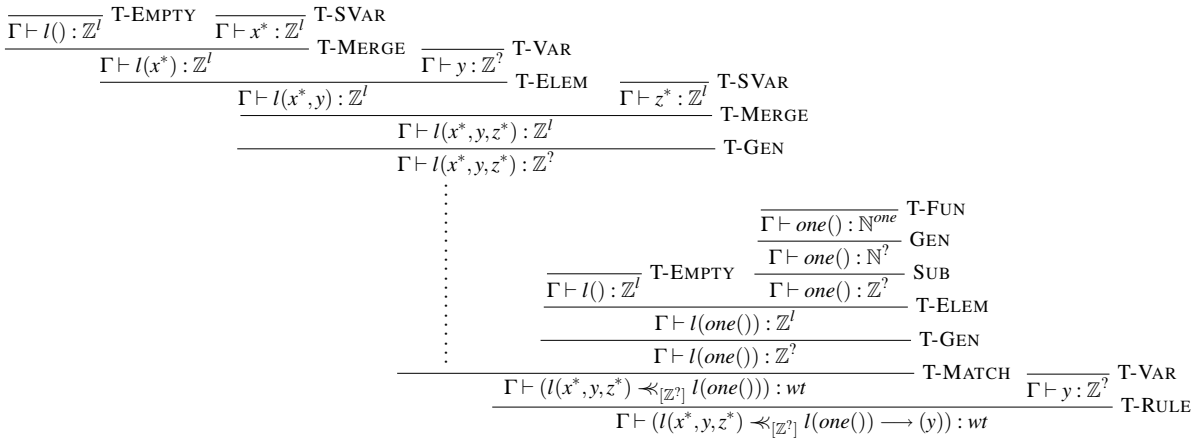
$$\frac{\Gamma \vdash (cond_1) : wt \qquad \dots \qquad \Gamma \vdash (cond_n) : wt}{\Gamma \vdash (cond_1 \wedge \dots \wedge cond_n) : wt}\ \text{T-CONJ}$$

$$\frac{\Gamma \vdash (cond) : wt \qquad \Gamma \vdash e_1 : s_1^{g_1} \qquad \dots \qquad \Gamma \vdash e_n : s_n^{g_n}}{\Gamma \vdash (cond \longrightarrow (e_1, \dots, e_n)) : wt}\ \text{T-RULE}$$
$$\text{if } \texttt{sortOf}(\Gamma, e_i) = s_i^{g_1}, \text{ for } g_i \in \mathscr{F} \cup \mathscr{F}_v \cup \{?\} \text{ and } i \in [1, n]$$

Figure 2: Type checking rules.

The type checking algorithm reads derivations in a bottom-up way. Since the rule [SUB] can be applied to any kind of term, we consider a strategy where it is applied iff no other typing rule can be applied. In practice, [SUB] will be combined with [T-VAR], [T-FUN] and [T-ELEM] and the type $s_1^?$ which appears in the premise will be defined according to the result of function $\texttt{sortOf}(\Gamma, e)$. The algorithm stops if it reaches the [T-VAR] or [T-SVAR] cases, ensuring that the original expression is well-typed, or if none of the type checking rules can be applied, raising an error.

Example 2.2. *Let* $\Gamma = \{l : (\mathbb{Z}^?)^* \to \mathbb{Z}^l, one : \to \mathbb{N}^{one}, x^* : \mathbb{Z}^l, z^* : \mathbb{Z}^l, y : \mathbb{Z}^?, \mathbb{N} <: \mathbb{Z}\}$. *Then the expression* $l(x^*, y, z^*) \prec_{[\mathbb{Z}^?]} l(one()) \longrightarrow (y)$ *is well-typed and its deduction tree is:*



3 Type inference

The type system presented in Section 2 needs rules to control its use in order to find the expected deduction tree of an expression. Without these rules, it is possible to find more than one deduction tree for the same expression. For instance, in Example 2.2 the rule [SUB] can be applied to the leaves resulting of application of rule [T-VAR]. The resulting tree will still be a valid deduction tree since the variables in the leaves will have type $\mathbb{N}^?$ instead of type $\mathbb{Z}^?$ declared in the context and $\mathbb{N} <: \mathbb{Z}$. For that reason, we are interested in defining another type system able to infer the most general types of terms. We add type variables in the set of types (defined up to here as a set of decorated sorts) to describe a possibly infinite set of decorated sorts. The set of types $\mathscr{T}_{ype}(\mathscr{D} \cup \{wt\}, \mathscr{V})$ is given by a set of decorated sorts $\mathscr{D}$, a set of type variables $\mathscr{V}$ and a special sort wt:

$$\tau ::= \alpha \mid s^g \mid wt$$

where $\tau \in \mathscr{T}_{ype}(\mathscr{D} \cup \{wt\}, \mathscr{V})$, $\alpha \in \mathscr{V}$, $g \in \mathscr{F} \cup \mathscr{F}_v \cup \{?\}$ and $s^g \in \mathscr{D}$.

In order to build the subtyping rule into the rules, we use a *constraint set* C to store all equality and subtyping constraints. These constraints limit types that terms can have. The language $\mathscr{C}$ is built from the set of types $\mathscr{T}_{ype}(\mathscr{D} \cup \{wt\}, \mathscr{V})$ and the operators "$=_s$" and "$<:_s$":

$$c ::= \tau_1 =_s \tau_2 \mid \tau_1 <:_s \tau_2$$

where $c \in \mathscr{C}$, $\tau_1, \tau_2 \in \mathscr{T}_{ype}(\mathscr{D} \cup \{wt\}, \mathscr{V})$.

A substitution σ is said to *satisfy* an equation $\tau_1 =_s \tau_2$ if $\sigma\tau_1 = \sigma\tau_2$. Moreover, σ is said to *satisfy* a subtype relation $\tau_1 <:_s \tau_2$ if $|\sigma\tau_1| <: |\sigma\tau_2|$, where function $|\_| : \mathscr{T}_{ype}(\mathscr{D} \cup \{wt\}, \mathscr{V}) \to \mathscr{S}$ is used to remove the decoration of decorated types and it is defined by $|s^g| = s$. Thus, σ satisfies C if it satisfies all constraints in C. This is written $\sigma \models C$. The set $\mathscr{V}(C)$ denotes the set of type variables in C.

Constraints are calculated according to application of rules of type inference system given in Fig. 3 where we can read the judgment $\Gamma \vdash_{ct} e : \tau \bullet C$ as "the term e has type τ under assumptions Γ whenever the constraints C are satisfied". More formally, this judgment states that $\forall \sigma \,.\, (\sigma \models C \to \sigma\Gamma \vdash e : \sigma\tau)$.

3.1 Type inference algorithm

In Fig. 3 we give a type inference system with constraints. Each type variable introduced in a sub-derivation is a fresh type variable and the fresh type variables in different sub-derivations are distinct.

As in Section 2.2, we explain the rules concerning lists: [CT-EMPTY] infers for an empty list $l()$ a type variable α with the constraint $\alpha = s^l$, s^l given by the signature of l; [CT-ELEM] treats applications of lists to elements which are neither lists with the same function symbol nor star variables; [CT-MERGE] is applied to concatenate two lists of same type s^l; and [CT-STAR] is applied to concatenate a list and a star variable of the same type s^l.

$$\frac{}{\Gamma(x:\tau) \vdash_{ct} x : \alpha \bullet \{\alpha = \tau\}}\ \text{CT-VAR} \qquad \frac{}{\Gamma(x^*:\alpha_1) \vdash_{ct} x^* : \alpha \bullet \{\alpha_1 = \alpha\}}\ \text{CT-SVAR}$$

$$\frac{\Gamma \vdash_{ct} e_1 : \alpha_1 \bullet C_1 \quad \dots \quad \Gamma \vdash_{ct} e_n : \alpha_n \bullet C_n \quad C = \{\alpha = s^f\} \bigcup_{i=1}^{n} C_i \cup \{\alpha_i <:_s s_i^?\}}{\Gamma(f : s_1^?, \dots, s_n^? \to s^f) \vdash_{ct} f(e_1, \dots, e_n) : \alpha \bullet C}\ \text{CT-FUN}$$

$$\frac{}{\Gamma(l : (s_1^?)^* \to s^l) \vdash_{ct} l() : \alpha \bullet \{\alpha = s^l\}}\ \text{CT-EMPTY}$$

$$\frac{\Gamma \vdash_{ct} l(e_1, \dots, e_n) : \alpha \bullet C_1 \quad \Gamma \vdash_{ct} e : \alpha_1 \bullet C_2 \quad C = C_1 \cup C_2 \cup \{\alpha = s^l, \alpha_1 <:_s s_1^?\}}{\Gamma(l : (s_1^?)^* \to s^l) \vdash_{ct} l(e_1, \dots, e_n, e) : \alpha \bullet C \quad \text{if } \texttt{sortOf}(\Gamma, e) \neq s^l \text{ and } e \neq x^*}\ \text{CT-ELEM}$$

$$\frac{\Gamma \vdash_{ct} l(e_1, \dots, e_n) : \alpha \bullet C_1 \quad \Gamma \vdash_{ct} e : \alpha \bullet C_2 \quad C = C_1 \cup C_2 \cup \{\alpha = s^l\}}{\Gamma(l : (s_1^?)^* \to s^l) \vdash_{ct} l(e_1, \dots, e_n, e) : \alpha \bullet C \quad \text{if } \texttt{sortOf}(\Gamma, e) = s^l}\ \text{CT-MERGE}$$

$$\frac{\Gamma \vdash_{ct} l(e_1, \dots, e_n) : \alpha \bullet C_1 \quad \Gamma \vdash_{ct} x^* : \alpha_1 \bullet C_2 \quad C = C_1 \cup C_2 \cup \{\alpha = s^l, \alpha_1 = s^l\}}{\Gamma(l : (s_1^?)^* \to s^l) \vdash_{ct} l(e_1, \dots, e_n, x^*) : \alpha \bullet C \quad \text{if } \texttt{sortOf}(\Gamma, x^*) \neq s^l}\ \text{CT-STAR}$$

$$\frac{\Gamma \vdash_{ct} e_1 : \alpha_1 \bullet C_1 \quad \Gamma \vdash_{ct} e_2 : \alpha_2 \bullet C_2 \quad C = C_1 \cup C_2 \cup \{\alpha_1 <:_s \tau, \alpha_2 = \tau\}}{\Gamma \vdash_{ct} (e_1 \prec_{[\tau]} e_2) : wt \bullet C}\ \text{CT-MATCH}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet C_1 \quad \dots \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet C_n \quad C = \bigcup_{i=1}^{n} C_i}{\Gamma \vdash_{ct} (cond_1 \wedge \dots \wedge cond_n) : wt \bullet C}\ \text{CT-CONJ}$$

$$\frac{\Gamma \vdash_{ct} (cond) : wt \bullet C_{cond} \quad \Gamma \vdash_{ct} e_1 : \tau_1 \bullet C_1 \quad \dots \quad \Gamma \vdash_{ct} e_n : \tau_n \bullet C_n \quad C = C_{cond} \bigcup_{i=1}^{n} C_i}{\Gamma \vdash_{ct} (cond \longrightarrow (e_1, \dots, e_n)) : wt \bullet C \quad \text{if } \texttt{sortOf}(\Gamma, e_i) = \tau_i, \text{ for } i \in [1, n]}\ \text{CT-RULE}$$

Figure 3: Type inference rules.

Example 3.1. *Let* $\Gamma = \{l : (\mathbb{Z}^?)^* \to \mathbb{Z}^l, one : \to \mathbb{N}^{one}, x^* : \alpha_1, y : \alpha_2, z^* : \alpha_3, \mathbb{N} <: \mathbb{Z}\}$. *Then the expression* $l(x^*, y, z^*) \prec_{[\alpha_4]} l(one()) \longrightarrow (y)$ *is well-typed and the deduction tree is:*

$$\dfrac{\dfrac{}{\Gamma \vdash_{ct} l() : \alpha_5 \bullet \{\alpha_5 = \mathbb{Z}^l\}}\ \text{CT-EMPTY} \quad \dfrac{}{\Gamma \vdash_{ct} x^* : \alpha_1 \bullet \{\alpha_{10} = \alpha_1\}}\ \text{CT-SVAR}}{\dfrac{C_2 = \{\alpha_5 = \mathbb{Z}^l\} \cup \{\alpha_{10} = \alpha_1\} \cup \{\alpha_5 = \mathbb{Z}^l, \alpha_{10} = \mathbb{Z}^l\}}{\Gamma \vdash_{ct} l(x^*) : \alpha_5 \bullet C_2}\ \text{CT-STAR} \quad \dfrac{}{\Gamma \vdash_{ct} y : \alpha_9 \bullet \{\alpha_9 = \alpha_2\}}\ \text{CT-VAR}}{\dfrac{C_1 = C_2 \cup \{\alpha_9 = \alpha_2\} \cup \{\alpha_5 = \mathbb{Z}^l, \alpha_9 <:_s \mathbb{Z}^?\}}{\Gamma \vdash_{ct} l(x^*, y) : \alpha_5 \bullet C_1}\ \text{CT-ELEM}}$$

$$\vdots$$

$$\dfrac{\dfrac{C_p = C_1 \cup \{\alpha_8 = \alpha_3\} \cup \{\alpha_5 = \mathbb{Z}^l, \alpha_8 = \mathbb{Z}^l\}}{\Gamma \vdash_{ct} l(x^*, y, z^*) : \alpha_5 \bullet C_p}\ \text{CT-STAR} \quad \dfrac{}{\Gamma \vdash_{ct} z^* : \alpha_8 \bullet \{\alpha_8 = \alpha_3\}}\ \text{CT-SVAR}}{}$$

(1)

$$\dfrac{\dfrac{}{\Gamma \vdash_{ct} l() : \alpha_6 \bullet \{\alpha_6 = \mathbb{Z}^l\}}\ \text{CT-EMPTY} \quad \dfrac{}{\Gamma \vdash_{ct} one() : \alpha_7 \bullet \{\alpha_7 = \mathbb{N}^{one}\}}\ \text{CT-FUN}}{\dfrac{C_s = \{\alpha_6 = \mathbb{Z}^l\} \cup \{\alpha_7 = \mathbb{N}^{one}\} \cup \{\alpha_6 = \mathbb{Z}^l, \alpha_7 <:_s \mathbb{Z}^?\}}{\Gamma \vdash_{ct} l(one()) : \alpha_6 \bullet C_s}\ \text{CT-ELEM}}$$

(2)

$$\dfrac{\dfrac{\textit{(1)} \quad \textit{(2)} \quad C_{cond} = C_p \cup C_s \cup \{\alpha_5 <:_s \alpha_4, \alpha_6 = \alpha_4\}}{\Gamma \vdash_{ct} (l(x^*, y, z^*) \prec\!\!\prec_{[\alpha_4]} l(one())) : wt \bullet C_{cond}}\ \text{CT-MATCH} \quad \dfrac{}{\Gamma \vdash_{ct} y : \alpha_2 \bullet \{\alpha_2 = \alpha_2\}}\ \text{CT-VAR}}{\dfrac{C_r = C_{cond} \cup \{\alpha_2 = \alpha_2\}}{\Gamma \vdash_{ct} (l(x^*, y, z^*) \prec\!\!\prec_{[\alpha_4]} l(one()) \longrightarrow (y)) : wt \bullet C_r}\ \text{CT-RULE}}$$

3.2 Constraint resolution

To determine if a rule expression is well-typed, its constraint set C needs to be solved in order to generate a *most general solution* σ of C from which all solutions can be generated straightforwardly. The substitution σ is said to be the *most general solution* for C if:

1. σ is a solution for C which means that $\sigma \models C$; and
2. for all solutions σ' for C, $\sigma'\alpha <: \sigma\alpha$ for all $\alpha \in \mathscr{V}$ in C.

The rules for the constraint resolution algorithm are provided in Fig. 4, where $g, g_1, g_2 \in \mathscr{F} \cup \mathscr{F}_v \cup \{?\}$. The algorithm starts by applying closure in Γ which means it generates an assertion $s_1 <: s_3$ in Γ whenever $\{s_1 <: s_2, s_2 <: s_3\} \subseteq \Gamma$ and two other assertions $s_1 <: s_1$ and $s_2 <: s_2$ in Γ whenever $s_1 <: s_2 \in \Gamma$. Then, rules (1)-(11) are recursively applied over C. More precisely, rules (1)-(3) work as a garbage collector removing constraints that are no more useful. Rules (4) and (5) generate σ. Rules (6) and (7) generate simplified constraints. Rules (8) and (9) treat failure. Rules (10) and (11) are applied when none of previous rules can be applied generating a new σ from a constraint over a type variable that has no other constraints. The algorithm stops if: a rule returns $C = \varnothing$, then the algorithm returns the most general solution σ; or if a rule returns $fail$ or C reaches a stable form, then the algorithm returns an error.

Example 3.2. *Let* $\Gamma = \{l : (\mathbb{Z}^?)^* \to \mathbb{Z}^l, one : \to \mathbb{N}^{one}, x^* : \alpha_1, y : \alpha_2, z^* : \alpha_3, \mathbb{N} <: \mathbb{Z}\}$ *and* $C_{cond} = \{\alpha_5 = \mathbb{Z}^l, \alpha_{10} = \alpha_1, \alpha_5 = \mathbb{Z}^l, \alpha_{10} = \mathbb{Z}^l, \alpha_9 = \alpha_2, \alpha_5 = \mathbb{Z}^l, \alpha_9 <:_s \mathbb{Z}^?, \alpha_8 = \alpha_3, \alpha_5 = \mathbb{Z}^l, \alpha_8 = \mathbb{Z}^l, \alpha_6 = \mathbb{Z}^l, \alpha_7 = \mathbb{N}^{one}, \alpha_6 = \mathbb{Z}^l, \alpha_7 <:_s \mathbb{Z}^?, \alpha_5 <:_s \alpha_4, \alpha_6 = \alpha_4, \alpha_2 = \alpha_2\}$ *from the Example 3.1. Let* $\sigma = \varnothing$ *and* $C = C_{cond}$. *The constraint resolution algorithm starts by:*

(1)	$\{\tau = \tau\} \uplus C', \sigma$	$\Longrightarrow$	C', σ
(2)	$\{\tau <:_s \tau\} \uplus C', \sigma$	$\Longrightarrow$	C', σ
(3)	$\{s_1^{g1} <:_s s_2^{g2}\} \uplus C', \sigma$	$\Longrightarrow$	C', σ if $\lvert s_1^{g1}\rvert <: \lvert s_2^{g2}\rvert \in \Gamma$
(4)	$\{\alpha = \tau\} \uplus C', \sigma$	$\Longrightarrow$	$[\alpha \mapsto \tau]C', \{\alpha \mapsto \tau\} \cup \sigma$
(5)	$\{\tau = \alpha\} \uplus C', \sigma$	$\Longrightarrow$	$[\alpha \mapsto \tau]C', \{\alpha \mapsto \tau\} \cup \sigma$
(6)	$\{\tau_1 <:_s \tau_2, \tau_2 <:_s \tau_1\} \uplus C', \sigma$	$\Longrightarrow$	$\{\tau_1 = \tau_2\} \cup C', \sigma$
(7)	$\{\alpha <:_s s_1^{g1}, \alpha <:_s s_2^{g2}\} \uplus C', \sigma$	$\Longrightarrow$	$\{\alpha <:_s min(s_1^{g1}, s_2^{g2})\} \cup C', \sigma$
(8)	$\{s_1^{g1} = s_2^{g2}\} \cup C', \sigma$	$\Longrightarrow$	$fail$ if $s_1^{g1} \neq s_2^{g2}$
(9)	$\{s_1^{g1} <:_s s_2^{g2}\} \cup C', \sigma$	$\Longrightarrow$	$fail$ if $\lvert s_1^{g1}\rvert <: \lvert s_2^{g2}\rvert \notin \Gamma$
(10)	$\{\alpha <:_s s^g\} \uplus C', \sigma$	$\Longrightarrow$	$\{\alpha = s^g\} \cup C', \{\alpha \mapsto s^g\} \cup \sigma$ if $\alpha \notin \mathscr{V}(C')$
(11)	$\{s^g <:_s \alpha\} \uplus C', \sigma$	$\Longrightarrow$	$\{s^g = \alpha\} \cup C', \{\alpha \mapsto s^g\} \cup \sigma$ if $\alpha \notin \mathscr{V}(C')$

Figure 4: Constraint resolution rules.

1. *Application of closure in* Γ*, generating* $\mathbb{N} <: \mathbb{N}$ *and* $\mathbb{Z} <: \mathbb{Z}$*;*
2. *Application of rules (1), (4) and (5) generating* $\{\mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \alpha_2 = \alpha_2, \mathbb{Z}^l = \mathbb{Z}^l, \alpha_2 <:_s \mathbb{Z}^?, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{N}^{one} = \mathbb{N}^{one}, \alpha_6 = \mathbb{Z}^l, \mathbb{N}^{one} <:_s \mathbb{Z}^?, \mathbb{Z}^l <:_s \mathbb{Z}^l, \alpha_6 = \mathbb{Z}^l\} \cup C$ *and* $\{\alpha_5 \mapsto \mathbb{Z}^l, \alpha_{10} \mapsto \alpha_1, \alpha_1 \mapsto \mathbb{Z}^l, \alpha_9 \mapsto \alpha_2, \alpha_8 \mapsto \alpha_3, \alpha_3 \mapsto \mathbb{Z}^l, \alpha_6 \mapsto \mathbb{Z}^l, \alpha_7 \mapsto \mathbb{N}^{one}, \alpha_4 \mapsto \mathbb{Z}^l\} \cup \sigma$
3. *Application of rules (1), (2) and (3) generating* $\{\alpha_2 <:_s \mathbb{Z}^?\}$ *and* σ*;*
4. *Application of rule (10) generating* $\varnothing$ *and* $\{\alpha_2 \mapsto \mathbb{Z}^?\} \cup \sigma$*, the algorithm then stops and returns* σ *providing a substitution for all type variables in the deduction tree of* $l(x^*, y, z^*) \prec\!\!\!\prec_{[\alpha_4]} l(one()) \longrightarrow (y)$.

4 Properties

Since our type checking system and our type inference system address the same issue, we must check two properties. First, we show that every typing judgment that can be derived from the inference rules also follows from the checking rules (Theorem 4.2), in particular the soundness. Then we show that a solution given by the checking rules can be extended to a solution proposed by the inference rules (Theorem 4.4).

Definition 4.1 (Solution)**.** *Let* Γ *be a context and e a term. A* solution for (Γ, e) *is a pair* (σ, s_1^g) *such that* $\sigma\Gamma \vdash e : s_1^g$*. Moreover, suppose that* $\Gamma \vdash e : \tau \bullet C$*. A* solution for (Γ, e, τ, C) *is a pair* (σ, s_2^g) *such that* σ *satisfies C and* $\lvert\sigma\tau\rvert <: s_2^g$*, where* $g \in \mathscr{F} \cup \mathscr{F}_v \cup \{?\}$ *and* $\tau, s_1^c, s_2^c \in \mathscr{T}_{ype}(\mathscr{D} \cup \{wt\}, \mathscr{V})$.

Theorem 4.2 (Soundness of constraint typing)**.** *Suppose that* $\Gamma \vdash_{ct} e : \tau \bullet C$*. If* (σ, s^g) *is a solution for* (Γ, e, τ, C)*, then it is also a solution for* (Γ, e).

Proof. By induction on the given constraint typing derivation for $\Gamma \vdash_{ct} e : \tau \bullet C$. □

Definition 4.3 (Normal form of typing derivation)**.** *A typing derivation is in* normal form *if it does not have successive applications of rule* [SUB].

Theorem 4.4 (Completeness of constraint typing)**.** *Suppose that $\pi = \Gamma \vdash_{ct} e : \tau \bullet C$ and write $V(\pi)$ for the set of all type variables mentioned in the last rule used to derive π. If (σ, s^g) is a solution for (Γ, e) and $dom(\sigma) \cap V(\pi) = \varnothing$, then there is some solution (σ', s^g) for (Γ, e, τ, C) such that $\sigma' = \sigma \cup V(\pi)$.*

Proof. By induction on the given constraint typing derivation in normal form, but we must take care with fresh names of variables. □

Proposition 4.5 (Uniqueness of type)**.** *Suppose that $\Gamma \vdash_{ct} e : \tau \bullet C$. If there are two solutions $(\sigma_1, s_1^{g_1})$ and $(\sigma_2, s_2^{g_2})$ for (Γ, e, τ, C) where σ_1 and σ_2 are two most general solutions for C then $\sigma_1 = \sigma_2$ and $s_1^{g_1} = s_2^{g_2}$.*

Theorem 4.6 (Termination of algorithm)**.** *The constraint resolution algorithm always terminates, failing when given a non-satisfiable constraint set as input and otherwise returning the most general solution. More formally:*

1. *the algorithm halts, either by failing or by returning a substitution, for all C;*
2. *if the algorithm returns a σ, then σ is a solution for C;*
3. *if there exists a σ' solution for C, then the algorithm returns a σ and $\sigma'\alpha <: \sigma\alpha$ for all $\alpha \in \mathcal{V}$ in C.*

We can already sketch a proof of Theorem 4.6 following Pierce [8].

Proof. For part 1, define the *degree* of a constraint set C to be the pair (m,n), where m is the number of constraints in C and n is the number of subtyping constraints in C. The algorithm terminates immediately (with success in the case of an empty constraint set or failure for an equation involving two different primitive types) or makes recursive calls to itself with a constraint set of lexicographically smaller degree.

For part 2, by induction on the number of recursive calls in the computation of the algorithm.

For part 3, by induction on the number of recursive calls in the computation of the algorithm, reasoning by cases on the shapes of the types involved in the constraints. □

5 Conclusion

In this paper we have presented a type system for the pattern matching constructs of Tom. The system is composed of type checking and type inference algorithms with subtyping over sorts. Since Tom also implements associative pattern matching over variadic operators, we were interested in defining both a way to distinguish these from syntatic operators and checking and inferring their types.

We have obtained the following: our type inference system is sound and complete w.r.t. checking, showed by Theorems 4.4 and 4.2. This is the first step towards an effective implementation, thus leading to a safer Tom. However, we still need to turn the Proposition 4.5 into a theorem, which we do not expect to be too difficult. Moreover, before the implementation of the type inference algorithm, we need to have a formal proof of termination — in contrast to the sketch we currently have — as stated by Theorem 4.6.

As we have considered a subset of the Tom language, future work will focus on extending the type system to handle the other constructions of the language such as anti-patterns [5, 6]. As a slightly more prospective research area, we also want parametric polymorphism over types for Tom: our type system will therefore have to be able to handle that as well.

References

[1] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In *In Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340. IEEE Computer Society Press, 1992.

[2] Emilie Balland. *Conception d'un langage dédié à l'analyse et la transformation de programmes*. PhD thesis, Université Henri Poincaré, 2009.

[3] Emilie Balland, Claude Kirchner, and Pierre-Etienne Moreau. Formal Islands. In Michael Johnson and Varmo Vene, editors, *11th International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *LNCS*, pages 51–65. Springer, 2006.

[4] Duggan Dominic. Finite subtype inference with explicit polymorphism. *Sci. Comput. Program.*, 39(1):57–92, 2001.

[5] Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-pattern matching. In *16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 110–124, Braga, Portugal, 2007. Springer.

[6] Radu Kopetz. *Contraintes d'anti-filtrage et programmation par réécriture*. PhD thesis, Institut National Polytechnique de Lorraine, 2008.

[7] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[8] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. Chapter 22.

[9] François Pottier. Simplifying subtyping constraints: a theory. *Inf. Comput.*, 170(2):153–183, 2001.

Programming Games and their Equilibria in Maude

Christiano Braga
Instituto de Computação, UFF
Brazil
cbraga@ic.uff.br

Edward Hermann Hæusler
Departamento de Informática, PUC-Rio
Brazil
hermann@inf.puc-rio.br

In this short paper we explore how strategic and extensive games, from Game Theory, may be represented as rewrite theories. We also discuss how to calculate Nash and Subgame Perfect Equilibria using term rewriting. We have coded some simple, and yet relevant, games in the rewriting language Maude, an implementation of Rewriting Logic. Our coding technique takes advantage of the Rewriting Logic calculus and rewriting modulo theories implemented in Maude.

1 Introduction

Game Theory (e.g. [12]) is a branch of Mathematics, extensively used by economists and social scientists, well-suited to define and study rational behavior of players in the context of cooperative and non-cooperative environments. Game Theory has been considered in many theoretical notions in Computer Science and Logic (e.g. [16, 14, 13, 1, 3]).

Equilibria notions (after the contributions of John Nash) play a central role in solution concepts regarding a great variety of Games assuming that the players are rational. Game Theory is mainly taken into account when researchers want to predict and study global phenomena from individual known behavior. Typical examples of this kind of research come from Markets, Auctions and Elections (e.g. [2]).

Game Theory provides general Game definitions, such as strategic and extensive games. It also provides solution concepts for each kind of Game, in the form of *equilibria*, pointing out relationships among the different solution concepts.

Algorithms for searching and formally identifying solution concepts in a (generic) game are worth studying for theoretical and pragmatical reasons. From one hand, it is known (e.g. [8]) that the decision problems associated with most of Game Theory solution concepts belong to high complexity classes, starting with the *NP* class, of course, and beyond. On the other hand, it seems to be no better alternative for the social sciences, for instance, mainly when analytical tools do not apply.

Let us start with the classical example of the Prisoner's Dilemma[1](PD). "Two suspects in crime are put in separate cells. If they both confess, each will be sentenced to three years in prison. If only one confess, he will be freed and used as witness against the other, who will receive a sentence of four years. If neither confess, they will both be convicted of a minor offense and spend one year in prison." This situation has the main and essential components of a Game: (i) there is more than one rational player, (ii) there are actions each player can take, and, finally, (iii) there are penalties (or profits) associated with each action. There is also the assumption of *rationality* of their players.

Let us interpret the Prisoner's Dilemma as a *Strategic Game*. In such games, an event happens only once, each player knows the details of the game and, as mentioned before, all players are assumed rational. The players choose their actions *simultaneously* and *independently*. Thus, each player is *unaware*

[1] In this paper we have chosen to present our ideas in an informal way. A more precise, mathematical presentation, will be left to a longer version of this work.

of the choices made by the other players. A player has no information to base his/her expectations of another player's behavior. In this way, strategic games can always be represented by a *matrix*. Figure 1 shows the matrix representation of the Prisoner's Dilemma. The rows represent one of the prisoners (player 1) and columns represent the other (player 2). Each row/column is named after a possible action to be taken by the respective player. The entries in the matrix inform the profit/penalty regarding the respective action taken by each player.

An alternative representation for a game, called *Extensive Game*, considers the actions taken by each player as time elapses. It is usually represented as a tree, where the leafs carry the information on profit or penalty, each internal node is associated with a player and the edges between nodes are labeled after the actions. Figure 2 shows extensive representation of the Prisoner's Dilemma. (Note that any game in extensive form can be also modeled as a strategic game. The strategic game matrix associated with a game in extensive form uses the players histories as rows or columns to index the corresponding leaf of the tree as an entry of the matrix.)

	$\langle C \rangle$	$\langle NC \rangle$
$\langle C \rangle$	2,2	4,0
$\langle NC \rangle$	0,4	3,3

Figure 1: Strategic PD

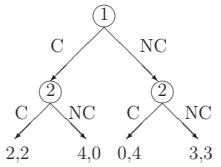


Figure 2: Extensive PD

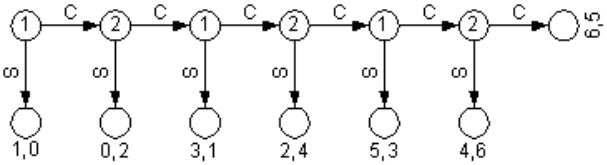


Figure 3: Extensive Centipede

Solution concepts are analysis of games regarding their global configuration. They are usually related with steady Equilibria states in the game. Nash Equilibrium (NE) [12] is one of the most well-known solution concepts. A NE requires from each players strategy to be optimal given the other players strategies. The only NE for the Prisoner's Dilemma is $\langle C, C \rangle$, meaning that both players should confess. When a game is modeled by its extensive form, a more informative solution concept is the Subgame Perfect Equilibrium (SPE) [12]. SPE requires from the action prescribed by each players strategy to be optimal given the other players strategies, after every history. SPE considers the structure of the extensive game explicitly as opposed to NE where the structure is implicit in the definition of the strategies. If one takes NE of the strategic form of an extensive game, one will see that NE *contains* SPE. A detailed study of this fact points out that NE considers *more* strategies than those really taken by rational agents on each turn of the game. NE Equilibrium ignores the sequential structure of the games; it considers the strategies as choices that are made *once and for all* before the game begins. Thus, it is always worth to have *both* solutions while performing a game analysis.

The Centipede game [12], represented in Figure 3 in its extensive representation, is interesting as an example that the SPE equilibrium is not always "reasonable": the SPE in the Centipede game is reached with a payoff of $(1,0)$, which is worse than most of the solutions for both players in the game.

In this article we discuss how strategic and extensive games can be represented as rewrite theories in Rewriting Logic [11] and NE and SPE equilibria computed as rewrites. Our embedding was carefully designed. Rewriting Logic's congruence rule, which may be interpreted as *parallel rewriting*, is crucial for efficiently searching for equilibria. Rewriting modulo axioms has an important rôle in Game representation as rewrite theories. In particular, multiset rewriting, that is, rewriting modulo associativity and commutativity, is essential to produce succinct and elegant Game representations. We have prototyped our embedding in the Maude language [6], a realization of Rewriting Logic. We explore how to use Maude to search for solutions to strategic and extensive games. The contribution of this paper is twofold:

first, we present an embedding from basic Game Theoretic concepts to Rewriting Logic that, by taking advantage of the Rewriting Logic calculus, allows for efficient search for Nash and SP equilibria and second, we present a prototype implementation in the Maude language for simple and yet meaningful examples of basic Games.

This short paper is organized as follows. Section 2 gives background information on the Maude language. Section 3 explains how we have represented strategic games, extensive games, NE equilibrium and SPE equilibrium in Maude. Section 4 concludes this paper with related and future work.

2 Maude

Maude[2] is an implementation of rewriting logic with a concrete syntax quite similar to its mathematical notation.

A signature in rewriting logic is an equational theory (Σ, E), where Σ is an equational signature and E is a set of Σ-equations. Rewriting operates on equivalence classes of terms modulo E. In this way, rewriting is freed from the syntactic constraints of a term representation and gain a much greater flexibility in deciding what counts as a data structure. For example, string rewriting is obtained by imposing an associativity axiom, and multiset rewriting by imposing associativity and commutativity. Standard term rewriting is obtained as the particular case in which the set of equations E is empty. Maude implements techniques for rewriting modulo equations using attributes given in operator declarations, such as associativity, commutativity, identity, and idempotency, to rewrite modulo such axioms.

Given a signature (Σ, E), sentences of rewriting logic are sequents of the form $[t]_E \rightarrow [t']_E$, where t and t' are Σ-terms, possibly involving some variables, and $[t]_E$ denotes the equivalence class of the term t modulo the equations E. A rewrite theory R is a 4-tuple $R = (\Sigma, E, L, R)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of labels, and R is a set of pairs $R \subseteq L \times T_{\Sigma,E}(X)^2$ whose first component is a label and whose second component is a pair of E-equivalence classes of terms, with $X = \{x_1, \ldots, x_n, \ldots\}$ a countably infinite set of variables. Elements of R are called rewrite rules. The rule $(r, ([t], [t']))$ is understood as a labeled sequent, written with the notation $r : [t] \rightarrow [t']$. The rule r: $[t(x_1, \ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)]$ indicates that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in either t or t', or, in abbreviated notation, $r : [t(\overline{x})] \rightarrow [t'(\overline{x})]$.

Given a rewrite theory R, R entails a sentence $[t] \rightarrow [t']$, or that $[t] \rightarrow [t']$ is a *concurrent* R-rewrite. The deduction $R \vdash [t] \rightarrow [t']$ holds if and only if $[t] \rightarrow [t']$ can be obtained by a finite application of the following rules of deduction (where it is assumed that all the terms are well formed and $t(\overline{w}/\overline{x})$ denotes the simultaneous substitution of w_i for x_i in t):

Reflexivity, for each $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] \rightarrow [t]}$,

Congruence, for each $f \in \Sigma_n, n \in \mathbb{N}$, $\dfrac{[t_1] \rightarrow [t'_1] \ldots [t_n] \rightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \rightarrow [f(t'_1, \ldots, t'_n)]}$,

Replacement, for each rule $r : [t(x_1, \ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)] \in R$, $\dfrac{[w_1] \rightarrow [w'_1] \ldots [w_n] \rightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \rightarrow [t'(\overline{w'}/\overline{x})]}$ and

Transitivity $\dfrac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$.

Maude's search command[3] performs a breadth-first search for rewrite proofs starting at a given term to a final term that matches a given pattern. When the search type `=>!` is used, only canonical final terms,

[2] This section adapts material from [5, Section 4.2.1].

[3] This prose adapts text from [7, Section 18.4]

that cannot be further rewritten, are allowed as solutions.

3 Programming Games in Maude

In this section we discuss our Maude implementation for strategic games, extensive games with perfect information, Nash Equilibrium and Subgame Perfect Equilibrium. The Maude code used in the following sections may be downloaded from `http://www.ic.uff.br/~cbraga/rule.maude` and run in the latest version of the Maude system which may be downloaded from `http://maude.cs.uiuc.edu`.

3.1 Programming strategic games

The matrix representation of a strategic game of two players and two actions may be coded in Maude quite straight forwardly by the *abstract* datatype `MatrixRep` constructed by a juxtaposition operation that puts payoffs (pairs of natural numbers) side-by-side. The lines and columns' labels are represented as constants. We also define the constant `matrix` of sort `MatrixRep` to capture the matrix of a particular strategic game. Essentially, the module `MATRIX` is a module parameterized by the payoffs and labels. We discuss the basic 2×2 case first and the general case later.

```
mod MATRIX is pr NAT .
    sorts PayOff MatrixRep Solution Action .
    op _,_ : Nat Nat -> PayOff .  op ____ : PayOff PayOff PayOff PayOff -> MatrixRep .
    ops l1 l2 c1 c2 : -> Action . op matrix : -> MatrixRep .
endm
```

The strategic representation of the Prisioner's Dilema (or simply Dilema) is programmed in Maude in the module `DILEMA`, extending `MATRIX`, that makes the abstract datatype `MatrixRep` concrete with the Dilema's payoffs and labels. Figure 1 shows the strategic representation of the Dilema game.

```
mod DILEMA is ex MATRIX .
    ops C NC : -> Action . eq l1 = C . eq l2 = NC . eq c1 = C . eq c2 = NC .
    eq matrix = 2,2 4,0
                0,4 3,3 .
endm
```

The games Battle of Sexes (where a couple must decide whether to attend a concert of an orchestra playing Bach or another playing Stravinsky) and Matching Pennies (where two children play "head and tail" with a coin and one must pay the other if their choices coincide) may be represented as instances of `MATRIX` in a similar manner.

The general case Given a n-player game, its strategic representation is a n-dimensional matrix. More precisely, it would be a *hyperrectangle*, which is a generalization of a rectangle for higher dimensions, with each player associated with a dimension in the hyperrectangle. The value of each cell in the matrix is a tuple with n components. Each projection of the tuple represents the payoff of the associated player for the actions that the cell represents.

A game with three players and two actions for each player For example, in a three-players strategic game with two actions each, the player in the third dimension would choose one of two possible 2×2 matrices (such as in the Prisioner's Dilema) representing the possible actions of the remaining two players. Figure 4 illustrates such a game. One player may take actions L and R (the column dimension),

another player may take actions T and B (line dimension) and the last player may take actions l and r (a dimension orthogonal to the column-line plane). For each action l and r there is an associated 2×2 matrix representing the possibilities for the remaining players. On each cell there is a triple representing the payoffs with the first projection representing the player associated with the line dimension, the second projection the player in the column dimension and the third projection the player in the orthogonal dimension. The game in Figure 4 has two NE when actions (B,L,l) and (T,L,l) are taken.

	L	R
T	1,1,1	1,0,1
B	1,1,1	0,0,1

l

	L	R
T	1,1,0	0,0,0
B	0,1,0	1,0,0

r

Figure 4: Three-players game

We may represent this game as an extension of the strategic game with two players and two actions. Essentially, our extension to the game representation includes an overloaded declaration for the PayOff constructor and a new multiset of matrices to represent the alternatives of the third player, as in Figure 4.

```
mod MATRIX-3 is ex MATRIX .
    sorts Matrix-3 .
    op _,_,_ : Int Int Int -> PayOff .
    ops o1 o2 : -> Action .
    op matrix-3 : -> Matrix-3 .
    op __ : MatrixRep MatrixRep -> Matrix-3 [assoc comm] .
endm

mod EXAMPLE2 is ex MATRIX-3 .
    ops L R T B l r : -> Action .
    eq matrix-3 = (1,1,1 1,0,1
                   1,1,1 0,0,1)
                  (1,1,0 0,0,0
                   0,1,0 1,0,0) .
    eq l1 = T . eq l2 = B . eq c1 = L . eq c2 = R .
    eq o1 = l . eq o2 = r .
endm
```

A fully generalized representation of strategic game is parameterized by the number of players and the number of actions for each player. It may be implemented in Maude following the ideas for the game with three players and two actions illustrated above. Such a representation is left to an extended version of this paper.

3.2 Programming the Nash equilibrium in strategic games

Intuitively speaking, the Nash equilibrium is a solution for a game when "I am happy when everyone else is happy". The best choice for an individual is also the best choice for everyone. (Pretty much as in the movie "Beatiful mind" when John Nash, played by Russel Crowe, develops a strategy to approach a woman in a bar in such a way that all his friends would also approach the women they wanted to.)

For a two players game with two actions, we defined a rule for each case with the appropriate comparisons of the payoffs. In a two player-two action game, a player must take an action if the payoff of the given action is better than the payoff for the alternative action. The nash equilibrium is achieved in a two

players-two actions game if the situation described in the previous sentence is achieved for *both* players *simultaneously*.

Let us analyse the first rule of module NASH, our specification for the Nash Equilibrium for a two players-two action game. The remaining rules follow the same idea. Variables $\mathtt{N_i}$ are number variables of type integer that represent payoffs. Variable P represents a pair of payoffs and is used here to simplify the readability of the pattern in the rule. The constants $\mathtt{l_1}$ and $\mathtt{c_1}$ are the same ones defined in module MATRIX.

```
crl nash( N1,N2 N3,N4
          N5,N6 P     ) => < l1 , c1 > if N1 >= N5 ∧ N2 >= N4 .
```

The rule compares the payoffs for the first and second players in the first cell c_{11} with the payoffs of their alternatives which are the first projection of the payoff in the cell c_{21} and the second projection in the cell c_{12}, where the indices i and j in c_{ij} represent line and column numbers of the matrix, respectively.

```
mod NASH is ex MATRIX .
    sorts Solution . op nash : MatrixRep -> Solution . op <_,_> : Action Action -> Solution .
    var P : PayOff . vars N1 N2 N3 N4 N5 N6 : Int .
    crl nash( N1,N2 N3,N4
              N5,N6 P     ) => < l1 , c1 > if N1 >= N5 ∧ N2 >= N4 .

    crl nash( N1,N2 N3,N4
              P     N5,N6 ) => < l1 , c2 > if N3 >= N5 ∧ N4 >= N2 .

    crl nash( N1,N2 P
              N3,N4 N5,N6 ) => < l2 , c1 > if N3 >= N1 ∧ N4 >= N6 .

    crl nash( P     N1,N2
              N3,N4 N5,N6 ) => < l2 , c2 > if N5 >= N1 ∧ N6 >= N4 .
endm
```

Note on efficiency Strategic games are called "one-shot" games. In our rewriting semantics this means precisely that they are solved in $O(1)*M$, where M is the complexity of Maude's associativity-commutativity matching algorithm. We refer to [9] for a through discussion on associativity-commutativity matching in Maude.

To calculate the nash equilibrium of a two players-two actions game we must declare yet another module that imports the module that defines the game together with the NASH module. Then, we simply run the search command over the matrix constant looking for canonical forms. Note that a game may have more than one Nash Equilibrium or even none at all. The Dilema has precisely one equilibrium. The Battle of Sexes has two (either the couple decide to attend the Bach concert or decide to attend Stravinsky concert) and Matching Pennies has none, as shown by the following Maude session.

```
==========================================
search in NASH-DILEMA : nash(matrix) =>! S:Solution .
Solution 1 (state 1)
states: 2  rewrites: 10 in 0ms cpu (0ms real) (98039 rewrites/second)
S:Solution --> < C,C >
No more solutions.
states: 2  rewrites: 10 in 0ms cpu (0ms real) (50761 rewrites/second)
==========================================
search in NASH-BS : nash(matrix) =>! S:Solution .
Solution 1 (state 1)
states: 3  rewrites: 13 in 0ms cpu (0ms real) (117117 rewrites/second)
```

```
S:Solution --> < B,B >
Solution 2 (state 2)
states: 3  rewrites: 13 in 0ms cpu (0ms real) (71823 rewrites/second)
S:Solution --> < S,S >
No more solutions.
states: 3  rewrites: 13 in 0ms cpu (0ms real) (48327 rewrites/second)
==========================================
search in NASH-MP : nash(matrix) =>! S:Solution .
Solution 1 (state 0)
states: 1  rewrites: 7 in 0ms cpu (0ms real) (92105 rewrites/second)
S:Solution --> nash(1,-1 -1,1 -1,1 1,-1)
No more solutions.
states: 1  rewrites: 7 in 0ms cpu (0ms real) (36842 rewrites/second)
```

The NE for a game with three players and two actions The implementation for the NE in the three players game follows the same idea of the two players game by comparing the projections of the PayOff tuple for each cell appropriately. Now, for the third player, in a given cell, we need to compare the third projection of the tuple of that cell of *each matrix*. Note that variables `N3` and `M3` represent the third projection of the tuple on each matrix. Due to space constraints, we show only two of the rules as the remaining ones follow the same idea. The complete specification can be downloaded from `http://www.ic.uff.br/~cbraga/rule.maude`.

```
mod NASH-3 is pr MATRIX-3 .
    sorts Solution .
    op nash : Matrix-3 -> Solution .
    op <_,_,_> : Action Action Action -> Solution .
    vars P1 P2 : PayOff .
    vars N1 N2 N3 N4 N5 N6 N7 N8 N9
         M1 M2 M3 M4 M5 M6 M7 M8 M9 : Int .

    crl nash( (N1,N2,N3 N4,N5,N6
               N7,N8,N9 P1      )
              (M1,M2,M3 M4,M5,M6
               M7,M8,M9 P2     ) ) => < l1, c1, o1 >
     if N1 >= N7 / N2 >= N5 / N3 >= M3 .

    crl nash( (N1,N2,N3 N4,N5,N6
               N7,N8,N9 P1      )
              (M1,M2,M3 M4,M5,M6
               M7,M8,M9 P2      ) ) => < l1, c1, o2 >
     if N1 >= N7 / N2 >= N5 / N3 < M3 .

    ...
endm
```

Searching for the NE of the game in Figure 4 produces the following Maude session.

```
==========================================
search in NASH-3-EXAMPLE2 : nash(matrix-3) =>! S:Solution .

Solution 1 (state 1)
states: 3  rewrites: 27 in 0ms cpu (0ms real) (150837 rewrites/second)
S:Solution --> < T,L,l >

Solution 2 (state 2)
```

```
states: 3  rewrites: 27 in 0ms cpu (0ms real) (117391 rewrites/second)
S:Solution --> < B,L,l >

No more solutions.
states: 3  rewrites: 27 in 0ms cpu (0ms real) (99630 rewrites/second)
```

Note on general strategic games To the best of our knowledge, when there are more than two players, different solution concepts, such as Core [12, Ch. 13], are used, instead of the NE. Moreover, in such situations, many results regarding equilibria equivalences are not valid [12].

3.3 Programming extensive games with perfect information

In extensive games the players take a decision based on the *history* of the game. Intuitively, if all the information is available the game is called extensive with perfect information. An extensive game is represented as a tree where the nodes are labeled with the turn and the arrows are labeled with the actions. The leaves of the tree hold the payoff of the associated branch. Figure 2 shows the extensive representation of the Dilema game.

We have coded the tree representation of an extensive game in Maude as a set of branches, declared as the sort `BranchSet` constructed with operator `mt-bs` and the composition operator `_,_`. The latter is declared with attributes `assoc`, `comm` and `id: mt-bs`, which means that the operator is associative, commutative and has the operator `mt-bs` as identity. Therefore, rewriting of terms of sort `BranchSet` occurs *modulo* associativity, commutativity and identity.

A branch is a list of pairs turn-action followed by a utility. A branch is represented by the sort `Branch` which is constructed by the operator `_:_` that created a pair of a list of turn-action pairs and a utility. A turn is a natural number and `Action` is an abstract sort.

```
mod EGPI is pr NAT .
 sorts Action TurnAction TurnActionList . subsort TurnAction < TurnActionList .
 sorts Utility . sorts Branch BranchSet . subsort Branch < BranchSet .
 op (_,_) : Nat Nat -> Utility . op (_,_) : Nat Action -> TurnAction .
 op mt-tal : -> TurnActionList .  op mt-bs : -> BranchSet .
 op __ : TurnActionList TurnActionList -> TurnActionList [assoc id: mt-tal prec 10] .
 op _:_ : TurnActionList Utility -> Branch [prec 20] .
 op _,_ : BranchSet BranchSet -> BranchSet [assoc comm id: mt-bs prec 40] .
endm
```

The Dilema game is coded by the following Maude module.

```
mod DILEMA is ex EGPI .
 ops C NC : -> Action .  op dilema : -> BranchSet .
 eq dilema = ((1, C)  (2, C)  : (2, 2)),
             ((1, C)  (2, NC) : (4, 0)),
             ((1, NC) (2, C)  : (0, 4)),
             ((1, NC) (2, NC) : (3, 3)) .
endm
```

The general case To generalize our extensional game representation to a n-players game, we would only need to extend the arity of the utility constructor operator to n. Since the different actions of a player are captured as branches in the tree, a general m_n-action game may already be represented with our current implementation.

3.4 Programming subgame perfect equilibrium in extensive games with perfect information

Subgame perfect equilibrium (SPE) in an extensive game with perfect information occurs when the action taken by each player on each history is optimal given the strategies of the remaining players.

We have implemented the computation of SPE using the concept known as *backward induction*. From the leafs to the root, we identify the action that, at a given level in the tree, represents the best action to be taken.

Module `SPE-2` performs rewriting modulo associativity, commutativity and identity on terms of sort `BranchSet`. Assuming that a node may have a maximum of K children (two in the case of module `SPE-2`), that is to say, at a given step, there is a maximum of K different actions, our implementation compares the payoffs of K branches and chooses the best one, replacing the matched subtree by the best payoff. Moreover, this happens in *parallel* due to the congruence rule of rewriting logic that allows for parallel rewriting to take place. That is, in our Maude code, in one rewriting step, many branches are compared at once.

Note on efficiency Thanks to the congruence rule of rewriting logic, the complexity of our algorithm is $O(log(n)) * M$ where n is the number of nodes on the tree representing the game and M is the complexity of associativity-commutativity matching algorithm implemented in Maude.

```
mod SPE-2 is pr NAT . pr EGPI .
 sorts ActionList . subsort Action < ActionList .
 op mt-al : -> ActionList . op _,_ : ActionList ActionList -> ActionList [assoc id: mt-al] .
 op spe : BranchSet -> ActionList .
 var TL : TurnActionList . vars A1 A2 : Action . vars N1 N2 N3 N4 : Nat . var BS : BranchSet .
 eq spe(mt-tal : (N1,N2)) = mt-al .

 rl [spe1] : spe(TL (1, A1) : (N1,N2) , TL (1, A2) : (N3,N4) , BS) =>
    if N1 >= N3 then (spe((TL : (N1,N2)), BS), A1) else (spe((TL : (N3, N4)), BS), A2) fi .

 rl [spe2] : spe(TL (2, A1) : (N1,N2) , TL (2, A2) : (N3,N4) , BS) =>
    if N2 >= N4 then (spe((TL : (N1,N2)), BS), A1) else (spe((TL : (N3, N4)), BS), A2) fi .
endm
```

The SPE is calculated by the operator `spe`. Its behavior is given by rules `spe1` and `spe2`, for players 1 and 2, respectively. Let us take a look at rule `spe1`. The rule compares two branches with the *same prefix* `TL`, of sort `TurnActionList`, and with the same turn in the last action. The rule chooses the branch which has a bigger payoff for the current player and calls the operator `spe` recursively.

```
 rl [spe1] : spe(TL (1, A1) : (N1,N2) , TL (1, A2) : (N3,N4) , BS) =>
    if N1 >= N3 then (spe((TL : (N1,N2)), BS), A1) else (spe((TL : (N3, N4)), BS), A2) fi .
```

The following Maude session shows the calculation of SPE for the Dilema and the Centipede games. The results of the session should be interpreted[4] as: the actions $\langle C \rangle$ and $\langle C,C \rangle$ should be followed by the first and second players in the Dilema game and the strategy $\langle S,S,S \rangle$ should be follows by both players in the Centipede game.

```
==========================================
search in SPE-DILEMA : spe(dilema) =>! AL:ActionList .
Solution 1 (state 4)
```

[4] A proper organization of the actions for each player, as an output of operator `spe`, must be implemented.

```
states: 5  rewrites: 33 in 0ms cpu (0ms real) (100303 rewrites/second)
AL:ActionList --> C,C,C
No more solutions.
states: 5  rewrites: 33 in 0ms cpu (0ms real) (87533 rewrites/second)
========================================
search in SPE-CENTIPEDE : spe(centipede) =>! AL:ActionList .
Solution 1 (state 6)
states: 7  rewrites: 39 in 0ms cpu (0ms real) (98236 rewrites/second)
AL:ActionList --> S,S,S,S,S,S
No more solutions.
states: 7  rewrites: 39 in 0ms cpu (0ms real) (85152 rewrites/second)
```

The general case The extension of the computation of SPE to a n-player game is twofold: first, the rules would have to be adapted to cope with n-ary utility function and, second, there would be as many rules as there are players.

4 Final Remarks

It is important to note that the extensive form that we deal with is particular to perfect information games. Future research involves to extend this approach to handle generic games, which include a theory transformation to generate SPE-K modules. Also, imperfect information games, as well as other solution concepts should be treated.

We refer to [4] in the context of relating Game Theory and Rewriting. The authors apply Conversion/Preference (C/P) games and abstract Nash Equilibria to models of Gene Regulation. C/P games subsume Strategic Games and abstract Nash Equilibria is a form of Change-of-Mind equlibria that coincides with Nash Equilibrium in concrete cases. They capture Change-of-Mind equilibria as a set of normal forms in the rewriting system corresponding to a C/P game. When compared to our approach, it does not provide any way to deal with Extensive Games, unless the game is transformed into its strategic form. This would however lose analysis capabilities since SPE could not be used. Also, their approach is not logic-based as ours. They can not take advantage of congruence or rewriting modulo axioms as we do. It remains to be seen if our approach could be used to implement their proposal.

Our approach differs from a Modal Logic approach, since our goal in this paper was to use the efficiency of Maude rewriting in order to perform game analysis using search. Maude's LTL model checker may be used in the context of Game based Model Checking as in [15, 10]. This is, however, left as future work. It is worth mentioning that our approach can also play the game in either form, strategic or extensive. This is not the case regarding Modal Logic approaches.

References

[1] T. Ågotnes, M. Wooldridge, and W. van der Hoek. On the logic of coalitional games. In P. Stone and G. Weiss, editors, *AAMAS'06: Proceedings of the Fifth International Conference on Autonomous Agents and Multiagent Systems*, pages 153–160, Hakodate, Japan, May 2006. ACM Press.

[2] R. Aumann and S. Hart, editors. *Handbook of Game Theory with Economic Applications*, volume 1. Elsevier North-Holland, 3rd. edition, 2002.

[3] A. Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56:151–166, 1992.

[4] C. Chettaoui, F. Delaplace, P. Lescanne, M. Vestergaard, and R. Vestergaard. Rewriting game theory as a foundation for state-based models of gene regulation. In C. Priami, editor, *CMSB*, volume 4210 of *LNCS*, pages 257–270. Springer-Verlag, 2006.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International. `http://maude.cs.uiuc.edu/maude1/manual/maude-manual-html/maude-manual_62.html`.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. M.-O. J. Meseguer, and C. Talcott. *Maude Manual version 2.4*, February 2009. `http://maude.cs.uiuc.edu/maude2-manual/html/maude-manualch18.html#x96-25500018.4`.

[8] C. Daskalakis, P. W. Goldberg, and C. H. Papadimitriou. The complexity of computing a Nash equilibrium. In *STOC'06*, pages 71–78, New York, NY, USA, 2006. ACM Press.

[9] S. Eker. Associative-commutative rewriting on large terms. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 14–29. Springer-Verlag, June 2003.

[10] M. Kacprzak and W. Penczek. Unbounded model checking for alternating-time temporal logic. In *AAMAS'04*, pages 646–653. IEEE Computer Society, 2004.

[11] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[12] M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.

[13] R. Parikh. The logic of games and its applications. In *Selected papers of the International Conference on Foundations of computation theory - Topics in the theory of computation*, pages 111–139, New York, NY, USA, 1985. Elsevier North-Holland, Inc.

[14] S. van Otterloo, W. van der Hoek, and M. Wooldridge. Preferences in game logics. In *AAMAS'04*, pages 152–159, Washington, DC, USA, 2004. IEEE Computer Society.

[15] D. R. Vasconcelos, E. H. Haeusler, and M. F. Benevides. Reasoning about games via temporal logic. In *7th Augustus de Morgan Workshop*, London, 2005.

[16] D. R. Vasconcelos, E. H. Haeusler, and M. F. Benevides. Defining agents via strategies: Towards a view of MAS as games. In *WRAC 2005: Workshop on Radical Agent Concepts*, volume 3825 of *Lecture Notes in Artificial Intelligence*, pages 299–311. Springer Verlag, 2006.

The Semantics of Graph Programs

Detlef Plump
Department of Computer Science
The University of York, UK

Sandra Steinert
Department of Computer Science
The University of York, UK

GP (for Graph Programs) is a rule-based, nondeterministic programming language for solving graph problems at a high level of abstraction, freeing programmers from handling low-level data structures. The core of GP consists of four constructs: single-step application of a set of conditional graph-transformation rules, sequential composition, branching and iteration. We present a formal semantics for GP in the style of structural operational semantics. A special feature of our semantics is how it uses the notion of *finitely failing* programs to define GP's powerful branching and iteration commands.

1 Introduction

This paper defines the semantics of GP, an experimental nondeterministic programming language for high-level problem solving in the domain of graphs. The language is based on conditional rule schemata for graph transformation (introduced in [10]) and thereby frees programmers from handling low-level data structures for graphs. The prototype implementation of GP compiles graph programs into bytecode for the York abstract machine, and comes with a graphical editor for programs and graphs [7].

GP has a simple syntax as its core contains only four commands: single-step application of a set of rule schemata, sequential composition, branching and as-long-as-possible iteration. Despite its simplicity, GP is computationally complete in that every computable function on graphs can be programmed [4]. A major goal of the GP project is to obtain a practical graph-transformation language that comes with a concise formal semantics, to facilitate program verification and other formal reasoning on programs. Also, a formal semantics provides implementors with a rigorous definition of the language that does not depend on a compiler or machine.

To the best of our knowledge, PROGRES [12] has been the only graph-transformation language with a complete formal semantics so far. The semantics given by Schürr in his dissertation [11], however, reflects the complexity of PROGRES and is in our opinion too complicated to be used for formal reasoning.

For GP, we adopt Plotkin's method of structural operational semantics [9] to define the meaning of programs. This approach is well established for imperative programming languages [8] but is novel in the field of graph transformation. In brief, the method consists in devising inference rules which inductively define the effect of commands on program states. Whereas a classic state consists of the values of all program variables at a certain point in time, the analogue for graph transformation is the graph on which the rules of a program operate.

As GP is nondeterministic, our semantics assigns to a program P and an input graph G *all* graphs that can result from executing P on G. A special feature of the semantics is the use of failing computations to define powerful branching and iteration constructs. (Failure occurs when a set of rule schemata to be executed is not applicable to the current graph.) While the conditions of branching commands in traditional programming languages are boolean expressions, GP uses arbitrary programs as conditions. The evaluation of a condition C succeeds if there *exists* an execution of C on the current graph that

produces a graph. On the other hand, the evaluation of C is unsuccessful if all executions of C on the current graph result in failure. In this case C *finitely fails* on the current graph.

In logic programming, finite failure (of SLD resolution) is used to define negation [2]. In the case of GP, it allows to "hide" destructive executions of the condition C of a statement `if` C `then` P `else` Q. This is because after evaluating C, the resulting graph is discarded and either P or Q is executed on the graph with which the branching statement was entered. Finite failure also allows to elegantly lift the application of as-long-as-possible iteration from sets of rule schemata (as in [10]) to arbitrary programs: the body of a loop can no longer be applied if it finitely fails on the current graph.

The rest of this paper is structured as follows. The next section briefly reviews the graph-transformation formalism underlying GP, the so-called double-pushout approach with relabelling, and then introduces conditional rule schemata as the building blocks of GP programs. In Section 3, we discuss an example program for graph colouring and define the abstract syntax of graph programs. Section 4 presents our formal semantics of GP in the style of structural operational semantics. In Section 5, we conclude and mention some topics for future work.

2 Graph Transformation with Conditional Rule Schemata

We briefly review the model of graph transformation underlying GP, the double-pushout approach with relabelling [5], and then introduce conditional rule schemata as the building blocks of GP programs.

2.1 Graphs and Rules

GP programs operate on graphs labelled with sequences of integers and strings. (The reason for using sequences will become clear in Section 3.) To formalise this, let $\mathbb{Z}$ be the set of integers and Char be a finite set of characters—we may think of Char as the characters that can be typed on a keyboard. We fix the label alphabet $\mathscr{L} = (\mathbb{Z} \cup \text{Char}^*)^+$ consisting of all nonempty sequences made up from integers and character strings.

A *partially labelled graph* over $\mathscr{L}$ (or *graph* for short) is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$, where V_G and E_G are finite sets of *nodes* (or *vertices*) and *edges*, $s_G, t_G \colon E_G \to V_G$ are the *source* and *target* functions for edges, $l_G \colon V_G \to \mathscr{L}$ is the partial node labelling function[1] and $m_G \colon E_G \to \mathscr{L}$ is the (total) edge labelling function. Graph G is *totally labelled* if l_G is a total function. The set of all totally labelled graphs over $\mathscr{L}$ is denoted by $\mathscr{G}$.

A *graph morphism* $g \colon G \to H$ between graphs G and H consists of two functions $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ that preserve sources, targets and labels (that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $m_H \circ g_E = m_G$, and $l_H(g(v)) = l_G(v)$ for all v such that $l_G(v) \neq \bot$). Morphism g is an *inclusion* if $g(x) = x$ for all nodes and edges x. It is *injective* if g_V and g_E are injective.

A *rule* $r = (L \leftarrow K \rightarrow R)$ consists of two inclusions $K \to L$ and $K \to R$ where L and R are totally labelled graphs. Graph K is the *interface* of r. Intuitively, an application of r to a graph will remove the items in $L - K$, preserve K, and add the items in $R - K$. Given a graph G in $\mathscr{G}$, an injective graph morphism $g \colon L \to G$ is a *match* for r if it satisfies the *dangling condition*: no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$. In this case G *directly derives* the graph H in $\mathscr{G}$ that is constructed from G as follows:[2]

1. Remove all nodes and edges in $g(L) - g(K)$.

[1] We write $l_G(v) = \bot$ if $l_G(v)$ is undefined.

[2] See [5] for an equivalent definition by graph pushouts.

2. Add disjointly all nodes and edges from $R-K$, keeping their labels. For $e \in E_R - E_K$, $s_H(e)$ is $s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $g_V(s_R(e))$. Targets are defined analogously.
3. For each node v in K with $l_K(v) = \bot$, $l_H(g_V(v))$ becomes $l_R(v)$.

We write $G \Rightarrow_{r,g} H$ (or just $G \Rightarrow_r H$) if G directly derives H as above.

To define conditional rules, we equip rules with predicates that restrict sets of matches. A *conditional rule* $q = (r, P)$ consists of a rule r and a predicate P on graph morphisms. Given totally labelled graphs G, H and a match $g\colon L \to G$ for q, we write $G \Rightarrow_{q,g} H$ (or just $G \Rightarrow_q H$) if $P(g)$ holds and $G \Rightarrow_{r,g} H$. For a set of conditional rules $\mathscr{R}$, we write $G \Rightarrow_{\mathscr{R}} H$ if there is some q in $\mathscr{R}$ such that $G \Rightarrow_q H$.

2.2 Conditional Rule Schemata

A GP program is essentially a list of declarations of conditional rule schemata together with a command sequence for controlling the application of the schemata. Rule schemata generalise rules in that labels can contain expressions over parameters of type integer or string. In this subsection, we give an abstract syntax for the textual components of conditional rule schemata and interpret them as sets of conditional rules.

Figure 1 shows an example for the declaration of a conditional rule schema. It consists of the identifier `bridge` followed by the declaration of formal parameters, the left and right graphs of the schema which are labelled with expressions over the parameters, the node identifiers `1`, `2`, `3` determining the interface of the schema, and the keyword `where` followed by the condition.

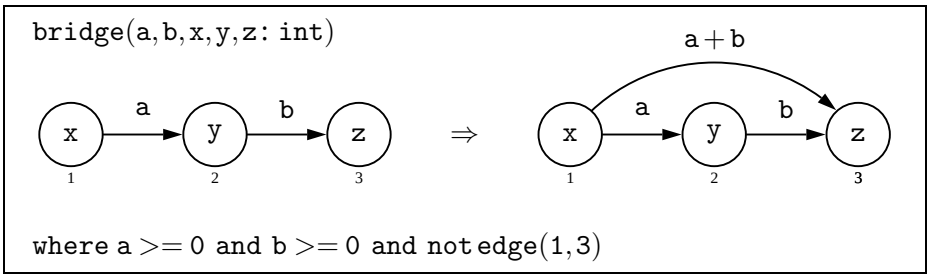


Figure 1: A conditional rule schema

In the GP programming system [7], rule schemata are constructed with a graphical editor. Figure 2 gives a grammar in Extended Backus-Naur Form for node and edge labels in the left and right graph of a rule schema (categories LeftLabel and RightLabel).[3] Labels can be sequences of expressions separated by underscores, as will be demonstrated by Example 1 in Section 3. We require that labels in the left graph must be simple expressions because their values at execution time are determined by graph matching. All variable identifiers in the right graph must also occur in the left graph. Every expression in category Exp has type `int` or `string`, where arithmetical operators expect arguments of type `int` and the type of variable identifiers is determined by their declarations.

The condition of a rule schema is a boolean expression built from expressions of category Exp and the special predicate `edge` (we omit the syntax). Again, all variable identifiers occurring in the condition must also occur in the left graph of the schema. The predicate `edge` demands the (non-)existence of an edge between two nodes in the graph to which the rule schema is applied. For example, the expression

[3]The grammar is ambiguous, we use parentheses to disambiguate expressions where necessary.

LeftLabel	::=	SimpleExp ['_' LeftLabel]
RightLabel	::=	Exp ['_' RightLabel]
SimpleExp	::=	['-'] Num | String | VarId
Exp	::=	SimpleExp | Exp ArithOp Exp
ArithOp	::=	'+' | '−' | '∗' | '/'
Num	::=	Digit {Digit}
String	::=	'"' {Char} '"'

Figure 2: Syntax of node and edge labels

`not edge`$(1,3)$ in the condition of Figure 1 forbids an edge from node 1 to node 3 when the left graph is matched.

We interpret a conditional rule schema as the (possibly infinite) set of conditional rules that is obtained by instantiating variables with any values and evaluating expressions. To define this, consider a declaration D of a conditional rule-schema. Let L and R be the left and right graphs of D, and c the condition. We write $\mathrm{Var}(D)$ for the set of variable identifiers occurring in D. Given x in $\mathrm{Var}(D)$, $\mathrm{type}(x)$ denotes the type associated with x. An *assignment* is a mapping $\alpha\colon \mathrm{Var}(D) \to (\mathbb{Z} \cup \mathrm{Char}^*)$ such that for each x in $\mathrm{Var}(D)$, $\mathrm{type}(x) = \texttt{int}$ implies $\alpha(x) \in \mathbb{Z}$, and $\mathrm{type}(x) = \texttt{string}$ implies $\alpha(x) \in \mathrm{Char}^*$.

Given a label l of category RightLabel occuring in D and an assignment α, the value $l^\alpha \in \mathscr{L}$ is inductively defined. If l is a numeral or a sequence of characters, then l^α is the integer or character string represented by l (which is independent of α). If l is a variable identifier, then $l^\alpha = \alpha(l)$. Otherwise, l^α is obtained from the values of l's components. If l has the form $e_1 \oplus e_2$ with $\oplus$ in ArithOp and e_1, e_2 in Exp, then $l^\alpha = e_1^\alpha \oplus_{\mathbb{Z}} e_2^\alpha$ where $\oplus_{\mathbb{Z}}$ is the integer operation represented by $\oplus$.[4] If l has the form e_m with e in Exp and m in RightLabel, then $l^\alpha = e^\alpha m^\alpha$. Note that our definition of l^α covers all labels in D since LeftLabel is a subcategory of RightLabel.

The value of the condition c in D not only depends on an assignment but also on a graph morphism. For, if c contains the predicate `edge`, then we need to consider the structure of the graph to which we want to apply the rule schema. Consider an assignment α and let L^α be obtained from L by replacing each label l with l^α. Let $g\colon L^\alpha \to G$ be a graph morphism with $G \in \mathscr{G}$. Then for each Boolean subexpression b of c, the value $b^{\alpha,g}$ in $\mathbb{B} = \{\texttt{tt}, \texttt{ff}\}$ is inductively defined. For example, if b has the form $e_1 \bowtie e_2$ with $\bowtie$ in RelOp and e_1, e_2 in Exp, then $b^{\alpha,g} = \texttt{tt}$ if and only if $e_1^\alpha \bowtie_{\mathbb{Z}} e_2^\alpha$ where $\bowtie_{\mathbb{Z}}$ is the relation on integers represented by $\bowtie$. A special case is given if b has the form $\texttt{edge}(v,w)$ where v, w are identifiers of interface nodes in D. We then have

$$b^{\alpha,g} = \begin{cases} \texttt{tt} & \text{if there is an edge from } g(v) \text{ to } g(w), \\ \texttt{ff} & \text{otherwise.} \end{cases}$$

Let now r be the rule-schema identifier associated with declaration D. For every assignment α, let $r^\alpha = (L^\alpha \leftarrow K \rightarrow R^\alpha, P^\alpha)$ be the conditional rule given as follows:

- L^α and R^α are obtained from L and R by replacing each label l with l^α.
- K is the discrete subgraph of L and R determined by the node identifiers for the interface, where all nodes are unlabelled.

[4] For simplicity, we consider division by zero as an implementation-level issue.

- P^α is defined by: $P^\alpha(g)$ if and only if g is a graph morphism $L^\alpha \to G$ such that $G \in \mathscr{G}$ and $c^{\alpha,g} = \mathtt{tt}$.

Now the *interpretation* of r is the rule set $\mathrm{I}(r) = \{r^\alpha \mid \alpha \text{ is an assignment}\}$. For notational convenience, we sometimes denote the relation $\Rightarrow_{\mathrm{I}(r)}$ by $\Rightarrow_r$.

For example, the upper rows of Figure 3 show the rule schema `bridge` of Figure 1 (without condition) and its instance $\mathtt{bridge}^\alpha$, where $\alpha(\mathtt{x}) = 0$, $\alpha(\mathtt{y}) = \alpha(\mathtt{z}) = 1$, $\alpha(\mathtt{a}) = 3$ and $\alpha(\mathtt{b}) = 2$. The condition c of `bridge` evaluates to the predicate P^α which is true for a match g of the left-hand graph if and only if there is no edge from $g(1)$ to $g(3)$. (Note that the subexpressions $\mathtt{a} >= 0$ and $\mathtt{b} >= 0$ evaluate to `tt` and hence can be ignored.) The lower rows of Figure 3 show an application of $\mathtt{bridge}^\alpha$ by a graph morphism satisfying P^α.

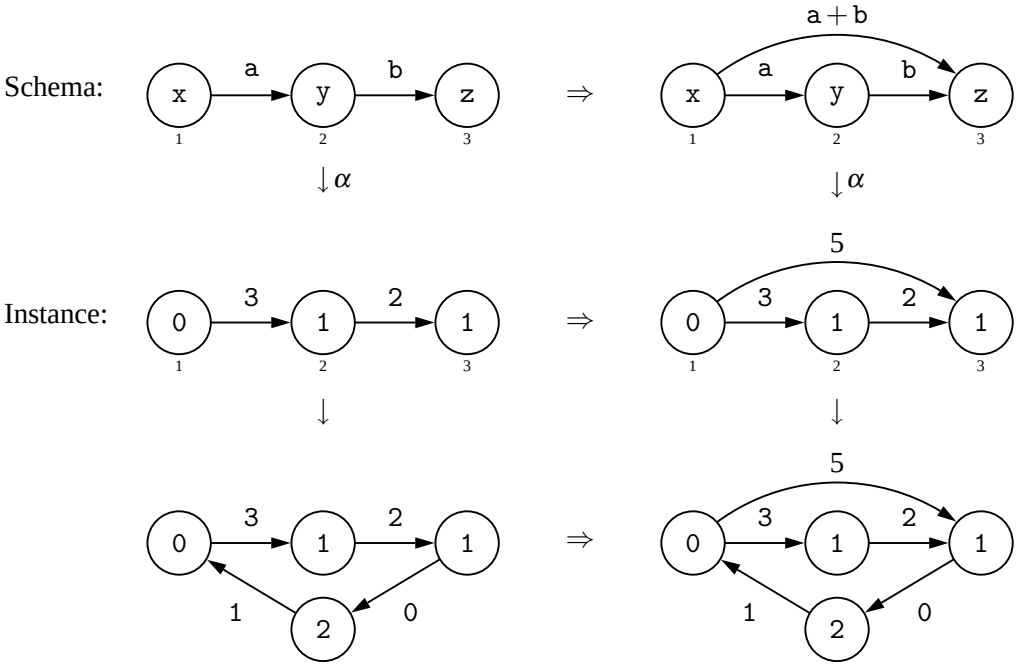


Figure 3: Application of a rule schema using instantiation

3 Graph Programs

We start by discussing an example program for graph colouring.

Example 1 (Computing a 2-colouring). A *colouring* for a graph is an assignment of colours (integers) to nodes such that the source and target of each edge have different colours. A graph is *2-colourable* (or *bipartite*) if it possesses a colouring with at most two colours. The program `2-colouring` in Figure 4 generates a 2-colouring for nonempty, connected input graphs without loops if such a colouring exists—otherwise the input graph is returned. The program consists of five rule-schema declarations, the *macro* `colour` representing the rule-schema set $\{\mathtt{colour1}, \mathtt{colour2}\}$, and the main command sequence following the key word `main`.

Given an integer-labelled input graph, the program first uses the rule schema `choose` to pick any node and replace its label x with x_0. The underscore operator allows to add a *tag* to a label, used

here to add colours to labels. In general, a tagged label consists of a sequence of expressions joined by underscores. After the first node has been coloured, the command `colour!` applies the rule schemata

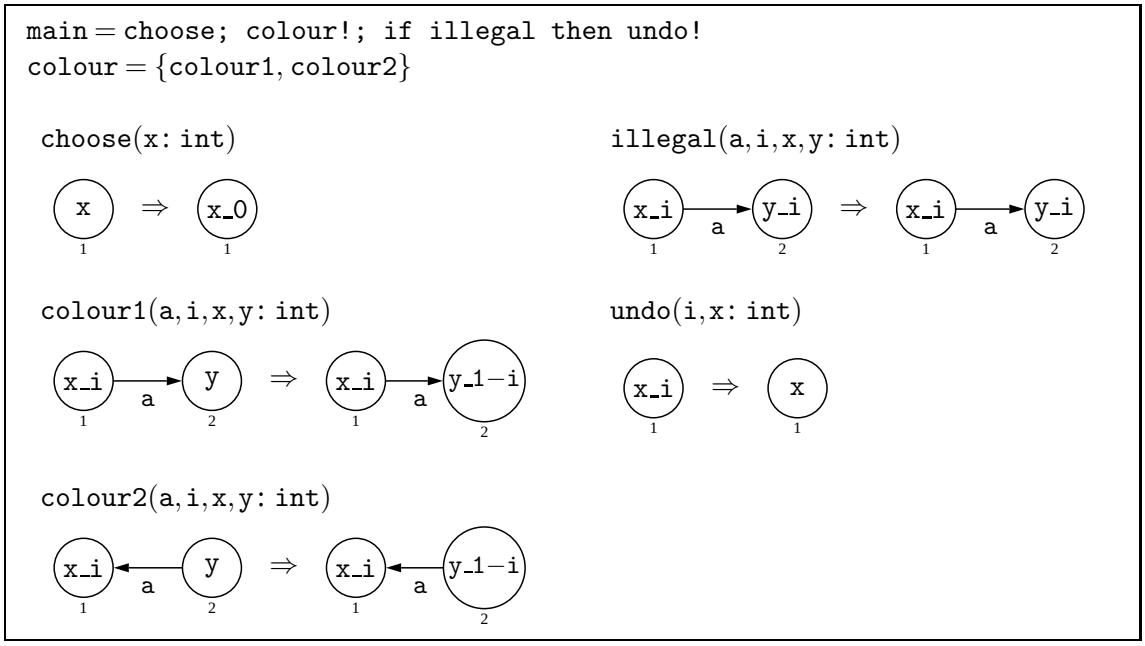


Figure 4: The program `2-colouring`

`colour1` and `colour2` nondeterministically as long as possible to colour all remaining nodes. In each iteration of the loop, an uncoloured node adjacent to an already coloured node v gets the colour in $\{0,1\}$ that is complementary to v's colour. If the input graph is connected, the graph resulting from `colour!` is correctly coloured if and only if the rule schema `illegal` is not applicable. The latter is checked by the if-statement. If `illegal` is applicable, then the input must contain an undirected cycle of odd length and hence is not 2-colourable (see for example [6]). In this case the loop `undo!` removes all tags to return the input graph unmodified. Note that the number of rule-schema applications performed by `2-colouring` is linear in the number of input nodes.

We can extend `2-colouring`'s applicability to graphs that are possibly empty or disconnected by inserting a nested loop:

```
main = (choose; colour!)!; if illegal then undo!.
```

Now if the input graph is empty, `choose` fails which causes the outer loop to terminate and return the current (empty) graph. On the other hand, if the input consists of several connected components, the body of the outer loop is repeatedly called to colour each component.

Figure 5 shows the abstract syntax of GP programs.[5] A program consists of a number of declarations of conditional rule schemata and macros, and exactly one declaration of a main command sequence. The rule-schema identifiers (category RuleId) occurring in a call of category RuleSetCall refer to declarations of conditional rule schemata in category RuleDecl (see Section 2.2). Semantically, each rule-schema identifier r stands for the set $\mathrm{I}(r)$ of conditional rules induced by that identifier. A call of the form $\{r_1,\ldots,r_n\}$ stands for the union $\bigcup_{i=1}^{n}\mathrm{I}(r_i)$.

[5] Where necessary we use parentheses to disambiguate programs.

Prog	::=	Decl {Decl}
Decl	::=	RuleDecl | MacroDecl | MainDecl
MacroDecl	::=	MacroId '=' ComSeq
MainDecl	::=	`main` '=' ComSeq
ComSeq	::=	Com {';' Com}
Com	::=	RuleSetCall | MacroCall
		| `if` ComSeq `then` ComSeq [`else` ComSeq]
		| ComSeq '!'
		| `skip` | `fail`
RuleSetCall	::=	RuleId | '{' [RuleId {',' RuleId}] '}'
MacroCall	::=	MacroId

Figure 5: Abstract syntax of GP

Macros are a simple means to structure programs and thereby to make them more readable. Every program can be transformed into an equivalent macro-free program by replacing macro calls with their associated command sequences (recursive macros are not allowed). In the next section we use the terms "program" and "command sequence" synonymously, assuming that all macro calls have been replaced.

The commands `skip` and `fail` can be expressed through the other commands (see next section), hence the core of GP includes only the call of a set of conditional rule schemata (RuleSetCall), sequential composition (';'), the if-then-else statement and as-long-as-possible iteration ('!').

4 Semantics of Graph Programs

We present a formal semantics of GP in the style of Plotkin's structural operational semantics [9]. As usual for this approach, inference rules inductively define a small-step transition relation $\rightarrow$ on *configurations*. In our setting, a configuration is either a command sequence together with a graph, just a graph or the special element fail:

$$\rightarrow \subseteq (\text{ComSeq} \times \mathscr{G}) \times ((\text{ComSeq} \times \mathscr{G}) \cup \mathscr{G} \cup \{\text{fail}\}).$$

Configurations in $\text{ComSeq} \times \mathscr{G}$ represent unfinished computations, given by a rest program and a state in the form of a graph, while graphs in $\mathscr{G}$ are proper results of computations. In addition, the element fail represents a failure state. A configuration γ is *terminal* if there is no configuration δ such that $\gamma \rightarrow \delta$.

Each inference rule in Figure 6 consists of a premise and a conclusion separated by a horizontal bar. Both parts contain meta-variables for command sequences and graphs, where R stands for a call in category RuleSetCall, C, P, P', Q stand for command sequences in category ComSeq and G, H stand for graphs in $\mathscr{G}$. Given a rule-set call R, let $\mathrm{I}(R) = \bigcup\{\mathrm{I}(r) \mid r \text{ is a rule-schema identifier in } \mathscr{R}\}$ (see Section 2.2 for the definition of $\mathrm{I}(r)$). The *domain* of $\Rightarrow_{\mathrm{I}(R)}$, denoted by $\mathrm{Dom}(\Rightarrow_{\mathrm{I}(R)})$, is the set of all graphs G in $\mathscr{G}$ such that $G \Rightarrow_{\mathrm{I}(R)} H$ for some graph H. Meta-variables are considered to be universally quantified. For example, the rule $[\mathrm{Call}_1]$ should be read as: "For all R in RuleSetCall and all G, H in $\mathscr{G}$, $G \Rightarrow_{\mathrm{I}(R)} H$ implies $\langle R, G\rangle \rightarrow H$."

Figure 6 shows the inference rules for the core constructs of GP. We write $\rightarrow^+$ and $\rightarrow^*$ for the transitive and reflexive-transitive closures of $\rightarrow$. A command sequence C *finitely fails* on a graph $G \in$

$\mathscr{G}$ if (1) there does not exist an infinite sequence $\langle C, G\rangle \to \langle C_1, G_1\rangle \to \ldots$ and (2) for each terminal configuration γ such that $\langle C, G\rangle \to^* \gamma$, $\gamma = \text{fail}$. In other words, C finitely fails on G if all computations starting from (C, G) eventually end in the configuration fail.

$$[\text{Call}_1]\ \frac{G \Rightarrow_{\text{I}(R)} H}{\langle R, G\rangle \to H} \qquad [\text{Call}_2]\ \frac{G \notin \text{Dom}(\Rightarrow_{\text{I}(R)})}{\langle R, G\rangle \to \text{fail}}$$

$$[\text{Seq}_1]\ \frac{\langle P, G\rangle \to \langle P', H\rangle}{\langle P;Q, G\rangle \to \langle P';Q, H\rangle} \qquad [\text{Seq}_2]\ \frac{\langle P, G\rangle \to H}{\langle P;Q, G\rangle \to \langle Q, H\rangle}$$

$$[\text{Seq}_3]\ \frac{\langle P, G\rangle \to \text{fail}}{\langle P;Q, G\rangle \to \text{fail}}$$

$$[\text{If}_1]\ \frac{\langle C, G\rangle \to^+ H}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, G\rangle} \qquad [\text{If}_2]\ \frac{C \text{ finitely fails on } G}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle}$$

$$[\text{Alap}_1]\ \frac{\langle P, G\rangle \to^+ H}{\langle P!, G\rangle \to \langle P!, H\rangle} \qquad [\text{Alap}_2]\ \frac{P \text{ finitely fails on } G}{\langle P!, G\rangle \to G}$$

Figure 6: Inference rules for core commands

The concept of finite failure stems from logic programming where it is used to define *negation as failure* [2]. In the case of GP, we use it to define powerful branching and iteration constructs. In particular, our definition of the if-then-else command allows to "hide" destructive tests.

Example 2 (Recognizing series-parallel graphs). A graph is a *series-parallel graph* if it reduces to a graph consisting of two nodes and an edge between them by the following two operations [1, 3]: (1) Replace a pair of parallel edges by an edge from their source to their target. (2) Given a node v with exactly one incoming edge e_1 and exactly one outgoing edge e_2 such that the source of e_1 and the target of e_2 are distinct, replace e_1, e_2 and v by an edge from the source of e_1 to the target of e_2.

Suppose that we want to check whether a connected, integer-labelled graph G is a series-parallel graph and, depending on the result, execute either a program P or a program Q on G. We can do this with the program

$$\texttt{main} = \texttt{if}\ \{\texttt{par}, \texttt{seq}\}!; \texttt{base}\ \texttt{then}\ P\ \texttt{else}\ Q.$$

The subprogram $\{\texttt{par}, \texttt{seq}\}!$ applies as long as possible the operations (1) and (2) to the input graph G, then the rule schema `base` checks if the resulting graph consists of two nodes connected by an edge. Graph G is a series-parallel graph if and only if `base` is applicable to the reduced graph. (Note that $\{\texttt{par}, \texttt{seq}\}!$ preserves connectedness and that, by the dangling condition, `base` is applicable only if the images of its left-hand nodes have degree one.) It is important to note that by the inference rules $[\text{If}_1]$ and $[\text{If}_2]$, the main program executes P or Q *on the input graph* G whereas the graph resulting from the test is discarded. The rule schemata `par`, `seq` and `base` are shown in Figure 7.

The meaning of the remaining GP commands is defined in terms of the meaning of the core commands, see Figure 8. We refer to these commands as *derived* commands.

We can now summarise the meaning of GP programs by a semantic function $[\![\_]\!]$ which assigns to each program P the function $[\![P]\!]$ mapping an input graph G to the set of all possible results of running P on G. The result set may contain, besides proper results in the form of graphs, the special value $\perp$ which indicates a nonterminating or stuck computation. The *semantic function* $[\![\_]\!]\colon \text{ComSeq} \to (\mathscr{G} \to 2^{\mathscr{G}\cup\{\perp\}})$

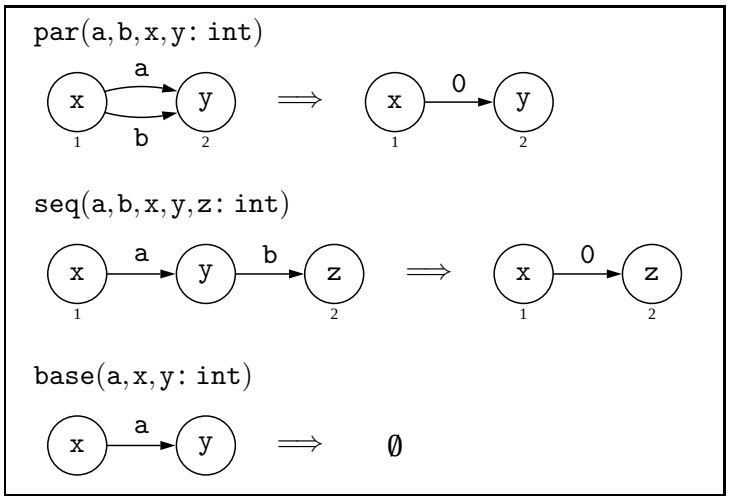


Figure 7: Rule schemata for recognizing series-parallel graphs

[Skip] $\langle \texttt{skip}, G\rangle \to \langle \texttt{r}_\emptyset, G\rangle$
where $\texttt{r}_\emptyset$ is an identifier for the rule schema $\emptyset \Rightarrow \emptyset$

[[Fail] $\langle \texttt{fail}, G\rangle \to \langle \{\}, G\rangle$

[If$_3$] $\langle \texttt{if } C \texttt{ then } P, G\rangle \to \langle \texttt{if } C \texttt{ then } P \texttt{ else skip}, G\rangle$

Figure 8: Inference rules for derived commands

is defined by[6]

$$[\![P]\!]G = \{H \in \mathscr{G} \mid \langle P, G\rangle \xrightarrow{+} H\} \cup \{\bot \mid P \text{ can diverge or get stuck from } G\}$$

where P *can diverge from* G if there is an infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \ldots$, and P *can get stuck from* G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$.

Note that $[\![P]\!]G = \emptyset$ if and only if P finitely fails on G. In Example 2, for instance, we have $[\![\{\texttt{par}, \texttt{seq}\}!; \texttt{base}]\!]G = \emptyset$ for every connected graph G containing a cycle. This is because the graph resulting from $\{\texttt{par}, \texttt{seq}\}!$ is still connected and cyclic, so the rule schema `base` is not applicable.

A program can get stuck only in two situations: either it contains a subprogram `if` C `then` P `else` Q where C both can diverge from some graph and cannot produce a proper result from that graph, or it contains a subprogram $B!$ where the loop's body B possesses the said property of C. The evaluation of these subprograms will get stuck because the inference rules for branching and iteration are not applicable.

5 Conclusion

GP is an experimental rule-based language for high-level problem solving in the domain of graphs, freeing programmers from handling low-level data structures. The hallmark of GP is syntactic and

[6] We write $[\![P]\!]G$ for the application of $[\![P]\!]$ to a graph G.

semantic simplicity. Conditional rule schemata for graph transformation allow to express application conditions and computations on labels, in addition to structural changes.

The operational semantics describes the effect of GP's control constructs in a natural way and captures the nondeterminism of the language. In particular, powerful branching and iteration commands have been defined using the concept of finite failure. Destructive tests on the current graph can be hidden in the condition of the branching command, and nested loops can be coded since arbitrary subprograms can be iterated as long as possible.

Future extensions of GP may include recursive procedures for writing complex algorithms (see [13]), and a type concept for restricting the shape of graphs. Our goal is to support formal reasoning on graph programs by developing static analyses for properties such as termination and confluence (uniqueness of results), and a calculus and tool support for program verification.

References

[1] Jørgen Bang-Jensen and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer-Verlag, 2000.

[2] Keith L. Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[3] R. J. Duffin. Topology of series-parallel networks. *Journal of Mathematical Analysis and Applications*, 10:303–318, 1965.

[4] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.

[5] Annegret Habel and Detlef Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer-Verlag, 2002.

[6] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[7] Greg Manning and Detlef Plump. The GP programming system. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, volume 10 of *Electronic Communications of the EASST*, 2008.

[8] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-Verlag, 2007.

[9] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[10] Detlef Plump and Sandra Steinert. Towards graph programs for graph algorithms. In *Proc. International Conference on Graph Transformation (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*, pages 128–143. Springer-Verlag, 2004.

[11] Andy Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitäts-Verlag, 1991. In German.

[12] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 13, pages 487–550. World Scientific, 1999.

[13] Sandra Steinert. *The Graph Programming Language GP*. PhD thesis, The University of York, 2007.

An Improved Algorithm for Generating Database Transactions from Relational Algebra Specifications

Daniel J. Dougherty
Worcester Polytechnic Institute

Abstract

Alloy is a lightweight modeling formalism based on relational algebra. In prior work with Fisler, Giannakopoulos, Krishnamurthi, and Yoo, we have presented a tool, Alchemy, that compiles Alloy specifications into implementations that execute against persistent databases. The foundation of Alchemy is an algorithm for rewriting relational algebra formulas into code for database transactions. In this paper we report on recent progress in improving the robustness and efficiency of this transformation.

1 Introduction

Alloy [Jac06] is a popular modeling language that implements the lightweight formal methods philosophy [JW96]. Its expressive power is that of first-order logic extended with transitive closure, and its syntax, based on relational algebra, is strongly influenced by object modeling notations. The language is accompanied by the Alloy Analyzer: the analyzer builds models (or "instances") for a specification using SAT-solving techniques. Users can employ a graphical browser to explore instances and counter-examples to claims.

Having written an Alloy specification, the user must then write the corresponding code by hand; consequently there are no formal guarantees that the resulting code has any relationship to the specification. The *Alchemy* project addresses this issue. Alchemy is a tool under active development [KDFY08, GDFK] at Worcester Polytechnic Institute and Brown University, by Kathi Fisler, Shriram Krishnamurthi, and the author, with our students Theo Giannakopoulos and Daniel Yoo, that compiles Alloy specifications into libraries of database operations. This is not a straightforward enterprise since, in contrast to Z [Spi92] and B [Abr96], where a notion of state machine is built into the language, Alloy does not have a native machine model.

Alchemy opens up a new way of working with Alloy specifications: as declarative notations for imperative programs. In this way Alloy models support a novel kind of rule-based programming, in which underspecification is a central aspect of program design.

In this note we report on recent progress in improving the process of generating imperative code for declarative specifications in a language like Alloy. This paper is a companion to [GDFK], which developed a better semantic foundation for interpreting Alloy predicates as operations. With this better foundation we are able to generate code for a wider class of predicates than that treated in [KDFY08] and also prove a more robust correctness theorem relating the imperative code to the original specification.

2 Alloy and Alchemy

Notation. *For consistency with the presentation and analysis of the algorithms below, we use standard mathematical notation in specifications in some places where Alloy uses ASCII notation. In particular $\cup$ is "+" in Alloy, and $\cap$ is "&" there.*

Some of the material in this expository section is taken from [KDFY08].

```
sig Submission {}
sig Grade {}
sig Student {}

sig Course {
   roster : set Student,
   work : roster → Submission,
   gradebook : work → lone Grade }

pred Enroll (c, c' : Course, sNew : Student) {
     c'.roster = c.roster ∪ sNew and
     c'.work[sNew] = ∅ }

pred Drop (c, c' : Course, s: Student) {
   s not in c'.roster }

pred SubmitForPair (c, c' : Course, s1, s2 : Student,
                    bNew : Submission) {
   // pre-condition
   s1 in c.roster and s2 in c.roster and
   // update
   c'.work = c.work ∪ <s1, bNew> ∪ <s2, bNew> and
   // frame condition
   c'.gradebook = c.gradebook }

pred AssignGrade (c, c' : Course, s : Student,
                  b : Submission, g : Grade) {
   c'.gradebook in c.gradebook ∪ <s, b, g> and
   c'.roster = c.roster }

fact SameGradeForPair {
   all c : Course, s1, s2 : Student, b : Submission |
       b in (c.work[s1] & c.work[s2]) implies
           c.gradebook[s1][b] = c.gradebook[s2][b] }
```

Figure 1: Alloy specification of a gradebook.

2.1 An overview of Alloy

An excellent introduction to Alloy is Daniel Jackson's book [Jac06]. Here we start with an informal introduction to Alloy syntax and semantics via an example. The example is a homework submission and grading system, shown in Figure 1. In this system, students may submit work in pairs. The gradebook stores the grade for each student on each submission. Students may be added to or deleted from the system at any time, as they enroll in or drop the course.

The system's data model centers around a course, which has three fields: a roster (set of students), submitted work (relation from enrolled students to submissions), and a gradebook. Alloy uses **sig**natures to capture the sets and relations that comprise a data model. Each **sig** (*Submission*, etc.) defines a unary relation. The elements of these relations are called *atoms*; the type of each atom is its containing relation.

Fields of signatures define additional relations. The **sig** for *Course*, for example, declares *roster* to be a relation on *Course*×*Student*. Similarly, the relation *work* is of type *Course*×*Student*×*Submission*, but with the projection on *Course* and *Student* restricted to pairs in the *roster* relation. The **lone** annotation on *gradebook* allows at most

one grade per submission.

The **pred**icates (*Enroll*, etc.) capture the actions supported in the system. The predicates follow a standard Alloy idiom for stateful operations: each has parameters for the pre- and post-states of the operation (c and c', respectively), with the intended interpretation that latter reflects a change applied to the former. Alloy **fact**s (such as *SameGradeForPair*) capture invariants on the models. This particular fact states that students who submit joint work get the same grade.

An important aspect of Alloy is that *everything is a relation.* In particular sets are viewed as unary relations, and individual atoms are viewed as singleton unary relations. As a consequence the **in** operator does double-duty: it is interpreted formally as subset, but also stands in for the "element-of" relation, in the sense that if—intuitively—a is an atom that is an element of a set r, this is expressed in Alloy as a **in** r, since a is formally a (singleton) set.

The Alloy semantics defines a set of models for the signatures and facts. Operators over sets and relations have their usual semantics: $+$ (union), & (intersection), $\langle\ ,\ \rangle$ (tupling), and . (join). As noted above, **in** denotes subset and is alos used to encode membership. Square brackets provide a convenient syntactic sugar for certain joins: $e2[e1]$ is equivalent to $e1.e2$. The following relations constitute a model under the Alloy semantics.

$Student = \{Harry, Meg\}$
$Submission = \{hwk1\}$
$Grade = \{A, A-, B+, B\}$
$Course = \{c0, c1\}$
$roster = (\langle c0, Harry\rangle, \langle c1, Harry\rangle, \langle c1, Meg\rangle)$
$work = \{\langle c1, Harry, hwk1\rangle\}$
$gradebook = \{\langle c1, Harry, hwk1, A-\rangle\}$

A model of a predicate also associates each predicate parameter with an atom in the model such that the predicate body holds. The above set of relations models the *Enroll* predicate under bindings *c = c0*, *c' = c1* and *sNew = Meg*. A model may include tuples beyond those required to satisfy a predicate: the *Enroll* predicate does not constrain the *work* relation for pre-existing students, so the appearance of tuple $\langle c1, Harry, hwk1\rangle$ in the *work* relation is semantically acceptable.

The reader may want to check that the relations shown do not happen to model the predicate *SubmitForPair*, in the sense that no bindings for $c, c', s1, s2, andbNew$ make the body of *SubmitForPair* true. Under *c0* and *c' = c1*, for example, the requirement $c'.gradebook = c.gradebook$ fails because the gradebook starting from c' has one tuple while that starting from c has none. The requirement on *work* also fails. Similar inconsistencies contradict other possible bindings for c and c'.

2.2 An overview of Alchemy

We illustrate Alchemy in the context of the gradebook specification from Figure 1. Alchemy creates a database table for each relation (e.g., *Submission*, *roster*), a procedure for each predicate (e.g., *Enroll*), and a function for creating new elements of each atomic signature (e.g., *CreateSubmission*). A sample session using Alchemy might proceed as follows. We create a course with two students using the following command sequence:

cs311 = *CreateCourse*("cs311");
pete = *CreateStudent*("Pete");
caitlin = *CreateStudent*("Caitlin");
Enroll(*cs311*, *pete*);
Enroll(*cs311*, *caitlin*)

Note that the *Enroll* function takes only one course-argument, in contrast to the two in the original Alloy predicate, since the implementation maintains only a single set of tables over time (the second course parameter in the predicate corresponds to the resulting updated table). Executing the *Enroll* function adds the pairs ⟨"cs311", "Pete"⟩ and ⟨"cs311", "Caitlin"⟩ to the *roster* table. The second clause of the *Enroll* specification guarantees that the *work* table will not have entries for either student.

Next, we submit a new homework for "Pete" and "Caitlin":

hwk1 = *CreateSubmission*("hwk1");
SubmitForPair(*cs311*, *pete*, *caitlin*, *hwk1*)

The implementation of *SubmitForPair* is straightforward relative to the specification. It treats the first clause in the specification as a pre-condition by terminating the computation with an error if the clause is false in the database at the start of the function execution. Next, it adds the *work* tuples required in the second (update) clause. It ensures that the *gradebook* table is unchanged, as required by the third clause.

Assigning a grade illustrates the way that Alloy facts constrain Alchemy's updates:

gradeA = *CreateGrade*("A");
AssignGrade(*cs311*, *pete*, *hwk1*, *gradeA*)

AssignGrade inserts a tuple into the *gradebook* relation according to the first clause, and checks that the roster is unchanged according to the second. If execution were to stop here, however, the resulting tables would contradict the *SameGradeForPair* invariant (which requires "Caitlin" to receive the same grade on the joint assignment). Alchemy determines that adding the tuple ⟨"cs311","Caitlin","hwk1", "A"⟩ to *gradebook* will satisfy both the predicate body and the *SameGradeForPair* fact, and executes this command automatically. If there is no way to update the database to respect both the predicate and the fact, Alchemy will raise an exception. This could happen, for example, if the first clause in *AssignGrade* used =instead of **in** : in this case, adding the repairing tuple would violate the predicate body).

Maintaining invariants Alloy's use of *facts* to constrain possibly-underspecified predicates offers a powerful lightweight modeling tool. The facts in an Alloy specification are axioms in the sense that they hold in any instance for the specification. We may view the facts as integrity constraints: they capture the fundamental invariants to be maintained across all transactions. Alchemy will guarantee preservation of all facts as database invariants. This is akin to the notion of *repair* of database transactions.

2.3 Formalities

Alloy specifications Formally, the *Alloy specifications* we treat in this paper are tuples of *signatures*, *predicates,* and *facts.* In practice Alloy specifications may also include *assertions* to be checked by the analyzer, but they do not play a direct role in Alchemy's code generation so we omit them here.

- A signature specifies its type name and a set of fields. Each field has a name and a type specification $A_0 \rightarrow A_1 \rightarrow \ldots \rightarrow A_n$, where each A_i is the type name of some signature.
- A predicate has a header and a body. The header declares a set of variable names, each with an associated signature type name; the body is a formula in which the only free variables are defined in the header.
- A fact is a closed formula, having the force of an axiom: models of a specification are required to satisfy these facts. Alloy permits the user to specify certain constraints on the signatures and fields when they are declared, such as "relation *r* may have at most one tuple." These can be alternatively expressed as facts and, for simplicity of presentation, we assume this is always done.

The following language for expressions and formulas is essentially equivalent to the Kernel language of Alloy [Jac06] (modulo the lexical differences between standard mathematical notation used here and Alloy's ASCII).

expr ::= rel | var | none | expr binop expr | unop expr
binop ::= $\cup$ | $\cap$ | $-$ | . | $\langle,\rangle$
unop ::= $\sim$| $*$

formula ::= elemFormula | compFormula | quantFormula
elemFormula ::= expr **in** expr | expr = expr
compFormula ::= **not** formula | formula $\wedge$ formula | formula $\vee$ formula
quantFormula ::= $\forall$ *var:* expr { formula } | $\exists$ *var:* expr { formula }

State-based specifications The elements of an Alloy specification suggest natural implementation counterparts. The signatures lay out relations that translate directly into persistent database schemas. The facts—those properties that are meant to hold of all models constructed by Alloy—function as database integrity constraints. Finally, under a commonly idiom, certain predicates in an Alloy specification connote state changes. It is these state-based specifications that Alchemy (currently) treats.

The state-transition idiom is a commonly understood convention rather than a formal notion in Alloy. To precisely define the class of specifications that Alchemy treats, we first require some terminology. Fix a distinguished signature, which we will call State. An *immutable type* is one with no occurrences of the State signature.

The assumptions Alchemy makes about the specifications it treats are:

- specifications are state-based, and
- facts have at most one variable of type State and that this variable is unprimed and universally quantified.

An operational semantics The static semantics of Alloy is based on the class of relational algebras. To give an operational semantics for state-based Alloy specifications, one that takes seriously the reading of predicates as state-transformers, we pass to the class of transition systems whose nodes are relational algebras. We also assume that each state have a single atom of type State. When individual relation algebras are read as database instances, transitions between states can be viewed as database update sequences transforming one state to another. We adopt a constant-domain assumption concerning our transition systems. Space consideration prohibit us from presenting the motivation and justification for this (including the explanation why it is not as great a restriction as it may appear); details are in [GDFK].

Since predicates have parameters, the meaning of a predicate is relative to bindings from variables to values. It is technically convenient to assume that for a given specification we identify, for each type, a universe of possible values at this type. Then an *environment* η is a mapping from typed variables to values.

Definition 1 (**Operational semantics of predicates**)**.** Let p be a predicate with the property that p has among its parameters exactly two variables `s` and `s'` of type `State`, and let η be an environment. The meaning $[\![p]\!]\eta$ of p under η is the set of pairs $\langle I, I' \rangle$ of instances such that

- η maps the parameters of p into the set of atoms of I (which equals the set of atoms of I'), mapping the unprimed State parameter to the State-atom of I and the primed State parameter to the State-atom of I';
- (I, I') makes the body of p true under the environment η: occurrences of the State variable s are interpreted in I, while occurrences of the State variable s' are interpreted in I'.

The meaning of a predicate p is a *set* of transitions because p can be applied to different nodes, with different bindings of the parameters of course, but also—and more interestingly—because predicates typically under-specify actions: different implementations of a predicate can yield different outcomes I' on the same input I. Any of these should be considered acceptable as long as the relation between pre- and post-states is described by the predicate.

3 Main Result

We observed that a predicate p determines a family of binary relations over instances, parametrized by environments. That is, for a given environment η:

$$[\![p]\!]\eta : Inst \to 2^{Inst}. \tag{1}$$

Now suppose t is a procedure defining a database transaction (so t is the sort of procedure that a predicate p specifies). Given an instance I and an environment η, t may return a new instance I', terminate with failure, or

may diverge. None of the procedures we describe in this paper will diverge, so we are considering procedures t that (under an environment) determine a function over instances:

$$[\![t]\!]\eta : Inst \rightarrow (Inst + fail). \tag{2}$$

Alchemy's job is precisely the following: given predicate p, construct a procedure $t = \mathsf{code}(p)$ such that the semantics of $\mathsf{code}(p)$ as given in Equation 2 refines the semantics of p as given in Equation 1 in the following sense.

Theorem 2 (Main theorem). *Let p be a predicate and let $\mathsf{code}(p)$ be a backtracking implementation of the algorithm $\mathbb{A}_p$, given in Definition 5 below. Then for each instance I and each environment η*

1. *$[\![\mathsf{code}(p)]\!]\eta$ terminates on I;*
2. *If there exists any instance I' such that (I,I') satisfies p under η then the result of $[\![\mathsf{code}(p)]\!]\eta$ is such an I'. In particular in this situation $[\![\mathsf{code}(p)]\!]$ does not return "failure" under η on I.*

It is worth noting that the task of generating updates from specification submits to an uninteresting trivial solution, particularly if we are willing to tolerate partial functions. Given predicate p we could define $\mathsf{code}(p)$ by:

> on input I, exhaustively generate all possible I'; for each one test whether (I,I') in $[\![p]\!]$. If and when such an I' is found, replace I by I'.

Obviously this is a silly algorithm, even though it is "correct" in a formal sense. Our goal with Alchemy is to write code that is intuitively reasonable, and still is correct in the sense of Theorem 2.

4 Code generation

Suppose we are given an Alloy predicate **p**. Alchemy generates code for a procedure with parameters corresponding to those of **p** (without the primed parameter).

As observed above, a crucial aspect of *Alloy* is that it encourages "lightweight" specifications of procedures: the designer is free to ignore details about the computation that she may consider inessential. As a consequence, *Alchemy* must be extremely flexible: different input instances may require quite different computations in order to satisfy a specification, yet Alchemy must generate code that works uniformly across all instances.

The top-level view of how Alchemy generates code for a procedure is as follows.

4.1 Outline

- In Definition 5 below we present a construction that, based on predicate p, builds a *non-deterministic procedure* $\mathbb{A}_p$.
- The code generated by Alchemy, $\mathsf{code}(p)$, is a backtracking implementation of $\mathbb{A}_p$. Computation paths that do not succeed are recognized as such and abandoned, and $\mathbb{A}_p$ is finite-branching, so $\mathsf{code}(p)$ will always terminate.
- If there exists any instance I' such that (I,I') satisfies p under η then some branch of $\mathbb{A}_p$ is guaranteed to compute such some such instance.

Coping with inconsistent predicates It is possible for the code for a predicate p to *fail* on a given database instance I, either because the predicate is internally inconsistent or because no update of I can implement p without violating the facts. Alchemy is guaranteed to detect such situations; we treat predicates as *transactions* that rollback if they cannot be executed without violating their bodies or a fact.

4.2 A normal form for predicates

The general form of an Alloy predicate that specifies an operation and that Alchemy treats is

$$\mathsf{pred}\ p(s,s' : \mathsf{State},\ a_1 : A_1, \ldots, a_n : A_n)\{\vec{Q}x\ .\ \beta(\vec{a},\vec{x})\}$$

where $\vec{Q}$ is a sequence of quantified atoms and β is a quantifier free formula of relational algebra. Before giving an imperative interpretation of a predicate it is convenient to massage it into a convenient form.

Skolemization By the classical technique of Skolemization any formula $\vec{Q}x\ .\ \beta(\vec{a},\vec{x})$ can be converted into a universal formula which is satisfiable if and only if $\vec{Q}x\ .\ \beta(\vec{a},\vec{x})$ is satisfiable. We exploit this trick in Alchemy as follows. Given a predicate p we convert it to a predicate $p^{\forall}$ whose the body is in universal form; this involves expanding the specification language to include the appropriate Skolem functions. Suppose we generate code for $p^{\forall}$ (over the expanded language). Then given an original instance I we may view it as an instance I_+ over the enlarged schema, and apply the generated code to obtain an instance I'_+. We ultimately return the instance I' that is the reduct of I'_+ to the original schema. So in what follows we restrict attention to predicates whose body is a universal formula.

Incorporating the facts Intuitively the facts in a specification comprise a separate set of constraints on how a predicate may build new instances from old ones. But by the following simple trick we can avoid treating the facts separately. When compiling a predicate to code we take each fact, prime every occurrence of the State sig, and add the fact to the body of the predicate. The use of primed State names means that the fact acts as a post-condition on the predicate. (Strictly speaking this is only true under an assumption of "state-boundedness" on the form of the facts, defined in [GDFK]. The specifics of this syntactic assumption are irrelevant to the current paper so we omit details) This in turn guarantees that any post-instance defined by the predicate will satisfy the facts.

The following is a convenient form for formulas.

Definition 3 (Special formulas). A *special* formula is a formula in either of the two forms

$$(e_1 \cap \ldots \cap e_k) = \emptyset \qquad \text{or} \qquad (e_1 \cap \ldots \cap e_k) \neq \emptyset$$

for $k \geq 1$, with each e_i not containing $\cup$ or $\emptyset$ and with converse applied only to variables and relation names.

Lemma 4. *Any quantifier-free formula can be transformed into an equivalent Boolean combination of special formulas.*

The proof of the Lemma is straightforward. The convenience afforded by special formulas will be made clear in the next section.

4.3 Algorithms

Bridging the declarative/imperative gap The main procedure $\mathbb{A}_p$ below is generated by an induction that walks the structure of the formula that is the body of p. There is a natural correspondence between the logical operators in the predicate and control-flow operators in the generated procedure. The disjunctive (logical $\vee$ and $\exists$) constructors in predicates naturally suggest imperative nondeterminism; this of course results in *backtracking* in generated code. Likewise, conjunctive (logical $\wedge$ and $\forall$) constructors lead naturally to *sequencing*. This is natural enough, but a difficulty arises due to the fact that the logical operators are commutative but command-sequencing certainly is not. Indeed, implementing one part of a predicate can undo the effect achieved by an earlier part. The solution is to iterate computation until a fixed-point is reached on the post-state. So we must be careful to ensure that such an iteration will always halt.

Compiling special formulas to code Consider for example the body of the *Drop* predicate in Figure 1. There are certainly many ways to update the data to make this true; for example we could delete all the tuples in the roster table! This is not what the specifier had in mind. But even this silly example points out the need for a principled approach to update. We start with the following goal: we attempt to make a *minimal* set of updates (measured by the number of tuples inserted or deleted into tables) to the system to satisfy the predicate.

The virtue of special formulas is that they facilitate identifying minimal updates to make a formula true. For example the formula a **in** $s'.r$, which, when a is an atom, is to say that a is in the relation $s'.r$ is equivalent to the formula $a - (s'.r) = \emptyset$. So suppose $a - (s'.r) = \emptyset$ is part of the body of a predicate. We evaluate the expression $a - (s'.r)$ in the prestate and the current poststate: if the value of this expression is indeed empty then there is nothing to do. If it is not empty then a is not in $s'.r$, and it is clear what action to take: add a to $s'.r$.

More generally, when confronted with a special formula $e = \emptyset$ we may view any tuples in the current value of e as *obstacles to the truth of the formula*. Then the action suggested by the formula is clear: make whatever insertions or deletions we can to ensure the formula becomes true. (The presence of the difference operator means that making an expression empty may involve insertions.) The important thing to note is that, obviously, we may focus exclusively on tuples that are already in the value of e in attempting to make $e = \emptyset$ in the updated state. This is our strategy for doing minimal updates for a predicate.

Inserting and deleting tuples We have seen that compiling a special formula amounts to orchestrating the insertion or deletion of individual tuples from the relations denoted by expressions. These expressions correspond to database *views*, and indeed the task of inserting or deleting a tuple from a view is an instance of the well-known *view update* problem [BLT86, BDH04]. Our code proceeds by a structural induction over the expression: see the procedures insertTuple and deleteTuple below.

Putting it all together After the preceding discussion the pseudocode for the Alchemy's translation algorithm should be largely self-explanatory. For simplicity in notation we adopt the following conventions. There are global variables pre-state and post-state ranging over instances, and a global variable Updates which keeps a record of the insertions and deletions done as the algorithm progresses.

We make use of the following function $\mathbb{E}\text{val}(e : \text{expression}, J, J' : \text{database instances})$ that returns the set of tuples denoted by expression e under the convention that immutable relation-name occurrences are interpreted in J and mutable relation-name occurrences are interpreted in J'. The pseudocode given here for procedures $\mathbb{A}_p$, $\mathbb{B}_p$, insertTuple, and deleteTuple is directly based on the discussion in the previous paragraphs.

Definition 5 (Algorithm $\mathbb{A}_p$)**.** Let p be a Alloy predicate of the form

$$\textsf{pred}\ \ p(s, s' : \textsf{State},\ a_1 : A_1, \ldots, a_n : A_n)\ .\ \{\forall \vec{x}\ .\ \bigwedge_i \bigvee_j \sigma_{i,j}\}$$

where each $\sigma_{i,j}$ is a special formula. The procedure $\mathbb{A}_p$ determined by p is as follows. Each of $\mathbb{A}_p$ and $\mathbb{B}_p$ reads the instance I globally and reads and writes I' and Updates globally.

procedure $\mathbb{A}_p$ (I: database instance) {
    initialize poststate I' to be I;
    initialize Updates to be empty;
    repeat $\mathbb{B}_p(a_1 : A_1, \ldots, a_n : A_n)$
    until no change in Updates
}
procedure $\mathbb{B}_p(a_1 : A_1, \ldots, a_n : A_n)$ {
    for each binding $\vec{b}$ of values in I for the variables in $\vec{x}$:
    let $\bigwedge_i \bigvee_j \bar{\sigma}_{i,j}$ be the body of p instantiated by $\vec{b}$:
        for each conjunct $\bigvee_j \bar{\sigma}_{i,j}$
            choose some $\bar{\sigma}_{i,j}$ and realize $\bar{\sigma}_{i,j}$ as follows:
            Case 1: $\bar{\sigma}_{i,j}$ is of the form $(e_1 \cap \ldots \cap e_k) = \emptyset$)
            set $e \equiv (e_1 \cap \ldots \cap e_k)$
            for each tuple t in $\mathbb{E}\text{val}(e, I, I')$:
                call *deleteTuple*(t, e, I, I');
            Case 2: $\bar{\sigma}_{i,j}$ is of the form $(e_1 \cap \ldots \cap e_k) \neq \emptyset$)
            set $e \equiv (e_1 \cap \ldots \cap e_k)$
            choose some t of the same type as e
                call *insertTuple(t, e. I, I')*
        update Updates accordingly;
}

procedure insertTuple(t : tuple, e: expression) {
match e:
atom a: if $a \neq t$ then FAIL else RETURN
immutable relation r: if $t \notin r$ then FAIL else RETURN
mutable relation r: if t has been previously deleted from r then FAIL
else add t to the table r in J'
$e_1 \cup e_2$: **choose** some e_i ; insertTuple(t, e_i)
$e_1 \cap e_2$: insertTuple(t, e_1) ; insertTuple(t, e_2)
$\sim e$: insertTuple(t, e)
$\langle e_1, e_2 \rangle$: let $t = \langle t_1, t_2 \rangle$ where t_i matches type of e_i; insertTuple(t_1, e_1) ; insertTuple(t_2, e_2)
$e_1 - e_2$: **choose:** insertTuple(t, e_1) or deleteTuple(t, e_2)
$e_1.e_2$: let T be the common sig-type that joins e_1 and e_2;
if T is the type of e_1 then for some a in $\mathbb{E}\text{val}(e_1, I, I')$, insertTuple$(\langle a, t \rangle, e_2)$
elseif T is the type of e_2 then for some a in $\mathbb{E}\text{val}(e_2, I, I')$, insertTuple$(\langle t, a \rangle, e_1)$
else **choose** $a : T$; set $t_1 = \langle s_1, a \rangle$ and set $t_2 = \langle a, s_2 \rangle$;
insertTuple(t_1, e_1) ; insertTuple(t_2, e_2)
$(e_1)^*$: insertTuple(t, e_1)

procedure deleteTuple(t : tuple, e: expression) {
match e:
atom a: if $a = t$ then FAIL else RETURN
immutable relation r: if $t \in r$ then FAIL else RETURN
mutable relation r: if t has been previously inserted into r then FAIL
else delete t from the table r in J'
$e_1 \cup e_2$: deleteTuple(t, e_1) ; deleteTuple(t, e_2)
$e_1 \cap e_2$: **choose** some e_i ; deleteTuple(t, e_i)
$\sim e$: deleteTuple(t, e)
$\langle e_1, e_2 \rangle$: let $t = \langle t_1, t_2 \rangle$ where t_i matches type of e_i; **choose** some e_i; deleteTuple(t_i, e_i)
$e_1 - e_2$: **choose:** deleteTuple(t, e_1) or insertTuple(t, e_2)
$e_1.e_2$: let T be the common sig-type that joins e_1 and e_2;
if T is the type of e_1 then for each a in $\mathbb{E}\text{val}(e_1, I, I')$, deleteTuple$(\langle a, t \rangle, e_2)$
elseif T is the type of e_2 then for each a in $\mathbb{E}\text{val}(e_2, I, I')$, deleteTuple$(\langle t, a \rangle, e_1)$
else for each $a : T$ such that for some s_1, s_2,
$\langle s_1, a \rangle = t_1$ is in e_1 and $\langle a, s_2 \rangle = t_2$ is in e_2 and $t_1.t_2 = t$;
choose e_i then deleteTuple(t_i, e_i)
$(e_1)^*$: for each $(x, y_1), (y_1, y_2), \ldots, (y_n, y)$ such that $t = (x, y)$ and each pair is in e_1
choose some pair (y_i, y_{i+1}); deleteTuple$(\langle y_i, y_{i+1} \rangle, e_1)$

4.4 Proof of correctness

Proof of Theorem 2 Theorem 2 follows from the following lemma about $\mathbb{A}_p$.

Lemma 6. *Let p be a predicate; let $\mathbb{A}_p$ be the non-deterministic procedure constructed from p by Definition 5. Then for every instance I and binding η for the parameters of p:*

1. *Every computation of $\mathbb{A}_p$ terminates on I under η, and if $\mathbb{A}_p$ returns an instance I', we have $(I, I') \in [\![p]\!]_\eta$;*
2. *If there is an instance I' such that $(I, I') \in [\![p]\!](\eta)$ then $\mathbb{A}_p$ will not fail.*

Proof of the lemma. For the first claim, first note that algorithm $\mathbb{B}_p$ proceeds by primitive recursion over the body of the predicates and algorithms insertTuple and deleteTuple proceed by primitive recursion over the body of expressions. So it suffices to argue that the iteration till fixed point in algorithm $\mathbb{A}_p$ always terminates. But this follows from the fact that we never add or delete the same tuple from a given relation and the total size of the domain we work with never changes. It is easy to see that when $\mathbb{A}_p$ halts without failure it is the case that the body of the predicate has been satisfied.

To establish the second claim we start with a definition. Given instances I and I' let us say that instance J is an (I, I')-*approximation* if

$$I - J \subseteq I - I' \qquad \text{and} \qquad J - I \subseteq I' - I.$$

We abuse notation slightly above: these calculations are done on a per-relation basis. Intuitively J is an (I, I')-approximation if J can be obtained from I by making *some* of the inserts and deletes that transform I into I'. Note that I is an (I, I')-approximation, as is I'. Now the second claim follows from the fact that, for initial instance I and chosen I' with $(I, I') \in [\![p]\!](\eta)$, whenever algorithm $\mathbb{B}_p$ is called (by $\mathbb{A}_p$) when the current value of the poststate is an (I, I')-approximation then there is a computation of $\mathbb{B}_p$ that (i) does not fail, and (ii) updates the poststate so that it still is an (I, I')-approximation. In particular $\mathbb{A}_p$ will never fail. □

Complexity There is nothing interesting that can be said about the run-time complexity of $\mathsf{code}(p)$ since it depends on the nature of the predicate p, and p can be an arbitrary predicate. On the other hand it is natural to ask about the complexity of $\mathsf{code}()$ itself. In other words, what is the running time *of Alchemy's code generation algorithm?* Since $\mathsf{code}(p)$ comprises a backtracking wrapper around the algorithm $\mathbb{A}_p$ the question is essentially the same as asking: what is the complexity of building the text of algorithm $\mathbb{A}_p$ from the text of predicate p? It is easy to see that this is linear in p. Note in particular that the procedures insertTuple and deleteTuple do not depend on p at all.

5 Related Work

For an extensive discussion of previous research relevant to the Alchemy project itself we refer the reader to the related work section in [KDFY08]. The relationship of the present paper to the previous work on Alchemy is as follows. In [KDFY08] we did not handle the relational difference operator, we did not treat Skolemization, and our correctness result was only for a subset of Alloy predicates (those admitting "homogeneous" implementations as defined there). But most importantly, the treatment of when relation names were evaluated in the pre-state and when in the post-state was ad-hoc: in the current paper this important semantic decision rests on the secure foundations of the work in [GDFK]. This allows us to prove a true soundness and completeness theorem (Theorem 2) for our code-generation algorithm.

References

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[BDH04] Vanessa P. Braganholo, Susan B. Davidson, and Carlos A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 276–287. Morgan Kaufmann, 2004.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In Carlo Zaniolo, editor, *SIGMOD Conference*, pages 61–71. ACM Press, 1986.

[GDFK] Theophilos Giannokopoulos, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Towards an operational semantics for Alloy. Submitted for publication.

[Jac06] Daniel Jackson. *Software Abstractions*. MIT Press, 2006.

[JW96] Daniel Jackson and Jeanette Wing. Lightweight formal methods. *IEEE Computer*, April 1996.

[KDFY08] Shriram Krishnamurthi, Daniel J. Dougherty, Kathi Fisler, and Daniel Yoo. Alchemy: Transmuting base Alloy specifications into implementations. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2008.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

Modeling and Reasoning over Distributed Systems using Aspect-Oriented Graph Grammars

Rodrigo Machado
Univ. Federal do Rio Grande do Sul
Porto Alegre, Brazil
rma@inf.ufrgs.br

Reiko Heckel
Univ. of Leicester
Leicester, UK
reiko@mcs.le.ac.uk

Leila Ribeiro
Univ. Federal do Rio Grande do Sul
Porto Alegre, Brazil
leila@inf.ufrgs.br

Aspect-orientation is a relatively new paradigm whose purpose is to provide better abstractions to represent system-wide policies. It is based on a composition operation that modifies a base system by performing changes at specific places. Aspect-oriented graph grammars are an extension of the classic graph grammar formalism, where aspects are defined as sets of rewriting rules that actuate over an original specification. Despite the obtained advantages of the paradigm, the implicit nature of the aspect weaving operation may introduce some issues when reasoning about the system behavior. By using aspect-oriented graph grammars we can apply known analysis techniques from the graph transformation literature to reason over diagrammatic aspect-oriented systems. In this paper, we present a case study of a distributed client-server system with global policies, modeled as an aspect-oriented graph grammar, and discuss how we may represent it in the AGG tool in order to analyze the aspect weaving operation.

1 Introduction

Aspect-oriented programming [10] is a relatively new paradigm that aims to provide better abstractions to global system requirements that usually affect many modules. It is based on a composition operation, called aspect weaving, that modifies a base system globally according to a global policy such as, for instance, "register in a global log all modifications in the value of the variable x of class C". As characterized in [6], an aspect is a module that *i)* identifies in other modules sets of execution points, which are called *pointcuts*, and *ii)* define transformation rules associated with pointcuts. Those rules are called *advices*. Once we have a pointcut language expressive enough, the implementation of global policies become extremely small, modularized and consistent as the system evolves, i.e. new modules will abide by the global policy in the same way as the current ones.

Those advantages stimulated the adoption of aspect-oriented programming in software development. Several languages now have aspect-oriented extensions, the most popular being the Java superset AspectJ [9]. Moreover, the usage of AOP-related concepts also started to appear in languages for system modeling, such as UML diagrams [8]. However, the widely usage of aspect-oriented concepts also introduces issues in the software development process. The pervasiveness of aspect influence may introduce unpredictable behaviors which are difficult to reason about by source code analysis. Also, when the system has more than one aspect they may interfere with each other, resulting in different final systems according to the order they are combined. To deal with those problems, the developer needs proper models to reason consistently about the aspect influence. On the formal side, several aspect-oriented calculi have been proposed to characterize aspect interference over programming languages [13, 7, 5, 4]. On the implementation side, integrated development environments start to offer support to new views related to aspect weaving [1]. However, outside the scope of source-code level aspects, there are still few models and techniques available to reason about aspect-oriented diagrams.

The current proposals for studying aspect weaving over diagrams have a strong connection with graph grammars, models where the system state is represented by a graph, and its execution, by the application of graph rewriting rules. This is due their common characteristics: diagrams may be naturally encoded as graphs, pointcuts resemble matches for graph rules, and advices resemble graph rules themselves. In [11], aspect-oriented graph grammars (AOGGs) were proposed as an extension of the traditional graph grammars, where aspects were modeled as second-order transformations over the original specification. The advantage of this approach is that the same rewriting mechanism is used for both aspect-composition and the base system execution, allowing to relate them formally. However, up to now it was not shown how to reason about AOGG models. In this work, we propose the use of AGG [2], a attributed graph grammar specification and analysis tool to reason over AOGG models. Since AGG does not support AOGGs, we propose to encode the whole base graph grammar as a single typed graph, with aspects being modeled as sets of rewriting rules.

The text is organized as follows: in Section 2, we review the graph grammar model, and introduce the base client-server example. In Section 3, we recall aspect-oriented graph grammars and present an example of aspect over the base model. In Section 4 we present the encoding of AOGGs as typed graphs in order to use the AGG tool, and discuss about the analysis results we may achieve using its capabilities. Final remarks, related work and future steps are discussed in Section 5.

2 Graph Grammars

A typed graph grammar is a visual model where the states of the system are graphs and the system behavior is described by the application of graph rewriting rules. Formally, a typed graph grammar is a tuple $\langle T, G_0, P, \pi \rangle$. The graph T is said to be the type graph and defines the kinds of nodes and edges allowed within the specification. The graph G_0 is the initial state of the system. The set P represents a set of rule names, and the function $\pi : P \to Rules(T)$ map every rule name to is respective typed graph transformation rule. A graph transformation rule is specified by a left-hand side (LHS) graph, and a right-hand side (RHS) graph. The LHS graph represents the pattern to be found within the current graph in order to apply the rule. The RHS graph is a modification of the LHS graph, with some elements being deleted, some being created, and some being preserved. The preserved elements must be identified as the same in the LHS and the RHS graphs. Roughly, the execution of a graph grammar may be described by the following steps:

1. Set G_0 as the current graph.
2. Find in the current graph all possible occurrences (or matches) of LHS graphs of rules in P.
3. If there is no match at all, then STOP. Otherwise, non-deterministically choose a rule and a match to be applied.
4. Delete from G all matched elements that occur in the LHS but not in the RHS. This will generate a graph G^-.
5. Create in G^- all matched elements that occur in the RHS but not in the LHS. This will generate a graph G^+.
6. Set G^+ as the current graph. Return to step 2.

A graph rewriting tool such as AGG also allows the use of several useful extensions to the basic typed graph grammar language, such as attributes for graph elements, layered rule execution or the definition

of application conditions for rules. Those and other features reduce considerably the effort of system modeling using graph grammars.

Example 1 (Graph grammar)**.** *Figure 1 depicts a simple example of a distributed client-server system modeled as a graph grammar. The type graph* `T` *defines four kinds of nodes (*`Client`*,* `Server`*,* `Data` *and* `Message`*) and four kinds of edges. The initial graph* G_0 *defines the initial state of the system: two clients with messages to be sent to three servers with data. The behavior of the system is given by its set of rules. The clients may retrieve values from servers or update the information contained in them by message exchange. A* `GET` *message is sent to the server by the rule* `SendGET`*, then it obtains the information by the rule* `ExecuteGET`*, and it is returned to the client by the rule* `ReceiveGET`*. The rules* `SendSET`*,* `ExecuteSET` *and* `ReceiveSET` *work in a similar way for* `SET` *messages.*

3 Aspect-Oriented Graph Grammars

Aspect-Oriented Graph Grammars (AOGGs) [11] are graph grammars with aspects, i.e. modular descriptions of system-wide policies. Formally, an AOGG is a pair $\langle \mathscr{G}, \Delta \rangle$, where $\mathscr{G} = \langle T, G_0, P, \pi \rangle$ is a base graph grammar, and $\Delta = [A_1, A_2, \ldots, A_n]$ is a sequence of graph aspects over $\mathscr{G}$. A graph aspect represents a set of modifications over the base graph grammar, and it is defined by a triple $\langle D, t, g \rangle$, where D is a set of graph advices, $t : T \hookrightarrow T'$ is an extension of the original type graph and $g : G_0 \hookrightarrow G'$ is an extension of the original initial graph. Graph advices modify the base system rules according to a given rule pattern. Advices are specified in the same way as conventional rules, but their LHS and RHS are graph rules. To make a distinction between advices and base system rules, the components of a graph advice receives new names: *pointcut* for the LHS, and *effect* for the RHS. As an example of graph aspect, we define a logging policy for the graph grammar of Figure 1.

Example 2 (Log Aspect)**.** *Suppose we want to implement a logging mechanism over the client-server system such that every operation leaves an execution trace over a global object (the system logger). To implement this functionality, we should extend the type graph to introduce the logger type, initialize it in the initial graph and modify all rules to register their execution in this global object. Using AOGG, all this modifications can be enclosed within one single aspect, as shown in Figure 2. This aspect has only one advice, which has an empty pointcut. The advice effect adds a* `Logger` *object to both the LHS and RHS of the rule such that the occurrence on the RHS has the rule name appended to the* `log` *string.* [1] *The empty pointcut matches all possible rules of the specification, thus all of them will be modified to read the global log object and to record its respective execution.*

Given a graph grammar $\mathscr{G} = \langle T, G_0, P, \pi \rangle$ and a graph aspect $< D, t : T \to T', g : G_0 \to G'_0 >$ over $\mathscr{G}$, the result of weaving A and $\mathscr{G}$ is the graph grammar $\mathscr{G}' = \langle T', G'_0, P', \pi' \rangle$. The pair (P', π') defines the rules of the resulting graph grammar, and is calculated based on (P, π), as follows:

- If a rule in (P, π) is not matched by any advice in A, them the rule appears in (P', π').
- If a rule is matched by at least one advice in A, all of their rewritings (considering all advices and matchings) appears in (P', π').

Notice that the rewriting for advices is non-reentrant, i.e. a given rule may not be modified more than once per advice and match. This assures termination for the weaving process, even for advices that do not delete anything from rules, such as the unique advice of the log aspect. The semantics of an AOGG $\langle \mathscr{G}, \Delta \rangle$ is given by its weaved graph grammar $\mathscr{G}_W^{\Delta}$, which is the result of weaving all aspects of Δ over $\mathscr{G}$ in order.

[1] this is possible only when information about rules is available as data within the model.

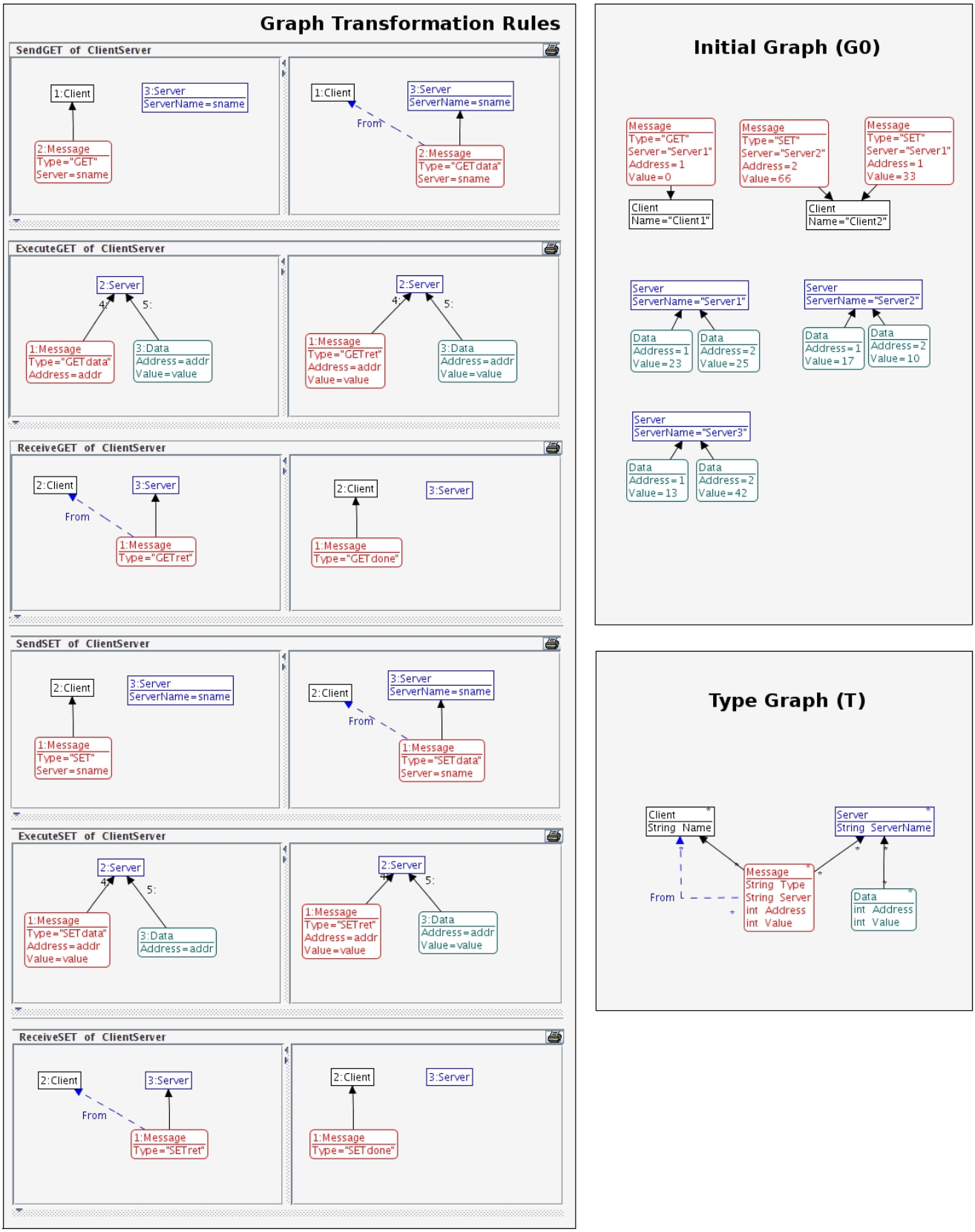


Figure 1: Client-server system as a graph grammar.

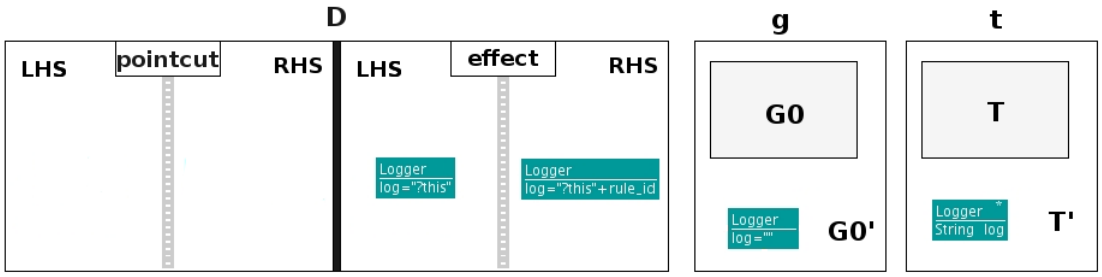


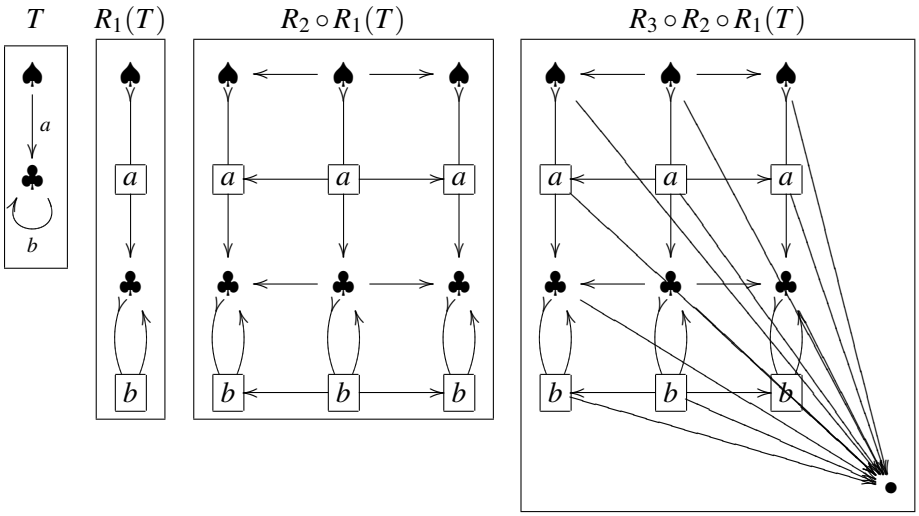
Figure 2: Log Aspect

4 Encoding AOGGs in AGG

The use of graph rewriting as an aspect weaving mechanism opens the possibility of using known techniques from the graph transformation area to reason about aspect-oriented systems. Those techniques allow the system modeler to identify potentially harmful behaviors of the final weaved system, such as unintended aspect interactions, at early stages of the system development. To be able to use existing tools, such as AGG, for modeling and analysis of aspects over graph transformation system, we propose an encoding of the whole base graph grammar as one single graph. Then, advices may be represented as conventional system rules and may be tested for conflicts and dependencies.

The main idea is to augment the type graph T with elements that allow the unambiguous representation of graph rules. Those new elements are calculated automatically based on the type graph T by means of the operation R. This operation is defined as $R_3 \circ R_2 \circ R_1$, the composition of three simpler transformations, described as follows. The transformation R_1 turns edges into nodes. The transformation R_2 creates three copies of the input graph, connected element-wise by edges. The left copy represent elements in the LHS, the right copy, elements in the RHS, and the central copy, elements that are preserved by the rule (i.e. that are the same in the LHS and the RHS). The transformation R_3 connects all types to a single node. This last step is needed to describe individual rules.

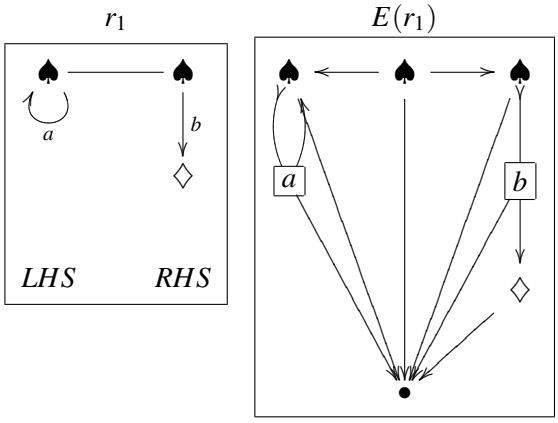
Example 3 (Type graph encoding R)**.** *The following example shows how to obtain $R(T)$ from a simple type graph T, stepwise.*



Let $\mathscr{G} = \langle T, G_0, P, \pi \rangle$ be a graph grammar, where $r_1, \ldots, r_n \in range(\pi)$ represent the individual rules. We say that the graph $G^{\mathscr{G}}$, defined as $G_0 + E(r_1) + \ldots + E(r_n)$ and typed over $T + R(T)$, is as the encoded representation of $\mathscr{G}$. The "+" operation means disjoint union of graphs. The initial graph G_0 remains typed over T, while all the rules are converted by means of the operation E to graphs typed over $R(T)$. The typing discipline allows one to differ the initial graph from the rules within $G^{\mathscr{G}}$.

The operation E encodes the rule structure into a graph. Initially it converts edges of LHS and RHS into nodes, just like R_1. Next, it executes the disjoint union of the LHS, RHS and preserved elements graph, and puts edges between the preserved elements and their respective representations in the LHS and RHS. Finally, it creates a new node and connects all other nodes to it.

Example 4 (Rule encoding E)**.** *The figure below shows the effect of the encoding E over a simple rule that deletes one edge, preserves one node and creates a new node with and incident edge.*



By using the transformations R and E, we are able to enter a whole base system as the initial graph of an AGG specification, as shown in Figure 3. However, the graph aspects must also be encoded. Given a graph aspect $A = \langle D, t : T \to T', g : G_0 \to G_0' \rangle$, type graph extension t and initial graph extension g may be directly applied over the specification. The only subtlety comes from the fact that the type graph becomes $T' + R(T')$, instead of $T' + R(T)$. When representing advices as graph rewriting rules, the following should be taken into account:

- the rule marker (node introduced during the last step in both R and E transformations) and their respective incident edges are needed to assure that the rewriting rule does not actuate over elements of different rules.
- it may be necessary to define application conditions (either positive or negative) in order to assure one-step rewriting.

The AGG tool allows to reason about a given graph grammar specification in many ways. From the point of view of aspect weaving, two are particularly interesting: critical pair analysis and termination checks. Critical pairs formalize the idea of a minimal example of a conflicting situation between rewriting rules. From the set of all critical pairs we can extract the objects and links that cause conflicts or dependencies. In the context of AOGG, it means that we have tool support to verify conflicting situations between aspects. The termination checker is important to help the system designer to avoid sets of graph advices that may generate infinite rewriting.

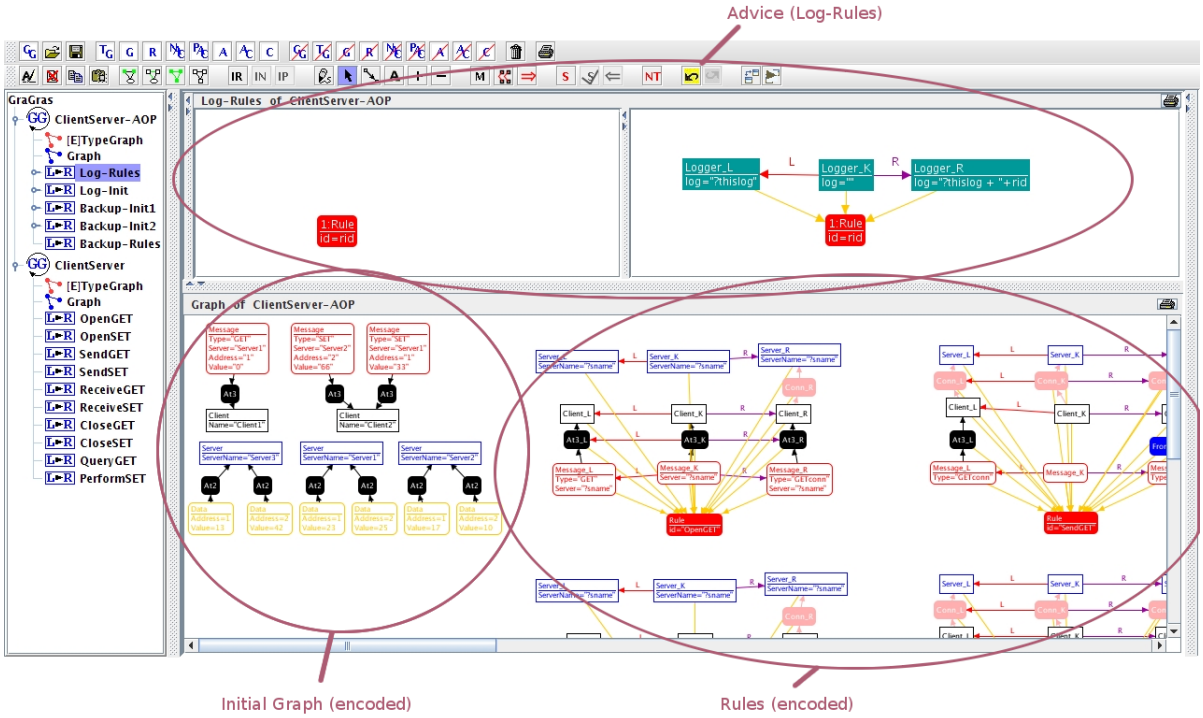


Figure 3: Client-Server system with Logging and Server Replication in AGG

5 Final Remarks

This work proposed the use of AOGG to model and reason over aspect-oriented diagrams. We started motivating the need for verification tools in aspect-oriented contexts. Next, we presented aspect-oriented graph grammars as a modeling tool to describe global policies over graph grammars. We described how to encode a whole graph grammar into a single typed graph specification, in order to fit an AOGG specification into the AGG tool. Finally, we briefly discussed about how AGG may be used as a reasoning tool to verify weaving in aspect-oriented models.

There are also some other proposals for studying aspects using graph grammars. The MATA approach [14], for instance, uses graph representation of diagrams to perform aspect-oriented composition. In [12], graph transformation used to verify aspect conflicts and dependencies between aspects defined over a variant of UML to model requirements. Both use AGG as modeling and analysis tool. The work by Aksit et al. [3] encodes a aspect-oriented formal language into a graph transformation tool, and uses the tool capabilities to study the interference of aspects in the space state of the rewriting. This last work uses the Groove tool.

As future work, we intend to define different sets of aspects over this base specification and explore the different kinds of interaction between them.

References

[1] Eclipse AJDT: AspectJ Development Tools. `http://www.eclipse.org/ajdt/`.

[2] The AGG Homepage. http://user.cs.tu-berlin.de/~gragra/agg/.

[3] Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 39–50, New York, NY, USA, 2009. ACM.

[4] Curtis Clifton and Gary T. Leavens. Minimao$_1$: an imperative core language for studying aspect-oriented reasonings. *Sci. Comput. Program.*, 63(3):321–374, 2006.

[5] Simplice Djoko Djoko, Rémi Douence, Pascal Fradet, and Didier Le Botlan. Casb: Common aspect semantics base,. Technical report, Research Report, Network of Excellence in AOSD (AOSD-Europe, August 2006, no D54)., 2006.

[6] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[7] Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Sci. Comput. Program.*, 63(3):267–296, 2006.

[8] José Uetanabara Júnior, Valter Vieira Camargo, and Christina Von Flach Chavez. UML-AOF: a profile for modeling aspect-oriented frameworks. In *AOM '09: Proceedings of the 13th workshop on Aspect-oriented modeling*, pages 1–6, New York, NY, USA, 2009. ACM.

[9] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP*, pages 327–353, 2001.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[11] Rodrigo Machado, Luciana Foss, and Leila Ribeiro. Aspects for graph grammars. In *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques*, 2009.

[12] Katharina Mehner, Mattia Monga, and Gabriele Taentzer. Interaction analysis in aspect-oriented models. *Requirements Engineering, 14th IEEE International Conference*, pages 69–78, Sept. 2006.

[13] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, New York, NY, USA, 2003. ACM Press.

[14] Jon Whittle and Praveen K. Jayaraman. Mata: A tool for aspect-oriented modeling based on graph transformation. In Holger Giese, editor, *MoDELS Workshops*, volume 5002 of *Lecture Notes in Computer Science*, pages 16–27. Springer, 2007.

Object-oriented Programming Laws for Annotated Java Programs

Gabriel Falconieri Freitas, Márcio Cornélio
Universidade de Pernambuco (UPE), Brazil
{grff, marcio}@dsc.upe.br

Tiago Massoni, Rohit Gheyi
Universidade Federal de Campina Grande (UFCG), Brazil
{massoni, rohit}@dsc.ufcg.edu.br

Object-oriented programming laws have been proposed in the context of languages that are not combined with a behavioral interface specification language (BISL). The strong dependence between source-code and interface specifications may cause a number of difficulties when transforming programs. In this paper we introduce a set of programming laws for object-oriented languages like Java combined with the Java Modeling Language (JML). The set of laws deals with object-oriented features taking into account their specifications. Other laws deal only with features of the specification language. These laws constitute a set of small transformations for the development of more elaborate ones.

1 Introduction

Programming laws serve as guidelines to informal programming practices and establish a basis for formal and rigorous program development. They are largely known for imperative programming [12, 18]. Also, functional programming and logic programming have a set of laws described by Bird and de Moor [3] and Seres [19], respectively. Laws of object-oriented programming have also been addressed in [4, 6, 7].

Design by Contract (DbC) [17] is development methodology that aims at the construction of reliable object-oriented systems. Its basic idea is that a contract is established among classes of a system. In this way, software developers should formally specify what is required and ensured by methods and types. The Java Modeling Language (JML) [1, 14] is a notation for formally specifying the behavior of Java classes and methods.

The set of programming laws for object-oriented programming we have nowadays is designed for program transformation with no relation to specifications languages designed for DbC. Changes in specification usually should discharge code updates, maintaining the conformance between code and specification. On the other hand, changes in program code may require changes in specifications as the behavior implemented by code may diverge from the meaning of the original specification. For instance, moving a redefined method to its superclass can be illegal if this transformation causes weakening of pre-conditions and strengthening of post-conditions.

In this paper, we define laws (Section 3) of object-oriented programming for Java that are aware of specifications written in JML, which we describe in Section 2. Our proposed hybrid laws were created by extending object-oriented programming laws from other works [4, 6, 7, 16]. Additionally, we present a law for invariants written in JML. The laws precisely indicate the modifications that can be done to a program, stating their corresponding proof obligations that are discharged for application. In Java and JML context, we need to guarantee that source-code continues meeting its specifications written in JML, taking into account the semantics of JML specifications along with the notion of specification inheritance [13]. The laws we present in this paper and other ones present in a more comprehensive set were applied to refactoring a JML-specified version of a core module from a Manufacturing Execution System (MES) [21].

This is a preliminary version of a paper
that will appear in Electronic Proceedings
in Theoretical Computer Science.

© Gabriel Falconieri Freitas et al.
This work is licensed under the Creative Commons
Attribution-Noncommercial-No Derivative Works License.

```
1  public class Person {
2    private /*@ spec_public @*/ String name;
3    private /*@ spec_public @*/ int weight;
4    /*@ private invariant !name.equals("") && weight >= 0;   @*/
5    public Person () { /* ... */ }
6
7    //@ also
8    //@ ensures \result != null;
9    public String toString() { /* ... */ }
10
11   //@ ensures \result == weight;
12   public /*@ pure @*/ int getWeight() { /* ... */ }
13 /*@ requires kgs > 0 &&  weight + kgs >= 0
14   @ assignable weight;
15   @ ensures weight == \old(weight + kgs); @*/
16   public void addKgs(int kgs) { /* ... */ }
17 }
```

Figure 1: JML specification of the class `Person`.

2 The Java Modeling Language

The Java Modeling Language (JML) is a behavioral interface specification language (BISL) [1, 14] tailored to Java [11]. Thus, JML serves to describe names and static information that appear in Java declarations and how they act, how they behave. JML specifications are written in the form of *special annotation* comments that are inserted directly in source code of programs. These comments must begin with an at-sign (@) and can be written in two ways: by using **//@ ...** or **/*@ ... @*/**. Comments in the forms **//** @ and **/*** @ are ignored by the JML compiler due to space between the backslashes and the at-sign(@).

In Figure 1 we present the class `Person` (this example was originally presented in [14]) . In Line 4, we introduce an instance invariant, which is a predicate that is true in all visible states of objects of a class [1]. The invariant in the example has public visibility and establishes that the value of attribute `name` is different from an empty string and the value of `weight` is greater than or equal to zero. The keyword **also** indicates that the method `toString` is extending the specification it inherits from its supertype.

JML uses the **requires** clause to specify the obligations of the caller of a method, what must be true to call a method. For instance, the precondition of the method `addKgs` insists on the value to be added to be greater than zero. A postcondition specifies the implementor's obligation, what must be true at the end of a method, just before it returns to the caller. In JML, the **ensures** clause introduces a postcondition. For instance, the Line 15 introduces a normal postcondition that asserts the value of the attribute `weight` at the end of the method `addKgs` is equal to the value of the expression "**\old**(`weight + kgs`)". By using the **\old** (.) operator we can refer to the value of an expression in the pre-state of a method.

The **assignable** clause gives a frame axiom for a specification. Only locations named, and locations in data groups (a set of locations) associated with these locations, can be assigned during method execution. In Line 14, we state that only the attribute `weight` is assigned in method `addKgs`. In Line 11, we state that when we execute the method `getWeight`, it returns the value of the attribute `weight`. The JML modifier **pure** (Line 12) indicates that the method doesn't have any side effects and hence can appear in specifications.

3 Laws

A catalog of primitive transformations (laws) to deal with JML annotations and JML-aware Java programs has been proposed in [9], which specifies about 80 laws. Here we present one law that deals only with JML specifications and two hybrid JML-aware Java laws that deals with attributes and methods, respectively. A law that only deals with JML can impose conditions only on JML elements present in the program, whereas laws that deal with Java code can involve both JML and Java elements for stating conditions.

The laws are written in an equational style. Each side of the equation corresponds to a template of a well-formed program. Programming laws, in which left-hand and right-hand sides are related by equality, are a concise presentation of a pair of laws. These laws precisely indicate the modifications that can be done to a program, stating their corresponding proof obligations. In fact, to apply a law, it is necessary to check (syntactic or semantic) side-conditions that ensure that the transformation is behavior-preserving and also maintains its well-formedness. In our approach we consider that we are dealing with only one package and working in a limited open system [7], in which classes of our system can depend on external libraries.

In the laws, we use $cds_1 =_{cds,Main} cds_2$ to denote the equivalence of sets of class declarations cds_1 and cds_2, where cds is a context of class declarations for cds_1 and cds_2. $Main$ corresponds to the unique class in the program which has a static main method. We use $cnds$, ads and mds inside a class to represent the class constructors, attributes and methods, respectively. We write '$\rightarrow$' to indicate the condition that need to be satisfied to apply a law from left to right. Likewise, we use '$\leftarrow$' to indicate what has to be satisfied when applying the law from right to left. Conditions that must hold in both directions are indicated by '$\leftrightarrow$'.

Law 1 ⟨*move invariant to superclass*⟩

class B **extends** A {
  //@ **invariant** ψ_1;
 	
  ads
  $cnds$
  mds
}
class C **extends** B {
  //@ **invariant** ψ'_1 && ψ'_2;
 	
  $cnds'$
  mds'
}
	$=_{cds,Main}$	**class** B **extends** A {
		  //@ **invariant** ψ_1 && ψ'_2;
		
		  ads
		  $cnds$
		  mds
		}
		class C **extends** B {
		  //@ **invariant** ψ'_1;
		
		  $cnds'$
		  mds'
		}

where

$\psi'_2 \mathrel{\widehat{=}}$ **this instanceof** C ==> $\psi_{\texttt{inv}}$

provided

(↔) **super** does not appear in ψ'_2.

(→) ψ'_2 does not contain occurrences of model fields declared in C, nor uncast occurrences of **this**.

□

The first law we present (**Law 1**) allows us to move an invariant ψ'_2 from a subclass C to its superclass B. The invariant we want to move only refers to instances of C as we require the invariant to be applicable only to instances of class C. To apply this law in any direction, we require that calls to **super** do not occur in ψ'_2, since after law application (in both directions) these calls may refer different elements. To apply this law from left to right, model fields cannot appear in ψ'_2 and occurrences of **this** must be cast otherwise the elements they refer may not be visible.

Concerning the soundness of this law, we take in account the inheritance of specifications in JML [13], in which inherited invariants are conjoined with locally added invariants. On the left-hand side, the invariant ψ'_2, which is present in class C, is inherited by the subclasses of C and holds for all subclasses. On the right-hand side of the law, the invariant ψ'_2 is inherited by all subclasses of B besides those that are not subclasses of C. For those classes that are subclasses of B, but not subclasses of C, the invariant holds because for objects of these classes the antecedent **instanceof** C fails and the whole implication is true, not changing the meaning of any original local invariant that inherits ψ'_2.

Law 2 ⟨*move reference type attribute to superclass*⟩

```
class B extends A {                    =cds,Main    class B extends A {
  ads                                                   /*@ nullable @*/ T a;
  cnds                                                  ads
  mds                                                   cnds
}                                                       mds
class C extends B {                                  }
  /*@ nullable @*/ T a;                             class C extends B {
  ads'                                                  ads'
  cnds'                                                 cnds'
  mds'                                                  mds'
}                                                    }
```

provided

JML:

(←) $D.a$, for any $D \leq B$ and $D \not\leq C$ does not occur inside specifications of *cds*, *Main*, *cnds*, *cnds'*, *mds* nor *mds'*.

Java:

(↔) T is not a primitive type.

(→) (1) a is not declared in *ads*; (2) The attribute name a is not declared by the subclasses of B in *cds*.

(←) $D.a$, for any $D \leq B$ e $D \not\leq C$ does not occur in *cds*, *Main*, *cnds*, *cnds'*, *mds* nor *mds'*.

□

By using **Law 2**, we can move an attribute to a superclass if it is not already declared in the superclass and if it does not cause name conflicts. The application of **Law 2**, from right to left, allows us to move an attribute downward. In this case, we prevent access to the attribute by the expression **this**, and we allow only accesses to a by C or subclasses of C, including accesses that appear in specifications.

In **Law 2**, we consider only attributes whose type is a reference type. There is another law for moving an attribute of primitive type. The reason for having two disctinct laws for dealing with attributes of primitive and reference types comes from the **nullable** keyword in **Law 2**. In JML, any declaration (except for local variables) whose type is a reference type is implicitly declared to be not null, except

when one adorns the declaration with a **nullable** annotation [1]. Thus, by default, JML always checks if a not nullable attribute is null in all visible states of the class that declares it. When we move an attribute to a superclass, this is not aware about the newly moved attribute and, therefore, this action can cause a undesirable behavior. In fact, if one instantiates the superclass, JML will raise an invariant exception reporting that the new attribute is null. To avoid this, we force attribute nullability to move it up. Then, if one wants to move a non-null a attribute, one needs to introduce **nullable** annotation before moving it. An attribute can become nullable applying a law named *make attribute nullable*, not presented here. Remember that, in Java, only reference types can be null.

Law 3 ⟨*move redefined method to superclass: overriden method with non-default specification case*⟩

```
class B extends A {                          class B extends A {
  ads                                          ads
  cnds                                         cnds
                                               //@ requires (!(this
  //@ requires ψ1;                                 instanceof C) && ψ1);
  //@ ensures ψ2;                              //@ ensures (!(this
  rt m(pds) { mbody }                             instanceof C) && ψ2);
  mds                                          //@ also
}                                              //@ requires (this
class C extends B {                  =cds,Main     instanceof C && ψ'1);
  ads'                                         //@ ensures (this
  cnds'                                           instanceof C && ψ'2);
                                               rt m(pds) {
  //@ also                                       if (!(this instanceof C))
  //@ requires ψ'1;                              { mbody } else { mbody' }
  //@ ensures ψ'2;                             }
  rt m(pds) { mbody' }                         mds
  mds'                                       }
}                                            class C extends B {
                                               ads' cnds' mds'
                                             }
```

provided

JML:

(↔) **super** does not appear in ψ'_1 nor in ψ'_2.

(→) Both ψ_1 and ψ_2 do not contain occurrences of model fields declared in C, nor uncast occurrences of **this**.

Java:

(↔) (1) **super** and private attributes do not appear in *mbody'*; (2) **super**.*m* does not appear in *mds'*

(→) *mbody'* does not contain uncast occurrences of **this** nor expressions of the form ((*C*)**this**).*a* and of the form ((*C*)**this**).*m*(*e*) for any attribute *a* nor method *m*, in *ads'* and *mds'*, respectively, with private visibility.

(←) *m*(*pds*) is not declared in *mds'*.

□

The last law we present here (**Law 3**), allows us to move a redefined method from a class to its superclass. The proviso concerning **super** is needed because its semantics may be affected when we move it from a subclass to a superclass, or vice-versa. We can only move the specification of a method if it does not refer to model fields of the class in which the method is originally declared. Furthermore, **this** expressions may occur in the target method specifications only if they are cast. In fact, as in the law the method has default visibility, only non-private elements can be referenced in its pre- and postconditions. This is similar to Java: the **this** expression may appear in *mbody'* if they have a cast and they mention only non-private attributes or methods of class C. The right-side of **Law 3** introduces **instanceof** tests in each one of the specifications. In this way we assure that the original pre- and postconditions of the redefined method of C will only be applied to callers that are instances of C or instances of any of C's subclasses.

4 Proving Laws

The proofs we present here are only concerned with the JML parts of the laws. In JML, specifications present in a class are inherited by its subclasses, provided they are not private. This leads us to two concepts: join of specifications and specification inheritance.

4.1 Join of specifications

In a program written in Java and annotated with JML, classes inherit not only attributes and methods from superclasses, they also inherit specifications of invariants, methods, history constraints, and initialisation predicates [13, 15]. Concerning methods, a method specification may consist of several specifications cases, which are introduced by the use of clauses such as **requires**, **assignable**, **ensures** [1]. Each specification case has a precondition (the default predicate is *true*) that states when the corresponding specification case applies to a call. The keyword **also** joins specifications cases. When a precondition of a specification case holds, the corresponding postcondition must hold also. The definitions we present here are taken from [15]. The notation $T \triangleright (pre, post)$ is related to a specification case of an instance method that type checks when its receiver (**this**) has static type T. It also type checks in contexts where **this** has some subtype of T. In what follows, we introduce the definition of the join of JML method specifications [15].

Definition 1 ***(Join of JML method specifications)*** *Let $T' \triangleright (pre', post')$ and $T \triangleright (pre, post)$ be specifications of an instance method m. Let U be a subtype of both T' and T. Then the join of $(pre', post')$ and $(pre, post)$ for U, written $(pre', post') \sqcup^U (pre, post)$, is the specification $U \triangleright (p, q)$ with precondition p:*

$$pre \parallel pre'$$

and postcondition q:

$$(\backslash old(pre') ==> post') \;\&\&\; (\backslash old(pre) ==> post)$$

□

In Definition 1, the precondition of the join of two method specifications is their disjunction. The postcondition of the join is a conjunction of implications (written ==> in JML's notation), stating that when a precondition holds (in the pre-state), the corresponding postcondition must hold.

4.2 Specification Inheritance

Specifications of subtypes in JML inherit specifications, besides attributes and methods. First, we introduce some notation for type specification. For a type T, the invariant predicate declared in the specification of T (without inheritance) is denoted by $added\_inv^T$. For a method m declared in a type T, the notation $added\_spec_m^T = (added\_pre_m^T, added\_post_m^T)$ is the join of the specification cases in type T for m. If m is declared in T with no specification and is not overriding any method, then $added\_spec_m^T = (true, true)$, which is the default specification in JML. We use $supers(T)$ to denote the set of all supertypes of T (including T) and $methods(\mathcal{T})$ to denote the set of all instance method names declared in the specifications of the types in a set $\mathcal{T}$.

Definition 2 ***(Extended specification)*** *Suppose T has supertypes $supers(T)$, which includes T itself. Then the extended specification of T is a specification such that:*
methods: *for all methods $m \in methods(supers(T))$, the extended specification of m is the join of all added specifications for m in T and all its proper supertypes*

$$ext\_spec_m^T = \bigsqcup^T \{added\_spec_m^U \mid U \in supers(T)\}$$

invariant: *the extended invariant of T is the conjunction of all added invariants in T and its proper supertypes*

$$ext\_int^T = \bigwedge^T \{added\_inv^U \mid U \in supers(T)\}$$

□

The definitions we present here were introduced in [15] and are the ones we use in this paper.

4.3 Proofs

Here we present proofs for **Laws 1 and 3**. Both proofs involves dealing with cases associated to the types of objects related to the classes that are emphasised in the laws. We present the proof for just one case of these laws. The provisos of the laws guarantee that both programs that appear in the laws are well-typed. Concerning **Law 2**, it is a law for attributes in which specification inheritance is not taken into consideration.

In Figure 1, we present the proof for the case of **Law 1** in which the we consider an object of exact type B. Notice that in **Law 1**, on the left-hand side, an object of exact type B has to establish the (added) invariant ψ_1. The added invariant is given by $\psi_1 \wedge (this\ instanceof\ C \Rightarrow \psi_{inv})$, on the right-hand side. For an object of type B, the type test is false and the whole implication results true. The whole effect is the same of the invariant of class B on the left-hand side.

The proof for the case of **Law 3** in which we consider an object of exact type B is presented in Figure 2. On the left-hand side of this law, the specification case for method m in class B has precondition ψ_1 and postcondition ψ_2. On the right-hand side, we enrich this specification case with type tests involving the class name C, but with no impacts for objects with distinct types from C. The other specification case for method m on the right-hand side also involves a type test, having no effects for classes other than class C.

5 Conclusion

In this paper, we propose laws for object-oriented programming in the presence of a behavioral interface specification language. Focusing the hybrid laws, we treat source-code transformation considering the

$$
\begin{array}{ll}
 & ext\_inv^{B}_{LHS} \\
= & \text{[by Definition 2]} \\
 & \bigwedge\{added\_inv^{U} \mid U \in supers(B_{LHS})\} \\
= & \text{[by set theory]} \\
 & \bigwedge\{added\_inv^{U} \mid U \in ((supers(B_{LHS}) \setminus supers(A)) \cup supers(A)\} \\
= & \text{[by definition of conjunction]} \\
 & (\bigwedge\{added\_inv^{U} \mid U \in ((supers(B_{LHS}) \setminus supers(A))\}) \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\}) \\
= & \text{[by definition of added invariant in } B_{LHS}\text{]} \\
 & \psi_1 \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\}) \\
= & \text{[by Propositional Logic]} \\
 & \psi_1 \wedge true \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\}) \\
= & \text{[by Propositional Logic]} \\
 & \psi_1 \wedge (false \Rightarrow \psi_{inv}) \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\}) \\
= & \text{[by type test for object of type } B\text{]} \\
 & \psi_1 \wedge (this\ instanceof\ C \Rightarrow \psi_{inv}) \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\}) \\
= & \text{[by definition of added invariant in } B_{RHS}\text{]} \\
 & (\bigwedge\{added\_inv^{U} \mid U \in ((supers(B_{RHS}) \setminus supers(A))\}) \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\}) \\
= & \text{[by definition of conjunction]} \\
 & \bigwedge\{added\_inv^{U} \mid U \in ((supers(B_{RHS}) \setminus supers(A)) \cup supers(A)\} \\
= & \text{[by set theory]} \\
 & \bigwedge\{added\_inv^{U} \mid U \in supers(B_{LHS})\} \\
= & \text{[by Definition 2]} \\
 & ext\_inv^{B}_{RHS}
\end{array}
$$

Figure 1: Proof of **Law 1** - case of object of exact type B

impacts caused by its internal specifications. These laws are based on programming laws – that do not consider specifications – from previous work [4, 6, 7].

Object-oriented programming laws were proposed by Borba *et al.* [4] for an object-oriented language called ROOL [5]. Cornélio [6] proves the laws with respect the copy semantics of ROOL [5] and formally justifies, by using programming laws and data refinement, refactoring practices documented by Fowler [8]. Silva, Sampaio, and Liu considers object-oriented programming laws in a language with a reference semantics [20], applying such laws to code refactoring. Duarte [7] adapts the programming laws initially written for ROOL for the Java programming and proposes other laws for language features that are not present in ROOL.

The laws presented here and the others of our catalog were used in [9] to show how a JML-specified version of a core module from a Manufacturing Execution System, get refactored from successive applications of primitive transformations expressed by means of our laws. Although our work does not provide a way to transform programs automatically yet, it provides a more reliable, rigorous and extensible alternative to address refactorings. Other works, as the one by Goldstein [10], which has tool support, apply non-systematic techniques to obtain behavior-preserving program transformations.

We have also applied our set of laws reducing a JML-specified Java program to a normal form [9] similar to the one presented by Duarte [7], which follows the main steps of the normal form reduction strategy of ROOL. A program in this normal form preserves the class hierarchy, but all attributes and methods that are non-recursive and with no mutually exclusive return points are located in the class

$$
\begin{aligned}
& ext\_spec_m^{B_{LHS}} \\
= & \text{ [by Definition 2]} \\
& \bigsqcup^B_{LHS} \{added\_spec_m^U \mid U \in supers(B)\} \\
= & \text{ [by set theory]} \\
& \bigsqcup^B_{LHS} \{added\_spec_m^U \mid U \in (supers(B) \setminus supers(A)) \cup supers(A)\} \\
= & \text{ [by definition of join with respect to } B\text{]} \\
& (\bigsqcup^B_{LHS} \{added\_spec_m^U \mid U \in (supers(B) \setminus supers(A))\}) \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= & \text{ [by definition of join of specification cases for } B_{LHS}\text{]} \\
& (\psi_1, \backslash old(\psi_1) \Rightarrow \psi_2) \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= & \text{ [by Propositional Logic]} \\
& ((\psi_1 \wedge true), \backslash old(\psi_1 \wedge true) \Rightarrow \psi_2) \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= & \text{ [by type test for object of type } B\text{]} \\
& ((\psi_1 \wedge \neg(this\ instanceof\ C)), \backslash old(\psi_1 \wedge \neg(this\ instanceof\ C)) \Rightarrow \psi_2) \\
& \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= & \text{ [by Propositional Logic]} \\
& ((\psi_1 \wedge \neg(this\ instanceof\ C)) \vee false, \backslash old(\psi_1 \wedge \neg(this\ instanceof\ C)) \Rightarrow \psi_2) \wedge true) \\
& \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= & \text{ [by type test for object of type } B \text{ and Propositional Logic]} \\
& ((\psi_1 \wedge \neg(this\ instanceof\ C)) \vee ((this\ instanceof\ C) \wedge \psi'_1), \\
& \quad (\backslash old(\psi_1 \wedge \neg(this\ instanceof\ C)) \Rightarrow \psi_2) \wedge (\backslash old((this\ instanceof\ C) \wedge \psi'_1) \Rightarrow \psi'_2)) \\
& \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= & \text{ [by definition of join of specification cases for } B_{RHS}\text{]} \\
& (\bigsqcup^B_{RHS} \{added\_spec_m^U \mid U \in (supers(B) \setminus supers(A))\}) \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= & \text{ [by definition of join with respect to } B\text{]} \\
& \bigsqcup^B_{RHS} \{added\_spec_m^U \mid U \in (supers(B) \setminus supers(A)) \cup supers(A)\} \\
= & \text{ [by set theory]} \\
& \bigsqcup^B_{RHS} \{added\_spec_m^U \mid U \in supers(B)\} \\
= & \text{ [by Definition 2]} \\
& ext\_spec_m^{B_{RHS}}
\end{aligned}
$$

Figure 2: Proof of **Law 3** - case of object of exact type B

Object. Also, invariants, initially-clauses and constraints are placed in the class Object. Specification cases of non-eliminated methods are written as JML assert statements.

Differently from laws that deal only with constructs of an object-oriented programming language, the presence of a behavioral interface specification language requires that we be aware of issues related, for instance, to the visibility of specification and code constructs, invariant preservation when introducing calls to super and changing a parameter type to a supertype requires introducing casts in occurrences of the parameter in specifications.

We have considered laws that address only a subset of the JML's Level 0 constructs [1], specially for lightweight specifications. Nevertheless, our preliminary focus is to cover most of the JML constructs that form the core notation used in the design by contract methodology. As future work, we intend to describe laws to support other JML clauses like **initially**, **constraint**, **represents**, and model fields. Also, we intend to work on proofs for the Java parts of the laws based on a reference semantics [2].

Acknowledgements

We are partially supported by the Brazilian Research Agency, CNPq, grant 484813/2007-2.

References

[1] G. T. Leavens et al. (2008): JML Reference Manual. Available at `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/`.

[2] A. Banerjee & D. A. Naumann (2005): Ownership confinement ensures representation independence for object-oriented programs. *J. ACM* 52(6), pp. 894–960.

[3] R. Bird & O. de Moor (1997): Algebra of Programming. Prentice Hall.

[4] P. Borba et al. (2004): Algebraic reasoning for object-oriented programming. *Sci. Comput. Program.* 52(1-3), pp. 53–100.

[5] A. L. C. Cavalcanti & D. A. Naumann (2000): A Weakest Precondition Semantics for Refinement of Object-oriented Programs . *IEEE Transactions on Software Engineering* 26(8), pp. 713–728.

[6] M. Cornélio (2004): Refactoring as Formal Refinements. Ph.D. thesis, Universidade Federal de Pernambuco.

[7] R. Duarte (2008): Parallelizing Java Programs Using Transformation Laws. Master's thesis, Universidade Federal de Pernambuco (UFPE).

[8] M. Fowler (1999): Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] G. F. Freitas, M. Cornélio & T. Massoni (2009): Transforming Java Annotated Programs. Technical Report, Universidade de Pernambuco, Departamento de Sistemas e Computação.

[10] M. Goldstein, Y. A. Feldman & S. Tyszberowicz (2006): Refactoring with Contracts. In: *AGILE '06: Proceedings of the conference on AGILE 2006*. IEEE Computer Society, Washington, DC, USA, pp. 53–64.

[11] J. Gosling, B. Joy, G. Steele & G. Bracha (2005): Java(TM) Language Specification, The. Addison-Wesley Professional, third edition.

[12] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey & B. A. Sufrin (1987): Laws of programming. *Commun. ACM* 30(8), pp. 672–686.

[13] G. T. Leavens (2006): JML's Rich, Inherited Specifications for Behavioral Subtypes. In: Zhiming Liu & Jifeng He, editors: *ICFEM*, *Lecture Notes in Computer Science* 4260. Springer, pp. 2–34.

[14] G. T. Leavens & Y. Cheon (2005): Design by Contract with JML. Available at `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`. Draft, available from jmlspecs.org.

[15] G. T. Leavens & D. A. Naumann (2006): Behavioral Subtyping, Specification Inheritance, and Modular Reasoning. Technical Report 06-20b, Department of Computer Science, Iowa State University.

[16] T. Massoni, R. Gheyi & P. Borba (2008): Formal Model-Driven Program Refactoring. In: José Luiz Fiadeiro & Paola Inverardi, editors: *FASE*, *Lecture Notes in Computer Science* 4961. Springer, pp. 362–376. Available at `http://dx.doi.org/10.1007/978-3-540-78743-3_27`.

[17] B. Meyer (1992): Applying design by contract. *IEEE Computer* 25, pp. 40–51.

[18] C. C. Morgan (1994): Programming from Specifications. Prentice Hall, second edition.

[19] S. Seres (2001): The Algebra of Logic Programming. Ph.D. thesis, Oxfor University Computing Laboratory.

[20] L. Silva, A. Sampaio & Z. Liu (2008): Laws of Object-Orientation with Reference Semantics. In: *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, Washington, DC, USA, pp. 217–226.

[21] R. R. Zagidullin & E. B. Frolov (2008): Control of manufacturing production by means of MES systems. *Russian Engineering Research* 28(2), pp. 166–168.

Automatic Generation of Proof Tactics for Finite-Valued Logics *

João Marcos
DIMAp / CCET
UFRN, Brazil
jmarcos@dimap.ufrn.br

Dalmo Mendonça
Undergraduate in Computer Engineering
UFRN, Brazil
dalmo3@gmail.com

A number of flexible tactic-based logical frameworks are nowadays available that can implement a wide range of mathematical theories using a common higher-order metalanguage. Used as proof assistants, one of the advantages of such powerful systems resides in their responsiveness to extensibility of their reasoning capabilities, being designed over rule-based programming languages that allow the user to build her own 'programs to construct proofs' — the so-called proof tactics.

The present contribution will discuss the implementation of an algorithm that generates sound and complete tableau systems for a very inclusive class of sufficiently expressive finite-valued propositional logics, and then illustrate some of the challenges and difficulties related to the algorithmic formation of automated theorem proving tactics for such logics. The procedure on whose implementation we will report is based on a generalized notion of analyticity of inference rules that is intended to guarantee termination of the corresponding automated tactics on what concerns theoremhood in our targeted logics.
Keywords: automated theorem proving, analyticity, rule-based programming, rewriting.

1 Introduction

The early history of the LCF family of theorem provers, first implemented as proof checkers by Robin Milner in the early 70s, based on Dana Scott's Logic for Computable Functions, can be said to be essentially based on Alonzo Church's proposal of a simple theory of types, developed three decades before (cf. [6]). Arguably, though, their great success as generic logical frameworks for the specification of a wide range of useful mathematical theories within a unified setting came in fact from later developments, namely: (1) the design of an accompanying powerful type-safe functional language that would allow for the needs of the theorem-proving community to be quite naturally expressed; (2) the decision to use a constructive higher-order logic as the underlying meta-language and to use higher-order unification as the underlying mechanism in which to specify diverse genera of inference systems as theories written in a common framework. The programming language that was designed in that process, `ML`, was intended to give support both to the expression of higher-order abstract syntax for the definition and manipulation of object-logics, and to advanced pattern-matching capabilities for the definition and manipulation of abstract high-level datatypes. From the point of view of theorem-proving, such flexible datatypes were to allow for the representation of useful objects such as *formulas*, *theorems* or even *proofs*, as well as some strategical operations over those objects, called *tactics*, that represented subgoaling strategies used in the construction of proofs. Higher-order operations for combining tactics and taking stricter control of the result of proof-search procedures were also available as the so-called *tacticals*. A modern heir of the LCF-style family of proof assistants and tactical provers, allowing for both interactive and automated reasoning, is the system `Isabelle` (cf. [5]), which will be utilized in what follows.

*Both authors acknowledge financial support by CNPq.

This is a preliminary version of a paper
that will appear in Electronic Proceedings
in Theoretical Computer Science.

A simple and elegant deductive formalism for the specification of proof procedures for both classical and non-classical logics is provided by the refutation-oriented method of *tableaux* (cf. [7]). In the classical bivalent propositional case, the inference rules of (signed or unsigned) tableau systems are based on adequate versions of a *subformula principle* that guarantees that the overall complexity of the involved formulas decreases as tableau rules are applied in the construction of a tableau derivation. The resulting collection of rules, in that case, is said to be *analytic*, and decidability, in general, follows from that. Indeed, analytical proof procedures eliminate the use of the so-called 'cut rule' (which often presupposes some ingenuity) and are very useful for automation as they greatly facilitate the finding of proofs. On the other hand, exactly because they eliminate cut, such procedures render the expression of proof lemmas more difficult, if not outright impossible. However, this expressive limitation can often be negotiated with an additional gain in the speed-up of the corresponding derivations if one considers systems allowing for the so-called 'analytic cuts' (cf. [3]). In one way or another, the objective is to define a rule-based framework for propositional logics in which the termination, with more or less efficiency, of a given theorem-proving task is guaranteed at the outset.

In [1] an algorithm was devised to extract bivalent (in general, non-truth-functional) characterizations for an extensive class of finite-valued propositional logics and then turn those characterizations into classic-like adequate tableau systems for those logics. We have used `ML` to implement that algorithm in [4],[1] and the output of our program is an `Isabelle` theory which can be used for computer-assisted proofs of theorems and derived rules of the corresponding finite-valued logics. Such proof systems, automatically extracted from the sets of truth-tables taken as input of our program, contained a non-eliminable version of the cut rule, and in fact no detailed proof was presented that analytic cuts would suffice for every proof system generated by the above mentioned algorithm. An improved axiom extraction algorithm has recently been proposed in [2], for the same class of logics, in which cut is an admissible rule. The latter algorithm has some interesting features, being based on a non-standard complexity measure that is intended to guarantee its analyticity. The present paper employs an illustration of the above procedure to briefly report on the challenges and difficulties related to the implementation of the mentioned novel algorithm, having again as output `Isabelle` theories, but this time extended with the automatic formation of proof tactics for the complete automation of the corresponding theorem-proving tasks.

2 Tableaux

A tableau system is both a proof and a counter-model building procedure based on the construction of refutation trees. A tableau rule is a schematic tree modifier, and its application allows us, given a branch in which we find an instance of the rule's heads, to extend the leaf of this branch by considering all the possibilities provided by the corresponding instances of the rules's daughters. For instance, the classical tableau rules for negation and implication can be represented as:

$$\begin{array}{cccc} F{:}(\neg\alpha) & T{:}(\neg\alpha) & F{:}(\alpha\to\beta) & T{:}(\alpha\to\beta) \\ | & | & | & \diagup\ \diagdown \\ T{:}\alpha & F{:}\alpha & T{:}\alpha & F{:}\alpha \quad T{:}\beta \\ & & F{:}\beta & \end{array} \tag{1}$$

This means, for instance, that a branch containing a signed formula of the form $F{:}(\alpha\to\beta)$ may be extended by adding in sequence new nodes of the form $T{:}\alpha$ and $F{:}\beta$. Similarly, a branch containing a signed formula of the form $T{:}(\alpha\to\beta)$ may be extended in two different ways, both by adding a new node of the form $F{:}\alpha$ and by adding a new node of the form $T{:}\beta$. The semantic reading of such rules

[1] Check also `http://tinyurl.com/5cakro`.

is obvious. The following *closure rule*, expressing an unobtainable semantic situation, completes the characterization of classical logic:

$$\begin{array}{c} T{:}\alpha \\ F{:}\alpha \\ | \\ * \end{array} \tag{2}$$

The rule is intended to say that a branch that contains an occurrence of the formula α labelled with the sign T and an occurrence of the same formula labelled with the sign F may be said to be *closed*. A whole tree is said to be closed if all of its branches are closed. Now, in case we want to verify the inference of a formula α from a set of premises $\gamma_1, \gamma_2, \ldots, \gamma_n$, using such 2-signed tableau rules for classical logic, what we do is to try and find a closed tableau tree starting from the linear sequence of labelled nodes $T{:}\gamma_1, T{:}\gamma_2, \ldots, T{:}\gamma_n, F{:}\alpha$.

The above tableau system for classical logic respects an obvious *subformula principle*, according to which each of the daughters of a non-closure rule are proper subformulas of some of the rule heads. It is easy to see that the following canonical *complexity measure* decreases with rule application:

$$\begin{array}{ll} (\ell 1) & \ell(p) = 0, \text{ where } p \text{ is an atom} \\ (\ell 2) & \ell(\neg\varphi_1) = \ell(\varphi_1) + 1 \\ (\ell 3) & \ell(\varphi_2 \to \varphi_3) = \ell(\varphi_2) + \ell(\varphi_3) + 1 \end{array} \tag{3}$$

Obviously, the closure rule is the only rule applicable to nodes with complexity zero. We say that a proof system is *analytical* if it only allows you to apply a rule when its daughters have smaller complexity than at least one of the corresponding heads. In other words, an analytical proof system is one to which a convenient proof strategy has been conveniently associated in such a way that complexity always decreases with rule application. This is obviously the case, without restriction, for each of the above rules for classical logic, applied in any particular order.

Analyticity guarantees *termination* of a proof procedure. We say that a tableau tree is terminated when: (T1) all of its branches are closed; (T2) there are open branches and no further rule is applicable. In case (T1) we may say the the initial inference has been successfully verified; in case (T2), the open branches allow us to extract all the counter-models to the initial inference.

3 Many-Valued Logics

Many-valued logics deviate from classical logic in allowing larger classes of truth-values, the so-called *designated* and *undesignated* values, to represent, respectively, 'degrees of truth' and 'degrees of falsity'. The rest remains pretty much the same, from the semantical point of view, so that for each assignment of truth-values to the atoms of a given m-ary formula φ there is a unique way of extending that into an interpretation $\widetilde{\varphi}$ of that formula as an m-ary operator over the extended algebra of truth-values.

An algorithm for obtaining analytic 2-signed tableau systems for finite-valued logics was described in [2], and we will illustrate it in what follows, for the instructive case of Łukasiewicz's four-valued logic Ł$_4$. This logic has 1 as its only designated value and $\frac{2}{3}$, $\frac{1}{3}$ and 0 as its undesignated values. Its connectives $\neg$ and $\to$ are interpreted as operators over $\{1, \frac{2}{3}, \frac{1}{3}, 0\}$ by way of the following definitions and their corresponding truth-tables:

$$\begin{array}{ll} (Ł_4\neg) & \widetilde{\neg} v = 1 - v \\ (Ł_4\to) & v_1 \widetilde{\to} v_2 = \mathsf{Min}(1, 1 - v_1 + v_2) \end{array} \tag{4}$$

Now, to produce a classic-like 2-signed tableau system for Ł$_4$ the idea is to associate to each truth-value of this logic a unique *binary print* in terms of the signs T and F that distinguishes this truth-value from any other truth-value. Given a collection of truth-values $\mathscr{V}$, its characteristic function $t : \mathscr{V} \to \{T,F\}$ is a mapping that associates T to designated values and F to undesignated values. Binary prints are sequences of unary formulas, called *separating formulas*, that use the latter characteristic functions to distinguish in between truth-values. In the case of Ł$_4$, the following separating formulas can be seen to do the job: $\theta_1(\varphi) = \neg\varphi$ and $\theta_2(\varphi) = \neg\neg(\varphi \to \neg\varphi)$. Consider indeed the table:

v	$t(v)$	$\widetilde{\theta}_1(v)$	$t(\widetilde{\theta}_1(v))$	$\widetilde{\theta}_2(v)$	$t(\widetilde{\theta}_2(v))$
0	F	1	T	1	T
$\frac{1}{3}$	F	$\frac{2}{3}$	F	1	T
$\frac{2}{3}$	F	$\frac{1}{3}$	F	$\frac{1}{3}$	F
1	T	0	F	0	F

(5)

Notice how each truth-value v is associated to a unique triple $\left\langle t(v), t(\widetilde{\theta}_1(v)), t(\widetilde{\theta}_2(v)) \right\rangle$.

All rules of the corresponding tableau system will have labelled binary prints as branches. For example, the rules corresponding to (Ł$_4\neg$) are:

$$\begin{array}{ccc} & F{:}\neg\alpha & \\ \swarrow & \downarrow & \searrow \\ F{:}\alpha & F{:}\alpha & T{:}\alpha \\ F{:}\theta_1(\alpha) & F{:}\theta_1(\alpha) & F{:}\theta_1(\alpha) \\ T{:}\theta_2(\alpha) & F{:}\theta_2(\alpha) & F{:}\theta_2(\alpha) \end{array} \qquad \begin{array}{c} T{:}\neg\alpha \\ | \\ F{:}\alpha \\ T{:}\theta_1(\alpha) \\ T{:}\theta_2(\alpha) \end{array} \tag{6}$$

An additional set of rules, with heads of the form $S{:}\theta_1(\varphi)$ and $S{:}\theta_2(\varphi)$, with $\varphi = \neg\alpha$ and $\varphi = \alpha \to \beta$, and $S \in \{T,F\}$, is needed to guarantee soundness and completeness of the 2-signed tableau system with respect to the initial finite-valued truth-tabular characterization of the target logic, Ł$_4$. Here are, by way of an illustration, the rules for $T{:}\theta_2(\alpha \to \beta)$ and $T{:}\theta_1(\neg\alpha)$:

$$\begin{array}{ccc} & T{:}\theta_2(\alpha \to \beta) & \\ \swarrow & \downarrow & \searrow \\ F{:}\alpha & T{:}\alpha & T{:}\alpha \\ F{:}\theta_1(\alpha) & F{:}\theta_1(\alpha) & F{:}\theta_1(\alpha) \\ F{:}\theta_2(\alpha) & F{:}\theta_2(\alpha) & F{:}\theta_2(\alpha) \\ F{:}\beta & F{:}\beta & F{:}\beta \\ T{:}\theta_1(\beta) & T{:}\theta_1(\beta) & F{:}\theta_1(\beta) \\ T{:}\theta_2(\beta) & T{:}\theta_2(\beta) & T{:}\theta_2(\beta) \end{array} \qquad \begin{array}{c} T{:}\theta_1(\neg\alpha) \\ | \\ T{:}\alpha \\ F{:}\theta_1(\alpha) \\ F{:}\theta_2(\alpha) \end{array} \tag{7}$$

Finally, the set of closure rules contains not only the classical rule (2), but also all other combinations of labelled binary prints that do *not* correspond to possible valuations, according to the truth-tables of Ł$_4$. In the case of this logic, the closure rules will be, then:

$$\begin{array}{cccc} F{:}\alpha & T{:}\alpha & T{:}\alpha & T{:}\alpha \\ T{:}\theta_1(\alpha) & T{:}\theta_1(\alpha) & F{:}\theta_1(\alpha) & F{:}\alpha \\ F{:}\theta_2(\alpha) & T{:}\theta_2(\alpha) & T{:}\theta_2(\alpha) & | \\ | & | & | & * \\ * & * & * & \end{array} \tag{8}$$

A closer look at the above four closure rules will reveal that the second and third rules, from left to right, only differ in signs for $\theta_1(\alpha)$. Obviously, $T{:}\theta_1(\alpha)$ and $F{:}\theta_1(\alpha)$ are the only two possible ways of labelling the formula $\theta_1(\alpha)$. Accordingly, those two rules should give place to a single rule:

$$\begin{array}{c} T{:}\alpha \\ T{:}\theta_2(\alpha) \\ | \\ * \end{array} \tag{9}$$

A similar approach can in fact be used to simplify other rules of the system, reducing the number of resulting branches and formulas (cf. [4]). Using that idea, for instance, the three branches of the rules $[F{:}\neg]$ and $[T{:}\theta_2\,{\to}]$, in the left halves of (6) and (7), could be simplified into just two branches.

Analyticity for the above system is ensured by enforcing a particular strategy that regulates rule applications based on an adequate non-canonical measure of complexity. To define that, all we have to do is to precede definition (3) by a further clause:

$$(\ell 0) \quad \ell(\theta(\varphi)) = \ell(\varphi), \text{ for every separating formula } \theta \tag{10}$$

Notice how different clauses of this novel definition of complexity may potentially apply to the same formula φ, if we look at it as a θ-formula or not. Notice that the new complexity measure is still well-defined as a function, once it is read from $(\ell 0)$ to $(\ell 3)$, in this order. On the other hand, even if we identify a given formula as a θ-formula, there might be, for instance, formulas φ_1 and φ_2 and separating formulas θ_1 and θ_2 such that $\theta_1(\varphi_1) = \varphi = \theta_2(\varphi_2)$. In that case, the rule to be applied should be the one that decreases the complexity the most, and this minimality requirement can also be conveniently internalized in the above definition of the complexity measure (check the details in [2]). For example, the signed formula $T{:}\neg\neg((\alpha \to \beta) \to \neg(\alpha \to \beta))$ might equally well be read as an instance of $T{:}\theta_1(\neg((\alpha \to \beta) \to \neg(\alpha \to \beta)))$ or an instance of $T{:}\theta_2(\alpha \to \beta)$. The three choices of reading would result in three different extensions of a tableau branch having the initial signed formula as one of its nodes. The first two choices are, according to the right halves of (6) and (7):

Rule $[T{:}\neg]$ is applied:	Rule $[T{:}\theta_1\neg]$ is applied:
$T{:}\neg\neg((\alpha \to \beta) \to \neg(\alpha \to \beta))$	$T{:}\theta_1(\neg((\alpha \to \beta) \to \neg(\alpha \to \beta)))$
$\vert$	$\vert$
$F{:}\neg((\alpha \to \beta) \to \neg(\alpha \to \beta))$	$T{:}((\alpha \to \beta) \to \neg(\alpha \to \beta))$
$T{:}\theta_1(\neg((\alpha \to \beta) \to \neg(\alpha \to \beta)))$	$F{:}\theta_1(((\alpha \to \beta) \to \neg(\alpha \to \beta)))$
$T{:}\theta_2(\neg((\alpha \to \beta) \to \neg(\alpha \to \beta)))$	$F{:}\theta_2(((\alpha \to \beta) \to \neg(\alpha \to \beta)))$

The third choice corresponds exactly to rule pictured at the left half of (7). Clearly, it is in this more 'concrete' case that the rule application results in less complex formulas. Our tableau strategy should take that into consideration. Just to illustrate the importance of this strategy, if we did not follow it in the example above, we could have chosen the first choice and then we would have observed that from the sequence of three resulting daughters, the second would be the head of the rule reiterated, and the third would be the more complex formula $T{:}\neg\neg((\neg((\alpha \to \beta) \to \neg(\alpha \to \beta))) \to \neg(\neg((\alpha \to \beta) \to \neg(\alpha \to \beta)))$. The tableau building procedure, in such a situation, would not necessarily be terminating.

4 Tactics

Our axiom extraction program takes as input the definition of a many-valued logic and generates a file with a theory ready to use with `Isabelle`. The theory includes the set of all tableau rules for the

object logic. In addition, taking advantage of the analytical character of the system defined by the new algorithm, rewrite rules and tactics for automated theorem proving are constructed.

In the ouput file for logic Ł$_4$, the rules for $F:\neg\alpha$, $T:\neg\alpha$, $T:\theta_2(\alpha\to\beta)$ and $T:\theta_1(\neg\alpha)$ exhibited at the last section are represented in `Isabelle`'s syntax by:

```
FNeg:
"[| [ $H, F:A0, F:t1(A0), T:t2(A0), $G ] ;
    [ $H, F:A0, F:t1(A0), F:t2(A0), $G ] ;
    [ $H, T:A0, F:t1(A0), F:t2(A0), $G ] |]
      ==> [ $H, F:~(A0), $G ]"

TNeg:
"[| [ $H, F:A0, T:t1(A0), T:t2(A0), $G ] |]
      ==> [ $H, T:~(A0), $G ]"

Tt1Neg:
"[| [ $H, T:A0, F:t1(A0), F:t2(A0), $G ] |]
      ==> [ $H, T:t1(~(A0)), $G ]"

Tt2Imp:
"[| [ $H, F:A0, F:t1(A0), F:t2(A0), F:A1, T:t1(A1), T:t2(A1), $G ] ;
    [ $H, T:A0, F:t1(A0), F:t2(A0), F:A1, T:t1(A1), T:t2(A1), $G ] ;
    [ $H, T:A0, F:t1(A0), F:t2(A0), F:A1, F:t1(A1), T:t2(A1), $G ] |]
      ==> [ $H, T:t2(A0 --> A1), $G ]"
```

In the above higher-order sequent-style syntax, where $ marks a context, the meta-implication ==> separates the branch representing the current goal at the right from its subgoals at the left. A closure rule such as the first one from (8), is represented as an axiom of the form:

```
CR1:    "[ $C1, F:A, $C2, T:t1(A), $C3, F:t2(A), $C4]"
```

We further add to the theory some convenient rewrite rules to allow the system to recognize given formulas as instances of separating formulas whenever possible. Only the outermost formulas may be instantiated as θ-formulas, as this rewrite is intended to be followed by a rule application, and there are no rules for formulas with nested θs.

```
t1_def:    "S:~A0          == S:t1(A0)"
t2_def:    "S:~~(A0-->~A0) == S:t2(A0)"
```

Again, to guarantee termination of proofs we must follow an order of instantiation, starting with the rewrite rules that reduce the most the complexity of the formula, namely θ_2. A tactic for ordered instantiation, in the case of Ł$_4$, may be defined by:

```
val auto_rw = (rewrite_goals_tac [t2_def]) THEN
              (rewrite_goals_tac [t1_def]);
```

where the command `rewrite_goals_tac [t2_def]` rewrites all formulas of the subgoal using the definition of `t2_def`, and similarly for `t1_def`. The tactical `THEN` makes sure that the second line will be executed only after the first one, and this strategy will guarantee the correct order of instantiation in the case where different θ-rules are applicable. Here is an illustration of the use of `auto_rw`:

```
 1. [F:~~(A-->~A), T:~~(A-->~B)]              (* Current state of proof *)
 2. [T:~~((A-->B)-->~(A-->B)), T:~A, F:~~A]

ML> by auto_rw;                               (* Using the tactic *)
 1. [F:t2(A), T:t1(~(A-->~B))]                (* New state of proof *)
 2. [T:t2(A-->B), T:t1(A), F:t1(~A)]
```

We may now use again the tacticals and construct a tactic for automatic theorem proving, by describing a procedure to exhaustively repeat, for every branch of the proof tree, the following steps:

1. instantiate formulas by rewriting (`auto_rw`), then
2. close the branch by applying one of the closure rules or
3. apply another rule of the system.

The first step will ensure that the right choice will be made when multiple rules are applicable to a formula. Next, the tactic tries to close the branch as soon as possible, to speed-up the process. If closure is not possible at that stage, the next step will try to apply another rule of the system, in the most convenient application order (for instance, postponing branching as much as possible), and start again. The procedure terminates, thanks to the analyticity of the system, and at the end either `Isabelle` will deliver a message of `No subgoals!`, meaning that the proof has been successfully concluded, or else there will be a list of subgoals — open branches — which are impossible to close and such that all their formulas have complexity zero, so that no further rule is applicable. From those open branches, as usual, counter-models can be built.

Extra details will be at hand to be surveyed by the interested reader as the full system is made available on-line, in open source.

References

[1] Carlos Caleiro, Walter Carnielli, Marcelo E. Coniglio & João Marcos (2005): *Two's company: "The humbug of many logical values"*. In: J.-Y. Béziau, editor: *Logica Universalis*. Birkhäuser Verlag, Basel, Switzerland, pp. 169–189. Preprint available at: `http://wslc.math.ist.utl.pt/ftp/pub/CaleiroC/05-CCCM-dyadic.pdf`.

[2] Carlos Caleiro & João Marcos (2009): *Classic-like analytic tableaux for finite-valued logics*. In: H. Ono, M. Kanazawa & R. de Queiroz, editors: *Proceedings of the* XVI *Workshop on Logic, Language, Information and Computation* (WoLLIC 2009), held in Tokyo, JP, June 2009, *Lecture Notes in Artificial Intelligence* 5514. Springer, pp. 268–280. Preprint available at: `http://wslc.math.ist.utl.pt/ftp/pub/CaleiroC/09-CM-ClATab4FVL.pdf`.

[3] Marcello D'Agostino & Marco Mondadori (1994): *The taming of the cut: classical refutations with analytic cut*. *Journal of Logic and Computation* 4(3), pp. 285–319.

[4] João Marcos & Dalmo Mendonça (2009): *Towards fully automated axiom extraction for finite-valued logics*. In: W. Carnielli, M. E. Coniglio & I. M. L. D'Ottaviano, editors: *The Many Sides of Logic*, Studies in Logic. College Publications, London. Preprint available at: `http://wslc.math.ist.utl.pt/ftp/pub/MarcosJ/08-MM-towards.pdf`.

[5] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, *LNCS* 2283. Springer.

[6] Lawrence C. Paulson (1987): *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press.

[7] Raymond M. Smullyan (1995): *First-Order Logic*. Dover.