Logic Mentoring Workshop - FLoC
Rewriting systems and applications

Maribel Fernández

July 2018
Motivations

Rewrite rules are an abstract way to specify transformations.

Ideal tool to model the dynamics of a system, analyse it, check properties.

Examples:

\[ \text{HCl} + \text{NaOH} \rightarrow \text{NaCl} + \text{H}_2\text{O} \]

\[ n + 0 \rightarrow n \]

\[ n + S(m) \rightarrow S(n + m) \]

\[
\begin{align*}
a\#P & \vdash P \land \forall[a]Q \rightarrow \forall[a](P \land Q) \\
a\#P & \vdash P \lor \exists[a]Q \rightarrow \exists[a](P \lor Q) \\
& \vdash \neg(\exists[a]Q) \rightarrow \forall[a]\neg Q \\
& \vdash \neg(\forall[a]Q) \rightarrow \exists[a]\neg Q
\end{align*}
\]
Rewriting systems as a modelling tool.

Expressivity:

- Rewriting is used to specify, in a uniform way, several computation paradigms (functional, logic, imperative and concurrent).
- ...
Motivations

Rewriting systems as a modelling tool.

Expressivity:

- Rewriting is used to specify, in a uniform way, several computation paradigms (functional, logic, imperative and concurrent).

- Rewriting is used to specify security features:
  - access control policies in various access control models (ACL, RBAC, CBAC), specified in a uniform and formal way.
  - analysis of cryptographic protocols
Motivations

Rewriting systems as a modelling tool.

Expressivity:

- Rewriting is used to specify, in a uniform way, several computation paradigms (functional, logic, imperative and concurrent).
- Rewriting is used to specify security features: access control policies in various access control models (ACL, RBAC, CBAC), specified in a uniform and formal way. analysis of cryptographic protocols
- Graph rewriting is used to specify complex systems in many areas: social networks, biochemical systems, interaction networks, software systems, etc.
Motivations

Why use rewriting systems as a modelling tool?

- A well-developed theory: rewriting techniques can be used to prove properties of the systems modelled (e.g., termination of a program, consistency of an access control policy, determinism, etc).
Motivations

Why use rewriting systems as a modelling tool?

• A well-developed theory: rewriting techniques can be used to prove properties of the systems modelled (e.g., termination of a program, consistency of an access control policy, determinism, etc).

• Availability of tools to test and experiment with evaluation strategies, to automate equational reasoning, and also for rapid prototyping: Maude, CiME, PORGY, GP, Kappa, . . .
There are several kinds of rewriting systems in the literature. My work has focused on:

- Term Rewriting (first and higher-order, and the “intermediate” Nominal Rewriting [FernandezGabbay2007])
There are several kinds of rewriting systems in the literature. My work has focused on:

- Term Rewriting (first and higher-order, and the “intermediate” Nominal Rewriting [FernandezGabbay2007])
- The Lambda Calculus (paradigmatic model of functional computation)
There are several kinds of rewriting systems in the literature. My work has focused on:

- Term Rewriting (first and higher-order, and the “intermediate” **Nominal Rewriting** [FernandezGabbay2007])
- The Lambda Calculus (paradigmatic model of functional computation)
- Graph Rewriting (interaction nets, portgraph rewriting cf. PORGY [CiE 2014])
There are several kinds of rewriting systems in the literature. My work has focused on:

- Term Rewriting (first and higher-order, and the “intermediate” Nominal Rewriting [FernandezGabbay2007])
- The Lambda Calculus (paradigmatic model of functional computation)
- Graph Rewriting (interaction nets, portgraph rewriting cf. PORGY [CiE 2014])
- Development of rewrite-based modelling languages, including strategy languages, joint work with the PORGY team.
Applications

• Programming languages (semantics, typing):
  Nominal rewriting: completion algorithms, type systems
  (polymorphism, dependent type systems – logical frameworks)

• Computation models, proof systems, resource analysis
  (linearity)

• Access control systems:
  Policy composition (modularity properties of rewriting
  systems), type systems to check consistency and totality

• Software systems (Theory of specifications, model
  transformations): Domain-Specific versions of PORGY
Inspired by nominal set theory (Fraenkel-Mostowski).
Key ideas: Freshness conditions $a\#t$, name swapping $(a \ b) \cdot t$.
Example: $\beta$ and $\eta$ rules

\[
app(lam([a]Z), Z') \rightarrow \ subst([a]Z, Z')
\]

\[
a\#M \vdash (\lambda([a]app(M, a)) \rightarrow M
\]

- Terms with binders
- Built-in $\alpha$-equivalence
- Efficient matching and unification algorithms
- Simple notion of substitution ("first-order"), non-capturing substitution has to be specified using rewrite rules.
- Dependencies of terms on names are implicit.
• Function symbols: \( f, g \ldots \)
Variables: \( M, N, X, Y, \ldots \)
Atoms: \( a, b, \ldots \)
Swappings: \((a\ b)\)
  Def. \((a\ b) a = b, (a\ b) b = a, (a\ b) c = c\)
Permutations: lists of swappings, denoted \( \pi \) \((\text{Id}\) empty).
• Nominal Terms:

\[
s, t ::= a \mid \pi \cdot X \mid [a]t \mid f\ t \mid (t_1, \ldots, t_n)
\]

\(\text{Id} \cdot X\) written as \(X\).
• Example (ML): \(\text{var}(a), \text{app}(t, t'), \text{lam}([a]t), \text{let}(t, [a]t'), \text{letrec}[f]([a]t, t'), \text{subst}([a]t, t')\)
Syntactic sugar:
\(a, (tt'), \lambda a.t, \text{let } a = t \text{ in } t', \text{letrec } f a = t \text{ in } t', t[a \mapsto t']\)
\(a \# X\) means \(a \not\in \text{fv}(X)\) when \(X\) is instantiated (avoiding name capture).

\[
\frac{\hspace{1em} \frac{a \approx_{\alpha} a}{\pi \cdot X \approx_{\alpha} \pi' \cdot X}}{\frac{\text{ds}(\pi, \pi') \# X}{(s_1, \ldots, s_n) \approx_{\alpha} (t_1, \ldots, t_n) \hspace{1em} s \approx_{\alpha} t}}
\]

\[
\frac{\frac{s_1 \approx_{\alpha} t_1 \cdots s_n \approx_{\alpha} t_n}{(s_1, \ldots, s_n) \approx_{\alpha} (t_1, \ldots, t_n)}}{\frac{fs \approx_{\alpha} ft}{s \approx_{\alpha} t \hspace{1em} a \# t \hspace{1em} s \approx_{\alpha} (a \ b) \cdot t}}
\]

\[
\frac{\frac{[a]s \approx_{\alpha} [a]t}{[a]s \approx_{\alpha} [b]t}}{\frac{[a]s \approx_{\alpha} [b]t}{[a]s \approx_{\alpha} [b]t}}
\]

where

\[
ds(\pi, \pi') = \{n | \pi(n) \neq \pi'(n)\}
\]

- \(a \# X, b \# X \vdash (a \ b) \cdot X \approx_{\alpha} X\)
\( a \# X \) means \( a \not\in \text{fv}(X) \) when \( X \) is instantiated (avoiding name capture).

\[
\begin{array}{c}
\text{\( a \approx_\alpha a \)}
\hline
\text{\( ds(\pi, \pi') \# X \)}
\end{array}
\begin{array}{c}
\text{\( \pi \cdot X \approx_\alpha \pi' \cdot X \)}
\end{array}
\]

\[
\begin{array}{c}
\text{\( s_1 \approx_\alpha t_1 \cdots s_n \approx_\alpha t_n \)}
\hline
\text{\( (s_1, \ldots, s_n) \approx_\alpha (t_1, \ldots, t_n) \)}
\end{array}
\begin{array}{c}
\text{\( s \approx_\alpha t \)}
\end{array}
\]

\[
\begin{array}{c}
\text{\( s \approx_\alpha t \)}
\hline
\text{\( a \# t \)}
\end{array}
\begin{array}{c}
\text{\( s \approx_\alpha (a \ b) \cdot t \)}
\end{array}
\]

\[
\begin{array}{c}
\text{\( [a]s \approx_\alpha [a]t \)}
\hline
\text{\( [a]s \approx_\alpha [b]t \)}
\end{array}
\]

where

\[
ds(\pi, \pi') = \{ n | \pi(n) \neq \pi'(n) \}\]

- \( a \# X, b \# X \vdash (a \ b) \cdot X \approx_\alpha X \)
- \( b \# X \vdash \lambda[a]X \approx_\alpha \lambda[b](a \ b) \cdot X \)
Also defined by induction:

\[
\begin{align*}
\frac{a \# b}{a \# [a]s}, & \quad \frac{\pi^{-1}(a) \# X}{a \# \pi \cdot X}, \quad \frac{a \# s_1 \cdots a \# s_n}{a \# (s_1, \ldots, s_n)}, \quad \frac{a \# s}{a \# fs}, \quad \frac{a \# s}{a \# [b]s}
\end{align*}
\]
Rewriting on nominal terms.

Applications:

- equational reasoning on data structures with binding;
- algebraic specifications of operators and data structures;
- operational semantics of programs;
- compilers, program transformations, etc.
Nominal Rewriting

Nominal Rewriting Rules:

\[ \Delta \vdash l \rightarrow r \quad V(r) \cup V(\Delta) \subseteq V(l) \]

Example: prenex normal forms in first-order logic

\[ a \# P \vdash P \land \forall[a]Q \rightarrow \forall[a](P \land Q) \]
\[ a \# P \vdash (\forall[a]Q) \land P \rightarrow \forall[a](Q \land P) \]
\[ a \# P \vdash P \land \exists[a]Q \rightarrow \exists[a](P \land Q) \]
\[ a \# P \vdash (\exists[a]Q) \land P \rightarrow \exists[a](Q \land P) \]
\[ \vdash \neg(\exists[a]Q) \rightarrow \forall[a]\neg Q \]
\[ \vdash \neg(\forall[a]Q) \rightarrow \exists[a]\neg Q \]

Reduction relation generated by (equivariant) nominal matching.
\[ l \approx_\alpha t \text{ where } V(l) \cap V(t) = \emptyset \text{ has solution } (\Delta, \theta) \text{ if } \]

\[ \Delta \vdash l \theta \approx_\alpha t \]

- Nominal matching is decidable [Urban, Pitts, Gabbay 2003]
- A solvable problem \( Pr \) has a unique most general solution: \((\Gamma, \theta)\) such that \( \Gamma \vdash Pr \theta \).
- Efficient algorithms: linear in time and space [Calves-F.2008]
- If rules are \textbf{closed}, nominal matching is sufficient (otherwise, equivariant nominal matching — NP [Cheney2004]).
Many techniques for first-order systems: well developed area.

Example: recursive path ordering [Dershowitz]

rpo: A well-founded precedence on function symbols generates a well-founded ordering on first-order terms.
Nominal rpo: [ICALP2012]
If an equational theory can be represented by a confluent and terminating rewrite system, then equational reasoning can be mechanised.

*Closed rewriting is sufficient to decide equality modulo a nominal equational theory* if the axioms can be oriented to form a confluent and terminating closed NRS [LFMTP2010].

Completion procedure [ICALP 2012]

To solve equations: *nominal narrowing* [FSCD2016] using nominal unification instead of matching.
AC nominal equality: AC matching and unification
[LSFA 2016, LOPSTR 2017]

Commutativity: *infinitary unification theory* if unifiers are represented with freshness contexts and substitutions, but *finitary* if using fixed point constraints and substitutions
[FSCD2018]
• Many sorted
• Polymorphic (Church/Curry styles) [TOCL 2018]
• Dependent types [TLCA 2015]
• Intersection types [TCS 2018]
Future Work

- Nominal Terms: extension of first-order syntax, efficient matching modulo $\alpha$.
- Clean semantics: Nominal Sets
- Equational reasoning and rewriting
  Completion as a tool to mechanise nominal equational logic
- Future work: functional abstraction and capture-avoiding substitution, implementing type checking algorithms, efficient AC-matching ...
Some (mentoring) conclusions:

Not really opposed:

- Theory vs Practice?
- Depth vs Breadth?
- Teaching vs Research?
- Diversity
- Positive approach... and team work