Invariant cost models for rewrite-based languages

Simone Martini

based on several joint papers with Ugo Dal Lago

Dipartimento di Scienze dell’Informazione
Alma mater studiorum • Università di Bologna

Linearity 2009, Coimbra – September 12, 2009
Dramatis personæ

- First order rewriting, FO
- Higher order rewriting: λ-calculus, λ
- Graph rewriting, GR

in the play

May I safely play your score?
FO  You know guys? / am able to simulate any of you.

λ  Of course you can. We are all Turing-universal. But / may simulate you more concisely. I am higher order.

GR  Come on! You are such a waste! You keep copying around subterms. / am more parsimonious of you all.

λ  Say the truth: To simulate me fully, you will need to duplicate, like me.
The question

*Is there a way in which our three characters can indeed simulate each other in a complexity sound and natural way?*

that is

- **sound**  polynomial
- **natural** the main cost parameter is naturally expressed in terms of the concepts the character itself understands
What we do not want

*Deus ex machina*

The Turing machine:

*I will simulate each of you in turn. If* *my* simulations are polynomially related in cost, then *you* all will be happy.
The question in full generality

What is a good cost model for a declarative, rule-based language, taking into account (only) the intrinsic description of that language, and not (also) its implementation on a conventional machine?

where

In the intrinsic description of a declarative language, the elementary computation step (e.g., resolution, $\beta$-reduction, firing of a rewrite rule, etc.) is not a constant-time operation.
For most such declarative languages, in their generality:

*We do not know.*

because the elementary computation step:

- not only *looks* non constant time
- but indeed *is* non constant (or even *non poly*) time
Our second character (almost): full $\lambda$-calculus

- **Terms**
  \[ M ::= x \mid \lambda x. M \mid MM \]

- **Reduction**
  \[
  (\lambda x. M) N \rightarrow M[N/x]
  \]

- **Terms may be duplicated during reduction**

- **Arbitrary size of terms during reduction**
Even with more compact reduction

- Lévy’s optimal reduction as graph reduction (à la Lamping)
- Have a notion of constant-time step

There are (simply typed) λ-terms which:
  - normalize in $k$ steps
  - require $\geq O(2^k)$ time on a TM

(Asperti and Mairson, POPL 1998; Asperti, Coppola and M., POPL 2000)
Restrict the calculus

- Linear $\lambda$-term
  Normalization is PTIME-complete
  The calculus has little expressivity

(Mairson, JFP 2004)

- Move to \textit{weak} reductions
  i.e., never reduce \textit{under} a $\lambda$:
  $\lambda x. M$ is always a normal form (in fact, a \textit{value})
The results, in general terms

- *Linear* simulations between
  - *Orthogonal constructor* term rewriting
  - *Weak λ-calculus*
  - *(Constructor) Term graph* rewriting

- each equipped with its *most natural, intrinsic cost* parameter,

- which is *polynomially related* to the actual cost of their normalization, as measured on a Turing machine

(Dal Lago and M., CiE 2006; ICALP 2009; and unpublished)
Part I

Term and Graph Rewriting
Given an orthogonal constructor rewrite system,

What is the relation between

- the derivational complexity of a term, i.e., the length of its derivation, and
- the time needed to rewrite it to normal form, on an efficient interpreter?

Answer: A polynomial relation, both under innermost and outermost reduction

Tool: a linear simulation of TR on GR.
Orthogonal constructor term rewriting

- Symbols, partitioned in constructors and functions
- Patterns: terms over constructors and variables

- Rules: \( f(p_1, \ldots, p_n) \rightarrow t \)
  - \( f \) is a function symbol; \( p_1, \ldots, p_n \) are patterns; \( t \) is a (general) term.

- Orthogonal: no rule overlapping; left-linear
- Innermost: the term substituted for variables in a firing do not contain any other redex
- Outermost: the term substituted for variables in a firing is not contained in any other redex
Term rewriting

- Strict separation between data (constructor terms) and programs (rules defined for functions)
- No critical pairs!

Given a term $t$, every innermost (outermost, respectively) reduction sequence leading $t$ to its normal form has the same length.

Well defined:
$Time_i(t)$
$Time_o(t)$
Term graph rewriting

- Represent a term $t$ with a graph $[t]_G$, fixing a root and allowing sharing

- $a(b(a(b(x), c)), b(a(b(x), c)))$

- Define a suitable “unsharing” of a graph, $\langle G \rangle_R$
Other terms graphs

\[ a(x, x) \quad b(x) \quad a(a(x, b(y)), b(a(x, b(y)))) \]
Constructor Term Graph Rewriting

- Fix a signature (with functions and constructors) labelling a graph
- In a pattern path \( v_1, \ldots, v_n \), \( \delta(v_i) \) is either a constructor symbol or is \( \bot \);
- In a left path, the first \( \delta(v_1) \) is a function symbol and \( v_2, \ldots, v_n \) is a pattern path.
Definition (Graph Rewrite Rules)

A graph rewrite rule over a signature $\Sigma$ is a triple $\rho = (G, r, s)$ such that:

- $G$ is a labelled graph;
- $r, s$ are vertices of $G$, called the left root and the right root of $\rho$, respectively.
- Any path starting in $r$ is a left path.
Represent rules with graph rewrite rules

\[ a(b(x), y) \rightarrow b(a(y, a(y, x))) \]
More examples:  $a$ is a function; $b, c, d$ are constructors.

$$a(b(d(b(x), c)), b(d(b(x), c))) \rightarrow x$$
Graph Rewrite Rules, 4

More examples: $a$ is a function; $b$, $c$, $d$ are constructors.

$$a(x, x) \rightarrow b(x) \hspace{2cm} a(b(x), b(y)) \rightarrow c$$
Applying a rule

Graph $G$ and rewriting rule $\rho = (H, r, s)$:

1. Locate a homomorphic copy of the “LHS” ($H \downarrow r$) of $\rho$ inside $G$
2. Add to $G$ a copy of the “RHS” of $\rho$
   ($H \downarrow s$ not contained in $H \downarrow r$)
Applying a rule, 2

2. Add to $G$ a copy of the “RHS” of $\rho$
3. Redirect the edges from the old to the new source
Applying a rule, 3

3. Redirect the edges from the old to the new root of the rule
4. Garbage collect the nodes unreachable from the root of the graph
4. Garbage collect the nodes unreachable from the root of the graph
Non overlapping

Definition
Two rules $\rho = (H, r, s)$ and $\sigma = (J, p, q)$ are overlapping iff there is a term graph $G$ and two homomorphism $\varphi$ and $\psi$ such that $(\rho, \varphi)$ and $(\sigma, \psi)$ are both redexes in $G$ with $\varphi(r) = \varphi(p)$.

Definition
A constructor graph rewrite system (CGRS) over a signature $\Sigma$ consists of a set of non-overlapping graph rewrite rules $\mathcal{G}$ on $\Sigma$. 
Lenght of reductions

- Theory of optimality: easy!
  recall: sharing, no overlapping

- **Outermost** reduction is the longest one

- A graph is redex-unshared iff there are no multiple paths from the root to a redex

- **Innermost** reduction preserves redex-unsharedness
Graph-reducing terms

Recall:

- Represent a term $t$ with a graph $[t]_G$, fixing a root and allowing sharing
- Define a suitable “unsharing” of a graph, $\langle G \rangle_{\mathcal{R}}$
- Reduction on graphs can be traced back to terms:

**Lemma**

If $G \rightarrow I$, then $\langle G \rangle_{\mathcal{R}} \rightarrow^+ \langle I \rangle_{\mathcal{R}}$. Moreover, if $G \rightarrow_i I$ and $G$ is redex-unshared, then $\langle G \rangle_{\mathcal{R}} \rightarrow \langle I \rangle_{\mathcal{R}}$. 
Graph reducibility

For every constructor rewrite system $\mathcal{R}$ over $\Sigma$ and for every term $t$ over $\Sigma$:

**Theorem (Outermost Graph-Reducibility)**

1. $t \rightarrow^n \sigma u$, where $u$ is in normal form; iff
2. $[t]_G \rightarrow^m G$, where $G$ is in normal form and $\langle G \rangle_{\mathcal{R}} = u$.

Moreover, $m \leq n$.

**Theorem (Innermost Graph Reducibility)**

1. $t \rightarrow^n \iota u$, where $u$ is in normal form; iff
2. $[t]_G \rightarrow^n G$, where $G$ is in normal form and $\langle G \rangle_{\mathcal{R}} = u$. 
Let \( t \) and \( G \) be such that \( [t]G \rightarrow^* G \).

Every graph rewriting step makes the graph bigger by at most the size of the rhs of a rewrite rule.

In \( [t]G \rightarrow^* G \rightarrow Ho \), \(|H| - |G| \leq k\); \( k \) depending on \( \mathcal{R} \) but not on \( t \)

\( [t]G \rightarrow^n Ho \) then \(|G| \leq nk + |t|\). Sharing!

If \( [t]G \rightarrow^n Ho \), computing a graph \( H \) such that \( G \rightarrow H \) takes polynomial time in \(|G|\), which is itself polynomially bounded by \( n \) and \(|t|\).
Complexity

Theorem

For every orthogonal, constructor term rewriting system $\mathcal{R}$, there is a polynomial $p : \mathbb{N}^2 \to \mathbb{N}$ such that for every term $t$ the normal form of $[t]_g$ can be computed in time at most $p(|t|, \text{Time}_o(M))$ when performing outermost graph reduction and in time $p(|t|, \text{Time}_i(M))$ when performing innermost graph reduction.

That is:
derivational complexity is a polynomially invariant cost model for orthogonal constructor term rewriting.
Part II

Term rewriting and λ-calculus
Our second character, revisited: weak call-by-value \(\lambda\)-calculus

- **Terms** \( M ::= x \mid \lambda x. M \mid MM \)

- **Values** \( V ::= x \mid \lambda x. M \)

- **Weak call-by-value reduction**

\[
\begin{align*}
(\lambda x. M)V & \rightarrow_v M[V/x] \\
ML & \rightarrow_v NL \\
LM & \rightarrow_v LN
\end{align*}
\]

- Values may be duplicated during reduction

- Is the number of reduction steps a good measure of actual cost?

  (Yes: Sands, Gustavsson, and Moran, 2002)
The result

From $\lambda$ to constructor rewriting:

$$M \rightarrow^n \nu N$$

$$[ ]\Phi \downarrow \quad \langle \rangle$$

$$[M]_\Phi \rightarrow^n t$$

From constructor rewriting to $\lambda$:

$$f(t_1, \ldots, t_h) \rightarrow^n u$$

$$[ ]\Lambda \quad \langle \rangle$$

$$[f]_\Lambda \langle \langle t_1 \rangle \rangle \ldots \langle \langle t_h \rangle \rangle \rightarrow^\kappa \nu \langle \langle u \rangle \rangle$$
The simulation cannot be obtained with Church-like encoding of data

- There is no constant-time predecessor on Church numerals
  
  Parigot, 1990

- Instead

\[
\begin{align*}
\text{Pred}(\text{succ}(n)) & \rightarrow^1 n \\
[pred] \land \langle \text{succ}(n) \rangle & \rightarrow^k \langle n \rangle
\end{align*}
\]
The simulation cannot be obtained with Church-like encoding of data

- There is no constant-time predecessor on Church numerals
  Parigot, 1990

- Instead

\[
\begin{align*}
\text{Pred}(\text{succ}(n)) & \rightarrow_{1} n \\
[pred] \land \langle \text{succ}(n) \rangle & \rightarrow_{k} \langle n \rangle
\end{align*}
\]
First simulation: From λ to constructor rewriting

Idea: full defunctionalization

Any λ-abstraction becomes a constructor

\[
\begin{align*}
[x]_{\Phi} &= x; \\
[\lambda x. M]_{\Phi} &= c_{x,M}(x_1, \ldots, x_n), \text{ where } FV(\lambda x. M) = x_1, \ldots, x_n; \\
[MN]_{\Phi} &= app([M]_{\Phi}, [N]_{\Phi}).
\end{align*}
\]

- Constructors: \(c_{x,M}\) for any \(M\) and any \(x\).
- Functions: \(app\).
- Reduction rules:

\[
app(c_{x,M}(x_1, \ldots, x_n), x) \rightarrow [M]_{\Phi}
\]
First simulation, 2

- In the other direction:

\[
\begin{align*}
\langle x \rangle_{\Lambda} &= x \\
\langle \text{app}(u, v) \rangle_{\Lambda} &= \langle u \rangle_{\Lambda} \langle v \rangle_{\Lambda} \\
\langle c_x, M(t_1, \ldots t_n) \rangle_{\Lambda} &= (\lambda x. M)[\langle t_1 \rangle_{\Lambda}/x_1, \ldots, \langle t_n \rangle_{\Lambda}/x_n]
\end{align*}
\]

- \(\langle [M]_{\Phi} \rangle_{\Lambda} = M\)

- For canonical \(t\), if \(t \rightarrow u\), then \(\langle t \rangle_{\Lambda} \rightarrow_{\nu} \langle u \rangle_{\Lambda}\)

**Theorem (Simulation)**

Let \(M\) be a closed \(\lambda\)-term. The following are equivalent:

1. \(M \rightarrow_{\nu}^n N\) where \(N\) is in normal form;
2. \([M]_{\Phi} \rightarrow_{\nu}^n t\) where \(\langle t \rangle_{\Lambda} = N\) and \(t\) is in normal form.
Second simulation:
From constructor rewriting to \( \lambda \)

**First:** Encode *data*, i.e. constructor terms

- Use Scott numerals-like encoding:

\[
\begin{align*}
0 & \equiv \lambda x_1. \lambda x_2. x_1 \\
n + 1 & \equiv \lambda x_1. \lambda x_2. n
\end{align*}
\]

- Here: \( \langle \cdot \rangle \wedge : \) constructor terms \( \rightarrow \) \( \lambda \)-terms
  
  For constructors \( c_1, \ldots, c_g \):

\[
\langle c_i(t_1 \ldots, t_n) \rangle \wedge \equiv \lambda x_1. \ldots. \lambda x_g. \lambda y. x_i \langle t_1 \rangle \wedge \ldots \langle t_n \rangle \wedge.
\]

- \( \perp \equiv \lambda x_1. \ldots. \lambda x_g. \lambda y. y \) denotes an error value
Second simulation, 2

Second: Encode *pattern matching*

- On an example:

  \[ f(p_1^1(x_1, x_2), p_2^1(x_3), p_3^1(x_4)) \rightarrow t_1 \]
  \[ f(p_1^2(x_5)), p_2^2(x_6, x_7), p_3^2(x_8)) \rightarrow t_2 \]

- Given such a sequence \( \alpha_1, \alpha_2 \) of patterns, construct a selector \( M_{\alpha_1, \alpha_2} \) s.t., for \( k \) depending only on \( \alpha_1, \alpha_2 \)
Second: Encode *pattern matching*

- On an example:

\[
f(p_1^1(x_1, x_2), p_2^1(x_3), p_3^1(x_4)) \rightarrow t_1
\]

\[
f(p_1^2(x_5), p_2^2(x_6, x_7), p_3^2(x_8)) \rightarrow t_2
\]

- Given such a sequence \( \alpha_1, \alpha_2 \) of patterns, construct a selector \( M_{\alpha_1, \alpha_2}^3 \) s.t., for \( k \) depending only on \( \alpha_1, \alpha_2 \)

\[
M_{\alpha_1, \alpha_2}^3 \langle \langle p_1^1(t_1, t_2) \rangle \rangle \wedge \langle \langle p_2^1(t_3) \rangle \rangle \wedge \langle \langle p_3^1(t_4) \rangle \rangle \wedge V_1 V_2
\rightarrow_k V_1 \langle \langle t_1 \rangle \rangle \wedge \cdots \langle \langle t_4 \rangle \rangle \wedge
\]
Second simulation, 2

Second: Encode *pattern matching*

- On an example:

  \[ f(p_1^1(x_1, x_2), p_2^1(x_3), p_3^1(x_4)) \rightarrow t_1 \]
  \[ f(p_1^2(x_5)), p_2^2(x_6, x_7), p_3^2(x_8)) \rightarrow t_2 \]

- Given such a sequence \( \alpha_1, \alpha_2 \) of patterns, construct a selector \( M_{\alpha_1, \alpha_2}^3 \) s.t., for \( k \) depending only on \( \alpha_1, \alpha_2 \)

  \[ M_{\alpha_1, \alpha_2}^3 \langle p_1^2(t_5) \rangle \land \langle p_2^2(t_6, t_7) \rangle \land \langle p_3^2(t_8) \rangle \land V_1 \land V_2 \]
  \[ \rightarrow_k^V V_2 \langle t_1 \rangle \land \ldots \langle t_4 \rangle \land \]
Second simulation, 2

Second: Encode *pattern matching*

- On an example:

\[
f(p_1^1(x_1, x_2), p_2^1(x_3), p_3^1(x_4)) \rightarrow t_1
\]

\[
f(p_1^2(x_5)), p_2^2(x_6, x_7), p_3^2(x_8)) \rightarrow t_2
\]

- Given such a sequence \(\alpha_1, \alpha_2\) of patterns, construct a selector \(M^3_{\alpha_1, \alpha_2}\) s.t., for \(k\) depending only on \(\alpha_1, \alpha_2\)

\[
M^3_{\alpha_1, \alpha_2} \begin{array}{cccccc}
X_1 & X_2 & X_3 & V_1 & V_2 \\
\rightarrow^k & \downarrow & \bot
\end{array}
\]

if any of the \(X_i\) does not match one of \(\alpha_1, \alpha_2\), or is \(\bot\).
Third: Solve *mutual recursion*

\[
\begin{align*}
  f_i(\alpha^1_i) & \to t^1_i \\
  \vdots & \\
  f_i(\alpha^n_i) & \to t^n_i.
\end{align*}
\]

*C-b-v fixpoint operators*

For any \( h \), there are \( H_1, \ldots, H_h \) and a bound \( m \), such that:

\[
H_i V_1 \ldots V_h \to^m_v V_i(\lambda x. H_1 V_1 \ldots V_h x) \ldots (\lambda x. H_h V_1 \ldots V_h x).
\]

\[
[f_i]_\wedge \equiv H_i V_1 \cdots (\lambda x. \lambda y. M_{\alpha^1_i, \ldots, \alpha^n_i} \overline{y}(\overline{\lambda z}(t_i^1)_\wedge) \ldots (\overline{\lambda z}(t_i^n)_\wedge)) \cdots V_h
\]
Second simulation, 4

Theorem: There is $k$ such that for any $f$

1. \[ f(t_1, \ldots, t_h) \xrightarrow{n} u \in C(\Xi) \]

2. \[ f(t_1, \ldots, t_h) \xrightarrow{n} u \not\in C(\Xi) \]

3. $f(t_1, \ldots, t_h)$ diverges, then $[f] \land \langle t_1 \rangle \ldots \langle t_h \rangle$ diverges.
Part III

Towards a conclusion
The results, in general terms

- **Linear** simulations between
  - *Orthogonal constructor* term rewriting
  - *Weak* $\lambda$-calculus
  - (Constructor) *Term graph* rewriting

- each equipped with its *most natural, intrinsic cost* parameter,

- which is *polynomially related* to the actual cost of their normalization, as measured on a Turing machine

  (Dal Lago and M., CiE 2006; ICALP 2009; and unpublished)
The context:

*Implicit* Computational Complexity

- A machine-free, logic-based investigation of the notion of feasible computation
- Feasibility through *language restrictions*, and not external measure conditions
- Incorporate computational complexity into formal methods in software development and programming language design
Implicit Computational Complexity

- **In the large**: study and characterize complexity classes e.g., Bellantoni-Cook; Girard's light logics; etc.

- **In the small**: study and relate machine-free models of computation *i.e.*, models with no notion of constant-time step
Implicit Computational Complexity

- **In the large**: study and characterize complexity classes e.g., Bellantoni-Cook; Girard’s light logics; etc.

- **In the small**: study and relate machine-free models of computation i.e., models with no notion of constant-time step