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?

The first lines of the script

FO You know guys? *I* am able to simulate any of you.

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

GR Come on! You are such a waste! You keep copying around subterms. *I* 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, β-reduction, firing of a rewrite rule, etc.) is not a constant-time operation.

An answer?

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 λ-calculus

- Terms $\qquad M ::= x \mid \lambda x.M \mid MM$

- Reduction

$$\frac{}{(\lambda x.M)N \to M\{N/x\}}$$

$$\frac{M \to N}{\lambda x.M \to \lambda x.N} \qquad \frac{M \to N \quad L \to P}{ML \to NP}$$

- Terms may be duplicated during reduction

- Arbitray 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 λ-term
  Normalization is PTIME-complete (Mairson, JFP 2004)
  The calculus has little expressivity
- Move to *weak* reductions
  i.e., never reduce *under* a λ:
  $\lambda x.M$ is always a normal form (in fact, a *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

Again: the question

- 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.

First character: Orthogonal constructor term rewriting

- Symbols, partioned in constructors and functions
- Patterns: terms over constructors and variables
- Rules: $\mathbf{f}(\mathbf{p}_1, \ldots, \mathbf{p}_n) \to_{\Xi} t$
  $\mathbf{f}$ is a function symbol; $\mathbf{p}_1, \ldots, \mathbf{p}_n$ 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

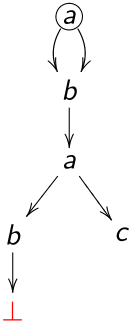
- Strict separation between data (constructor terms) and programs (rules defined for functions)
- No critical pairs!

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

Well defined:
$Time_i(t)$
$Time_o(t)$

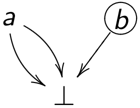
Third character: Term graph rewriting

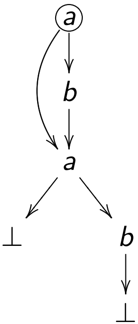
- Represent a term t with a graph $[t]_{\mathcal{G}}$, fixing a root and allowing sharing



- $a(b(a(b(x), c)), b(a(b(x), c)))$
- Define a suitable "unsharing" of a graph, $\langle G \rangle_{\mathcal{R}}$

Other terms graphs





$a(x, x) \quad b(x) \qquad\qquad a(a(x, b(y)), b(a(x, b(y))))$

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 $\perp$;
- In a *left path*, the first $\delta(v_1)$ is a function symbol and $v_2, \ldots, v_n$ is a pattern path.

Graph Rewrite Rules

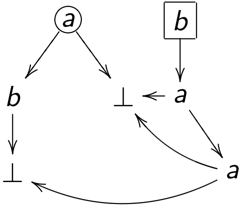
Definition (Graph Rewrite Rules)

A *graph rewrite rule* over a signature Σ 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 ρ, respectively.
- Any path starting in r is a left path.

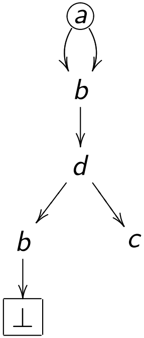
- Represent rules with graph rewrite rules

$$a(b(x), y) \to b(a(y, a(y, x)))$$



Graph Rewrite Rules, 3

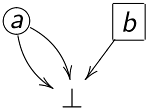
More examples: a is a function; b, c, d are constructors.

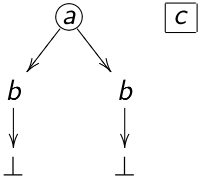


$$a(b(d(b(x), c)), b(d(b(x), c))) \longrightarrow x$$

Graph Rewrite Rules, 4

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

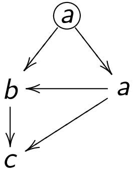




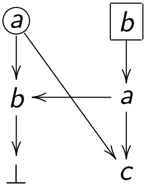
$a(x, x) \longrightarrow b(x)$ $\qquad$ $a(b(x), b(y)) \longrightarrow c$

Applying a rule

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



G

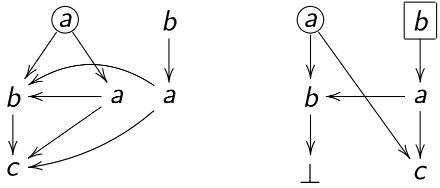


ρ

1. Locate a homomorphic copy of the "LHS" $(H \downarrow r)$ of ρ inside G
2. Add to G a copy of the "RHS" of ρ

$(H \downarrow s$ not contained in $H \downarrow r)$

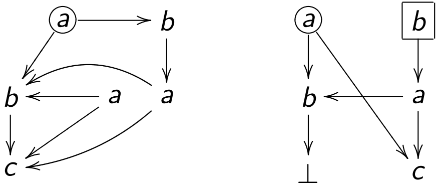
Applying a rule, 2



ρ

2. Add to G a copy of the “RHS” of ρ
3. Redirect the edges from the old to the new source

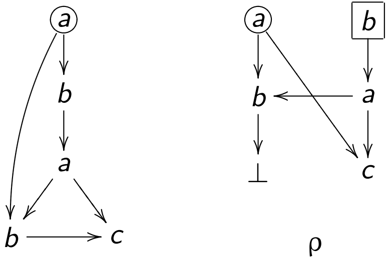
Applying a rule, 3



ρ

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

Applying a rule, 4



ρ

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 φ and ψ such that (ρ, φ) and (σ, ψ) are both redexes in G with $\varphi(r) = \varphi(p)$.

Definition

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

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 $[t]_{\mathcal{G}}$, 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 $G \to I$, then $\langle G\rangle_{\mathcal{R}} \to^{+} \langle I\rangle_{\mathcal{R}}$. Moreover, if $G \to_i I$ and G is redex-unshared, then $\langle G\rangle_{\mathcal{R}} \to \langle I\rangle_{\mathcal{R}}$.

Graph reducibility

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

Theorem (Outermost Graph-Reducibility)

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

Moreover, $m \leq n$.

Theorem (Innermost Graph Reducibility)

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

Complexity

- Let t and G be such that $[t]_{\mathcal{G}} \rightarrow_o^* G$.
- Every graph rewriting step makes the graph bigger by at most the size of the rhs of a rewrite rule.
  In $[t]_{\mathcal{G}} \rightarrow_o^* G \rightarrow_o H$, $|H| - |G| \leq k$; k depending on $\mathcal{R}$ but not on t
- $[t]_{\mathcal{G}} \rightarrow_o^n G$ then $|G| \leq nk + |t|$. Sharing!
- If $[t]_{\mathcal{G}} \rightarrow_o^n G$, computing a graph H such that $G \rightarrow H$ takes polynomial time in $|G|$, which is itself polynomially bounded by n and $|t|$.

Theorem

For every orthogonal, constructor term rewriting system $\mathcal{R}$, there is a polynomial $p : \mathbb{N}^2 \rightarrow \mathbb{N}$ such that for every term t the normal form of $[t]_{\mathcal{G}}$ can be computed in time at most $p(|t|, \mathcal{T}\!\mathit{ime}_o(M))$ when performing outermost graph reduction and in time $p(|t|, \mathcal{T}\!\mathit{ime}_i(M))$ 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 λ-calculus

- Terms $\quad M ::= x \mid \lambda x.M \mid MM$
- Values $\quad V ::= x \mid \lambda x.M$
- Weak call-by-value reduction

$$\frac{}{(\lambda x.M)V \to_v M\{V/x\}} \qquad \frac{M \to_v N}{ML \to_v NL} \qquad \frac{M \to_v N}{LM \to_v LN}$$

- 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 λ to constructor rewriting:

$$\begin{tikzcd}
M \arrow[r, "n"', "v"] \arrow[d, mapsto, "{[\ ]_\Phi}"'] & N \\
{[M]_\Phi} \arrow[r, "n"] & t \arrow[u, mapsto, "\langle\ \rangle"']
\end{tikzcd}$$

From constructor rewriting to λ:

$$\begin{tikzcd}
\mathbf{f}(t_1, \ldots, t_h) \arrow[r, "n"] \arrow[d, mapsto, "{[\ ]_\Lambda}"', "\langle\!\langle\ \rangle\!\rangle"] & u \arrow[d, mapsto] \\
{[\mathbf{f}]_\Lambda \langle\!\langle t_1 \rangle\!\rangle \ldots \langle\!\langle t_h \rangle\!\rangle} \arrow[r, "kn", "v"'] & \langle\!\langle u \rangle\!\rangle
\end{tikzcd}$$

Relevance: specific

- The simulation cannot be obtained with Church-like encoding of data
  - There is no constant-time predecessor on Church numerals

    Parigot, 1990

  - Instead

$$\begin{tikzcd}
Pred(succ(n)) \arrow[r, "1"] \arrow[d, mapsto] & n \arrow[d, mapsto] \\
[pred]_{\Lambda}\langle\langle succ(n)\rangle\rangle \arrow[r, "k", "v"'] & \langle\langle n\rangle\rangle
\end{tikzcd}$$

Relevance: specific

- The simulation cannot be obtained with Church-like encoding of data
  - There is no constant-time predecessor on Church numerals

    Parigot, 1990
  - Instead

$$
\begin{tikzcd}
\mathit{Pred}(\mathit{succ}(n)) \arrow[r, "1"] \arrow[d, mapsto] & n \arrow[d, mapsto] \\
{[\mathit{pred}]_{\Lambda}\langle\!\langle \mathit{succ}(n)\rangle\!\rangle} \arrow[r, "k", "v"'] & \langle\!\langle n\rangle\!\rangle
\end{tikzcd}
$$

First simulation: From λ to constructor rewriting

Idea: full defunctionalization

Any λ-abstraction becomes a constructor

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

- Constructors: $\mathbf{c}_{x,M}$ for any M and any x.
- Functions: $\mathbf{app}$.
- Reduction rules:

$$\mathbf{app}(\mathbf{c}_{x,M}(x_1,\ldots,x_n),x) \to [M]_\Phi$$

First simulation, 2

- In the other direction:

$$
\begin{aligned}
\langle x \rangle_\Lambda &= x \\
\langle \mathbf{app}(u, v) \rangle_\Lambda &= \langle u \rangle_\Lambda \langle v \rangle_\Lambda \\
\langle \mathbf{c}_{x,M}(t_1, \ldots t_n) \rangle_\Lambda &= (\lambda x.M)\{\langle t_1 \rangle_\Lambda / x_1, \ldots, \langle t_n \rangle_\Lambda / x_n\}
\end{aligned}
$$

- $\langle [M]_\Phi \rangle_\Lambda = M$
- For *canonical* t, if $t \to u$, then $\langle t \rangle_\Lambda \to_v \langle u \rangle_\Lambda$

Theorem (Simulation)

Let M be a closed λ-term. The following are equivalent:

1. *$M \to_v^n N$ where N is in normal form;*
2. *$[M]_\Phi \to^n t$ where $\langle t \rangle_\Lambda = N$ and t is in normal form.*

Second simulation: From constructor rewriting to λ

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

- Use Scott numerals-like encoding:

$$
\begin{aligned}
\underline{0} &\equiv \lambda x_1.\lambda x_2.x_1 \\
\underline{n+1} &\equiv \lambda x_1.\lambda x_2.\underline{n}
\end{aligned}
$$

- Here: $\langle\!\langle\ \rangle\!\rangle_\Lambda$: constructor terms $\longrightarrow$ λ-terms
  For constructors $\mathbf{c}_1, \ldots, \mathbf{c}_g$:

$$\langle\!\langle \mathbf{c}_i(t_1 \ldots, t_n) \rangle\!\rangle_\Lambda \equiv \lambda x_1.\ldots.\lambda x_g.\lambda y.x_i \langle\!\langle t_1 \rangle\!\rangle_\Lambda \ldots \langle\!\langle t_n \rangle\!\rangle_\Lambda.$$

- $\perp \equiv \lambda x_1.\ldots.\lambda x_g.\lambda y.y$ denotes an error value

Second simulation, 2

Second: Encode *pattern matching*

- On an example:

$$\begin{array}{rcl} f(\mathbf{p}^1_1(x_1,x_2),\mathbf{p}^1_2(x_3),\mathbf{p}^1_3(x_4)) & \longrightarrow & t_1 \\ f(\mathbf{p}^2_1(x_5)),\mathbf{p}^2_2(x_6,x_7),\mathbf{p}^2_3(x_8)) & \longrightarrow & t_2 \end{array}$$

- Given such a sequence α_1,α_2 of patterns, construct a selector $M^3_{\alpha_1,\alpha_2}$ s.t., for k depending only on α_1,α_2

Second simulation, 2

Second: Encode *pattern matching*

- On an example:

$$
\begin{array}{rcl}
f(\mathbf{p}_1^1(x_1, x_2), \mathbf{p}_2^1(x_3), \mathbf{p}_3^1(x_4)) & \rightarrow & t_1 \\
f(\mathbf{p}_1^2(x_5)), \mathbf{p}_2^2(x_6, x_7), \mathbf{p}_3^2(x_8)) & \rightarrow & t_2
\end{array}
$$

- Given such a sequence α_1, α_2 of patterns, construct a selector $M^3_{\alpha_1,\alpha_2}$ s.t., for k depending only on α_1, α_2

$$
\begin{aligned}
M^3_{\alpha_1,\alpha_2} \langle\!\langle \mathbf{p}_1^1(t_1, t_2) \rangle\!\rangle_\Lambda, \langle\!\langle \mathbf{p}_2^1(t_3) \rangle\!\rangle_\Lambda, \langle\!\langle \mathbf{p}_3^1(t_4) \rangle\!\rangle_\Lambda \; V_1 \; V_2 \\
\rightarrow_v^k \; V_1 \langle\!\langle t_1 \rangle\!\rangle_\Lambda \ldots \langle\!\langle t_4 \rangle\!\rangle_\Lambda
\end{aligned}
$$

Second simulation, 2

Second: Encode *pattern matching*

- On an example:

$$\begin{array}{rcl} f(\mathbf{p}_1^1(x_1, x_2), \mathbf{p}_2^1(x_3), \mathbf{p}_3^1(x_4)) & \rightarrow & t_1 \\ f(\mathbf{p}_1^2(x_5)), \mathbf{p}_2^2(x_6, x_7), \mathbf{p}_3^2(x_8)) & \rightarrow & t_2 \end{array}$$

- Given such a sequence α_1, α_2 of patterns, construct a selector $M^3_{\alpha_1,\alpha_2}$ s.t., for k depending only on α_1, α_2

$$\begin{array}{r} M^3_{\alpha_1,\alpha_2} \langle\langle \mathbf{p}_1^2(t_5) \rangle\rangle_\Lambda, \langle\langle \mathbf{p}_2^2(t_6, t_7) \rangle\rangle_\Lambda, \langle\langle \mathbf{p}_3^2(t_8) \rangle\rangle_\Lambda \; V_1 \; V_2 \\ \rightarrow_v^k \; V_2 \langle\langle t_1 \rangle\rangle_\Lambda \ldots \langle\langle t_4 \rangle\rangle_\Lambda \end{array}$$

Second simulation, 2

Second: Encode *pattern matching*

- On an example:

$$\begin{array}{rcl} f(\mathbf{p}_1^1(x_1, x_2), \mathbf{p}_2^1(x_3), \mathbf{p}_3^1(x_4)) & \to & t_1 \\ f(\mathbf{p}_1^2(x_5)), \mathbf{p}_2^2(x_6, x_7), \mathbf{p}_3^2(x_8)) & \to & t_2 \end{array}$$

- Given such a sequence α_1, α_2 of patterns, construct a selector $M^3_{\alpha_1,\alpha_2}$ s.t., for k depending only on α_1, α_2

$$\begin{array}{l} M^3_{\alpha_1,\alpha_2}\ X_1\ X_2\ X_3\ V_1\ V_2 \\ \qquad\qquad\qquad\quad \to_v^k\ \bot \end{array}$$

if any of the X_i does not match one of α_1, α_2, or is $\bot$.

Second simulation, 3

Third: Solve *mutual recursion*

$$\begin{array}{rcl} \mathbf{f}_i(\alpha_i^1) & \rightarrow & t_i^1 \\ & \vdots & \\ \mathbf{f}_i(\alpha_i^n) & \rightarrow & t_i^n. \end{array}$$

C-b-v fixpoint operators

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

$$H_i V_1 \ldots V_h \rightarrow_v^m V_i(\lambda x.H_1 V_1 \ldots V_h x) \ldots (\lambda x.H_h V_1 \ldots V_h x).$$

$$[\mathbf{f}_i]_\Lambda \equiv H_i\ V_1 \cdots (\overline{\lambda x}.\ \overline{\lambda y}.M_{\alpha_i^1,\ldots\alpha_i^n}\overline{y}(\overline{\lambda z}\langle\!| t_i^1 |\!\rangle_\Lambda) \ldots (\overline{\lambda z}\langle\!| t_i^n |\!\rangle_\Lambda)) \cdots V_h$$

Second simulation, 4

Theorem: There is k such that for any $\mathbf{f}$

1. $$\begin{tikzcd} \mathbf{f}(t_1,\ldots,t_h) \arrow[r, "n"] \arrow[d, mapsto, "{[\,]_\Lambda}"', "\langle\!\langle\ \rangle\!\rangle"] & u \in \mathcal{C}(\Xi) \arrow[d, mapsto] \\ {[\mathbf{f}]_\Lambda\langle\!\langle t_1\rangle\!\rangle \ldots \langle\!\langle t_h\rangle\!\rangle} \arrow[r, "kn"', "v"] & \langle\!\langle u\rangle\!\rangle \end{tikzcd}$$

2. $$\begin{tikzcd} \mathbf{f}(t_1,\ldots,t_h) \arrow[r, "n"] \arrow[d, mapsto, "{[\,]_\Lambda}"', "\langle\!\langle\ \rangle\!\rangle"] & u \notin \mathcal{C}(\Xi) \arrow[d, mapsto] \\ {[\mathbf{f}]_\Lambda\langle\!\langle t_1\rangle\!\rangle \ldots \langle\!\langle t_h\rangle\!\rangle} \arrow[r, "kn"', "v"] & \perp \end{tikzcd}$$

3. $\mathbf{f}(t_1,\ldots,t_h)$ diverges, then $[\mathbf{f}]_\Lambda\langle\!\langle t_1\rangle\!\rangle \ldots \langle\!\langle t_h\rangle\!\rangle$ diverges.

Part III

Towards a conclusion

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)

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 ligth 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 ligth logics; etc.

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