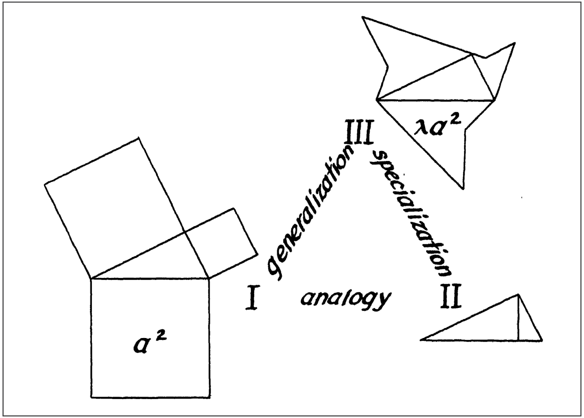
Accounting for the cost of substitution in structured rewriting

Vincent van Oostrom

Part I: Structured Rewriting

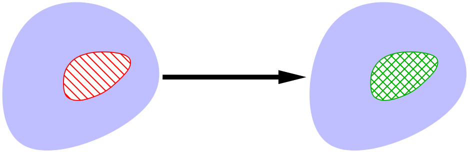
Part II: Implementation of 👀

Pólya’s triangle



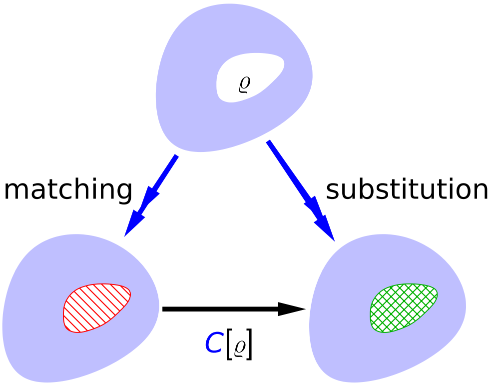
Mathematics and Plausible Reasoning, Volume1, 1954, Fig. 2.3

Pólya’s triangle in structured rewriting



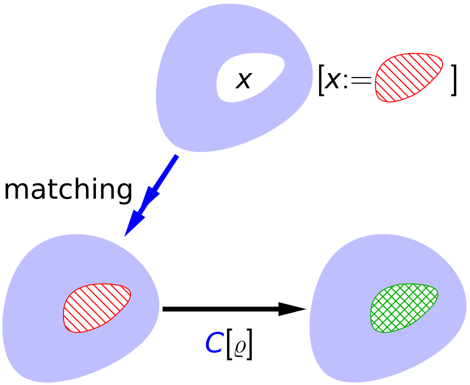
rewrite step $C[\ell]{\downarrow} \to C[r]{\downarrow}$ for rewrite rule $\varrho : \ell \to r$ and context C

Pólya's triangle in structured rewriting



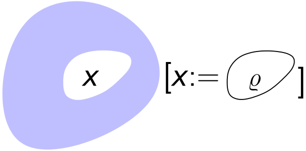
rewrite step $C[\varrho] : C[\ell]{\downarrow} \to C[r]{\downarrow}$ for rule $\varrho : \ell \to r$ and context C

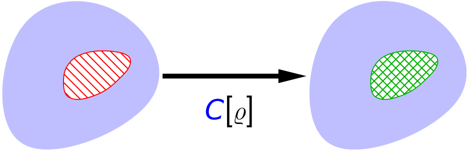
Pólya's triangle in structured rewriting



matching for rewrite step $C[\varrho] : C[\ell]{\downarrow} \to C[r]{\downarrow}$ for structure $C[x]$ and rule $\varrho : \ell \to r$

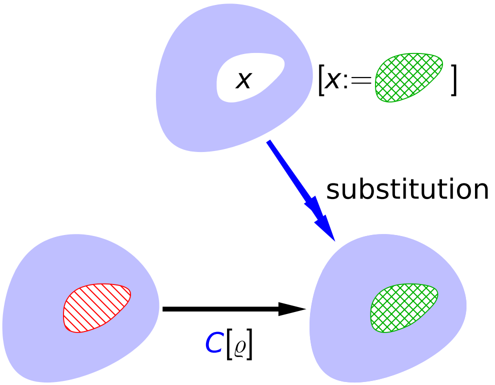
Pólya's triangle in structured rewriting





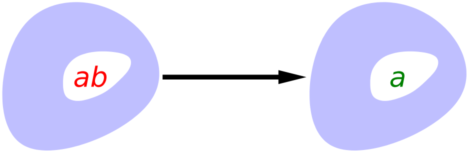
rewrite step $C[\varrho] : C[\ell]{\downarrow} \to C[r]{\downarrow}$ for structure $C[x]$ and rule $\varrho : \ell \to r$

Pólya's triangle in structured rewriting



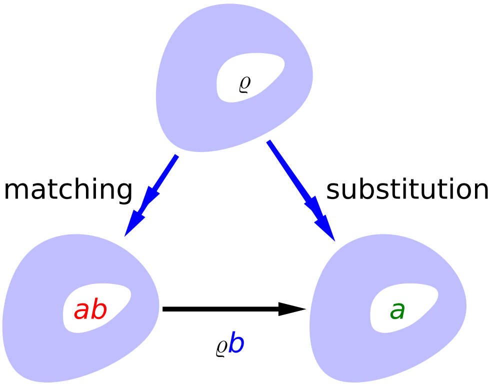
substitution for rewrite step $C[\varrho] : C[\ell]{\downarrow} \rightarrow C[r]{\downarrow}$ for structure $C[x]$ and rule $\varrho : \ell \rightarrow r$

Pólya's triangle in string rewriting



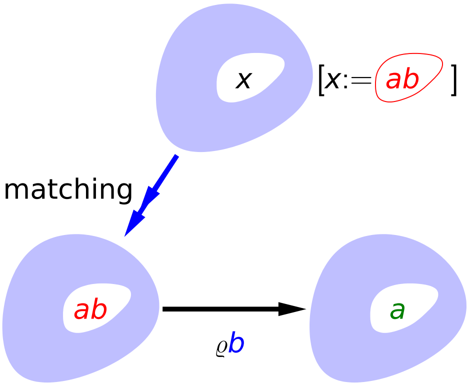
rewrite step $abb \to ab$ for rewrite rule $\varrho : ab \to b$ and context b

Pólya's triangle in string rewriting



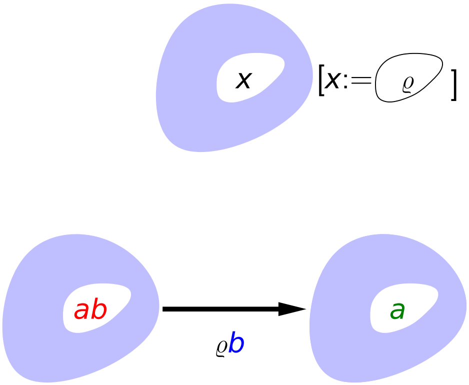
rewrite step $\varrho b : abb \to ab$ for rewrite rule $\varrho : ab \to b$ and context b

Pólya's triangle in string rewriting



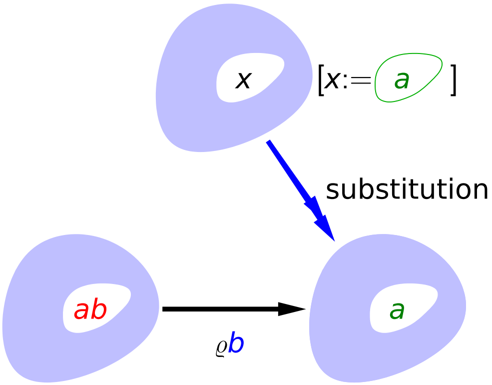
matching for rewrite step $\varrho b : abb \to ab$ for structure xb and rule $\varrho : ab \to b$

Pólya’s triangle in string rewriting



rewrite step $\varrho b : abb \to ab$ for structure xb and rule $\varrho : ab \to b$

Pólya's triangle in string rewriting



substitution for rewrite step $\varrho b : abb \to ab$ for structure xb and rule $\varrho : ab \to b$

Matching and substitution (may) come at a cost

Definition (of string)

string is an element of the free monoid A^* over alphabet A

poor: only interface to user à la abstract data types; no data structure

Matching and substitution (may) come at a cost

Definition (of Haskell strings)

string built from empty list Nil by consing (:) characters, *e.g.*, $a : (b : (b : \mathrm{Nil}))$

Matching and substitution (may) come at a cost

Example (of string rewrite step from a structured perspective)

step $\varrho b : abb \to ab$ by rewrite rule $\varrho : ab \to a$

but note $a : (b : \mathrm{Nil})$ does not occur in $a : (b : (b : \mathrm{Nil}))$

Matching and substitution (may) come at a cost

Example (of string rewrite step from a structured perspective)

step $\varrho b : abb \to ab$ by rewrite rule $\varrho : ab \to a$

how to apply $\varrho : a : (b : \mathrm{Nil}) \to a : \mathrm{Nil}$ for Haskell strings?

$$\begin{tikzcd}
 & \varrho : (b : \mathrm{Nil}) \arrow[dl, "\text{matching}"'] \arrow[dr, "\text{substitution}"] \arrow[d, phantom, "?"] & \\
a : (b : (b : \mathrm{Nil})) \arrow[rr] & {} & a : (b : \mathrm{Nil})
\end{tikzcd}$$

Matching and substitution (may) come at a cost

Example (of string rewrite step from a structured perspective)

step $\varrho b : abb \to ab$ by rewrite rule $\varrho : ab \to a$

Idea

all obtained by substituting for string variable x in $x +\!\!+ (b : \mathrm{Nil}) = x +\!\!+ b$

Matching and substitution (may) come at a cost

Example (of string rewrite step from a structured perspective)

step $\varrho b : abb \to ab$ by rewrite rule $\varrho : ab \to a$

- matching-phase decomposes $abb = a : (b : (b : \mathrm{Nil}))$ as substituting $a : (b : \mathrm{Nil})$ for x in $x ++ (b : \mathrm{Nil})$
- step $\varrho b = \varrho : (b : \mathrm{Nil})$ is formed by substituting $\varrho : \mathrm{Nil}$ for x in $x ++ (b : \mathrm{Nil})$
- substitution-phase yields $ab = a : (b : \mathrm{Nil})$ by substituting $a : \mathrm{Nil}$ for x in $x ++ (b : \mathrm{Nil})$

Matching and substitution (may) come at a cost

Example (of string rewrite step from a structured perspective)

step $\varrho b : abb \to ab$ by rewrite rule $\varrho : ab \to a$

- matching-phase decomposes $abb = a : (b : (b : \mathrm{Nil}))$ as substituting $a : (b : \mathrm{Nil})$ for x in $x +\!\!+ (b : \mathrm{Nil})$
- step $\varrho b = \varrho : (b : \mathrm{Nil})$ is formed by substituting $\varrho : \mathrm{Nil}$ for x in $x +\!\!+ (b : \mathrm{Nil})$
- substitution-phase yields $ab = a : (b : \mathrm{Nil})$ by substituting $a : \mathrm{Nil}$ for x in $x +\!\!+ (b : \mathrm{Nil})$

still easy but shows matching- and substitution-phases (may) come at a cost

Structured rewriting

Definition (of structured rewriting modulo substitution calculus)

- structures over a signature having variables $x, y, \ldots$ over structures
- substitution calculus $\rightarrow_{\mathcal{SC}}$ on structures; $\downarrow$ denotes $\mathcal{SC}$-normal form ($\mathcal{SC}$-nf)
- rules $\varrho : \ell \rightarrow r$ with ϱ in signature and ℓ, r structures
- contexts like $C[x]$, $D[x, y]$ indicating variable occurrences
- $C[s]$ denotes replacement of variable occurrence x by structure s in C

Structured rewriting

Definition (of structured rewriting modulo substitution calculus)

- structures over a signature having variables $x, y, \ldots$ over structures
- substitution calculus $\rightarrow_{\mathcal{SC}}$ on structures; $\downarrow$ denotes $\mathcal{SC}$-normal form ($\mathcal{SC}$-nf)
- rules $\varrho : \ell \rightarrow r$ with ϱ in signature and ℓ, r structures
- contexts like $C[x]$, $D[x, y]$ indicating variable occurrences
- $C[s]$ denotes replacement of variable occurrence x by structure s in C

Definition (of structured rewrite step)

step $C[\varrho] : s \rightarrow t$, for context C and structures s, t in $\mathcal{SC}$-nf and rule $\varrho : \ell \rightarrow r$ if

$$s = C[\ell]{\downarrow}\ {}_{\mathcal{SC}}\!\twoheadleftarrow C[\ell] \rightarrow_{\varrho} C[r] \twoheadrightarrow_{\mathcal{SC}} C[r]{\downarrow} = t$$

Structured rewriting

$$\begin{tikzcd}
C[\ell] \arrow[r, "\text{replacement}"] \arrow[d, "\text{matching}"'] & C[r] \arrow[d, "\text{substitution}"] \\
t \arrow[r, "\text{step}"'] & s
\end{tikzcd}$$

Definition (of structured rewrite step)

step $C[\varrho] : s \to t$, for context C and structures s, t in $\mathcal{SC}$-nf and rule $\varrho : \ell \to r$ if

$$s = C[\ell]{\downarrow} \; {}_{\mathcal{SC}}{\twoheadleftarrow} \; C[\ell] \to_{\varrho} C[r] \twoheadrightarrow_{\mathcal{SC}} C[r]{\downarrow} = t$$

Structured rewriting

$$
\begin{tikzcd}
C[\ell_1,\dots,\ell_n] \arrow[r, "\text{replacement}"] \arrow[d, "\text{matching}"'] & C[r_1,\dots,r_n] \arrow[d, "\text{substitution}"] \\
t \arrow[r, "\circ", "\text{multistep}"'] & s
\end{tikzcd}
$$

Definition (of structured rewrite multistep)

multistep $C[\vec{\varrho}] : s \mathrel{-\!\!\circ\!\!\to} t$, for context C, structures s, t in $\mathcal{SC}$-nf, rules $\varrho_i : \ell_i \to r_i$ if

$$s = C[\ell_1,\dots,\ell_n]{\downarrow}\ {}_{\mathcal{SC}}{\twoheadleftarrow}\ C[\ell_1,\dots,\ell_n] \mathrel{-\!\!\circ\!\!\to}_{\vec{\varrho}} C[r_1,\dots,r_n] \twoheadrightarrow_{\mathcal{SC}} C[r_1,\dots,r_n]{\downarrow} = t$$

Structured rewriting

$$\begin{tikzcd}
C[\ell] \arrow[r, "\text{replacement}"] \arrow[d, "\text{matching}"'] & C[r] \arrow[d, "\text{substitution}"] \\
t \arrow[r, "\text{step}"'] & s
\end{tikzcd}$$

Definition (of cost of step)

cost of step $C[\varrho] : t \to s$ is sum of costs of matching, replacement, substitution

Higher-order terms as structures

Instance

- structures: simply typed λ-terms over a simply typed signature, in long η-nf
- substitution calculus: $\mathcal{SC}$ is β modulo α

- notation: we write $x.t$ for abstraction in the $\mathcal{SC}$ (no λ)
- 1–1 correspondence between structured rewrite steps and usual HRS-steps

Higher-order terms as structures

Instance

- structures: simply typed λ-terms over a simply typed signature, in long η-nf
- substitution calculus: $\mathcal{SC}$ is β modulo α

Example (of HRS step and its cost)

- rule $\varrho : x.f(x) \to x.d(x,x)$ over obvious signature (in fact first-order)

Higher-order terms as structures

Instance

- structures: simply typed λ-terms over a simply typed signature, in long η-nf
- substitution calculus: $\mathcal{SC}$ is β modulo α

Example (of HRS step and its cost)

- rule $\varrho : x.f(x) \to x.d(x,x)$ over obvious signature
- step $f(\varrho(f(a))) : f(f(f(a))) \to f(d(f(a),f(a)))$ (substituting for X in $f(X(f(a))))$

Higher-order terms as structures

Instance

- structures: simply typed λ-terms over a simply typed signature, in long η-nf
- substitution calculus: $\mathcal{SC}$ is β modulo α

$$\begin{tikzcd}
f((x.f(x))\,f(a)) \arrow[r, "\text{replacement}"] \arrow[d, "\text{matching}"'] & f((x.d(x,x))\,f(a)) \arrow[d, "\text{substitution}"] \\
f(f(f(a))) \arrow[r, "f(\varrho(f(a)))"'] & f(d(f(a),f(a)))
\end{tikzcd}$$

Higher-order terms as structures

Instance

- structures: simply typed λ-terms over a simply typed signature, in long η-nf
- substitution calculus: $\mathcal{SC}$ is β modulo α

Example (of HRS step and its cost)

- rule $\varrho : x.f(x) \to x.d(x, x)$ over obvious signature
- step $f(\varrho(f(a))) : f(f(f(a))) \to f(d(f(a), f(a)))$
- $\mathcal{SC}$-cost should take replication (in rhs) into account; choose algebra: n-ary function symbol by $(n_1, \ldots, n_n) \mapsto 1 + \sum_i n_i$ and ϱ by $n \mapsto 1 + 2n$

Higher-order terms as structures

Instance

- structures: simply typed λ-terms over a simply typed signature, in long η-nf
- substitution calculus: $\mathcal{SC}$ is β modulo α

Example (of HRS step and its cost)

- rule $\varrho : x.f(x) \to x.d(x,x)$ over obvious signature
- step $f(\varrho(f(a))) : f(f(f(a))) \to f(d(f(a), f(a)))$
- n-ary function symbol by $(n_1, \ldots, n_n) \mapsto 1 + \sum_i n_i$ and ϱ by $n \mapsto 1 + 2n$
  cost of step $f(\varrho(f(a)))$ is $1 + (1 + 2(1 + 1)) = 6$

Higher-order terms as structures

Instance

- structures: simply typed λ-terms over a simply typed signature, in long η-nf
- substitution calculus: $\mathcal{SC}$ is β modulo α

Example (of HRS step and its cost)

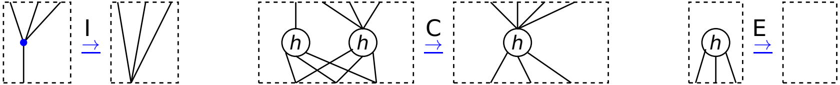
- rule $\varrho : x.f(x) \to x.d(x,x)$ over obvious signature
- step $f(\varrho(f(a))) : f(f(f(a))) \to f(d(f(a), f(a)))$
- n-ary function symbol by $(n_1, \ldots, n_n) \mapsto 1 + \sum_i n_i$ and ϱ by $n \mapsto 1 + 2n$
  cost of step $f(\varrho(f(a)))$ is $1 + (1 + 2(1 + 1)) = 6$

reversing rule $\varrho^{-1} : x.d(x,x) \to x.f(x)$, step $f(\varrho^{-1}(f(a)))$, same cost analysis

Termgraphs as structures

Instance

- structures: rooted dags over a signature extended with indirection •
- substitution calculus: the ж-calculus



Termgraphs as structures

Instance

- structures: rooted dags over a signature extended with indirection $\bullet$
- substitution calculus: the ж-calculus

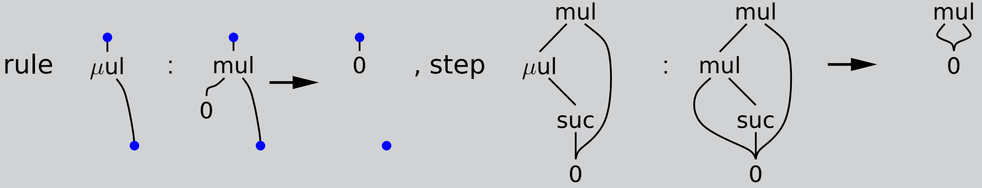
- ж-calculus has implicit garbage collection
- termgraphs in ж-normal form are maximally shared

Termgraphs as structures

Instance

- structures: rooted dags over a signature extended with indirection •
- substitution calculus: the ж-calculus

Example (of termgraph step modulo ж)

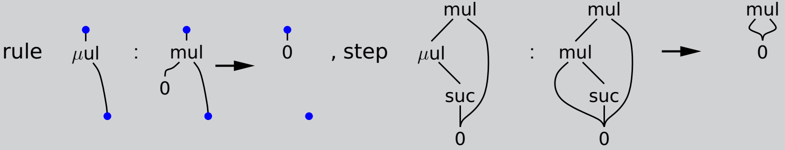


Termgraphs as structures

Instance

- structures: rooted dags over a signature extended with indirection •
- substitution calculus: the ж-calculus

Example (of termgraph step modulo ж)



cost: substitution may knock-on erasures and sharing (bounded by graph size)

Substitution Calculi

Example (of some substitution calculi)

- (higher-order) term rewriting: simply typed $\lambda\alpha\beta\eta$-calculus

Substitution Calculi

Example (of some substitution calculi)

- (higher-order) term rewriting: simply typed $\lambda\alpha\beta\eta$-calculus
- termgraph rewiting: the ж-calculus

Substitution Calculi

Example (of some substitution calculi)

- (higher-order) term rewriting: simply typed $\lambda\alpha\beta\eta$-calculus
- termgraph rewiting: the ж-calculus
- interaction nets: indirection-calculus $-\!\!\bullet\!\!- \;\rightarrow\; ——$

Substitution Calculi

Example (of some substitution calculi)

- (higher-order) term rewriting: simply typed $\lambda\alpha\beta\eta$-calculus
- termgraph rewiting: the ж-calculus
- interaction nets: indirection-calculus $-\!\bullet\!- \;\to\; —$
- net rewriting: proofnet-calculus (PN)

Substitution Calculi

Example (of some substitution calculi)

- (higher-order) term rewriting: simply typed $\lambda\alpha\beta\eta$-calculus
- termgraph rewiting: the ж-calculus
- interaction nets: indirection-calculus $-\bullet- \to -$
- net rewriting: proofnet-calculus (PN)
- term rewriting: linear substitution calculus (LSC)?

Substitution Calculi

Example (of some substitution calculi)

- (higher-order) term rewriting: simply typed $\lambda\alpha\beta\eta$-calculus
- termgraph rewiting: the ж-calculus
- interaction nets: indirection-calculus $-\!\bullet\!- \;\to\; -\!-$
- net rewriting: proofnet-calculus (PN)
- term rewriting: linear substitution calculus (LSC)?
- sharing graph rewriting: deep inference?

Substitution Calculi

Example (of some substitution calculi)

- (higher-order) term rewriting: simply typed $\lambda\alpha\beta\eta$-calculus
- termgraph rewiting: the ж-calculus
- interaction nets: indirection-calculus $-\!\bullet\!- \rightarrow -\!\!-$
- net rewriting: proofnet-calculus (PN)
- term rewriting: linear substitution calculus (LSC)?
- sharing graph rewriting: deep inference?
- sub-calculi and strategies for $\lambda\beta$: machines?

Substitution Calculi

Example (of some substitution calculi)

- (higher-order) term rewriting: simply typed $\lambda\alpha\beta\eta$-calculus
- termgraph rewiting: the ж-calculus
- interaction nets: indirection-calculus $-\!\bullet\!- \;\to\; -\!-$
- net rewriting: proofnet-calculus (PN)
- term rewriting: linear substitution calculus (LSC)?
- sharing graph rewriting: deep inference?
- sub-calculi and strategies for $\lambda\beta$: machines?
- ...

Axioms on substitution calculi (SC)

Axioms

A1 the SC is complete (confluent and terminating)

A2 the SC is only needed for gluing (rules are closed)

A3 multisteps can be sequentialised / serialised (some development)

Axioms on substitution calculi (SC)

Axioms

A1 the SC is complete (confluent and terminating)

A2 the SC is only needed for gluing (rules are closed)

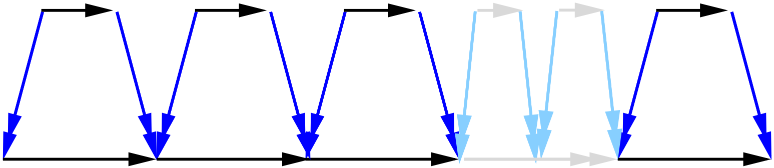
A3 multisteps can be sequentialised / serialised (some development)

$$
\begin{tikzcd}
C[\ell_0,\vec{\ell}] \arrow[rrr, "{C[\varrho_0,\vec{\ell}]}"'] \arrow[dd] \arrow[dr] & & & C[r_0,\vec{\ell}] \arrow[rrr, "{C[r_0,\vec{r}]}"'] \arrow[dl] \arrow[dd] \arrow[dr] & & & C[r_0,\vec{r}] \arrow[dl] \arrow[dd] \\
& D[\ell_0] \arrow[r, "{D[\varrho_0]}"'] \arrow[dl] & D[r_0] \arrow[dr] & & E[\vec{\ell}] \arrow[r, "{E[\vec{\varrho}]}"'] \arrow[dl] \arrow[dr, phantom, "\mathrm{IH}"] & E[\vec{r}] \arrow[dr] & \\
t_0 \arrow[rrr, "\mathcal{T}"'] & & & t_1 \arrow[rrr, two heads, "\mathcal{T}"'] & & & t_n
\end{tikzcd}
$$

Exploiting substitution calculi to redistribute costs

Cost-saving observations

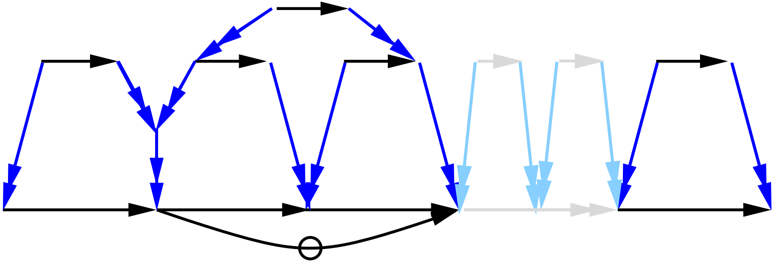
- should avoid substitution and subsequent matching inverse to each other
- matching more lhss simultaneously (multisteps) enables parallelism
- by not going to $\mathcal{SC}$-normal forms we may sometimes eliminate matching



Exploiting substitution calculi to redistribute costs

Cost-saving observations

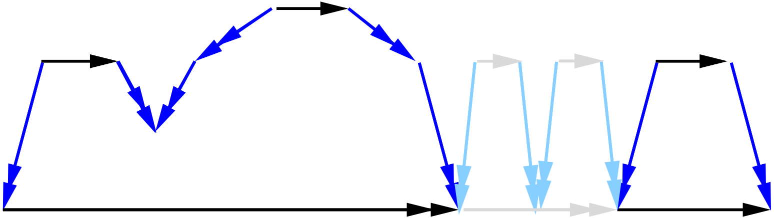
- should avoid substitution and subsequent matching inverse to each other
- matching more lhss simultaneously (multisteps) enables parallelism
- by not going to $\mathcal{SC}$-normal forms we may sometimes eliminate matching



Exploiting substitution calculi to redistribute costs

Cost-saving observations

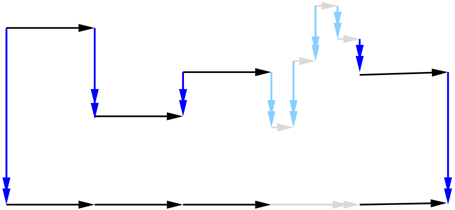
- should avoid substitution and subsequent matching inverse to each other
- matching more lhss simultaneously (multisteps) enables parallelism
- by not going to $\mathcal{SC}$-normal forms we may sometimes eliminate matching



Exploiting substitution calculi to redistribute costs

Cost-saving observations

- should avoid substitution and subsequent matching inverse to each other
- matching more lhss simultaneously (multisteps) enables parallelism
- by not going to $\mathcal{SC}$-normal forms we may sometimes eliminate matching

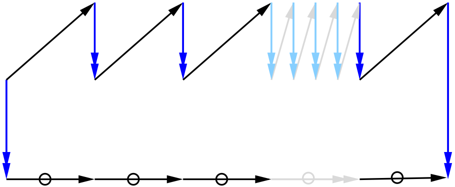


deterministic rewrite system $\Longrightarrow$ dipper SC

Exploiting substitution calculi to redistribute costs

Cost-saving observations

- should avoid substitution and subsequent matching inverse to each other
- matching more lhss simultaneously (multisteps) enables parallelism
- by not going to $\mathcal{SC}$-normal forms we may sometimes eliminate matching

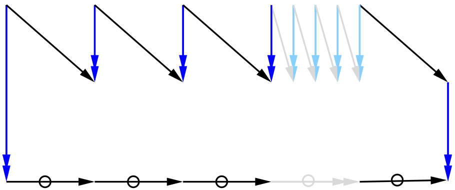


matching trivial $\Longrightarrow$ cascading SC

Exploiting substitution calculi to redistribute costs

Cost-saving observations

- should avoid substitution and subsequent matching inverse to each other
- matching more lhss simultaneously (multisteps) enables parallelism
- by not going to $\mathcal{SC}$-normal forms we may sometimes eliminate matching



substitution trivial $\Longrightarrow$ stepping SC

Dipper SC Example: leftmost string rewriting

Dipper idea

record current location in string (state) by splitting it into three strings: to the left, matched, to the right

since costly to scan from the start after each step

Dipper SC Example: leftmost string rewriting

Example

for rule $\varrho : ab \to a$ and string *aaba*

$$(\varepsilon, \varepsilon, babb) \leftarrow (b, \varepsilon, abb) \leftarrow (b, a, bb) \leftarrow (b, ab, b) \leftarrow (b, \varrho, b) \to$$

$$(b, a, b) \to (b, ab, \varepsilon) \to (b, \varrho, \varepsilon) \to (b, a, \varepsilon) \to (\varepsilon, \varepsilon, ba)$$

- state (split) components combine to current string
- state represents how far we have currently dipped into the string
- substitution calculus scans string from left to right
- back up (at most) 1 after $\to$-step

Implementation of 👀

Motivation for 👀

- TRSs interesting as target when compiling functional programming
- matching is simple (lhss linear and exactly two function symbols; cascading)
- substitution can be made to avoid replication by termgraph rewriting
- cost (time and space) linear by combining the above two items

Applicative Inductive Interaction System ()

Definition (of an)

TRS with signature $\{@/2, C_1/n_1, C_2/n_2, \ldots\}$ and for each i, rule $\varrho_{C_i}(x_0, x_1, \ldots, x_{n_i})$:

$$C_i(x_1, \ldots, x_{n_i})\, x_0 \quad \to \quad r$$

right-hand side r constructed from variables, @, and constructors C_j, for $j < i$

notational conventions:

- application @ infix, implicit as in Combinatory Logic (CL)
- usually leave arguments of rule symbols implicit (derivable from lhs of rule)

Applicative Inductive Interaction System (👁)

Definition (of an 👁)

TRS with signature $\{@/2, C_1/n_1, C_2/n_2, \ldots\}$ and for each i, rule $\varrho_{C_i}(x_0, x_1, \ldots, x_{n_i})$:

$$C_i(x_1, \ldots, x_{n_i})\, x_0 \quad \to \quad r$$

right-hand side r constructed from variables, @, and constructors C_j, for $j < i$

Example (of an 👁)

$$\begin{array}{rcrcl} \varrho_C(x_0, x_1, x_2) & : & C(x_1, x_2)\, x_0 & \to & x_1\, (x_2\, x_0) \\ \varrho_D(x_0) & : & D\, x_0 & \to & C(x_0, x_0) \end{array}$$

Applicative Inductive Interaction System (👁)

Definition (of an 👁)

TRS with signature $\{@/2, C_1/n_1, C_2/n_2, \ldots\}$ and for each i, rule $\varrho_{C_i}(x_0, x_1, \ldots, x_{n_i})$:

$$C_i(x_1, \ldots, x_{n_i})\, x_0 \quad \rightarrow \quad r$$

right-hand side r constructed from variables, @, and constructors C_j, for $j < i$

Example (👁 confluent (via orthogonality), Turing complete (via CL))

$$\begin{array}{rrcl}
\varrho_{S_2}: & S_2(x_1, x_2)\, x_0 & \rightarrow & (x_1\, x_0)\,(x_2\, x_0) \\
\varrho_{S_1}: & S_1(x_1)\, x_0 & \rightarrow & S_2(x_1, x_0) \\
\varrho_{S}: & S\, x_0 & \rightarrow & S_1(x_0)
\end{array}
\qquad
\begin{array}{rrcl}
\varrho_{K_1}: & K_1(x_1)\, x_0 & \rightarrow & x_1 \\
\varrho_{K}: & K\, x_0 & \rightarrow & K_1(x_0)
\end{array}$$

Applicative Inductive Interaction System (👁)

Example (of an 👁)

$$\begin{array}{rcrcl} \varrho_C(x_0, x_1, x_2) & : & C(x_1, x_2)\,x_0 & \to & x_1\,(x_2\,x_0) \\ \varrho_D(x_0) & : & D\,x_0 & \to & C(x_0, x_0) \end{array}$$

Example (two-step reduction $(\varrho_C(D, D)\,z_1) \cdot (\varrho_D(D\,z_1))$)

$$C(D, D)\,z_1 \to_{\varrho_C(z_1, D, D)} D\,(D\,z_1) \to_{\varrho_D(D\,z_1)} C(D\,z_1, D\,z_1)$$

duplicates $D\,z_1$ redex; ends in (constructor C-)head normal form

Implementing 👀

Question (on implementation of 👁)

do 👀 have an efficient (hyper-(head-))normalising reduction strategy?

efficient in time / space

Implementing 👁👁

Question (on implementation of 👁)

do 👁👁 have an efficient (hyper-(head-))normalising reduction strategy?

efficient in time / space

Observations (explored further below)

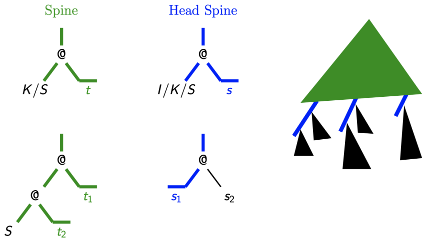
- spine strategy is (hyper-(head-))normalising
  since every 👁 is left-normal orthogonal TRSs
- matching-phase is trivial (since lhss left-linear, comprise two symbols)
  substitution-phase not trivial (rhss may replicate arguments)

Spine strategy

Definition

Spine: if head normal form recur, else Head Spine.

Head Spine: recur on left.



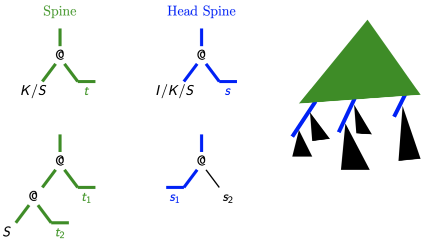
Example

S(SISI)(K(IK))

Spine strategy

Definition

Spine: if head normal form recur, else Head Spine.
Head Spine: recur on left.



Lemma

Every term not in normal form has Spine redex

Spine strategy for 👁

Definition (of spine for 👁-terms)

- spine: t or $x\, t_1\,,\ldots,\, t_n$
- head spine: x or $C(t_1,\ldots,t_n)$ or $t\, s$

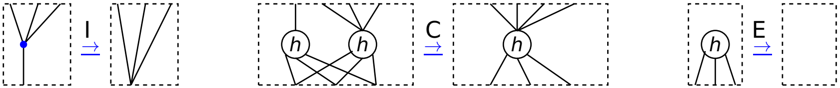
Lemma (normalising strategy)

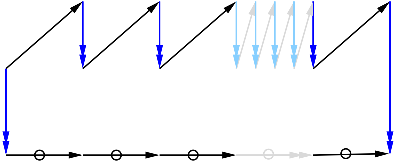
- *every term not in normal form has redex-pattern on spine, so a strategy*
- *spine strategy is a normalising strategy having random descent*

- random descent: reductions to normal form have same length / measure
- leftmost–outermost strategy is a spine-strategy

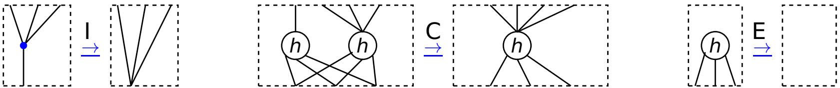
Implementing 👁 in termgraphs by cascading ж

Recall termgraph rewriting with ж-calculus as SC, and cascading:



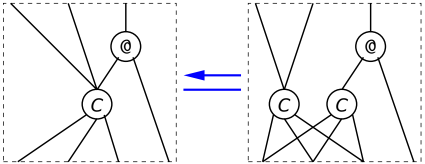


Implementing 👁 in termgraphs by cascading ж



Idea (minimal unsharing; Wadsworth's admissibility)

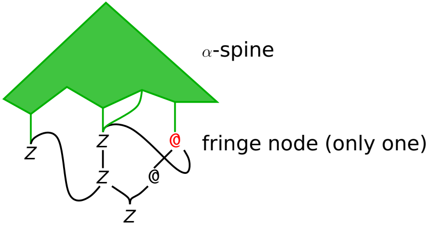
- instead of maximal sharing, unshare only constructors in redex-patterns
- goal: amortise cost of ж-steps by charging 👁-steps



Termgraph α-spine strategy

Definition (of (head / α-)spine nodes)

- spine: head spine, or such in normal form (hsnf) with spine vertebrae
- head spine: path from root through bodies of @,● to variable or constructor
- α-spine: spine prefix; fringe nodes: nodes covered by α-spine



Termgraph α-spine strategy

Definition (of (head / α-)spine nodes)

- spine: head spine, or such in normal form (hsnf) with spine vertebrae
- head spine: path from root through bodies of @,● to variable or constructor
- α-spine: spine prefix; fringe nodes: nodes covered by α-spine

Lemma

every termgraph not in normal form has a spine redex-pattern, and any (proper) α-spine prefix of it has a non-empty fringe

Proof.

by minimality using acyclicity of termgraphs □

Termgraph α-spine strategy

Definition (of (head / α-)spine nodes)

- spine: head spine, or such in normal form (hsnf) with spine vertebrae
- head spine: path from root through bodies of @,• to variable or constructor
- α-spine: spine prefix; fringe nodes: nodes covered by α-spine

Definition (of α-spine strategy)

reduce head spines from fringe nodes to hsnf and recurse on spine vertebrae

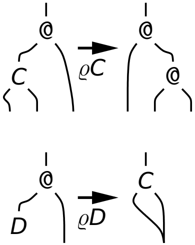
by lemma always some step possible until whole termgraph is α-spine (in nf)

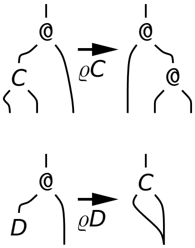
Example α-spine reduction (Java code ⇒ dot ⇒ graphs)

recall 👁-rules:

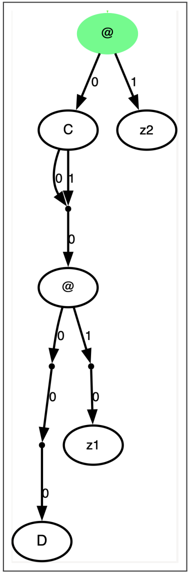
$\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$
$\varrho_D : D\, x_0 \to C(x_0, x_0)$

as termgraph rules:

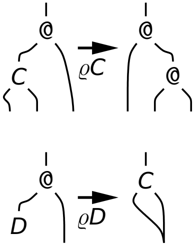




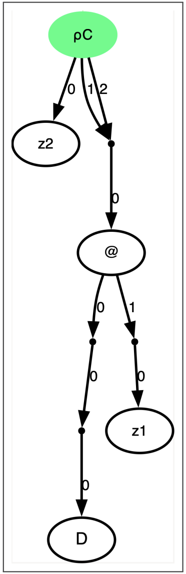
@
C
ϱC
@
@
@
D
ϱD
C



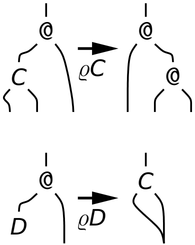
@
0
1
C
z2
0
1
0
@
0
1
0
0
z1
0
D

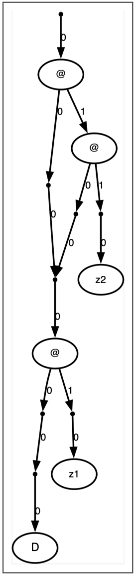


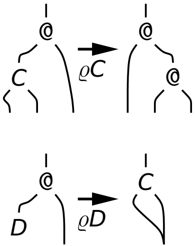
@
C
ϱC
@
@
@
D
ϱD
C

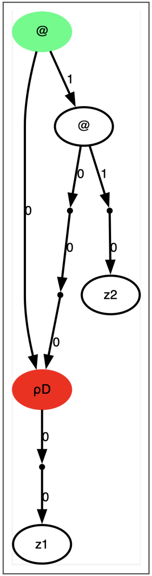


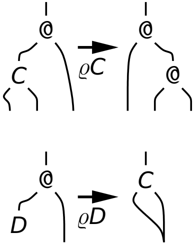
ρC
0
1
2
z2
0
@
0
1
0
0
z1
0
D

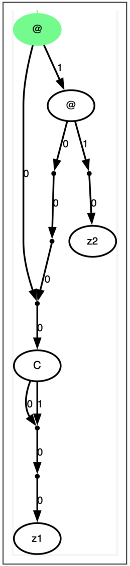

@
C
ϱC
@
@
@
D
ϱD
C

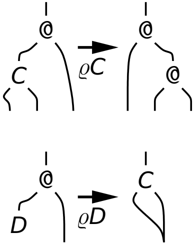

@
@
z2
@
z1
D

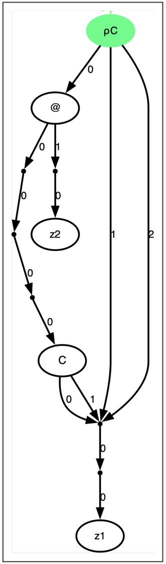

@
C
ϱC
@
@
@
D
ϱD
C

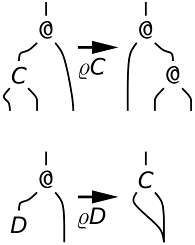

@
1
@
0
1
0
0
0
z2
0
ρD
0
0
z1


@
@
ϱC
C
@
@
C
ϱD
D

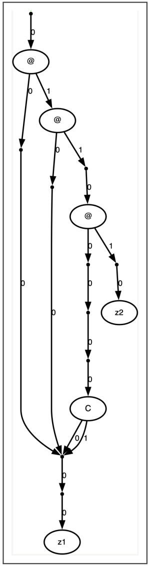

@
1
@
0
1
0
0
0
z2
0
0
C
0
1
0
0
z1


@
C
ϱC
@
@
@
D
ϱD
C

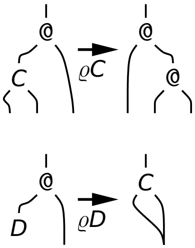

ρC
@
z2
C
z1



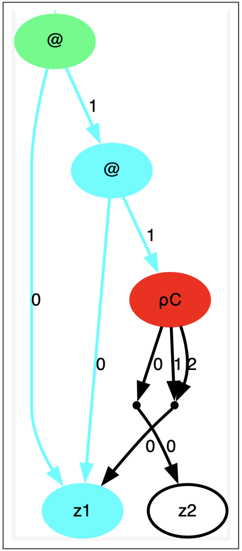
@
C
ϱC
@
@
@
D
ϱD
C



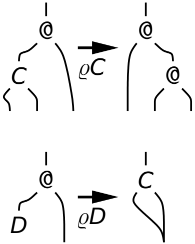
@
@
@
z2
C
z1

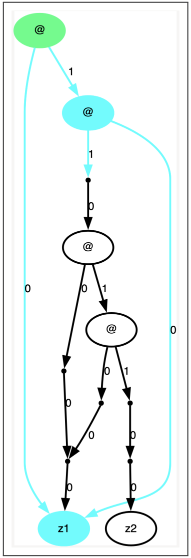


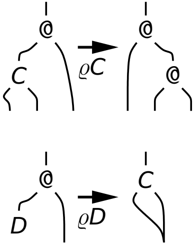
@
C
ϱC
@
@
@
D
ϱD
C

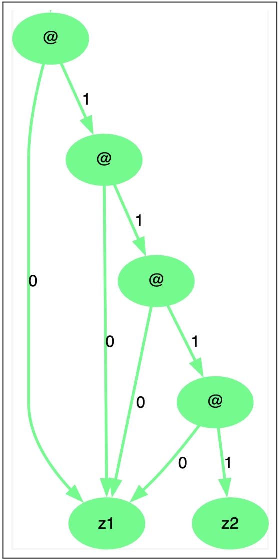


@
1
0
@
1
0
ρC
0
1
2
0
0
z1
z2


@
C
ϱC
@
@
@
D
ϱD
C


@
1
@
1
0
@
0
1
0
@
0
0
1
0
0
0
0
0
0
z1
z2


@
C
ϱC
@
@
@
D
ϱD
C


@
1
0
@
1
0
@
1
0
@
0
1
z1
z2

Correspondence between termgraphs and terms

Theorem

- *α-spine step maps to multistep having at least one spine redex ((hyper-)(head) normalising strategy)*

Correspondence between termgraphs and terms

Theorem

- *α-spine step maps to multistep having at least one spine redex*
- *multistep comprises redex-patterns having same creation history (family-step, so optimal strategy (qua horizontal sharing))*

Correspondence between termgraphs and terms

Theorem

- *α-spine step maps to multistep having at least one spine redex*
- *multistep comprises redex-patterns having same creation history*
- *cost and size linear in number of termgraph steps*
  (graph grows linearly; strategy visits links only few times à la DFS)

Correspondence between termgraphs and terms

Theorem

- *α-spine step maps to multistep having at least one spine redex*
- *multistep comprises redex-patterns having same creation history*
- *cost and size linear in number of termgraph steps*

- α-spine reduction length not longer than spine
  (👀 are orthogonal for which doing more in parallel is better)
- number of spine steps always the same (random descent property)
- reduction length not longer than that of leftmost–outermost stategy

Decompiling 👁 to the λ-calculus

Definition (of tree homomorphism $(\ )_\lambda$ into λ-terms)

$C_i(t_1,\ldots,t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1,\ldots,x_n{:=}t_1,\ldots,t_n]$

- capture avoiding substitution (avoid capture of free variables of the t_k)
- $(t[\vec{x}{:=}\vec{t}])_\lambda = (t)_\lambda[\vec{x}{:=}\overrightarrow{(t)_\lambda}]$ (substitution lemma)
- well-defined by 👁 being inductive (in r only C_j for $j < i$ may occur)

Decompiling to the λ-calculus

Definition (of tree homomorphism $(\ )_\lambda$ into λ-terms)

$C_i(t_1, \ldots, t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1, \ldots, x_n{:=}t_1, \ldots, t_n]$

Example (of tree homomorphism for example)

$$
\begin{array}{rcl|rcl}
 & \text{rule} & & & \text{tree homomorphism} & \\
\varrho_C : C(x_1, x_2)\, x_0 & \to & x_1\,(x_2\, x_0) & C(t_1, t_2) & \mapsto & \lambda x_0.t_1\,(t_2\, x_0) \\
\varrho_D : \qquad D\, x_0 & \to & C(x_0, x_0) & D & \mapsto & \lambda x_0 x_0'.x_0\,(x_0\, x_0')
\end{array}
$$

as $D \mapsto \lambda x_0.(C(x_0, x_0))_\lambda = \lambda x_0.(\lambda x_0.x_1\,(x_2\, x_0))[x_1, x_2{:=}x_0, x_0] =_\alpha \lambda x_0 x_0'.x_0\,(x_0\, x_0')$

Decompiling 👁 to the λ-calculus

Definition (of tree homomorphism $(\ )_\lambda$ into λ-terms)

$$C_i(t_1,\ldots,t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1,\ldots,x_n{:=}t_1,\ldots,t_n]$$

Example (of tree homomorphism for example 👁)

	rule			tree homomorphism		
$\varrho_C :$	$C(x_1,x_2)\,x_0$	$\to$	$x_1\,(x_2\,x_0)$	$C(t_1,t_2)$	$\mapsto$	$\lambda x_0.t_1\,(t_2\,x_0)$
$\varrho_D :$	$D\,x_0$	$\to$	$C(x_0,x_0)$	D	$\mapsto$	$\lambda x_0 x_0'.x_0\,(x_0\,x_0')$

- D maps to the Church numeral $\underline{2}$ for $\underline{n} := \lambda sz.s^n\,z$
- S maps to $\lambda xyz.x\,z\,(y\,z)$ and K to $\lambda xy.x$ as expected / hoped for

Decompiling 👁 to the λ-calculus

Definition (of tree homomorphism $(\ )_\lambda$ into λ-terms)

$C_i(t_1, \ldots, t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1, \ldots, x_n := t_1, \ldots, t_n]$

Lemma (implementation of 👁 by $\lambda\beta$)

if $t \rightarrow_{\text{👁}} s$ *then* $(t)_\lambda \rightarrow_\beta (s)_\lambda$

Decompiling 👁 to the λ-calculus

Definition (of tree homomorphism $(\ )_\lambda$ into λ-terms)

$C_i(t_1, \ldots, t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1, \ldots, x_n{:=}t_1, \ldots, t_n]$

Lemma (implementation of 👁 by $\lambda\beta$)

if $t \to_{\text{👁}} s$ *then* $(t)_\lambda \to_\beta (s)_\lambda$

Example (of implementing $D\,(D\,z_1) \to_{\text{👁}} C(D\,z_1, D\,z_1)$)

$$(D\,(D\,z_1))_\lambda = (\lambda xy.x\,(x\,y))\,(\underline{2}\,z_1) \to_\beta \lambda y.\underline{2}\,z_1\,(\underline{2}\,z_1\,y) =_\alpha (C(D\,z_1, D\,z_1))_\lambda$$

Compiling the λ-calculus to 👁

Lemma (??)

if $M \rightarrow_{\beta} N$ *then* $(M)_{👁} \rightarrow_{\mathcal{I}} (N)_{👁}$ *for* $\mathcal{I}$ *an* 👁

Compiling the λ-calculus to 👁

Lemma (??)

if $M \to_\beta N$ *then* $(M)_{\text{👁}} \to_{\mathcal{I}} (N)_{\text{👁}}$ *for* $\mathcal{I}$ *an* 👁

- no implementation $(\;)_{\text{👁}}$ can achieve that, for full β
- for weak β (wβ; contract redex if has no variable bound outside) it can:
- weak β is first-order (α-conversion never needed), and
- weak β basis of Haskell (no contraction under λ, but that's not confluent)

Compiling the λ-calculus to 👁

Lemma (??)

if $M \to_\beta N$ then $(M)_{👁} \to_{\mathcal{I}} (N)_{👁}$ for $\mathcal{I}$ an 👁

Definition (of $(\ )_{👁}$ mapping a λ-term to a pair of an 👁 and term in it)

- $(x)_{👁} := (\emptyset, x)$
- $(M_1\, M_2)_{👁} := (\mathcal{I}_1 \cup \mathcal{I}_2, t_1\, t_2)$, where $(\mathcal{I}_i, t_i) := (M_i)_{👁}$ for $i \in \{1, 2\}$
- $(\lambda x.M)_{👁} := (\{\varrho_C : C(z_1, \ldots, z_n)\, x \to r[z_1, \ldots, z_n]\} \cup \mathcal{I}, C(t_1, \ldots, t_n))$, where $(\mathcal{I}, r[t_1, \ldots, t_n]) := (M)_{👁}$, r skeleton, t_i maximal x-free subterm occurrences

do allow components to share constructors when these have the same rules
compilation known variation on the abstraction algorithm (custom combinators)

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

- $(x)_{👁} := (\emptyset, x)$
- $(M_1\, M_2)_{👁} := (\mathcal{I}_1 \cup \mathcal{I}_2, t_1\, t_2)$, where $(\mathcal{I}_i, t_i) := (M_i)_{👁}$ for $i \in \{1, 2\}$
- $(\lambda x.M)_{👁} := (\{\varrho_C : C(z_1, \ldots, z_n)\, x \to r[z_1, \ldots, z_n]\} \cup \mathcal{I}, C(t_1, \ldots, t_n))$, where $(\mathcal{I}, r[t_1, \ldots, t_n]) := (M)_{👁}$, r skeleton, t_i maximal x-free subterm occurrences

Example (of $(\underline{2})_{👁}$; recall $\underline{2} := \lambda xy.x\,(x\,y)$)

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

- $(x)_{\text{👁}} := (\emptyset, x)$
- $(M_1\, M_2)_{\text{👁}} := (\mathcal{I}_1 \cup \mathcal{I}_2, t_1\, t_2)$, where $(\mathcal{I}_i, t_i) := (M_i)_{\text{👁}}$ for $i \in \{1, 2\}$
- $(\lambda x.M)_{\text{👁}} := (\{\varrho_C : C(z_1, \ldots, z_n)\, x \to r[z_1, \ldots, z_n]\} \cup \mathcal{I}, C(t_1, \ldots, t_n))$, where $(\mathcal{I}, r[t_1, \ldots, t_n]) := (M)_{\text{👁}}$, r skeleton, t_i maximal x-free subterm occurrences

Example (of $(\underline{2})_{\text{👁}}$; recall $\underline{2} := \lambda xy.x\,(x\,y)$)

- $(x\,(x\,y))_{\text{👁}} := (\emptyset, x\,(x\,y))$ using only first two items of the definition

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

- $(x)_{\text{👁}} := (\emptyset, x)$
- $(M_1\, M_2)_{\text{👁}} := (\mathcal{I}_1 \cup \mathcal{I}_2, t_1\, t_2)$, where $(\mathcal{I}_i, t_i) := (M_i)_{\text{👁}}$ for $i \in \{1, 2\}$
- $(\lambda x.M)_{\text{👁}} := (\{\varrho_C : C(z_1, \ldots, z_n)\, x \to r[z_1, \ldots, z_n]\} \cup \mathcal{I}, C(t_1, \ldots, t_n))$, where $(\mathcal{I}, r[t_1, \ldots, t_n]) := (M)_{\text{👁}}$, r skeleton, t_i maximal x-free subterm occurrences

Example (of $(\underline{2})_{\text{👁}}$; recall $\underline{2} := \lambda xy.x\,(x\,y)$)

- $(x\,(x\,y))_{\text{👁}} := (\emptyset, x\,(x\,y))$, so
- $(\lambda y.x\,(x\,y))_{\text{👁}} := (\{\varrho_C : C(z_1, z_2)\, y \to z_1\,(z_2\, y)\}, C(x, x))$
  since x and x are maximal y-free subterm occurrences in $x\,(x\,y)$

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

- $(x)_{\text{👁}} := (\emptyset, x)$
- $(M_1\, M_2)_{\text{👁}} := (\mathcal{I}_1 \cup \mathcal{I}_2, t_1\, t_2)$, where $(\mathcal{I}_i, t_i) := (M_i)_{\text{👁}}$ for $i \in \{1, 2\}$
- $(\lambda x.M)_{\text{👁}} := (\{\varrho_C : C(z_1, \ldots, z_n)\, x \to r[z_1, \ldots, z_n]\} \cup \mathcal{I}, C(t_1, \ldots, t_n))$, where $(\mathcal{I}, r[t_1, \ldots, t_n]) := (M)_{\text{👁}}$, r skeleton, t_i maximal x-free subterm occurrences

Example (of $(\underline{2})_{\text{👁}}$; recall $\underline{2} := \lambda xy.x\,(x\,y)$)

- $(x\,(x\,y))_{\text{👁}} := (\emptyset, x\,(x\,y))$, so
- $(\lambda y.x\,(x\,y))_{\text{👁}} := (\{\varrho_C : C(z_1, z_2)\, y \to z_1\,(z_2\,y)\}, C(x, x))$, so
- $(\lambda xy.x\,(x\,y))_{\text{👁}} := (\{\varrho_C : C(z_1, z_2)\, y \to z_1\,(z_2\,y), \varrho_D : D\, x \to C(x, x)\}, D)$ since no x-free subterm occurrence in $C(x, x)$

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

- $(x)_{\text{👁}} := (\emptyset, x)$
- $(M_1\, M_2)_{\text{👁}} := (\mathcal{I}_1 \cup \mathcal{I}_2, t_1\, t_2)$, where $(\mathcal{I}_i, t_i) := (M_i)_{\text{👁}}$ for $i \in \{1, 2\}$
- $(\lambda x.M)_{\text{👁}} := (\{\varrho_C : C(z_1, \ldots, z_n)\, x \to r[z_1, \ldots, z_n]\} \cup \mathcal{I}, C(t_1, \ldots, t_n))$, where $(\mathcal{I}, r[t_1, \ldots, t_n]) := (M)_{\text{👁}}$, r skeleton, t_i maximal x-free subterm occurrences

Example (of $(\underline{2})_{\text{👁}}$; recall $\underline{2} := \lambda xy.x\,(x\,y)$)

- $(x\,(x\,y))_{\text{👁}} := (\emptyset, x\,(x\,y))$, so
- $(\lambda y.x\,(x\,y))_{\text{👁}} := (\{\varrho_C : C(z_1, z_2)\, y \to z_1\,(z_2\, y)\}, C(x, x))$, so
- $(\lambda xy.x\,(x\,y))_{\text{👁}} := (\{\varrho_C : C(z_1, z_2)\, y \to z_1\,(z_2\, y), \varrho_D : D\, x \to C(x, x)\}, D)$

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

- $(x)_{\text{👁}} := (\emptyset, x)$
- $(M_1\, M_2)_{\text{👁}} := (\mathcal{I}_1 \cup \mathcal{I}_2, t_1\, t_2)$, where $(\mathcal{I}_i, t_i) := (M_i)_{\text{👁}}$ for $i \in \{1, 2\}$
- $(\lambda x.M)_{\text{👁}} := (\{\varrho_C : C(z_1, \ldots, z_n)\, x \to r[z_1, \ldots, z_n]\} \cup \mathcal{I}, C(t_1, \ldots, t_n))$, where $(\mathcal{I}, r[t_1, \ldots, t_n]) := (M)_{\text{👁}}$, r skeleton, t_i maximal x-free subterm occurrences

Lemma (👁-lifting)

if $M \to_{\mathrm{w}\beta} N$ then $(M)_{\text{👁}} \to_{\mathcal{I}} (N)_{\text{👁}}$ for some 👁-lifting $\mathcal{I}$.

Proof.

if $M \to_{\mathrm{w}\beta} N$ and $(\mathcal{I}, t) := (M)_{\text{👁}}$ then $t \to_{\mathcal{I}} s$ for some $(\mathcal{I}', s) := (N)_{\text{👁}}$ with $\mathcal{I} \supseteq \mathcal{I}'$ $\square$

Implementing wβ-reduction via

Observations

- wβ never needs α-conversion, so essentially first-order (that's why it was chosen for Haskell)
- indeed, any λ-term *M* compiles to an and term *t* in it, such that rewriting from *M* respectively *t* is isomorphic
- compilation (finding mfss) can be done efficiently in time and space

Implementing wβ-reduction via 👁

Observations

- wβ never needs α-conversion, so essentially first-order
- indeed, any λ-term *M* compiles to an 👁 and term *t* in it, such that rewriting from *M* respectively *t* is isomorphic
- compilation can be done efficiently in time and space

Corollary

results for 👁👁 carry over to wβ

Implementing wβ-reduction via 👁

Observations

- wβ never needs α-conversion, so essentially first-order
- indeed, any λ-term M compiles to an 👁 and term t in it, such that rewriting from M respectively t is isomorphic
- compilation can be done efficiently in time and space

Corollary

results for 👁👁 carry over to wβ

Perspective

Haskell is based on orthogonal 1^{st}-order term rewriting (👁👁), not λ-calculus

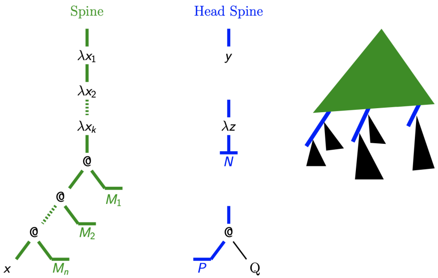
What about Spine strategies for full β?

Spine strategy

Definition

Spine: if head normal form recur, else Head Spine.

Head Spine: recur on left.



Example

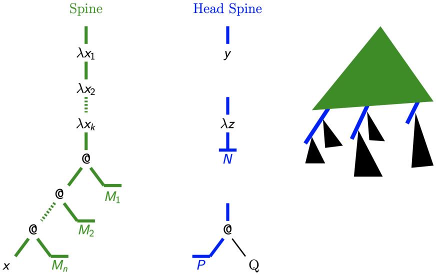
$x((\lambda x.(\lambda z.zz))y)(xx)(II)$

Spine strategy

Definition

Spine: if head normal form recur, else Head Spine.

Head Spine: recur on left.



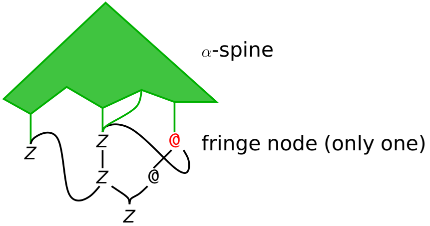
Lemma

Every term not in normal form has Spine redex

Termgraph α-spine strategy **adapted** to spine-β

Definition (of (head / α-)spine nodes)

- spine: head spine, or such in normal form (hsnf) with spine vertebrae
- head spine: path from root through bodies of @,• to variable or constructor
- α-spine: spine prefix; fringe nodes: nodes covered by α-spine



Termgraph α-spine strategy **adapted** to spine-β

Definition (of (head / α-)spine nodes)

- spine: head spine, or such in normal form (hsnf) with spine vertebrae
- head spine: path from root through bodies of @,● to variable or constructor
- α-spine: spine prefix; fringe nodes: nodes covered by α-spine

Definition (of α-spine strategy)

reduce head spines from fringe nodes to hsnf and recurse on spine vertebrae
rewrite fringe constructor $C(t_1, \ldots, t_n)$ to $\lambda x.C(t_1, \ldots, t_n)\, x$ for x fresh

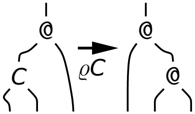
idea: a combinator on fringe / α-spine is a λ-abstraction (in the β-nf), so may iterate on its body, effectuated in 👁 by suppling a fresh variable

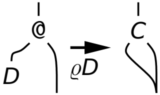
Example α-spine reduction (Java code $\Rightarrow$ dot $\Rightarrow$ graphs)

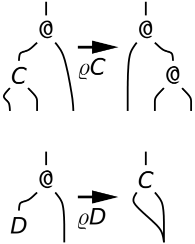
recall -rules:

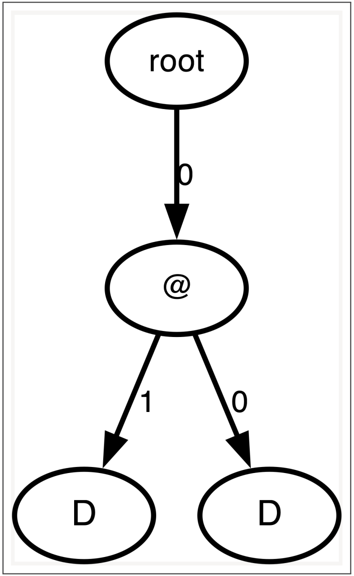
$$\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$$
$$\varrho_D : D\, x_0 \to C(x_0, x_0)$$

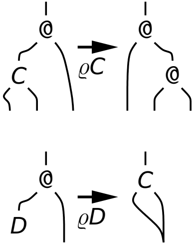
and termgraph rules:



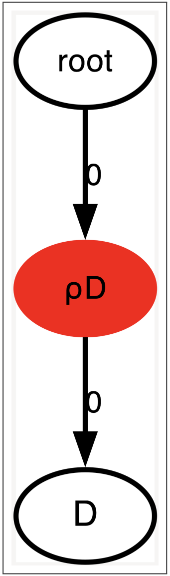



@
C
ϱC
@
@
@
D
ϱD
C

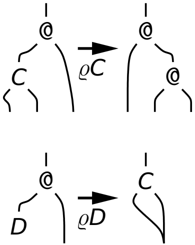

root
0
@
1
0
D
D

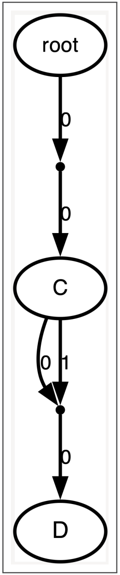


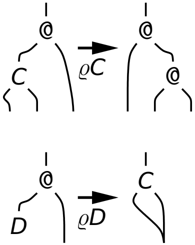
@
C
ϱC
@
@
@
D
ϱD
C

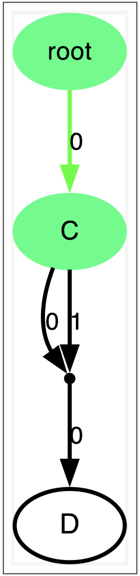


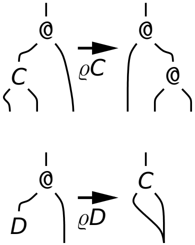
root
0
ρD
0
D

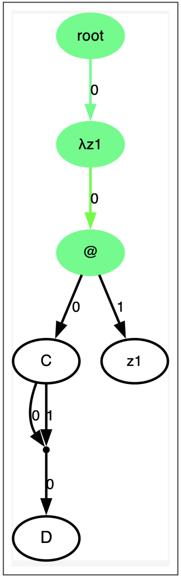

@
C
@
ϱC
@
@
D
ϱD
C

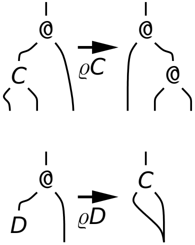

root
0
0
C
0
1
0
D

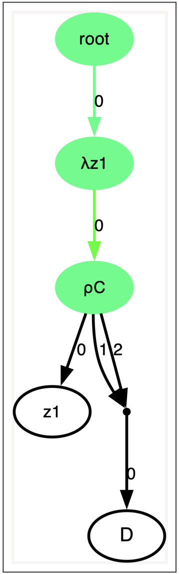

@
C
ϱC
@
@
@
D
ϱD
C

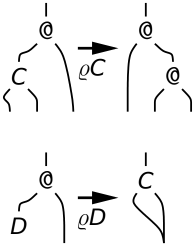

root
0
C
0
1
0
D

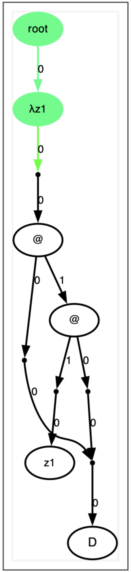

@
C
ϱC
@
@
@
D
ϱD
C

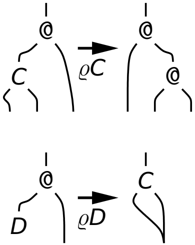

root
0
λz1
0
@
0
1
C
z1
0
1
0
D

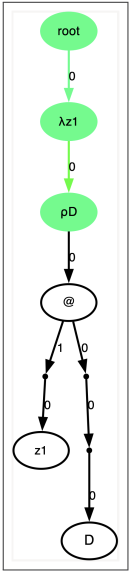

@
@
ϱC
C
@
@
C
ϱD
D

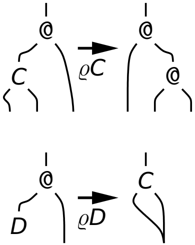

root
0
λz1
0
ρC
0
1
2
z1
0
D

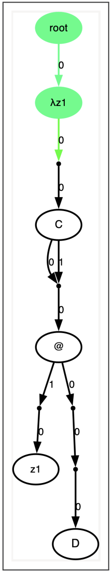

@
C
@
@
ϱC
@
D
C
ϱD

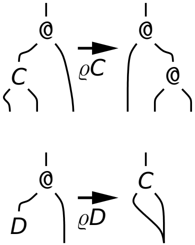

root
0
λz1
0
0
@
0
1
@
1
0
0
0
0
z1
0
D

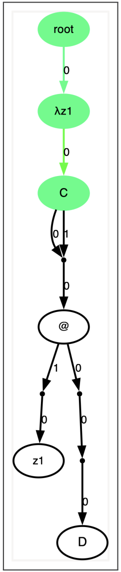

@
C
ϱC
@
@
@
D
ϱD
C

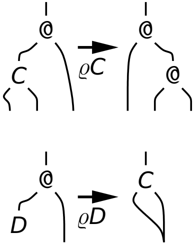

root
0
λz1
0
ρD
0
@
1
0
0
0
z1
0
D


@
C
ϱC
@
@
@
D
ϱD
C

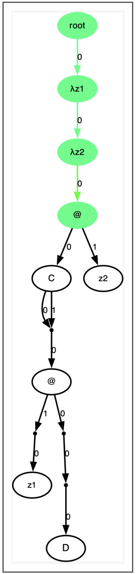

root
0
λz1
0
0
C
0
1
0
@
1
0
0
0
z1
0
D


@
C
ϱC
@
@
@
D
ϱD
C


root
λz1
C
@
z1
D



@
C
ϱC
@
@
@
D
ϱD
C



root
0
λz1
0
λz2
0
@
0
1
C
z2
0
1
0
@
1
0
0
0
z1
0
D

Implementing some spine-β-strategy via 👁

Observations

- β can be implemented via iterating wβ (for same 👁)

Implementing some spine-β-strategy via 👁

Observations

- β can be implemented via iterating wβ (for same 👁)
- constructor-steps correspond to needed α-conversions

Implementing some spine-β-strategy via 👁

Observations

- β can be implemented via iterating wβ (for same 👁)
- constructor-steps correspond to needed α-conversions
- how many α-conversions needed to β-reduce $((\underline{2}\,\underline{8})\,(\underline{4}\,\underline{9}))\,(\underline{5}\,\underline{7})\,(\underline{4}\,\underline{2})$ to nf?

Implementing some spine-β-strategy via

Observations

- β can be implemented via iterating wβ (for same)
- constructor-steps correspond to needed α-conversions
- how many α-conversions needed to β-reduce $((\underline{2}\,\underline{8})\,(\underline{4}\,\underline{9}))\,(\underline{5}\,\underline{7})\,(\underline{4}\,\underline{2})$ to nf?
- answer: ≤ 2 because output is a Church numeral, which has 2 λs

Implementing some spine-β-strategy via 👁

Observations

- β can be implemented via iterating wβ (for same 👁)
- constructor-steps correspond to needed α-conversions
- how many α-conversions needed to β-reduce $((\underline{2}\ \underline{8})\ (\underline{4}\ \underline{9}))\ (\underline{5}\ \underline{7})\ (\underline{4}\ \underline{2})$ to nf?
- answer: ≤ 2 because output is a Church numeral, which has 2 λs
- cost of constructor-steps amortised by other steps, for the same reason

Implementing some spine-β-strategy via 👁

Corollary

results for wβ carry over to spine-β, in particular that the cost of reduction to β-normal form is linear in the number of leftmost–outermost β-steps to β-nf

Implementing some spine-β-strategy via 👁

Corollary

results for wβ carry over to spine-β, in particular that the cost of reduction to β-normal form is linear in the number of leftmost–outermost β-steps to β-nf

Perspective

classical 1^{st}-order term(graph) rewrite theory trivialises (extant) cost-analyses

Implementing β-reduction

Complexity unavoidable

convertibility of simply typed λ-calculus is non-elementary. Upshot: whatever way you slice the pie (split into β and substitutions) that can't be overcome.

Implementing β-reduction

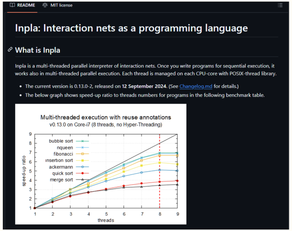
Complexity unavoidable

convertibility of simply typed λ-calculus is non-elementary. Upshot: whatever way you slice the pie (split into β and substitutions) that can't be overcome.

Non-consequence

Optimal reduction for full β is non-interesting. By the same token all implementations shown here would be non-interesting as they are optimal but for wβ.

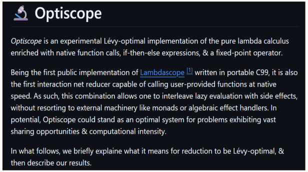
Springtime for interaction nets!



github.com/inpla/inpla

higherorderco.com

vine.dev



github.com/etiams/optiscope

Some conclusions

- unit-time steps *a priori* unreasonable for structured rewriting

Some conclusions

- unit-time steps *a priori* unreasonable for structured rewriting
- rewriting useful both for simple description and efficient implementation (do away with abstract machines)

Some conclusions

- unit-time steps *a priori* unreasonable for structured rewriting
- rewriting useful both for simple description and efficient implementation
- substitution calculi give a way to account for the cost of substitution (how to slice the pie, between replacement and substitution)

Some conclusions

- unit-time steps *a priori* unreasonable for structured rewriting
- rewriting useful both for simple description and efficient implementation
- substitution calculi give a way to account for the cost of substitution
- α-spine is 1^{st}-order optimal for 👁, wβ and β
  (only need skeletons present in initial λ-term; no creation of such)

Some conclusions

- unit-time steps *a priori* unreasonable for structured rewriting
- rewriting useful both for simple description and efficient implementation
- substitution calculi give a way to account for the cost of substitution
- α-spine is 1st-order optimal for 👁, wβ and β
- α-spine time and space linear in #steps (via TGRS, in Java)

Some conclusions

- unit-time steps *a priori* unreasonable for structured rewriting
- rewriting useful both for simple description and efficient implementation
- substitution calculi give a way to account for the cost of substitution
- α-spine is 1st-order optimal for 👁, wβ and β
- α-spine time and space linear in #steps (via TGRS, in Java)
- amortised analysis: discounting •-steps via #nodes, α-steps via β-steps (former based on path-compression of in-edges of •-nodes)

Some conclusions

- unit-time steps *a priori* unreasonable for structured rewriting
- rewriting useful both for simple description and efficient implementation
- substitution calculi give a way to account for the cost of substitution
- α-spine is 1^{st}-order optimal for 👁, wβ and β
- α-spine time and space linear in #steps (via TGRS, in Java)
- amortised analysis: discounting $\bullet$-steps via #nodes, α-steps via β-steps
- higher-order term rewriting useful to bridge λ-calculus and 👁👁

Standing on the shoulders of giants

1. Newman (1942): for rewrite systems and random descent

Standing on the shoulders of giants

1. Newman (1942): for rewrite systems and random descent
2. Wadsworth (1971): graph rewriting implementation of β-reduction

Standing on the shoulders of giants

1. Newman (1942): for rewrite systems and random descent
2. Wadsworth (1971): graph rewriting implementation of β-reduction
3. Barendregt, Bergstra, Klop, Volken (1976): no computable optimal β-strat

Standing on the shoulders of giants

1. Newman (1942): for rewrite systems and random descent
2. Wadsworth (1971): graph rewriting implementation of β-reduction
3. Barendregt, Bergstra, Klop, Volken (1976): no computable optimal β-strat
4. Lévy (1978): concept of β-family and optimality of lmo-β-family strategy

Standing on the shoulders of giants

1. Newman (1942): for rewrite systems and random descent
2. Wadsworth (1971): graph rewriting implementation of β-reduction
3. Barendregt, Bergstra, Klop, Volken (1976): no computable optimal β-strat
4. Lévy (1978): concept of β-family and optimality of lmo-β-family strategy
5. Huet, Lévy (1979): concept of needed reduction and it being normalising

Standing on the shoulders of giants

1. Newman (1942): for rewrite systems and random descent
2. Wadsworth (1971): graph rewriting implementation of β-reduction
3. Barendregt, Bergstra, Klop, Volken (1976): no computable optimal β-strat
4. Lévy (1978): concept of β-family and optimality of lmo-β-family strategy
5. Huet, Lévy (1979): concept of needed reduction and it being normalising
6. Barendregt, Kennaway, Klop, Sleep (1987): concept of (head) spine strategy

Standing on the shoulders of giants

1. Newman (1942): for rewrite systems and random descent
2. Wadsworth (1971): graph rewriting implementation of β-reduction
3. Barendregt, Bergstra, Klop, Volken (1976): no computable optimal β-strat
4. Lévy (1978): concept of β-family and optimality of lmo-β-family strategy
5. Huet, Lévy (1979): concept of needed reduction and it being normalising
6. Barendregt, Kennaway, Klop, Sleep (1987): concept of (head) spine strategy
7. Lamping (1990): sharing graph implementation of β-families

Standing on the shoulders of giants

1. Newman (1942): for rewrite systems and random descent
2. Wadsworth (1971): graph rewriting implementation of β-reduction
3. Barendregt, Bergstra, Klop, Volken (1976): no computable optimal β-strat
4. Lévy (1978): concept of β-family and optimality of lmo-β-family strategy
5. Huet, Lévy (1979): concept of needed reduction and it being normalising
6. Barendregt, Kennaway, Klop, Sleep (1987): concept of (head) spine strategy
7. Lamping (1990): sharing graph implementation of β-families
8. Asperti, Mairson (1998): complexity of β-family reduction is non-elementary

Standing on the shoulders of giants

1. Newman (1942): for rewrite systems and random descent
2. Wadsworth (1971): graph rewriting implementation of β-reduction
3. Barendregt, Bergstra, Klop, Volken (1976): no computable optimal β-strat
4. Lévy (1978): concept of β-family and optimality of lmo-β-family strategy
5. Huet, Lévy (1979): concept of needed reduction and it being normalising
6. Barendregt, Kennaway, Klop, Sleep (1987): concept of (head) spine strategy
7. Lamping (1990): sharing graph implementation of β-families
8. Asperti, Mairson (1998): complexity of β-family reduction is non-elementary
9. Grégoire, Leroy (2002): β via iterated wβ

Standing on the shoulders of giants

1. Newman (1942): for rewrite systems and random descent
2. Wadsworth (1971): graph rewriting implementation of β-reduction
3. Barendregt, Bergstra, Klop, Volken (1976): no computable optimal β-strat
4. Lévy (1978): concept of β-family and optimality of lmo-β-family strategy
5. Huet, Lévy (1979): concept of needed reduction and it being normalising
6. Barendregt, Kennaway, Klop, Sleep (1987): concept of (head) spine strategy
7. Lamping (1990): sharing graph implementation of β-families
8. Asperti, Mairson (1998): complexity of β-family reduction is non-elementary
9. Grégoire, Leroy (2002): β via iterated wβ
10. Blanc, Lévy, Maranget (2005): wβ-family, implemented here (Wadsworth)

Contributions

1. concept of substitution calculus (1994)
2. optimal implementation of lmo-β-family by scope nodes (2004)
3. wβ being isomorphic to orthogonal TRS, given a λ-term (2005)
4. optimality of wβ being an instance of optimality of orthogonal TRSs (2005)
5. the α-spine strategy for 👁👁 (2024)
6. Haskell code implementing wβ into an 👁 and vice versa (2024);
7. linear TGRS implementation of 👁/ wβ / spine-β (2024)
8. Java code for that implementation (2025)
9. naming applicative inductive interaction systems 👁👁 (2025)

Amortised complexity

Idea

measure complexity by averaging over reductions (Tarjan)
(instead of measuring per step)

Amortised complexity

Idea

measure complexity by averaging over reductions

Example

incrementing a counter in binary $011 \rightarrow_{\mathrm{inc}} 111 \rightarrow_{\mathrm{inc}} 0001 \rightarrow_{\mathrm{inc}} 1001 \rightarrow_{\mathrm{inc}} \ldots$
($\rightarrow_{\mathrm{inc}}$-steps not unit-time; $\#$bit-flips unbounded)

Amortised complexity

Idea

measure complexity by averaging over reductions

Example

incrementing a counter in binary $011 \to_{\mathrm{inc}} 111 \to_{\mathrm{inc}} 0001 \to_{\mathrm{inc}} 1001 \to_{\mathrm{inc}} \ldots$

Example (inc as term rewrite system; $\to_{\mathrm{inc}} := \to_i \cdot \to_b^!$)

$$s \to_i i(s) \qquad i(0(x)) \to_b 1(x) \qquad i(1(x)) \to_b 0(i(x)) \qquad i(\bullet) \to_b 1(\bullet)$$

Amortised complexity

Idea

measure complexity by averaging over reductions

Example

incrementing a counter in binary $011 \rightarrow_{\mathrm{inc}} 111 \rightarrow_{\mathrm{inc}} 0001 \rightarrow_{\mathrm{inc}} 1001 \rightarrow_{\mathrm{inc}} \ldots$

Example (inc as term rewrite system; $\rightarrow_{\mathrm{inc}} := \rightarrow_i \cdot \rightarrow_b^!$)

$$s \rightarrow_i i(s) \qquad i(0(x)) \rightarrow_b 1(x) \qquad i(1(x)) \rightarrow_b 0(i(x)) \qquad i(\bullet) \rightarrow_b 1(\bullet)$$

$0(1(1(\bullet))) \rightarrow_i i(0(1(1(\bullet)))) \rightarrow_b 1(1(1(\bullet))) \rightarrow_i i(1(1(1(\bullet)))) \rightarrow_b 0(i(1(1(\bullet)))) \rightarrow_b$
$0(0(i(1(\bullet)))) \rightarrow_b 0(0(0(i(\bullet)))) \rightarrow_b 0(0(0(1(\bullet)))) \rightarrow_i \ldots$

Banker's / accounting method in TRSs

Idea

distinguish between charge $\hat{c}$ and cost c of steps. i-steps add charge to pay for cost of subsequent b-steps; labelled ($\mathbb{N}$) symbols as saving-account for charges

Banker's / accounting method in TRSs

Idea

distinguish between charge $\hat{c}$ and cost c of steps. i-steps add charge to pay for cost of subsequent b-steps; labelled ($\mathbb{N}$) symbols as saving-account for charges

Example

$s \rightarrow_{\hat{3},1} i^{\hat{2}}(s) \qquad i^{\hat{2}}(0(x)) \rightarrow_{\hat{0},1} 1^{\hat{1}}(x) \qquad i^{\hat{2}}(1^{\hat{1}}(x)) \rightarrow_{\hat{0},1} 0(i^{\hat{2}}(x)) \qquad i^{\hat{2}}(\bullet) \rightarrow_{\hat{0},1} 1^{\hat{1}}(\bullet)$

(no need to label 0's or $\bullet$'s)

Banker's / accounting method in TRSs

Idea

distinguish between charge $\hat{c}$ and cost c of steps. i-steps add charge to pay for cost of subsequent b-steps; labelled ($\mathbb{N}$) symbols as saving-account for charges

Example

$$s \to_{\hat{3},1} i^{\hat{2}}(s) \qquad i^{\hat{2}}(0(x)) \to_{\hat{0},1} 1^{\hat{1}}(x) \qquad i^{\hat{2}}(1^{\hat{1}}(x)) \to_{\hat{0},1} 0(i^{\hat{2}}(x)) \qquad i^{\hat{2}}(\bullet) \to_{\hat{0},1} 1^{\hat{1}}(\bullet)$$

- $\hat{\imath}$ initially labels (closed): charge i with $\hat{2}$ and 1 with $\hat{1}$; preserved by steps

Banker's / accounting method in TRSs

Idea

distinguish between charge $\hat{c}$ and cost c of steps. i-steps add charge to pay for cost of subsequent b-steps; labelled ($\mathbb{N}$) symbols as saving-account for charges

Example

$$s \rightarrow_{\hat{3},1} i^{\hat{2}}(s) \qquad i^{\hat{2}}(0(x)) \rightarrow_{\hat{0},1} 1^{\hat{1}}(x) \qquad i^{\hat{2}}(1^{\hat{1}}(x)) \rightarrow_{\hat{0},1} 0(i^{\hat{2}}(x)) \qquad i^{\hat{2}}(\bullet) \rightarrow_{\hat{0},1} 1^{\hat{1}}(\bullet)$$

- $\hat{\iota}$ initially labels: charge i with $\hat{2}$ and 1 with $\hat{1}$; preserved by steps
- is a labelling: if $t \twoheadrightarrow s$, then $t^{\hat{\iota}} \twoheadrightarrow s^{\hat{\iota}}$

Banker's / accounting method in TRSs

Idea

distinguish between charge $\hat{c}$ and cost c of steps. i-steps add charge to pay for cost of subsequent b-steps; labelled ($\mathbb{N}$) symbols as saving-account for charges

Example

$$s \to_{\hat{3},1} i^{\hat{2}}(s) \qquad i^{\hat{2}}(0(x)) \to_{\hat{0},1} 1^{\hat{1}}(x) \qquad i^{\hat{2}}(1^{\hat{1}}(x)) \to_{\hat{0},1} 0(i^{\hat{2}}(x)) \qquad i^{\hat{2}}(\bullet) \to_{\hat{0},1} 1^{\hat{1}}(\bullet)$$

- $\hat{\iota}$ initially labels: charge i with $\hat{2}$ and 1 with $\hat{1}$; preserved by steps
- is a labelling: if $t \twoheadrightarrow s$, then $t^{\hat{\iota}} \twoheadrightarrow s^{\hat{\iota}}$
  (in general: cost subtracted; charges must remain non-negative, cover costs of steps; $\hat{c} + \sum \ell \geq c + \sum r$ for each (linear) rule $\ell \to_{\hat{c},c} r$)

Banker's / accounting method in TRSs

Idea

distinguish between charge $\hat{c}$ and cost c of steps. i-steps add charge to pay for cost of subsequent b-steps; labelled ($\mathbb{N}$) symbols as saving-account for charges

Example

$$s \to_{\hat{3},1} i^{\hat{2}}(s) \qquad i^{\hat{2}}(0(x)) \to_{\hat{0},1} 1^{\hat{1}}(x) \qquad i^{\hat{2}}(1^{\hat{1}}(x)) \to_{\hat{0},1} 0(i^{\hat{2}}(x)) \qquad i^{\hat{2}}(\bullet) \to_{\hat{0},1} 1^{\hat{1}}(\bullet)$$

- $\hat{\iota}$ initially labels: charge i with $\hat{2}$ and 1 with $\hat{1}$; preserved by steps
- is a labelling: if $t \twoheadrightarrow s$, then $t^{\hat{\iota}} \twoheadrightarrow s^{\hat{\iota}}$
- cost of reduction from t bounded by amortized cost, $\leq 3 \cdot \#i + \sum t^{\hat{\iota}}$

Reduction to (wh)nf in $\lambda\beta$, naïvely, in Haskell

```
data Lam = Lam Head [Lam] deriving (Show)
data Head = Var String | Abs String Lam deriving (Show)
subst x s (Lam h l) = let
  (Lam h' l') = case h of
     (Var y)   | x == y -> s
     (Abs y u) | x /= y -> Lam (Abs y (subst x s u)) []
     _                  -> Lam h [] in (Lam h' (l'++(map (subst x s) l)))
whnf (Lam (Abs x t) (u:l)) = let Lam h s = subst x u t in whnf (Lam h (s++l))
whnf t = t
nf = rnf (\x -> 1)
rnf f t = let
  (Lam h l) = whnf t
  f' x      = \y -> f y + (if (x==y) then 1 else 0)
  v x       = x++"_"++show (f x) in case h of
    (Abs x _) -> Lam (Abs (v x) (rnf (f' x) (Lam h [Lam (Var (v x)) []]))) []
    _         -> Lam h (map (rnf f) l)
```