Termgraph rewriting for implementing the λ-calculus refactored (WiP)

Vincent van Oostrom & Clemens Grabmayer

Part I: Implementing closed β by orthogonal TRS (👁)

Part II: Implementing 👁 through termgraphs G👁

Part III : Implementing β through G👁

Running example (from $\lambda\beta$-calculus)

Example (2^2)

- Church numeral $\underline{2} := \lambda sz.s\,(s\,z)$

Running example

Example (2^2)

- $\underline{2} := \lambda sz.s\,(s\,z)$
- application of Church numerals is exponentiation: $\underline{k}\,\underline{n} \twoheadrightarrow_\beta \underline{n^k}$

Running example

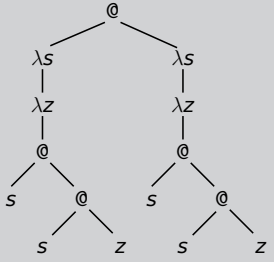
Example (2^2)

- $\underline{2} := \lambda sz.s\,(s\,z)$
- $\underline{2}\,\underline{2} = (\lambda sz.s\,(s\,z))\,\underline{2} \to_\beta \lambda z.\underline{2}\,(\underline{2}\,z) =_\alpha \lambda s'.(\lambda sz.s\,(s\,z))\,(\underline{2}\,s') \to_\beta$
  $\lambda s'z.\underline{2}\,s'\,(\underline{2}\,s'\,z) =_\alpha \lambda sz.\underline{2}\,s\,(\underline{2}\,s\,z) \twoheadrightarrow_\beta \lambda sz.\underline{2}\,s\,(s\,(s\,z)) \twoheadrightarrow_\beta \lambda sz.s\,(s\,(s\,(s\,z))) =: \underline{4}$

Running example

Example (2^2)

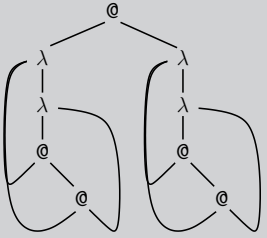
- $\underline{2} := \lambda sz.s\,(s\,z)$
- $\underline{2}\,\underline{2} \twoheadrightarrow_\beta \underline{2^2} =: \underline{4} =: \lambda sz.s\,(s\,(s\,(s\,z)))$
- abstract syntax tree of $\underline{2}\,\underline{2}$:



Running example

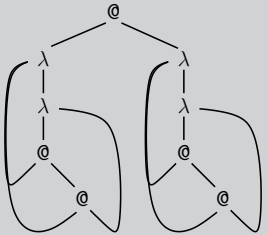
Example (2^2)

- $\underline{2} := \lambda sz.s\,(s\,z)$
- $\underline{2}\,\underline{2} \twoheadrightarrow_\beta \underline{2^2} =: \underline{4} =: \lambda sz.s\,(s\,(s\,(s\,z)))$
- abstract binding tree of $\underline{2}\,\underline{2}$:



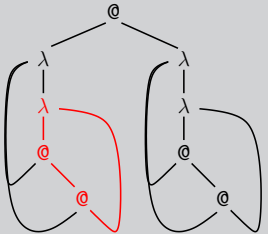
Wadsworth refactored through term rewriting; idea

Example (skeleton of anonymous function $\mapsto$ named function)



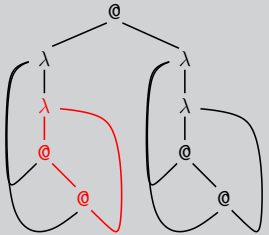
Wadsworth refactored through term rewriting; idea

Example (skeleton of anonymous function $\mapsto$ named function)



Wadsworth refactored through term rewriting; idea

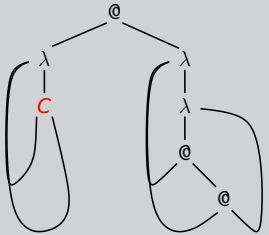
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : @(C(x_1, x_2), x_0) \to x_1\,(x_2\,x_0)$

Wadsworth refactored through term rewriting; idea

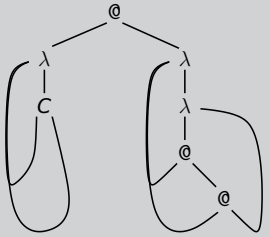
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : @(C(x_1, x_2), x_0) \to x_1\,(x_2\,x_0)$

Wadsworth refactored through term rewriting; idea

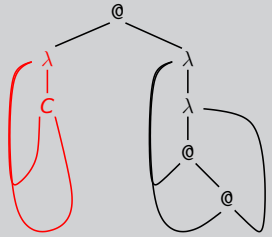
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$

Wadsworth refactored through term rewriting; idea

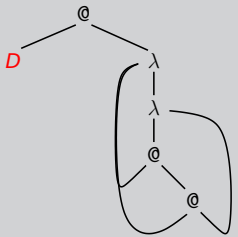
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$

Wadsworth refactored through term rewriting; idea

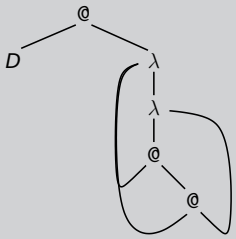
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$
function symbol D with TRS rule $\varrho_D : D\, x_0 \to C(x_0, x_0)$

Wadsworth refactored through term rewriting; idea

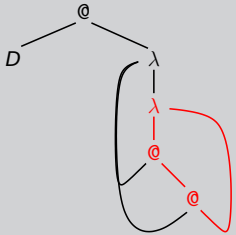
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$
function symbol D with TRS rule $\varrho_D : D\, x_0 \to C(x_0, x_0)$

Wadsworth refactored through term rewriting; idea

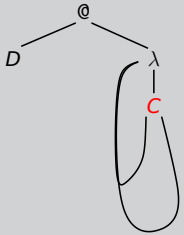
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$
function symbol D with TRS rule $\varrho_D : D\, x_0 \to C(x_0, x_0)$

Wadsworth refactored through term rewriting; idea

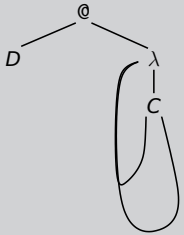
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$ (reuse)
function symbol D with TRS rule $\varrho_D : D\, x_0 \to C(x_0, x_0)$

Wadsworth refactored through term rewriting; idea

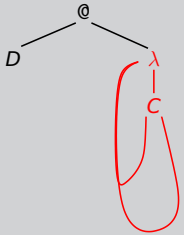
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$
function symbol D with TRS rule $\varrho_D : D\, x_0 \to C(x_0, x_0)$

Wadsworth refactored through term rewriting; idea

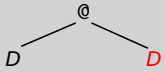
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$
function symbol D with TRS rule $\varrho_D : D\, x_0 \to C(x_0, x_0)$

Wadsworth refactored through term rewriting; idea

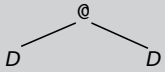
Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : C(x_1, x_2)\, x_0 \to x_1\,(x_2\, x_0)$
function symbol D with TRS rule $\varrho_D : D\, x_0 \to C(x_0, x_0)$ (reuse)

Wadsworth refactored through term rewriting; idea

Example (skeleton of anonymous function $\mapsto$ named function)



function symbol C with TRS rule $\varrho_C : C(x_1, x_2)\, x_0 \to x_1\, (x_2\, x_0)$
function symbol D with TRS rule $\varrho_D : D\, x_0 \to C(x_0, x_0)$

Compiling a λ-term into a term in a TRS

Definition ($\circledcirc$ maps a λ-term M to pair $(\mathcal{R},t)$ of TRS $\mathcal{R}$ and term t in it)

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

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

Compiling a λ-term into a term in a TRS

Definition (👁-lifting; Hughes 82)

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

Example

$(\underline{2}\,\underline{2})_{👁} := (\{\varrho_C : C(x_1,x_2)\, x_0 \to x_1\, (x_2\, x_0), \varrho_D : D\, x_0 \to C(x_0,x_0)\}, D\, D)$
since $(\underline{2})_{👁} := (\{\varrho_C : C(x_1,x_2)\, x_0 \to x_1\, (x_2\, x_0), \varrho_D : D\, x_0 \to C(x_0,x_0)\}, D)$

Compiling a λ-term into a term in a TRS

Definition ($\circledcirc$-lifting)

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

Example

$(\underline{2}\,\underline{2})_{\circledcirc} := (\{\varrho_C : C(x_1, x_2)\, x_0 \to x_1\, (x_2\, x_0), \varrho_D : D\, x_0 \to C(x_0, x_0)\}, D\, D)$

Example

$\underline{2}\,\underline{2} \to_\beta \lambda z.\underline{2}\,(\underline{2}\, z)$ implemented by $D\, D \to_{\varrho_D} C(D, D)$

Standard example: compiling combinators

Example (combinators in the λ-calculus)

$$\mathfrak{I} := \lambda x.x \qquad \mathfrak{K} := \lambda x\,y.x \qquad \mathfrak{S} := \lambda x\,y\,z.x\,z\,(y\,z)$$

eye-lifting of $\mathfrak{I}\,\mathfrak{K}\,\mathfrak{S}$?

Standard example: compiling combinators

Example (combinators in the λ-calculus)

$$\mathfrak{I} := \lambda x.x \qquad \mathfrak{K} := \lambda x\,y.x \qquad \mathfrak{S} := \lambda x\,y\,z.x\,z\,(y\,z)$$

$(\mathfrak{I}\,\mathfrak{K}\,\mathfrak{S})_{\text{👁}} := (\mathcal{R}, I\,K\,S)$ where $\mathcal{R}$ has rules:

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

flat Combinatory Logic ($\mathrm{CL}^\flat$); Asperti & Laneve 96, Goncharov & coauthors 24…
idea: first collect sufficiently many arguments, then reduce to right-hand side

Standard example: compiling combinators

Example (combinators in the λ-calculus)

$$\mathfrak{I} := \lambda x.x \qquad \mathfrak{K} := \lambda x\,y.x \qquad \mathfrak{S} := \lambda x\,y\,z.x\,z\,(y\,z)$$

$(\mathfrak{I}\,\mathfrak{K}\,\mathfrak{S})_{\text{👁}} := (\mathcal{R}, I\,K\,S)$ where $\mathcal{R}$ has rules:

$$\begin{array}{llll} \varrho_{S_2} : S_2(x_1,x_2)\,x_0 & \to & (x_1\,x_0)\,(x_2\,x_0) & \qquad \varrho_{K_1} : K_1(x_1)\,x_0 \to x_1 \\ \varrho_{S_1} : S_1(x_1)\,x_0 & \to & S_2(x_1,x_0) & \qquad \varrho_K : K\,x_0 \to K_1(x_0) \\ \varrho_S : S\,x_0 & \to & S_1(x_0) & \qquad \varrho_I : I\,x_0 \to x_0 \end{array}$$

Example

$\mathfrak{I}\,\mathfrak{K}\,\mathfrak{S} \to_\beta \mathfrak{K}\,\mathfrak{S} \to_\beta \lambda y.\mathfrak{S}$ implemented by $I\,K\,S \to_{\varrho_I} K\,S \to_{\varrho_K} K_1(S)$

Implementation of β?

Question

What is an implementation?

preciously little literature on that (but *cf.*, *e.g.*, Sabry & Wadler); go for it!

Implementation of β?

Definition

implementation is an isomorphism of rewrite systems

unrealistically strong, but a starting point

Implementation of β?

Definition

implementation is an isomorphism of rewrite systems

Question

β-reduction graph of M isomorphic to $\mathcal{R}$-reduction graph of t, for $(\mathcal{R},t) := (M)_{\text{👁}}$?

Implementation of β?

Definition

implementation is an isomorphism of rewrite systems

Question

β-reduction graph of M isomorphic to $\mathcal{R}$-reduction graph of t, for $(\mathcal{R},t) := (M)_{\text{👁}}$?

Example (no!)

$M = \lambda x.\mathfrak{I}\, x \to_\beta \lambda x.x =: \mathfrak{I}$
$t := C$ in normal form for $\mathcal{R} := \{\varrho_I : I\, x_0 \to x_0, \varrho_C : C\, x_0 \to I\, x_0\}$

Implementation of β?

Definition

implementation is an isomorphism of rewrite systems

Question

β-reduction graph of M isomorphic to $\mathcal{R}$-reduction graph of t, for $(\mathcal{R},t) := (M)_{\text{👁}}$?

Example (no!)

$M = \lambda x.\mathfrak{I}\,x \to_\beta \lambda x.x =: \mathfrak{I}$
$t :=$ C in normal form for $\mathcal{R} := \{\varrho_I : I\,x_0 \to x_0, \varrho_C : C\,x_0 \to I\,x_0\}$

t in normal form for any closed M of shape $\lambda x.N(x)$; *cf.* abstraction algorithm
intuition: nameless $\mapsto$ named implementation hides non-free redexes from view

Implementation of $\bar{\beta}$

Definition

$\bar{\beta}$ is closed β: no variable in redex bound above it (Çagman & Hindley, Mackie. . .)

for programs (no free variables) all $\bar{\beta}$-redexes are indeed closed

Example

$\lambda x.\mathcal{I}\,x$ is in $\bar{\beta}$-normal form since x (free) in β-redex $\mathcal{I}\,x$ and bound above it (by λx)
$\lambda x.\mathcal{I}\,\mathcal{I}$ has $\bar{\beta}$-redex occurrence $\mathcal{I}\,\mathcal{I}$

Implementation of $\bar{\beta}$

Definition

$\bar{\beta}$ is closed β: no variable in redex bound above it

$\bar{\beta}$ needs no (α-)renaming; no variables in argument $\implies$ no variable capture
(motivation for usage of $\bar{\beta}$ in functional programming; Peyton Jones)
$\bar{\beta}$ suffices for computing values (CbN / Haskell, CbV / OCAML)
$\bar{\beta}$-normal forms 1st approximations to normal forms in type-checking (Automath)

Implementation of $\bar{\beta}$

Definition

$\bar{\beta}$ is closed β: no variable in redex bound above it

Example ($\bar{\beta}$ is well-behaved)

$\lambda y.\mathfrak{I}\ {}_{\bar{\beta}}\!\leftarrow \lambda y.\mathfrak{I}\,\mathfrak{I}\ {}_{\bar{\beta}}\!\leftarrow \mathfrak{K}\,(\mathfrak{I}\,\mathfrak{I}) \rightarrow_{\bar{\beta}} \mathfrak{K}\,\mathfrak{I} \rightarrow_{\bar{\beta}} \lambda y.\mathfrak{I}$ (unique normal form)

Implementation of $\bar{\beta}$

Definition

$\bar{\beta}$ is closed β: no variable in redex bound above it

Example ($\bar{\beta}$ is well-behaved)

$\lambda y.\mathfrak{I} \;{}_{\bar{\beta}}\!\leftarrow \lambda y.\mathfrak{I}\,\mathfrak{I} \;{}_{\bar{\beta}}\!\leftarrow \mathfrak{K}\,(\mathfrak{I}\,\mathfrak{I}) \rightarrow_{\bar{\beta}} \mathfrak{K}\,\mathfrak{I} \rightarrow_{\bar{\beta}} \lambda y.\mathfrak{I}$

Lemma

$\bar{\beta}$ is confluent

Implementation of $\bar{\beta}$

Definition

$\bar{\beta}$ is closed β: no variable in redex bound above it

Lemma (👁-lifting)

if $M \to_{\bar{\beta}} N$ *and* $(t,\mathcal{R}) := (M)_{👁}$ *then* $t \to_{\mathcal{R}} s$ *for some* $(s,\mathcal{S}) := (N)_{👁}$ *with* $\mathcal{R} \supseteq \mathcal{S}$

only $\supseteq$ because redexes may be erased and constructors shared in compilation
closed redex nested in another redex is free in it, so a parameter in compilation
to formalise use higher-order term rewriting (combines λ with TRS)

Implementation of $\bar{\beta}$

Definition

$\bar{\beta}$ is closed β: no variable in redex bound above it

Corollary

homomorphism from $\bar{\beta}$-reduction from M to $\mathcal{R}$-reduction from t for $(t,\mathcal{R}) := (M)_{\text{👁}}$

Implementation of $\bar{\beta}$

Definition

$\bar{\beta}$ is closed β: no variable in redex bound above it

Corollary

homomorphism from $\bar{\beta}$-reduction from M to $\mathcal{R}$-reduction from t for $(t,\mathcal{R}) := (M)_{\text{👁}}$

what about decompilation?

Decompiling a term in a TRS into a λ-term

Definition (decompilation $(t)_\lambda$ homomorphism)

$C_i(t_1, \ldots, t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1, \ldots, x_n := t_1, \ldots, t_n]$ for rule $C_i(x_1, \ldots, x_n)\, x_0 \to r$ in $\mathcal{R}$

capture avoiding substitution (avoiding capture of free variables of the t_k)

Decompiling a term in a TRS into a λ-term

Definition (decompilation $(t)_\lambda$ homomorphism)

$C_i(t_1, \ldots, t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1, \ldots, x_n{:=}t_1, \ldots, t_n]$ for rule $C_i(x_1, \ldots, x_n)\, x_0 \to r$ in $\mathcal{R}$

Example (running; decompiling Church numeral $\underline{2}$)

$$\begin{array}{lrcl@{\qquad}rcl} \varrho_C : & C(x_1, x_2)\, x_0 & \to & x_1\,(x_2\, x_0) & C(t_1, t_2) & \mapsto & \lambda x_0.t_1\,(t_2\, x_0) \\ \varrho_D : & D\, x_0 & \to & C(x_0, x_0) & D & \mapsto & \lambda x_0 x_0'.x_0\,(x_0\, x_0') =_\alpha \underline{2} \end{array}$$

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

Decompiling a term in a TRS into a λ-term

Definition (decompilation $(t)_\lambda$ homomorphism)

$C_i(t_1,\ldots,t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1,\ldots,x_n{:=}t_1,\ldots,t_n]$ for rule $C_i(x_1,\ldots,x_n)\,x_0 \to r$ in $\mathcal{R}$

Example (running; decompiling Church numeral $\underline{2}$)

$$\begin{array}{lrcl@{\qquad}rcl} \varrho_C: & C(x_1,x_2)\,x_0 & \to & x_1\,(x_2\,x_0) & C(t_1,t_2) & \mapsto & \lambda x_0.t_1\,(t_2\,x_0) \\ \varrho_D: & D\,x_0 & \to & C(x_0,x_0) & D & \mapsto & \lambda x_0 x_0'.x_0\,(x_0\,x_0') =_\alpha \underline{2} \end{array}$$

$(D\,D)_\lambda = \underline{2}\,\underline{2}$

both $((\underline{2}\,\underline{2})_{\text{◉}})_\lambda =_\alpha \underline{2}\,\underline{2}$ and $((D\,D)_\lambda)_{\text{◉}} = D\,D$ up to naming combinators

Decompiling a term in a TRS into a λ-term

Definition (decompilation $(t)_\lambda$ homomorphism)

$C_i(t_1,\ldots,t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1,\ldots,x_n{:=}t_1,\ldots,t_n]$ for rule $C_i(x_1,\ldots,x_n)\,x_0 \to r$ in $\mathcal{R}$

Example (decompiling combinators)

$$
\begin{array}{rrcl@{\qquad}rcl}
\varrho_{S_2}\colon & S_2(x_1,x_2)\,x_0 & \to & (x_1\,x_0)\,(x_2\,x_0) & S_2(t_1,t_2) & \mapsto & \lambda x_0.(t_1\,x_0)\,(t_2\,x_0)\\
\varrho_{S_1}\colon & S_1(x_1)\,x_0 & \to & S_2(x_1,x_0) & S_1(t_1) & \mapsto & \lambda x_0 x_0'.(t_1\,x_0')\,(x_0\,x_0')\\
\varrho_{S}\colon & S\,x_0 & \to & S_1(x_0) & S & \mapsto & \lambda x_0 x_0' x_0''.(x_0\,x_0'')\,(x_0'\,x_0'') =_\alpha \mathfrak{S}\\
\varrho_{K_1}\colon & K_1(x_1)\,x_0 & \to & x_1 & K_1(t_1) & \mapsto & \lambda x_0.t_1\\
\varrho_{K}\colon & K\,x_0 & \to & K_1(x_0) & K & \mapsto & \lambda x_0 x_0'.x_0 =_\alpha \mathfrak{K}\\
\varrho_{I}\colon & I\,x_0 & \to & x_0 & I & \mapsto & \lambda x_0.x_0 =_\alpha \mathfrak{I}
\end{array}
$$

both $((\mathfrak{I}\,\mathfrak{K}\,\mathfrak{S})_{\circledcirc})_\lambda =_\alpha \mathfrak{I}\,\mathfrak{K}\,\mathfrak{S}$ and $((I\,K\,S)_\lambda)_{\circledcirc} = I\,K\,S$ up to naming combinators

Decompiling a term in a TRS into a λ-term

Definition (decompilation $(t)_\lambda$ homomorphism)

$C_i(t_1,\ldots,t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1,\ldots,x_n{:=}t_1,\ldots,t_n]$ for rule $C_i(x_1,\ldots,x_n)\,x_0 \to r$ in $\mathcal{R}$

Definition (applicative inductive interaction TRS, aij, 👁)

$\varrho_C : C(x_1,\ldots,x_{n_C})\,x_0 \to r$ with r built from @, $\vec{x}$ and $< C$ (well-founded $<$)

assume r linear in $x_1,\ldots,x_{n_C}$ preserving tree-order (x_0 may be replicated)
applicative since application @ only destructor
inductive since right-hand side r constructed from smaller constructors
interaction (Asperti & Laneve) since left-hand sides destructor–constructor

Decompiling a term in a TRS into a λ-term

Definition (decompilation $(t)_\lambda$ homomorphism)

$C_i(t_1, \ldots, t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1, \ldots, x_n := t_1, \ldots, t_n]$ for rule $C_i(x_1, \ldots, x_n)\, x_0 \to r$ in $\mathcal{R}$

Definition (applicative inductive interaction TRS, aij, 👁)

$\varrho_C : C(x_1, \ldots, x_{n_C})\, x_0 \to r$ with r built from $@$, $\vec{x}$ and $< C$ (well-founded $<$)

Example

Church numeral: $C < D$
combinators: $K_1 < K$ and $S_2 < S_1 < S$

Decompiling a term in a TRS into a λ-term

Definition (decompilation $(t)_\lambda$ homomorphism)

$C_i(t_1,\ldots,t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1,\ldots,x_n{:=}t_1,\ldots,t_n]$ for rule $C_i(x_1,\ldots,x_n)\,x_0 \to r$ in $\mathcal{R}$

Definition (applicative inductive interaction TRS, aij, 👁)

$\varrho_C : C(x_1,\ldots,x_{n_C})\,x_0 \to r$ with r built from $@$, $\vec{x}$ and $< C$ (well-founded $<$)

Theorem (implementation)

$((M)_👁)_\lambda =_\alpha M$ for any λ-term M and $((t)_\lambda)_👁 = t$ for any 👁-term t (up to naming) if $t \to_👁 s$ then $(t)_\lambda \to_\beta (s)_\lambda$ and if $M \to_\beta N$ then $(t)_👁 \to_👁 (s)_👁$

$(t[\vec{x}{:=}\vec{t}])_\lambda = (t)_\lambda[\vec{x}{:=}\overrightarrow{(t)_\lambda}]$ (substitution lemma)

Decompiling a term in a TRS into a λ-term

Definition (decompilation $(t)_\lambda$ homomorphism)

$C_i(t_1, \ldots, t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1, \ldots, x_n := t_1, \ldots, t_n]$ for rule $C_i(x_1, \ldots, x_n)\, x_0 \to r$ in $\mathcal{R}$

Theorem (implementation)

$((M)_{\circledcirc})_\lambda =_\alpha M$ *for any* λ*-term* M *and* $((t)_\lambda)_{\circledcirc} = t$ *for any* $\circledcirc$*-term* t *(up to naming)*
if $t \to_{\circledcirc} s$ *then* $(t)_\lambda \to_{\bar{\beta}} (s)_\lambda$ *and if* $M \to_{\bar{\beta}} N$ *then* $(t)_{\circledcirc} \to_{\circledcirc} (s)_{\circledcirc}$

Example

$D\,(D\,z_1) \to_{\circledcirc} C(D\,z_1, D\,z_1)$ by rule $\varrho_D : D\,x_0 \to C(x_0, x_0) \iff$
$(D\,(D\,z_1))_\lambda = (\lambda xy.x\,(x\,y))\,(\underline{2}\,z_1) \to_\beta \lambda y.\underline{2}\,z_1\,(\underline{2}\,z_1\,y) =_\alpha (C(D\,z_1, D\,z_1))_\lambda$

Decompiling a term in a TRS into a λ-term

Definition (decompilation $(t)_\lambda$ homomorphism)

$C_i(t_1, \ldots, t_n) \mapsto (\lambda x_0.(r)_\lambda)[x_1, \ldots, x_n := t_1, \ldots, t_n]$ for rule $C_i(x_1, \ldots, x_n)\, x_0 \to r$ in $\mathcal{R}$

Theorem (implementation)

$((M)_{\text{👁}})_\lambda =_\alpha M$ for any λ-term M and $((t)_\lambda)_{\text{👁}} = t$ for any 👁-term t (up to naming) if $t \to_{\text{👁}} s$ then $(t)_\lambda \to_{\bar{\beta}} (s)_\lambda$ and if $M \to_{\bar{\beta}} N$ then $(t)_{\text{👁}} \to_{\text{👁}} (s)_{\text{👁}}$

Corollary (implementation of $\bar{\beta}$ by 👁)

isomorphism between $\bar{\beta}$-reduction from M, $\mathcal{R}$-reduction from t, for $(t,\mathcal{R}) := (M)_{\text{👁}}$

Functional programming theory is TRS theory (not λ)

Idea (reductionist)

reduce the theory of closed β-reduction to that of 👁-reduction (✌ 05)

Functional programming theory is TRS theory

Idea (reductionist)

reduce the theory of closed β-reduction to that of 👁-reduction ( 05)

Example (some TRS theory that $\bar{\beta}$ theory can be reduced to)

- isn't FP theory about $\tilde{\beta}$? (weak β; redex not below a λ)

$$\lambda y.\mathfrak{I}\,\mathfrak{I} \;{}_{\tilde{\beta}}\!\leftarrow\; \mathfrak{K}\,(\mathfrak{I}\,\mathfrak{I}) \;\rightarrow_{\tilde{\beta}}\; \mathfrak{K}\,\mathfrak{I} \;\rightarrow_{\tilde{\beta}}\; \lambda y.\mathfrak{I}$$

(distinct $\tilde{\beta}$-normal forms)

Functional programming theory is TRS theory

Idea (reductionist)

reduce the theory of closed β-reduction to that of 👁-reduction (✌ 05)

Example (some TRS theory that $\bar{\beta}$ theory can be reduced to)

- isn't FP theory about $\tilde{\beta}$? $\tilde{\beta}$ subsystem of $\bar{\beta}$ ($\widetilde{👁}$ of 👁: not below constructor)
  $\lambda y.\mathfrak{I} \;{}_{\bar{\beta}}\!\leftarrow\; \lambda y.\mathfrak{I}\,\mathfrak{I} \;{}_{\tilde{\beta}}\!\leftarrow\; \mathfrak{K}\,(\mathfrak{I}\,\mathfrak{I}) \rightarrow_{\tilde{\beta}} \mathfrak{K}\,\mathfrak{I} \rightarrow_{\tilde{\beta}} \lambda y.\mathfrak{I}$ (same $\bar{\beta}$-normal form)

Functional programming theory is TRS theory

Idea (reductionist)

reduce the theory of closed β-reduction to that of 👁-reduction ( 05)

Example (some TRS theory that $\bar{\beta}$ theory can be reduced to)

- isn't FP theory about $\tilde{\beta}$? $\tilde{\beta}$ subsystem of $\bar{\beta}$ ( subsystem of 👁)
- $\bar{\beta}$ confluent?

Functional programming theory is TRS theory

Idea (reductionist)

reduce the theory of closed β-reduction to that of 👁-reduction (✌ 05)

Example (some TRS theory that $\bar{\beta}$ theory can be reduced to)

- isn't FP theory about $\tilde{\beta}$? $\tilde{\beta}$ subsystem of $\bar{\beta}$ ($\widetilde{👁}$ subsystem of 👁)
- $\bar{\beta}$ confluent? because 👁 is orthogonal (left-linear and non-overlapping)

Functional programming theory is OTRS theory

Idea (reductionist)

reduce the theory of closed β-reduction to that of 👁-reduction (✌ 05)

Example (some TRS theory that $\bar{\beta}$ theory can be reduced to)

- isn't FP theory about $\tilde{\beta}$? $\tilde{\beta}$ subsystem of $\bar{\beta}$ ($\widetilde{👁}$ subsystem of 👁)
- $\bar{\beta}$ confluent? because 👁 is orthogonal (OTRS)
- leftmost–outermost (lmo) strategy (hyper-)((weak-)head-)normalising?

Functional programming theory is OTRS theory

Idea (reductionist)

reduce the theory of closed β-reduction to that of 👁-reduction (✌ 05)

Example (some TRS theory that $\bar{\beta}$ theory can be reduced to)

- isn't FP theory about $\tilde{\beta}$? $\tilde{\beta}$ subsystem of $\bar{\beta}$ ($\widetilde{👁}$ subsystem of 👁)
- $\bar{\beta}$ confluent? because 👁 is orthogonal (OTRS)
- lmo strategy (hyper-)((weak-)head-)normalising? as 👁 is left-normal OTRS

Functional programming theory is OTRS theory

Idea (reductionist)

reduce the theory of closed β-reduction to that of 👁-reduction ( 05)

Example (some TRS theory that $\bar{\beta}$ theory can be reduced to)

- isn't FP theory about $\tilde{\beta}$? $\tilde{\beta}$ subsystem of $\bar{\beta}$ ($\widetilde{👁}$ subsystem of 👁)
- $\bar{\beta}$ confluent? because 👁 is orthogonal (OTRS)
- lmo strategy (hyper-)((weak-)head-)normalising? as 👁 is left-normal OTRS
- termgraph implementation of $\bar{\beta}$?

Functional programming theory is OTRS theory

Idea (reductionist)

reduce the theory of closed β-reduction to that of 👁-reduction ( 05)

Example (some TRS theory that $\bar{\beta}$ theory can be reduced to)

- isn't FP theory about $\tilde{\beta}$? $\tilde{\beta}$ subsystem of $\bar{\beta}$ ($\widetilde{👁}$ subsystem of 👁)
- $\bar{\beta}$ confluent? because 👁 is orthogonal (OTRS)
- lmo strategy (hyper-)((weak-)head-)normalising? as 👁 is left-normal OTRS
- termgraph implementation of $\bar{\beta}$? through termgraph implementation of 👁 here: Wadsworth's termgraph implementation of $\bar{\beta}$ refactored through known termgraph implementation of OTRS (*e.g.*, Staples 80s)

Functional programming theory is OTRS theory

Idea (reductionist)

reduce the theory of closed β-reduction to that of 👁-reduction (✌ 05)

Example (some TRS theory that $\bar{\beta}$ theory can be reduced to)

- isn't FP theory about $\tilde{\beta}$? $\tilde{\beta}$ subsystem of $\bar{\beta}$ ($\widetilde{👁}$ subsystem of 👁)
- $\bar{\beta}$ confluent? because 👁 is orthogonal (OTRS)
- lmo strategy (hyper-)((weak-)head-)normalising? as 👁 is left-normal OTRS
- termgraph implementation of $\bar{\beta}$? through termgraph implementation of 👁
- standardisation, external, needed, strategies, . . .

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

type schemata as usual

types of constructors defined by recursion via $<$; $@ : ((\tau \to \sigma) \times \tau) \to \sigma$ as usual

$(\vec{\tau})$ product type with length arity of C; by convention τ denotes $() \to \tau$ as usual

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Example (D; compilation of Church numeral $\underline{2}$)

C : $((\upsilon \to \sigma) \times (\tau \to \upsilon)) \to \tau \to \sigma$ since $x_1 : \upsilon \to \sigma, x_2 : \tau \to \upsilon, x_0 : \tau \vdash x_1\,(x_2\, x_0) : \sigma$

D : $(\sigma \to \sigma) \to \sigma \to \sigma$ since $x_0 : \sigma \to \sigma \vdash C(x_0, x_0) : \sigma \to \sigma$ and by convention

$D : ((\sigma \to \sigma) \to \sigma \to \sigma) \to (\sigma \to \sigma) \to \sigma \to \sigma$ for left D in compilation $D\,D$ of $\underline{2}$

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Example (simply typed CL$^\flat$ (xTCL in Goncharov & al. 24))

$$
\begin{array}{rcl}
I & : & \sigma \to \sigma \quad \text{since } x_0 : \sigma \vdash x_0 : \sigma \text{ and by convention} \\
K_1 & : & (\sigma) \to \tau \to \sigma \quad \text{since } x_1 : \sigma, x_0 : \tau \vdash x_1 : \sigma \\
K & : & \sigma \to \tau \to \sigma \quad \text{since } x_0 : \sigma \vdash K_1(x_0) : \tau \to \sigma \text{ and by convention} \\
S_2 & : & (\tau \to \upsilon \to \sigma) \times (\tau \to \upsilon) \to \tau \to \sigma \quad \text{since} \\
 & & x_1 : \tau \to \upsilon \to \sigma, x_2 : \tau \to \upsilon, x_0 : \tau \vdash (x_1\, x_0)\, (x_2\, x_0) : \sigma \\
S_1 & : & (\tau \to \upsilon \to \sigma) \to (\tau \to \upsilon) \to \tau \to \sigma, x_1 : \tau \to \upsilon \to \sigma, x_0 : \tau \to \upsilon \vdash S_2(x_1, x_0) : \tau \to \sigma \\
S & : & (\tau \to \upsilon \to \sigma) \to (\tau \to \upsilon) \to \tau \to \sigma \text{ as } x_0 : \tau \to \upsilon \to \sigma \vdash S_1(x_0) : (\tau \to \upsilon) \to \tau \to \sigma
\end{array}
$$

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Definition (strongly computable (SC; Tait, Girard))

t is SC if $t\,\vec{s}$ is terminating for all SC terms $\vec{s}$

SC inductive since $\vec{s}$ have types smaller than t
SC $\implies$ terminating (take $\vec{s}$ the empty vector)
application $t\,v$ is SC if subterms t and v both are SC (by rebracketing)

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Proof.

sufficient to show $h := C(\vec{t})$ is SC if $\vec{t}$ are SC.

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Proof.

sufficient to show $h := C(\vec{t})$ is SC if $\vec{t}$ are SC. by induction on C ordered by $<$.

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Proof.

sufficient to show $h := C(\vec{t})$ is SC if $\vec{t}$ are SC. by induction on C ordered by $<$.
$h\,\vec{s} \twoheadrightarrow h'\,\vec{s'} = C(\vec{x}^{\zeta'})\, x_0^{\zeta'}\, \vec{v'} \to r^{\zeta'}\, \vec{v'} \to \ldots$ shape of ∞ reduction for rule $C(\vec{x})\, x_0 \to r$.

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Proof.

sufficient to show $h := C(\vec{t})$ is SC if $\vec{t}$ are SC. by induction on C ordered by $<$.
$h\,\vec{s} \twoheadrightarrow h'\,\vec{s'} = C(\vec{x}^{\zeta'})\, x_0^{\zeta'}\, \vec{v'} \to r^{\zeta'}\, \vec{v'} \to \ldots$ shape of ∞ reduction for rule $C(\vec{x})\, x_0 \to r$.
$h\,\vec{s} = C(\vec{x}^{\zeta})\, x_0^{\zeta}\, \vec{v} \to r^{\zeta}\, \vec{v} \twoheadrightarrow r^{\zeta'}\, \vec{v'}$ for some ζ,$\vec{v}$ and $r^{\zeta}\, \vec{v}$ SC by IH so terminating ↯ □

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Remark (Goncharov & al. 24 on their new proof of termination of xTCL)

The reader should compare the above compact argument to the laborious original proof given in Example 15

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Remark (Goncharov & al. 24 on their new proof of termination of xTCL)

The reader should compare the above compact argument to the laborious original proof given in Example 15

marketing trick: first raise the price to then advertise a discount?

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Remark (Goncharov & al. 24 on their new proof of termination of xTCL)

The reader should compare the above compact argument to the laborious original proof given in Example 15

techniques interesting; apply to 👁; suggest cross-fertilisation with PRS theory

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Summary of Part I: Implementing closed β by orthogonal TRS (👁)

functional programming theory reduces to OTRS (👁) theory
OTRS theory (trees, substitution, algebra) simpler than theory for λ (binding . . .)

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Summary of Part I: Implementing closed β by orthogonal TRS (👁)

functional programming theory reduces to OTRS (👁) theory
OTRS theory (trees, substitution, algebra) simpler than theory for λ (binding . . .)

what about implementation of OTRS / 👁?

Simply typing (reductionist)

Definition (simply typed 👁)

$C : (\vec{\tau}) \to \tau \to \sigma$ if infer $\vec{x} : \vec{\tau}, x_0 : \tau \vdash r : \sigma$ for $\varrho_C : C(x_1, \ldots, x_n)\, x_0 \to r$

Lemma

simply typed 👁 is terminating and simply typedness preserved by compilation

Summary of Part I: Implementing closed β by orthogonal TRS (👁)

functional programming theory reduces to OTRS (👁) theory
OTRS theory (trees, substitution, algebra) simpler than theory for λ (binding . . .)

what about implementation of OTRS / 👁? up next
idea: factor through termgraph implementation of 👁 horizontal sharing; dags

Why OTRSs not a convincing model of computation?

Quote (Computability and λ-definability; Turing 37)

The identification of 'effectively calculable' functions with computable functions is possibly more convincing than an identification with the λ-definable or general recursive functions.

Why OTRSs not a convincing model of computation?

Quote (Computability and λ-definability; Turing 37)

The identification of 'effectively calculable' functions with computable functions is possibly more convincing than an identification with the λ-definable or general recursive functions.

My perspective

all (also OTRSs) are fine for effectivity (whether), not for efficiency (complexity) due to implicit replication. Turing Machines have convincing unit-time steps.

TMs are sequential others causal (asynchronous) models of computations

Why OTRSs not a convincing model of computation?

Quote (Computability and λ-definability; Turing 37)

The identification of 'effectively calculable' functions with computable functions is possibly more convincing than an identification with the λ-definable or general recursive functions.

My perspective

all (also OTRSs) are fine for effectivity (whether), not for efficiency (complexity) due to implicit replication. Turing Machines have convincing unit-time steps.

Plan for Part II: Implementing 👁 through termgraphs G👁 (Staples 80s)

argue termgraphs (G👁) have convincing unit-time steps. compile 👁 to G👁 with steps implementing contraction of family of 👁-redex-patterns (shared in G👁).

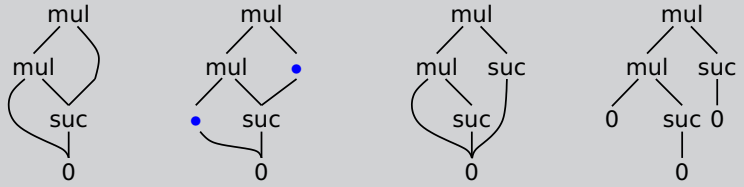
Termgraphs revisited

Termgraphs revisited

Definition (Termgraph)

finite directed acyclic graph (dag) with designated root. nodes labelled by signature extended with indirection • of arity 1. node labelled by symbol of arity n has 1 input port (number of in-edges) and n output ports (1 out-edge per port).

Example (four termgraphs for term $\mathrm{mul}(\mathrm{mul}(0,\mathrm{suc}(0)),\mathrm{suc}(0))$**)**



root implicit as input port without in-edge (here: mul); edges directed downward

Termgraphs revisited

Definition (Termgraph)

finite directed acyclic graph (dag) with designated root. nodes labelled by signature extended with indirection $\bullet$ of arity 1. node labelled by symbol of arity n has 1 input port (number of in-edges) and n output ports (1 out-edge per port).

Notations

$\boxed{t}$ to denote the termgraph of term t (a tree up to sharing of variables)
$\widehat{G}$ to denote the term of termgraph G (reading back from designated root)

mnemonic: terms are triangles $\triangle$ and graphs are squares $\square$

Termgraphs revisited

Definition (Termgraph)

finite directed acyclic graph (dag) with designated root. nodes labelled by signature extended with indirection $\bullet$ of arity 1. node labelled by symbol of arity n has 1 input port (number of in-edges) and n output ports (1 out-edge per port).

Notations

$\boxed{t}$ to denote the termgraph of term t (a tree up to sharing of variables)
$\widehat{G}$ to denote the term of termgraph G (reading back from designated root)

mnemonic: terms are triangles $\triangle$ and graphs are squares $\square$

note $\widehat{\boxed{t}} = t$ for any term t

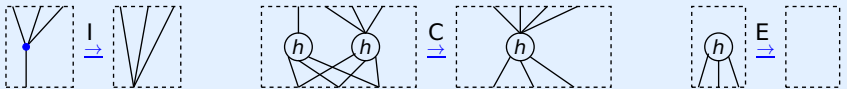
what about the converse? how does $\boxed{\widehat{G}}$ relate to G?

Termgraphs revisited

Definition (Termgraph)

finite directed acyclic graph (dag) with designated root. nodes labelled by signature extended with indirection $\bullet$ of arity 1. node labelled by symbol of arity n has 1 input port (number of in-edges) and n output ports (1 out-edge per port).

Definition (Ж; zhe, a substitution-calculus & van Raamsdonk)



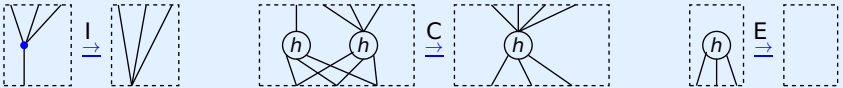
$\underline{\rightarrow}$-normal forms maximally shared (ATerm); $\underline{\leftrightarrow}$ denotes equivalence closure of $\underline{\rightarrow}$ $G \underline{\leftrightarrow} H$ iff $\widehat{G} = \widehat{H}$ (collecting garbage E, unsharing C^{-1}, directing I gives term-tree)

Termgraphs revisited

Definition (Termgraph)

finite directed acyclic graph (dag) with designated root. nodes labelled by signature extended with indirection $\bullet$ of arity 1. node labelled by symbol of arity n has 1 input port (number of in-edges) and n output ports (1 out-edge per port).

Definition (ж SC)



$\underline{\rightarrow}$-normal forms maximally shared; $\underline{\leftrightarrow}$ denotes equivalence closure of $\underline{\rightarrow}$

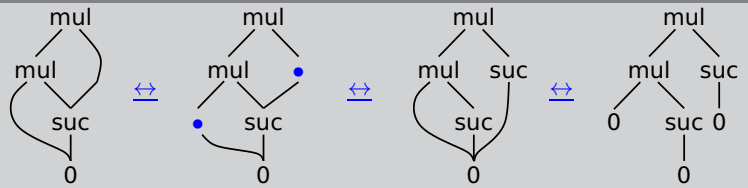
$G \underline{\leftrightarrow} H$ iff $\widehat{G} = \widehat{H}$ so $\boxed{\widehat{G}} \underline{\leftrightarrow} G$; terms as ж- / $\underline{\leftrightarrow}$-equivalence classes of termgraphs

Termgraphs revisited

Definition (Termgraph)

finite directed acyclic graph (dag) with designated root. nodes labelled by signature extended with indirection • of arity 1. node labelled by symbol of arity n has 1 input port (number of in-edges) and n output ports (1 out-edge per port).

Example (four termgraphs for term $\mathrm{mul}(\mathrm{mul}(0,\mathrm{suc}(0)),\mathrm{suc}(0))$**)**



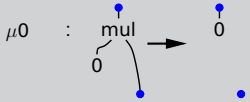
inserting indirections, removing them again and unsharing suc, then unsharing 0

Termgraph rewriting (TGRS) refactored through SC

Definition (TGRS)

rule has source (lhs) and target (rhs) termgraphs having same interface.

Example (of rule $\mu 0 : \mathsf{mul}(0, x_0) \to 0$)



interface of indirection nodes • corresponding to rule $\mu 0$ having arity 1

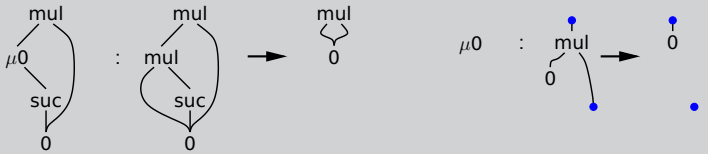
TGRS rules linear; all replication relegated to ж

Termgraph rewriting refactored through SC

Definition (TGRS)

rule has source (lhs) and target (rhs) termgraphs having same interface.
step replaces lhs by rhs modulo $\varkappa$ substitution calculus (& van Raamsdonk)

Example (of step $\mathsf{mul}(\mu 0(\mathsf{suc}(0)), 0) : \mathsf{mul}(\mathsf{mul}(0, \mathsf{suc}(0))), 0) \to \mathsf{mul}(0, 0)$**)**



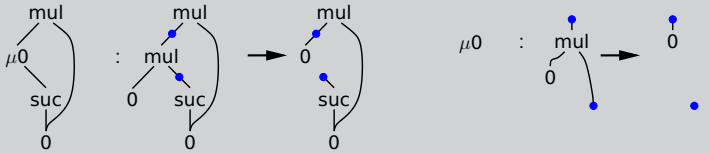
(i) unsharing 0 and inserting •, (ij) $\mu 0$-replacement, (iij) gc and sharing 0s

Termgraph rewriting refactored through SC

Definition (TGRS)

rule has source (lhs) and target (rhs) termgraphs having same interface.
step replaces lhs by rhs modulo ж SC (both in matching and substitution)

Example (of step $\mathsf{mul}(\mu 0(\mathsf{suc}(0)), 0) : \mathsf{mul}(\mathsf{mul}(0, \mathsf{suc}(0))), 0) \to \mathsf{mul}(0, 0)$**)**



(i) unsharing 0 and inserting •, (ij) $\mu 0$-replacement, (iij) gc and sharing 0s

multistep is termgraph over signature + rules (single / parallel step special case)

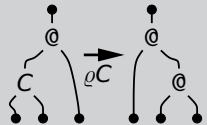
Termgraph rewriting refactored through SC

Definition (TGRS)

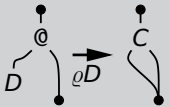
rule has source (lhs) and target (rhs) termgraphs having same interface.
step replaces lhs by rhs modulo ж

Example (G👁 rules corresponding to 👁 for Church numeral $\underline{2}$)

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



$\varrho_D : D\, x_0 \to C(x_0, x_0)$



Termgraph rewriting refactored through SC

Definition (TGRS)

rule has source (lhs) and target (rhs) termgraphs having same interface.
step replaces lhs by rhs modulo ж

Lemma (implementation of TRS by TGRS)

$\phi : t \multimap s$ *iff* $\Phi : G \multimap H$ (*multi*step; unsharing preserves redex-patterns *in* Φ)
$\phi : t \multimap\!\!\!\!\multimap s$ *iff* $\Phi : G \multimap\!\!\!\!\multimap H$ (*parallel step; unsharing preserves parallelism in* Φ)
for $t = \widehat{G}$, $s = \widehat{H}$ *and* $\phi = \widehat{\Phi}$

Proof.

by ж-unsharing (multi / parallel) termgraph to (multi / parallel) term-tree □

step $\Phi : G \to H$ yields step $\phi : G \to H$ only if rule in Φ has unique access path

Termgraph rewriting refactored through SC

Definition (TGRS)

rule has source (lhs) and target (rhs) termgraphs having same interface.
step replaces lhs by rhs modulo $\rtimes$

Lemma (implementation of TRS by TGRS)

$\phi : t \multimap s$ iff $\Phi : G \multimap H$ (multistep; unsharing preserves redex-patterns in Φ)
$\phi : t \multimap\!\!\!\!\multimap s$ iff $\Phi : G \multimap\!\!\!\!\multimap H$ (parallel step; unsharing preserves parallelism in Φ)
for $t = \widehat{G}$, $s = \widehat{H}$ and $\phi = \widehat{\Phi}$

since TRS not a convincing model of computation also TGRS not (by lemma)

Termgraph rewriting refactored through SC

Definition (TGRS)

rule has source (lhs) and target (rhs) termgraphs having same interface.
step replaces lhs by rhs modulo ж

Lemma (implementation of TRS by TGRS)

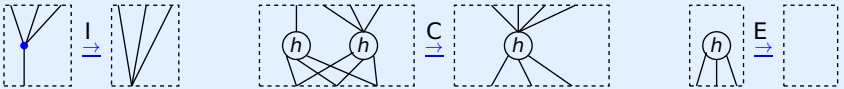
$\phi : t \multimap s$ iff $\Phi : G \multimap H$ (multistep; unsharing preserves redex-patterns in Φ)
$\phi : t \multimap\!\!\!\!\!\rightarrow s$ iff $\Phi : G \multimap\!\!\!\!\!\rightarrow H$ (parallel step; unsharing preserves parallelism in Φ)
for $t = \widehat{G}$, $s = \widehat{H}$ and $\phi = \widehat{\Phi}$

since TRS not a convincing model of computation also TGRS not
caused by modulo ж. idea: 👁 are simple so can largely abstain from modulo ж

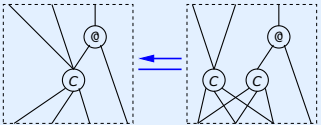
TGRS G👁 by constructor-unsharing (cu)

Definition (G👁 by cu; Wadsworth's admissibility)

recall $\underline{\rightarrow}$ rules gives maximal sharing in ж:



constructor-unsharing G👁 restricts ж to $\mathsf{I}, \mathsf{C}^{-1}$ needed to exhibit 👁-redexes:

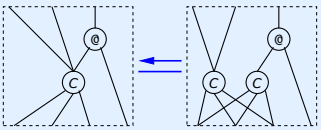


never unshare redex-patterns in termgraph G (only constructors of redexes)
sufficient to exhibit (all) redex-patterns

TGRS G👁 by constructor-unsharing (cu)

Definition (G👁)

constructor-unsharing G👁 restricts $\rightarrow_{\mathbb{K}}$ to $\mathsf{I}, \mathsf{C}^{-1}$ needed to exhibit 👁-redexes:



Lemma (shared steps do at least as much)

if $\phi : \widehat{G} \rightarrow_{👁} s$, *then* $\Phi : G \rightarrow_{G👁} H$ *for some parallel step* Φ *extending* ϕ ($\phi \sqsubseteq \widehat{\Phi}$)

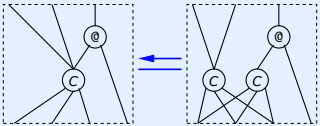
Proof.

exhibit the redex-pattern of ϕ in G to yield Φ. that may be shared however, giving rise to a parallel step $\widehat{\Phi} : \widehat{G} \twoheadrightarrow_{👁} v$ for some term v with $s \twoheadrightarrow_{👁} v$. □

TGRS G👁 by constructor-unsharing (cu)

Definition (G👁)

constructor-unsharing G👁 restricts $\mathfrak{K}$ to $\mathsf{I}, \mathsf{C}^{-1}$ needed to exhibit 👁-redexes:



Lemma (shared steps do at least as much)

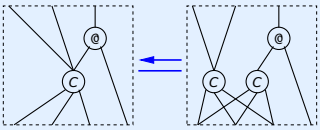
if $\phi : \widehat{G} \to_{👁} s$, *then* $\Phi : G \to_{G👁} H$ *for some parallel step* Φ *extending* ϕ ($\phi \sqsubseteq \widehat{\Phi}$)

contracting more is better in 👁 (in any orthogonal TRS)
for R a G👁-reduction from $\boxed{t}$, parallel 👁-reduction $\widehat{R}$ from t is a family-reduction

TGRS G👁 by constructor-unsharing (cu)

Definition (G👁)

constructor-unsharing G👁 restricts $\rtimes$ to $\mathsf{I}, \mathsf{C}^{-1}$ needed to exhibit 👁-redexes:



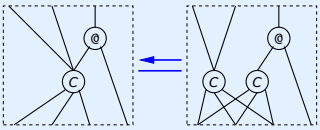
Example (family reduction, for simple OTRS)

for $\varrho : f(x) \to g(x,x)$ and $\vartheta : a \to b$, there is a family reduction
$f(a) \twoheadrightarrow g(a,a) \twoheadrightarrow g(b,b)$ (because both a-redex-patterns are shared), but
$f(a) \twoheadrightarrow g(a,a) \twoheadrightarrow g(b,a)$ is not (it would require unsharing a-redex-pattern)

TGRS G👁 by constructor-unsharing (cu)

Definition (G👁)

constructor-unsharing G👁 restricts $\rightarrow\!\!\!\!\!\!\kappa$ to $\mathsf{I}, \mathsf{C}^{-1}$ needed to exhibit 👁-redexes:



Example (family reduction, for simple OTRS)

for $\varrho : f(x) \to g(x,x)$ and $\vartheta : a \to b$, there is a family reduction
$f(a) \twoheadrightarrow g(a,a) \twoheadrightarrow g(b,b)$ (because both a-redex-patterns are shared), but
$f(a) \twoheadrightarrow g(a,a) \twoheadrightarrow g(b,a)$ is not (it would require unsharing a-redex-pattern)

termgraph rewriting induces family reduction on terms (shared redex-pattern)

G👁 is convincing model of computation

G👁 is convincing model of computation

Linear space

a G👁-reduction R from $\boxed{t}$ for some term t uses space $O(n)$ after n steps, since the number of symbols introduced by a step of R is bounded by the size of rhss.

constructor-unsharing creates ≤ 1 constructor (and out-edges; removes •s)

G👁 is convincing model of computation

Linear space

a G👁-reduction R from $\boxed{t}$ for some term t uses space $O(n)$ after n steps, since the number of symbols introduced by a step of R is bounded by the size of rhss.

Pseudo-linear time ($O(\alpha n)$ per step amortised; where α is Ackermann^{-1})

G👁 is convincing model of computation

Linear space

a G👁-reduction R from $\boxed{t}$ for some term t uses space $O(n)$ after n steps, since the number of symbols introduced by a step of R is bounded by the size of rhss.

Pseudo-linear time

how to exhibit an @–C redex-pattern in termgraph G starting from @-node?

G👁 is convincing model of computation

Linear space

a G👁-reduction R from $\boxed{t}$ for some term t uses space $O(n)$ after n steps, since the number of symbols introduced by a step of R is bounded by the size of rhss.

Pseudo-linear time

how to exhibit an @–C redex-pattern in termgraph G starting from @-node? indirection-nodes and edges between them constitute disjoint–set forest for union–find algorithm. exhibiting @–C corresponds to a find (causing path-compression). replacement of a lhs by a rhs may give rise to unions by uniting trees of indirection nodes, but number bounded by size of rhs.

G👁 is convincing model of computation

Linear space

a G👁-reduction R from $\boxed{t}$ for some term t uses space $O(n)$ after n steps, since the number of symbols introduced by a step of R is bounded by the size of rhss.

Pseudo-linear time

how to exhibit an @–C redex-pattern in termgraph G starting from @-node? indirection-nodes and edges between them constitute disjoint–set forest for union–find algorithm. exhibiting @–C corresponds to a find (causing path-compression). replacement of a lhs by a rhs may give rise to unions by uniting trees of indirection nodes, but number bounded by size of rhs.

illustrated next by run for running example $\underline{2}\,\underline{2}$ (naïve stacking union)

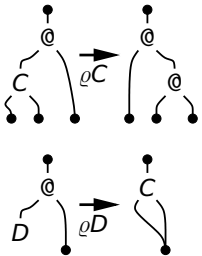
A constructor-unsharing run on $C(D\,z_1, D\,z_1)\,z_2$

👁 rules:

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

$\varrho_D(x_0) : D\,x_0 \to C(x_0, x_0)$

G👁 rules:

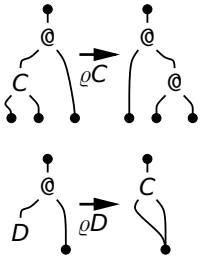


👁 rules:

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

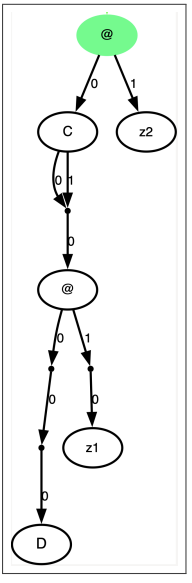
$\varrho_D(x_0) : D\, x_0 \to C(x_0, x_0)$

G👁 rules:



term:

$C(D\, z_1, D\, z_1)\, z_2$

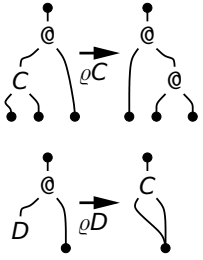


👁 rules:

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

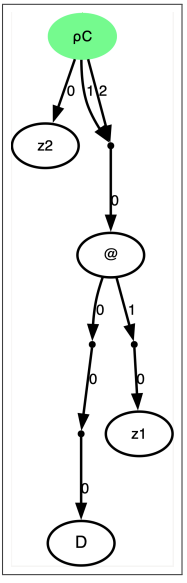
$\varrho_D(x_0) : D\, x_0 \to C(x_0, x_0)$

G👁 rules:



family step:

$\varrho_C(z_2, D\, z_1, D\, z_1) : C(D\, z_1, D\, z_1)\, z_2 \twoheadrightarrow D\, z_1\, (D\, z_1\, z_2)$

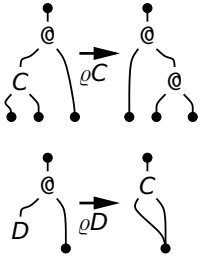


👁 rules:

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

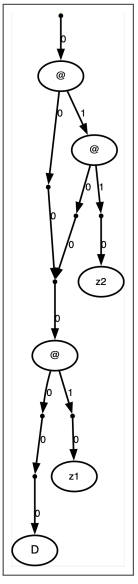
$\varrho_D(x_0) : D\, x_0 \to C(x_0, x_0)$

G👁 rules:



term:

$D\, z_1\, (D\, z_1\, z_2)$

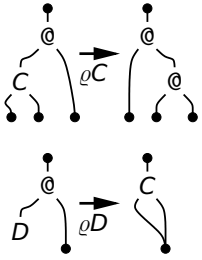


👁 rules:

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

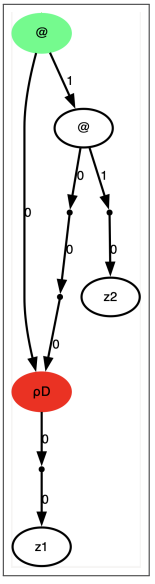
$\varrho_D(x_0) : D\, x_0 \to C(x_0, x_0)$

G👁 rules:



family step:

$\varrho_D(z_1)\,(\varrho_D(z_1)\, z_2) : D\, z_1\, (D\, z_1\, z_2) \twoheadrightarrow C(z_1, z_1)\, (C(z_1, z_1)\, z_2)$

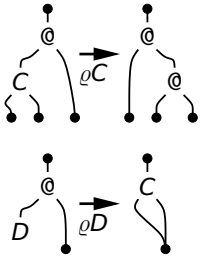


👁 rules:

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

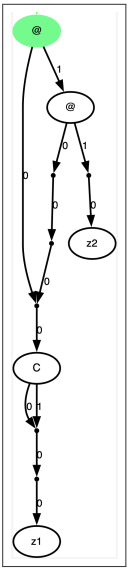
$\varrho_D(x_0) : D\, x_0 \to C(x_0, x_0)$

G👁 rules:



term:

$C(z_1, z_1)\, (C(z_1, z_1)\, z_2)$

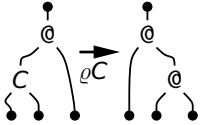


👁 rules:

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

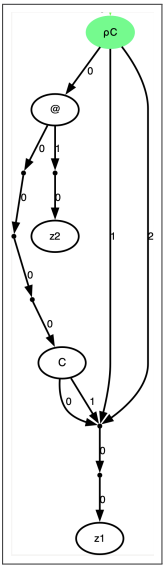
$\varrho_D(x_0) : D\, x_0 \to C(x_0, x_0)$

G👁 rules:



family step:

$\varrho_C(C(z_1, z_1)\, z_2, z_1, z_1) : C(z_1, z_1)\, (C(z_1, z_1)\, z_2) \twoheadrightarrow z_1\, (z_1\, (C(z_1, z_1)\, z_2))$

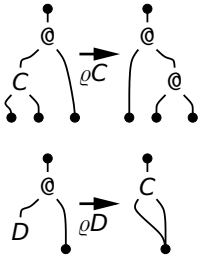


👁 rules:

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

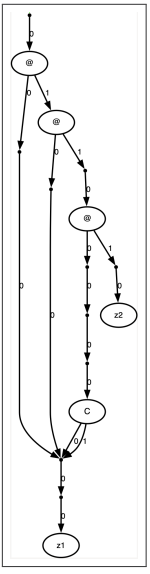
$\varrho_D(x_0) : D\, x_0 \to C(x_0, x_0)$

G👁 rules:



term:

$z_1\, (z_1\, (C(z_1, z_1)\, z_2))$

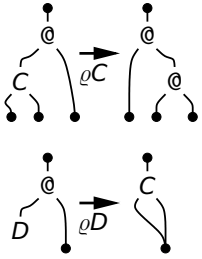


👁 rules:

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

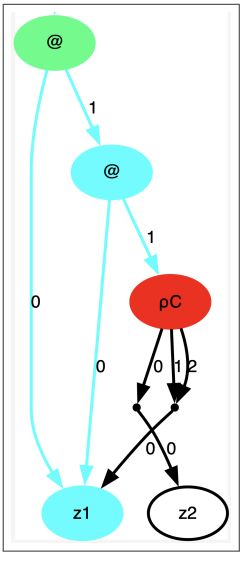
$\varrho_D(x_0) : D\, x_0 \to C(x_0, x_0)$

G👁 rules:



family step:

$z_1\, (z_1\, \varrho_C(z_2, z_1, z_1)) : z_1\, (z_1\, (C(z_1, z_1)\, z_2)) \twoheadrightarrow z_1\, (z_1\, z_1\, (z_1\, z_2))$

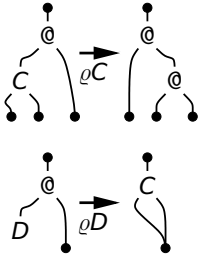


👁 rules:

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

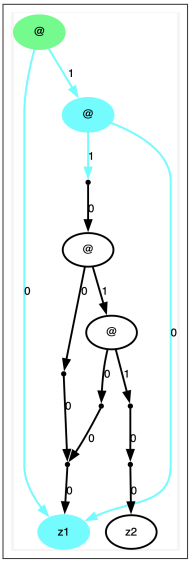
$\varrho_D(x_0) : D\, x_0 \to C(x_0, x_0)$

G👁 rules:



term:

$z_1\, (z_1\, z_1\, (z_1\, z_2))$

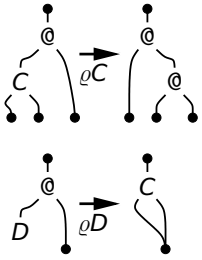


👁 rules:

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

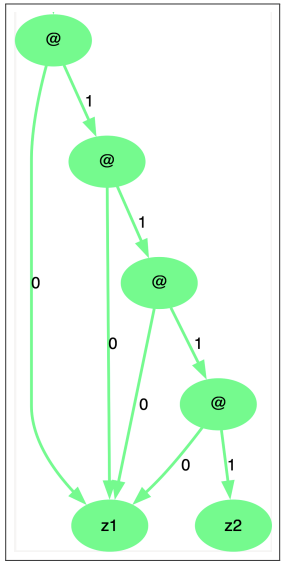
$\varrho_D(x_0) : D\, x_0 \to C(x_0, x_0)$

G👁 rules:



term:

$z_1\, (z_1\, z_1\, (z_1\, z_2))$



Steps as structures (Terese 03)

Idea for representation of steps

- rewrite steps as structures themselves in structure-rewriting
- objects (steps) as structures over signature of function (and rule) symbols
- src / tgt by homomorphism mapping rule symbols to lhs / rhs

Steps as structures

Idea for representation of steps

- rewrite steps as structures themselves in structure-rewriting
- objects (steps) as structures over signature of function (and rule) symbols
- src / tgt by homomorphism mapping rule symbols to lhs / rhs

Example (term rewrite step as term)

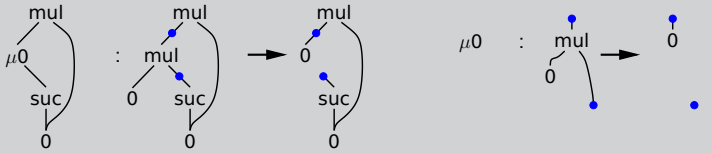
- $\varrho_D(z_1)\,(\varrho_D(z_1)\,z_2)$ is term representing (parallel) term rewrite step
- term over function symbols $@, D, C$ and rule symbol ϱ_D
- source $D\,z_1\,(D\,z_1\,z_2)$ / target $C(z_1,z_1)\,(C(z_1,z_1)\,z_2)$ by mapping ϱ_D in step to lhs $D\,x_0$ / rhs $C(x_0,x_0)$ for rule $\varrho_D(x_0) : D\,x_0 \to C(x_0,x_0)$

Steps as structures

Idea for representation of steps

- rewrite steps as structures themselves in structure-rewriting
- objects (steps) as structures over signature of function (and rule) symbols
- src / tgt by homomorphism mapping rule symbols to lhs / rhs

Example (termgraph rewrite step as termgraph)



step as termgraph over function symbols $\mathrm{mul}, \mathrm{suc}, 0$ and rule symbol $\mu 0$

Steps as structures

Idea for representation of steps

- rewrite steps as structures themselves in structure-rewriting
- objects (steps) as structures over signature of function (and rule) symbols
- src / tgt by homomorphism mapping rule symbols to lhs / rhs

Canonicity of representation

canonical (concrete) way of representing steps based on homomorphism
no extra-curricular *ad hoc* choices (contexts, substitutions, positions, . . .)

Steps as structures

Idea for representation of steps

- rewrite steps as structures themselves in structure-rewriting
- objects (steps) as structures over signature of function (and rule) symbols
- src / tgt by homomorphism mapping rule symbols to lhs / rhs

Canonicity of representation

canonical (concrete) way of representing steps based on homomorphism
no extra-curricular *ad hoc* choices (contexts, substitutions, positions, . . .)

my experience: works well across the board (theory, proofs, implementation)
e.g., prove properties of term rewrite steps by induction on terms, and
in formalisation build step properties on term properties (TRS Kirk & Middeldorp)

Who’s your family depends on the way of creation

Family intuition (Lévy 78)

given a reduction $R : t_0 \to t_1 \to \ldots$ two redex-pattern occurrences in t_i belong to the same family if they have the same creation history.

that’s vague, so Lévy gave 3 characterisations via labelling, zig-zag, extraction
he would have preferred 4^{th} via sharing in graph rewriting (Lamping, Kathail 90)

Who’s your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \twoheadrightarrow \widehat{G_1} \twoheadrightarrow \ldots$ in same family if represented by same in graph G_i

Who's your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \multimap \widehat{G_1} \multimap \ldots$ in same family if represented by same in graph G_i

idea: shared redex-pattern occurrences share their creation-history
history-dependent: a in $f(a,a) \to f(a,a)$ for rules $a \to \ldots$ and $f(x,y) \to f(x,x)$??
OGTRS / G👁 termgraphs: only horizontal sharing so $\multimap$-steps in fact $\multimap\!\!\!\!\to$ steps

Who's your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \multimap \widehat{G_1} \multimap \ldots$ in same family if represented by same in graph G_i

Example

D-redex-pattern occurrences in same family in running example run (page 16)
$\varrho_D(z_1)\,(\varrho_D(z_1)\,z_2) : D\,z_1\,(D\,z_1\,z_2) \twoheadrightarrow C(z_1,z_1)\,(C(z_1,z_1)\,z_2)$

Who's your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \multimap \widehat{G_1} \multimap \ldots$ in same family if represented by same in graph G_i

Example

a's in same family in s, but not in same family in t in
$t := f(a, a) \to f(a, a) =: s$ in sharing TGRS, for rules $a \to \ldots$ and $f(x, y) \to f(x, x)$

Who’s your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \multimap \widehat{G_1} \multimap \ldots$ in same family if represented by same in graph G_i

Example (family depending on notion of reduction, implementation)

let $M := (\lambda x.x\,(x\,\mathfrak{S}))\,\lambda y.N$, where N is some (closed) redex, say $\mathfrak{I}\,\mathfrak{K}$

Who’s your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \twoheadrightarrow \widehat{G_1} \twoheadrightarrow \ldots$ in same family if represented by same in graph G_i

Example (family depending on notion of reduction, implementation)

let $M := (\lambda x.x\,(x\,\mathfrak{S}))\,\lambda y.N$, where N is some (closed) redex, say $\mathfrak{I}\,\mathfrak{K}$
after $M \to (\lambda y.N)\,((\lambda y.N)\,\mathfrak{S}) \to (\lambda y.N)\,N =: L$, should occurrences of N be shared?

Who's your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \twoheadrightarrow \widehat{G_1} \twoheadrightarrow \ldots$ in same family if represented by same in graph G_i

Example (family depending on notion of reduction, implementation)

let $M := (\lambda x.x\,(x\,\mathfrak{S}))\,\lambda y.N$, where N is some (closed) redex, say $\mathfrak{I}\,\mathfrak{K}$
after $M \to (\lambda y.N)\,((\lambda y.N)\,\mathfrak{S}) \to (\lambda y.N)\,N =: L$, should occurrences of N be shared?
(yes) for β-steps implemented through sharing graphs; Lamping 90:
contraction of $(\lambda y.N)\,\mathfrak{S}$ keeps redex N shared in L
the white box of the function $\lambda y.N$ is opened in 2nd graph step

Who's your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \multimap \widehat{G_1} \multimap \ldots$ in same family if represented by same in graph G_i

Example (family depending on notion of reduction, implementation)

let $M := (\lambda x.x\,(x\,\mathfrak{S}))\,\lambda y.N$, where N is some (closed) redex, say $\mathfrak{I}\,\mathfrak{K}$
after $M \to (\lambda y.N)\,((\lambda y.N)\,\mathfrak{S}) \to (\lambda y.N)\,N =: L$, should occurrences of N be shared?
(no) for $\bar{\beta}$-steps implemented through termgraphs G👁:
contraction of $(\lambda y.N)\,\mathfrak{S}$ unshares (the constructor representing) function $\lambda y.N$
the function $\lambda y.N$ is black box (constructor) hiding redex N in termgraph of L

Who's your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \twoheadrightarrow \widehat{G_1} \twoheadrightarrow \ldots$ in same family if represented by same in graph G_i

Example (family depending on notion of reduction, implementation)

let $M := (\lambda x.x\,(x\,\mathfrak{S}))\,\lambda y.N$, where N is some (closed) redex, say $\mathfrak{I}\,\mathfrak{K}$
after $M \to (\lambda y.N)\,((\lambda y.N)\,\mathfrak{S}) \to (\lambda y.N)\,N =: L$, should occurrences of N be shared?

upshot: family is intensional property depending on rules; from just the step $f(a) \to f(a,a)$ can't tell whether a's in same family in $f(a,a)$ or not

Who's your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \multimap \widehat{G_1} \multimap \ldots$ in same family if represented by same in graph G_i

Example (family depending on notion of reduction, implementation)

let $M := (\lambda x.x\,(x\,\mathfrak{S}))\,\lambda y.N$, where N is some (closed) redex, say $\mathfrak{I}\,\mathfrak{K}$
after $M \to (\lambda y.N)\,((\lambda y.N)\,\mathfrak{S}) \to (\lambda y.N)\,N =: L$, should occurrences of N be shared?

upshot: family is intensional property depending on rules; from just the step $f(a) \to f(a,a)$ can't tell whether a's in same family in $f(a,a)$ or not
for rule $f(a) \to f(a,a)$ no, but for rule $f(x) \to f(x,x)$ yes

Who's your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \multimap \widehat{G_1} \multimap \ldots$ in same family if represented by same in graph G_i

Example (family depending on notion of reduction, implementation)

let $M := (\lambda x.x\,(x\,\mathfrak{S}))\,\lambda y.N$, where N is some (closed) redex, say $\mathfrak{I}\,\mathfrak{K}$
after $M \to (\lambda y.N)\,((\lambda y.N)\,\mathfrak{S}) \to (\lambda y.N)\,N =: L$, should occurrences of N be shared?

upshot: family is intensional property depending on rules
induced by residuals of redex-patterns (Church & Rosser 36, Newman 42, . . .)
residuals factor through descendants of symbols (Morris, ⩛ proofterm algebra)

Who's your family depends on the way of creation

Definition (Family for a given graph implementation)

for graph reduction $R : \boxed{t_0} =: G_0 \to G_1 \to \ldots$ redex-pattern occurrences in term $t_i := \widehat{G_i}$ in $\widehat{R} : \widehat{G_0} \multimap \widehat{G_1} \multimap \ldots$ in same family if represented by same in graph G_i

Example (family depending on notion of reduction, implementation)

let $M := (\lambda x.x\,(x\,\mathfrak{S}))\,\lambda y.N$, where N is some (closed) redex, say $\mathfrak{I}\,\mathfrak{K}$
after $M \to (\lambda y.N)\,((\lambda y.N)\,\mathfrak{S}) \to (\lambda y.N)\,N =: L$, should occurrences of N be shared?

upshot: family is intensional property depending on rules
induced by residuals of redex-patterns (Church & Rosser 36, Newman 42, . . .)
residuals factor through descendants of symbols (Morris,  proofterm algebra)
slogan: tell me your reduction history and rules, and i'll tell you your family

Random descent for G👁

Theorem (in essence Staples 80)

G👁 has random descent (Newman 42, 🕊 07): if G reduces to a normal form, then G is complete (confluent and terminating) and all reductions from G end in the same normal form, in the same number of steps

Random descent for G👁

Theorem (in essence Staples 80)

G👁 has random descent: if G reduces to a normal form, then G is complete and all reductions from G end in the same normal form, in the same number of steps

Proof.

G👁 is constructor-unsharing and orthogonal. redex-pattern occurrences are non-overlapping and commute directly: $\leftarrow \cdot \rightarrow \subseteq = \cup (\rightarrow \cdot \leftarrow)$ then by 07 $\square$

termgraph rewriting is linear; term rewriting is replicating

Random descent for G👁

Theorem (in essence Staples 80)

G👁 has random descent: if G reduces to a normal form, then G is complete and all reductions from G end in the same normal form, in the same number of steps

Corollary

G👁 reduction is optimal (if normal form exists, reached in least number of steps)

trivially so by random descent; one can descend randomly toward normal form

Random descent for G👁

Theorem ()

G👁 has random descent: if G reduces to a normal form, then G is complete and all reductions from G end in the same normal form, in the same number of steps

Corollary

G👁 reduction is optimal

Caveat

G👁 reduction to normal form includes reduction inside garbage

Random descent for G👁

Theorem ()

G👁 has random descent: if G reduces to a normal form, then G is complete and all reductions from G end in the same normal form, in the same number of steps

Caveat

G👁 reduction to normal form includes reduction inside garbage

Example

for rule $f(x) \to a$, normalising termgraph $\boxed{f(t)}$ in TGRS requires normalising t, though t could be garbage collected after $f(t) \to a$ (unreachable from root)

Random descent for G👁

Theorem ()

G👁 has random descent: if G reduces to a normal form, then G is complete and all reductions from G end in the same normal form, in the same number of steps

Lemma

leftmost–outermost reduction $\rightarrow_{lmo}$ is optimal for reduction to rrnf (root-reachable normal form; subgraph reachable from root in normal form)

Random descent for G👁

Theorem ()

G👁 has random descent: if G reduces to a normal form, then G is complete and all reductions from G end in the same normal form, in the same number of steps

Lemma

leftmost–outermost reduction $\to_{lmo}$ is optimal for reduction to rrnf

Proof.

$\to_{\mathsf{lmo}}$ is better than $\to$ by OLCOM($\to_{\mathsf{lmo}}$,$\to$): ${}_{\mathsf{lmo}}\!\leftarrow \cdot \to \subseteq = \cup \left(\to^{=} \cdot {}_{\mathsf{lmo}}\!\leftarrow\right)$ □

Random descent for G👁

Theorem ()

G👁 has random descent: if G reduces to a normal form, then G is complete and all reductions from G end in the same normal form, in the same number of steps

Lemma

leftmost–outermost reduction $\to_{lmo}$ is optimal for reduction to rrnf

Proof.

$\to_{\mathsf{lmo}}$ is better than $\to$ by OLCOM($\to_{\mathsf{lmo}}$,$\to$): ${}_{\mathsf{lmo}}\!\leftarrow \cdot \to \subseteq = \cup \left(\to^{=} \cdot {}_{\mathsf{lmo}}\!\leftarrow\right)$ □

ordered local commutation (🕊 07) for comparing strategies by local peaks at ISR 2017 for program improvement; reinvented by Hamana & Muroya 24

Random descent for G👁

Theorem ()

G👁 has random descent: if G reduces to a normal form, then G is complete and all reductions from G end in the same normal form, in the same number of steps

Lemma

leftmost–outermost reduction $\to_{lmo}$ is optimal for reduction to rrnf

Proof.

$\to_{\mathsf{lmo}}$ is better than $\to$ by OLCOM($\to_{\mathsf{lmo}}$,$\to$): ${}_{\mathsf{lmo}}\!\leftarrow \cdot \to \subseteq = \cup \left(\to^{=} \cdot {}_{\mathsf{lmo}}\!\leftarrow\right)$ □

next: lmo known to have good properties on terms; what about on termgraphs?

Pseudo-linearity of the leftmost–outermost strategy

Contraction vs. Finding

nice that redex-patterns can be exhibited and contracted in (pseudo-)linear time, but what about selecting and finding them?

Pseudo-linearity of the leftmost–outermost strategy

Contraction vs. Finding

nice that redex-patterns can be exhibited and contracted in (pseudo-)linear time, but what about selecting and finding them?

question because TRS / TGRS causal MoC unlike sequential TMs, determining the next thing to do may not be immediate from the previous thing done

Pseudo-linearity of the leftmost–outermost strategy

Contraction vs. Finding

nice that redex-patterns can be exhibited and contracted in (pseudo-)linear time, but what about selecting and finding them?

question because TRS / TGRS causal MoC unlike sequential TMs, determining the next thing to do may not be immediate from the previous thing done
selection is governed by strategy whose complexity reflects how complex following it is (finding redex-pattern according to the strategy)

Pseudo-linearity of the leftmost–outermost strategy

Definition (for rewrite system $\rightarrow$)

strategy is a subsystem of $\rightarrow$ having same sets of objects and normal forms as $\rightarrow$

don’t allow to stop computation prematurely; no change of semantics

Pseudo-linearity of the leftmost–outermost strategy

Definition (for rewrite system $\rightarrow$)

strategy is a subsystem of $\rightarrow$ having same sets of objects and normal forms as $\rightarrow$

Lemma

leftmost–outermost (lmo) strategy is pseudo-linear in length of reduction for G👁

for G👁 normal form s has shape $C(\vec{s})$ or $x\,\vec{s}$ for normal forms $\vec{s}$

Pseudo-linearity of the leftmost–outermost strategy

Definition (for rewrite system $\to$)

strategy is a subsystem of $\to$ having same sets of objects and normal forms as $\to$

Lemma

lmo strategy is pseudo-linear in length of reduction for G👁

Proof idea (exploit locality of causality in graph rewriting / G👁; diff).

implementable by dipper data-structure dipping down the head-spine toward a constructor or variable, contracting redex-patterns, followed by proceeding recursively along subterms $\vec{t}$ (for $C(\vec{t})$) or vertebrae $\vec{t}$ (for $x\,\vec{t}$)

Pseudo-linearity of the leftmost–outermost strategy

Definition (for rewrite system $\rightarrow$)

strategy is a subsystem of $\rightarrow$ having same sets of objects and normal forms as $\rightarrow$

Lemma

lmo strategy is pseudo-linear in length of reduction for G👁

Proof idea (exploit locality of causality in graph rewriting / G👁; diff).

implementable by dipper data-structure dipping down the head-spine toward a constructor or variable, contracting redex-patterns, followed by proceeding recursively along subterms or vertebrae
linear number of visits of symbols other than indirections by dipper, so pseudo-linear by finding the arguments of these symbols. □

Properties of lefmost–outermost strategy on G

Leftmost–outermost on terms vs. termgraphs

nice that lmo is pseudo-linear in termgraph rewriting, but does it have good properties and if so, do these transfer to term rewriting?

Properties of lefmost–outermost strategy on G👁

Leftmost–outermost on terms vs. termgraphs

nice that lmo is pseudo-linear in termgraph rewriting, but does it have good properties and if so, do these transfer to term rewriting?
e.g., is lmo (hyper-)((weak) head-)normalising on G👁?
if so, does that transfer to the same for 👁?

Properties of lefmost–outermost strategy on G👁

Leftmost–outermost on terms vs. termgraphs

nice that lmo is pseudo-linear in termgraph rewriting, but does it have good properties and if so, do these transfer to term rewriting?
e.g., is lmo (hyper-)((weak) head-)normalising on G👁?
if so, does that transfer to the same for 👁?

reductionist idea: exploit lmo is (hyper-)((weak) head-)normalising on 👁 and on β because they are left-normal orthogonal PRSs.

Properties of lefmost–outermost strategy on G👁

reductionist idea: exploit lmo is (hyper-)((weak) head-)normalising on 👁 and on β because they are left-normal orthogonal PRSs.
family step for a redex-pattern does at least as much as step; specialises to lmo:

Lemma

if $\Phi : G \to H$ is lmo in G👁, then family step $\widehat{\Phi} : \widehat{G} \twoheadrightarrow \widehat{H}$ contracts the lmo redex-pattern in the term $\widehat{G}$ (and possibly others)

Properties of lefmost–outermost strategy on G👁

reductionist idea: exploit lmo is (hyper-)((weak) head-)normalising on 👁 and on β because they are left-normal orthogonal PRSs.

Lemma

if $\Phi : G \to H$ is lmo in G👁, then family step $\widehat{\Phi} : \widehat{G} \twoheadrightarrow \widehat{H}$ contracts the lmo redex-pattern in the term $\widehat{G}$

Corollary

optimal implementation of family reduction of $\overline{\beta}$

lmo family rewriting is optimal among $\twoheadrightarrow_{\overline{\beta}}$-family-strategies on λ-terms

Properties of lefmost–outermost strategy on G👁

reductionist idea: exploit lmo is (hyper-)((weak) head-)normalising on 👁 and on β because they are left-normal orthogonal PRSs.

Lemma

if $\Phi : G \to H$ is lmo in G👁, then family step $\widehat{\Phi} : \widehat{G} \twoheadrightarrow \widehat{H}$ contracts the lmo redex-pattern in the term $\widehat{G}$

Corollary

lmo closed β-reduction ($\bar{\beta}$) to (weak head) normal form implemented by G👁 in space linear and time pseudo-linear in length of the reduction

Properties of lefmost–outermost strategy on G👁

reductionist idea: exploit lmo is (hyper-)((weak) head-)normalising on 👁 and on β because they are left-normal orthogonal PRSs.

Corollary

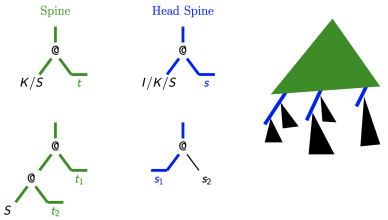
lmo weak β-reduction ($\tilde{\beta}$) to (weak head) normal form implemented by G👁 in space linear and time pseudo-linear in length of the reduction

Generalising from lmo to spine strategies for 👁

Spine strategy

Definition

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



Example

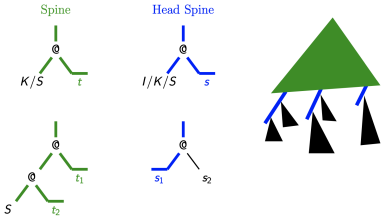
S(SISI)(K(IK))

Generalising from lmo to spine strategies for

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

Generalising from lmo to spine strategies for 👁

Definition (of spine for 👁-terms)

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

Generalising from lmo to spine strategies for 👁

Definition (of spine for 👁-terms)

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

Generalising from lmo to spine strategies for 👁

Definition (of spine for 👁-terms)

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

Part II: Implementing 👁 through termgraphs G👁

Summary of Part II

- constructor-unsharing (cu) termgraph rewriting G👁 is a convincing causal (asynchronous) model of computation
- lmo family-$\twoheadrightarrow_{\bar{\beta}}$ optimal (needed) for family-$\twoheadrightarrow_{\bar{\beta}}$ on λ-terms
- closed ($\bar{\beta}$) and weak ($\tilde{\beta}$) reduction of λ-terms can be implemented by compiling to termgraph rewriting G👁, factoring through term rewriting 👁
- any spine family strategy is (weakly head) normalising and can be implemented in linear space and pseudo-linear time through spine in G👁

Part II: Implementing 👁 through termgraphs G👁

Summary of Part II

- constructor-unsharing (cu) termgraph rewriting G👁 is a convincing causal (asynchronous) model of computation
- lmo family-$\twoheadrightarrow_{\bar{\beta}}$ optimal (needed) for family-$\twoheadrightarrow_{\bar{\beta}}$ on λ-terms
- closed ($\bar{\beta}$) and weak ($\tilde{\beta}$) reduction of λ-terms can be implemented by compiling to termgraph rewriting G👁, factoring through term rewriting 👁
- any spine family strategy is (weakly head) normalising and can be implemented in linear space and pseudo-linear time through spine in G👁

what about implementing (full) β itself?

Part II: Implementing 👁 through termgraphs G👁

Summary of Part II

- constructor-unsharing (cu) termgraph rewriting G👁 is a convincing causal (asynchronous) model of computation
- lmo family-$\twoheadrightarrow_{\bar{\beta}}$ optimal (needed) for family-$\twoheadrightarrow_{\bar{\beta}}$ on λ-terms
- closed ($\bar{\beta}$) and weak ($\tilde{\beta}$) reduction of λ-terms can be implemented by compiling to termgraph rewriting G👁, factoring through term rewriting 👁
- any spine family strategy is (weakly head) normalising and can be implemented in linear space and pseudo-linear time through spine in G👁

what about implementing (full) β itself? next up
idea: factor through special spine reduction G👁, adjoining explicit α

Factoring β through $\tilde{\beta}$ and explicit α

Idea for (full) β

if head-spine is a constructor $C(t_1, \ldots, t_n)$ then some β-redexes may be hidden by C. then decompile C. can be implemented for 👁 by printing λx for x fresh, then recursively compute $C(t_1, \ldots, t_n)\, x$

factoring β through $\tilde{\beta}$ old idea: De Bruijn, Coquand, Grégoire & Leroy

Factoring β through $\tilde{\beta}$ and explicit α

Idea for (full) β

if head-spine is a constructor $C(t_1, \ldots, t_n)$ then some β-redexes may be hidden by C. then decompile C. can be implemented for 👁 by printing λx for x fresh, then recursively compute $C(t_1, \ldots, t_n)\, x$

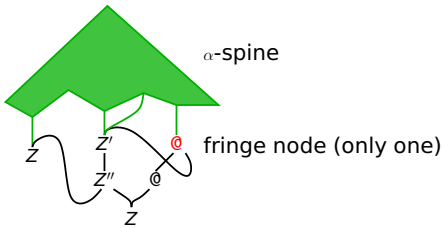
Caveat for termgraphs

if constructor is shared it may be head-spine for one but non-head-spine for others. risk of premature decompilation (unsharing; breaking space linearity)

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)

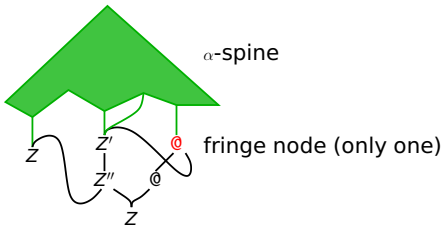
reduce head spines from fringe nodes to hsnf and recurse on spine vertebrae

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

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**

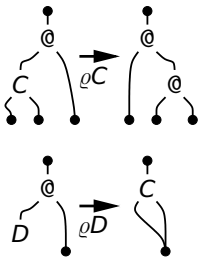
idea: a combinator on fringe / α-spine is a λ-abstraction (in the β-nf), so may recurse on its body, effectuated in 👁 by suppling a fresh variable

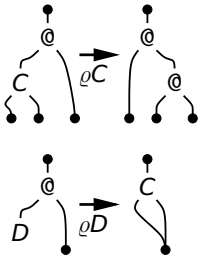
Example α-spine reduction (Java code $\Rightarrow$ dot $\Rightarrow$ graphs)

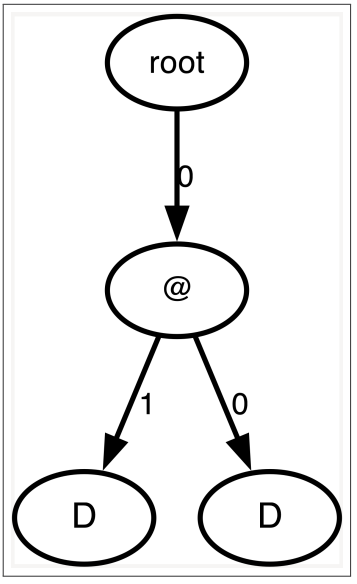
recall 👁-rules:

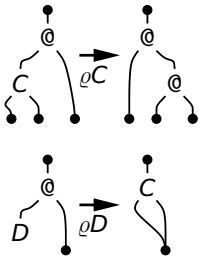
$\varrho_C : C(x_1, x_2)\, x_0 \to x_1\, (x_2\, x_0)$
$\varrho_D : D\, x_0 \to C(x_0, x_0)$

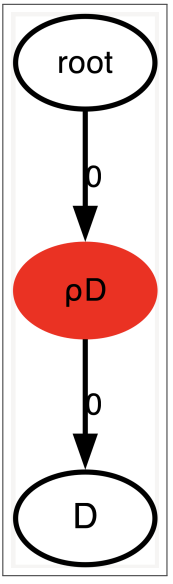
and termgraph rules:

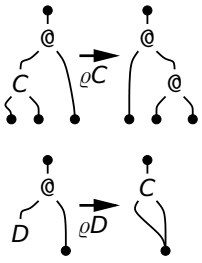


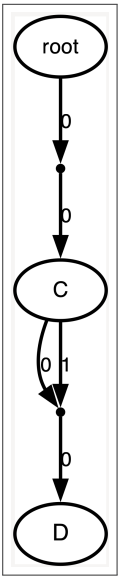

@
C
ϱC
@
@
@
D
ϱD
C

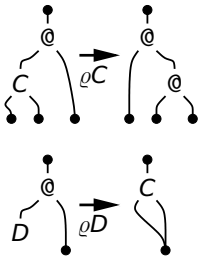

root
0
@
1
0
D
D

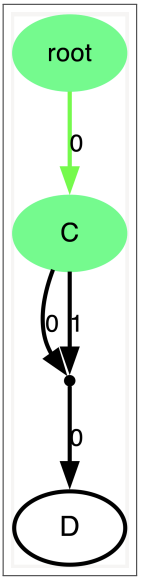

@
C
ϱC
@
@
@
D
ϱD
C

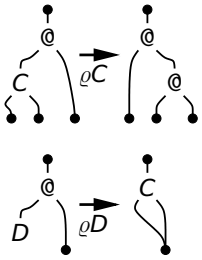

root
0
ρD
0
D

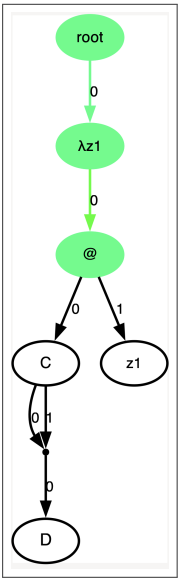

@
C
ϱC
@
@
@
D
ϱD
C

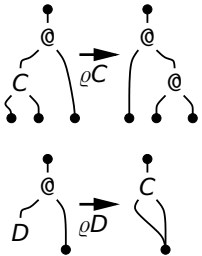

root
0
0
C
0
1
0
D

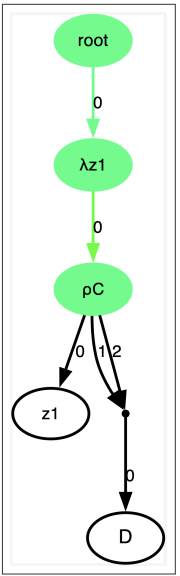

@
C
ϱC
@
@
@
D
ϱD
C

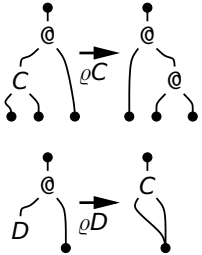

root
0
C
0
1
0
D

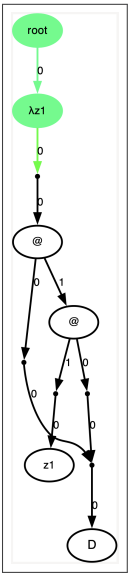

@
C
@
@
@C
@
D
@
C
@D

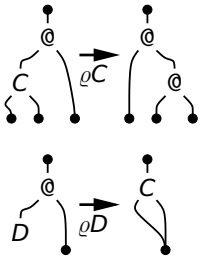

root
0
λz1
0
@
0
1
C
z1
0
1
0
D

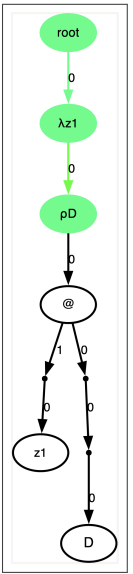

@
C
ϱC
@
@
@
D
ϱD
C

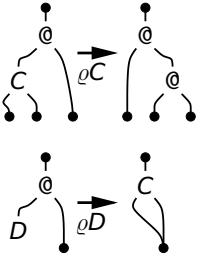

root
0
λz1
0
ρC
0
1
2
z1
0
D

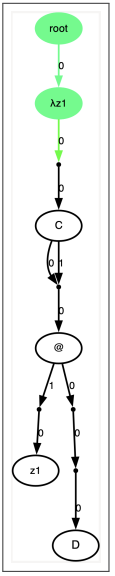

@
C
ϱC
@
@
@
D
ϱD
C

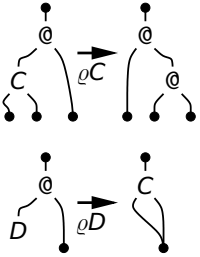

root
0
λz1
0
0
@
0
1
@
1
0
0
0
0
z1
0
D

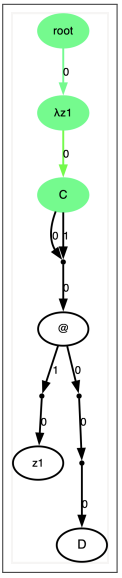

@
C
ϱC
@
@
@
D
ϱD
C

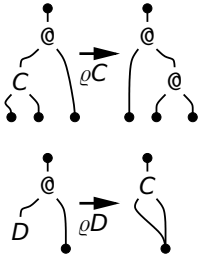

root
0
λz1
0
ρD
0
@
1
0
0
0
z1
0
D

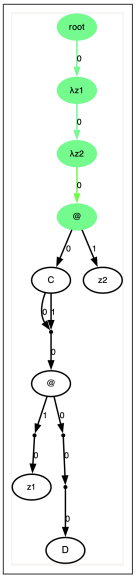

@
C
ϱC
@
@
@
D
ϱD
C


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


@
C
ϱC
@
@
@
D
ϱD
C


root
0
λz1
0
C
0
1
0
@
1
0
0
0
z1
0
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 $\tilde{\beta}$ (for same 👁)

Implementing some spine-β-strategy via 👁

Observations

- β can be implemented via iterating $\tilde{\beta}$ (for same 👁)
- constructor-steps correspond to α-conversions needed

Implementing some spine-β-strategy via 👁

Observations

- β can be implemented via iterating $\tilde{\beta}$ (for same 👁)
- constructor-steps correspond to α-conversions needed
- 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 $\tilde{\beta}$ (for same 👁)
- constructor-steps correspond to α-conversions needed
- 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 $\tilde{\beta}$ (for same 👁)
- constructor-steps correspond to α-conversions needed
- 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 bounded by size of final term

Implementing some spine-β-strategy via 👁

Corollary

results for $\tilde{\beta}$ carry over to spine-β. the cost of reduction to β-normal form is (pseudo)-linear in the number of leftmost–outermost β-steps to β-nf

Implementing some spine-β-strategy via 👁

Corollary

results for $\tilde{\beta}$ carry over to spine-β. the cost of reduction to β-normal form is (pseudo)-linear in the number of leftmost–outermost β-steps to β-nf

Reductionist view

- refactor Wadsworth through OTRS 👁 and constructor-unsharing OTGRS *G*👁 (Balabonski PhD)
- refactor theory and cost analysis for $\tilde{\beta}$ / $\bar{\beta}$ through (left-normal) OTRS theory
- refactor theory for lmo β and costs analysis through $\tilde{\beta}$
- refactor notion through graph rewriting
- refactor structure-rewriting through substitution calculus

Springtime for interaction nets!

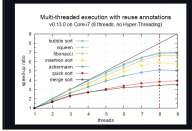
README | MIT license

Inpla: Interaction nets as a programming language

What is Inpla

Inpla is a multi-threaded parallel interpreter of interaction nets. Once you write programs for sequential execution, it works also in multi-threaded parallel execution. Each thread is managed on each CPU-core with POSIX-thread library.

- The current version is 0.13.0-2, released on **12 September 2024**. (See ChangeLog.md for details.)
- The below graph shows speed-up ratio to threads numbers for programs in the following benchmark table.



github.com/inpla/inpla

Paper GitHub

Higher Order Company

WELCOME TO THE PARALLEL FUTURE OF COMPUTATION

BEND

A PARALLEL LANGUAGE

With Bend you can write parallel code for multi-core CPUs/GPUs without being a CUDA/GPU expert with 15 years of experience. It feels just like Python!

No need to deal with the complexity of concurrent programming, locks, mutexes, atomics... **any** work that can be done in parallel **will** be done in parallel.

Example Program
Simple Parallel Sum

Try it Now →

higherorderco.com

The Vine Programming Language

Vine is an experimental new programming language based on interaction nets.

Vine is a multi-paradigm language, featuring seamless interop between functional and imperative patterns.

See `vine/examples/` for examples of Vine.

```
cargo run -r --bin vine run vine/examples/ENAME.vi
```

If you're curious to learn more, join the Vine Discord server.

(Vine is still under heavy development; many things will change.)

vine.dev

Optiscope

Optiscope is an experimental Lévy-optimal implementation of the pure lambda calculus enriched with native function calls, if-then-else expressions, & a fixed-point operator.

Being the first public implementation of Lambdascope [1] written in portable C99, it is also the first interaction net reducer capable of calling user-provided functions at native speed. As such, this combination allows one to interleave lazy evaluation with side effects, without resorting to external machinery like monads or algebraic effect handlers. In potential, Optiscope could stand as an optimal system for problems exhibiting vast sharing opportunities & computational intensity.

In what follows, we briefly explaine what it means for reduction to be Lévy-optimal, & then describe our results.

github.com/etiams/optiscope