Conway’s Game of Life and other orthogonal rewrite systems

Vincent van Oostrom

Part I: Game of Life as Orthogonal Graph Rewriting

Part II: Orthogonal Structured Rewriting

Part III: Premium content

Conway's Game of Life: Glider Gun

click for movie of Glider Gun

movie made of Troy Kidd's presentation (August 2025)

Conway’s Game of Life: Cellular Automaton

Cellular Automata

Typically, a cellular automaton (CA) is a regular network (line/grid/etc.) of cells with discrete states.

Cells update simultaneously as a function of neighboring cells. Each cell replaces its state with $f(s_1, s_2, \ldots) \in S$, where s_i are states of the cells in its neighborhood.

A *configuration* describes the state of all cells at some point in time. It is considered to extend infinitely in all directions, and can be represented as a function $c : \mathbb{Z}^d \to S$.

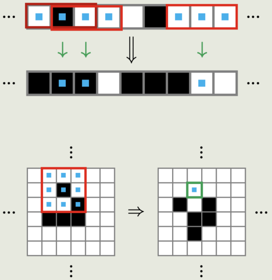
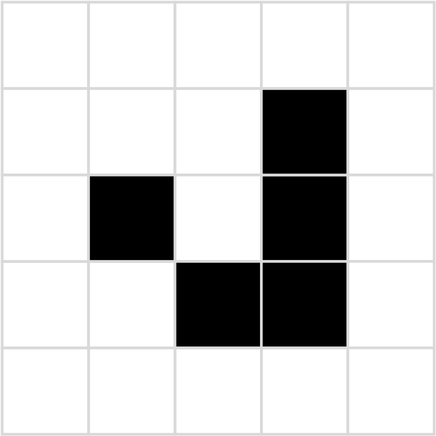


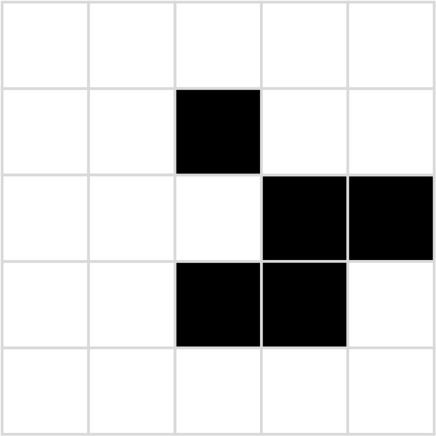
Figure 10. Examples of one step of computation, for 1-dimensional and 2-dimensional automatons.

Troy Kidd; osoi.dev/inet-slides

Conway’s Game of Life: CA Glider Step



Conway’s Game of Life: CA Glider Step

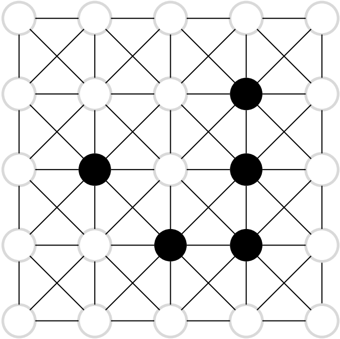


Conway's Game of Life: Graph Rewrite System

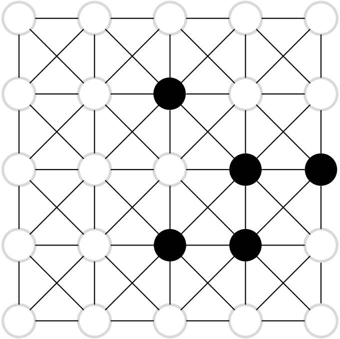
Idea: discrete topology

- labelled nodes represent cells
- ports (8 per node, ordered deosil) discretely represent cell boundaries
- wires (links; between ports) represent adjacency of cell boundaries

Conway’s Game of Life: GRS Glider Step



Conway's Game of Life: GRS Glider Step



Conway's Game of Life: Orthogonal GRS?

Orthogonal: local, asynchronous, parallel rewriting

- problem: CA cells must be updated synchronously

Conway’s Game of Life: Orthogonal GRS?

Orthogonal: local, asynchronous, parallel rewriting

- problem: CA cells must be updated synchronously
- GoL state ■ ■ ■

Conway's Game of Life: Orthogonal GRS?

Orthogonal: local, asynchronous, parallel rewriting

- problem: CA cells must be updated synchronously
- GoL state ■ ■ ■
- next GoL state should be ⁝ (an oscillator)

Conway's Game of Life: Orthogonal GRS?

Orthogonal: local, asynchronous, parallel rewriting

- problem: CA cells must be updated synchronously
- GoL state ■ ■ ■
- next GoL state should be ■
■
■
- but may be empty if evaluate asynchronously
  (strategy: update alive cells first, outside–in; then all counts ≤ 1 so all die)

Conway’s Game of Life: Orthogonal GRS?

Orthogonal: local, asynchronous, parallel rewriting

- problem: CA cells must be updated synchronously
- GoL state ■ ■ ■
- next GoL state should be ■
■
■
- but may be empty if evaluate asynchronously

Conway's Game of Life: Orthogonal GRS!

Orthogonal: local, asynchronous, parallel rewriting

- problem: CA cells must be updated synchronously
- GoL state ■ ■ ■
- next GoL state should be ■
■
■
- but may be empty if evaluate asynchronously

Solution here

- let each cell interact once with each of its neighbours before update

Conway's Game of Life: Orthogonal GRS!

Orthogonal: local, asynchronous, parallel rewriting

- problem: CA cells must be updated synchronously
- GoL state ■ ■ ■
- next GoL state should be ■
■
■
- but may be empty if evaluate asynchronously

Solution here

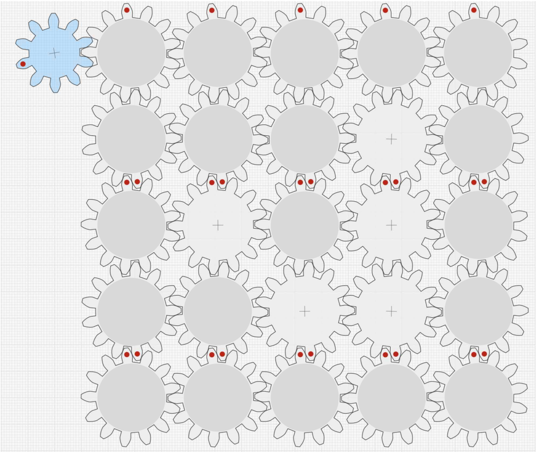
- let each cell interact once with each of its neighbours before update
- orchestrate these interactions by rotating (through all 8 ports of each cell)

CA ©lockwork

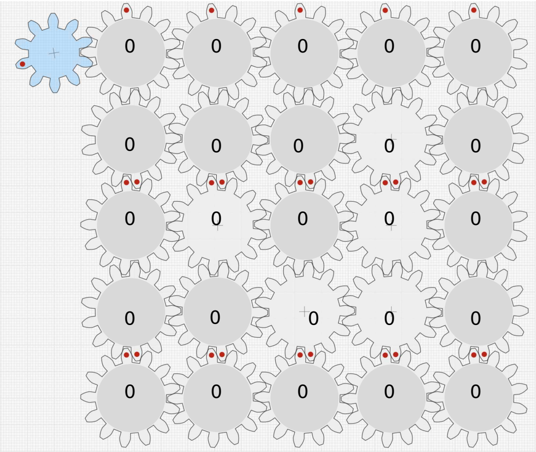
click for movie of ©lockwork

made using gear generator (August 2025)

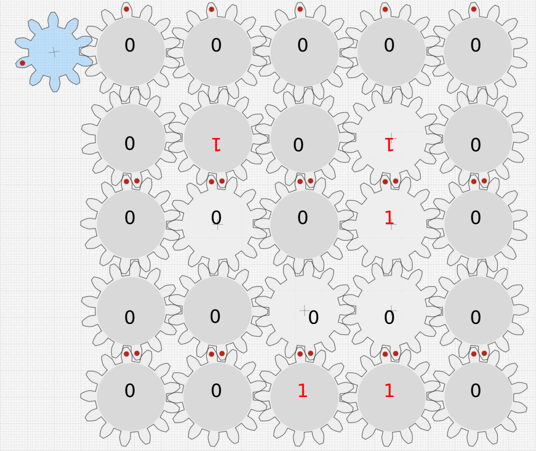
GoL ©lockwork for Glider Step



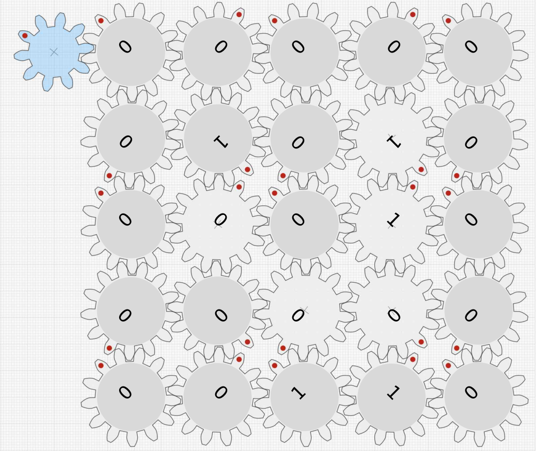
Initialise alive-neighbour counters to 0



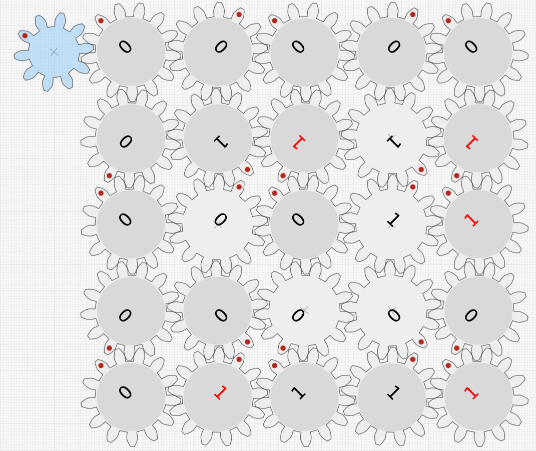
Increment each counter ●-opposite ○-alive neighbour



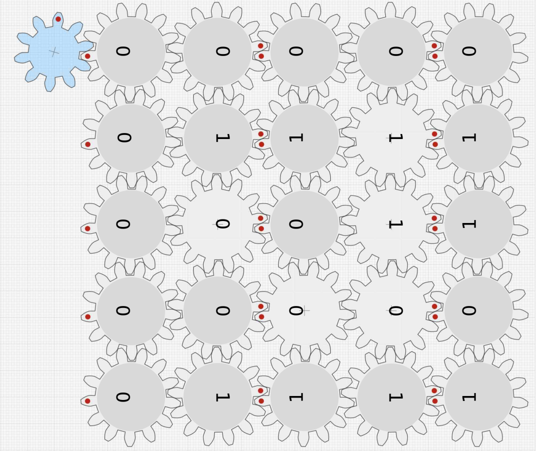
Rotate cogwheels in ©lockstep



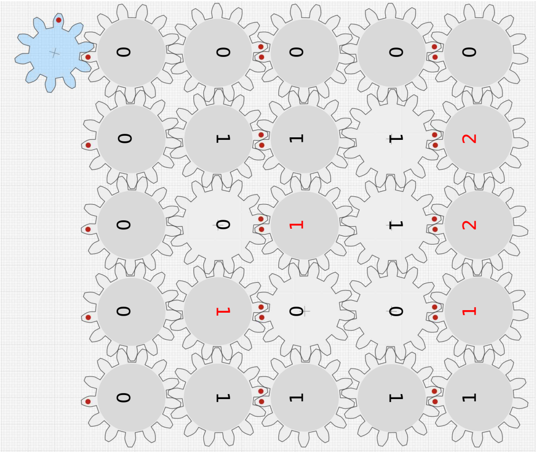
Increment each counter ●-opposite ○-alive neighbour



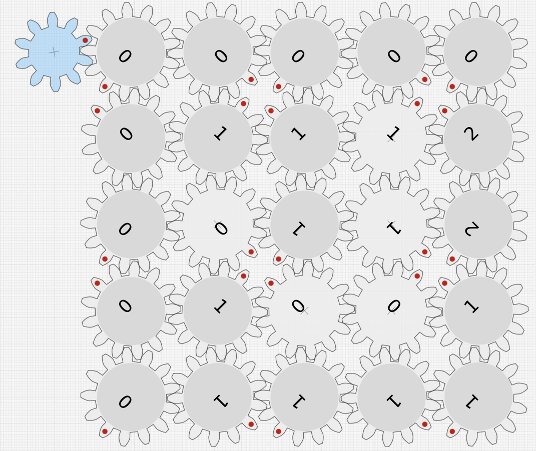
Rotate cogwheels in ©lockstep



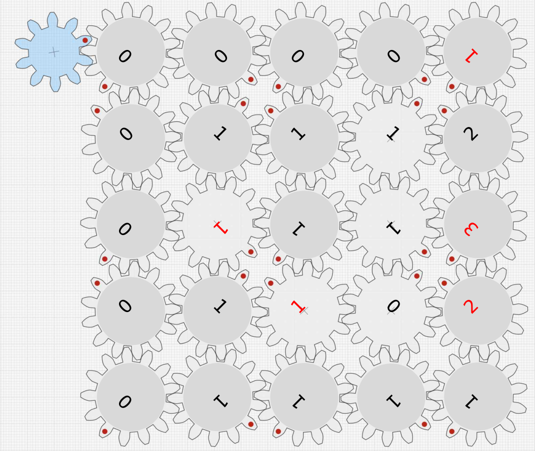
Increment each counter ●-opposite ○-alive neighbour



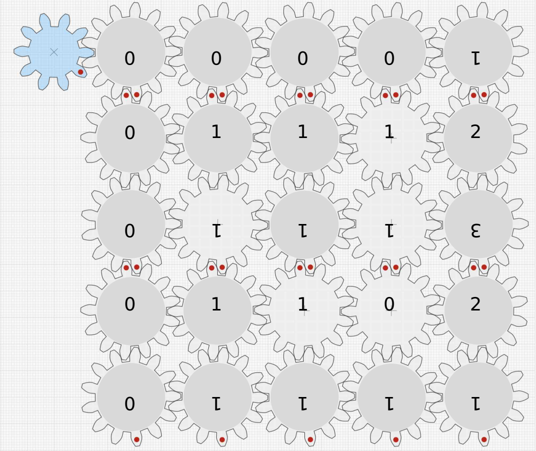
Rotate cogwheels in ©lockstep



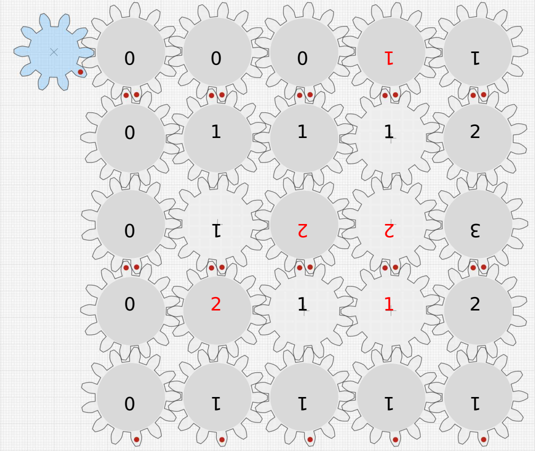
Increment each counter ●-opposite ○-alive neighbour



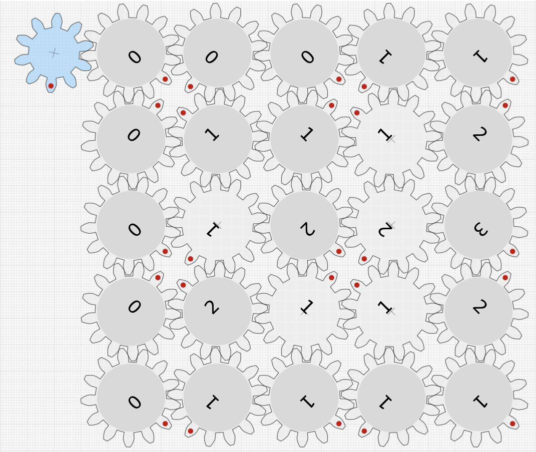
Rotate cogwheels in ©lockstep



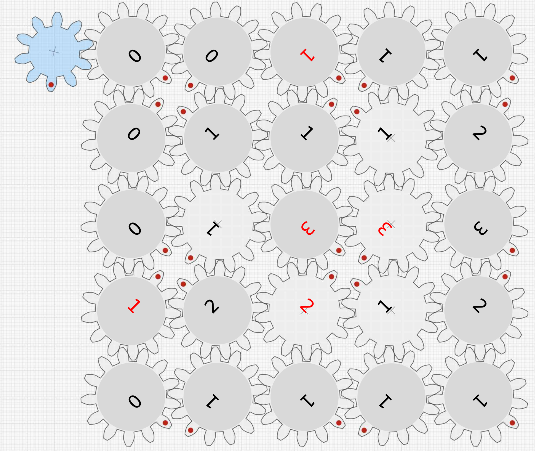
Increment each counter ●-opposite ○-alive neighbour



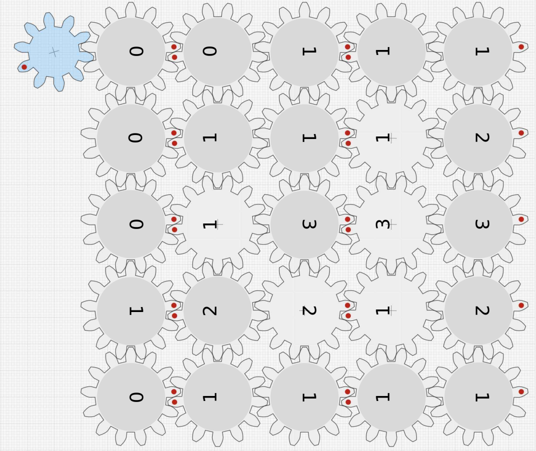
Rotate cogwheels in ©lockstep



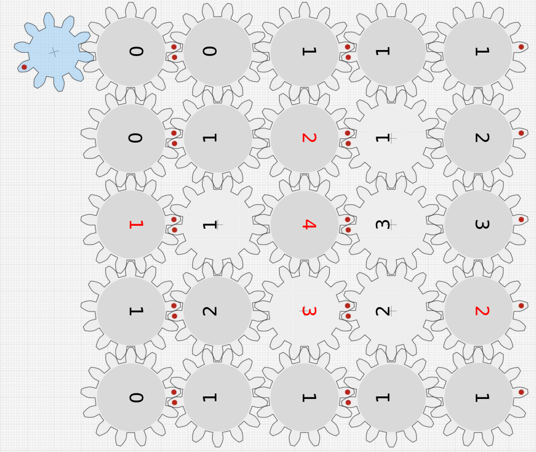
Increment each counter ●-opposite ○-alive neighbour



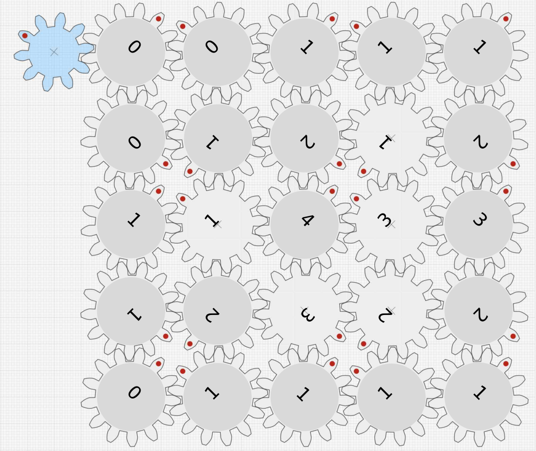
Rotate cogwheels in ©lockstep



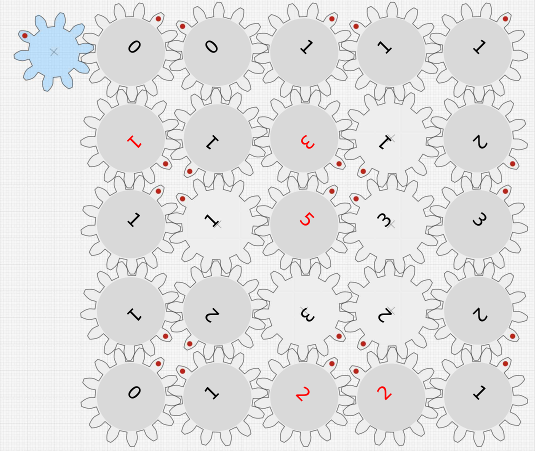
Increment each counter ●-opposite ○-alive neighbour



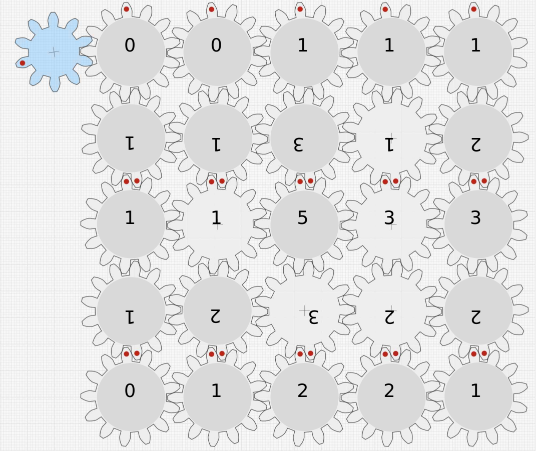
Rotate cogwheels in ©lockstep



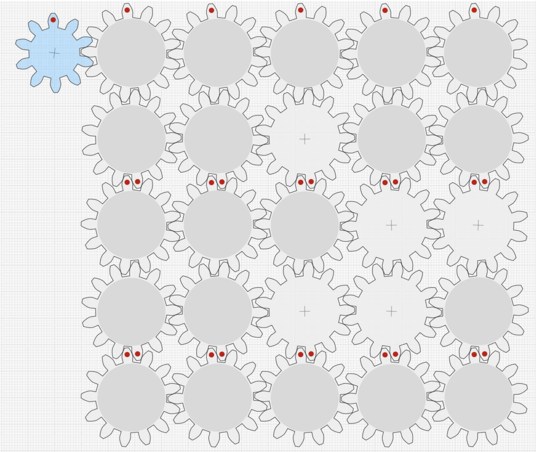
Increment each counter ●-opposite ○-alive neighbour



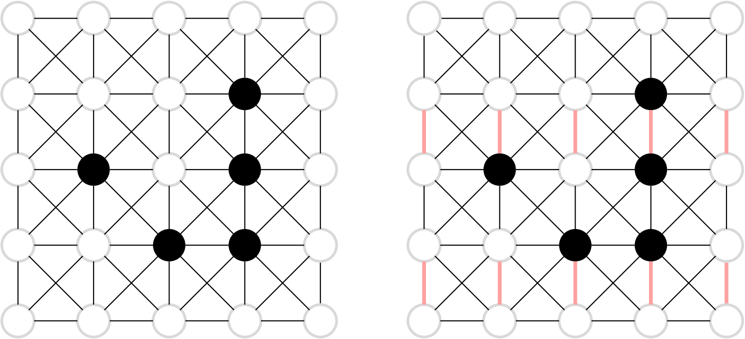
Rotate cogwheels in ©lockstep



Next GoL state (repeat . . .)

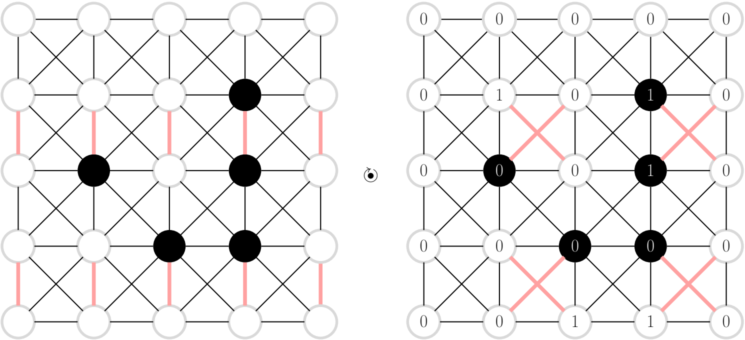


8 GRS ©locksteps for 1 GoL Glider Step



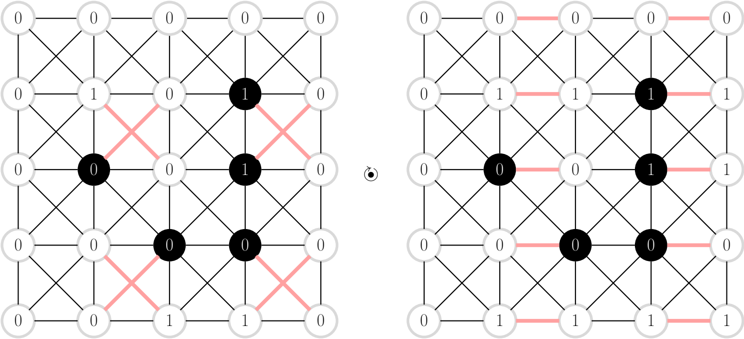
alternating rows of inactive and **active** links

GRS ©locksteps for GoL Glider Step



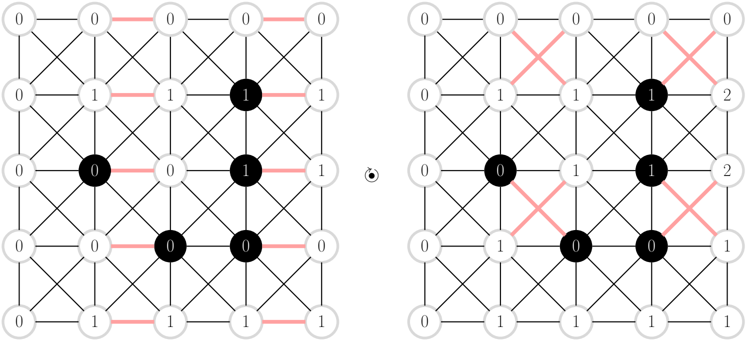
increment if other alive; rotate deosil / widdershins if row+column odd / even

GRS ©locksteps for GoL Glider Step



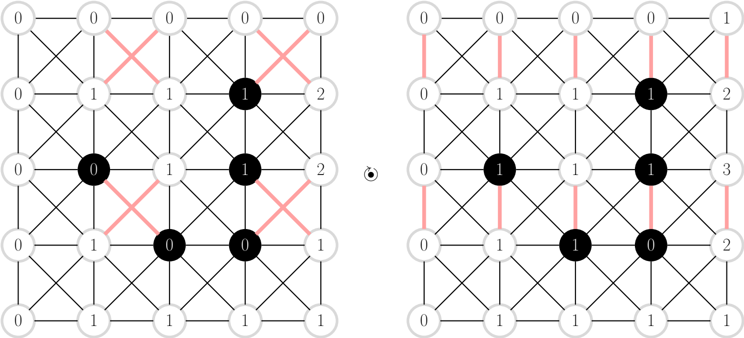
increment if other alive; rotate deosil / widdershins if row+column odd / even

GRS ©locksteps for GoL Glider Step



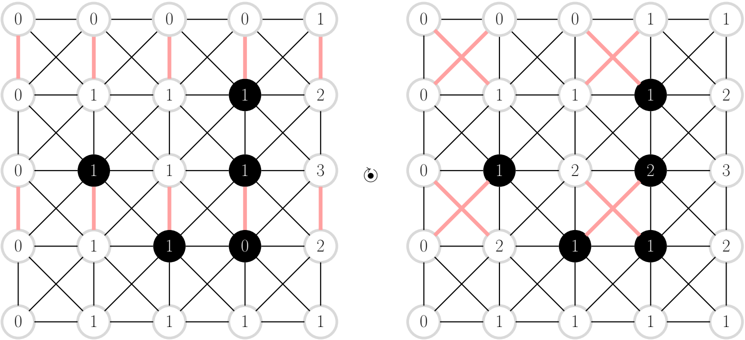
increment if other alive; rotate deosil / widdershins if row+column odd / even

GRS ©locksteps for GoL Glider Step



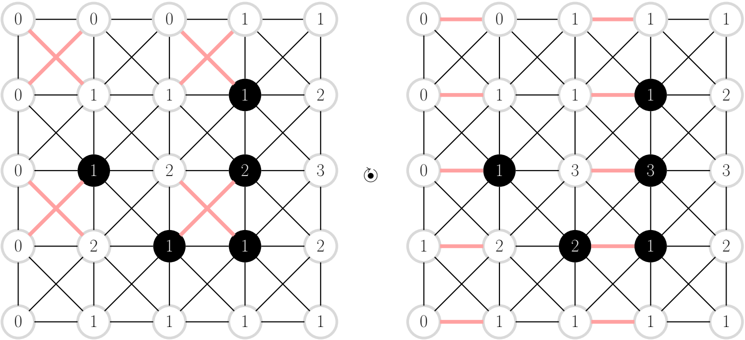
increment if other alive; rotate deosil / widdershins if row+column odd / even

GRS ©locksteps for GoL Glider Step



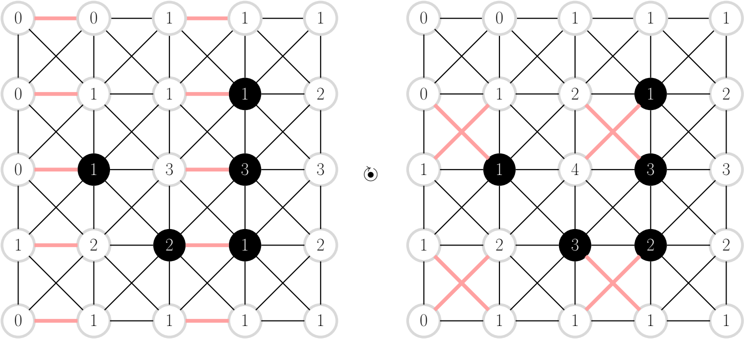
increment if other alive; rotate deosil / widdershins if row+column odd / even

GRS ©locksteps for GoL Glider Step



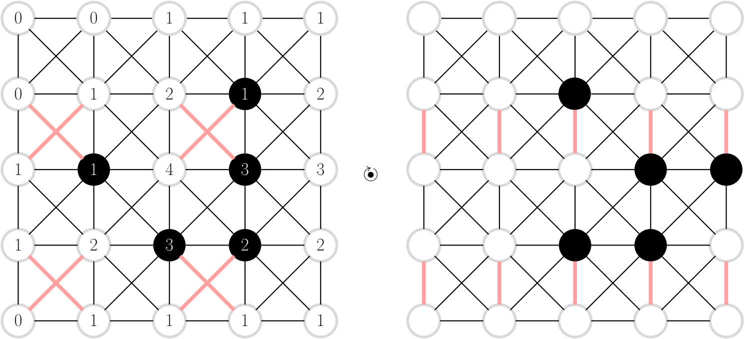
increment if other alive; rotate deosil / widdershins if row+column odd / even

GRS ©locksteps for GoL Glider Step



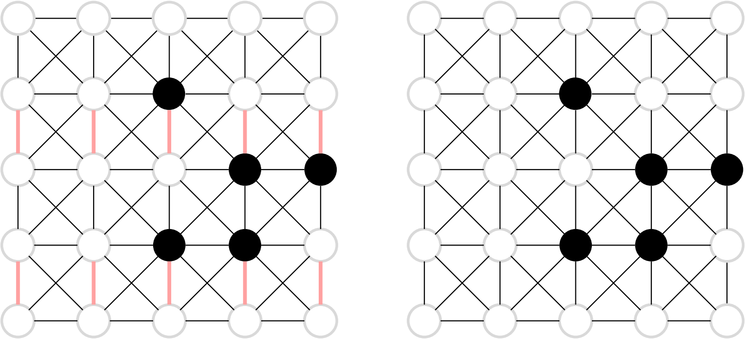
increment if other alive; rotate deosil / widdershins if row+column odd / even

GRS ©locksteps for GoL Glider Step



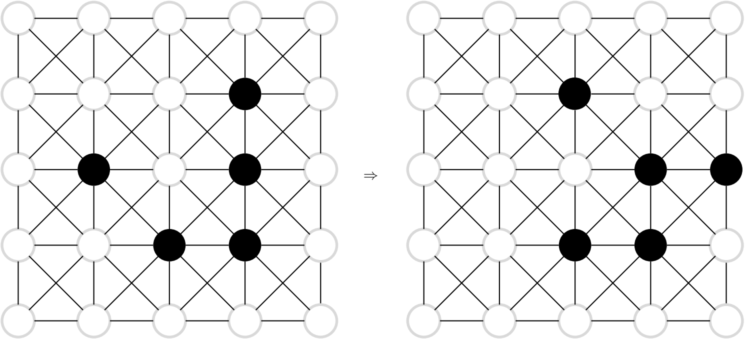
increment, rotate and update according to GoL

GRS ©locksteps for GoL Glider Step



result after 8 GRS ©locksteps

GRS ©locksteps for GoL Glider Step



Combining all 8 GRS ©locksteps into 1 Glider Step

Orthogonal GRS: Interaction Nets (Lafont 1990)

Definition 1. An *interaction net* is a finite set of labeled *cells* (each having some number of *ports*), a set of *free ports* not associated with any cells, and a set of *wires*, connecting each port to another one.

Cells have one *principal* port and $n \geq 0$ *auxiliary ports* (numbered in clockwise order), where n is the *arity* of the cell's symbol.

Wires may connect ports of the same cell or exist as a *cyclic wire* not connecting any ports.

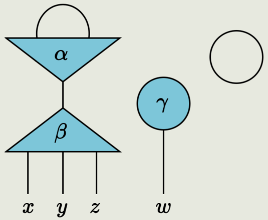


Figure 1. An interaction net.

Troy Kidd; osoi.dev/inet-slides

Orthogonal GRS: IN rule

Definition 2. An *interaction rule* is a pair of interaction nets having the same set of free ports.

The left-side net must consist of two cells with a wire between their principal ports, and a wire between each free port and an auxiliary port.

Rules may have more than two cells on the right, allowing for an exponentially increasing number of computations per step.

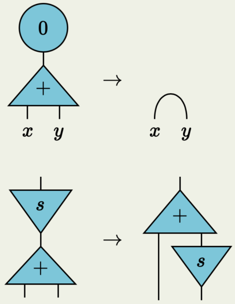


Figure 2. Two interaction rules. The first represents inferring $y = x$ from $y = 0 + x$.

Troy Kidd; osoi.dev/inet-slides

Orthogonal GRS: IN step

Definition 2. An *interaction rule* is a pair of interaction nets having the same set of free ports.

The left-side net must consist of two cells with a wire between their principal ports, and a wire between each free port and an auxiliary port.

Rules may have more than two cells on the right, allowing for an exponentially increasing number of computations per step.

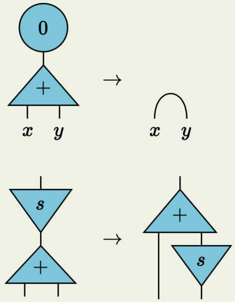
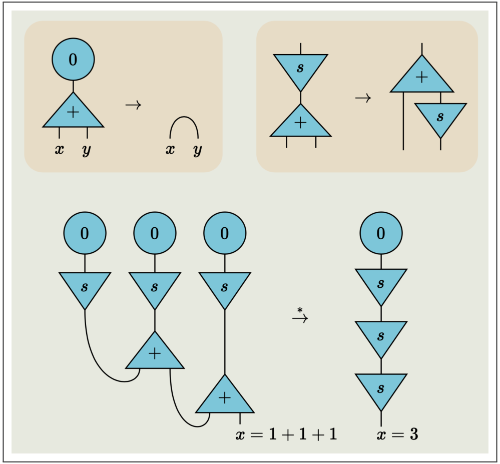


Figure 2. Two interaction rules. The first represents inferring $y = x$ from $y = 0 + x$.

Troy Kidd; osoi.dev/inet-slides

Orthogonal GRS: IN reduction



Troy Kidd; osoi.dev/inet-slides

Orthogonal GRS: IN parallel

Interaction nets were developed by Yves Lafont in 1990, as a practical model for parallel programming.

In this model, information is represented with a collection of cells and ports, connected by wires.

During one computational step, if a pair of cells matches a rule, they are replaced in a way that doesn't leave disconnected wires.

Many replacements can occur in parallel and can be repeated until there are no rule matches (in the case of a terminating computation).

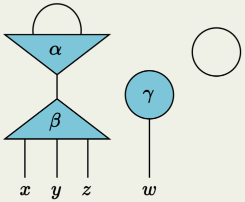


Figure 1. An interaction net.

Troy Kidd; osoi.dev/inet-slides

Orthogonal GRS: IN parallel reduction

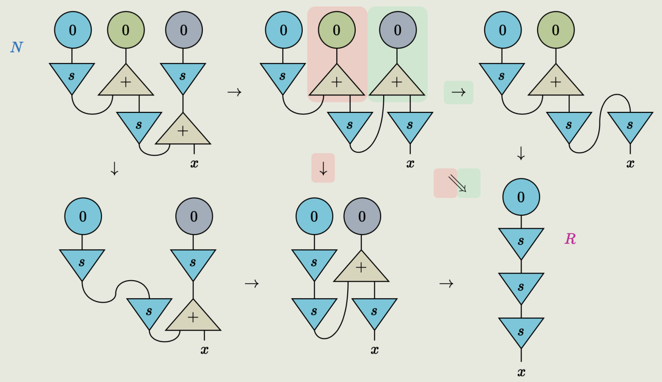


Figure 5. All reduction paths from net N to net R.

Troy Kidd; osoi.dev/inet-slides

Interaction Nets: Orthogonal GRS?

steps and multisteps

- local ✓
  (size of left- and right-hand side of rule bounded; for GoL 2 linked nodes)

Interaction Nets: Orthogonal GRS?

steps and multisteps

- local ✓
- asynchronous ✓
  (each node or link occurs in ≤ 1 redex-pattern; non-overlapping)

Interaction Nets: Orthogonal GRS?

steps and multisteps

- local ✓
- asynchronous ✓
- parallel ✓
  (result of contracting set of redex-patterns independent of order)

Interaction Nets: Orthogonal GRS!

steps and multisteps

- local ✓
- asynchronous ✓
- parallel ✓

Interaction Nets: Orthogonal GRS!

steps and multisteps

- local ✓
- asynchronous ✓
- parallel ✓

GoL signature

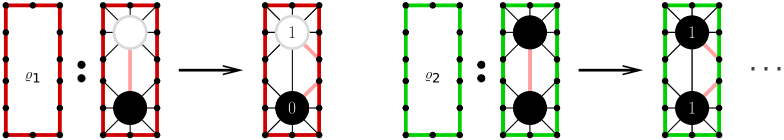
- symbols (arity 8): 

  $\ldots$
  ($\leq 2 \times 2 \times 10 \times 8 = 320$ symbols: alive?,rot,#neighbours,principal port)

GoL rule signature

- symbols: 

- rule symbols (arity 14):

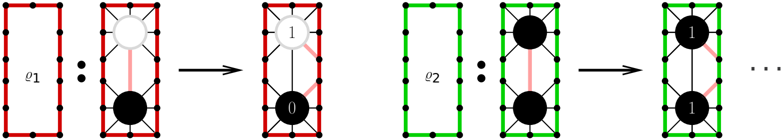


$(\leq (2 \times 10)^2 \times 2 \times 8 = 6400$ rule symbols: symbol,rot,port,symbol)

GoL signature

- symbols: 

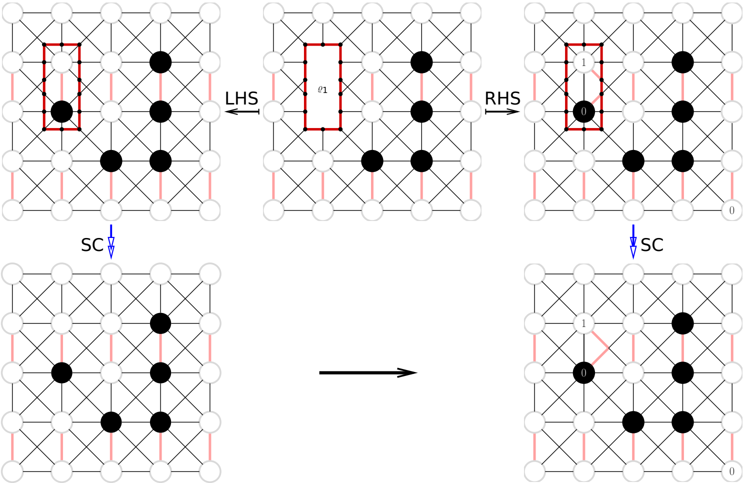
- rule symbols:



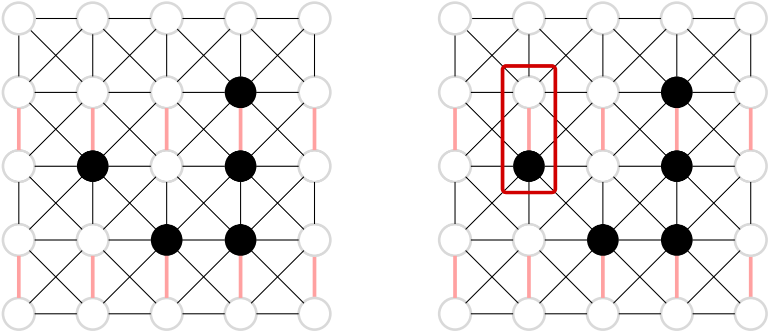
- normalised rewriting modulo Substitution Calculus (SC):

(indirection)

GoL step →

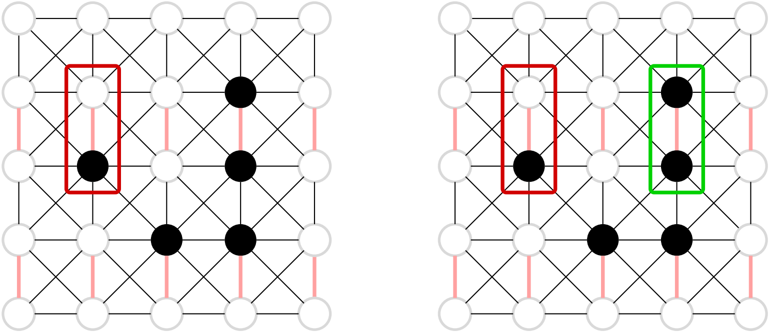


GoL ©lockstep (full multistep)



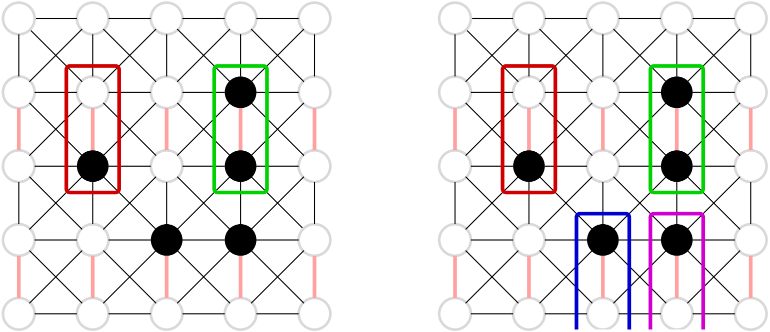
locating a redex-pattern

GoL ©lockstep



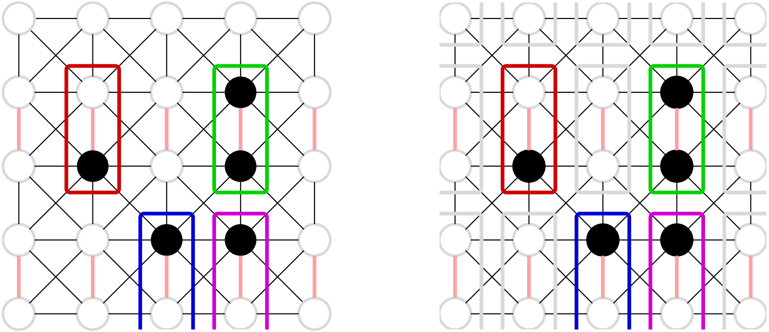
locating another redex-pattern (non-overlapping)

GoL ©lockstep



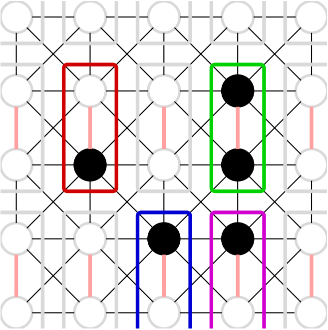
locating yet other redex-patterns (all pairwise non-overlapping)

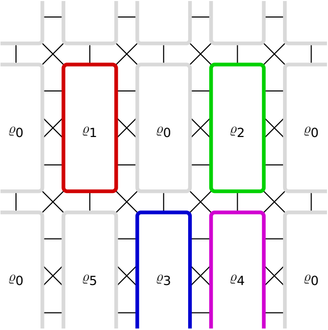
GoL ©lockstep



locating all redex-patterns (each node occurs in some redex-pattern)

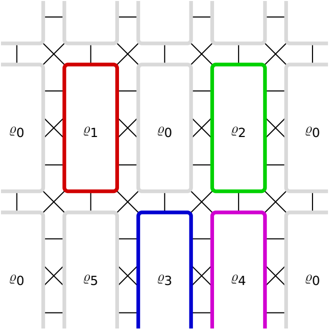
GoL ©lockstep

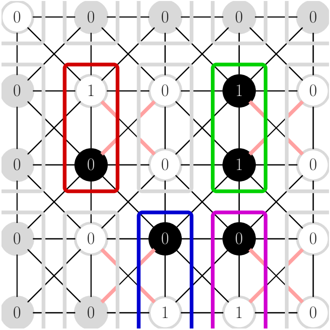




abstracting all redex-patterns into rule symbols; arity 14 $(= 2 \cdot (8 - 1))$

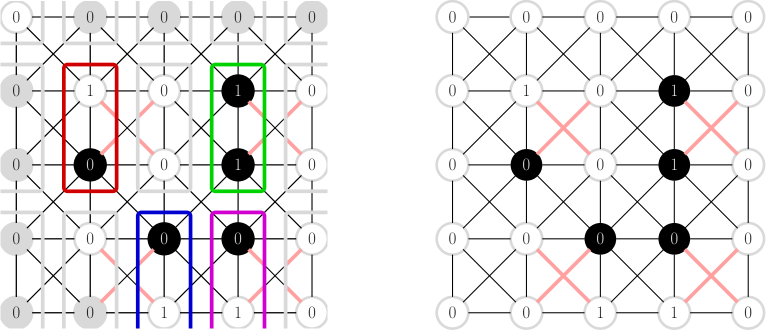
GoL ©lockstep





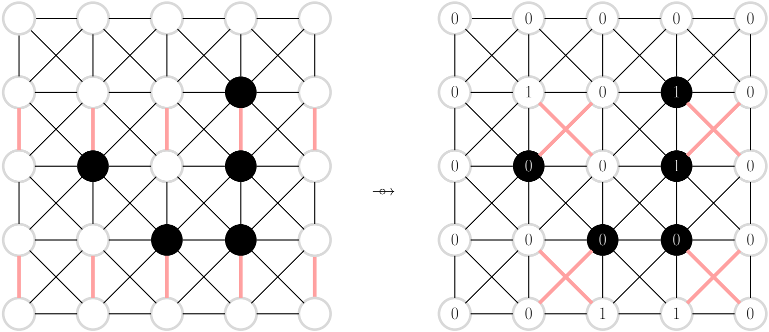
replacing all rule symbols by rhss; ©lockstep

GoL ©lockstep



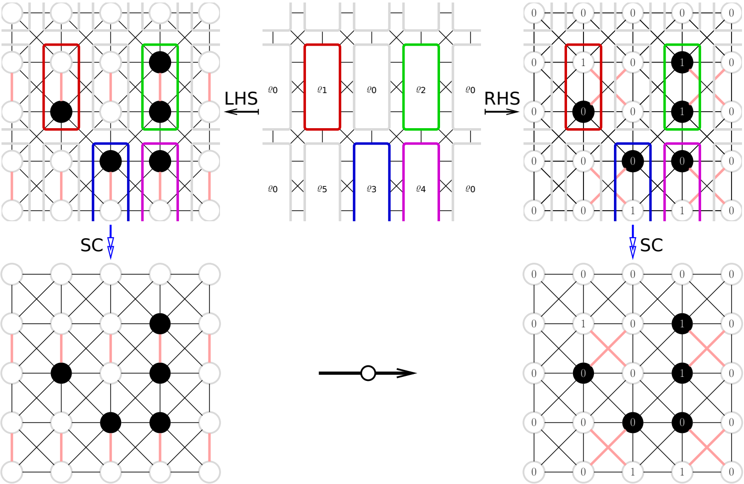
substituting rhss in graph (by substitution calculus)

GoL ©lockstep

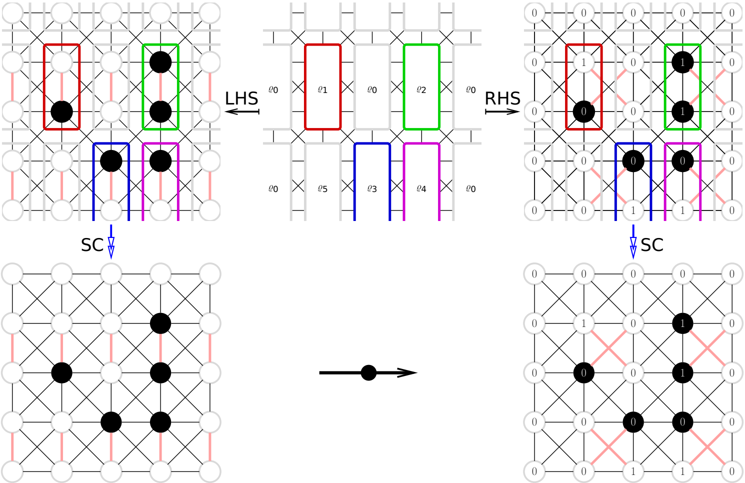


(includes deosil / widdershins rotation))

GoL ©lockstep; multistep ⊸→



GoL ©lockstep; full multistep ⟶•⟶



GoL is orthogonal

Theory of Orthogonality

- sequentialisation: $\rightarrow\ \subseteq\ \multimap\!\!\rightarrow\ \subseteq\ \twoheadrightarrow$

GoL is orthogonal

Theory of Orthogonality

- sequentialisation: $\to \subseteq \multimap\!\!\!\to \subseteq \twoheadrightarrow$
- confluence-by-parallelism: $\multimap\!\!\!\to$ has the diamond property (by residuation)

GoL is orthogonal

Theory of Orthogonality

- sequentialisation: $\to \subseteq \mathrel{-\!\!\circ\!\!\to} \subseteq \twoheadrightarrow$
- confluence-by-parallelism: $\mathrel{-\!\!\circ\!\!\to}$ has the diamond property
- cube: tiling 3-peak with diamonds yields a cube
  (entails co-initial reductions form semi-lattice; least upperbounds)

GoL is orthogonal

Theory of Orthogonality

- sequentialisation: $\to \subseteq \mathrel{-\!\!\circ\!\!\to} \subseteq \twoheadrightarrow$
- confluence-by-parallelism: $\mathrel{-\!\!\circ\!\!\to}$ has the diamond property
- cube: tiling 3-peak with diamonds yields a cube
- finite developments: every development of $\mathrel{-\!\!\circ\!\!\to}$ is finite
  (development of multistep is reduction only contracting residuals)

GoL is orthogonal

Theory of Orthogonality

- sequentialisation: $\to \subseteq \multimap\!\!\!\!\to \subseteq \twoheadrightarrow$
- confluence-by-parallelism: $\multimap\!\!\!\!\to$ has the diamond property
- cube: tiling 3-peak with diamonds yields a cube
- finite developments: every development of $\multimap\!\!\!\!\to$ is finite
- full multistep strategy (©lockstep) is normalising

GoL is orthogonal

Theory of Orthogonality

- sequentialisation: $\to \subseteq \mathrel{-\!\!\circ\!\!\to} \subseteq \twoheadrightarrow$
- confluence-by-parallelism: $\mathrel{-\!\!\circ\!\!\to}$ has the diamond property
- cube: tiling 3-peak with diamonds yields a cube
- finite developments: every development of $\mathrel{-\!\!\circ\!\!\to}$ is finite
- full multistep strategy is normalising
- ...

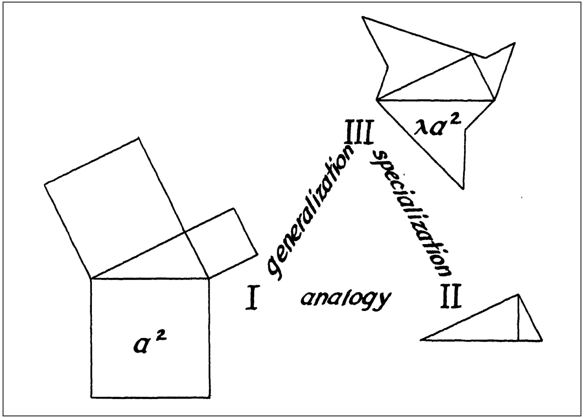
GoL is orthogonal

Theory of Orthogonality

- sequentialisation: $\rightarrow \subseteq \mathrel{-\!\circ\!\!\rightarrow} \subseteq \twoheadrightarrow$
- confluence-by-parallelism: $\mathrel{-\!\circ\!\!\rightarrow}$ has the diamond property
- cube: tiling 3-peak with diamonds yields a cube
- finite developments: every development of $\mathrel{-\!\circ\!\!\rightarrow}$ is finite
- full multistep strategy is normalising
- ...

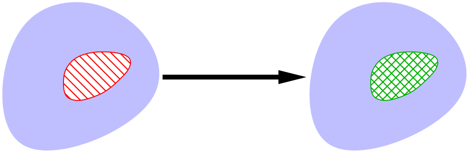
INs are linear so have random descent
(WN $\implies$ SN for nets; reductions to normal form all same length)

Pólya’s triangle



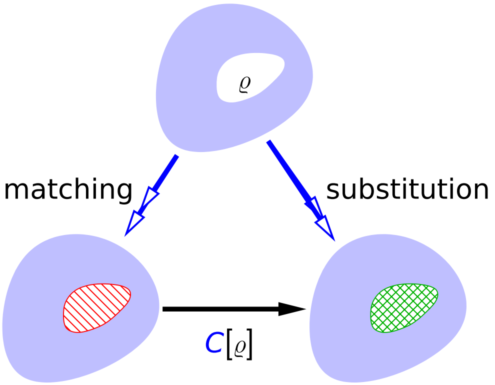
Mathematics and Plausible Reasoning, Volume1, 1954, Fig. 2.3

Pólya's triangle in structured rewriting



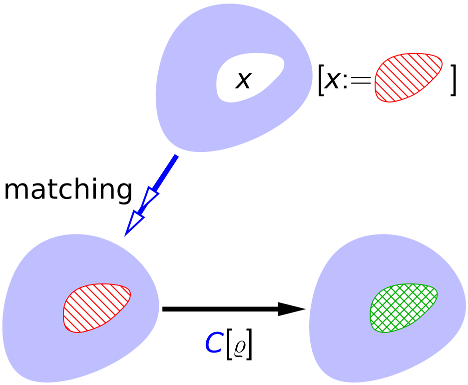
rewrite step $C[\ell]{\downarrow} \to C[r]{\downarrow}$ for rewrite rule $\varrho : \ell \to r$ and context C

Pólya's triangle in structured rewriting



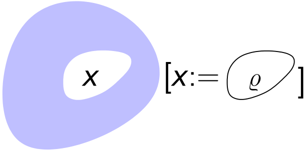
rewrite step $C[\varrho] : C[\ell]\!\downarrow \to C[r]\!\downarrow$ for rule $\varrho : \ell \to r$ and context C

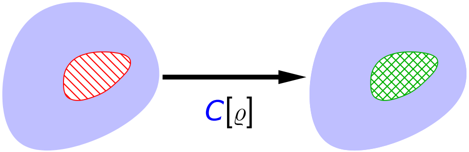
Pólya’s triangle in structured rewriting



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

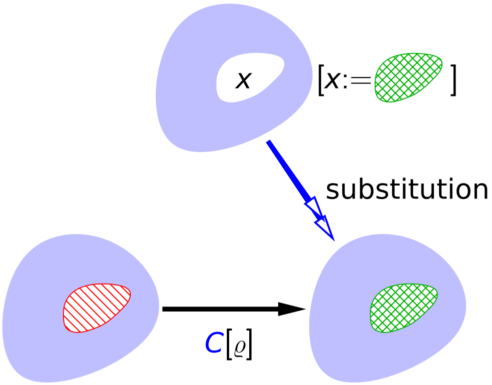
Pólya’s triangle in structured rewriting





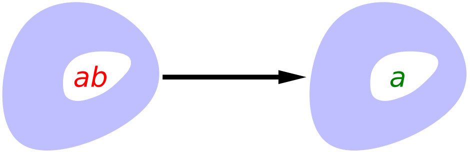
rewrite step $C[\varrho] : C[\ell]{\downarrow} \to C[r]{\downarrow}$ for structure $C[x]$ and rule $\varrho : \ell \to r$

Pólya's triangle in structured rewriting



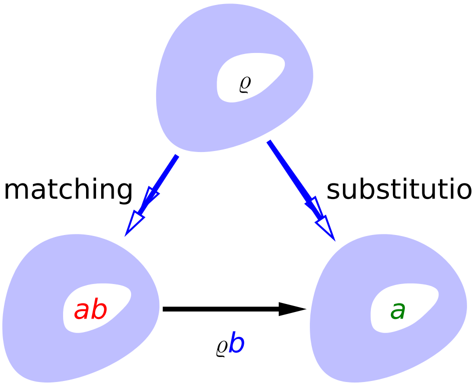
substitution for rewrite step $C[\varrho] : C[\ell]{\downarrow} \to C[r]{\downarrow}$ for structure $C[x]$ and rule $\varrho : \ell \to r$

Pólya's triangle in string rewriting



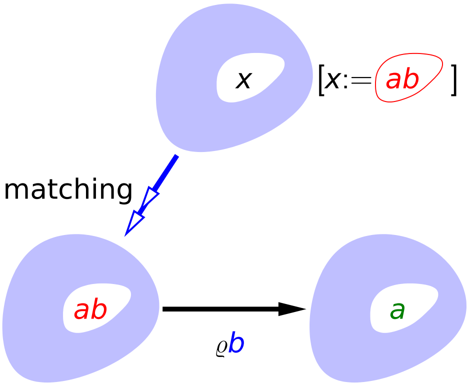
rewrite step $abb \to ab$ for rewrite rule $\varrho : ab \to b$ and context b

Pólya’s triangle in string rewriting



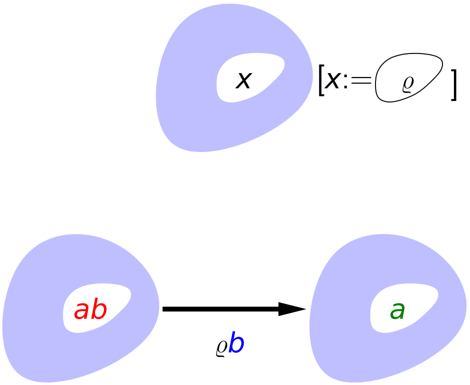
rewrite step $\varrho b : abb \to ab$ for rewrite rule $\varrho : ab \to b$ and context b

Pólya's triangle in string rewriting



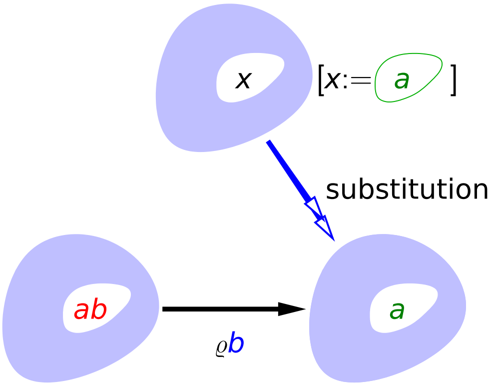
matching for rewrite step $\varrho b : abb \to ab$ for structure xb and rule $\varrho : ab \to b$

Pólya’s triangle in string rewriting



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

Pólya's triangle in string rewriting



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

Structured rewriting

Definition (of structured rewriting modulo substitution calculus)

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

Structured rewriting

Definition (of structured rewriting modulo substitution calculus)

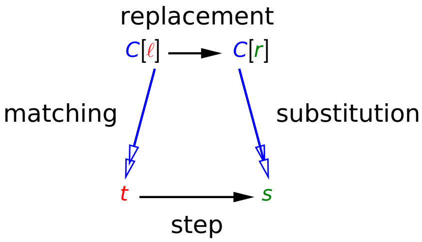
- structures over a signature having variables $x, y, \ldots$ over structures
- substitution calculus $\rightarrow_{\mathcal{SC}}$ on structures; $\downarrow$ denotes $\mathcal{SC}$-normal form ($\mathcal{SC}$-nf)
- rules $\varrho : \ell \rightarrow r$ with ϱ in signature and ℓ, r structures
- contexts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

Definition (of structured rewrite step)

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

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

Structured rewriting: step

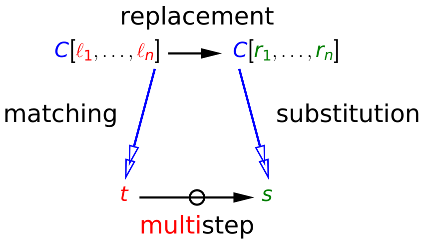


Definition (of structured rewrite step)

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

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

Structured rewriting: multistep

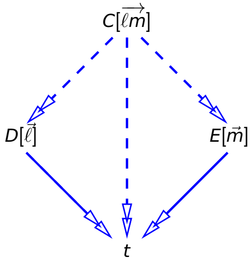


Definition (of structured rewrite multistep)

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

$$s = C[\ell_1, \ldots, \ell_n]{\downarrow}\; {}_{\mathcal{SC}}\!\twoheadleftarrow C[\ell_1, \ldots, \ell_n] \multimap_{\vec{\varrho}} C[r_1, \ldots, r_n] \twoheadrightarrow_{\mathcal{SC}} C[r_1, \ldots, r_n]{\downarrow} = t$$

Structured Orthogonality



occurrences of redex-patterns can be abstracted from in parallel
($\overrightarrow{\ell m}$ is union of $\vec{\ell}$ and $\vec{m}$)

Substitution Calculi (SC)

Example

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

Substitution Calculi (SC)

Example

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

Substitution Calculi (SC)

Example

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

Substitution Calculi (SC)

Example

- (higher-order) term rewriting: simply typed $\lambda\alpha\beta\eta$-calculus
- termgraph rewiting: the ж-calculus
- interaction net: indirection-calculus —●— $\rightarrow$ ——
- net rewriting: proofnet-calculus (PN)

Substitution Calculi (SC)

Example

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

Substitution Calculi (SC)

Example

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

Substitution Calculi (SC)

Example

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

Substitution Calculi (SC)

Example

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

Axioms on substitution calculi (SC)

Axioms

A1 the SC is complete (confluent and terminating)

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

A3 multisteps can be sequentialised / serialised (some development)

Axioms on substitution calculi (SC)

Axioms

A1 the SC is complete (confluent and terminating)

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

A3 multisteps can be sequentialised / serialised (some development)

Theory of Orthogonality

- sequentialisation: $\to \subseteq \multimap \subseteq \twoheadrightarrow$

Axioms on substitution calculi (SC)

Axioms

A1 the SC is complete (confluent and terminating)

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

A3 multisteps can be sequentialised / serialised (some development)

Theory of Orthogonality

- sequentialisation: $\rightarrow \subseteq \mathrel{-\!\!\circ\!\!\!\rightarrow} \subseteq \twoheadrightarrow$
- confluence-by-parallelism: $\mathrel{-\!\!\circ\!\!\!\rightarrow}$ has the diamond property

Axioms on substitution calculi (SC)

Axioms

A1 the SC is complete (confluent and terminating)

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

A3 multisteps can be sequentialised / serialised (some development)

Theory of Orthogonality

- sequentialisation: $\rightarrow \subseteq \multimap\!\!\rightarrow \subseteq \twoheadrightarrow$
- confluence-by-parallelism: $\multimap\!\!\rightarrow$ has the diamond property
- finite developments: every development of $\multimap\!\!\rightarrow$ is finite
  (development of multistep is reduction only contracting residuals)

Axioms on substitution calculi (SC)

Axioms

A1 the SC is complete (confluent and terminating)

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

A3 multisteps can be sequentialised / serialised (some development)

Theory of Orthogonality

- sequentialisation: $\to \subseteq \multimap\!\!\!\to \subseteq \twoheadrightarrow$
- confluence-by-parallelism: $\multimap\!\!\!\to$ has the diamond property
- finite developments: every development of $\multimap\!\!\!\to$ is finite
- cube: tiling 3-peak with diamonds yields a cube
  (entails co-initial reductions form semi-lattice; least upperbounds)

Axioms on substitution calculi (SC)

Axioms

A1 the SC is complete (confluent and terminating)

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

A3 multisteps can be sequentialised / serialised (some development)

Theory of Orthogonality

- sequentialisation: $\to \subseteq \multimap \subseteq \twoheadrightarrow$
- confluence-by-parallelism: $\multimap$ has the diamond property
- finite developments: every development of $\multimap$ is finite
- cube: tiling 3-peak with diamonds yields a cube
- full multistep strategy is normalising

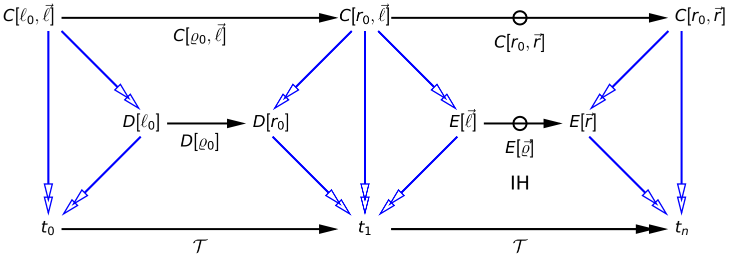
Axioms on substitution calculi (SC)

Axioms

A1 the SC is complete (confluent and terminating)

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

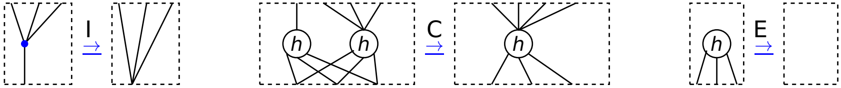
A3 multisteps can be sequentialised / serialised (some development)



Termgraphs as structures

Instance

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



Termgraphs as structures

Instance

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

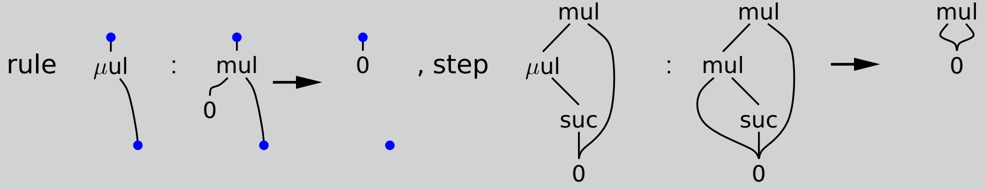
- ж-calculus has implicit garbage collection
- termgraphs in ж-normal form are maximally shared

Termgraphs as structures

Instance

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

Example (of termgraph step modulo ж)

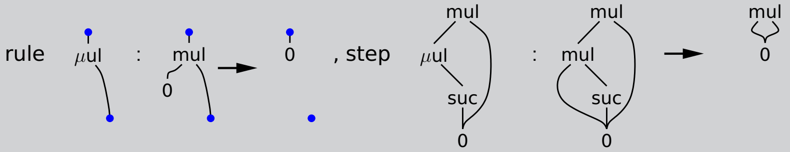


Termgraphs as structures

Instance

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

Example (of termgraph step modulo ж)



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

Conclusions

- normalised rewriting with respect to substitution calculus (SC)

Conclusions

- normalised rewriting with respect to substitution calculus (SC)
- orthogonality guarantees redex-patterns simultaneously abstractable (structure obtained by simultaneous substitution redex-patterns by SC)

Conclusions

- normalised rewriting with respect to substitution calculus (SC)
- orthogonality guarantees redex-patterns simultaneously abstractable

-

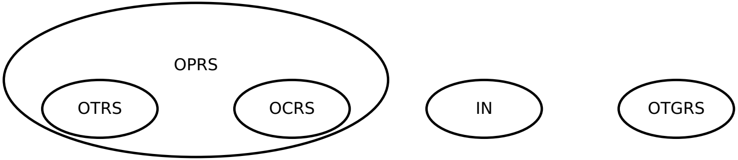
Conclusions

- normalised rewriting with respect to substitution calculus (SC)
- orthogonality guarantees redex-patterns simultaneously abstractable

-

Conclusions

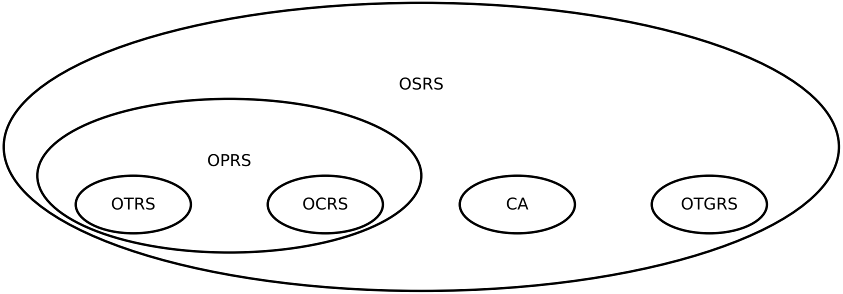
- normalised rewriting with respect to substitution calculus (SC)
- orthogonality guarantees redex-patterns simultaneously abstractable



-

Conclusions

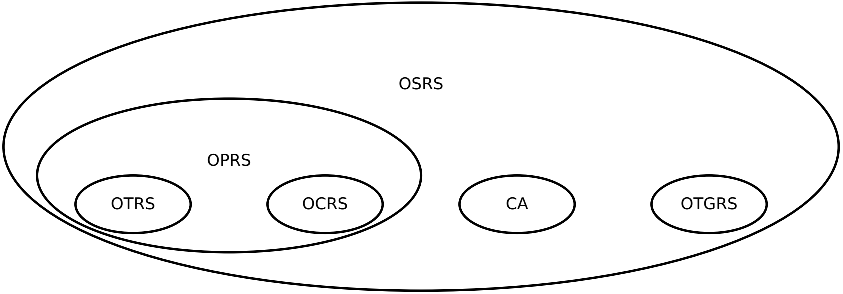
- normalised rewriting with respect to substitution calculus (SC)
- orthogonality guarantees redex-patterns simultaneously abstractable



-

Conclusions

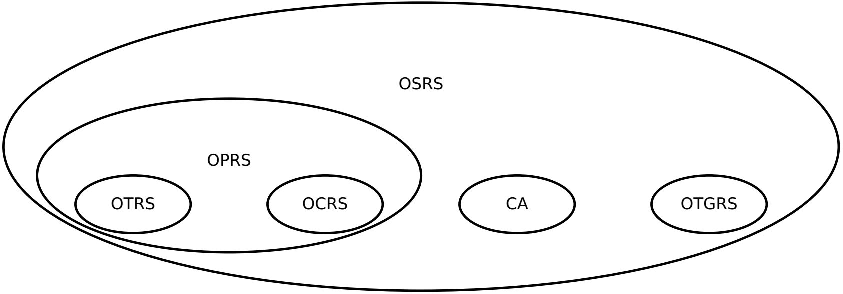
- normalised rewriting with respect to substitution calculus (SC)
- orthogonality guarantees redex-patterns simultaneously abstractable



-
- steps as structures

Conclusions

- normalised rewriting with respect to substitution calculus (SC)
- orthogonality guarantees redex-patterns simultaneously abstractable

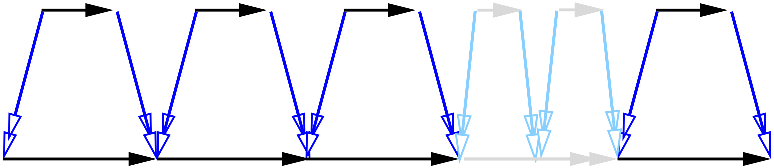


-
- steps as structures
- theory of orthogonality

Exploiting substitution calculi to redistribute steps

Cost-saving observations

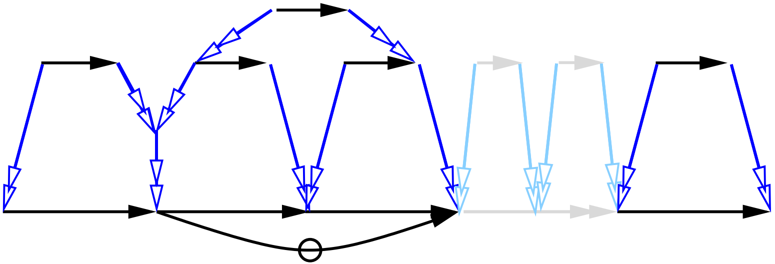
- should avoid substitution and subsequent matching inverse to each other
- matching more lhss simultaneously (multisteps) enables parallelism
- by not going to $\mathcal{SC}$-normal forms we may sometimes eliminate matching



Exploiting substitution calculi to redistribute steps

Cost-saving observations

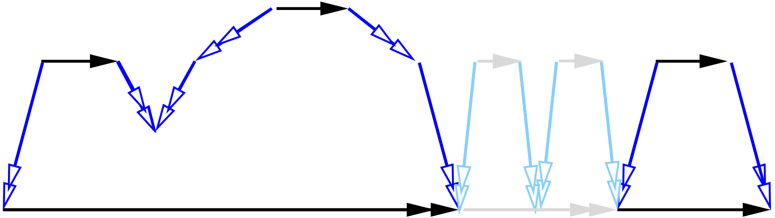
- should avoid substitution and subsequent matching inverse to each other
- matching more lhss simultaneously (multisteps) enables parallelism
- by not going to $\mathcal{SC}$-normal forms we may sometimes eliminate matching



Exploiting substitution calculi to redistribute steps

Cost-saving observations

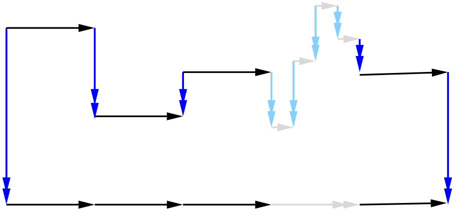
- should avoid substitution and subsequent matching inverse to each other
- matching more lhss simultaneously (multisteps) enables parallelism
- by not going to $\mathcal{SC}$-normal forms we may sometimes eliminate matching



Exploiting substitution calculi to redistribute steps

Cost-saving observations

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

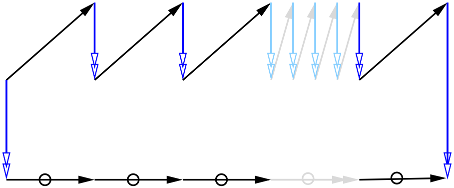


deterministic rewrite system $\Longrightarrow$ dipper SC

Exploiting substitution calculi to redistribute steps

Cost-saving observations

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

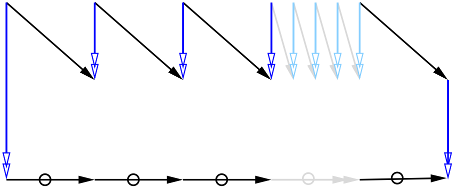


matching trivial $\Longrightarrow$ cascading SC

Exploiting substitution calculi to redistribute steps

Cost-saving observations

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



substitution trivial $\Longrightarrow$ stepping SC

Implementation of 👀

Motivation for 👀

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

Applicative Inductive Interaction System (👁)

Definition (of an 👁)

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

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

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

notational conventions:

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

Applicative Inductive Interaction System (👁)

Definition (of an 👁)

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

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

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

Example (of an 👁)

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

Applicative Inductive Interaction System (👁)

Definition (of an 👁)

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

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

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

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

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

Applicative Inductive Interaction System (👁)

Example (of an 👁)

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

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

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

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

Implementing 👀

Question (on implementation of 👁)

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

efficient in time / space

Implementing 👁👁

Question (on implementation of 👁)

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

efficient in time / space

Observations (explored further below)

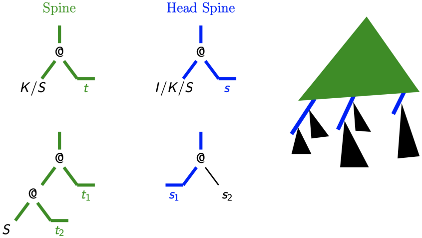
- spine strategy is (hyper-(head-))normalising
  since every 👁 is left-normal orthogonal TRSs
- matching-phase is trivial (since lhss left-linear, comprise two symbols)
  substitution-phase not trivial (rhss may replicate arguments)

Spine strategy

Definition

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

Head Spine: recur on left.



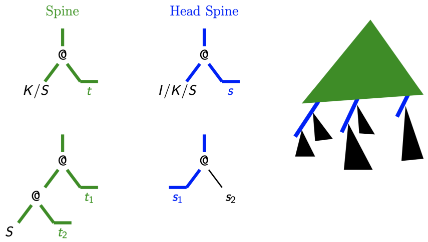
Example

S(SISI)(K(IK))

Spine strategy

Definition

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



Lemma

Every term not in normal form has Spine redex

Spine strategy for 👁

Definition (of spine for 👁-terms)

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

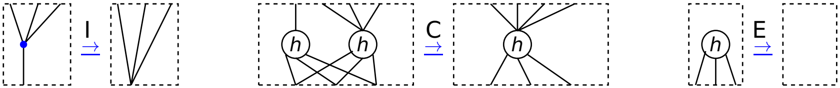
Lemma (normalising strategy)

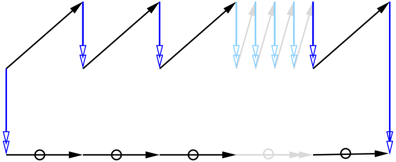
- *every term not in normal form has redex-pattern on spine, so a strategy*
- *spine strategy is a normalising strategy having random descent*

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

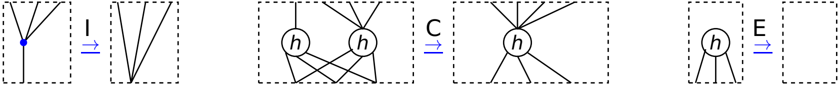
Implementing 👁 in termgraphs by cascading ж

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



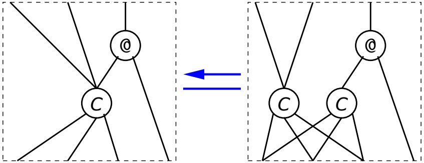


Implementing 👁 in termgraphs by cascading ж



Idea (minimal unsharing; Wadsworth's admissibility)

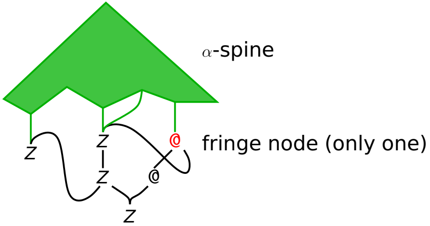
- instead of maximal sharing, unshare only constructors in redex-patterns
- goal: amortise cost of ж-steps by charging 👁-steps



Termgraph α-spine strategy

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

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



Termgraph α-spine strategy

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

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

Lemma

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

Proof.

by minimality using acyclicity of termgraphs □

Termgraph α-spine strategy

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

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

Definition (of α-spine strategy)

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

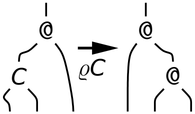
by lemma always some step possible until whole termgraph is α-spine (in nf)

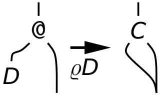
Example α-spine reduction (Java code ⇒ dot ⇒ graphs)

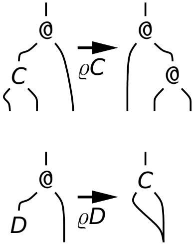
recall 👁-rules:

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

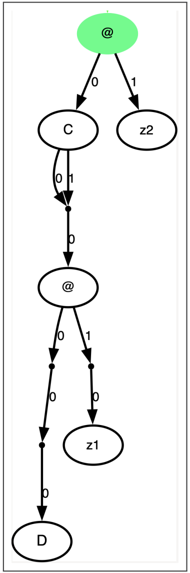
as termgraph rules:



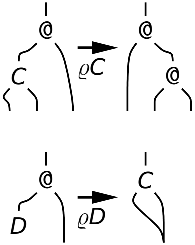


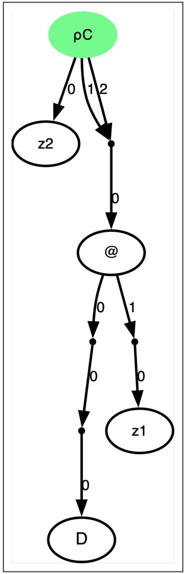


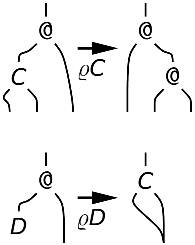
@
C
ϱC
@
@
@
D
ϱD
C

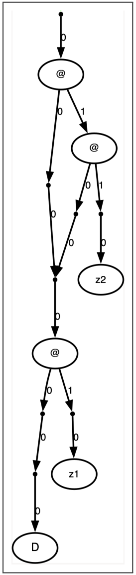


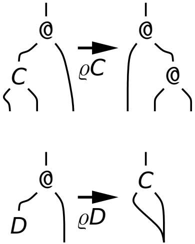
@
0
1
C
z2
0
1
0
@
0
1
0
0
z1
0
D

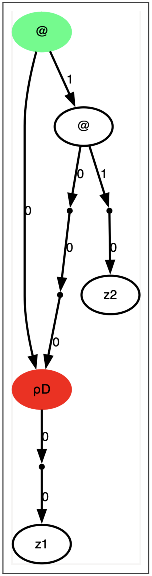

@
C
ϱC
@
@
@
D
ϱD
C

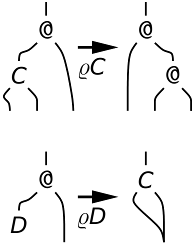

ρC
0
1
2
z2
0
@
0
1
0
0
z1
0
D

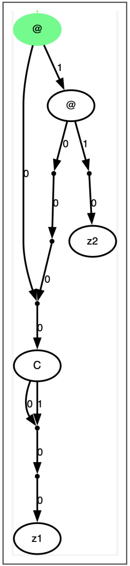

@
C
ϱC
@
@
@
D
ϱD
C

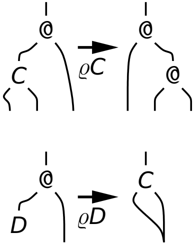

@
@
z2
@
z1
D

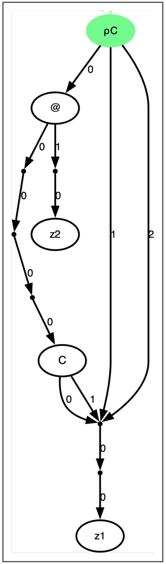

@
C
ϱC
@
@
@
D
ϱD
C

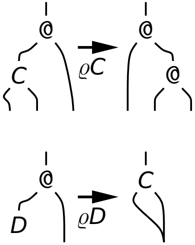

@
1
@
0
1
0
0
0
z2
0
ρD
0
0
z1

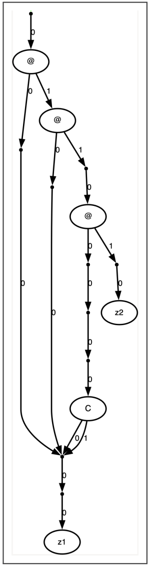

@
C
ϱC
@
@
@
D
ϱD
C

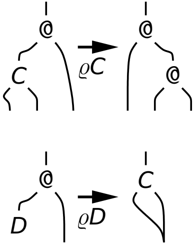

@
1
@
0
1
0
0
0
z2
0
0
C
0
1
0
0
z1

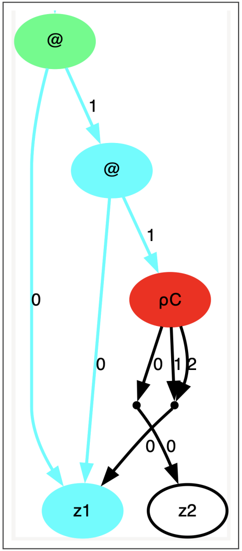

@
C
ϱC
@
@
@
D
ϱD
C

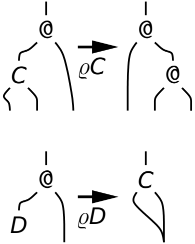

ρC
@
z2
C
z1


@
C
ϱC
@
@
@
D
ϱD
C

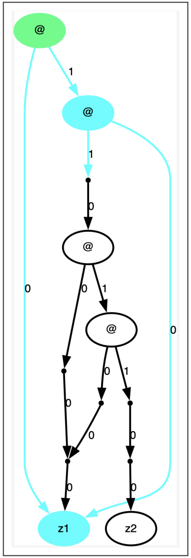

@
@
@
z2
C
z1


@
C
ϱC
@
@
@
D
ϱD
C

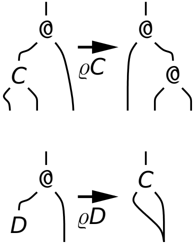

@
1
0
@
1
ρC
0
0
1
2
0
0
z1
z2



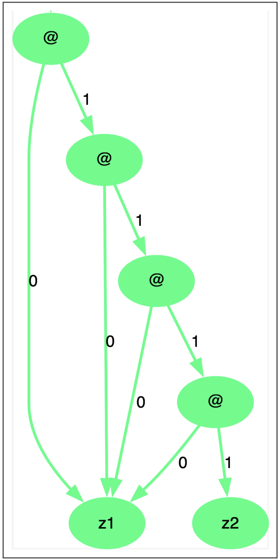
@
C
ϱC
@
@
@
D
ϱD
C



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



@
C
ϱC
@
@
@
D
ϱD
C



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

Correspondence between termgraphs and terms

Theorem

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

Correspondence between termgraphs and terms

Theorem

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

Correspondence between termgraphs and terms

Theorem

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

Correspondence between termgraphs and terms

Theorem

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

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

Decompiling 👁 to the λ-calculus

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

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

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

Decompiling 👁 to the λ-calculus

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

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

Example (of tree homomorphism for example 👁)

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

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

Decompiling 👁 to the λ-calculus

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

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

Example (of tree homomorphism for example 👁)

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

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

Decompiling 👁 to the λ-calculus

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

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

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

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

Decompiling 👁 to the λ-calculus

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

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

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

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

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

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

Compiling the λ-calculus to 👁

Lemma (??)

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

Compiling the λ-calculus to 👁

Lemma (??)

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

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

Compiling the λ-calculus to 👁

Lemma (??)

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

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

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

do allow components to share constructors when these have the same rules

compilation known variation on the abstraction algorithm (custom combinators)

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

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

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

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

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

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

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

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

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

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

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

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

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

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

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

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

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

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

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

Compiling the λ-calculus to 👁

Definition (of 👁-lifting)

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

Lemma (👁-lifting)

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

Proof.

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

Implementing wβ-reduction via 👁

Observations

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

Implementing wβ-reduction via 👁

Observations

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

Corollary

results for 👁👁 carry over to wβ

Implementing wβ-reduction via 👁

Observations

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

Corollary

results for 👁👁 carry over to wβ

Perspective

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

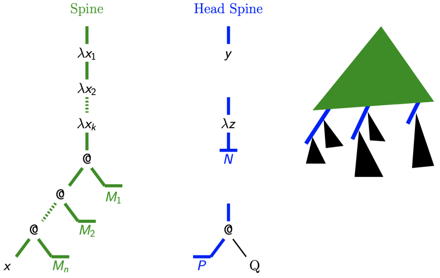
What about Spine strategies for full β?

Spine strategy

Definition

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

Head Spine: recur on left.



Example

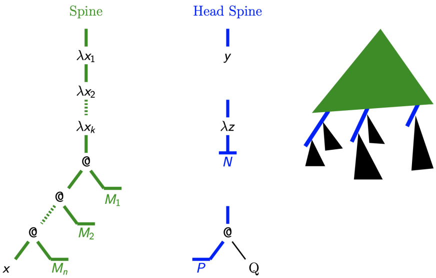
$x((\lambda x.(\lambda z.zz))y)(xx)(II)$

Spine strategy

Definition

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

Head Spine: recur on left.



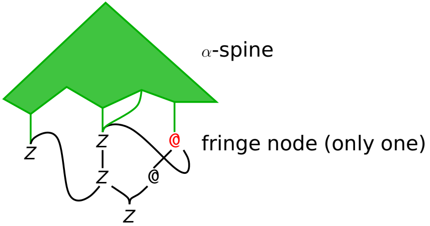
Lemma

Every term not in normal form has Spine redex

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

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

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



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

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

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

Definition (of α-spine strategy)

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

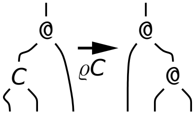
idea: a combinator on fringe / α-spine is a λ-abstraction (in the β-nf), so may iterate on its body, effectuated in 👁 by suppling a fresh variable

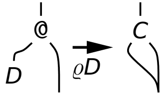
Example α-spine reduction (Java code $\Rightarrow$ dot $\Rightarrow$ graphs)

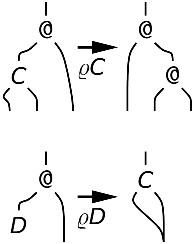
recall 👁-rules:

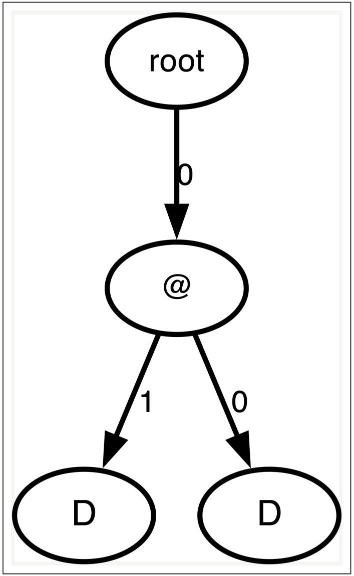
$\varrho_C : C(x_1, x_2)\, x_0 \to x_1\, (x_2\, x_0)$
$\varrho_D : D\, x_0 \to C(x_0, x_0)$

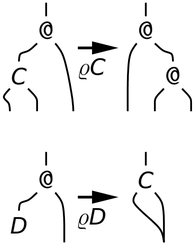
and termgraph rules:

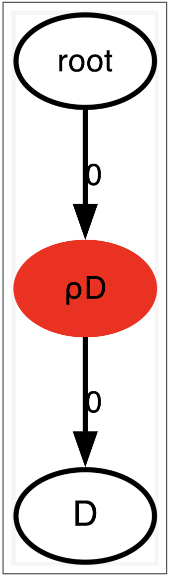


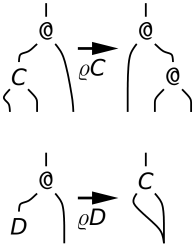


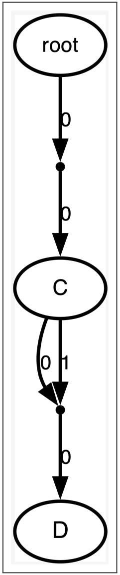

@
C
ϱC
@
@
@
D
ϱD
C

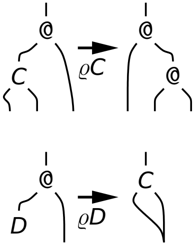

root
0
@
1
0
D
D

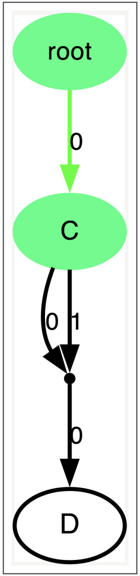

@
C
ϱC
@
@
@
D
ϱD
C

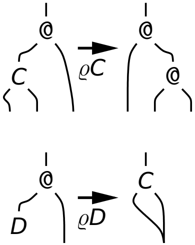

root
0
ρD
0
D

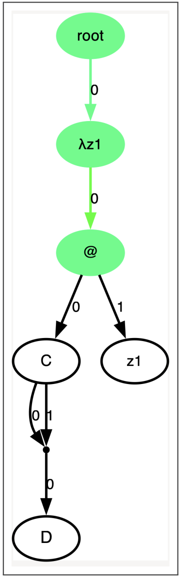

@
C
ϱC
@
@
@
D
ϱD
C

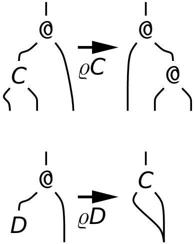

root
0
0
C
0
1
0
D


@
C
ϱC
@
@
@
D
ϱD
C

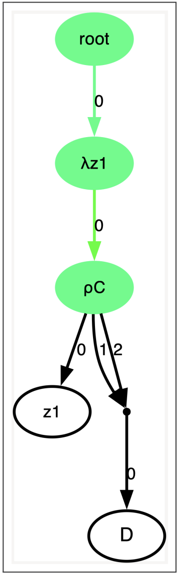

root
0
C
0
1
0
D


@
C
ϱC
@
@
@
D
ϱD
C

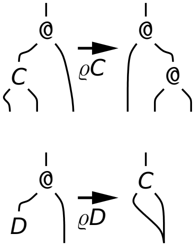

root
0
λz1
0
@
0
1
C
z1
0
1
0
D

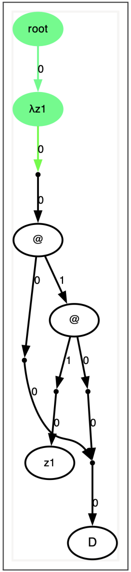


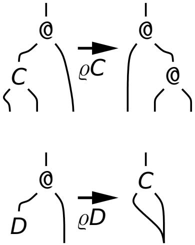
@
C
ϱC
@
@
@
D
ϱD
C

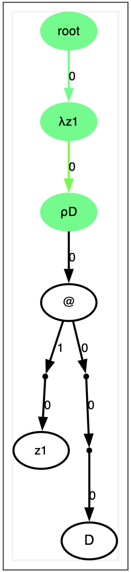


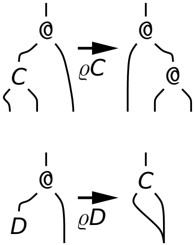
root
0
λz1
0
ρC
0
1
2
z1
0
D

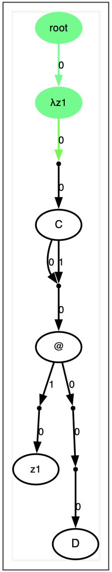

@
C
ϱC
@
@
@
D
ϱD
C

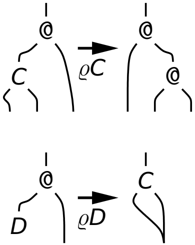

root
0
λz1
0
0
@
0
1
@
1
0
0
0
0
z1
0
D

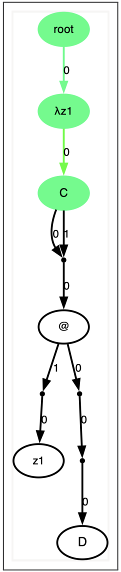

@
C
ϱC
@
@
@
D
ϱD
C

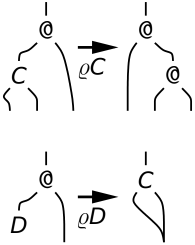

root
0
λz1
0
ρD
0
@
1
0
0
0
z1
0
D


@
C
ϱC
@
@
@
D
ϱD
C

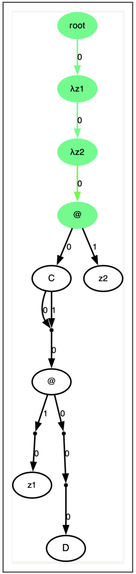

root
0
λz1
0
0
C
0
1
0
@
1
0
0
0
z1
0
D


@
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
λz1
λz2
@
C
z2
@
z1
D

Implementing some spine-β-strategy via 👁

Observations

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

Implementing some spine-β-strategy via 👁

Observations

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

Implementing some spine-β-strategy via 👁

Observations

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

Implementing some spine-β-strategy via 👁

Observations

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

Implementing some spine-β-strategy via 👁

Observations

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

Implementing some spine-β-strategy via 👁

Corollary

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

Implementing some spine-β-strategy via 👁

Corollary

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

Perspective

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

Implementing β-reduction

Complexity unavoidable

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

Implementing β-reduction

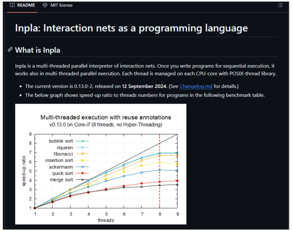
Complexity unavoidable

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

Non-consequence

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

Springtime for interaction nets!



README MIT license

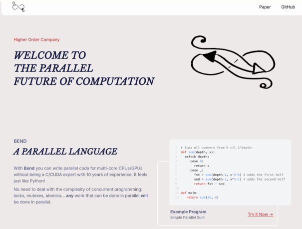
Inpla: Interaction nets as a programming language

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 C/CUDA expert with 10 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.

```
# Sums all numbers from 0 til 2^depth:
def sum(depth, x):
  switch depth:
    case 0:
      return x
    case _:
      fst = sum(depth-1, x*2+0) # adds the first half
      snd = sum(depth-1, x*2+1) # adds the second half
      return fst + snd

def main:
  return sum(30, 0)
```

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.

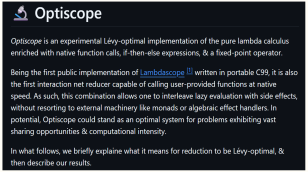
Vine is a multi-paradigm language, featuring seamless interop between functional and imperative patterns.

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

```
cargo run -r --bin vine run vine/examples/$NAME.vi
```

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

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



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.

vine.dev

github.com/etiams/optiscope

More conclusions

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

More conclusions

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

More conclusions

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

More conclusions

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

More conclusions

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

More conclusions

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

More conclusions

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

Standing on the shoulders of giants

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

Standing on the shoulders of giants

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

Standing on the shoulders of giants

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

Standing on the shoulders of giants

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

Standing on the shoulders of giants

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

Standing on the shoulders of giants

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

Standing on the shoulders of giants

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

Standing on the shoulders of giants

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

Standing on the shoulders of giants

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

Standing on the shoulders of giants

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

Contributions

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

Amortised complexity

Idea

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

Amortised complexity

Idea

measure complexity by averaging over reductions

Example

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

Amortised complexity

Idea

measure complexity by averaging over reductions

Example

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

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

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

Amortised complexity

Idea

measure complexity by averaging over reductions

Example

incrementing a counter in binary $011 \to_{\mathsf{inc}} 111 \to_{\mathsf{inc}} 0001 \to_{\mathsf{inc}} 1001 \to_{\mathsf{inc}} \dots$

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

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

$0(1(1(\bullet))) \to_i i(0(1(1(\bullet)))) \to_b 1(1(1(\bullet))) \to_i i(1(1(1(\bullet)))) \to_b 0(i(1(1(\bullet)))) \to_b$
$0(0(i(1(\bullet)))) \to_b 0(0(0(i(\bullet)))) \to_b 0(0(0(1(\bullet)))) \to_i \dots$

Banker's / accounting method in TRSs

Idea

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

Banker's / accounting method in TRSs

Idea

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

Example

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

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

Banker's / accounting method in TRSs

Idea

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

Example

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

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

Banker's / accounting method in TRSs

Idea

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

Example

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

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

Banker's / accounting method in TRSs

Idea

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

Example

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

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

Banker's / accounting method in TRSs

Idea

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

Example

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

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