On naïvely implementing the $\lambda\beta$-calculus

Vincent van Oostrom
University of Sussex
School of Engineering and Informatics
Brighton, United Kingdom
vvo@sussex.ac.uk

We present naïve (Haskell) implementations of reduction to (weak-head) normal form in the $\lambda\beta$-calculus. As known, a λ-term M can be lifted to a supercombinator term $\mathscr{L}(M)$ and TRS $\mathscr{T}_M$ such that left–outer (weak-)β-reduction from M to (wh)nf is isomorphic to supercombinator reduction in $\mathscr{T}_M$ from $\mathscr{L}(M)$, and this yields an efficient implementation by term graph rewriting. We show this is naïvely achieved by supplying fresh variables to *stuck* terms, (λ-abstractions resp. supercombinators with too few arguments) and recast it as naïve term graph rewriting *modulo the* ж*-calculus*.

1 A naïve implementation of β-reduction to nf via whnf

We proffer a rewriting perspective on implementing the $\lambda\beta$-calculus. Since in the literature other perspectives are dominant, notably the machine view, cf. the SECD-machine [38] and the Krivine machine [34], and the semantic view as represented by normalisation-by-evaluation [14], we first make the case for our rewriting view. Throughout, we assume familiarity with the λ-calculus [11, 64], term rewriting [9, 64], and term graph rewriting [12]; if neither a definition / lemma nor an explicit reference for a notion / result is given, it can be found in those works or our accompanying code [55]. Additional clarifying notes, organised per section, can be found in the appendices. Fig. 1 presents a naïve but fully

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

Figure 1: Naïve β-reduction of λ-terms to whnf and to nf by left–outer reduction in Haskell

functional Haskell implementation of reduction to (weak head) normal form in the (weak) $\lambda\beta$-calculus by means of the left–outer strategy on (morally)[1] *closed* λ-terms. The (partial) function `whnf` **implements**

[1] It fails on $(\lambda xy.x)\,y$ but succeeds on $(\lambda xy.x)\,z$ as substitution is *naïve*, performs no α-renaming; z acts as a constant, y not.

Submitted to:
TERMGRAPH 2024

weak head β-reduction to weak head normal form [59]; λ-terms of shape $\lambda x.t$ or $x\vec{t}$, and the (partial) function `nf` **implements** β-reduction to normal form [11]; λ-terms recursively in whnf, based on the *lo* (left–outer; *leftmost* or *normal* order) strategy. We implement `nf` on top of `whnf` in a *context-free* [64, Def. 9.1.29] way (§) recursing on direct subterms of whnfs. On whnfs of shape $\lambda x.t$ this is implemented by calling `nf` recursively on $(\lambda x.t)\,x_i$ and returning $\lambda x_i.s$ if that outputs s, where subscripting makes x_i *fresh* (onus on user)(‡). To strike a balance between Haskell and mathematical notation [11], we (1) omit empty argument vectors, (2) let the body of λ-abstractions extend to the right as far as possible, (3) omit parentheses as much as possible without creating ambiguity.

Example 1. The self-applicator $\delta = \lambda x.xx$ and Church-numeral $\underline{1} := \lambda yz.yz$ are represented as:

```
delta = (Lam (Abs "x" (Lam (Var "x") [Lam (Var "x") []])) [])
  one = (Lam (Abs "y" (Lam (Abs "z" (Lam (Var "y") [Lam (Var "z") []])) [])) [])
```

Evaluating `nf (Lam (Abs "x" (Lam (Var "x") [Lam (Var "x") []])) [one])` represents β-normalising $\delta\,\underline{1}$ yielding `Lam (Abs "z_1" (Lam (Abs "z_2" (Lam (Var "z_1") [Lam (Var "z_2") []])) [])) []`. Evaluating $\delta\,\delta$, i.e. `nf (Lam (Abs "x" (Lam (Var "x") [Lam (Var "x") []])) [delta])` loops.

Below we use code [55] that is less clunky, it offers abbreviations and pretty-printing.[2] We merely wanted to showcase that a rewrite-based implementation can rival any other qua *compactness* and *simplicity*. But now we show these ideas extend to *implementing* λ-calculus *efficiently* by graph rewriting along the lines of [10, 24] (†).

2 Implementations

Say a rewrite system [64, Defs. 8.2.2] **implements** another if they are isomorphic (so on objects and on steps). Note that isomorphism is a very strong form of implementation; one usually makes do with weaker notions [62, 13]. We illustrate our notion of implementation by means of the rewrite systems in play here, the $\lambda\beta$-calculus [11], supercombinators [31, 59], and term graph rewrite systems [64, Ch. 13], and translations between them [41, 10, 24].

From the λ-calculus to supercombinators (†) we base our developments on a syntax encompassing both, PRSs [43] [64, Ch. 11]. To stay as light-weight as possible we adopt notational conventions that are a mixture of those for the λ-calculus, leaving application implicit and using *parentheses* if association to the left is not intended (as above), and those for term rewriting but using *square brackets* to enclose arguments. For each rule of a PRS $\mathscr{P}$ we assume there is a corresponding rule *symbol* (§) in the signature [64, Sec. 8.2.2]. The *$\mathscr{P}$-multistep* rewrite system $\multimap_{\mathscr{P}}$ has $\mathscr{P}$-terms with variables of base type over that signature (without rule-symbols) as steps (objects), and src / tgt are the homomorphic extensions of mapping a rule-symbol to its lhs / rhs.[3] Restricting multisteps to have single occurrences of rule-symbols yields $\to_{\mathscr{P}}$ (*single*) steps, and to occurrences at disjoint positions $\pitchfork_{\mathscr{P}}$ *parallel* steps.[4] We use $\Phi,\Psi,X,\ldots$ to range over (non-empty) multisteps and parallel steps and $\phi,\psi,\chi,\ldots,\ldots$ for steps. Reductions arise as usual from this, as (possibly empty) sequences of steps, and will be denoted by doubling the arrowhead; e.g. $\twoheadrightarrow_{\mathscr{P}}$ denotes a $\mathscr{P}$-reduction of $\mathscr{P}$-steps $\to_{\mathscr{P}}$. The rewrite systems for the λ-calculus arise from adjoining the rule-symbol β to the PRS [43, Ex. 3.4][64, Ex. 11.2.6(i)] having a signature comprising application and abstraction, with src and tgt the extensions of mapping $\beta[x.M[x],N]$ to $(\lambda x.M[x])\,N$ and $M[N]$.

Example 2. Let $\delta := \lambda x.xx$ and $\Omega := \delta\,\delta$ as usual, in the λ-term $M := \lambda y.(\lambda z.y)\,\Omega$. There are 4 multisteps *from* M, i.e. having M as source: M (the *empty* multistep) and $\phi := \lambda y.(\lambda z.y)\,\beta[x.xx,\delta]$ (a *loop*) both *to* M, i.e. having M as target, and $\psi := \lambda y.\beta[z.y,\Omega]$ and $X := \lambda y.\beta[z.y,\beta[x.xx,\delta]]$ both to $\lambda y.y$.

That there are 4 multisteps $\multimap_{\beta}$ from M corresponds to the fact that in the classical view [11] M contains 2 β-redexes, giving rise to $2^2 = 4$ subsets that can be contracted / completely developed in one go. The second step ϕ and third ψ are β-steps as they contain exactly one rule-symbol, contract one β-redex. The fourth X is a β-multistep but *not* a parallel β-step because the Ω-redex is *nested* inside the redex $\beta[z.y,\beta[x.xx,\delta]]$, as $\beta[x.xx,\delta]$.

We use wCHβ (*weak-CH-β* [19]) to denote the restriction of β to redexes not containing variables bound outside them, and wβ (*weak-β* [59]) the further restriction to not being a redex below a λ-abstraction at all.

[2] The experiments reported here were done with GHCi 8.10.7 on a MacBook Pro (2019, 16 inch, Intel) with macOS 13.6.

[3] Cf. [52, Sec. 3] for a progenitor of this idea, of introducing rule-symbols to have *steps as terms / graphs* [64, Rem. 9.4.30].

[4] Beware the definition of *parallel* step differs in the literature on the λ-calculus and TRSs; ours is that of the latter.

Example 3. In Ex. 2 ϕ is weak-CH-β but not weak-β, and ψ is β but neither weak-β nor weak-CH-β. We have M is in weak-β-nf, but loops $M \to_{\mathsf{wCH}\beta} M$ and in fact M has no weak-CH-β-nf, yet $\psi : M \to_\beta \lambda y.y$, which is in β-nf. (The first and last can be checked by evaluating `whnf` and `nf` on `example2M` in the Haskell code of [55].)

Given a signature comprising application and a finite number of supercombinators κ_i, a *supercombinator* rewrite system arises by adjoining for every supercombinator κ_i of arity n, a rule-symbol γ_i of arity $n+1$, with src and tgt the extensions of mapping $\gamma_i[x_1,\ldots,x_n,x_0]$ to $\kappa_i[x_1,\ldots,x_n]x_0$ respectively r, with r a term over $x_0,\ldots,x_n$ and the supercombinators it *depends* on. Dependency is required to be well-founded (for the system as a whole).

Example 4. Using the format $rule : lhs \to rhs$, the following constitutes a supercombinator system $\mathscr{T}$:

$$\gamma_0[y,x] : \kappa_0[y]\,x \;\to\; y \qquad \gamma_1[x] : \kappa_1[]\,x \;\to\; x\,x \qquad \gamma_2[y,x] : \kappa_2[y]\,x \;\to\; \kappa_0[x]\,y$$

Dependency is well-founded indeed: the only dependency is of κ_2 on κ_0 as seen by the rhs of the last rule.

Stuck terms, of shape $\kappa_i[\vec{t}]$, are the supercombinator equivalent of λ-abstractions. Just like the latter, they can come unstuck by supplying a further argument; *fresh* variable arguments allow to *observe* the function body / rhs:

Example 5. For the supercombinator system of Ex. 4, the supercombinator term $t := \kappa_2[\kappa_1[]\,\kappa_1[]]$ only allows the looping step $\phi' := \kappa_2[\gamma_1[\kappa_1[]]]$ (checked by evaluating `swchnf` on arguments `example4trs` and `example4trm`). The term t is stuck. Supplying the fresh variable y, allows to reach an nf / to observe its function body; informally: $t \to_y \kappa_0[y\,(\kappa_1[]\,\kappa_1[])] \to y$ where the first step is enabled by supplying y and the second step is $\gamma_0[y,(\kappa_1[]\,\kappa_1[])]$.

Since that $t\,y$ reduces to the nf y means t is (β-)equivalent to $\lambda y.y$, the nf-computations in Ex. 3 and 5 are in fact the same. This is no coincidence, but a consequence (†) of that the supercombinator system $\mathscr{T}$ and the term t were obtained by *lifting* the λ-term M (evaluating `lift example2M` in our code yields the pair of (the rhss of) the supercombinator system $\mathscr{T}$ and t). We *describe* lifting, referring to [65, 31, 59, 10, 24, 55] for details and code:

Description 1. *Lifting* $\mathscr{L}$ transforms a λ-term M into a TRS $\mathscr{T}_M$ and a term $\mathscr{L}(M)$ over its signature. It acts homomorphically on variables and applications, but on a λ-abstraction subterm $\lambda x.N$, first N is recursively lifted and then the resulting supercombinator term is split (by `split`) into the vector of its *maximal x-free subexpressions* and its *skeleton* r (having only xs at its leaves), giving rise to a new supercombinator with arity the length of the vector, and rule for it having r as rhs. (By M being finite $\mathscr{L}$ terminates, with well-founded dependencies.) *Expanding* $\mathscr{E}$ transforms a supercombinator term back into a λ-term by recursively mapping each n-ary supercombinator κ_i with rhs r_i to its λ-abstracted rhs $\lambda[x_1,\ldots,x_n]x_0.r_i$. ($\mathscr{E}$ terminates by well-foundedness of dependencies.)

Lifting commutes with wβ-reduction in the sense that if $M \to_{\mathsf{w}\beta} N$ then $\mathscr{T}_N$ is a sub-TRS (up to naming of supercombinators) of $\mathscr{T}_M$ and $\mathscr{L}(M) \to_{\mathscr{T}_M} \mathscr{L}(N)$ (up to naming again). From that, the implementation result (†) of [10, 24] follows; using our not(at)ions and letting $\langle a \to \rangle$ denote the restriction of a rewrite system $\to$ to objects reachable from a, it can be succinctly expressed as that $\langle \mathscr{L}(M) \to_{\mathscr{T}_M} \rangle$ **implements** $\langle M \to_{\mathsf{wCH}\beta} \rangle$. That could be (para)phrased, cf. [41], as the catchy slogan **lazy functional programming is orthogonal term rewriting**. There's a catch however. Ex. 3 exhibits a discrepancy between wβ as in lazy functional programming and wCHβ as in the implementation result: the latter may not terminate in cases where the former does (even if it has a β-nf). Our naïve idea (‡) to reconcile wβ with wCHβ, reflected in the code above and in [55], is based on two observations:

First, the left–outer strategies for wβ and wCHβ coincide on any λ-term M *until reaching a whnf*; only after wβ and wCHβ may diverge. Lifting such β-reductions to whnf yields left–outer supercombinator reductions from $\mathscr{L}(M)$ to whnf, where whnf now means a supercombinator term that is either stuck or of shape $x\vec{t}$. Using whβ and wh to denote the respective weak head steps, we thus have $\langle \mathscr{L}(M) \to_{\mathsf{wh}} \rangle$ **implements** $\langle M \to_{\mathsf{wh}\beta} \rangle$.

Second, both λ-abstractions and stuck supercombinator terms may be released by applying them to a fresh variable as in Ex. 5. $\mathscr{L}$ commutes with such α*-steps* as it acts homomorphically on applications and variables.[5] Using loβ and lo to denote the respective left–outer steps, where on whnfs we either perform an α-step supplying a fresh variable and repeat, or recurse on the subterms (note that being in whnf or not is trivially decidable and whnfs are stable under reduction [23, 45], in either case), we have $\langle \mathscr{L}(M) \to_{\mathsf{lo}} \rangle$ **implements** $\langle M \to_{\mathsf{lo}\beta} \rangle$.

Per construction, the generated supercombinator systems have a simple shape (†): they are orthogonal, left-normal, and all lhss of rules are in applicator–constructor format. That shape is shared by the λ-calculus and combinatory logic, all of them ISs (*interaction systems* [6, 7]) and *left–outer Dyck systems* [58] affording nice properties, e.g. *confluence* [64, Thm. 11.6.19] and (*hyper-*)*normalisation* of the *left–outer* strategy [58, Def. 27,Thm. 50]:

Lemma 1. *For nf M', we have M is β-convertible to M' iff M loβ-reduces to M' iff $\mathscr{L}(M)$ lo-reduces to M' in $\mathscr{T}_M$.*

[5] $\mathscr{L}$ would not act homomorphically if α-steps were to introduce λ-abstractions; think of λs as being adjoined *a posteriori*.

From supercombinators to maximal sharing graphs we view TRSs and TGRSs (*term graph rewrite systems* [64, Ch. 13][16, 60, 8]) as systems rewriting *structures* (terms resp. term graphs) modulo a *substitution* calculus (∗) [57, 50, 61][64, Sec. 11.3.2]. On that basis we implement TRSs by TGRSs [60, 64, 8, 24], by example:

Example 6. Let ρ be the rule $\ell := g[x,a] \to a =: r$ of a TRS $\mathscr{T}$. It can be instantiated by substituting an arbitrary term for x; the variable x is *implicitly* universally quantified in ρ. The idea is to make substitution *explicit* by means of *some* calculus. A naïve way to do so is to take the simply typed λ-calculus $\lambda^{\to}$;[6] in $\lambda^{\to}$ the substitution of s for x in t (implicit) can be brought about as the result of β-normalising $(x.t)\,s$ (we leave λ implicit). Accordingly, we λ-abstract x in both the lhs ℓ and the rhs r of the rule yielding a, now closed, rule $\overline{\rho} : x.g[x,a] \to x.a$; no free xs.

Contracting a $\overline{\rho}$-redex is then enacted in three stages: *matching* (β-expansion), followed by replacement (according to $\overline{\rho}$), and substitution (β-reduction). E.g., contracting the rightmost redex in $t := f[a, g[a,a], g[a,a]]$ by $f[a, g[a,a], \overline{\rho}\,a]$ proceeds as $t \underset{\beta}{\leftarrow} f[a, g[a,a], (x.g[x,a])\,a] \to_{\rho} f[a, g[a,a], (x.a)\,a] \twoheadrightarrow_{\beta} f[a, g[a,a], a] =: s$. Both redexes in t can be contracted in one go via the parallel step $\Phi := f[a, \overline{\rho}\,a, \overline{\rho}\,a]$. The *matching* stage serves to search and exhibit an occurrence of a lhs in a term, and the *substitution* stage serves to obtain a term, a *unique substitution-normal form*, by *expansion* respectively *reduction* steps of the explicit substitution calculus $\lambda^{\to}$. To implement $\mathscr{T}$

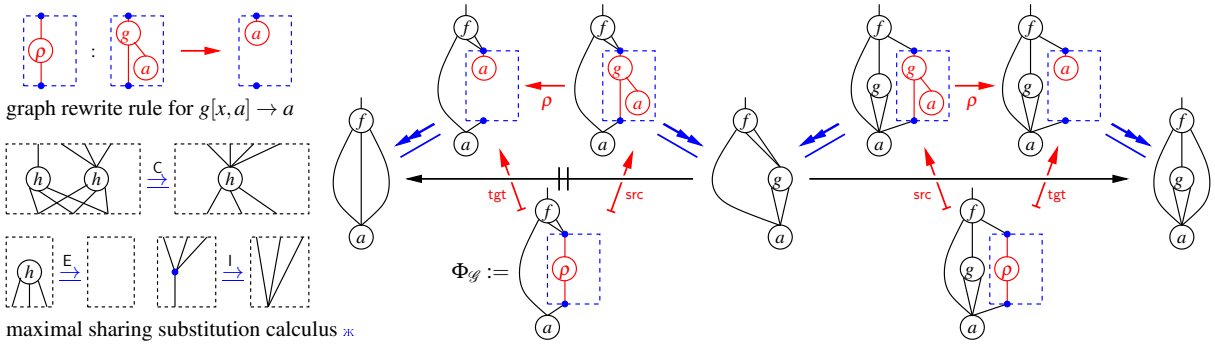

Figure 2: Maximal sharing term graph rewrite steps corresponding to $_{\mathscr{T}}\!\leftarrow\!\!\shortmid\!\!\shortmid\!\!-$ and $\to_{\mathscr{T}}$-steps from term t

via a TGRS $\mathscr{G}$ we employ msg's (*maximal sharing* directed term *graphs* [17, 60, 26]). Needing to implement *finite* terms only, makes we are in a sweet spot: msgs are finite and acyclic and only have *horizontal* sharing [16]. As signature of $\mathscr{G}$ we take an *indirection* symbol • [16, Sec. 3.3] having 1 output and for each function / rule-symbol h of arity n of the TRS $\mathscr{T}$, a corresponding symbol having n outputs. The three rule schemes of the substitution calculus ж are given at the bottom–left in Fig. 2, cf. [16, Fig. 3.9]), and we denote ж-steps and reductions by $\underline{\to}$ and $\underline{\twoheadrightarrow}$. Given a TGRS $\mathscr{G}$ its multisteps $\multimap_{\mathscr{G}}$ (objects) are msgs[3] over the signature (without rule-symbols) with src / tgt on a msg $\mathscr{G}$ defined by first replacing each rule-symbol by its lhs / rhs (see the top–left of Fig. 2 for ρ), followed by $\underline{\to}$-normalisation yielding a unique msg (up to graph isomorphism). Parallel steps $\shortmid\!\!\shortmid\!\!\to_{\mathscr{G}}$ are multisteps where rule-symbols are not reachable from each other, and (single) steps $\to_{\mathscr{G}}$ have exactly 1 rule-symbol to which there is 1 path from the root (the multistep toward the left in Fig. 2 is parallel since though ρ occurs only once it is reachable via 2 paths from the root; the one to the right is a step). By being only an alternate account of msgs in the literature $\langle t_{\mathscr{G}} \to_{\mathscr{G}} \rangle$ **implements** $\langle t \to_{\mathscr{T}} \rangle$ for any t in $\mathscr{T}$; cf. [60, 8, 24]. This extends seamlessly to parallel and multisteps, $\langle t_{\mathscr{G}} \shortmid\!\!\shortmid\!\!\to_{\mathscr{G}} \rangle$ **implements** $\langle t \shortmid\!\!\shortmid\!\!\to_{\mathscr{T}} \rangle$ and $\langle t_{\mathscr{G}} \multimap_{\mathscr{G}} \rangle$ **implements** $\langle t \multimap_{\mathscr{T}} \rangle$ (e.g. mapping the parallel $\mathscr{T}$-step $\Phi := f[a, \overline{\rho}\,a, \overline{\rho}\,a]$ above to its tree and collapsing that to an msg yields the parallel step $\Phi_{\mathscr{G}}$ as displayed (with indirection nodes inserted) in Fig. 2), and to strategies considered here owing to that they are *positional* [64, Ch. 9] and *positions* in a term t are the same as *access paths* in its msg $t_{\mathscr{G}}$; finally, prepending an @-node applied to a fresh variable, to an msg **implements** α-steps on stuck supercombinator terms.

From supercombinators to sharing graphs we now assume TRSs are orthogonal and have lhss in applicator–constructor shape [41] (†). We study minimising usage of the ж-calculus. To that end, we omit the E-rule and allow no C-reduction and C-expansion only to exhibit lhss [65] of rules as in Fig. 3, calling the resulting $\underline{\to}$-normal forms

[6] Beware, here $\lambda^{\to}$ is only used to model substitution in TRSs; it is independent of the (untyped) λ-calculus above.

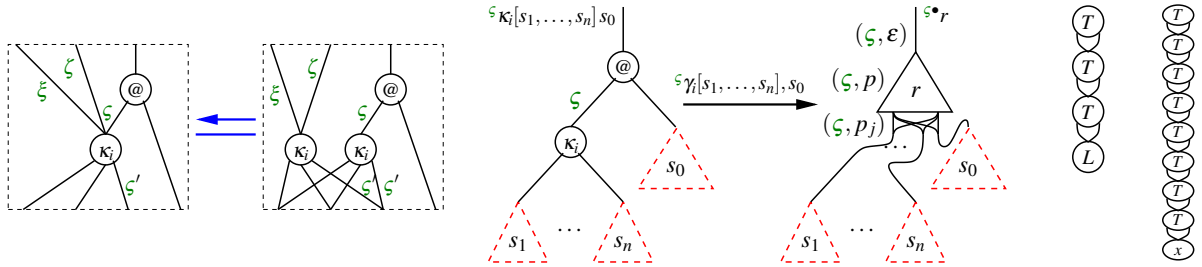

Figure 3: exhibiting redex-pattern (left) Lévy-labelling rule (middle) $((2^2)^2)^2$ and $\overbrace{2\cdot\ldots\cdot 2}^{8}$ xs (right)

sg's (*sharing graphs*). Observe then we have (1) no garbage-collection, (2) no maximisation of sharing *during* reduction, and (3) no unsharing of redex-patterns, only of constructors *in* them. Due to left-linearity steps cannot be blocked despite (2). By (3) each single step on an sg G corresponds to a parallel step contracting *some* redexes of the corresponding term t, but which ones? Maranget's precise answer [41] is: $\langle G \to_{\mathscr{G}} \rangle$ **implements** $\langle t \twoheadrightarrow_{\mathscr{F}} \rangle$ where $\mathscr{F}$ denotes that we contract *families*, all redex-patterns with the same *labelling*, as exemplified by:

Example 7. Renaming the result of lifting $M := \underline{2}\,\underline{2}$ so of `lift (ap two two)` for the Church numeral $\underline{2} := \lambda yz.y\,(y\,z)$ into more palatable TRS notation yields the term $t := @[L,L]$ for rules $@[T[y,z],x] \to @[y,@[z,x]]$ and $@[L,x] \to T[x,x]$, where $@$ is application and T / L are the supercombinators for the λy- / λz-abstractions. Then $R\colon @[L,L] \to T[L,L] \to_x @[T[L,L],x] \to @[L,@[L,x]] \to T[@[L,x],@[L,x]] \twoheadrightarrow T[T[x,x],T[x,x]]$ by lo-reduction supplying the fresh variable x to the stuck term $T[L,L]$. Supplying another fresh variables yields $\underline{4}$.

Labelling the rules per Fig. 3 enumerating the (5 resp. 3) subterms of the rhss yields: $@[{}^{\varsigma}T[y,z],x] \to {}^{(\varsigma,0)}@[{}^{(\varsigma,1)}y,{}^{(\varsigma,2)}@[{}^{(\varsigma,3)}z,{}^{(\varsigma,4)}x]]$ and $@[{}^{\varsigma}L,x] \to {}^{(\varsigma,0)}T[{}^{(\varsigma,1)}x,{}^{(\varsigma,2)}x]$. Labelling R for initially labelled term $t^I := {}^{a}@[{}^{b}L,{}^{c}L]$ yields (omitting the last step): $t^I \to {}^{\zeta}T[{}^{(b,1)c}L,{}^{(b,1)c}L] \to_x {}^{d}@[{}^{\zeta}T[{}^{(b,1)c}L,{}^{(b,1)c}L],{}^{e}x] \to {}^{d(\zeta,0)}@[{}^{\xi}L,{}^{(\zeta,2)}@[{}^{(\zeta,3)(b,2)c}L,{}^{(\zeta,4)e}x]] \to {}^{d(\zeta,0)(\xi,0)}T[{}^{(\xi,1)(\zeta,2))}s,{}^{(\xi,2)(\zeta,2)}s]$ where $s := @[{}^{(\zeta,3)(b,2)c}L,{}^{(\zeta,4)e}x]$ and $\zeta := a(b,0)$ and $\xi := (\zeta,1)(b,1)c$. The copies of s belong to the same *family* both having label $(\zeta,3)(b,2)c$.

3 Some naïve conclusions on complexity

Example 8. Sharing can be seen at play at the intermediate stages and their interpretation when computing $(\underline{2}\,\underline{2})\,(\underline{2}\,\underline{2})$ as shown in Fig. 3, after computing the wCHβ-nf (in 16 steps) and after supplying a fresh variable x (in 359 steps); supplying another y yields (in 256 steps) the Church numeral $\underline{256}$.

In msg rewriting of G each step can only cause *constant change* in width and height of G; e.g. change is not more than 1 in Ex. 7. It follows from the implementation results that measuring the complexity of a reduction to nf via its number of steps is reasonable [8, 2] for TRSs, and via lifting into a supercombinator system first, it also is for the lo-strategy in the λ-calculus. Despite that sg rewriting does not maximise sharing *during* a reduction it still has the constant change property, so again is reasonable, and (in the absence of family-duplication) lo is a *normalising* and *minimal* / *optimal* strategy in the standard sense of [53, Thm. 2].

We have argued that factoring reduction to nf in the λ-calculus through reduction to whnf (i) allows for a trivial implementation of the λ-calculus in Haskell, and (ii) makes it clear that loβ-reduction can be efficiently implemented via the standard techniques of lifting and maximal sharing graphs. Next, (iii) relying on standard theory for PRSs, and (iv) using graph rewriting modulo a substitution calculus, enabled connecting various implementation results in a smooth and compact way. Still this is about first-order TRS (encodings of λ-calculus) only; whether implemented in msgs, sgs, or via a let-construct, horizontal sharing can only do so much [4, Fig. 3]; our interest lies more in cyclic sharing and beyond.

References

[1] S. Abramsky (1990): *The Lazy λ-Calculus*. In D. Turner, editor: *Research Topics in Functional Programming*, Addison Wesley, pp. 65–117.

[2] B. Accattoli & U. Dal Lago (2016): *(Leftmost-Outermost) Beta Reduction is Invariant, Indeed*. *Logical Methods in Computer Science* 12(1), doi:10.2168/LMCS-12(1:4)2016.

[3] C. Appel, V. van Oostrom & J.G. Simonsen (2010): *Higher-Order (Non-)Modularity*. In C. Lynch, editor: *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11–13, 2010, Edinburgh, Scotland, UK*, *LIPIcs* 6, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 17–32, doi:10.4230/LIPIcs.RTA.2010.17.

[4] A. Asperti (2017): *About the efficient reduction of lambda terms*, doi:10.48550/arXiv.1701.04240.

[5] A. Asperti & S. Guerrini (1998): *The optimal implementation of functional programming languages*. *Cambridge Tracts in Theoretical Computer Science* 45, Cambridge University Press.

[6] A. Asperti & C. Laneve (1995): *Interaction Systems I: the theory of optimal reductions*. *Mathematical Structures in Computer Science* 4(4), pp. 457–504, doi:10.1017/S0960129500000566.

[7] A. Asperti & C. Laneve (1996): *Interaction Systems II: the practice of optimal reductions*. *Theoretical Computer Science* 159(2), pp. 191–244, doi:10.1016/0304-3975(95)00062-3.

[8] M. Avanzini & G. Moser (2010): *Closing the Gap Between Runtime Complexity and Polytime Computability*. In C. Lynch, editor: *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11–13, 2010, Edinburgh, Scotland, UK*, *LIPIcs* 6, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 33–48, doi:10.4230/LIPICS.RTA.2010.33.

[9] F. Baader & T. Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press.

[10] T. Balabonski (2012): *A unified approach to fully lazy sharing*. In J. Field & M. Hicks, editors: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, ACM, pp. 469–480, doi:10.1145/2103656.2103713.

[11] H.P. Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*, 2nd revised edition. *Studies in Logic and the Foundations of Mathematics* 103, North-Holland, Amsterdam.

[12] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, R. Kennaway, M.J. Plasmeijer & M.R. Sleep (1987): *Term Graph Rewriting*. In J.W. de Bakker, A.J. Nijman & P.C. Treleaven, editors: *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15–19, 1987, Proceedings*, *Lecture Notes in Computer Science* 259, Springer, pp. 141–158, doi:10.1007/3-540-17945-3_8.

[13] G. Barthe, J. Hatcliff & M.H. Sørensen (1997): *Reflections on Reflections*. In H. Glaser, P.H. Hartel & H. Kuchen, editors: *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Trach on Declarative Programming Languages in Education, Southampton, UK, September 3–5, 1997, Proceedings*, *Lecture Notes in Computer Science* 1292, Springer, pp. 241–258, doi:10.1007/BFB0033848.

[14] U. Berger & H. Schwichtenberg (1991): *An Inverse of the Evaluation Functional for Typed lambda-calculus*. In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15–18, 1991*, IEEE Computer Society, pp. 203–211, doi:10.1109/LICS.1991.151645.

[15] T. Blanc, J.-J. Lévy & L. Maranget (2005): *Sharing in the Weak Lambda-Calculus*. In A. Middeldorp, V. van Oostrom, F. van Raamsdonk & R.C. de Vrijer, editors: *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 3838, Springer, pp. 70–87, doi:10.1007/11601548_7.

[16] S.C.C. Blom (2001): *Term Graph Rewriting, syntax and semantics*. Ph.D. thesis, Vrije Universiteit Amsterdam. Available at `https://research.vu.nl/en/publications/term-graph-rewriting-syntax-and-semantics`.

[17] M. van den Brand & P. Klint (2007): *ATerms for manipulation and exchange of structured data: It's all about sharing*. *Information and Software Technology* 49(1), pp. 55–64, doi:10.1016/j.infsof.2006.08.009.

[18] A. Burroni (1993): *Higher-dimensional word problems with applications to equational logic*. *Theoretical Computer Science* 115(1), pp. 43–62, doi:10.1016/0304-3975(93)90054-W.

[19] N. Çağman & J.R. Hindley (1998): *Combinatory Weak Reduction in Lambda Calculus*. *Theoretical Computer Science* 198(1-2), pp. 239–247, doi:10.1016/S0304-3975(97)00250-8.

[20] J. Endrullis, C. Grabmayer, J.W. Klop & V. van Oostrom (2011): *On equal μ-terms*. *Theoretical Computer Science* 412(28), pp. 3175–3202, doi:10.1016/J.TCS.2011.04.011.

[21] S. Frontull, G. Moser & V. van Oostrom (2023): *α-Avoidance*. In M. Gaboardi & F. van Raamsdonk, editors: *8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 260, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 22:1–22:22, doi:10.4230/LIPIcs.FSCD.2023.22.

[22] J.-Y. Girard (1987): *Linear logic*. *Theoretical Computer Science* 50(1), pp. 1–101, doi:https://doi.org/10.1016/0304-3975(87)90045-4.

[23] J.R.W. Glauert & Z. Khasidashvili (1996): *Relative Normalization in Deterministic Residual Structures*. In H. Kirchner, editor: *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, Linköping, Sweden, April, 22-24, 1996, Proceedings*, *LNCS* 1059, Springer, pp. 180–195, doi:10.1007/3-540-61064-2_37.

[24] C. Grabmayer (2016): *Linear Depth Increase of Lambda Terms along Leftmost-Outermost Beta-Reduction*. *CoRR* abs/1604.07030, doi:10.48550/arXiv.1604.07030.

[25] C. Grabmayer & V. van Oostrom (2016): *Nested Term Graphs (Work In Progress)*. *CoRR* abs/1405.6380, doi:10.48550/arXiv.1405.6380.

[26] C. Grabmayer & J. Rochel (2014): *Maximal sharing in the Lambda calculus with letrec*. In J. Jeuring & M.M.T. Chakravarty, editors: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, ACM, pp. 67–80, doi:10.1145/2628136.2628148.

[27] Makoto Hamana (2022): *Complete algebraic semantics for second-order rewriting systems based on abstract syntax with variable binding*. *Mathematical Structures in Computer Science* 32(4), pp. 542–573, doi:10.1017/S0960129522000287.

[28] D. Hendriks & V. van Oostrom (2003): $\lambda\!\!\lambda$. In F. Baader, editor: *Proceedings of CADE 19*, *Lecture Notes in Artificial Intelligence* 2741, Springer, pp. 136–150, doi:10.1007/978-3-540-45085-6_11.

[29] N. Hirokawa, J. Nagele, V. van Oostrom & M. Oyamaguchi (2019): *Confluence by Critical Pair Analysis Revisited*. In: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings*, *Lecture Notes in Computer Science* 11716, Springer, pp. 319–336, doi:10.1007/978-3-030-29436-6_19.

[30] G. Huet (1997): *The Zipper*. *Journal of Functional Programming* 7(5), p. 549–554, doi:10.1017/S0956796897002864.

[31] R. J. M. Hughes (1982): *Super-combinators a new implementation method for applicative languages*. In: *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, Association for Computing Machinery, New York, NY, USA, p. 1–10, doi:10.1145/800068.802129.

[32] S. Kahrs (1995): *Towards a Domain Theory for Termination Proofs*. In J. Hsiang, editor: *Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5–7, 1995, Proceedings*, *Lecture Notes in Computer Science* 914, Springer, pp. 241–255, doi:10.1007/3-540-59200-8_-60.

[33] J.W. Klop (1980): *Combinatory Reduction Systems*. Ph.D. thesis, Rijksuniversiteit Utrecht.

[34] J.-L. Krivine (2007): *A call-by-name lambda-calculus machine*. *Higher-Order and Symbolic Computation* 20(3), pp. 199–207, doi:10.1007/s10990-007-9018-9.

[35] Y. Lafont (1990): *Interaction Nets*. In: *17th POPL*, ACM Press, pp. 95–108, doi:10.1145/96709.96718.

[36] Y. Lafont (1995): *From Proof-Nets to Interaction Nets*. In J.-Y. Girard, Y. Lafont & L. Regnier, editors: *Advances in Linear Logic*, *London Mathematical Society Lecture Note Series* 222, Cambridge University Press, pp. 225–248.

[37] J. Lamping (1990): *An Algorithm for Optimal Lambda Calculus Reduction*. In F.E. Allen, editor: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, ACM Press, pp. 16–30, doi:10.1145/96709.96711.

[38] P. J. Landin (1964): *The Mechanical Evaluation of Expressions*. *The Computer Journal* 6(4), pp. 308–320, doi:10.1093/comjnl/6.4.308.

[39] J. Leo (2014): *Thinking in a Coordinate-Free Way about Relations*. *Dialectica* 68(2), pp. 263–282. Available at `https://www.jstor.org/stable/42968507`.

[40] J.-J. Lévy (1978): *Réductions correctes et optimales dans le λ-calcul*. Thèse de doctorat d'état, Université Paris VII. Available at `http://pauillac.inria.fr/~levy/pubs/78phd.pdf`.

[41] L. Maranget (1991): *Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems*. In: *Proceedings of the 18th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, POPL '91, Association for Computing Machinery, New York, NY, USA, p. 255–269, doi:10.1145/99583.99618.

[42] C. Marché (1996): *Normalized Rewriting: An Alternative to Rewriting Modulo a Set of Equations*. *Journal of Symbolic Computation* 21(3), pp. 253–288, doi:10.1006/JSCO.1996.0011.

[43] R. Mayr & T. Nipkow (1998): *Higher-Order Rewrite Systems and their Confluence*. *Theoretical Computer Science* 192, pp. 3–29, doi:10.1016/S0304-3975(97)00143-6.

[44] P.-A. Melliès (1996): *Description Abstraite des Systèmes de Réécriture*. Thèse de doctorat, Université Paris VII. Available at `http://www.irif.fr/~mellies/phd-mellies.pdf`.

[45] P.-A. Melliès (1998): *A Stability Theorem in Rewriting Theory*. In: *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*, IEEE Computer Society, pp. 287–298, doi:10.1109/LICS.1998.705665.

[46] J. Meseguer (1992): *Conditional rewriting logic as a unified model of concurrency*. *Theoretical Computer Science* 96, pp. 73–155, doi:10.1016/0304-3975(92)90182-F.

[47] A. Middeldorp, V. van Oostrom, F. van Raamsdonk & R.C. de Vrijer, editors (2005): *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*. *Lecture Notes in Computer Science* 3838, Springer, doi:10.1007/11601548.

[48] M.H.A. Newman (1942): *On theories with a combinatorial definition of "equivalence"*. *Annals of Mathematics* 43, pp. 223–243, doi:10.2307/2269299.

[49] S. Okui (1998): *Simultaneous Critical Pairs and Church–Rosser Property*. In T. Nipkow, editor: *RTA-98*, *LNCS* 1379, Springer, pp. 2–16, doi:10.1007/BFb0052357.

[50] V. van Oostrom (1994): *Confluence for Abstract and Higher-Order Rewriting*. Ph.D. thesis, Vrije Universiteit, Amsterdam. Available at `https://research.vu.nl/en/publications/confluence-for-abstract-and-higher-order-rewriting`.

[51] V. van Oostrom (1997): *FD à la Melliès*. Available at `http://www.javakade.nl/research/ps/FDalaMellies.ps`. Vrije Universiteit Amsterdam.

[52] V. van Oostrom (1997): *Finite Family Developments*. In H. Comon, editor: *Rewriting Techniques and Applications, 8th International Conference, RTA-97, Sitges, Spain, June 2–5, 1997, Proceedings*, *Lecture Notes in Computer Science* 1232, Springer, pp. 308–322, doi:10.1007/3-540-62950-5_80.

[53] V. van Oostrom (2007): *Random Descent*. In: *RTA*, *LNCS* 4533, Springer, pp. 314–328, doi:10.1007/978-3-540-73449-9_24.

[54] V. van Oostrom (2023): *On Causal Equivalence by Tracing in String Rewriting*. In C. Grabmayer, editor: Proceedings Twelfth International Workshop on *Computing with Terms and Graphs*, Technion, Haifa, Israel,

1st August 2022, *Electronic Proceedings in Theoretical Computer Science* 377, Open Publishing Association, pp. 27–43, doi:10.4204/EPTCS.377.2.

[55] V. van Oostrom (2024): *A naïve rewrite-based implementation of the λ-calculus*. Available at `http://www.javakade.nl/research/haskell/is.hs`. Haskell source file.

[56] V. van Oostrom, K.-J. van de Looij & M. Zwitserlood (2004): *Lambdascope Another optimal implementation of the lambda-calculus*. Extended Abstract for the Workshop on Algebra and Logic on Programming Systems (ALPS), Kyoto, April 10th 2004. `http://www.javakade.nl/research/pdf/lambdascope.pdf`.

[57] V. van Oostrom & F. van Raamsdonk (1994): *Weak Orthogonality Implies Confluence: The Higher Order Case*. In: *LFCS'94*, *LNCS* 813, Springer, pp. 379–392, doi:10.1007/3-540-58140-5_35.

[58] V. van Oostrom & Y. Toyama (2016): *Normalisation by Random Descent*. In: *FSCD*, *LIPIcs* 52, pp. 32:1–32:18, doi:10.4230/LIPIcs.FSCD.2016.32.

[59] Simon L. Peyton Jones (1987): *The Implementation of Functional Programming Languages*. Prentice Hall.

[60] D. Plump (2002): *Essentials of Term Graph Rewriting*. *Electronic Notes in Theoretical Computer Science* 51, pp. 277–289, doi:10.1016/S1571-0661(04)80210-X. GETGRATS Closing Workshop.

[61] F. van Raamsdonk (1996): *Confluence and Normalisation for Higher-Order Rewriting*. Ph.D. thesis, Vrije Universiteit Amsterdam. Available at `https://research.vu.nl/en/publications/confluence-and-normalisation-of-higher-order-rewriting`.

[62] A. Sabry & P. Wadler (1997): *A Reflection on Call-by-Value*. *ACM Transactions on Programming Languages and Systems* 19(6), pp. 916–941, doi:10.1145/267959.269968.

[63] S.D. Swierstra (2013): *Informatica: de Kunst van het Abstraheren*. Available at `https://dspace.library.uu.nl/bitstream/handle/1874/286829/Swierstra_afscheidsrede.pdf`. Afscheidsrede.

[64] Terese (2003): *Term rewriting systems*. *Cambridge Tracts in Theoretical Computer Science* 55, Cambridge University Press.

[65] C.P. Wadsworth (1971): *Semantics and Pragmatics of the Lambda-Calculus*. Ph.D. thesis, University of Oxford.

[66] J. Waldmann (2018): *When You Should Use Lists in Haskell (Mostly, You Should Not)*. *CoRR* abs/1808.08329, doi:10.48550/arXiv.1808.08329.

A Remarks on Sec. 1

Naïveté I refer to the implementations and techniques as *naïve*, first of all because they were the first thing that came to my mind / I tried, but second because several of the insights on which these are based go all the way back to my scientific birth (cf. the etymology of naïve) in 1994, and almost all of them are prepubescent, from before 2005. Accordingly, I have included references throughout the text tracing, to the best of my knowledge, my intellectual history w.r.t. these insights, to the following earlier works.

(∗). rewriting modulo a substitution calculus [57, 50, 61] (1994);

(†). review as presented in App. C (2005);

(‡). optimal implementation (with Java implementation) of λ-calculus using a single *scope* node [56] (2004);

(§). abstract notions of strategy, labelling, rules as symbols [64] (2003);

(¶). generating fresh variables via updating [28, 56] (2002);

Of course, to stay in the metaphor, I grew up in a rich environment where I was exposed early on to many of the key players and their ideas on the topics pertaining to this note, Lévy, Gonthier, Asperti, Laneve, Danos, Regnier, Girard, Plump, etc. and interacted with them and many co-authors and other colleagues, as witnessed by the very incomplete but already quite lengthy list of references.

In these appendices I give some further background information on the ideas in the main text, lest it be forgotten.

On Wadsworth's characterisation of λ-terms In the implementation in Fig. 1 and [55] a λ-term has shape $h\vec{t}$ comprising a *head h*, which either is *variable x* or a *λ-abstraction* $\lambda.t$ with *body t*, *applied* to a vector $\vec{t}$ of λ-terms, its *arguments*. This facilitates checking whnf-hood (since we have direct access to the tip of the spine so to say): a λ-term is not in whnf iff its head is a λ-abstraction and its vector of arguments is non-empty; then the head and the first argument constitute a weak head redex. The representation is based on Wadsworth's characterisation of Church's classical notion of λ-terms [11]. The three term-formers (variable, application, λ-abstraction) of the latter can be embedded into the former by a natural skeuomorphism W: the embedding $\mathsf{W}(MN)$ is obtained by appending the single λ-term $\mathsf{W}(N)$ to the vector of arguments of $\mathsf{W}(M)$, and variables and λ-abstractions are embedded by supplying them with an empty vector of arguments. Accordingly, we coerce a head h into a λ-term by supplying it with an empty vector ε of arguments.

Remarks on how α-conversion is (for whnfs) and is not (for nfs) avoided In line with that β-reduction to whnf of expressions in lazy functional programs, i.e. closed expressions, does not require to α-rename variables [59], contraction of weak head redexes only employs *naïve* [21] substitution (`subst` in Fig. 1).

On the other hand, when trying to compute the normal form of $\lambda x.t$ we can't directly recurse on t since weak head reduction cannot reduce under a λ. That couldn't even work since α-renaming may be essential when β-reducing to nf as observed by Hindley, e.g. for $\delta\underline{1}$ as in Ex. 1 [21].

The (standard) solution via *freshness* we explored initially (‡) in our Java implementation of [56]. It serves to express that *nothing is known* about the object denoted by the supplied argument x_i (cf. applicative bisimulation [1]); if it were to occur in t already, that could be violated, indicating a *dependence*. For instance, for $t = \lambda y.x$ setting $x_i := y$ wouldn't work (as y already occurs in t) but setting $x_i := z$ does. Indeed of the resulting nfs $\lambda y.\lambda y.y$ and $\lambda z.\lambda y.z$ only the latter is as intended; in the former the supplied argument y was *captured* yielding a non-intended function (as can be seen by supplying 1 and 0 to both yielding 0 respectively 1).

It could be considered conceptually wrong that x_i is a fresh *variable* since substituting it then yields an *open* term, violating the invariant of operating on closed λ-terms, but we perceive *freshness* to mean such a variable is a *constant*, a variable (implicitly) bound in the context. Thus starting from a *closed* λ-term (without free variables) as we assume, only ever closed λ-terms occur during normalisation. (Cf. [20] for an extensive discussion offering various perspectives, on dealing implicitly and explicitly with contexts, in the setting of μ-terms.)

The same was described in [56] and in our (Java) implementation of it (‡), the idea being that there's in fact no need to go full De-Bruijn-indices in implementations.[7] In fact, as pioneered and expounded in [28] one can simply keep variable names around, and supply them with an index updated on demand (¶). (In our code [55] this is achieved via *update* functions, the first parameter of `rnf` / `srnf` when β / supercombinator reducing to normal form.) Accordingly, we perceive x_i as the name x to which an index i is affixed (subscripted). In this way, we keep track of the original variable name x facilitating debugging, program slicing etc.; e.g. in the resulting nf in Ex. 1 z occurs twice, but with distinct indices 1 and 2, reflecting that they both originated with z in $\underline{1}$, but also that they had to be α-renamed apart, to avoid a name clash as discussed above.

Why Haskell? Why not? We have chosen Haskell (and even use lists in spite of [66]) for its descriptive simplicitly. Though Haskell is particularly well-suited to deal with inductive structures like terms, an implementation in say Java or C should not be much less compact.

B Remarks on Sec. 2

We recall for convenience the main ingredients of the definitions of rewrite system and strategy from [64].

[7] Everyone will have encountered inscrutable error messages as the result of the implementation using De Bruijn indices; messages that contain variable names that seemingly have nothing to do with the code one has written.

Definition 1. A *rewrite* system [64, Defs. 8.2.2] comprises sets A of objects $a,b,c,\ldots$ and Φ of steps $\phi,\psi,\chi,\ldots$, with each step ϕ having a *source* and a *target* object, $\mathsf{src}(\phi)$ and $\mathsf{tgt}(\phi)$. A *strategy* [64, Def. 9.1.1] is a *sub*-rewrite system, i.e. subsets of A and Φ with restrictions of src and tgt, that has the same set of normal forms. A *history-aware* strategy [64, Def. 9.1.1] is a strategy for a rewrite[8] labelling of it, where a *rewrite* labelling $\to'$ of $\to$ combines a *labelling* of the objects such that given any labelling a' of an object a and any step $\phi : a \to b$, there's a unique labelled step $\phi' : a' \to' b'$ for b a labelling of b, with a function mapping each $\to$-object to an $\to'$-objects, its *initial* labelling.

The formal definition of rewrite systems as collections of objects and steps equipped with source and target maps from the latter to the former, is quite abstract. As a consequence, it has been invented many times over in various fields, hence goes under various names, e.g. programming language theorists might call them *abstract machines*, algebraic or categorical mathematicians *quivers*, and mathematicians in graph theory *multidigraphs*.

Remark 1. Despite finding many *examples of labellings* when doing our background research for [64, Sec. 8.4], at the time we didn't find a general definition, so we came up with definitions of what constitutes a labelling ourselves, both for abstract and term rewriting. Coming up with a new definition is a hazardous activity, so we would be interested in learning of (non-)use cases.

A research question on labellings in structured rewriting systems is: does the maximal labelling as used in the main text for characterising steps in sharing graphs in terms of parallel steps on terms, *always* characterise permutation equivalence? In the case of term rewriting systems and the λ-calculus it does; the maximal labelling is known there as the Lévy labelling (see the main text [40]), and allows to *reconstruct* reductions from the Lévy labelling of their final term, but only *up to permutation equivalence*. Intuitively, much like in a hologram each symbol in the final term carries in principle the whole reduction-history as label, but only as far as it could perceive / see that, only its own *causal* history, cf. [54]; since labels / histories of symbols are coherent among each other (by their nature of belonging to the same term), the *global* history, the history of the term as a whole, can be reconstructed from the *local* histories, the history of its symbols, but only up to *causal independence*. It should be interesting to investigate whether these results extend to various graph rewriting formats. E.g. we expect they do for Lafont's interaction nets [35, 36] and for string diagrams.

What is presented by a rewrite system? That we are interested in the rewrite system underlying the λ-calculus system only up to isomorphism, not only enables to speak of their implementation by means of rewrite systems that have objects other than λ-terms (here: supercombinator terms and term graphs, both maximal sharing graphs msgs and sharing graphs sgs), it also allows to view them just as means to *present* that underlying rewrite system. This allows to rephrase the question (we learned from Klop [11]) whether some graph is the reduction graph of a λ-term, as a presentation question. For instance, one may ask for each of the 6 systems in Fig. 4 whether

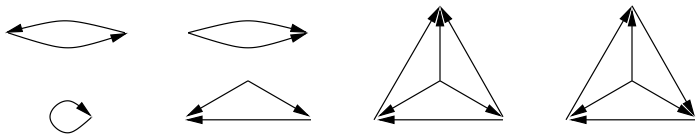

Figure 4: Can a rewrite system be presented as a rewrite system below M, for some λ-term M?

it could be presented as the rewrite system below some λ-term M, where *below* signifies the restriction of $\to_\beta$ to objects $\to_\beta$-reachable from M; the system bottom–left can be presented by $\Omega := \delta\,\delta := (\lambda x.xx)\,(\lambda x.xx)$ since $\Omega \to_\beta \Omega$ (and there are no other $\to_\beta$-steps) and the one next to it by $M := (\lambda y.\mathsf{I})(\mathsf{II})$ where $\mathsf{I} := \lambda x.x$. In both cases, there are many other λ-terms that present these rewrite systems; For others it is less clear whether they can be so presented, so left as an exercise.

The concept of *presenting* a system is pervasive in the algebraic literature. To mention a few basic examples viewed through a rewriting lense:

[8] The word 'rewrite' is inadvertently missing in [64, Def. 9.1.1]; we *must* have an initial labelling on top of the labelling.

- a Hasse diagram can be seen as a rewrite relation $\to$ presenting a partial order $\twoheadrightarrow$;
- a rewrite system $\to$ can be seen as a presentation of the category $\twoheadrightarrow$ of its (possibly-empty) reductions; and
- a string rewrite system can be seen as a presentation of the monoid having convertibility-classes of strings as elements, the class of the empty string as unit, and concatenation modulo as product.

The first two items highlight the divide one finds in the rewriting literature between whether, say, a term rewriting systems presents a rewrite *relation* or a rewrite *system*, with the former arising as the special case of the latter where there's at most one step from any object to another (cf. quasi-orders vs. categories). Since in the former view we don't have the ability to say that there are *distinct* steps from one object to another, to express whether steps are *equal* or not, to speak about so-called *syntactic accidents* [40], we adhere to the latter.[9]

Footnote 3: steps as structures over rule-symbols In view of the above, and adhering to having steps as first-class citizens immediately raises the question: what are the *steps* of the λ-calculus or of a supercombinator / term rewrite system? This question is not often addressed in the literature, let alone that a generally accepted answer exists currently. We proposed in [64, Sec. 8.2.2] that steps are *terms* over the signature extended with *rule*-symbols. Those terms we dubbed *proof* terms[10] as they can be viewed as witnessing a proof in rewrite logic [46] that one can term rewrite from its *source* to its *target*. This leaves another choice though, namely *how many* rule-symbols may occur in a proof term, how much work can be done in a single step? We construe it as an advantage that this choice exists at all, as it allows to express the usually disjointed notions of *single* step $\to$ and *parallel* step $\mathrel{+\!\!+\!\!\to}$ as both arising as natural restrictions of the notion of *multistep* $\mathrel{-\!\!\circ\!\!\to}$ [64, Prop. 8.2.22]. The idea of steps as the structures at hand to which rules are adjoined as basic constructs, as symbols, extends to other structured rewrite formats, as elaborated here a bit further for term graph rewriting, as was suggested in [64, Rem. 9.4.30].

Of course, right from the start adjoining symbols (labels, underlining, markings, ...) to the signature for marking purposes is pervasive in the rewriting literature. But of the general idea to have *steps* as structures over the signature extended with *rule-symbols*, we found (apart from in our own work) only few instances, mainly in the categorical literature on (higher-dimensional) string rewriting (where the perspective is the opposite, namely of *non*-rule-symbols as representing whiskering). We would be interested in learning of more use-cases.

A related research question on adjoining symbols is why variables are omnipresent in term rewriting but seem to be largely absent in graph rewriting. They would seem to give a principled approach to graph rewriting; e.g. the account of term graph rewriting in the main text naturally factors through them, by reifying the boxes (which at the moment are just visual tools, cf. [25]) into variables that can be substituted for. See Fig. 10 for a picture of the rules for a *mix* operation, simultaneously computing the minimum and maximum of two unary natural numbers,[11] using 'open' rules and Fig. 11 for an idea to 'close' such rules by enclosing their lhs and rhs in a box, name its complement (*context* / *congraph*?) and then turn that into a variable node.[12] Some such mechanism seems of interest to interaction nets / string diagrams.

Why PRSs? We employ Nipkow's PRSs (higher-order pattern rewrite systems) as our term rewriting format because we prefer them, but any term rewriting format that is at least second-order and encompasses fully-extended (no-occur-checks need not be expressible) left-normal orthogonal systems where left-hand sides comprise exactly two symbols (an application symbol combined with an abstraction symbol for the β-rule and with a supercombinator for the γ-rules), will do, e.g. Klop's CRSs (combinatory reduction systems [33]) as were used by Balabonski

[9] Though Newman expounded the rewrite–system-view in [48] (using terminology from his research on homotopy), explicitly stating that he was moving away from relations / orders, the rewrite–relation-view had become dominant in the 80s and 90s during the heydays of term rewriting research. Nowadays we see things changing back again, due to tool, formalisation, and certification efforts and the advent of higher category theory [18], with notions such as critical *peaks* instead of critical *pairs*.

[10] Proof terms as presented there also have operations corresponding to the inference rule for transitivity (composition) and if one would wish so also for symmetry and reflexivity; they do not feature here.

[11] Note this is neither an interaction net nor a term graph rewrite system.

[12] Taken from an unpublished note from 2003, presented in Leicester a few years later, on generalising Okui's theorem [49] from TRSs to PRSs and to graph rewrite systems via a 2-categorical approach.

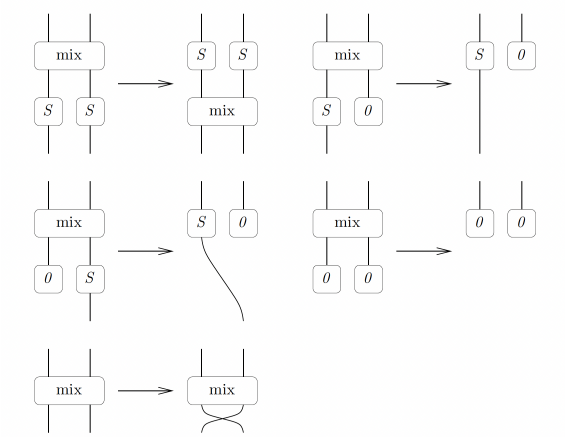

Figure 10: Mix graph rewrite rules in 'open' representation

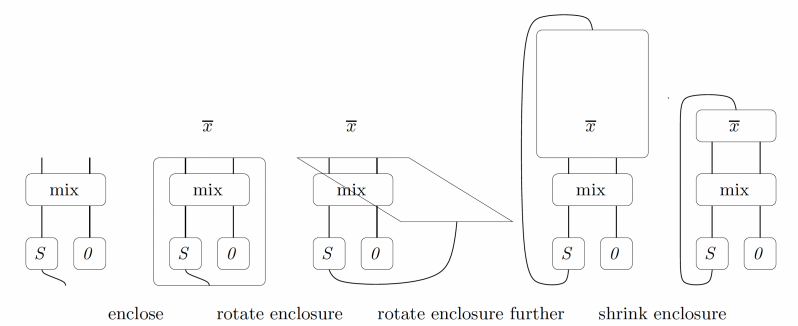

Figure 11: From 'open' to 'closed' representation

in [10], or more narrowly Asperti and Laneve's ISs (interaction systems [6, 7]), or also Hamana's second-order CSs (computations systems [27]).

Among the formats mentioned we prefer PRSs since they have good *abstraction* properties and there is a clear interest in computer science in abstraction, as it avoids duplication, which is a cause of code bloat and software engineering problems [63]. For example, in a 1*st* order term rewrite system (TRS) it is clearly of interest to be able to *abstract* from a pattern encompassed by a term, to replace such a pattern by a variable, cf. [29, Def. 3]. However, this then needs a 2nd order term, so we end up outside the class of 1st-order TRSs, i.e. having to redo things. Similarly for 2nd order systems like CRSs, ISs and CSs.

Of course, the abstraction monster is never satisfied; e.g. PRSs are based on simple types (they have simply typed λ-calculus as substitution calculus, as we would say), but it's clearly of interest to be able to abstract from the same function but for different types, so polymorphism is desirable etc.. Moreover, I share some of the misgivings people may have about *l'abstraction pour l'abstraction*; cf. also the discussion in [4] on when the sharing graph techniques developed for higher-order optimality, could come into play.

Still, to us PRSs seem to be in a sweet spot: they have a simple definition, as term rewriting modulo the well-understood simply typed λ-calculus $\lambda^{\to}$, and are closed under abstraction in the sense discussed above, which is useful in our experience, as for developing syntactic meta-theory for nth order systems it is convenient to work in $(n+1)$st order systems.

3 **stages of implementation: the ordered, positional and anti-positional views on objects?** In [39], Leo discusses three views on *relations* (for the moment, one may think of relations as used in the semantics of predicates in first-order logic as usual) due to Kit Fine, the *ordered* (*standard*) view where the things related come in order, the *positional* view where the things related have positions, and the *anti-positional* view where the constituents come in *complexes* where there is neither order nor position, but which form a network interrelated by substitutions, as he puts it. I think that irrespective of the philosophical debate one can have about whether this is appropriate, it is technically interesting whether one can implement something the same for objects instead of relations, the relevance to the subject of this submission being that *terms* can be construed as objects in the *ordered* view and that (*port*) *graphs* can be seen as objects in the positional view (highlighting that it's not a mere technicality mediating between the ordered and positional views; it is conceptually interesting). Given the superiority of the antipositional view, as shown by Leo for relations, it should be interesting to develop a substitutional view on objects (and e.g. see if / how it relates to single / double pushout approaches to rewriting).

Why supercombinator systems? We agree with the contemplations in [41] that there's quite some freedom in choosing which term rewrite systems to target. There is no real reason why we have taken supercombinator systems other than that they are well-behaved (orthogonal, left-normal, even constructor–destructor) TRSs, and it is profitable to reduce theory for a language with binders like the (weak) λ-calculus to one without binders, even more so if all the problems one is interested in for the language with binders, can be reduced to problems that have been solved already for the language without binders, as it turns out to be the case here (†). That being said, and as expounded in [10], there are many variations that could be and have been explored in the literature.

From the λ-calculus to supercombinators We give another example of that lifting commutes with reduction to normal form (w.r.t. the lefmost–outermost strategy for β- and supercombinator-steps, respectively).

Example 9. The lifting $\mathscr{L}(\delta\,(\delta\,\underline{2}))$ where $\underline{2} := \lambda yz.y\,(y\,z)$ is the Church-numeral 2, results in a pair comprising the term $\kappa_0[]\,(\kappa_0[]\,(\kappa_2[]))$ and a supercombinator systems comprising the three supercombinators κ_0, κ_2 both nullary and binary κ_1, with corresponding term rewrite rules

$$\gamma_0[x] : \kappa_0[]\,x \;\to\; x\,x \qquad \gamma_1[y,z,x] : \kappa_1[y,z]\,x \;\to\; y\,(z\,x) \qquad \gamma_2[x] : \kappa_2[]\,x \;\to\; \kappa_1[x,x]$$

Since self-application as enacted by δ represents exponentation on Church-numerals, the original λ-term $\delta\,(\delta\,\underline{2})$ should β-evaluate to the Church numberal $\underline{256}$, and its supercombinator translation $\kappa_0[]\,(\kappa_0[]\,(\kappa_2[]))$ should supercombinator-evaluate to the same. Indeed they do in our code, as can be checked by evaluating `last (nf n256)` (yielding the last term of the β-reduction to nf) respectively first evaluating `lift n256` (to yield the supercominator system and term) and then `last (snf (fst it) (snd it))` (yielding the last term of the supercombinator reduction to nf) in GHCi. A coarser way to do the same, to test Lem. 1, is to check the lengths of the reductions are the same: `let l256 = lift n256 in length (nf n256) == length (snf (fst l256) (snd l256))` evaluates to `True`.

Example 10. Lifting the Church numeral $\underline{2} := \lambda yzy\,(y\,z)$ yields supercombinators with rules (after renaming into more standard TRS notation and making application into an explicit binary operation @):

$$\begin{aligned} @[i(y,z),x] &\to @[y,@[z,x]] \\ @[o,x] &\to i(x,x) \end{aligned}$$

for i binary and o supercombinators liftings of the inner and outer λ-abstraction respectively. This TRS is clearly terminating (e.g. shown using RPO). On the other hand the single rule TRS $@[d,x] \to @[x,x]$ obtained from lifting Ω is just as clearly non-terminating.

Thus the question arises whether, given a λ-term M, the TRS $\mathscr{T}_M$ obtained via the lifting $\mathscr{L}(M)$ is terminating, and if so in an easy way. More concretely given M is a simply typable (hence terminating) λ-term does termination of $\mathscr{L}(M)$ follow, and (if it does) automatically so, by methods implemented in termination tools?

On commutation of lifting with weak β-steps To see that lifting commutes with wβ-steps in the way described in the main text, consider a λ-term that contains such a redex $(\lambda x.M(x))N$ that is then contracted to $M(N)$, and consider a λ-abstraction $\lambda y.L'$ occurring in $M(N)$. We distinguish cases as to whether or not that occurrence is in a substituted copy of N.

Suppose it is, say at position q within the occurrence of N (itself occurring at some position p in $M(N)$). Then the supercombinator generated by it in the recursion of $\mathscr{L}$ on the contractum, is a renaming of the supercombinator generated by it when arriving at position q in the recursion on the argument N of the redex (note that by the redex being a weak β-redex, it is not inside a λ-abstraction, so indeed the lifting recursion when arriving at the redex acts homomorpically on it, yielding separate computations $\mathscr{L}((\lambda x.M(x)))$ and $\mathscr{L}(N)$).

If it is not, then let $\lambda y.L$ be the subterm at the same position in $M(x)$. Now note that neither the substituted copies of x in $M(x)$ nor the substituted copies of N in $M(N)$ can contain any occurrences of y, per construction respectively per the definition of (higher-order) substitution. That is, in both cases they do not belong to the skeleton, always to the maximal free subexpression of some (the same) subterm of M containing them. Hence, $\lambda y.L'$ and $\lambda y.L$ give rise to the same supercombinator.

Interestingly, this observation as to why commutation of lifting holds[13] yields a good property of left–outer reduction in the λ-calculus as was observed in [24]. There it entails that the height of a λ-term M with respect to the left–outer strategy, grows only linearly along the steps of a reduction $R : M = M_0 \to_{\mathsf{lo}\beta} M_1 \to_{\mathsf{lo}\beta} \ldots$. That is, the height of M_n is bounded by n times the maximal height of the *minimal bound contexts* in M (plus the height of M). This translates into the maximum of the heights of the λ-abstractions in M, with the *height* of $\lambda x.N$ being the maximal *depth* at which there is an occurrence of x in N bound by the λx-abstraction (so the maximal depth of its skeleton, where the *skeleton* comprises all the paths from the λx-abstraction to the occurrences of x bound by it, as expected).

To see this let ℓ / r be the lhs / rhs of the β-rule (as a PRS rule), i.e. ℓ is its redex-pattern, consider a substitution-instance ℓ^σ for substitution σ occurring as the left–outer redex in M at position p, and let N arise from replacing it by its contractum r^σ. We distinguish cases on the position q of a λ-abstraction $\lambda y.L'$ occurring in N, relative to p.

- By the β-rule being left-normal, any λ-abstraction occurring to the left or outside p can never descend to one forming a redex, so such q can be ignored;
- Otherwise q descends from the position o of a λ-abstraction that is either to the right of the redex p or in a term substituted by σ for either the body- or the argument-metavariable in ℓ. But then the whole skeleton of q descends unchanged from the skeleton of o in M (it must by the properties of 2nd-order substitution).

We have set-up this reasoning in a generic way so as to show the crucial rôle of left-normality (in the 1st item; and the β-rule is a prime left-normal example [33]). Our approach here can be seen to exploit it, by supplying a fresh variable to a λ-abstraction or a stuck supercombinator, i.e. if we know these can never descend to one forming a redex, in the given term.

The property, and thereby linear growth of the left–outer strategy, holds for left-normal PRSs that are both (at most) 2*nd order* and such that *any (non-vacuous) abstraction-subterms occurring in the right-hand side r of a rule already occur in its lhs* ℓ.

Remark 2. The analysis of such dynamics of binding goes (for me) back to Melliès' notion of *gripping* [44]. It is a special case of paths in the λ-calculus, which can be seen as allowing to statically describe the dynamics of a λ-term (with Girard's execution formula an extreme case for the λ-calculus). It is versatile tool though, useful beyond the λ-calculus, cf. [64, Ch. 12] for an application in infinitary term rewriting, or [51] for an application to developments, and [20] for an application to the μ-calculus.

The second restriction is necessary (I seem to recall knowing it from the work of Bloo and Rose). E.g. though the property holds for the rule $\mu x.Z[x] \to Z[\mu x.Z[x]]$ and also for $\mu x.Z[x] \to \mu x.Z[Z[a]]$ for a a constant, it obviously fails for $\mu x.Z[x] \to \mu x.Z[Z[x]]$, as that can double the height in every step. We leave it to the reader to check whether the first restriction, to 2nd order PRSs is.

[13] In (†) phrased as *(Minimal) bound contexts only descend to themselves along a reduction, in particular they are not enlarged.*

A lifting view on lazy functional programming, the λ-calculus and orthogonal term rewriting Viewing functional programming as the combination of *parameter passing* (the λ-calculus) with definition by *pattern matching* / *cases* (TRSs), the lifting result of [10] shows that if one does not evaluate under a λ, functions can be faithfully modelled as data, that is, represented by means of constructor terms and passed around as such, doing away with the complexities of passing higher-order functions. Only when a supercombinator is *activated*, cf. [33, p. 278][3, p. 18], i.e. when it becomes applied to some argument, do we turn it from data into a function by replacing it with its rhs. Thus $\mathscr{L}$ allows to reduce the study of lazy functional programming to that of first-order term rewriting, which in a way is a satisfactory state of affairs since there is a lot of term rewriting theory, founded on universal algebra. (Cf. [1] for a semantic take on why the theories of the λ-calculus (head-normal forms) and the lazy λ-calculus (weak-head normal forms) are not to be confounded.)

On the other hand, by compiling them away into data one loses all the higher-order aspects, turns everything into a first-order game. In particular all the higher-order sharing possibilities [40, 37, 5], and to me that's where the interesting challenges are [4], with the first step into the right direction being the study of cyclic sharing as can be expressed by letrec [26]. Still, that type of twisted sharing [16], is far removed from the complex form of sharing that occurs when using Lamping's technique. I hope and expect that the recent development of the appropriate languages (deep inference, string diagrams) will enable a canonical syntactic and semantic account of higher-order implementation techniques (e.g. with some notion of bisimulation collapse for Lamping's graphs) in the not too far future.

(Hyper-)(weak head) normalisation of the left–outer strategy Note that in the statement of Lem. 1 that $\mathscr{L}(M)$ lo-reduces to M' in $\mathscr{T}_M$, we have M' is a λ-term despite that $\mathscr{L}(M)$ is a supercombinator term and $\mathscr{T}_M$ a supercombinator system, so in particular neither contains λ-abstractions; this only seems absurd since lo-steps *introduce* λ-abstractions upon releasing a stuck term but only *a posteriori*: the combinator term is applied to a fresh variable, then supercombinator reduction recursively continues with the result term (still a supercombinator term), and only afterward puts the resulting reduction (that, by recursion, now may already have λ-abstractions) inside the λ-abstraction, resolving the conundrum.

For further normalisation lemmata for notions of result other the set of normal forms, e.g. for head or weak-head normal forms, see [23, 45].

Lemma 2. *If M is $\to_{\mathsf{w}\beta}$-convertible to a whnf, then* loβ *(hyper-)weak-head normalises M.*

Proof. We first recall that $\to_{\mathsf{w}\beta}$ is not confluent [19]. For $M := (\lambda yz.y)(\mathsf{II})$ the reductions $M \to_{\mathsf{w}\beta} \lambda z.\mathsf{II}$ and $M \to_{\mathsf{w}\beta} (\lambda yz.y)\mathsf{I} \to_{\mathsf{w}\beta} \lambda z.\mathsf{II}$ end in distinct whnfs.

To overcome the problem note that in the first reduction the whnf is reached by contracting the weak-head redex, and after the first step in the second reduction, a whnf can be reached by contracting the *residual* of the weak-head-redex contracted in the first. This holds in general: if M is β-convertible to a whnf, then by confluence of β and whnfs being preserved under β, M β-reduces to a whnf. This reduction may be assumed to be standard, hence by left-normality, as being composed of a weak-head reduction followed by a non-weak-head reduction. Since non-weak-head steps cannot create whnfs, the former must end in a whnf. From that we conclude by weak-head reduction being deterministic and noting that on non-whnfs, loβ coincides with weak-head reduction.

Hyper weak-head normalisation follows since weak-head steps can be preponed before non-weak-head steps. □

From supercombinators to maximal sharing graphs Although term graphs can be represented in many ways, we are interested here in *term graph* rewriting (the objects of interest are terms, the λ-terms, and graphs are used to implement them), not in *graph term* rewriting (where the objects of interest are graphs, and terms are used to represent them); cf. [64, (footnote 4 below) Rem. 9.4.30]. Accordingly, we work here with graphs, neither with let-expressions nor with equational term graphs, striving to avoid bureaucracy in the same way that proof nets do.

Rewriting modulo a substitution calculus In what we call *structured* rewrite systems,[14], steps are enacted in three stages, decomposing a structure into a context and a lhs, replacing the lhs by the rhs, and composing the context with the rhs again. As soon as (de)composition becomes a complex process itself it may be worthwhile to model it itself by a (simpler) rewrite system, that's what we dubbed a *substitution* calculus.

This was certainly the case in higher-order term rewriting [57, 50, 61], where higher-order substitution is a highly complex operation. We suggest that, given the continual stream of papers developing highly sophisticated categorical ways of expressing (de)composition, also in graph rewriting (de)composition is amply subtle and complex enough to warrant analysing it by means of a substitution calculus. A substitution calculus should be chosen wisely, so that it allows a 'good' division of labour between rewrite steps and substitution calculus steps[15] We put forward a simple such substitution calculus, the ж-calculus for term graph rewriting as considered here.

Remark 3. One might think that working *modulo* a substitution calculus would commit one, when proving termination of the rewrite system, to models where the lhs and rhs of *all* steps of the substitution calculus are interpreted as the same element of a termination model, after all we are working *modulo* the substitution calculus.

This is not the case, as was expl(ain/oit)ed in the case of higher-order term rewriting in [32]. The idea is that when performing rewrite steps, during the matching phase typically only *highly constrained expansion* steps of the substitution calculus are allowed, and only those may need to be interpreted as equalities, as they will typically be undone during the substitution phase. For example, for the case of the ж-calculus though it is not unsound to introduce arbitrarily many indirection nodes by $\underline{\to}$-expansion steps or arbitrary garbage nodes (not reachable from the root of the term graph), for generating a rewrite step that is not allowed (though it would not be unsound as they would be removed again during ж-normalisation in the substitution phase; it would not be very conducive to efficiency; this is analogous to how β-expansion could be so 'abused' in higher-order term rewriting modulo $\lambda^{\to}$). Accordingly, one *may* interpret the garbage collection-rule of the ж-calculus as being oriented from left to right in termination proof, and similarly for steps introducing multiple consecutive indirection nodes. Fnally, also that we limited copying of nodes to ones on a path to the root, and then only once, could be exploited.

For another take on the same: *normalised* rewriting as proposed in [42] can be viewed as an instance of the above as it essentially restricts one to rewriting only *representatives* (normal forms) of the equivalence classes of the substitution calculus, and the point of that work was to get *better* (termination) properties, not worse.

It may seem from Ex. 6 that of the three stages (matching, replacement, substitution) in term rewriting, only *substitution* may exhibit behaviour linear in the size of its arguments, due to replicating (erasing, duplicating, triplicating,...) the arguments. But also *matching* might. For instance, to see whether or not the lhs of the rule $\varpi : \mathsf{eq}[x,x] \to \mathsf{True}$ matches with the term $\mathsf{eq}[t,s]$ amounts to checking whether or not t and s are equal; only if $t = s$ can $\mathsf{eq}[t,s]$ be β-expanded into $(x.\mathsf{eq}[x,x])\,t$.

A multistep $t \mathrel{-\!\!\circ\!\!\to_{\mathscr{T}}} s$ of a TRS $\mathscr{T}$ may be used to contract an arbitrary number of redex-pattern occurrences in t, but only if these are pairwise non-overlapping and respect the non-linearity constraints. Explained by example: the root f-redex-pattern and the left–inner a in the term $f[a,a]$ cannot be contracted in the single multistep for a TRS having rules $\rho : f[x,x] \to \ldots$ and $\varpi : a \to \ldots$; we can only contract *neither* of the as by the (single) step $\rho[a]$ *or* both by the multistep $\rho[\varpi]$.

Since this is the same in translation to the TGRS $\mathscr{G}$, the implementation result is not affected by it. Anyway, since supercombinator systems are *orthogonal* this issue will not surface in our application of implementing the λ-calculus: multisteps can contract arbitrary collections of redex-pattern-occurrences in a term.

The ж-calculus The three rule schemes of the ж-calculus (ж is the letter zh of the cyrillic script) in Fig. 2 are to be interpreted as that each edge to the boundary must be connected to the port of some node (to an output port of a node for edges to the upper boundary and the input port of a node for edges to the lower boundary). Replacement of a rule-symbol by its lhs (rhs) detaches the rule-symbol from its indirection nodes reattaching its lhs (rhs).

[14] We have not axiomatised this notion, but the reader may substitute their favourite structure, like string, term or graph, for it, continue reading, and see whether that makes sense.

[15] As a rule of thumb, if one finds oneself proving results about substitutions when dealing with rewrite steps something is off. The insistence on working with *closed* expressions for lhss / rhss of rules, is derived from that: it tries to minimise the interaction. Similarly for the proposal to work with boxes / variables in graph rewriting; that aims at separating steps from the substitutions as modularly as possible,

Msgs being maximally shared, an msg rewrite step $G \to H$ never causes wholesale replication of the context of the step (of the lhs and rhs of the rule), neither in the matching nor in the substitution phase (that's their point), as implemented in tools [17] and described in the literature.

The matching-phase of an msg rewrite step may proceed as described in [8, 24] by using $\xleftarrow{}$-steps (inverse $\xrightarrow{}$-steps): first the symbols of the redex-pattern and the path from the root to it in G are unshared in such a way that there's exactly one path from the root of the msg to the root of the lhs. (An analogon for dags of Huet's *zipper* for trees [30] suggests itself; we expect it exists, but if not it maybe should.) After that, its indirection nodes are inserted on the edges along the boundary of the lhs (towards its root, and from its non-nullary leaves), upon which it is ready for replacement. The matching phase only requires unsharing (*unzipping*) of the nodes in the lhs, the redex-pattern, and nodes along its path to the root. Not of its arguments though, not even for non-left-linear rules since subterm equality coincides with pointer equality in msgs, cf. the discussion and results on collapsing in [60]; if the same (sub)term were represented by distinct nodes, then sharing would not be maximal and the nodes could be collapsed (possibly after collapsing nodes closer to the leaves first) so should have been.

In general, to go from a term graph not in ж-normal form to one that is, an msg, one can proceed by garbage-collecting nodes not reachable from the root, remove indirection nodes, and bottom-up share nodes maximally (say via some bisimulation collapse algorithm [26]). But the substitution-phase of a rewrite step requires to check this only for the symbols introduced in its rhs and on the path to the root (*zipping* up again).

From supercombinators to sharing graphs

Additional explanations for Ex. 7 Coming back to the question what $\twoheadrightarrow_{\mathscr{T}}$-steps on terms are exactly implemented by $\to_{\mathscr{G}}$-steps on sgs? The answer, due to Maranget [41] in a bit more detail is: the applicator–constructor pairs whose edge have **the same label**, where *labels* $\varsigma, \zeta, \xi, \ldots$ are defined by the BNF $\varsigma := \alpha \mid (\varsigma, p) \mid \varsigma\varsigma$ for *atomic* labels in $\alpha \in A$ and positions p.[16] Labelling the subterms of a term t / the edges of the corresponding sg G, and labelling their steps per Fig. 3, for label ς, an ς-*family* step is a parallel step contracting *all* redex-patterns whose supercombinator / edge have label ς.

The crucial property as established by Maranget is that labels of families occur uniquely along reductions. In particular, two occurrences of labels in the same term can not be sublabels of each other since that would entail one to be causally dependent on the other.

Since Maranget showed this on terms, and it might be instructive and a bit simpler [6, 7] to show it for sgs (since in the sgs the *edges* are unique too), we present the crucial ingredients adapted from [41] from terms to sgs.

Say an sg G is *consistent* $G{\uparrow}$ if its multiset of label is, i.e. if no element is a *sub*label $\leq$ of another, where $\leq$ is the reflexive–transitive closure of $<^1$ relating ς to $\zeta(\varsigma, p)\xi$ [41, Def. 4.8]. We adapt preservation of consistency under family steps in TRSs [41, Lem. 4.10] to sgs.

Lemma 3. *if $G \multimap_{\varsigma} H$ and G is consistent, then so is H.*

Proof. Note $\leq$ is a partial order, for labels are $<^1$-related only to bigger ones. Let us denote consistency of G by $\uparrow$. By consistency of G, in fact $G \to_{\varsigma} H$ say by rule $\ell \to r$. Let ζ, ξ be labels of distinct edges in H. To show consistency of the multiset $\mu := [\zeta, \xi]$ distinguish cases for ζ, ξ on whether it is in the context C, or not. Note that labels in C are in G, and that labels not in C have shape $\zeta'(\varsigma, p)\xi'$, with non-empty ζ', ξ' in G.

If both ζ, ξ are in C then μ is consistent by $\uparrow$.

If both are in not in C, then w.l.o.g. $\zeta = \zeta'(\varsigma, p)\xi' \leq \zeta''(\varsigma, p')\xi'' = \xi$. If they are equal, then since labels uniquely read as sequences of non-composite labels $p = p'$ (impossible per construction) or (ς, p) is a label ζ'' or ξ'' is composed of, but then $\varsigma \leq \zeta''$ or $\varsigma \leq \xi''$ contradicting $\uparrow$. Otherwise $\zeta = \zeta'(\varsigma, p)\xi' \leq \varsigma' <^1 \zeta''(\varsigma, p')\xi'' = \xi$ for some ς'. by unique reading again ς' must then be $<^1$-related to (ς, p') (impossible by size) or to either of ζ'', ξ'', which is inconsistent with $\uparrow$.

If one, say ζ, is not in C and the other, ξ, is, then ζ has shape $\zeta'(\varsigma, p)\xi'$ and $\zeta'(\varsigma, p)\xi' \leq \xi$ is impossible as it would yield $\varsigma \leq \xi$ contradicting $\uparrow$. But also $\xi \leq \varsigma' <^1 \zeta'(\varsigma, p)\xi'$ for some ς', is impossible, since by unique reading ς' must then be $<^1$-related to (ς, p) or ζ' or ξ' each of which (e.g. via $\xi \leq \varsigma' = \varsigma$) contradicts $\uparrow$. □

[16] Used to uniquely address the subterms of rhss of rules, though in Ex. 7 we just enumerated the subterms for brevity.

Families In fact, not just lo but any strategy contracting a family containing *a* needed redex is normalising and minimal for sgs, as follows from ordered local commutation of such needed family-steps w.r.t. other steps. Of course, this is well-known but seeing it as an instance of the general abstract theory of normalisation and minimality of strategies of [53], should make clear that optimal reduction really is, in the formal sense optimal, but w.r.t. a rewrite system not many people are familiar with, family reduction.

By adhering to the lo strategy one automatically avoids reducing inside garbage, so one may postpone the garbage-collection steps of the explicit substitution calculus ж, of so desired.

Classifying sharing Sharing graphs as employed in implementing reduction for the history aware multistep strategy contracting family multisteps of the $\lambda\beta$-calculus [40, 5] escape Blom's classification [16], into horizontal (think of acyclic term graphs), vertical (think of μ), and twisted sharing (think of letrec). This is due to the fact they allow higher-order sharing; this causes that though such a sharing graph may be *cyclic* it still represents a *finite* λ-term. Syntactically this is embodied by representing a *box*, a replicable resource in the sense of Girard's linear logic [22, 36], via their *ports* only, we dubbed *scope* nodes in [56] unifying the *brackets* and *croissants* in earlier approaches [5]. To keep control of such a *distributed representation* of a box, keeping its ports only (think of a park where we remove the fence around it but do keep the gates on its access paths, or a room with no walls only doors) is highly non-trivial; the dynamic local interaction between scope nodes breaks two tenets on boxes:

- two boxes are either disjoint or one is nested in the other, they can't partially overlap;
- a box separates its inside from its outside.

Partially sharing boxes, necessarily breaking the first, was the revolutionary idea [37] allowing for the first implementation of the λ-calculus that was optimal in the sense of [40]. Due to the cyclic nature of sharing graphs, such partial sharing may be cyclic as well, yielding that boxes may partially overlap *themselves*, breaking the second tenet too; a box may be wrapped up inside itself like a Möbius strip or a Klein bottle. Formally, given a box in the λ-term tree t readback from a sharing graph G, nodes both from inside and outside the box in the λ-term t may have been readback from the *same* node in the graph G; thus one can't sensibly speak of the latter being on the *inside* or the *outside* of the box.

Stated differently, whilst reading back a λ-term tree t from a sharing graph G one may come across the *same* node in G several times, with each inducing a node in t but belonging to a *different / disjoint* box [7, 7, 5].

C Review at the basis of this submission

Below I reproduce my review from 2005 of a submission by Blanc, Lévy and Maranget for [47]. I only reproduce the conceptual part of that review here, since the concrete remarks (typos and the like) have been incorporated by the authors in the accepted and published version [15], so would not be intelligible at all without the original submission, and it is not for me to decide to publish that or not.[17] I reproduce that part here since this submission is largely based on it, but it was ignored in the final published version [15] of the paper.[18]

At the time that left my reviewing efforts a bit stranded. But in the meantime ideas in the review have found their way, in particular to Jan Rochel and Clemens Grabmayer as investigators on the NWO project Realising Optimal Sharing[19] and to Thibaut Balabonski, at the time a PhD student of Delia Kesner who visited us at Utrecht University in 2010 (if memory serves me right) as he was working on the same theme. Our fruitful discussions were reflected in [10, 24]. Still, I thought this was a nice occasion to revisit this old material.

[17] Also the citations in the part included here, are not really intelligible as they refer to the submitted version, but I think the gist of this part is clear nonetheless. In fact, reading it back after all these years, I found it surprisingly readable.

[18] It is the prerogative of authors to decide what and what not to include in their paper; the paper [15] is i.m.o. very interesting and can very well stand on its own.

[19] `https://www.nwo.nl/en/projects/612000935` at Utrecht University, 2009–2015 (joint with Doaitse Swierstra).

Summary

In his Phd Thesis, J.-J. Lévy set forth his seminal theory of optimal sharing for the λ-calculus. In this paper a partial account of the same is presented, but now for a weak λ-calculus, i.e. a λ-calculus where reduction steps are not necessarily closed under λ-abstraction.

Inspired by [7], two of the authors introduced in [19] a weak λ-calculus which allows reduction steps to take place under a λ-abstraction iff the contracted redex is free for the variable bound by the abstraction. The technical points in favour of this particular weak λ-calculus was that not only does it allow to implement the strategies found in standard implementations of functional programming languages, but that it also yields a notion of reduction enjoying both standardization and confluence, which other weak λ-calculi were lacking.

Like for the ordinary λ-calculus one may then ask: what is is the theory of optimal reduction for this calculus, in other words, what redexes have the same reduction history so could be shared in an implementation? Here it is important to note that the reduction history of a redex in the weak λ-calculus may be different from that in the ordinary λ-calculus. To wit, the step $(\lambda x.Ix)K \to IK$ *creates* the redex IK in the weak λ-calculus due to the fact that Ix is not yet a redex in the source (x is bound from the outside), whereas IK is a *residual* of the redex Ix in the source in the λ-calculus. Technically, the weak λ-calculus has two ways of creation rather than a single one:

1. by creation of a redex *pattern*, e.g. $(xN)[x{:=}\lambda x.M]$, substituting a λ-abstraction for an active variable, as in the ordinary λ-calculus.
2. by making a redex *free*, e.g. $(Ix)[x := K]$, substituting for the variables bound outside it.

To reflect the new form of creation, the authors extend the usual Lévy labeled λ-calculus (which only took into account the first form): the labels are extended with extra constructs reflecting the second form of creation, and β-reduction now includes a *diffusion* or *broadcast* operation in order to put these extra constructs at the appropriate places, i.e. where redexes are freed.

It is shown that the resulting labeled weak λ-calculus is non-deterministic but confluent. This is a good start since it allows for different reduction (implementation) strategies, but the real aim is to show that if two redex-patterns receive the same label then they are identical, i.e. shared, in the (Wadsworth) graph implementation. The main result of the paper is a proof of the first half of this claim. It is shown that if two redex-patterns receive the same label along a maximal reduction, then the corresponding redexes are identical. Implementation-wise this indicates that for a graph implementation of this labeled weak λ-calculus, dags (directed acyclic graphs) are (could be) sufficient, which is to be contrasted to the cyclic graphs needed for an optimal graph implementation of the λ-calculus in the style of Lamping.

Interpretation in TRSs

The restriction that for a subterm to be a redex in the weak λ-calculus it mustn't contain variables bound outside it, entails that residuals behave like residuals in first-order TRSs; as you remark yourself on page 7, in the weak λ-calculus residuals of disjoint redexes remain disjoint. Since the whole theory of optimality is based on residuals, this seems to make the situation you are dealing with in the paper analogous to that in first-order TRSs, as can be found e.g. in the book *Term Rewriting Systems*, CUP, 2003, by the writers collective Terese (which includes Klop). I'll try to make this a bit more precise now, but beware, I have not checked the details! If it does not work out in the end, I apologize and only hope to see some remark on where things break down.

Any λ-abstraction $\lambda x.M$ can be uniquely written as $C[\vec{F}]$ with each element of the vector F a maximal free subterm not containing the bound variable x, so one could call C a *minimal bound* context (cf. λ-lifting). Then we define the *pattern* of a weak β-redex $(\lambda x.M)N$ to be $(\lambda x.C[\vec{Z}])X$ for meta-variables $\vec{Z}$ and X. For instance, the pattern of the weak β-redex $(\lambda x.Ix(\lambda y.Ixy))S$ is $(\lambda x.Z_1x(\lambda y.Z_2xZ_3))X$ giving rise to the 'first-order' rule $(\lambda x.Z_1x(\lambda z.Z_2xZ_3))X \to Z_1X(\lambda y.Z_2XZ_3)$,

where the right-hand side is obtained as $C[\vec{Z}][x{:=}X]$. Substituting I for both Z_1 and Z_2, y for Z_3 and S for x yields the step $(\lambda x.Ix(\lambda y.Ixy))S \to IS(\lambda y.ISy)$, mimicking the β-step. [3]

This simulation not just makes explicit the intuition that β-reduction in the weak λ-calculus is first-order-like in that it does not involve manipulating (outside) bound variables. The real point is that the TRS W consisting of *all* such rules is orthogonal ((minimal) bound contexts of redexes cannot have overlap) and that the standard TRS notions of (both static and dynamic) residual for the TRS W captures *exactly* the notion of residual in the weak λ-calculus (note that a redex having overlap with the (minimal) bound context in the rhs is *not* a static residual but a created, i.e. dynamic, one). This will be elaborated in the following.

Confluence of the weak λ-calculus follows from the diamond property for parallel steps (simultaneously contracting redexes at parallel/disjoint positions) as usual in first-order TRSs. Thus, to me it seems that employing any of: the Tait & Martin-Löf method based on simultaneously contracting redexes at parallel/disjoint *or* nested positions, or Melliès axiomatic method based on the notion of *gripping*, or an embedding into Klop's Combinatory Reduction Systems, as you suggest to do now to prove confluence complicates rather than clarifies matters; each of them is designed to handle complex (nesting) situations which are absent here.

The importance of the finite developments theorem for your aims seems diminished. Developing n parallel redexes yields a reduction of length n, regardless of the order. Even if you are interested in finite developments of nested residuals, that follows trivially by recursive path orders.

Standardization is an instance of standardization for first-order TRSs. Again, parallel reductions suffice.

(Minimal) bound contexts only descend to themselves along a reduction, in particular they are not enlarged. This allows one to treat them as atomic until they become the body of a contracted redex (possibly enabling bridging the gap to supercombinator implementations).

Of the above, I'm quite sure. Of the rest I'm less sure, but I think that, apart from the static residual relation also the dynamic residual relation can be obtained via the TRS W, via its Lévy labeling. Switching to a syntax like yours (on page 9), this boils down to changing only your definition of labels into:

$$\alpha, \beta ::= a \mid [\alpha', i] \mid [\alpha', \beta]$$

and defining labeled W-reduction by

$$(\alpha' \cdot \lambda x.C[\vec{Z}])X \to C^{\alpha'}[x := X]$$

where $C^{\alpha'}$ is C where each label β is replaced by $[\alpha', \beta]$[4], and each meta-variable Z_i by $[\alpha', i]{:}Z_i$[5]. Then one shows in order:

1. Labels and degrees are created uniquely (once) along any maximal reduction (Lemma 7).

   Such a result seems to be abstract, i.e. to hold for any labeling of any rewriting system obeying that a label of one redex is a sublabel of another iff it contributed to its creation (hence the former precedes the latter in the reduction sequence), and contracting all redexes having some label does not recreate the label.

[3] An alternative, not corresponding to diffusion but rather to the Shivers & Wand calculus (but still giving rise to the same amount of sharing), would be to completely close the pattern (note that in the above the variable y which is bound in the pattern is not part of the pattern, but substituted for Z_3) under taking minimal bound contexts of λs in the context, yielding patterns of *bound* contexts. For instance, the bound context of the weak β-redex $(\lambda x.Ix(\lambda y.Ixy))S$ is $(Z_1x(\lambda y.Z_2xz))X$ obtained by closing under the minimal bound context of the λy in the pattern, gives rise to the 'first-order' rule $(\lambda x.Z_1x(\lambda y.Z_2xy))X \to Z_1x(\lambda y.Z_2xy)$. Substituting I for both y_1 and y_2 and S for x yields the step $(\lambda x.Ix(\lambda y.Ixy))S \to IS(\lambda y.ISy)$, again mimicking the β-step.

[4] This is a bit different from (simpler than) the Lévy labeling of TRSs where all subterms of the right-hand side receive different labels. However, note that initially all labels in a (minimal) bound context are pairwise distinct, and since they descend statically until contracted, as noted in the previous paragraph, this distinction is preserved.

[5] To take creation by freeing into account. Again this is a bit different from the Lévy labeling of TRSs, but note that this would not be needed if not only 'edges' but also 'vertices' (as is in fact the case in the Lévy labeling of TRSs) would have been labeled; if a redex would be created by freeing its application symbol must be part of the (minimal) bound context, so would automatically be labeled by α'.

2. If the labels directly left-below two occurrences of an application are identical, then the labels directly above them are identical as well (Lemma 9).

   This follows by easy inspection, using the next invariant, as in your proof of Lemma 9.

3. Subterms having the same label are identical (Lemma 10).

   As for Lemma 10.

If correct, this[6] should result in shorter proofs, simply because there are fewer cases to consider for the labeling. It would also avoid the technical invariant (Invariant 2), avoid working up to an equivalence on labels (page 13), and answer the question whether underlining and overlining are necessary (page 16) in the negative.