Maximal Sharing in the Lambda Calculus with letrec [1]

Clemens Grabmayer
Dept. of Computer Science, VU University Amsterdam
de Boelelaan 1081a, 1081 HV Amsterdam
c.a.grabmayer@vu.nl

Jan Rochel
Dept. of Computing Sciences, Utrecht University
Princetonplein 5, 3584 CC Utrecht, The Netherlands
jan@rochel.info

Abstract

Increasing sharing in programs is desirable to compactify the code, and to avoid duplication of reduction work at run-time, thereby speeding up execution. We show how a maximal degree of sharing can be obtained for programs expressed as terms in the lambda calculus with letrec. We introduce a notion of 'maximal compactness' for $\lambda_{\mathsf{letrec}}$-terms among all terms with the same infinite unfolding. Instead of defined purely syntactically, this notion is based on a graph semantics. $\lambda_{\mathsf{letrec}}$-terms are interpreted as first-order term graphs so that unfolding equivalence between terms is preserved and reflected through bisimilarity of the term graph interpretations. Compactness of the term graphs can then be compared via functional bisimulation.

We describe practical and efficient methods for the following two problems: transforming a $\lambda_{\mathsf{letrec}}$-term into a maximally compact form; and deciding whether two $\lambda_{\mathsf{letrec}}$-terms are unfolding-equivalent. The transformation of a $\lambda_{\mathsf{letrec}}$-term L into maximally compact form L_0 proceeds in three steps: (i) translate L into its term graph $G = [\![L]\!]$; (ii) compute the maximally shared form of G as its bisimulation collapse G_0; (iii) read back a $\lambda_{\mathsf{letrec}}$-term L_0 from the term graph G_0 with the property $[\![L_0]\!] = G_0$. Then L_0 represents a maximally shared term graph, and it has the same unfolding as L.

The procedure for deciding whether two given $\lambda_{\mathsf{letrec}}$-terms L_1 and L_2 are unfolding-equivalent computes their term graph interpretations $[\![L_1]\!]$ and $[\![L_2]\!]$, and checks whether these are bisimilar.

For illustration, we also provide a readily usable implementation.

Categories and Subject Descriptors D.3.3 [*Language constructs and features*]: Recursion; F.3.3 [*Studies of Programming Constructs*]: Functional constructs

General Terms functional programming, compiler optimisation

Keywords Lambda Calculus with letrec; unfolding semantics; subterm sharing; maximal sharing; higher-order term graphs

1. Introduction

Explicit sharing in pure functional programming languages is typically expressed by means of the letrec-construct, which facilitates cyclic definitions. The λ-calculus with letrec, $\lambda_{\mathsf{letrec}}$ forms a syntactic core of these languages, and it can be viewed as their abstraction. As such $\lambda_{\mathsf{letrec}}$ is well-suited as a test bed for developing program transformations in functional programming languages. This certainly holds for the transformation presented here that has a strong conceptual motivation, is justified by a form of semantic reasoning, and is best described first for an expressive, yet minimal language.

1.1 Expressing sharing and infinite λ-terms

For the programmer the letrec-construct offers the possibility to write a program compactly by utilising subterm sharing. letrec-expressions bind subterms to variables; these variables then denote occurrences of the respective subterms and can be used anywhere inside of the letrec-expression. In this way, instead of repeating a subterm multiple times, a single definition can be given that is referenced from multiple positions.

We will denote the construct letrec here by let as in Haskell.

Example 1.1. Consider the λ-term $(\lambda x.\, x)\,(\lambda x.\, x)$ with two occurrences of the subterm $\lambda x.\, x$. These occurrences can be shared with as result the $\lambda_{\mathsf{letrec}}$-term $(\mathsf{let}\ id = \lambda x.\, x\ \mathsf{in}\ id\, id)$.

As let-bindings permit definitions with cyclic dependencies, terms in $\lambda_{\mathsf{letrec}}$ are able to finitely denote infinite λ-terms (for short: λ^∞-terms). The λ^∞-term M represented by a $\lambda_{\mathsf{letrec}}$-term L can be obtained by a typically infinite process in which the let-bindings in L are unfolded continually with M as result in the limit. Then we say that M is the *infinite unfolding* of L, or that M is the denotation of L in the *unfolding semantics*, indicated symbolically by $M = [\![L]\!]_{\lambda^\infty}$.

Example 1.2. For the $\lambda_{\mathsf{letrec}}$-terms L and P and the λ^∞-term M:

$$L := \lambda f.\, \mathsf{let}\ r = f\, r\ \mathsf{in}\ r$$
$$P := \lambda f.\, \mathsf{let}\ r = f\,(f\, r)\ \mathsf{in}\ r \qquad M := \lambda f.\, f\,(f\,(\ldots))$$

it holds that both L and P (which represent fixed-point combinators) have M as their infinite unfolding: $[\![L]\!]_{\lambda^\infty} = [\![P]\!]_{\lambda^\infty} = M$.

L and P in this example are 'unfolding-equivalent'. Note that L represents M in a more compact way than P. It is intuitively clear that there is no $\lambda_{\mathsf{letrec}}$-term that represents M more compactly than L. So L can be called a 'maximally shared form' of P (and of M).

We address, and efficiently solve, the problems of computing the maximally shared form of a $\lambda_{\mathsf{letrec}}$-term, and of determining whether two $\lambda_{\mathsf{letrec}}$-terms are unfolding-equivalent. Note that these notions are based on the static unfolding semantics. We *do not consider* any dynamic semantics based on evaluation by β-reduction or otherwise.

1.2 Recognising potential for sharing

A general risk for compilers of functional programs is "[to construct] multiple instances of the same expression, rather than sharing a single copy of them. This wastes space because each instance occupies separate storage, and it wastes time because the instances will be reduced separately. This waste can be arbitrarily large, [...]"

[1] This work was supported by NWO in the framework of the project *Realising Optimal Sharing (ROS)*, project number 612.000.935.

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-2873-9 /14/09...$15.00.
http://dx.doi.org/10.1145/2628136.2628148

([28, p.243]). Therefore practical compilers increase sharing, and do so typically for supercombinator translations of programs (such as fully-lazy lambda-lifting). Thereby two goals are addressed: to increase sharing based on a syntactical analysis of the 'static' form of the program; and to prevent splits into too many supercombinators when an anticipation of the program's 'dynamic' behaviour is able to conclude that no sharing at run-time will be gained.

A well-known method for the 'static' part is common subexpression elimination (CSE) [6]. For the 'dynamic' part, a predictive syntactic program analysis has been proposed for fine-tuning sharing of partial applications in supercombinator translations [10].

We focus primarily on the 'static' aspect of introducing sharing. We provide a conceptual solution that substantially extends CSE. But instead of maximising sharing for a supercombinator translation of a program, we carry out the optimisation on the program itself (the $\lambda_{\mathsf{letrec}}$-term). And instead of applying a purely syntactical program analysis, we use a term graph semantics for $\lambda_{\mathsf{letrec}}$-terms.

1.3 Approach based on a term graph semantics

We develop a combination of techniques for realising maximal sharing in $\lambda_{\mathsf{letrec}}$-terms. For this we proceed in four steps: $\lambda_{\mathsf{letrec}}$-terms are interpreted as higher-order term graphs; the higher-order term graphs are implemented as first-order term graphs; maximally compact versions of such term graphs can be computed by standard algorithms; $\lambda_{\mathsf{letrec}}$-terms that represent compacted term graphs (or in fact arbitrary ones) can be retrieved by a 'readback' operation.

In more detail, the four essential ingredients are the following:

(1) A *semantics* $[\![\cdot]\!]_{\mathcal{H}}$ for interpreting $\lambda_{\mathsf{letrec}}$-terms as *higher-order term graphs*, which are first-order term graphs enriched with a feature for describing binding and scopes. We call this specific kind of higher-order term graphs 'λ-ho-term-graphs'.

The variable binding structure is recorded in this term graph concept because it must be respected by any addition of sharing. The term graph interpretation adequately represents sharing as expressed by a $\lambda_{\mathsf{letrec}}$-term. It is not injective: a λ-ho-term-graph typically is the interpretation of various $\lambda_{\mathsf{letrec}}$-terms. Different degrees of sharing as expressed by $\lambda_{\mathsf{letrec}}$-terms can be compared via the λ-ho-term-graph interpretations by a sharing preorder, which is defined as the existence of a homomorphism (functional bisimulation).

While comparing higher-order term graphs via this preorder is computable in principle, standard algorithms do not apply. Therefore efficient solvability of the compactification problem and the comparison problem is, from the outset, not guaranteed. For this reason we devise a first-order implementation of λ-ho-term-graphs:

(2) An *interpretation* $\mathcal{HT}$ of λ-ho-term-graphs into a specific kind of *first-order term graphs*, which we call 'λ-term-graphs'. It preserves and reflects the sharing preorder.

$\mathcal{HT}$ reduces bisimilarity between λ-ho-term-graphs (higher-order) to bisimilarity between λ-term-graphs (first-order), and facilitates:

(3) The use of standard methods for *checking* bisimilarity and for computing the bisimulation *collapse* of λ-term-graphs. Via $\mathcal{HT}$ also the analogous problems for λ-ho-term-graphs can be solved.

Term graphs can be represented as deterministic process graphs (labelled transition systems), and even as deterministic finite-state automata (DFAs). That is why it is possible to apply efficient algorithms for state minimisation and language equivalence of DFAs.

Finally, an operation to return from term graphs to $\lambda_{\mathsf{letrec}}$-terms:

(4) A *readback* function rb from λ-term-graphs to $\lambda_{\mathsf{letrec}}$-terms that, for every λ-term-graph G, computes a $\lambda_{\mathsf{letrec}}$-term L from the set of $\lambda_{\mathsf{letrec}}$-terms that have G as their interpretation via $[\![\cdot]\!]_{\mathcal{H}}$ and $\mathcal{HT}$ (i.e. a $\lambda_{\mathsf{letrec}}$-term for which it holds that $\mathcal{HT}([\![L]\!]_{\mathcal{H}}) = G$).

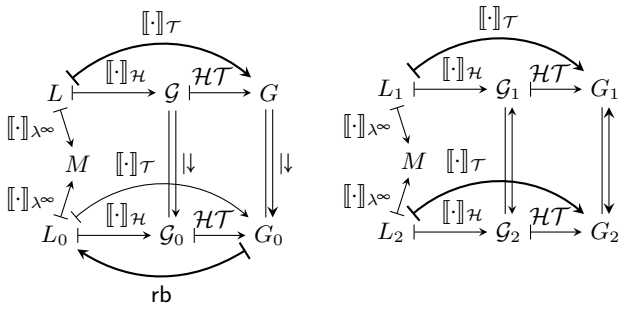


Figure 1. Component-step build-up of the methods for computing a maximally shared form L_0 of a $\lambda_{\mathsf{letrec}}$-term L (left), deciding unfolding equivalence of $\lambda_{\mathsf{letrec}}$-terms L_1 and L_2 via bisimilarity $\leftrightarrows$ (right).

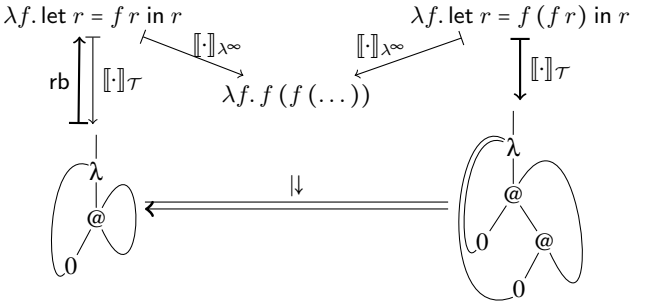


Figure 2. Computing a maximally compact version of the term P from Ex. 1.2 (right) by using composition of term graph semantics $[\![\cdot]\!]_{\mathcal{T}}$, collapse $|\!\downarrow$, and readback rb, yielding the term L (left).

1.4 Methods and their correctness

On the basis of the concepts above we develop efficient methods for introducing maximal sharing, and for checking unfolding equivalence, of $\lambda_{\mathsf{letrec}}$-terms, as sketched below.

In describing these methods, we use the following notation:

$\mathcal{H}$: class of λ-ho-term-graphs, the image of the semantics $[\![\cdot]\!]_{\mathcal{H}}$;
$\mathcal{T}$: class of λ-term-graphs, the image of the interpretation $\mathcal{HT}$;
$[\![\cdot]\!]_{\mathcal{T}} := \mathcal{HT} \circ [\![\cdot]\!]_{\mathcal{H}}$: first-order term graph semantics for $\lambda_{\mathsf{letrec}}$-terms;
$|\!\downarrow$: bisimulation collapse on $\mathcal{H}$ and $\mathcal{T}$;
rb : readback mapping from λ-term-graphs to $\lambda_{\mathsf{letrec}}$-terms.

We obtain the following methods (for illustrations, see Fig. 1):

- *Maximal sharing*: for a given $\lambda_{\mathsf{letrec}}$-term, a maximally shared form can be obtained by collapsing its first-order term graph interpretation, and then reading back the collapse: $\mathsf{rb} \circ |\!\downarrow \circ [\![\cdot]\!]_{\mathcal{T}}$
- *Unfolding equivalence*: for given $\lambda_{\mathsf{letrec}}$-terms L and P, it can be decided whether $[\![L]\!]_{\lambda^\infty} = [\![P]\!]_{\lambda^\infty}$ by checking whether their term graph interpretations $[\![L]\!]_{\mathcal{T}}$ and $[\![P]\!]_{\mathcal{T}}$ are bisimilar.

See Fig. 2 for an illustration of the application of the maximal sharing method to the $\lambda_{\mathsf{letrec}}$-terms L and P from Ex. 1.2.

The correctness of these methods hinges on the fact that the term graph translation and the readback satisfy the following properties:

(P1) $\lambda_{\mathsf{letrec}}$-terms L and P have the same infinite unfolding if and only if the term graphs $[\![L]\!]_{\mathcal{T}}$ and $[\![P]\!]_{\mathcal{T}}$ are bisimilar.

(P2) The class $\mathcal{T}$ of λ-term-graphs is closed under homomorphism.

(P3) The readback rb is a right inverse of $[\![\cdot]\!]_{\mathcal{T}}$ up to isomorphism $\simeq$, that is, for all term graphs $G \in \mathcal{T}$ it holds: $([\![\cdot]\!]_{\mathcal{T}} \circ \mathsf{rb})(G) \simeq G$.

Note: (P2) and (P3) will be established only for a subclass $\mathcal{T}_{\text{eag}}$ of $\mathcal{T}$. Furthermore, practicality of these methods depends on the property:

(P4) Translation $[\![\cdot]\!]_{\mathcal{T}}$ and readback rb are efficiently computable.

1.5 Overview of the development

In the Preliminaries (Section 2) we fix basic notions and notations for first-order term graphs. $\lambda_{\mathsf{letrec}}$-terms and their unfolding semantics are defined in Section 3. In Section 4 we develop the concept of 'λ-ho-term-graph', which gives rise to the class $\mathcal{H}$, and the higher-order term graph semantics $[\![\cdot]\!]_{\mathcal{H}}$ for $\lambda_{\mathsf{letrec}}$-terms.

In Section 5 we develop the concept of first-order 'λ-term-graph' in the class $\mathcal{T}$, and define the interpretation $\mathcal{HT}$ of λ-ho-term-graphs into λ-term-graphs as a mapping from $\mathcal{H}$ to $\mathcal{T}$. This induces the first-order term graph semantics $[\![\cdot]\!]_{\mathcal{T}} := \mathcal{HT} \circ [\![\cdot]\!]_{\mathcal{H}}$, for which we also provide a direct inductive definition.

In Section 6 we define the readback rb with the desired property as a function from λ-term-graphs to $\lambda_{\mathsf{letrec}}$-terms. Subsequently in Section 7 we report on the complexity of the described methods, individually, and in total for the methods described in Subsection 1.4.

In Section 8 we link to our implementation of the presented methods. Finally in Section 9 we explain easy modifications, describe possible extensions, and sketch potential practical applications.

1.6 Applications and scalability

While our contribution is at first a conceptual one, it holds the promise for a number of practical applications:

- Increasing the efficiency of the execution of programs by transforming them into their maximally shared form at compile-time.
- Increasing the efficiency of the execution of programs by repeatedly compactifying the program at run time.
- Improving systems for recognising program equivalence.
- Providing feedback to the programmer, along the lines: 'This code has identical fragments and can be written more compactly.'

These and a number of other potential applications are discussed in more detail in Section 9.

The presented methods scale well to larger inputs, due to the quadratic bound on their runtime complexity (see Section 7).

1.7 Relationship with other concepts of sharing

The maximal sharing method is targeted at increasing 'static' sharing in the sense that a program is transformed at compile time into a version with a higher degree of sharing. It is not (at least not a priori) a method for 'dynamic' sharing, i.e. for an evaluator that maintains a certain degree of sharing at run time, such as graph rewrite mechanisms for fully-lazy [31] or optimal evaluation [1] of the λ-calculus. However, we envisage run-time collapsing of the program's graph interpretation integrated with the evaluator (see Section 9).

The term 'maximal sharing' stems from work on the ATERM library [5]. It describes a technique for minimising memory usage when representing a set of terms in a first-order term rewrite system (TRS). The terms are kept in an aggregate directed acyclic graph by which their syntax trees are shared as much as possible. Thereby terms are created only if they are entirely new; otherwise they are referenced by pointers to roots of sub-dags. Our use of the expression 'maximal sharing' is inspired by that work, but our results generalise that approach in the following ways:

- Instead of first-order terms we consider terms in a higher-order language with the letrec-construct for expressing sharing.
- Since letrec typically defines cyclic sharing dependencies, we interpret terms as cyclic graphs instead of just dags.
- We are interested in increasing sharing by bisimulation collapse instead of by identifying isomorphic sub-dags.

ATERM only checks for equality of subexpressions. Therefore it only introduces horizontal sharing and implements a form of *common subexpression elimination (CSE)* [28, p. 241]. Our approach is stronger than CSE: while Ex. 1.1 can be handled by CSE, this is not the case for Ex. 1.2. In contrast to CSE, our approach increases also vertical and twisted sharing.[2]

1.8 Contribution of this paper in context

Blom introduces *higher-order term graphs* [4], which are extensions of first-order term graphs by adding a scope function that assigns a set of vertices, its *scope*, to every abstraction vertex.

In the paper [12] we introduced, for interpreting $\lambda_{\mathsf{letrec}}$-terms, a modification of Blom's higher-order term graphs (the λ-ho-term-graphsof the class $\mathcal{H}$) in which scopes are represented by means of 'abstraction prefix functions'. We also investigated first-order term graphs with scope-delimiter vertices. In particular, we examined which specific class of first-order λ-term-graphs can faithfully represent higher-order λ-ho-term-graphs in such a way that compactification of the latter can be realised through bisimulation collapse of the former (this led to the λ-term-graphs of the class $\mathcal{T}$).

Whereas in the paper [12] we exclusively focused on the graph formalisms, and investigated them in their own right, here we connect the results obtained there to the language $\lambda_{\mathsf{letrec}}$ for expressing sharing and cyclicity. Since the methods presented here are based on the graph formalisms, and rely on their properties for correctness, we recapitulate the concepts and the relevant results in Sec. 4 and 5.

The translation $[\![\cdot]\!]_{\mathcal{T}}$ of $\lambda_{\mathsf{letrec}}$-terms into first-order term graphs was inspired by related representations that use scope delimiters to indicate end of scopes. Such representations are generalisations of a de Bruijn index notation for λ-terms [8] in which the de Bruijn indexes are numerals of the form $\mathsf{S}(\ldots(\mathsf{S}(0))\ldots)$. In the generalised form, due to Patterson and Bird [3], the symbol S can occur anywhere between a variable occurrence and its binding abstraction. The idea to view S as a scope delimiter was employed by Hendriks and van Oostrom, who defined an end-of-scope symbol $\lambda\!\!\!/$ [17]. It is also crucial for Lambdascope-graphs (interaction nets) on which van Oostrom defines an optimal evaluator for the λ-calculus [26].

In the report [11], and in the paper [13] we used a closely related higher-order rewrite system in order to precisely characterise those λ^{∞}-terms that can be expressed by (in the sense that they arise as infinite unfoldings of) finite terms in $\lambda_{\mathsf{letrec}}$, and respectively, in λ_{μ}.

2. Preliminaries

By $\mathbb{N}$ we denote the natural numbers including zero. For words w over an alphabet A, the length of w is denoted by $|w|$.

Let Σ be a TRS-signature [30] with arity function $ar : \Sigma \to \mathbb{N}$. A *term graph over* Σ (or a Σ-*term-graph*) is a tuple $\langle V, lab, args, r\rangle$ where: V is a set of *vertices*, $lab : V \to \Sigma$ the *(vertex) label function*, $args : V \to V^*$ the *argument function* that maps every vertex v to the word $args(v)$ consisting of the $ar(lab(v))$ successor vertices of v (hence $|args(v)| = ar(lab(v))$), and r, the *root*, is a vertex in V. Term graphs may have infinitely many vertices.

Let G be a term graph over signature Σ. As useful notation for picking out an arbitrary vertex, or the i-th vertex, from among the ordered successors of a vertex v in G, we define for each $i \in \mathbb{N}$ the indexed edge relation $\rightarrowtail_i \subseteq V \times V$, and additionally the (not indexed) edge relation $\rightarrowtail \subseteq V \times V$, by stipulating for all $w, w' \in V$:

$$w \rightarrowtail_i w' :\Leftrightarrow \exists w_0, \ldots, w_n \in V.\ args(w) = w_0 \cdots w_n \wedge w' = w_i$$
$$w \rightarrowtail w' :\Leftrightarrow \exists i \in \mathbb{N}.\ w \rightarrowtail_i w'$$

A *path* in G is described by $w_0 \rightarrowtail_{k_1} w_1 \rightarrowtail_{k_2} \cdots \rightarrowtail_{k_n} w_n$, where $w_0, w_1, \ldots, w_n \in V$ and $n, k_1, k_2, \ldots, k_n \in \mathbb{N}$. An *access path* of a

[2] For definitions of horizontal, vertical, and twisted sharing we refer to [4].

vertex w of G is a path that starts at the root of G, ends in w, and does not visit any vertex twice. Access paths need not be unique. A term graph is *root-connected* if every vertex has an access path.

Note: By a 'term graph' we will, from now on, always mean a root-connected term graph.

Let $G_1 = \langle V_1, lab_1, args_1, r_1\rangle$, $G_2 = \langle V_2, lab_2, args_2, r_2\rangle$ be term graphs over signature Σ, in the sequel.

A *bisimulation* between G_1 and G_2 is a relation $R \subseteq V_1 \times V_2$ such that the following conditions hold, for all $\langle w, w'\rangle \in R$:

$$\left.\begin{array}{rr} \langle r_1, r_2\rangle \in R & \text{(roots)} \\ lab_1(w) = lab_2(w') & \text{(labels)} \\ \langle args_1(w), args_2(w')\rangle \in R^* & \text{(arguments)} \end{array}\right\} \quad (1)$$

where the extension $R^* \subseteq V_1^* \times V_2^*$ of R to a relation between words over V_1 and words over V_2 is defined as:

$$\begin{aligned} R^* := \{\langle w_1 \cdots w_k, w_1' \cdots w_k'\rangle \mid \\ w_1, \ldots, w_k \in V_1, w_1', \ldots, w_k' \in V_2, \\ \text{for } k \in \mathbb{N} \text{ such that } \langle w_i, w_i'\rangle \in R \text{ for all } 1 \le i \le k\}. \end{aligned}$$

We write $G_1 \leftrightarrows G_2$ if there is a bisimulation between G_1 and G_2, and we say, in this case, that G_1 and G_2 are *bisimilar*. Bisimilarity $\leftrightarrows$ is an equivalence relation on term graphs.

A *functional bisimulation* from G_1 to G_2 is a bisimulation that is the graph of a function from V_1 to V_2. An alternative characterisation of this concept is that of *homomorphism* from G_1 to G_2: a morphism from the structure G_1 to the structure G_2, that is, a function $h : V_1 \to V_2$ such that, for all $v \in V_1$ it holds:

$$\left.\begin{array}{rr} h(r_1) = r_2 & \text{(roots)} \\ lab_1(v) = lab_2(h(v)) & \text{(labels)} \\ h^*(args_1(v)) = args_2(h(v)) & \text{(arguments)} \end{array}\right\} \quad (2)$$

where h^* is the homomorphic extension $h^* : V_1^* \to V_2^*$, $v_1 \cdots v_n \mapsto h(v_1) \cdots h(v_n)$ of h to words over V_1. We write $G_1 \rightarrow G_2$ if there is a functional bisimulation (a homomorphism) from G_1 to G_2. An *isomorphism* between G_1 and G_2 is a bijective homomorphism $i : V_1 \to V_2$ from G_1 to G_2. If there is an isomorphism between G_1 and G_2, we write $G_1 \simeq G_2$, and say that G_1 and G_2 are *isomorphic*.

Let $f \in \Sigma$. An *f-homomorphism* between G_1 and G_2 is a homomorphism h between G_1 and G_2 that shares only vertices with the label f: $h(w_1) = h(w_2) \;\Rightarrow\; lab_1(w_1) = lab_1(w_2) = f$ holds for all $w_1, w_2 \in V_1$. If this is the case, we write $G_1 \rightarrow^f G_2$. An *f-bisimulation* between G_1 and G_2 is a bisimulation between G_1 and G_2 such that its restriction to vertices with labels different from f is a bijective function. We use $\leftrightarrows^f$ to indicate *f-bisimilarity*.

The relation $\rightarrow$ is a preorder, the *sharing preorder* on the class of term graphs over a given signature Σ. It induces a partial order on the isomorphism equivalence classes of term graphs over Σ.

Let $G = \langle V, lab, args, r\rangle$ be a term graph. A *bisimulation collapse* of G is a maximal element in the class $\{G' \mid G \rightarrow G'\}$ up to $\simeq$, that is, a term graph G_0' with $G \rightarrow G_0'$ such that if $G_0' \rightarrow G_0''$ for some term graph G_0'', then $G_0'' \simeq G_0'$. The *canonical bisimulation collapse* $G|\!\downarrow$ of G is defined as the root-connected part of the 'factor term graph' $G/_R$ of G with respect to the largest bisimulation R between G and G (the largest 'self-bisimulation' on G), which is an equivalence relation on V. The *factor term graph* $G/_\sim$ of G with respect to an equivalence relation $\sim$ on V is defined as $G/_\sim := \langle V/_\sim, lab/_\sim, args/_\sim, [r]_\sim\rangle$ where $V/_\sim$ is the set of $\sim$-equivalence classes of vertices in V, $[r]_\sim$ is the $\sim$-equivalence class of r, and $lab/_\sim$ and $args/_\sim$ are the mappings on $V/_\sim$ that are induced by lab and $args$, respectively. Every two bisimulation collapses of G are isomorphic. This justifies the common abbreviation of saying that 'the bisimulation collapse' of G is unique up to isomorphism.

3. Unfolding Semantics of $\lambda_{\mathsf{letrec}}$-terms

Informally, we regard *$\lambda_{\mathsf{letrec}}$-terms* as being defined by the following grammar:

$$\begin{array}{lcll} L & ::= & \lambda x.\, L & (\textit{abstraction}) \\ & | & L\, L & (\textit{application}) \\ & | & x & (\textit{variable}) \\ & | & \mathsf{let}\ B\ \mathsf{in}\ L & (\textit{letrec}) \\ B & ::= & f_1 = L, \ldots, f_n = L & (\textit{equations}) \\ & & (f_1, \ldots, f_n \in \mathcal{R} \text{ all distinct}) & \end{array}$$

Formally, we consider $\lambda_{\mathsf{letrec}}$-terms to be defined correspondingly as terms in the formalism of Combinatory Reduction Systems (CRS) [30]. CRSs are a higher-order term rewriting framework tailor-made for formalising and manipulating expressions in higher-order languages (i.e. languages with binding constructs like λ-abstractions and let-bindings). They provide a sound basis for defining our language and for reasoning with letrec-expressions. By formalising a system of unfolding rules as a CRS we conveniently externalise issues like name capturing and α-renaming, which otherwise would have to be handled by a calculus of explicit substitution. Also, we can lean on the rewriting theory of CRSs for the proofs.

As CRS-signature we use $\Sigma_{\lambda_{\mathsf{letrec}}} = \Sigma_\lambda \cup \{\mathsf{let}_n, \mathsf{rec\text{-}in}_n \mid n \in \mathbb{N}\}$, with $\Sigma_\lambda = \{\mathsf{abs}, \mathsf{app}\}$, where the unary symbol abs and the binary symbol app represent λ-abstraction and application, respectively; the symbols let_n of arity one, and $\mathsf{rec\text{-}in}_n$ of arity $n+1$ together formalise let-expressions with n bindings. By $|L|$ we denote the size (the number of symbols) of a $\lambda_{\mathsf{letrec}}$-term L. By $Ter(\lambda_{\mathsf{letrec}})$ we denote the set of CRS-terms over $\Sigma_{\lambda_{\mathsf{letrec}}}$. For readability, we rely on the informal first-order notation.

Infinite λ-terms are formalised as iCRS-terms (terms in an infinitary CRS [22]) over Σ_λ, forming the set $Ter(\lambda^\infty)$. Informally, infinite λ-terms are generated co-inductively by the alternatives (*abstraction*), (*application*), and (*variable*) of the grammar above.

In order to formally define the infinite unfolding of $\lambda_{\mathsf{letrec}}$-terms we utilise a CRS whose rewrite rules formalise unfolding steps [11]. Every $\lambda_{\mathsf{letrec}}$-term L that represents an infinite λ-term M can be rewritten by a typically infinite rewrite sequence that converges to M in the limit. However, not every $\lambda_{\mathsf{letrec}}$-term represents an λ^∞-term. For instance the $\lambda_{\mathsf{letrec}}$-term $Q = \lambda x.\, \mathsf{let}\ f = f\ \mathsf{in}\ f\, x$ with a meaningless let-binding for f does not unfold to a λ^∞-term. Therefore we introduce a constant symbol $\bullet$, called 'black hole', for expressing meaningless bindings, in order to define the unfolding operation as a total function. The unfolding semantics of Q will then be $\lambda x.\, \bullet\, x$. So we extend the signature Σ_λ to $\Sigma_{\lambda_\bullet}$ including $\bullet$, and denote the set of infinite λ-terms over Σ_λ by $Ter(\lambda_\bullet^\infty)$. Similarly, the rules below are defined for terms in $Ter(\lambda_{\mathsf{letrec},\bullet})$ based on signature $\Sigma_{\lambda_{\mathsf{letrec},\bullet}}$ that extends $\Sigma_{\lambda_{\mathsf{letrec}}}$ by the blackhole constant.

Definition 3.1 (unfolding CRS for $\lambda_{\mathsf{letrec}}$-terms)**.** The rules:

$$\begin{array}{ll} (@) & \mathsf{let}\ B\ \mathsf{in}\ L_0\, L_1 \;\to\; (\mathsf{let}\ B\ \mathsf{in}\ L_0)\, (\mathsf{let}\ B\ \mathsf{in}\ L_1) \\ (\lambda) & \mathsf{let}\ B\ \mathsf{in}\ \lambda x.\, L_0 \;\to\; \lambda x.\, \mathsf{let}\ B\ \mathsf{in}\ L_0 \\ (\mathsf{let\_in}) & \mathsf{let}\ B_0\ \mathsf{in}\ \mathsf{let}\ B_1\ \mathsf{in}\ L \;\to\; \mathsf{let}\ B_0, B_1\ \mathsf{in}\ L \\ (\mathsf{let\text{-}rec}) & \mathsf{let}\ B_1, f = L, B_2\ \mathsf{in}\ f \;\to\; \mathsf{let}\ B_1, f = L, B_2\ \mathsf{in}\ L \\ (\mathrm{gc}) & \mathsf{let}\ f_1 = L_1, \ldots, f_n = L_n\ \mathsf{in}\ P \;\to\; P \\ & \qquad (\text{if } f_1, \ldots, f_n \text{ do not occur in } P) \\ (\mathrm{tighten}) & \mathsf{let}\ B_1, f = g, B_2\ \mathsf{in}\ L \\ & \qquad \to\ \mathsf{let}\ B_1[f := g], B_2[f := g]\ \mathsf{in}\ L[f := g] \\ & \qquad (\text{where } g \text{ with } g \neq f \text{ a recursion variable in } B_1 \text{ or } B_2) \\ (\bullet) & \mathsf{let}\ B_1, f = f, B_2\ \mathsf{in}\ L \;\to\; \mathsf{let}\ B_1, f = \bullet, B_2\ \mathsf{in}\ L \end{array}$$

define, in informal notation, the *unfolding CRS* for $\lambda_{\mathsf{letrec}}$-terms with rewrite relation $\to_{\mathrm{unf}}$. Here is the CRS-notation for two of the rules:

$$(\lambda)\quad \mathsf{let}_n([\vec{f}]\,\mathsf{rec\text{-}in}_n(X_1(\vec{f}),\ldots,X_n(\vec{f}),\mathsf{abs}([x]\,Z(\vec{f},x))))$$
$$\to\ \mathsf{abs}([x]\,\mathsf{let}_n([\vec{f}]\,\mathsf{rec\text{-}in}_n(X_1(\vec{f}),\ldots,X_n(\vec{f}),Z(\vec{f},x))))$$
$$(\mathsf{let\_in})\quad \mathsf{let}_n([\vec{f}]\,\mathsf{rec\text{-}in}_n(\vec{X}(\vec{f}),\mathsf{let}_m([\vec{g}]\,\mathsf{rec\text{-}in}_m(\vec{Y}(\vec{f},\vec{g}))),Z(\vec{f},\vec{g})))$$
$$\to\ \mathsf{let}_{n+m}([\vec{f}\vec{g}]\,\mathsf{rec\text{-}in}_{n+m}(\vec{X}(\vec{f}),\vec{Y}(\vec{f},\vec{g}),Z(\vec{f},\vec{g})))$$

Example 3.2 (Unfolding derivation of L from Ex. 1.2)**.**
$\lambda f.\,\mathsf{let}\ r = f\ r\ \mathsf{in}\ r \to^{(\text{let-rec})}_{\text{unf}} \lambda f.\,\mathsf{let}\ r = f\ r\ \mathsf{in}\ f\ r \to^{(@)}_{\text{unf}}$
$\lambda f.\,(\mathsf{let}\ r = f\ r\ \mathsf{in}\ f)\ (\mathsf{let}\ r = f\ r\ \mathsf{in}\ r) \to^{(\text{gc})}_{\text{unf}}$
$\lambda f.\,f\ (\mathsf{let}\ r = f\ r\ \mathsf{in}\ r) \to^{(\text{let-rec})}_{\text{unf}}\ \ldots$

We say that a $\lambda_{\mathsf{letrec}}$-term L *unfolds to* an $\lambda^{\infty}_{\bullet}$-term M, or that L *expresses* M, if there is a (typically) infinite $\to_{\text{unf}}$-rewrite sequence from L that converges to M, symbolically $L \twoheadrightarrow_{\text{unf}} M$. Note that any such rewrite sequence is strongly convergent (the depth of the contracted redexes tends to infinity), because the resulting term does not contain any let-expressions.

Lemma 3.3. *Every $\lambda_{\mathsf{letrec}}$-term unfolds to precisely one $\lambda^{\infty}_{\bullet}$-term.*

Proof (Outline). Infinite normal forms of $\to_{\text{unf}}$ are $\lambda^{\infty}_{\bullet}$-terms since: every occurrence of a let-expression in a $\lambda_{\mathsf{letrec},\bullet}$-term gives rise to a redex; and infinite $\lambda_{\mathsf{letrec},\bullet}$-terms without let-expressions are $\lambda^{\infty}_{\bullet}$-terms. Also, outermost-fair rewrite sequences in which the rules (tighten) and ($\bullet$) are applied eagerly are (strongly) convergent.

Unique infinite normalisation of $\to_{\text{unf}}$ follows from finitary confluence of $\to_{\text{unf}}$. In previous work [11] we proved confluence for the slightly simpler CRS without the final two rules, which together introduce black holes in terms with meaningless bindings. That confluence proof can be adapted by extending the argumentation to deal with the additional critical pairs. □

Definition 3.4. The *unfolding semantics* for $\lambda_{\mathsf{letrec}}$-terms is defined by the function $[\![\cdot]\!]_{\lambda^{\infty}_{\bullet}} : Ter(\lambda_{\mathsf{letrec}}) \to Ter(\lambda^{\infty}_{\bullet})$, where $L \mapsto [\![L]\!]_{\lambda^{\infty}_{\bullet}}$:= the infinite unfolding of L.

Remark 3.5 (Regular and strongly regular λ^{∞}-terms)**.** λ^{∞}-terms that arise as infinite unfoldings of $\lambda_{\mathsf{letrec}}$-terms form a proper subclass of those λ^{∞}-terms that have a regular term structure [11]. λ^{∞}-terms that belong to this subclass are called 'strongly regular', and can be characterised by means of a decomposition rewrite system, and as those that contain only finite 'binding–capturing chains' [11, 13].

4. Lambda higher-order term graphs

In this section we motivate the use of higher-order term graphs as a semantics for $\lambda_{\mathsf{letrec}}$-terms; we introduce the class $\mathcal{H}$ of 'λ-ho-term-graphs' and define the semantics $[\![\cdot]\!]_{\mathcal{H}}$ for interpreting $\lambda_{\mathsf{letrec}}$-terms as λ-ho-term-graphs. Finally, we sketch a proof of the correctness of $[\![\cdot]\!]_{\mathcal{H}}$ with respect to unfolding equivalence (the property (P1)).

We start out from a natural interpretation of $\lambda_{\mathsf{letrec}}$-terms as first-order term graphs: occurrences of abstraction variables are resolved as edges pointing to the corresponding abstraction; occurrences of recursion variables as edges to the subgraph belonging to the respective binding. We therefore consider term graphs over the signature $\Sigma^{\lambda}_{\bullet} = \{@, \lambda, 0, \bullet\}$ with arities $ar(@) = 2$, $ar(\lambda) = 1$, $ar(0) = 1$, and $ar(\bullet) = 0$. These function symbols represent applications, λ-abstractions, abstraction variables, and black holes.

We will later define a subclass of these term graphs that excludes meaningless graphs. In line with the choice to regard all terms as higher-order terms (thus modulo α-conversion), we consider a nameless graph representation, so that α-equivalence of two terms can be recognised as their graph interpretations being isomorphic.

For a term graph G over $\Sigma^{\lambda}_{\bullet}$ with set V of vertices we will henceforth denote by $V(@)$, $V(\lambda)$, $V(0)$, and $V(\bullet)$ the sets of *application vertices*, *abstraction vertices*, *variable vertices*, and *blackhole vertices*, that is, those with label @, λ, 0, $\bullet$, respectively.

Example 4.1 (Natural first-order interpretation)**.** The $\lambda_{\mathsf{letrec}}$-terms L and P in Ex. 1.2 can be represented as the term graphs in Fig. 2.

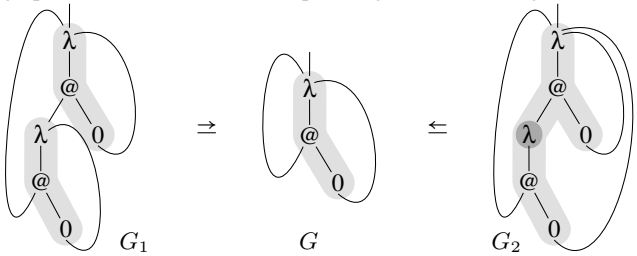
These two graphs are bisimilar, which suggests that L and P are unfolding-equivalent. Moreover, there is a functional bisimulation from the larger term graph to the smaller one, indicating that L expresses more sharing than P, or in other words: L is more compact. Also, there is no smaller term graph that is bisimilar to L and P. We conclude that L is a maximally shared form of P.

However, this translation is incorrect in the sense that bisimilarity does not in general guarantee unfolding equivalence, the desired property (P1). This is witnessed by the following counterexample.

Example 4.2 (Incorrectness of the natural first-order interpretation)**.**

$$L_1 = \mathsf{let}\ f = \lambda x.\,(\lambda y.\,f\ y)\ x\ \mathsf{in}\ f$$
$$L = \mathsf{let}\ f = \lambda x.\,f\ x\ \mathsf{in}\ f$$
$$L_2 = \mathsf{let}\ f = \lambda x.\,(\lambda y.\,f\ x)\ x\ \mathsf{in}\ f$$

While $[\![L_1]\!]_{\lambda^{\infty}} = [\![L]\!]_{\lambda^{\infty}}$ and $[\![L]\!]_{\lambda^{\infty}} \neq [\![L_2]\!]_{\lambda^{\infty}}$, all of their term graphs G_1, G, G are bisimilar (please ignore the shading for now):



Consequently this interpretation lacks the necessary structure for correctly modelling compactification via bisimulation collapse.

We therefore impose additional structure on the term graphs. This is indicated by the shading in the picture above, and in the graphs throughout this paper. A shaded area depicts the *scope* of an abstraction: it comprises all positions between the abstraction and its bound variable occurrences as well as the scope of any abstraction on these positions. By this stipulation, scopes are properly nested.

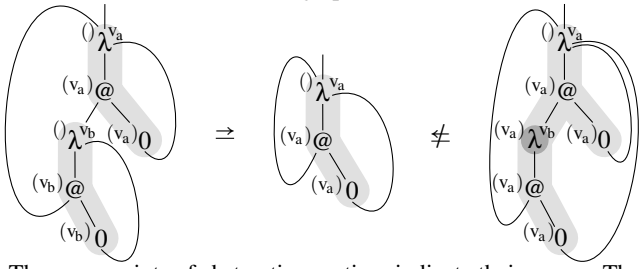
Now note that the functional bisimulation on the right in the picture in Ex. 4.2 does *not* respect the scopes: The scope of the topmost abstraction vertex in the term graph G_2 interpreting L_2 contains another λ-abstraction; hence the image of this scope under the functional bisimulation cannot fit into, and is not contained in, the single scope in the term graph G of L. Also, the trivial scope of the vacuous abstraction in G_2 is not mapped to a scope in G. Thus the natural first-order interpretation is incorrect, in the sense that functional bisimulation does not preserve scopes on the first-order term graphs that are interpretations of $\lambda_{\mathsf{letrec}}$-terms.

To prevent that interpretations of not unfolding-equivalent terms like L_1 and L_2 in Ex. 4.2 become bisimilar, we enrich first-order term graphs by a formal concept of scope. More precisely, *abstraction prefixes* are added as vertex labels. They also serve the purpose of defining the subclass of meaningful term graphs over $\Sigma^{\lambda}_{\bullet}$ that sensibly represent cyclic λ-terms. In the enriched term graphs, each vertex v is annotated with a label $P(v)$, the *abstraction prefix* of v, which is a list of vertex names that identifies the abstraction vertices in whose scope v resides. Alternatively scopes can be represented by a scope function (as in [4]) that assigns to every abstraction vertex the set of vertices in its scope. In the article [12] we show that higher-order term graphs with scope functions correspond bijectively to those with abstraction prefix functions.

Abstraction prefixes can be determined by traversing over the graph and recording every binding encountered. When passing an abstraction vertex v while descending into the subgraph representing the body of the abstraction, one enters or opens the scope of v. This is recorded by appending v to the abstraction prefix of v's successor.

v is removed from the prefix at positions under which the abstraction variable is no longer used, but not before any other variable that was added to the prefix in the meantime has itself been removed. In other words, the abstraction prefix behaves like a stack. We call term graphs for representing $\lambda_{\mathsf{letrec}}$-terms that are equipped with abstraction-prefixes 'λ-higher-order term graphs' (λ-ho-term-graphs).

Example 4.3 (The λ-ho-term-graphs of the terms in Ex. 4.2)**.**



The superscripts of abstraction vertices indicate their names. The abstraction prefix of a vertex is annotated to its top left. Note that abstraction vertices themselves are not included in their own prefix.

We define λ-ho-term-graphs as term graphs over $\Sigma^{\lambda}_{\bullet}$ together with an abstraction-prefix function that assigns to each vertex an abstraction prefix. It has to respect certain correctness conditions restricting the λ-ho-term-graphs to exclude meaningless term graphs.

Definition 4.4 (correct abstraction-prefix function for term graphs over $\Sigma^{\lambda}_{\bullet}$)**.** Let $G = \langle V, lab, args, r\rangle$ be a $\Sigma^{\lambda}_{\bullet}$-term-graph.

An *abstraction-prefix function* for G is a function $P : V \to V^*$ from vertices of G to words of vertices. Such a function is called *correct* if for all $w, w_0, w_1 \in V$ and $k \in \{0, 1\}$ it holds:

$$P(r) = \epsilon \qquad \text{(root)}$$

$$P(\bullet) = \epsilon \qquad \text{(black hole)}$$

$$w \in V(\lambda) \ \wedge \ w \rightarrowtail_0 w_0 \ \Rightarrow \ P(w_0) \leq P(w)w \qquad (\lambda)$$

$$w \in V(@) \ \wedge \ w \rightarrowtail_k w_k \ \Rightarrow \ P(w_k) \leq P(w) \qquad (@)$$

$$w \in V(0) \ \wedge \ w \rightarrowtail_0 w_0 \ \Rightarrow \ \begin{cases} w_0 \in V(\lambda) \\ \ \wedge \ P(w_0)w_0 = P(w) \end{cases} \qquad (0)$$

Here and later we denote by $\leq$ the 'is-prefix-of' relation.

Definition 4.5 (λ-ho-term-graph)**.** A *λ-ho-term-graph* over $\Sigma^{\lambda}_{\bullet}$ is a five-tuple $\mathcal{G} = \langle V, lab, args, r, P\rangle$ where $G_{\mathcal{G}} = \langle V, lab, args, r\rangle$ is a term graph over $\Sigma^{\lambda}_{\bullet}$, called the term graph *underlying* $\mathcal{G}$, and P is a correct abstraction-prefix function for $G_{\mathcal{G}}$. The class of λ-ho-term-graphs over $\Sigma^{\lambda}_{\bullet}$ is denoted by $\mathcal{H}$.

Definition 4.6 (homomorphism, bisimulation for λ-ho-term-graphs)**.** Let $\mathcal{G}_1 = \langle V_1, lab_1, args_1, r_1, P_1\rangle$ and $\mathcal{G}_2 = \langle V_2, lab_2, args_2, r_2, P_2\rangle$ be λ-ho-term-graphs over $\Sigma^{\lambda}_{\bullet}$.

A *bisimulation* between $\mathcal{G}_1$ and $\mathcal{G}_2$ is a relation $R \subseteq V_1 \times V_2$ that is a bisimulation between the term graphs $G_{\mathcal{G}_1}$ and $G_{\mathcal{G}_2}$ underlying $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively, and for which also the following condition:

$$\langle P_1(w), P_2(w')\rangle \in R^* \qquad \text{(abstraction-prefix functions)} \quad (3)$$

(for R^* see p. 4 below (1)) is satisfied for all $w \in V_1$ and all $w' \in V_2$. If there is a such bisimulation between $\mathcal{G}_1$ and $\mathcal{G}_2$, then we say that $\mathcal{G}_1$ and $\mathcal{G}_2$ are *bisimilar*, and denote this fact by $\mathcal{G}_1 \leftrightarrows \mathcal{G}_2$.

A *homomorphism* (a *functional bisimulation*) from $\mathcal{G}_1$ to $\mathcal{G}_2$ is a morphism from the structure $\mathcal{G}_1$ to the structure $\mathcal{G}_2$, or more explicitly, it is a homomorphism $h : V_1 \to V_2$ from $G_{\mathcal{G}_1}$ to $G_{\mathcal{G}_2}$ that additionally satisfies, for all $w \in V_1$, the following condition:

$$h^*(P_1(w)) = P_2(h(w)) \qquad \text{(abstraction-prefix functions)} \quad (4)$$

where h^* is the homomorphic extension of h to words over V_1. We write $\mathcal{G}_1 \rightrightarrows \mathcal{G}_2$ if there is a homomorphism from $\mathcal{G}_1$ to $\mathcal{G}_2$.

4.1 Interpretation of $\lambda_{\mathsf{letrec}}$-terms as λ-ho-term-graphs

In order to interpret a $\lambda_{\mathsf{letrec}}$-term L as λ-ho-term-graph, the translation rules $\mathcal{R}$ from Fig. 3 are applied to a 'translation box' $\boxed{(*[])\,L}$. It contains L furnished with a prefix consisting of a dummy variable $*$, and an empty set $[]$ of binding equations. The translation process proceeds by induction on the syntactical structure of the prefixed $\lambda_{\mathsf{letrec}}$-expression's body. Ultimately, a term graph G over $\Sigma^{\lambda}_{\bullet}$ is produced, together with a correct abstraction-prefix function for G.

For reading the rules $\mathcal{R}$ in Fig. 3 correctly, observe the details as described here below. Illustrations of the translation process when applied to two $\lambda_{\mathsf{letrec}}$-terms used here can be found in Appendix A of the extended version [15] of this paper.

- A translation box $\boxed{(\vec{p})\,L}$ contains a prefixed, partially decomposed $\lambda_{\mathsf{letrec}}$-term L. The prefix contains a vector $\vec{p}$ of annotated λ-abstractions that have already been translated and whose scope typically extends into L. Every prefix abstraction is annotated with a set of binding equations that are defined at its level. There is special dummy variable denoted by $*$ at the left of the prefix that carries top-level function bindings, i.e. binding equations that are not defined under any enclosing λ-abstraction. The λ-rule strips off an abstraction from the body of the expression, and pushes the abstraction variable into the prefix, which initially contains an empty set of function bindings.
- Names of abstraction vertices are indicated to the right, and abstraction-prefixes to the left of the created vertices. In order to refer to the vertices in the prefix we use the following notation: $vs(\vec{p}) = v_1 \cdots v_n$ if $\vec{p} = *[B_0]\ x_1^{v_1}[B_1]\ \ldots\ x_n^{v_n}[B_n]$.
- Vertices drawn with dashed lines have been created in earlier translation steps, and are referenced by edges in the current step.
- In the S-rule, which takes care of closing scopes, $FV(L)$ stands for the set of free variables in L.
- The let-rule for translating let-expressions creates a box for the in-part as well as for each function binding. The translation of each of the bindings starts with an *indirection vertex*. These vertices guarantee the well-definedness of the process when it translates meaningless bindings such as $f = f$, or $g = h$, $h = g$, which would otherwise give rise to loops without vertices. The let-rule pushes the function bindings into the abstraction prefix, associating each function binding with one of the variables in the abstraction prefix. There is some freedom as to which variable a function binding is assigned to. This freedom is limited by scoping conditions that ensure that the prefixed term is a valid CRS-term: function bindings may only depend on variables and functions that occur further to the left in the prefix. The chosen association also directly determines the prefix lengths used in the translation boxes for the function bindings.
- Indirection vertices are eliminated by an erasure process at the end: Every indirection vertex that does not point to itself is removed, redirecting all incoming edges to the successor vertex. Finally every loop on a single indirection vertex is replaced by a *black hole* vertex that represents a meaningless binding. Abstraction prefixes for such black holes are defined to be empty.

Definition 4.7. We say that a term graph G over $\Sigma^{\lambda}_{\bullet}$ and an abstraction-prefix function P is *$\mathcal{R}$-generated from* a $\lambda_{\mathsf{letrec}}$-term L if G and P are obtained by applying the rules $\mathcal{R}$ from Fig. 3 to $\boxed{(*[])\,L}$.

Proposition 4.8. Let L be a $\lambda_{\mathsf{letrec}}$-term. Suppose that a term graph G over $\Sigma^{\lambda}_{\bullet}$, and an abstraction-prefix function P are $\mathcal{R}$-generated from L. Then P is a correct abstraction-prefix function for G, and consequently, G and P together form a λ-ho-term-graph in $\mathcal{H}$.

There are two sources of non-determinism in this translation: the S-rule for shortening prefixes can be applicable at the same time as other rules; also the let-rule does not fix the lengths $l_1, \ldots, l_k$ of the

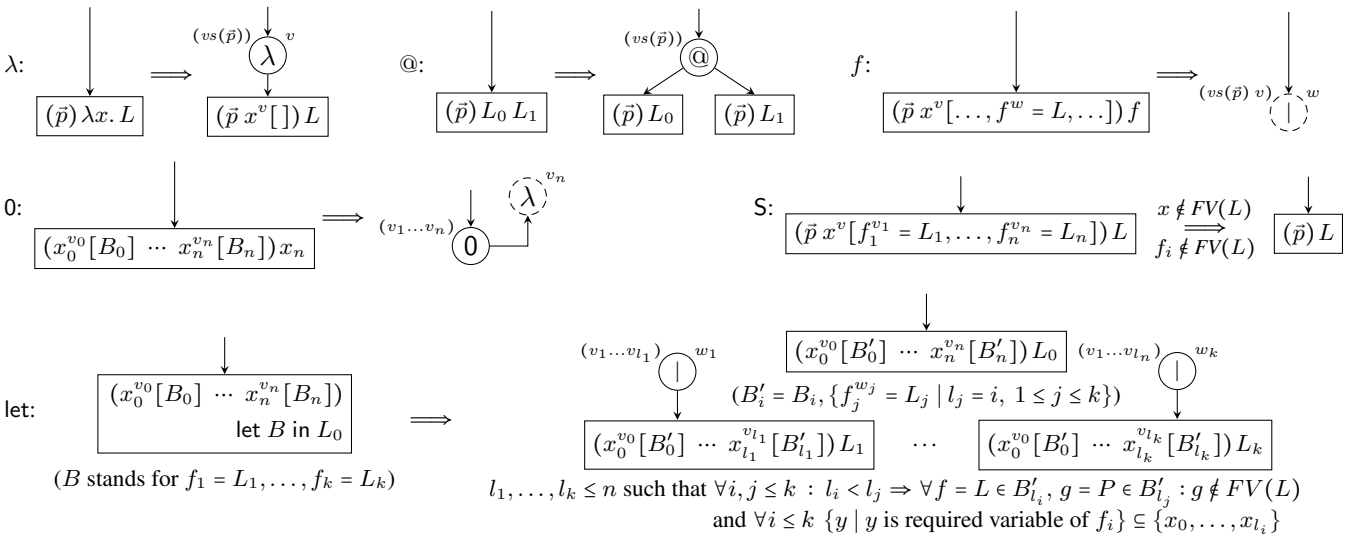


Figure 3. Translation rules $\mathcal{R}$ for interpreting $\lambda_{\mathsf{letrec}}$-terms as λ-ho-term-graphs. See Section 4.1 for explanations.

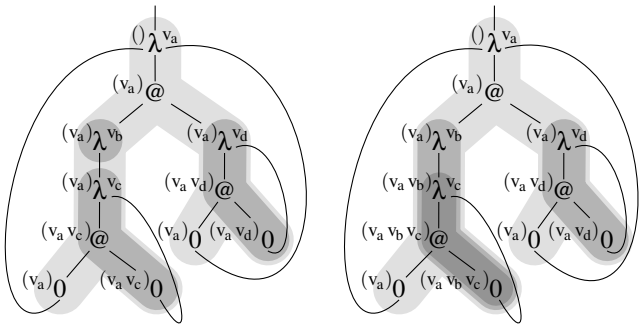


Figure 4. Translation of $\lambda a.\,(\lambda b.\,\lambda c.\,a\,c)\,(\lambda d.\,a\,d)$ with eager scope-closure (left), and with lazy scope-closure (right). While in the left term graph four vertices can be shared, with as result the translation of the term $\lambda a.\,\mathsf{let}\ f = \lambda c.\,a\,c\ \mathsf{in}\ (\lambda b.\,f)\,f$, in the right term graph only a single variable occurrence can be shared.

abstraction prefixes for the translations of the binding equations, but admits various choices of prefixes that are shorter than the prefix of the left-hand side. Neither kind of non-determinism affects the term graph that is produced, but in general several abstraction-prefix functions, and thus different λ-ho-term-graphs, can be obtained.

4.2 Interpretation as eager-scope λ-ho-term-graphs

Of the different translations of a $\lambda_{\mathsf{letrec}}$-term into λ-ho-term-graphs we are most interested in the one with the shortest possible abstraction prefixes. We say that such a term graph has 'eager scope-closure', or that it is 'eager-scope'.The reason for this choice is illustrated in Fig. 4: eager-scope closure allows for more sharing.

Definition 4.9 (eager scope)**.** Let $\mathcal{G} = \langle V, lab, args, r, P\rangle$ be a λ-ho-term-graph. $\mathcal{G}$ is called *eager-scope* if for every $w \in V$ with $P(w) = pv$ for $p \in V^*$ and $v \in V$, there is a path $w = w_0 \twoheadrightarrow w_1 \twoheadrightarrow \cdots \twoheadrightarrow w_m \twoheadrightarrow_0 v$ in $\mathcal{G}$ from w to v with $P(w) \le P(w_i)$ for all $i \in \{1, \ldots, m\}$, and (this follows) $w_m \in V(\mathsf{0})$ and $v \in V(\lambda)$.

Hence if a λ-ho-term-graph is not eager-scope, then it contains a vertex w with abstraction-prefix $v_1 \ldots v_n$ from which v_n is only reachable, if at all, by leaving the scope of v_n. It can be shown that in this case another abstraction-prefix function with shorter prefixes exists, and in which v_n has been removed from the prefix of w.

Proposition 4.10 (eager-scope = minimal scope; uniqueness of eager-scope λ-ho-term-graphs)**.** Let $\mathcal{G}_i = \langle V, lab, args, r, P_i\rangle$ for $i \in \{1, 2\}$ be λ-ho-term-graphs with the same underlying term graph. If $\mathcal{G}_1$ is eager-scope, then $|P_1(w)| \le |P_2(w))|$ for all $w \in V$. If, in addition, also $\mathcal{G}_2$ is eager-scope, then $P_1 = P_2$. Hence eager-scope λ-ho-term-graphs over the same underlying term graph are unique.

Also, we will call a translation process 'eager-scope' if it resolves the non-determinism in $\mathcal{R}$ in such a way that it always yields eager-scope λ-ho-term-graphs. In order to obtain an eager-scope translation we have to consider the following aspects.

Garbage removal. In the presence of *garbage*, unused function bindings, a translation cannot be eager-scope. Consider the term $\lambda x.\,\lambda y.\,\mathsf{let}\ f = x\ \mathsf{in}\ y$. The expendable binding $f = x$ prevents the application of the S-rule, and hence the closure of the scope of λx, directly below λx. Therefore we will assume that *all unused function bindings are removed* prior to applying the rules $\mathcal{R}$. A $\lambda_{\mathsf{letrec}}$-term without garbage will be called *garbage-free*.

Short enough prefix lengths in the* let*-rule. For obtaining an eager-scope translation, we will usually stipulate that the S-rule is applied eagerly, i.e. it is given precedence over the other rules. This is clearly necessary for keeping the abstraction prefixes minimal. But how do we choose the prefix lengths $l_1, \ldots, l_k$ in the let-rule? The prefix lengths l_i determine at which position a binding $f_i = L_i$ is inserted into the abstraction prefixes. Therefore l_i may not be chosen too short; otherwise a function f depending on a function g may end up to the right of g, and hence may be removed from the prefix by the S-rule prematurely, preventing completion of the translation. Yet simply choosing $l_i = n$ may prevent scopes from being minimal. For example, when translating the term $\lambda a.\,\lambda b.\,\mathsf{let}\ f = a\ \mathsf{in}\ a\,a\,(f\,a)\,b$, it is crucial to allow shorter prefixes for the binding than for the in-part. As shown in Fig. 5 the graph on the left does not have eager scope-closure even if the S-rule is applied eagerly. Consequently the opportunity for sharing the lower application vertices is lost.

Required variable analysis. For choosing the prefixes in the let-rule correctly, the translation process must know for each function binding which λ-variables are 'required' on the right-hand side of its definition. For this we use an analysis obtaining the required variables for positions in a $\lambda_{\mathsf{letrec}}$-term as employed by algorithms for lambda-lifting [7, 21]. The term 'required variables' was coined

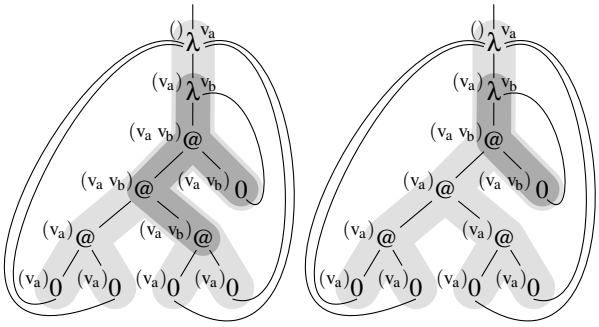


Figure 5. Translation of $\lambda a.\, \lambda b.\, \mathsf{let}\ f = a\ \mathsf{in}\ a\, a\, (f\, a)\, b$ with equal (left) and with minimal prefix lengths (right) in the let-rule.

by Morazán and Schultz [24]. A λ-variable x is called *required at a position* p in a $\lambda_{\mathsf{letrec}}$-term L if x is bound by an abstraction above p, and has a free occurrence in the complete unfolding of L below p (also recursion variables from above p are unfolded). The required variables at position p in L can be computed as those λ-variables with free occurrences that are reachable from p by a downwards traversal with the stipulations: on encountering a let-binding the in-part is entered; when encountering a recursion variable the traversal continues at the right-hand side of the corresponding function binding (even if it is defined above p).

With the result of the required variable analysis at hand, we now define properties of the translation process that can guarantee that the resulting λ-ho-term-graph is eager-scope.

Definition 4.11 (eager-scope and minimal-prefix generated)**.** Let L be a $\lambda_{\mathsf{letrec}}$-term, and let $\mathcal{G}$ be a λ-ho-term-graph.

We say that $\mathcal{G}$ is *eager-scope* $\mathcal{R}$-generated from L if $\mathcal{G}$ is $\mathcal{R}$-generated from L by a translation process with the following property: for every translation box reached during the process with label $(\vec{p}\, x^v[B])\, P$, where P is a subterm of L at position q, it holds that if x is not a required variable at q in L, then in the next translation step performed to this box either one of the rules f or let is applied, or the prefix is shortened by the S-rule.

We say that $\mathcal{G}$ is $\mathcal{R}$-generated *with minimal prefixes* from L if $\mathcal{G}$ is $\mathcal{R}$-generated from L by a translation process in which minimal prefix lengths are achieved by giving applications of the S-rule precedence over applications of all other rules, and by always choosing prefixes minimally in applications of the let-rule.

Proposition 4.12. Let $\mathcal{G}$ be a λ-ho-term-graph that is $\mathcal{R}$-generated from a garbage-free $\lambda_{\mathsf{letrec}}$-term L. The following statements hold:

(i) If $\mathcal{G}$ is eager-scope $\mathcal{R}$-generated from L, then $\mathcal{G}$ is eager-scope.
(ii) If $\mathcal{G}$ is $\mathcal{R}$-generated with minimal prefixes from L, then $\mathcal{G}$ is eager-scope $\mathcal{R}$-generated from L, hence by (i) $\mathcal{G}$ is eager-scope.

Definition 4.13. The semantics $[\![\cdot]\!]_{\mathcal{H}}$ of $\lambda_{\mathsf{letrec}}$-terms as λ-ho-term-graphs is defined as $[\![\cdot]\!]_{\mathcal{H}} : \mathit{Ter}(\lambda_{\mathsf{letrec}}) \to \mathcal{H}$, $L \mapsto [\![L]\!]_{\mathcal{H}} :=$ the λ-ho-term-graph that is $\mathcal{R}$-generated with minimal prefixes from a garbage-free version L' of L.

Proposition 4.14. For every $\lambda_{\mathsf{letrec}}$-term L, $[\![L]\!]_{\mathcal{H}}$ is eager-scope.

4.3 Correctness of $[\![\cdot]\!]_{\mathcal{H}}$ with respect to unfolding semantics

In preparation of establishing the desired property (P1) in Sect. 5, we formulate, and outline the proof of, the fact that the semantics $[\![\cdot]\!]_{\mathcal{H}}$ is correct with respect to the unfolding semantics on $\lambda_{\mathsf{letrec}}$-terms.

Theorem 4.15. *$[\![L_1]\!]_{\lambda^\infty} = [\![L_2]\!]_{\lambda^\infty}$ if and only if $[\![L_1]\!]_{\mathcal{H}} \leftrightarrows [\![L_2]\!]_{\mathcal{H}}$, for all $\lambda_{\mathsf{letrec}}$-terms L_1 and L_2.*

Sketch of Proof. Central for the proof are λ-ho-term-graphs that have tree form and only contain variable backlinks, but no recursive backlinks. They form the class $\mathcal{H}_T \subsetneq \mathcal{H}$. Every $\mathcal{G} \in \mathcal{H}$ has a unique 'tree unfolding' $\mathit{Tree}(\mathcal{G}) \in \mathcal{H}_T$. We make use of the following statements. For all $L, L_1, L_2 \in \mathit{Ter}(\lambda_{\mathsf{letrec}})$, $M, M_1, M_2 \in \mathit{Ter}(\lambda^\infty_\bullet)$, $\mathcal{G}, \mathcal{G}_1, \mathcal{G}_2 \in \mathcal{H}$, and $\mathcal{T}r, \mathcal{T}r_1, \mathcal{T}r_2 \in \mathcal{H}_T$ it can be shown that:

$$L_1 \to_{\mathsf{unf}} L_2 \ \Rightarrow\ [\![L_1]\!]_{\mathcal{H}} \leftarrow [\![L_2]\!]_{\mathcal{H}} \tag{5}$$

$$L \twoheadrightarrow_{\mathsf{unf}} M \ (\text{hence } [\![L]\!]_{\lambda^\infty} = M) \ \Rightarrow\ [\![L]\!]_{\mathcal{H}} \leftarrow [\![M]\!]_{\mathcal{H}} \tag{6}$$

$$[\![M]\!]_{\mathcal{H}} \in \mathcal{H}_T \tag{7}$$

$$[\![M_1]\!]_{\mathcal{H}} \simeq [\![M_2]\!]_{\mathcal{H}} \ \Rightarrow\ M_1 = M_2 \tag{8}$$

$$\mathcal{G} \leftarrow \mathit{Tree}(\mathcal{G}) \tag{9}$$

$$\mathcal{T}r_1 \leftrightarrows \mathcal{T}r_2 \ \Rightarrow\ \mathcal{T}r_1 \simeq \mathcal{T}r_2 \tag{10}$$

$$\mathcal{G}_1 \leftrightarrows \mathcal{G}_2 \ \Rightarrow\ \mathit{Tree}(\mathcal{G}_1) \simeq \mathit{Tree}(\mathcal{G}_2) \tag{11}$$

Hereby (5) is used for proving (6), and (9) with (10) for (11). Now for proving the theorem, let L_1 and L_2 be arbitrary $\lambda_{\mathsf{letrec}}$-terms.

"$\Rightarrow$": Suppose $[\![L_1]\!]_{\lambda^\infty} = [\![L_2]\!]_{\lambda^\infty}$. Let M be the infinite unfolding of L_1 and L_2, i.e., $[\![L_1]\!]_{\mathcal{H}} = M = [\![L_2]\!]_{\mathcal{H}}$. Then by (6) it follows $[\![L_1]\!]_{\mathcal{H}} \leftarrow [\![M]\!]_{\mathcal{H}} \rightarrow [\![L_2]\!]_{\mathcal{H}}$, and hence $[\![L_1]\!]_{\mathcal{H}} \leftrightarrows [\![L_2]\!]_{\mathcal{H}}$.

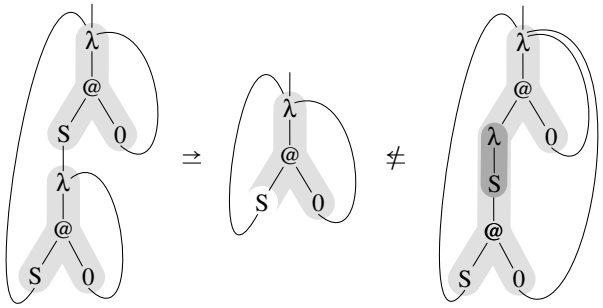
"$\Leftarrow$": Suppose $[\![L_1]\!]_{\mathcal{H}} \leftrightarrows [\![L_2]\!]_{\mathcal{H}}$. Then by (11) it follows that $\mathit{Tree}([\![L_1]\!]_{\mathcal{H}}) \simeq \mathit{Tree}([\![L_2]\!]_{\mathcal{H}})$. Let $M_1, M_2 \in \mathit{Ter}(\lambda^\infty_\bullet)$ be the infinite unfoldings of L_1 and L_2, i.e. $M_1 = [\![L_1]\!]_{\lambda^\infty}$, and $M_2 = [\![L_2]\!]_{\lambda^\infty}$. Then (6) together with the assumption entails $[\![M_1]\!]_{\mathcal{H}} \leftrightarrows [\![M_2]\!]_{\mathcal{H}}$. Since $[\![M_1]\!]_{\mathcal{H}}, [\![M_2]\!]_{\mathcal{H}} \in \mathcal{H}_T$ by (7), it follows by (10) that $[\![M_1]\!]_{\mathcal{H}} \simeq [\![M_2]\!]_{\mathcal{H}}$. Finally, by using (8) we get $M_1 = M_2$, and hence $[\![L_1]\!]_{\lambda^\infty} = M_1 = M_2 = [\![L_2]\!]_{\lambda^\infty}$. □

5. Lambda term graphs

While modelling sharing expressed by $\lambda_{\mathsf{letrec}}$-terms through λ-ho-term-graphs facilitates comparisons via bisimilarity, it is not immediately clear how the compactification of λ-ho-term-graphs via the bisimulation collapse $\downarrow\downarrow$ for λ-ho-term-graphs (which has to respect scopes in the form of the abstraction-prefix functions) can be computed efficiently. We therefore develop an implementation as first-order term graphs, for which standard methods are available.

As witnessed by Ex. 4.2, the scoping information cannot just be discarded, as functional bisimilarity on the underlying term graphs does not faithfully implement functional bisimilarity on λ-ho-term-graphs. Therefore the scoping information has to be incorporated in the first-order interpretation, which we accomplish by extending $\Sigma^\lambda_\bullet$ with S-vertices, scope delimiters, that signify the end of scopes. When translating a λ-ho-term-graph into a first-order term graph, S-vertices are placed along those edges in the underlying term graph at which the abstraction prefix decreases in the λ-ho-term-graph.

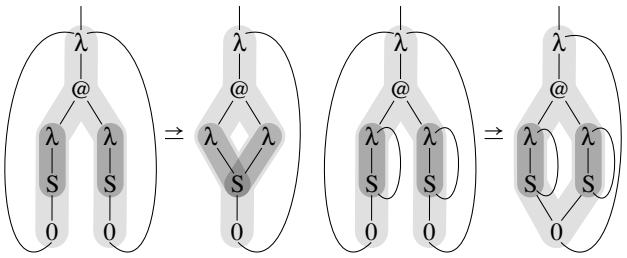
Example 5.1 (Adding S-vertices)**.** Consider the terms in Ex. 4.2 and their λ-ho-term-graphs in Ex. 4.3. In the first-order interpretation below, the shading is just for illustration purposes; it is *not* part of the structure, and does *not directly* impair functional bisimulation.



The addition of scope delimiters resolves the problem of Ex. 4.2. They adequately represent the scoping information.

As for λ-ho-term-graphs, we will define correctness conditions by means of an abstraction-prefix function. However, the current approach with unary delimiter vertices leads to a problem.

Example 5.2 (S-backlinks)**.** The term graph with scope delimiters on the left admits a functional bisimulation that fuses two S-vertices that close different scopes. We cannot hope to find a unique abstraction prefix for the resulting fused S-vertex. This is remedied on the right by using a variant representation that requires backlinks from each S-vertex to the abstraction vertex whose scope it closes. Then S-vertices can only be fused if the corresponding abstractions have already been merged. Hence in the presence of S-backlinks, as in the right illustration below, only the variable vertex can be shared.



Therefore we consider term graphs over the extension $\Sigma^{\lambda}_{\mathsf{S},\bullet}$ of $\Sigma^{\lambda}_{\bullet}$ with a symbol S of arity 2; one edge targets the successor vertex, the other is a backlink. We give correctness conditions, similar as for λ-ho-term-graphs, and define the arising class of 'λ-term-graphs'.

Definition 5.3 (correct abstraction-prefix function for term graphs over $\Sigma^{\lambda}_{\mathsf{S},\bullet}$)**.** Let $G = \langle V, lab, args, r\rangle$ be a $\Sigma^{\lambda}_{\mathsf{S},\bullet}$-term-graph.

An *abstraction-prefix function* $P : V \to V^*$ on G is called *correct* if for all $w, w_0, w_1 \in V$ and $k \in \{0, 1\}$ it holds:

$$P(r) = \epsilon \qquad \text{(root)}$$

$$P(\bullet) = \epsilon \qquad \text{(black hole)}$$

$$w \in V(\lambda) \;\wedge\; w \rightarrowtail_0 w_0 \;\Rightarrow\; P(w_0) = P(w)w \qquad (\lambda)$$

$$w \in V(@) \;\wedge\; w \rightarrowtail_k w_k \;\Rightarrow\; P(w_k) = P(w) \qquad (@)$$

$$w \in V(\mathsf{0}) \;\wedge\; w \rightarrowtail_0 w_0 \;\Rightarrow\; \begin{cases} w_0 \in V(\lambda) \\ \;\wedge\; P(w_0)w_0 = P(w) \end{cases} \qquad (\mathsf{0})_1$$

$$w \in V(\mathsf{S}) \;\wedge\; w \rightarrowtail_0 w_0 \;\Rightarrow\; \begin{cases} P(w_0)v = P(w) \\ \quad \text{for some } v \in V \end{cases} \qquad (\mathsf{S})_1$$

$$w \in V(\mathsf{S}) \;\wedge\; w \rightarrowtail_1 w_1 \;\Rightarrow\; \begin{cases} w_1 \in V(\lambda) \\ \;\wedge\; P(w_1)w_1 = P(w) \end{cases} \qquad (\mathsf{S})_2$$

While in λ-ho-term-graphs the abstraction prefix can shrink by several vertices along an edge (cf. Def. 4.4), here the situation is strictly regulated: the prefix can only shrink by one variable, and only along the outgoing edge of a delimiter vertex.

Proposition 5.4 (uniqueness of the abstraction prefix function)**.** Let G be a term graph over the signature $\Sigma^{\lambda}_{\mathsf{S},\bullet}$. If P_1 and P_2 are correct abstraction prefix functions of G, then $P_1 = P_2$.

Definition 5.5 (λ-term-graph)**.** A *λ-term-graph* is a term graph $G = \langle V, lab, args, r\rangle$ over $\Sigma^{\lambda}_{\mathsf{S},\bullet}$ that has a correct abstraction-prefix function (which is not a part of G). The class of λ-term-graphs is $\mathcal{T}$.

Definition 5.6 (eager scope)**.** A λ-term-graph G is called *eager-scope* if together with its abstraction-prefix function it meets the condition in Def. 4.9. $\mathcal{T}_{\text{eag}}$ denotes the class of eager-scope graphs.

5.1 Correspondence between λ-ho- and λ-term-graphs

The correspondences between λ-ho-term-graphs and λ-term-graphs:

$$\mathcal{HT} : \mathcal{H} \to \mathcal{T} \qquad\qquad \mathcal{TH} : \mathcal{T} \to \mathcal{H}$$

are defined as follows: For obtaining $\mathcal{HT}(\mathcal{G})$ for a $\mathcal{G} \in \mathcal{H}$, insert scope-delimiters wherever the prefix decreases, as illustrated in Fig. 6. For obtaining $\mathcal{TH}(G)$ for a $G \in \mathcal{T}$, retain the abstraction-prefix function, and remove every delimiter vertex from G, thereby connecting its incoming edge with its outgoing edge. For formal definitions and well-definedness of $\mathcal{TH}$ and $\mathcal{HT}$, see [12].

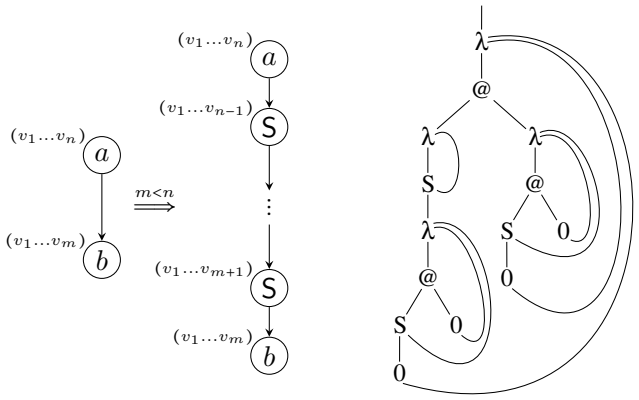


Figure 6. Left: definition of $\mathcal{HT}$ by inserting S-vertices, between edge-connected vertices of a λ-ho-term-graph. Right: interpretation $\mathcal{HT}(\mathcal{G})$ of the eager-scope λ-ho-term-graph $\mathcal{G}$ in Fig. 4.

Note that a λ-ho-term-graph may have multiple corresponding λ-term-graphs that differ only with respect to their 'degree' of S-sharing (the extent to which S-vertices occur shared). $\mathcal{HT}$ maps to a λ-term-graph with no sharing of S-vertices at all.

The proposition below guarantees the usefulness of the translation $\mathcal{HT}$ for implementing functional bisimulation on λ-ho-term-graphs. In particular, this is due to items (iii) and (iv). As formulated by item (i), $\mathcal{TH}$ is a retraction of $\mathcal{HT}$ (and $\mathcal{HT}$ a section of $\mathcal{TH}$). The converse is not the case, yet it holds up to S-sharing by item (ii). For the proof, we refer to our article [12].

Proposition 5.7 (correspondence with λ-ho-term-graphs)**.**

(i) $\mathcal{TH} \circ \mathcal{HT} = \mathrm{id}_{\mathcal{H}}$.
(ii) $(\mathcal{HT} \circ \mathcal{TH})(G) \rightrightarrows^{\mathsf{S}} G$ holds for all $G \in \mathcal{T}$.
(iii) $\mathcal{TH}$ and $\mathcal{HT}$ preserve and reflect functional bisimulation $\rightrightarrows$ and bisimulation $\leftrightarrows$ on $\mathcal{H}$ and $\mathcal{T}$.
(iv) $\mathcal{TH}$ and $\mathcal{HT}$ preserve and reflect the property eager-scope.
(v) $\mathcal{T}$ is closed under $\rightrightarrows^{\mathsf{S}}$, $\leftarrow^{\mathsf{S}}$, and $\leftrightarrows^{\mathsf{S}}$.
(vi) $\mathcal{HT}$ and $\mathcal{TH}$ induce isomorphisms between $\mathcal{H}$ and $\mathcal{T}/_{\leftrightarrows^{\mathsf{S}}}$.

5.2 Closedness of $\mathcal{T}$ under functional bisimulation

While preservation of $\rightrightarrows$ by $\mathcal{HT}$ is necessary for its implementation via $\rightrightarrows$ on $\mathcal{T}$, the practicality of the interpretation $\mathcal{HT}$ also depends on the closedness of $\mathcal{T}$ under $\rightrightarrows$. Namely, if the bisimulation collapse $G = \mathcal{HT}(\mathcal{G})\!\downarrow\!\downarrow$ of the interpretation of some $\mathcal{G} \in \mathcal{H}$ were not contained in $\mathcal{T}$, then the converse interpretation $\mathcal{TH}$ could not be applied to G in order to obtain the bisimulation collapse of $\mathcal{G}$.

A subclass $\mathcal{K}$ of the term graphs over a signature Σ is called *closed under functional bisimulation* if, for all term graphs G, G' over Σ, whenever $G \in \mathcal{K}$ and $G \rightrightarrows G'$, then also $G' \in \mathcal{K}$.

Note that for obtaining this property the use of variable backlinks, and backlinks for delimiter vertices is crucial (cf. Ex. 5.2).

Yet the class $\mathcal{T}$ is actually not closed under $\rightrightarrows$: See Fig. 7 at the top for a homomorphism from a non-eager-scope λ-term-graph to

a term graph over $\Sigma^{\lambda}_{\mathsf{S},\bullet}$ that is not a λ-term-graph (as suggested by the overlapping scopes). But the bisimulation collapse of an eager-scope version of this λ-term-graph is again a λ-term-graph (at the bottom). This motivates the following theorem, which is proved in the extended report of [12]. It justifies property (P2) with $\mathcal{T}_{\text{eag}}$ for $\mathcal{T}$.

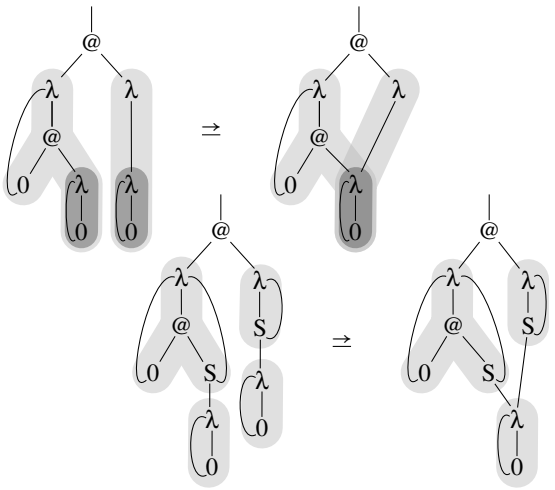


Figure 7. $\mathcal{T}$ is not closed under functional bisimulation, but $\mathcal{T}_{\text{eag}}$ is.

Theorem 5.8. *The class $\mathcal{T}_{\text{eag}}$ of eager-scope λ-term-graphs is closed under functional bisimulation $\rightleftarrows$.*

5.3 λ-term-graph semantics for $\lambda_{\mathsf{letrec}}$-terms

We will consider in fact two interpretations of $\lambda_{\mathsf{letrec}}$-terms as λ-term-graphs: first we define $[\![\cdot]\!]^{\mathsf{min}}_{\mathcal{T}}$ as the composition of $[\![\cdot]\!]_{\mathcal{H}}$ and $\mathcal{HT}$; then we define the semantics $[\![\cdot]\!]_{\mathcal{T}}$ with more fine-grained S-sharing, which is necessary for defining a readback with the property (P3).

By composing the interpretation $\mathcal{HT}$ of λ-ho-term-graphs as λ-term-graphs with the λ-ho-term-graph semantics $[\![\cdot]\!]_{\mathcal{H}}$, a semantics of $\lambda_{\mathsf{letrec}}$-terms as λ-term-graphs is obtained. There is, however, a more direct way to define this semantics: by using an adaptation of the translation rules $\mathcal{R}$ in Fig. 3, on which $[\![\cdot]\!]_{\mathcal{H}}$ is based. For this, let $\mathcal{R}_{\mathsf{S}}$ be the result of replacing the rule S in $\mathcal{R}$ by the version in Fig. 8. While applications of this variant of the S-rule also shorten the abstraction-prefix, they additionally produce a delimiter vertex.

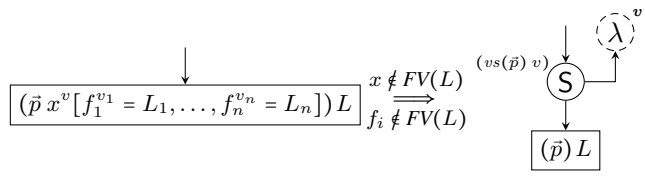


Figure 8. Delimiter-vertex producing version of the S-rule in Fig. 3

Here, at the end of the translation process, every loop on an indirection vertex with a prefix of length n is replaced by a chain of n S-vertices followed by a black hole vertex.Note that, while the system $\mathcal{R}_{\mathsf{S}}$ inherits all of the non-determinism of $\mathcal{R}$, the possible degrees of freedom have additional impact on the result, because now they also determine the precise degree of S-vertex sharing.

By analogous stipulations as in Def. 4.11 we define the conditions under which a λ-term-graph is called *eager-scope* $\mathcal{R}_{\mathsf{S}}$-generated, or $\mathcal{R}_{\mathsf{S}}$-generated *with minimal prefixes*, from a $\lambda_{\mathsf{letrec}}$-term. For these notions, statements entirely analogous to Prop. 4.12 hold.

Definition 5.9. The *semantics* $[\![\cdot]\!]^{\mathsf{min}}_{\mathcal{T}}$ for $\lambda_{\mathsf{letrec}}$-terms as λ-term-graphs is defined as $[\![\cdot]\!]^{\mathsf{min}}_{\mathcal{T}} : Ter(\lambda_{\mathsf{letrec}}) \to \mathcal{T}_{\text{eag}}$, $L \mapsto [\![L]\!]^{\mathsf{min}}_{\mathcal{T}} :=$ the

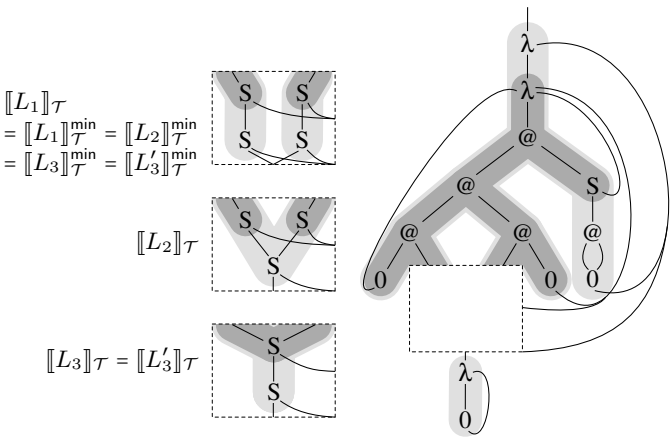


Figure 9. Translation of the $\lambda_{\mathsf{letrec}}$-terms from Ex. 5.14 with the semantics $[\![\cdot]\!]^{\mathsf{min}}_{\mathcal{T}}$ and $[\![\cdot]\!]_{\mathcal{T}}$. For legibility some backlinks are merged.

λ-term-graph that is $\mathcal{R}_{\mathsf{S}}$-generated with minimal prefixes from a garbage-free version L' of L.

For an example, see Ex. 5.14 below. In $[\![\cdot]\!]^{\mathsf{min}}_{\mathcal{T}}$, 'min' also indicates that λ-term-graphs obtained via this semantics exhibit minimal (in fact no) sharing (two or more incoming edges) of S-vertices. This is substantiated by the next proposition, in the light of the fact that $\mathcal{HT}$ does not create any shared S-vertices.

Proposition 5.10. $[\![\cdot]\!]^{\mathsf{min}}_{\mathcal{T}} = \mathcal{HT} \circ [\![\cdot]\!]_{\mathcal{H}}$.

Hence $[\![\cdot]\!]^{\mathsf{min}}_{\mathcal{T}}$ only yields λ-term-graphs without sharing of S-vertices, and therefore its image cannot be all of $\mathcal{T}_{\text{eag}}$. As a consequence, we cannot hope to define a readback function rb with respect to $[\![\cdot]\!]^{\mathsf{min}}_{\mathcal{T}}$ that has the desired property (P3), because that requires that the image of the semantics is $\mathcal{T}_{\text{eag}}$ in its entirety.

Therefore we modify the definition of $[\![\cdot]\!]^{\mathsf{min}}_{\mathcal{T}}$ to obtain a λ-term-graph semantics $[\![\cdot]\!]_{\mathcal{T}}$ with image $\mathrm{im}([\![\cdot]\!]_{\mathcal{T}}) = \mathcal{T}_{\text{eag}}$. This is achieved by letting the let-binding-structure of the $\lambda_{\mathsf{letrec}}$-term influence the degree of S-sharing as much as possible, while staying eager-scope.

We say that a λ-ho-term-graph $\mathcal{G}$ is *eager-scope $\mathcal{R}$-generated with maximal prefixes* from a $\lambda_{\mathsf{letrec}}$-term L if $\mathcal{G}$ is $\mathcal{R}$-generated from L by a translation process in which in applications of the let-rule the prefixes are chosen maximally, but so that the eager-scope property of the process is not compromised. It can be shown that this condition fixes the prefix lengths per application of the let-rule.

Definition 5.11. The *semantics* $[\![\cdot]\!]_{\mathcal{T}}$ for $\lambda_{\mathsf{letrec}}$-terms as λ-term-graphs is defined as $[\![\cdot]\!]_{\mathcal{T}} : Ter(\lambda_{\mathsf{letrec}}) \to \mathcal{T}_{\text{eag}}$, $L \mapsto [\![L]\!]_{\mathcal{T}} :=$ the λ-term-graph that is eager-scope $\mathcal{R}_{\mathsf{S}}$-generated with maximal prefixes from a garbage-free version L' of L.

Proposition 5.12. $[\![L]\!]^{\mathsf{min}}_{\mathcal{T}} \rightleftarrows^{\mathsf{S}} [\![L]\!]_{\mathcal{T}}$ holds for all $\lambda_{\mathsf{letrec}}$-terms L.

Now due to this, and due to Prop. 5.7, (iii), the statement of Thm. 4.15 can be transferred to $\mathcal{T}$, yielding property (P1) for $[\![\cdot]\!]_{\mathcal{T}}$.

Theorem 5.13. *For all $\lambda_{\mathsf{letrec}}$-terms L_1 and L_2 the following holds: $[\![L_1]\!]_{\lambda^\infty} = [\![L_2]\!]_{\lambda^\infty}$ if and only if $[\![L_1]\!]_{\mathcal{T}} \leftrightarrows [\![L_2]\!]_{\mathcal{T}}$.*

Example 5.14. Consider the following four $\lambda_{\mathsf{letrec}}$-terms:

$L_1 = \mathsf{let}\ I = \lambda z.\, z\ \mathsf{in}\ \lambda x.\, \lambda y.\, \mathsf{let}\ f = x\ \mathsf{in}\ ((y\, I)\, (I\, y))\, (f\, f)$

$L_2 = \lambda x.\, \mathsf{let}\ I = \lambda z.\, z\ \mathsf{in}\ \lambda y.\, \mathsf{let}\ f = x\ \mathsf{in}\ ((y\, I)\, (I\, y))\, (f\, f)$

$L_3 = \lambda x.\, \lambda y.\, \mathsf{let}\ I = \lambda z.\, z,\ f = x\ \mathsf{in}\ ((y\, I)\, (I\, y))\, (f\, f)$

$L_3' = \lambda x.\, \mathsf{let}\ I = \lambda z.\, z\ \mathsf{in}\ \lambda y.\, \mathsf{let}\ f = x,\ g = I\ \mathsf{in}\ ((y\, g)\, (g\, y))\, (f\, f)$

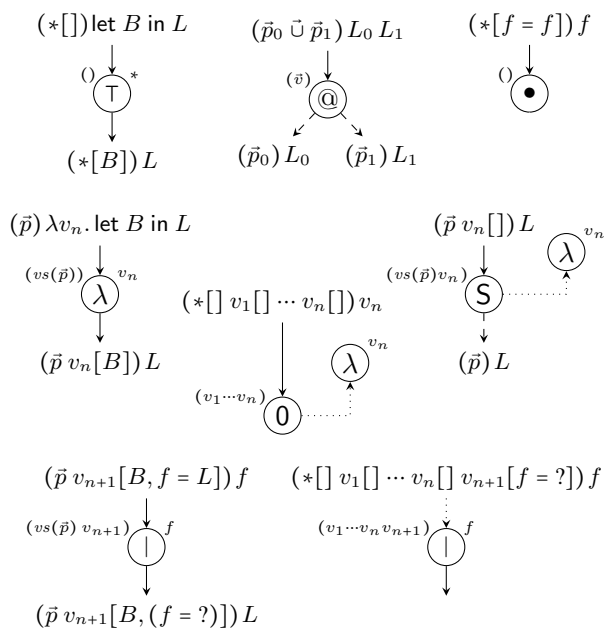


Figure 10. Readback synthesis rules for computing a representing $\lambda_{\mathsf{letrec}}$-term from a λ-term-graph. The rules for $\top$- and λ-vertices have variants for B is empty. For explanations, see Def. 6.1, (Rb-5).

The three possible fillings of the dashed area in Fig. 9 depict the translations $[\![L_1]\!]_{\mathcal{T}}$, $[\![L_2]\!]_{\mathcal{T}}$, and $[\![L_3]\!]_{\mathcal{T}} = [\![L_3']\!]_{\mathcal{T}}$. The translations of the four terms with $[\![\cdot]\!]^{\mathsf{min}}$ are identical:
$[\![L_1]\!]^{\mathsf{min}}_{\mathcal{T}} = [\![L_2]\!]^{\mathsf{min}}_{\mathcal{T}} = [\![L_3]\!]^{\mathsf{min}}_{\mathcal{T}} = [\![L_3']\!]^{\mathsf{min}}_{\mathcal{T}} = [\![L_1]\!]_{\mathcal{T}}$.

6. Readback of λ-term-graphs

In this section we describe how from a given λ-term-graph G a $\lambda_{\mathsf{letrec}}$-term L that represents G (i.e. for which $[\![L]\!]_{\mathcal{T}} = G$ holds) can be 'read back'. For this purpose we define a process based on term synthesis rules. It defines a readback function from λ-term-graphs to $\lambda_{\mathsf{letrec}}$-terms. We illustrate this process by an example, formulate its most important properties, and sketch the proof of (P3).

The idea underlying the definition of the readback procedure is the following: For a given λ-term-graph G, a spanning tree T for G (augmented with a dedicated root node) is constructed that severs cycles of G at (some) recursive bindings, and at variable and S-backlinks. Now the spanning tree T facilitates an inductive bottom-up (from the leafs upwards) synthesis process along T, which labels the edges of G (except for variable backlinks) with prefixed $\lambda_{\mathsf{letrec}}$-terms. For this process we use local rules (see Fig. 10) that synthesise labels for incoming edges of a vertex from the labels of its outgoing edges. Eventually the readback of G is obtained as the label for the edge that singles out the root of term graph.

The design of the readback rules is based on a decision about where let-bindings are placed in the synthesised term. Namely there exists some freedom for these placements, as certain kind of shifts of let-expressions (let-floating steps [14]) preserve the λ-term-graph interpretation. Here, let-bindings will always be declared in a let-expression that is placed as high up in the term as possible: a binding arising from the term synthesised for a shared vertex w is placed in a let-expression that is created at the enclosing λ-abstraction of w (the leftmost vertex in the abstraction-prefix $P(w)$ of w).

Definition 6.1 (readback of λ-term-graphs)**.** Let $G \in \mathcal{T}$ be a λ-term-graph. The process of computing the readback of G (a $\lambda_{\mathsf{letrec}}$-term) consists of the following five steps, starting on G:

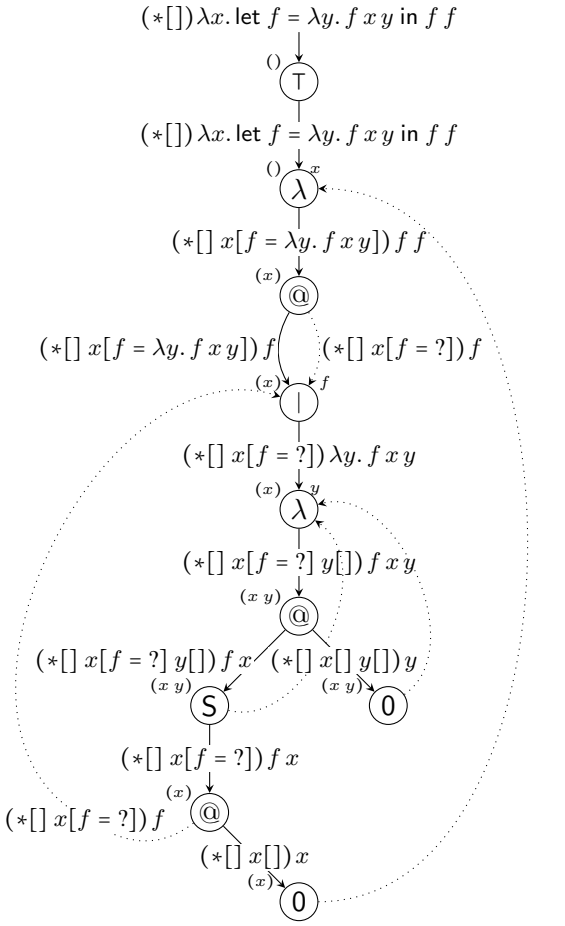


Figure 11. Example of the readback synthesis from a λ-term-graph.

(Rb-1) Determine the abstraction-prefix function P for G by performing a traversal over G, and associate with every vertex w of G its abstraction-prefix $P(w)$.

(Rb-2) Add a new vertex on top with label $\top$, arity 1, and empty abstraction prefix. Let G' be the resulting term graph, and P' its abstraction-prefix function.

(Rb-3) Introduce indirection vertices to organise sharing: For every vertex w of G' with two or more incoming non-variable-backlink edges, add an indirection vertex w_0, redirect the incoming edges of w that are not variable backlinks to w_0, and direct the outgoing edge from w_0 to w. In the resulting term graph G'' only indirection vertices are shared[3]; their names will be used. Extend P' to an abstraction-prefix function P'' for G'' so that every indirection vertex w_0 gets the prefix of its successor w.

(Rb-4) Construct a spanning tree T'' of G'' by using a depth-first search (DFS) on G''. Note that all variable backlinks and S-backlinks, and some of the recursive back-bindings, of G'', are not contained in T'', because they are back-edges of the DFS.

(Rb-5) Apply the readback synthesis rules from Fig. 10 to G'' with respect to T''. By this a complete labelling of the edges of G'' by prefixed $\lambda_{\mathsf{letrec}}$-terms is constructed. The rules define how the labelling for an incoming edge (on top) of a vertex w is synthesised under the assumption of an already determined labelling of an outgoing edge of (and below) w. If the outgoing

[3] Incoming variable backlinks are not counted as sharing here.

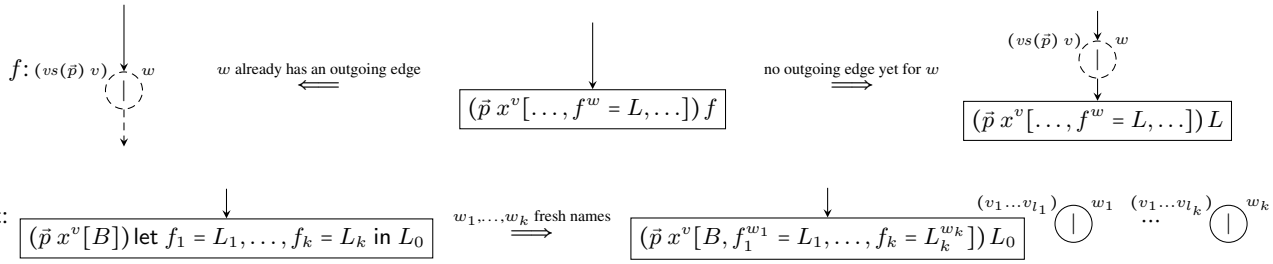


Figure 12. Modification of (two of) the translation rules in Fig. 3 for a variant definition of the λ-term-graph interpretation of $\lambda_{\mathsf{letrec}}$-terms. Here the translation of a let-expression does not directly spawn translations for the binding equations, but the in-part has to be translated first.

edge in the rule does not carry a label, then the labelling of the incoming edge can happen regardless. Note that in these rules:

- full line (dotted line) edges indicate spanning tree (non-spanning tree) edges, broken line edges either of these sorts;
- abstraction prefixes of vertices are crucial for the 0-vertex, and the second indirection vertex rule, where the prefixes in the synthesised terms are created; in the other rules the prefix of the assumed term is used; for indicating a correspondence between a term's and a vertex's abstraction prefix we denote by $v(\vec{p})$ the word of vertices occurring in a term's prefix $\vec{p}$;
- the rule for indirection vertices with incoming non-spanning tree edge introduces an unfinished binding $f = ?$ for f; unfinished bindings are completed in the course of the process;
- the @-vertex rule applies only if $v(\vec{p}_0) = v(\vec{p}_1)$; the operation $\vec{\cup}$ used in the synthesised term's prefix builds the union per prefix variable of the pertaining bindings; if the prefixed terms $(\vec{p}_0)\ L_0$ and $(\vec{p}_1)\ L_1$ assumed in this rule contain a yet unfinished binding equation $f = ?$ and a completed equation $f = P$ at a λ-variable z, then the synthesised term contains the completed binding $f = P$ for f at z;
- not depicted in Fig. 10 are variants of $\top$- and λ-vertices rules for the cases with empty B: then no let-binding is introduced in the synthesised term, but the term from the in-part is used.

If this process yields the label $(*[])L$ for the (root-)edge pointing to the new top vertex of G'', where L is a $\lambda_{\mathsf{letrec}}$-term, then we call L the *readback of* G.

Note that firing of the rules in step (Rb-5) of the readback process proceeds in bottom-up direction in the spanning tree, starting from the back-edges, with some room for parallelism concerning work in different subtrees. Furthermore observe that on all directed edges e (spanning tree edges or back edges) the rule applied to derive the edge label is uniquely determined by (is tied to) the label of the target vertex v of e, with the single exception of v being an indirection vertex. In that case one of the two indirection vertex rules applies, depending on whether e is a spanning-tree edge or a back-edge.

Proposition 6.2. Let G be a λ-term-graph. The process described in Def. 6.1 produces a complete edge labelling of the (modified) term graph, with label $(*[])L$ for the topmost edge, where L is a $\lambda_{\mathsf{letrec}}$-term. Hence it yields L as the readback of G. Thus Def. 6.1 defines a function $\mathsf{rb} : \mathcal{T} \to Ter(\lambda_{\mathsf{letrec}})$, the *readback function*.

Example 6.3. See Fig. 11 for the illustration of the synthesis of the readback from an example λ-term-graph. Full line edges are in the spanning tree, dotted line edges are not. Note that at the top vertex, no empty let-binding is created since the variant of the $\top$-vertex rule for empty binding groups is applied.

The following theorem validates property (P3), with $\mathcal{T}_{\mathsf{eag}}$ for $\mathcal{T}$.

Theorem 6.4. *For all* $G \in \mathcal{T}_{\mathsf{eag}}$*:* $(\llbracket\cdot\rrbracket_{\mathcal{T}} \circ \mathsf{rb})(G) = \llbracket\mathsf{rb}(G)\rrbracket_{\mathcal{T}} \simeq G$*, i.e.,* rb *is a right-inverse of* $\llbracket\cdot\rrbracket_{\mathcal{T}}$*, and* $\llbracket\cdot\rrbracket_{\mathcal{T}}$ *a left-inverse of* rb*, up to* $\simeq$*. Hence* rb *is injective, and* $\llbracket\cdot\rrbracket_{\mathcal{T}}$ *is surjective, thus* $im(\llbracket\cdot\rrbracket_{\mathcal{T}}) = \mathcal{T}_{\mathsf{eag}}$*.*

Sketch of Proof. Graph translation steps can be linked with corresponding readback steps in order to establish that the former roughly reverse the latter. Roughly, because e.g. reversing a λ-readback step necessitates both a λ- and a let-translation step. (For illustrations of the stepwise reversal of readback steps through translation steps we refer to two figures in the extended version of this paper [15].) However, this correspondence holds only for a modification of the translation rules $\mathcal{R}_S$ from Fig. 3, Fig. 8 where the rules let (for let-expressions) and f (for occurrences of recursion variables) are replaced by the locally-operating versions in Fig. 12, and a rule for creating a top vertex is added. Now the translation of a let-expression does no longer directly spawn translations of the defined recursive bindings, but the bindings will only be translated later once their calls have been reached during the translation process of the in-part, or of the definitions of other already translated bindings. Note that in the let-rule in Fig. 12 function bindings are associated with the rightmost variable in the prefix, which corresponds to choosing $l_i = n$ in the let-rule in Fig. 3. While such a stipulation does not guarantee the eager-scope translation of every term, it actually does so for all $\lambda_{\mathsf{letrec}}$-terms that are obtained by the readback (on these terms the translation so defined coincides with $\llbracket\cdot\rrbracket_{\mathcal{T}}$ from Def. 4.13).

The proof uses induction on access paths, and an invariant that relates the eager-scope property localised for a vertex v with the applicability of the S-rule to the readback term synthesised at v. □

7. Complexity analysis

Here we report on a complexity analysis for the individual operations from the previous sections, for the used standard algorithms, and overall, for compactification and unfolding equivalence.

In the lemma below, (ii) and (v) justify the property (P4) of our methods. Items (iii) and (iv) detail the complexity of standard methods when used for computing bisimulation collapse and bisimilarity of λ-term-graphs. Note that first-order term graphs can be modelled by deterministic process graphs, and hence by DFAs. Therefore bisimilarity of term graphs can be computed via language equivalence of corresponding DFAs [19] (in time $O(n\alpha(n))$ [25], where α is the quasi-constant *inverse Ackermann function*), and bisimulation collapse via state minimisation of DFAs (in time $O(n \log n)$) [18].

Lemma 7.1. *(i)* $\mathrm{size}(\llbracket L\rrbracket_{\mathcal{T}}) \in O(|L|^2)$ *for* $L \in Ter(\lambda_{\mathsf{letrec}})$*.*
(ii) Translating $L \in Ter(\lambda_{\mathsf{letrec}})$ *into* $\llbracket L\rrbracket_{\mathcal{T}} \in \mathcal{T}$ *takes time* $O(|L|^2)$*.*
(iii) Collapsing $G \in \mathcal{T}$ *to* $G{\downarrow}$ *is in* $O(\mathrm{size}(G) \log \mathrm{size}(G))$*.*
(iv) Deciding bisimilarity of $G_1, G_2 \in \mathcal{T}$ *requires time* $O(n\alpha(n))$ *for* $n = \max\{\mathrm{size}(G_1), \mathrm{size}(G_2)\}$*.*
(v) Computing the readback $\mathsf{rb}(G)$ *for a given* $G \in \mathcal{T}$ *requires time* $O(n \log n)$*, for* $n = \mathrm{size}(G)$*.*

Based on this lemma, and on further considerations, we obtain the following complexity statements for our methods.

Theorem 7.2. *(i) The computation, for a $\lambda_{\mathsf{letrec}}$-term L with $|L| = n$, of a maximally compactified form $(\mathsf{rb} \circ |\!\downarrow \circ [\![\cdot]\!]_{\mathcal{T}})(L)$ requires time $O(n^2 \log n)$. By using an S-unsharing operation $\mathrm{unsh}_{\mathsf{S}}$, a (typically smaller) $\lambda_{\mathsf{letrec}}$-term $(\mathsf{rb} \circ \mathrm{unsh}_{\mathsf{S}} \circ |\!\downarrow \circ [\![\cdot]\!]_{\mathcal{T}})(L)$ of size $O(n \log n)$ can be obtained, with the same time complexity.*
(ii) The decision of whether $\lambda_{\mathsf{letrec}}$-terms L_1 and L_2 are unfolding-equivalent requires time $O(n^2 \alpha(n))$ for $n = \max\{|L_1|, |L_2|\}$.

8. Implementation

We have implemented our methods in Haskell using the Utrecht University Attribute Grammar System. The implementation is available at `http://hackage.haskell.org/package/maxsharing/`. The output produced for the examples in this paper, and explanations can be found in the appendix of the extended version [15] of this paper.

9. Modifications, extensions and applications

We have described an adaptation of the bisimulation proof method for $\lambda_{\mathsf{letrec}}$-terms. Recognising unfolding equivalence and increasing sharing are reduced to problems involving first-order term graphs. The principal idea is to use the nested scope structure of higher-order terms for an interpretation by term graphs with scope delimiters.

We conclude by describing easy modifications, rather direct extensions, and finally, promising areas of application for our methods.

9.1 Modifications

Implicit sharing of λ-variables. Multiple occurrences of the same λ-variable in a $\lambda_{\mathsf{letrec}}$-term L are not shared (represented by a shared variable vertex) in the graph interpretation $[\![L]\!]_{\mathcal{H}}$. Consequently, our method compactifies the term $\lambda x.\, x\, x$ into $\lambda x.\, \mathsf{let}\ f = x\ \mathsf{in}\ f\, f$. Such explicit sharing of variables is excessive for many applications. It can be remedied easily, namely by unsharing variable vertices before applying the readback, or by preventing the readback from introducing let-bindings when only a variable vertex is shared.

Avoiding aliases produced by the readback. The readback function in Section 6 is sensitive to the degree of sharing of S-vertices in the given λ-term-graph: it maps two λ-term-graphs that only differ in what concerns sharing of S-vertices to different $\lambda_{\mathsf{letrec}}$-terms. Typically, for λ-term-graphs with maximal sharing of S-vertices this can produce let-bindings that are just 'aliases', such as g is alias for I in L_3' from Ex. 5.14. This can be avoided in two ways: by slightly adapting the readback function, or by performing maximal unsharing of S-vertices before applying the readback as defined.

Preventing disadvantageous sharing. Introducing sharing at compile-time can cause 'space leaks', i.e. a needlessly high memory footprint, at run-time, because 'a large data structure becomes shared [...], and therefore its space which before was reclaimed by garbage collection now cannot be reclaimed until its last reference is used' [9]. For this reason, realisations of CSE [6] restrict the locally operating rewrite rules employed for introducing sharing by suitable conditions that account for the type of potentially shared subexpressions, and their strictness in the program. For our global method of introducing sharing via the bisimulation collapse, a different approach is needed.

Here the bisimulation collapse can be restricted so that sharing is not introduced at vertices that should not be shared. More precisely, it can be prevented that any unshared vertex (in-degree one) from a pre-determined set of 'sharing-unfit' vertices would have a shared vertex (in-degree greater than one) as its image in the bisimulation collapse. This can be achieved by modifying the graph interpretation $[\![\cdot]\!]_{\mathcal{T}}$. Any set of sharing-unfit positions in L gives rise to a set of sharing-unfit vertices in $[\![L]\!]_{\mathcal{T}}$. In the modification of $[\![L]\!]_{\mathcal{T}}$, special back-links are added from every sharing-unfit vertex with in-degree one to its immediate successor. These back-links prevent that such a sharing-unfit vertex v can collapse with another vertex v' without that also the predecessors of v and v' would collapse as well.

A more general notion of readback. Condition (P3) is rather rigorous in that it imposes a sharing structure on $\lambda_{\mathsf{letrec}}$ that is specific to λ-term-graphs (degrees of S-sharing). For a weaker version of (P3) with $\leftrightarrows^{\mathsf{S}}$ in place of isomorphism, a readback does not have to be injective, and, independently of how much S-sharing a translation into λ-term-graphs introduces, a readback function always exists.

9.2 Extensions

Full functional languages. In order to support programming languages that are based on $\lambda_{\mathsf{letrec}}$ like Haskell, additional language constructs need to be supported. Such languages can typically be desugared into a core language, which comprises only a small subset of language constructs such as constructors, case statements, and primitives. These constructs can be represented in an extension of $\lambda_{\mathsf{letrec}}$ by additional function symbols. In conjunction with a desugarer our methods are applicable to full programming languages.

Other programming languages, and calculi with binding constructs. Most programming languages feature constructs for grouping definitions that are similar to letrec. We therefore expect that our methods can be adapted to many imperative languages in particular, and may turn out to be fruitful for optimising compilers. Our methods for achieving maximal sharing certainly generalise to theoretical frameworks, and calculi with binding constructs, such as the π-calculus [23], and higher-order rewrite systems (e.g. CRSs and HRSs, [30]) as used here for the formalisation of $\lambda_{\mathsf{letrec}}$.

Fully-lazy lambda-lifting. There is a close connection between our methods and fully-lazy lambda-lifting [20, 28]. In particular, the required-variable and scope analysis of a $\lambda_{\mathsf{letrec}}$-term L on which the λ-term-graph-translation $[\![L]\!]_{\mathcal{T}}$ is based is closely analogous to the one needed for extracting from L the supercombinators in the result $\hat{L}$ of fully-lazy lambda-lifting L. Moreover, the fully-lazy lambda-lifting transformation can even be implemented in a natural way on the basis of our methods. Namely as the composition $\mathsf{rb}_{LL} \circ [\![\cdot]\!]_{\mathcal{T}}$ of the translation $[\![\cdot]\!]_{\mathcal{T}}$ into λ-term-graphs, where rb_{LL} is a variant readback function that, for a given λ-term-graph, synthesises the system $\hat{L}$ of supercombinators, instead of the $\lambda_{\mathsf{letrec}}$-term $\mathsf{rb}(L)$.

Maximal sharing on supercombinator translations of $\lambda_{\mathsf{letrec}}$-terms. $\lambda_{\mathsf{letrec}}$-terms L correspond to supercombinator systems $\hat{L}$, the result of fully-lazy lambda-lifting L: the combinators in $\hat{L}$ correspond to 'extended scopes' [11] (or 'skeletons' [2]) in L, and supercombinator reduction steps on $\hat{L}$ correspond to weak β-reduction steps L. In the case of λ-calculus this has been established by Balabonski [2]. Via this correspondence the maximal-sharing method for $\lambda_{\mathsf{letrec}}$-terms can be lifted to obtain a maximal-sharing method systems of supercombinators obtained by fully-lazy lambda-lifting.

Non-eager scope-closure strategies. We focused on eager-scope translations, because they facilitate maximal sharing, and guarantee that interpretations of unfolding-equivalent $\lambda_{\mathsf{letrec}}$-terms are bisimilar. Yet every scope-closure strategy [11] induces a translation and its own notion of maximal sharing. For adapting our maximal sharing method it is necessary to modify the translation into first-order term graphs in such a way that the image class obtained is closed under homomorphism ($\mathcal{T}$ is not closed under $\twoheadrightarrow$, unlike its subclass $\mathcal{T}_{\mathrm{eag}}$). This can be achieved by using delimiter vertices also below variable vertices to close scopes that are still open [12, report].

Weaker notions of sharing. The presented methods deal with sharing as expressed by letrec that is horizontal, vertical, or twisted [4]. By contrast, the construct μ [4, 13] expresses only vertical, and the non-recursive let only horizontal, sharing. By restricting bisimulation, our methods can be adapted to the λ-calculus with μ, or with let.

Nested term graphs. The nested scope structure of a $\lambda_{\mathsf{letrec}}$-term can also be represented by a nested structure of term graphs. The representation of a $\lambda_{\mathsf{letrec}}$-term as a 'nested term graph' [16] starts with an ordinary term graph in which some of the vertices are labelled by 'nested' symbols that designate outermost bindings together with their scope. Any such vertex is additionally associated with a usual term graph that specifies the subterm context describing the scope, where any inner scopes are again expressed by nested symbols. The association between nested symbols and their term graph specifications is required to be tree-like. The implementation result developed here can be generalised to show that nested term graphs can be implemented faithfully by first-order term graphs [16].

9.3 Applications

Maximal sharing at run-time. Maximal sharing can be applied repeatedly at run-time in order to regain a maximally shared form, thereby speeding up evaluation. This is reminiscent of 'collapsed tree rewriting' [29] for evaluating first-order term graphs represented as maximally shared dags. Since the state of a program in the memory at run-time is typically represented as a supercombinator graph, compactification by bisimulation collapse can take place directly on that graph (see Sec. 9.2), no translation is needed. Compactification can be coupled with garbage collection as bisimulation collapse subsumes some of the work required for a mark and sweep garbage collector. However, a compromise needs to be found between the costs for the optimisation and the gained efficiency.

Additional prevention of disadvantageous sharing. While static analysis methods for preventing sharing that may be disadvantageous at run-time can be adapted from CSE to the maximal-sharing method (see Sec. 9.1), this has yet to be investigated for binding-time analysis [27] and a sharing analysis of partial applications [10].

Code improvement. In programming it is generally desirable to avoid duplication of code. As an extension of CSE, our method is able to detect code duplication. The bisimulation collapse of the term graph interpretation of a program can, together with the readback, provide guidance on how code can be refactored into a more compact form. This application requires some fine-tuning to avoid excessive behaviour like the explicit sharing of variable occurrences (see Sec. 9.1). Yet for this only lightweight additional machinery is needed, such as size constraints or annotations to restrict the bisimulation collapse.

Function equivalence. Recognising whether two programs implement the same function is undecidable. Still, this problem is tackled by proof assistants, and by automated theorem provers used in type-checkers of compilers for dependently-typed programming languages such as Agda. For such systems co-inductive proofs are more difficult to find than inductive ones, and require more effort by the user. Our method for deciding unfolding equivalence could help to develop new approaches to finding co-inductive proofs.

Acknowledgment We want to thank Vincent van Oostrom for extensive feedback on a draft, Doaitse Swierstra and Dimitri Hendriks for helpful comments, and Jeroen Keiren for a suggestion concerning restricting the bisimulation collapse. We also thank the anonymous reviewers for their comments, and a number of stimulating questions.

References

[1] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998.

[2] T. Balabonski. A unified approach to fully lazy sharing. In *Proceedings of POPL '12*, pages 469–480, New York, NY, USA, 2012. ACM.

[3] R. S. Bird and R. Patterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[4] S. Blom. *Term Graph Rewriting – Syntax and Semantics*. PhD thesis, Vrije Universiteit Amsterdam, 2001.

[5] M. v. d. Brand and P. Klint. ATERMs for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, 49(1):55–64, 2007.

[6] O. Chitil. Common Subexpressions Are Uncommon in Lazy Functional Languages. In *Selected Papers from the 9th International Workshop IFL (IFL '97)*, pages 53–71, London, UK, UK, 1998. Springer-Verlag.

[7] O. Danvy and U. P. Schultz. Lambda-lifting in quadratic time. *Journal of Functional and Logic Programming*, 2004, 2004.

[8] N. G. de Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Applic. to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[9] A. L. de Medeiros Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.

[10] B. Goldberg and P. Hudak. Detecting Sharing of Partial Applications in Functional Programs. Technical Report YALEU/DCS/RR-526, Department of Computer Science, Yale University, March 1987.

[11] C. Grabmayer and J. Rochel. Expressibility in the Lambda Calculus with Letrec. Technical report, arXiv, August 2012. arXiv:1208.2383.

[12] C. Grabmayer and J. Rochel. Term Graph Representations for Cyclic Lambda Terms. In *Proceedings of TERMGRAPH 2013*, number 110 in EPTCS, 2013. For an extended report see: arXiv:1308.1034.

[13] C. Grabmayer and J. Rochel. Expressibility in the Lambda Calculus with μ. In *Proceedings of RTA 2013*, 2013. Report: arXiv:1304.6284.

[14] C. Grabmayer and J. Rochel. Confluent Let-Floating. In *Proceedings of IWC 2013 (2nd International Workshop on Confluence)*, 2013.

[15] C. Grabmayer and J. Rochel. Maximal Sharing in the Lambda Calculus with letrec. Technical report, 2014. arXiv:1401.1460.

[16] C. Grabmayer and V. van Oostrom. Nested Term Graphs. Technical report, arXiv, May 2014. arXiv:1405.6380.

[17] D. Hendriks and V. van Oostrom. λ. In F. Baader, editor, *Proceedings CADE-19*, volume 2741 of *LNAI*, pages 136–150. Springer, 2003.

[18] J. Hopcroft. An $n \log n$ Algorithm for Minimizing States in a Finite Automata. Technical report, Stanford University, CA, USA, 1971.

[19] J. Hopcroft and R. Karp. A Linear Algorithm for Testing Equivalence of Finite Automata. Technical report, Cornell University, 1971.

[20] R. Hughes. Supercombinators: A new implementation method for applicative languages. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 1–10, 1982.

[21] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *FPCA*, pages 190–203, 1985.

[22] J. Ketema and J. G. Simonsen. Infinitary Combinatory Reduction Systems. *Information and Computation*, 209(6):893 – 926, 2011.

[23] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[24] M. T. Morazán and U. P. Schultz. Optimal lambda lifting in quadratic time. In *Workshop IAFL 2007*, number 5083 in LNCS. Springer, 2008.

[25] D. A. Norton. Algorithms for Testing Equivalence of Finite Automata. Master's thesis, Dept. of Computer Science, Rochester Institute of Technology, 2009. `https://ritdml.rit.edu/handle/1850/8712`.

[26] V. v. Oostrom, K.-J. van de Looij, and M. Zwitserlood. Lambdascope. Extended Abstract, Workshop ALPS, Kyoto, April 10th 2004, 2004.

[27] J. Palsberg and M. Schwartzbach. Binding-time analysis: abstract interpretation versus type inference. In *Int. Conf. on Computer Languages, 1994*, pages 289–298, 1994.

[28] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., 1987.

[29] D. Plump. *Evaluation of Functional Expressions by Hypergraph Rewriting*. PhD thesis, Universität Bremen, 1993.

[30] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[31] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, University of Oxford, 1971.