Confluent Let-Floating

Clemens Grabmayer[1] and Jan Rochel[2]

[1] Department of Philosophy, Utrecht University
clemens@phil.uu.nl
[2] Department of Computing Sciences, Utrecht University
jan@rochel.info

Abstract

We develop a rewrite analysis for floating (moving) let-bindings in expressions of $\boldsymbol{\lambda}_{\mathsf{letrec}}$, the λ-calculus with the construct letrec that is denoted by let (as in the programming language Haskell). In particular we consider a HRS (higher-order rewrite system) for let-lifting, which moves let-bindings upward, and another HRS for let-sinking, which moves let-bindings downward. We show confluence and termination of the let-lifting and let-sinking rewrite systems, yielding the existence of unique normal forms. Our confluence proofs use a critical pair analysis and the critical pair theorem to establish local confluence, and the termination of these systems to obtain confluence by applying Newman's Lemma.

Let-floating is an operation employed by transformations that simplify and optimize program code as part of compilers of functional languages. For example the lambda-lifting transformation of functional programs into supercombinators contains a step called 'let-floating' [4, 15.5.4] or 'block-floating' [1], in which let-bindings are floated out (upward, we call it 'let-lifting'). Lambda-lifting transforms a let-block-structured program into a set of recursive equations whose right-hand sides are supercombinators. This transformation has an inverse called lambda-dropping [1], which contains the step 'block-sinking' in which let-bindings are floated in (downward, we call it 'let-sinking'). The use of let-floating operations in either direction for optimizing and fine-tuning the execution behavior of compiled functional programs has been studied in [8].

As a more general concept, let-floating acts on expressions of $\boldsymbol{\lambda}_{\mathsf{letrec}}$, the λ-calculus with the construct letrec for formulating recursion and explicit substitution. We denote letrec as let like in the programming language Haskell (no confusion should arise with the non-recursive explicit-substitution construct let), but keep the symbol $\boldsymbol{\lambda}_{\mathsf{letrec}}$. In our terminology, 'floating' stands for movements in either direction, whereas 'lifting' and 'sinking' indicate upward and downward shifts in the syntax tree, respectively. Let-floating manipulates the structure of let-bindings in $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressions, but preserves the unfolding semantics of the expressions (the denoted infinite λ-terms). A let-binding-group B can be lifted up toward the innermost λ-abstraction that has a free variable occurrence in B. A group of n interdependent let-bindings $\vec{f} = \vec{F}(\vec{f})$ with $\vec{f} = \langle f_1, \ldots, f_n \rangle$ can be sunk until an applicative term is encountered where both in its function subterm and in its argument subterm some recursion variable f_i with $i \in \{1, \ldots, n\}$ occurs.

Our interest in let-floating stems from an investigation of the relationship between $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressions and term graph representations for cyclic λ-terms [3]. Translations of $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressions into representing term graphs typically ignore the precise positioning of the let-bindings, and instead extract the cyclic structure of the term. Therefore such translations map $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressions that are related by let-floating to the same term graph. For the definition of (left-)inverses of such translations, it is desirable to obtain natural representatives of let-floating equivalence classes by restricting the direction of let-floating operations to upward or downward.

We develop a rewrite analysis of let-floating. When decomposed into locally applicable rewrite steps on $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expression, let-floating operations typically move let-bindings upward or downward over applications and abstractions, or merge different let-binding groups, given that such steps do not interfere with the structure of the λ-bindings. We formalize $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressions as higher-order rewriting system (HRS) terms [10], and define two HRSs that describe different kinds of let-floating transformations as rewrite systems: let-lifting for moving let-bindings upward, and let-sinking for moving them downward. In both cases let-bindings are split whenever necessary for moves, and merged whenever possible. We show confluence and termination of the let-lifting and let-sinking rewrite systems, and by that, unique normalization.

1 Let-lifting

We formulate expressions in (untyped) $\boldsymbol{\lambda}_{\mathsf{letrec}}$ as HRS-terms [10] over the signature $\{\mathsf{abs},\mathsf{app}\} \cup \{\mathsf{let}_n\mathsf{\_in} \mid n \in \mathbb{N}\}$, where $\mathsf{abs} : (\mathsf{trm} \to \mathsf{trm}) \to \mathsf{trm}$, $\mathsf{app} : \mathsf{trm} \to \mathsf{trm} \to \mathsf{trm}$, and for all $n \in \mathbb{N}$, $\mathsf{let}_n\mathsf{\_in} : (\mathsf{trm}^n \to \mathsf{trm}^{n+1}) \to \mathsf{trm}$ over the base type trm. As an example, consider the $\boldsymbol{\lambda}_{\mathsf{letrec}}$-term:

$$\lambda x.\, \textbf{let}\ f = g,\ g = x\ \textbf{in}\ f\, x \qquad\qquad \mathsf{abs}(x.\, \mathsf{let}_2\mathsf{\_in}\,(f g.\, (g, x, \mathsf{app}(f, x))))$$

in familiar (first-order) notation and in a formulation as HRS-term. Here the index 2 in the symbol $\mathsf{let}_2\mathsf{\_in}$ indicates the number of bindings in the binding group of the let-expression. While building on this HRS-formulation, we will generally use the familiar syntax for let-expressions.

We consider five schemes of rules for lifting let-bindings, see below. A step according to a rule from $({}_{\text{let}}{\nearrow}\,@_0)$ or $({}_{\text{let}}{\nearrow}\,@_1)$ lifts a let-binding-group over an application. In steps according to rules from $({}_{\text{let}}{\nearrow}\,\lambda)$, a let-binding-group immediately below an abstraction is either lifted over the abstraction in its entirety, or it is split into a part that is lifted and a part that stays behind. Steps according to rules in $(\textbf{let-in}_{\text{-}\ \text{let}}{\nearrow})$ merge the binding-groups of two let-expressions where one forms the in-part of the other. A step according to rules from $(\textbf{let}_{\text{-}\ \text{let}}{\nearrow})$ lifts, out of its position, the binding-group B' of a let-expression that defines a recursive variable g in a let-binding-group B, merges B with B', and adapts the definition of g accordingly. Sequences of steps due to (exchange)-rules can rearrange the order in which let-bindings occur in a binding-group.

$$({}_{\text{let}}{\nearrow}\,@_0) \quad (\textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ E_0(\vec{f}))\, E_1 \ \to\ \textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ E_0(\vec{f})\, E_1$$

$$({}_{\text{let}}{\nearrow}\,@_1) \quad E_0\, (\textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ E_1(\vec{f})) \ \to\ \textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ E_0\, E_1(\vec{f})$$

$$({}_{\text{let}}{\nearrow}\,\lambda) \quad \lambda x.\, \textbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g},x)\ \textbf{in}\ E(\vec{f},\vec{g},x)$$
$$\to \begin{cases} \textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ \lambda x.\, E(\vec{f},x) & \text{if } \vec{g} \text{ is empty} \\ \textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ \lambda x.\, \textbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g},x)\ \textbf{in}\ E(\vec{f},\vec{g},x) & \text{if neither } \vec{f} \text{ nor } \vec{g} \text{ are empty} \end{cases}$$

$$(\textbf{let-in}_{\text{-}\ \text{let}}{\nearrow}) \quad \textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ \textbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \textbf{in}\ E(\vec{f},\vec{g})$$
$$\to\ \textbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \textbf{in}\ E(\vec{f},\vec{g})$$

$$(\textbf{let}_{\text{-}\ \text{let}}{\nearrow}) \quad \textbf{let}\ \vec{f} = \vec{F}(\vec{f},g),\ g = \textbf{let}\ \vec{h} = \vec{H}(\vec{f},g,\vec{h})\ \textbf{in}\ G(\vec{f},g,\vec{h})\ \textbf{in}\ E(\vec{f},g)$$
$$\to\ \textbf{let}\ \vec{f} = \vec{F}(\vec{f},g),\ g = G(\vec{f},g,\vec{h}),\ \vec{h} = \vec{H}(\vec{f},g,\vec{h})\ \textbf{in}\ E(\vec{f},g)$$

$$(\text{exchange}) \quad \textbf{let}\ B_0,\ f_i = F_i(\vec{f}),\ f_{i+1} = F_{i+1}(\vec{f}),\ B_1\ \textbf{in}\ E(\vec{f})$$
$$\to\ \textbf{let}\ B_0,\ f_{i+1} = F_{i+1}(\vec{f}),\ f_i = F_i(\vec{f}),\ B_1\ \textbf{in}\ E(\vec{f})$$

Here we have used the familiar syntax of let-expressions instead of the underlying HRS-syntax.[1]

[1] E.g. $\mathsf{app}((\mathsf{let}_n\mathsf{\_in}\,(\vec{y}.\,(x_1(\vec{y}),\ldots,x_n(\vec{y}),z_0(\vec{y})))), z_1) \ \to\ \mathsf{let}_n\mathsf{\_in}\,(\vec{y}.\,(x_1(\vec{y}),\ldots,x_n(\vec{y}),\mathsf{app}(z_0(\vec{y}),z_1)))$ are the rules of scheme $({}_{\text{let}}{\nearrow}\,@_0)$ in HRS-notation with the leading abstractions $x_1 \ldots x_n z_0 z_1.$ on either side kept implicit.

Note that an alternative formulation of $({}_{\text{let}}\!\nearrow\lambda)$ that only can lift a let-binding-group over an abstraction in its entirety, but that does not allow to split it, has a drawback. In order to obtain the same let-lifting rewrite relation, also a rule for splitting binding-groups is required, for example the converse of $(\textbf{let-in}_{-\ \text{let}}\!\nearrow)$. But then together with the rule $(\textbf{let-in}_{-\ \text{let}}\!\nearrow)$ itself, which is needed for confluence, avoidable non-termination is introduced in the let-lifting system (which is of a different kind than the non-termination caused by (exchange)-steps alone).

By $\boldsymbol{R}_{\text{let}\nearrow}$ we denote the HRS consisting of the first five rules above. By $\boldsymbol{R}_{\text{let}\nearrow\text{ex}}$ we denote the HRS consisting of all six rules above, thus the extension of $\boldsymbol{R}_{\text{let}\nearrow}$ with the rule (exchange). The rewrite relations of $\boldsymbol{R}_{\text{let}\nearrow}$ and $\boldsymbol{R}_{\text{let}\nearrow\text{ex}}$ are denoted by ${}_{\text{let}}\!\nearrow$ and ${}_{\text{let}}\!\nearrow_{\text{ex}}$, respectively. The rewrite relation $\rightarrow_{\text{ex}}$ is induced by steps according to the rule (exchange), and $=_{\text{ex}}$ is the convertibility relation with respect to $\rightarrow_{\text{ex}}$. The *let-lifting rewrite relation* ${}_{\text{let}}\!\nearrow$ *on* $\boldsymbol{\lambda}_{\mathsf{letrec}}$*-terms* is defined as the rewrite relation ${}_{\text{let}}\!\nearrow$ modulo $=_{\text{ex}}$, that is (see below), by ${}_{\text{let}}\!\nearrow\ := \ =_{\text{ex}} \cdot {}_{\text{let}}\!\nearrow \cdot =_{\text{ex}}$. For example:

$$\lambda x.\,(\textbf{let}\ f = \textbf{let}\ g = x\ \textbf{in}\ g\ \textbf{in}\ f\,)\,x \ {}_{\text{let}}\!\nearrow\ \lambda x.\,(\textbf{let}\ f = g,\ g = x\ \textbf{in}\ f\,)\,x \ {}_{\text{let}}\!\nearrow\ \lambda x.\,\textbf{let}\ f = g,\ g = x\ \textbf{in}\ f\,x$$

is a ${}_{\text{let}}\!\nearrow$-rewrite sequence (and even a ${}_{\text{let}}\!\nearrow$-rewrite sequence) to a normal form. Another final ${}_{\text{let}}\!\nearrow$-step here yields the $=_{\text{ex}}$-equivalent term $\lambda x.\,\textbf{let}\ g = x,\ f = g\ \textbf{in}\ f\,x$. Therefore ${}_{\text{let}}\!\nearrow$ is not confluent. However, it will turn out that ${}_{\text{let}}\!\nearrow$ is 'confluent modulo' $=_{\text{ex}}$.

An *abstract equational rewrite system* $\mathcal{A} = \langle A, \rightarrow, \sim\rangle$ is an abstract rewrite system $\langle A, \rightarrow\rangle$ that is endowed with an equivalence relation $\sim$ on A. The rewrite relation $\rightarrow_{/\sim/}$ of $\rightarrow$ *modulo* $\sim$ is defined as $\rightarrow_{/\sim/} := \sim \cdot \rightarrow \cdot \sim$. The *class rewrite relation* $\rightarrow_{[\sim]}$ *of* $\rightarrow$ *with respect to* $\sim$ is induced by $\rightarrow_{/\sim/}$ on the $\sim$-equivalence classes on A by: for all $a, b \in A$, $[a]_\sim \rightarrow_{[\sim]} [b]_\sim$ if and only if $a \rightarrow_{/\sim/} b$.

The rewrite relation $\rightarrow$ is called *locally confluent modulo* $\sim$ (resp. *confluent modulo* $\sim$) if it holds: $\leftarrow \cdot \rightarrow\ \subseteq\ \twoheadrightarrow \cdot \sim \cdot \twoheadleftarrow$ (resp. $\twoheadleftarrow \cdot \twoheadrightarrow\ \subseteq\ \twoheadrightarrow \cdot \sim \cdot \twoheadleftarrow$). The lemma below reduces confluence properties for $\rightarrow_{/\sim/}$ and $\rightarrow_{[\sim]}$ to corresponding properties of a rewrite relation subsumed by $\rightarrow_{/\sim/}$.

Lemma 1. *Let* $\langle A, \rightarrow, \sim\rangle$ *be an abstract equational rewrite system with* $\sim\ =\ \leftrightarrow^*_\sim$ *for a rewrite relation* $\rightarrow_\sim$ *on* A*. Then it holds: if* $\sim \cdot \rightarrow\ \cup\ \rightarrow_\sim$ *is locally confluent (confluent), then* $\rightarrow_{/\sim/}$ *is locally confluent modulo* $\sim$ *(confluent modulo* $\sim$*), and* $\rightarrow_{[\sim]}$ *is locally confluent (confluent).*

The *let-lifting rewrite relation* ${}_{[\text{let}]}\!\nearrow$ *on* $=_{\text{ex}}$*-equivalence classes of* $\boldsymbol{\lambda}_{\mathsf{letrec}}$*-terms* is defined as the class rewrite relation ${}_{[\text{let}]}\!\nearrow\ := {}_{\text{let}}\!\nearrow_{[=_{\text{ex}}]}$ (note that ${}_{\text{let}}\!\nearrow\ = {}_{\text{let}}\!\nearrow_{/=_{\text{ex}}/}$):

$$[L]_{=_{\text{ex}}}\ {}_{[\text{let}]}\!\nearrow\ [L']_{=_{\text{ex}}}\ :\Longleftrightarrow\ L\ {}_{\text{let}}\!\nearrow\ L' \qquad \text{(for all } \boldsymbol{\lambda}_{\mathsf{letrec}}\text{-terms } L,\ L')\ .$$

Lemma 2. ${}_{\text{let}}\!\nearrow$ *is locally confluent modulo* $=_{\text{ex}}$*, and* ${}_{[\text{let}]}\!\nearrow$ *is locally confluent.*

Proof (Outline). We define a HRS $\boldsymbol{R}_{\text{let}\nearrow\text{ex}}$ with $=_{\text{ex}} \cdot {}_{\text{let}}\!\nearrow\ \cup \rightarrow_{\text{ex}}$ as its rewrite relation, by extending $\boldsymbol{R}_{\text{let}\nearrow\text{ex}}$ through adding, for each rule ρ in $\boldsymbol{R}_{\text{let}\nearrow}$, all variant rules ρ_ϕ with respect to $=_{\text{ex}}$-permutation steps $=^{\phi}_{\text{ex}}$ on the left-hand sides of the pattern of ρ. In this way each rule scheme (σ) of $\boldsymbol{R}_{\text{let}\nearrow}$ gives rise to a rule scheme $(\sigma)_{=_{\text{ex}}}$ of $\boldsymbol{R}_{\text{let}\nearrow\text{ex}}$. Then every step $=^{\phi}_{\text{ex}} \cdot \rightarrow_\rho$ for the rewrite relation $=_{\text{ex}} \cdot {}_{\text{let}}\!\nearrow$, where $\rightarrow_\rho$ is a step according to a rule ρ of scheme (σ) in $\boldsymbol{R}_{\text{let}\nearrow}$, is a step $\rightarrow_{\rho_\phi}$ according to a variant rule ρ_ϕ of scheme $(\sigma)_{=_{\text{ex}}}$ in $\boldsymbol{R}_{\text{let}\nearrow\text{ex}}$.

Now it can be checked that all critical pairs of $\boldsymbol{R}_{\text{let}\nearrow\text{ex}}$ are joinable. For example, solving a critical overlap between rules $({}_{\text{let}}\!\nearrow @_0)$ in $({}_{\text{let}}\!\nearrow @_0)_{=_{\text{ex}}}$ and $({}_{\text{let}}\!\nearrow @_1)$ in $({}_{\text{let}}\!\nearrow @_1)_{=_{\text{ex}}}$:

$$\begin{array}{ccc}
(\textbf{let}\ \vec{f} = F(\vec{f}\,)\ \textbf{in}\ E_0(\vec{f}\,))\,(\textbf{let}\ \vec{g} = G(\vec{g})\ \textbf{in}\ E_1(\vec{g})) & \xrightarrow[({}_{\text{let}}\nearrow @_0)]{} & \textbf{let}\ \vec{f} = F(\vec{f}\,)\ \textbf{in}\ E_0(\vec{f}\,)\,\textbf{let}\ \vec{g} = G(\vec{g})\ \textbf{in}\ E_1(\vec{g}) \\
\downarrow ({}_{\text{let}}\nearrow @_1) & & \vdots\ ({}_{\text{let}}\nearrow @_1) \\
\textbf{let}\ \vec{g} = G(\vec{g})\ \textbf{in}\ (\textbf{let}\ \vec{f} = F(\vec{f}\,)\ \textbf{in}\ E_0(\vec{f}\,))\,E_1(\vec{g}) & & \textbf{let}\ \vec{f} = F(\vec{f}\,)\ \textbf{in}\ \textbf{let}\ \vec{g} = G(\vec{g})\ \textbf{in}\ E_0(\vec{f}\,)\,E_1(\vec{g}) \\
\vdots\ ({}_{\text{let}}\nearrow @_0) & & \vdots\ (\textbf{let-in}_{-\ \text{let}}\nearrow) \cdot =_{\text{ex}} \\
\textbf{let}\ \vec{g} = G(\vec{g})\ \textbf{in}\ \textbf{let}\ \vec{f} = F(\vec{f}\,)\ \textbf{in}\ E_0(\vec{f}\,)\,E_1(\vec{g}) & \xrightarrow[(\textbf{let-in}_{-\ \text{let}}\nearrow)]{\cdots\cdots} & \textbf{let}\ \vec{g} = G(\vec{g}),\ \vec{f} = F(\vec{f}\,)\ \textbf{in}\ E_0(\vec{f}\,)\,E_1(\vec{g})
\end{array}$$

Then the critical pair theorem for HRSs [6] [10, Thm. 11.6.44] (note that the possibility to find all critical pairs for a HRS is based on a matching algorithm for HRS first described in [6]) yields that $=_{\mathrm{ex}} \cdot {}_{\mathrm{let}}\!\nearrow \cup \to_{\mathrm{ex}}$ is locally confluent. From this, it follows by Lemma 1 that ${}_{\mathrm{let}}\!\nearrow\; = {}_{\mathrm{let}}\!\nearrow_{/=_{\mathrm{ex}}/}$ is locally confluent modulo $=_{\mathrm{ex}}$, and that ${}_{[\mathrm{let}]}\!\nearrow$ is locally confluent. □

Remark 3. This proof (or actually that of Theorem 6) could also be based on an HRS-analogue of a critical pair theorem by Petersen and Stickel [7, Thm. 9.3] for TRSs that are endowed with an equational theory. Other versions of critical pair theorems for TRSs that are based on 'critical $\to$-pairs modulo $\sim$' (e.g. Jouannaud [5]) suppose that $\to$ is $\sim$-*coherent*: if $t \sim s$ and $t \to^{+} t_1$, then there there exist t_1' and s' with $t_1 \twoheadrightarrow t_1'$ and $s \to^{+} s'$ such that $t_1' \sim s'$. Yet the relation ${}^{\mathrm{let}}\!\searrow$ here is *not* $=_{\mathrm{ex}}$-coherent: while $\lambda x.\, \mathbf{let}\ f = \lambda y.\, y,\ g = x\ \mathbf{in}\ f\, g$ admits an ${}_{\mathrm{let}}\!\nearrow$-step according to a rule of $({}_{\mathrm{let}}\!\nearrow \lambda)$, the $=_{\mathrm{ex}}$-equivalent term $\lambda x.\, \mathbf{let}\ g = x,\ f = \lambda y.\, y\ \mathbf{in}\ f\, g$ is a ${}_{\mathrm{let}}\!\nearrow$-normal form. In order to apply (an HRS-analogue of) such a theorem, the system has to be extended to one with rewrite relation $=_{\mathrm{ex}} \cdot {}_{\mathrm{let}}\!\nearrow$ by introducing variant rules as in the proof above (also done in [7]).

Proposition 4. *${}_{\mathrm{let}}\!\nearrow$ and ${}_{[\mathrm{let}]}\!\nearrow$ are terminating.*

Proposition 5. *In every ${}_{\mathrm{let}}\!\nearrow$-normal form, subterms starting with* **let** *occur only at the root or below λ-abstractions. The same holds for every term representing a ${}_{[\mathrm{let}]}\!\nearrow$-normal form.*

Theorem 6. *${}_{[\mathrm{let}]}\!\nearrow$ is confluent and terminating, and has the unique normalization property.*

Proof. From Lemma 2 and Proposition 4 by Newman's Lemma [10, Thm. 1.2.1]. □

2 Let-sinking

A candidate for a rewrite system for sinking let-bindings is the HRS that arises from the let-lifting HRS $\boldsymbol{R}_{\mathrm{let}\nearrow}$ by reversing all of its rules. Unfortunately the resulting system is not confluent. The problem is that the splitting rules for binding-groups, the converses of rules in ($\mathbf{let\text{-}in}_{-}\ {}_{\mathrm{let}}\!\nearrow$), allow to sink, for a let-binding-group with two independent parts, each part into the other, so that, in many situations, the results cannot be joined again. We note that adding ($\mathbf{let\text{-}in}_{-}\ {}_{\mathrm{let}}\!\nearrow$) would remedy the situation, but at the cost of yielding a non-terminating let-sinking system.

Here we disallow the splitting rules for let-binding-groups altogether, but keep their converses from ($\mathbf{let\text{-}in}_{-}\ {}_{\mathrm{let}}\!\nearrow$), yet now call the scheme (${}^{\mathrm{let}}\!\searrow \mathbf{let}_{-}$). Yet we integrate the splitting rules into those let-binding-movement rules for which sinking of entire binding-groups is not always possible, namely rules for sinking let-bindings into the left or right subterm of an application, see the rule schemes $({}_{\mathrm{let}}\!\nearrow @_0)$ and $({}_{\mathrm{let}}\!\nearrow @_1)$ below. As reflected in rules from $({}^{\mathrm{let}}\!\searrow \lambda)$, let-binding-groups can always be sunk into a λ-abstraction. The rule ($\mathbf{let}_{-}\,{}^{\mathrm{let}}\!\searrow$) is the converse of ($\mathbf{let}_{-}\ {}_{\mathrm{let}}\!\nearrow$). So we consider the following five rule schemes for sinking let-bindings:

$$({}_{\mathrm{let}}\!\nearrow @_0)\quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E_0(\vec{f},\vec{g})\ E_1(\vec{f})$$
$$\to \begin{cases} \big(\mathbf{let}\ \vec{g} = \vec{G}(\vec{g})\ \mathbf{in}\ E_0(\vec{g})\big)\ E_1 & \text{if } \vec{f} \text{ is empty} \\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \big(\mathbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E_0(\vec{f},\vec{g})\big)\ E_1(\vec{f}) & \text{if neither } \vec{f} \text{ nor } \vec{g} \text{ are empty} \end{cases}$$

$$({}_{\mathrm{let}}\!\nearrow @_1)\quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E_0(\vec{f})\ E_1(\vec{f},\vec{g})$$
$$\to \begin{cases} E_0\ \big(\mathbf{let}\ \vec{g} = \vec{G}(\vec{g})\ \mathbf{in}\ E_1(\vec{g})\big) & \text{if } \vec{f} \text{ is empty} \\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\ \big(\mathbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E_1(\vec{f},\vec{g})\big) & \text{if neither } \vec{f} \text{ nor } \vec{g} \text{ are empty} \end{cases}$$

$$({}^{\mathrm{let}}\!\searrow \lambda)\quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \lambda x.\, E(\vec{f},x) \;\to\; \lambda x.\, \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E(\vec{f},x)$$

$({}^{\text{let}}{\searrow}\,\textbf{let}_{\_})$ $\quad \textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ \textbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \textbf{in}\ E(\vec{f},\vec{g}) \ \to\ \textbf{let}\ \vec{f} = \vec{F}(\vec{f}), \vec{g} = \vec{G}(\vec{f},\vec{g})\ \textbf{in}\ E(\vec{f},\vec{g})$

$(\textbf{let}_{\_}{}^{\text{let}}{\searrow})$ $\quad \textbf{let}\ \vec{f} = \vec{F}(\vec{f},g),\ g = G(\vec{f},g,\vec{h}),\ \vec{h} = \vec{H}(\vec{f},g,\vec{h})\ \textbf{in}\ E(\vec{f},g)$

$$\to\ \textbf{let}\ \vec{f} = \vec{F}(\vec{f},g),\ g = \textbf{let}\ \vec{h} = \vec{H}(\vec{f},g,\vec{h})\ \textbf{in}\ G(\vec{f},g,\vec{h})\ \textbf{in}\ E(\vec{f},g)$$

and additionally, the rules of the scheme (exchange) from $\boldsymbol{R}_{\text{let}\nearrow}$. By $\boldsymbol{R}^{\text{let}\searrow}$ we denote the HRS consisting of the five rules above, and by $\boldsymbol{R}^{\text{let}\searrow\text{ex}}$ its extension with the rule (exchange). The rewrite relations of $\boldsymbol{R}^{\text{let}\searrow}$ and $\boldsymbol{R}^{\text{let}\searrow\text{ex}}$ are denoted by ${}^{\text{let}}{\searrow}$ and ${}^{\text{let}}{\searrow}^{\text{ex}}$, respectively.

Since the binding-group merge rules with induced rewrite relation $\to_{\text{merge}}$ are part of both $\boldsymbol{R}_{\text{let}\nearrow}$ and $\boldsymbol{R}^{\text{let}\searrow}$ (in the schemes $(\textbf{let-in}_{\_\ \text{let}}{\nearrow})$ in $\boldsymbol{R}_{\text{let}\nearrow}$ and $({}^{\text{let}}{\searrow}\,\textbf{let}_{\_})$ in $\boldsymbol{R}^{\text{let}\searrow}$), the induced let-lifting and let-sinking rewrite relations are not precisely each other's converse. See e.g.:

$$\lambda x.\,\textbf{let}\ f = x,\ g_1 = g_2\,f,\ g_2 = g_1\,f\ \textbf{in}\ g_1\,g_2 \quad \overset{\text{let}}{\not\searrow} \ {}^{\nwarrow}{}_{\text{let}} \quad \lambda x.\,\textbf{let}\ f = x\ \textbf{in}\ \textbf{let}\ g_1 = g_2\,f,\ g_2 = g_1\,f\ \textbf{in}\ g_1\,g_2$$

Observe that the term on the left is a ${}^{\text{let}}{\searrow}$-normal form, and that the ${\nwarrow}_{\text{let}}$-step is a $\leftarrow_{\text{merge}}$-step. This example also shows that let-sinking does not always stack let-bindings as deeply as possible. This, however, is consistent with the definition of 'lambda-dropping' and 'block-sinking' in [1].

Proposition 7. *Every* ${}^{\text{let}}{\searrow}$*-step is either a* $\to_{\text{merge}}$*-step or the converse of a* ${}_{\text{let}}{\nearrow}$*-step followed by at most one* $\to_{\text{merge}}$*-step. Every* ${}_{\text{let}}{\nearrow}$*-step is either a* $\to_{\text{merge}}$*-step or the converse of a* ${}^{\text{let}}{\searrow}$*-step followed by at most one* $\to_{\text{merge}}$*-step.*

The *let-sinking rewrite relation* ${}^{\text{let}}{\searrow}$ *on* $\boldsymbol{\lambda}_{\text{letrec}}$*-terms* is defined as the rewrite relation ${}^{\text{let}}{\searrow}$ modulo $=_{\text{ex}}$, that is, by: ${}^{\text{let}}{\searrow} \ :=\ {}^{\text{let}}{\searrow}_{/=_{\text{ex}}/} \ =\ =_{\text{ex}} \cdot {}^{\text{let}}{\searrow} \cdot =_{\text{ex}}$. The *let-sinking rewrite relation* ${}^{[\text{let}]}{\searrow}$ *on* $=_{\text{ex}}$*-equivalence classes of* $\boldsymbol{\lambda}_{\text{letrec}}$*-terms* is defined as the class rewrite relation ${}^{[\text{let}]}{\searrow} := {}^{\text{let}}{\searrow}_{[=_{\text{ex}}]}$.

As an example we consider the following ${}^{\text{let}}{\searrow}$-rewrite sequence (it is actually a ${}^{\text{let}}{\searrow}$-rewrite sequence) to normal form (this is the converse of the example above for ${}_{\text{let}}{\nearrow}$):

$$\lambda x.\,\textbf{let}\ f = g,\ g = x\ \textbf{in}\ f\,x \ \ {}^{\text{let}}{\searrow}\ \ \lambda x.\,(\textbf{let}\ f = g,\ g = x\ \textbf{in}\ f)\,x \ \ {}^{\text{let}}{\searrow}\ \ \lambda x.\,(\textbf{let}\ f = \textbf{let}\ g = x\ \textbf{in}\ g\ \textbf{in}\ f)\,x$$

For similar (trivial) reasons as explained for ${}_{\text{let}}{\nearrow}$, also ${}^{\text{let}}{\searrow}$ is not confluent. But while ${}_{\text{let}}{\nearrow}$ is confluent modulo $=_{\text{ex}}$, this is not the case for ${}^{\text{let}}{\searrow}$, and neither is ${}^{[\text{let}]}{\searrow}$ confluent, yet. In order to see this, consider the following forking ${}^{\text{let}}{\searrow}$-steps:

$$\lambda x.\,\lambda y.\,(\textbf{let}\ f = \lambda z.\,z\ \textbf{in}\ x)\,y \ \ {\swarrow}^{\text{let}}\ \ \lambda x.\,\lambda y.\,\textbf{let}\ f = \lambda z.\,z\ \textbf{in}\ x\,y \ \ {}^{\text{let}}{\searrow}\ \ \lambda x.\,\lambda y.\,x\,(\textbf{let}\ f = \lambda z.\,z\ \textbf{in}\ y)$$

Here the $=_{\text{ex}}$-equivalence classes of the reducts (obtained by rules in $({}_{\text{let}}{\nearrow}\,@_0)$ and $({}_{\text{let}}{\nearrow}\,@_1)$ respectively) cannot be joined, because the redundant let-binding $f = \lambda z.\,z$ cannot be removed. Therefore we extend the system by two rules for removing redundant and empty let-bindings:

$$\text{(reduce)}\quad \textbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \textbf{in}\ E(\vec{f}) \ \to\ \textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ E(\vec{f})$$

$$\text{(nil)}\quad \textbf{let}\ \ \textbf{in}\ L \ \to\ L$$

which can be called rules for *garbage collection* (in analogy with literature on explicit substitution). The rewrite relation $\to_{\text{gc}}$ is induced by steps according to the rules (reduce) and (nil). The *let-sinking/reduce rewrite relation* ${}^{\text{let}}{\searrow}^{\text{gc}}$ is defined as the rewrite relation ${}^{\text{let}}{\searrow} \cup \to_{\text{gc}}$ modulo $=_{\text{ex}}$, that is, by: ${}^{\text{let}}{\searrow}^{\text{gc}} \ :=\ ({}^{\text{let}}{\searrow} \cup \to_{\text{gc}})_{/=_{\text{ex}}/} \ =\ =_{\text{ex}} \cdot ({}^{\text{let}}{\searrow} \cup \to_{\text{gc}}) \cdot =_{\text{ex}}$ And the *let-sinking/reduce rewrite relation* ${}^{[\text{let}]}{\searrow}^{[\text{gc}]}$ *on* $=_{\text{ex}}$*-equivalence classes of* $\boldsymbol{\lambda}_{\text{letrec}}$*-terms* is defined as the class rewrite relation ${}^{[\text{let}]}{\searrow}^{[\text{gc}]} := {}^{\text{let}}{\searrow}^{\text{gc}}{}_{[=_{\text{ex}}]}$.

Using these relations we can join the forking steps from above as follows:

$$\lambda x.\,\lambda y.\,(\textbf{let}\ f = \lambda z.\,z\ \textbf{in}\ x)\,y \ \twoheadrightarrow_{\text{gc}}\ \lambda x.\,\lambda y.\,x\,y \ \twoheadleftarrow_{\text{gc}}\ \lambda x.\,\lambda y.\,x\,(\textbf{let}\ f = \lambda z.\,z\ \textbf{in}\ y)$$

Remark 8. In [2, 9] we introduce and study a rewrite system (formalized as a Combinatory Reduction System) for unfolding $\boldsymbol{\lambda}_{\mathsf{letrec}}$-terms into infinite λ-terms. This system contains a rule scheme that enables more general steps than those of the scheme (reduce), namely:

$$(\varrho_{\triangledown}^{\text{reduce}}): \quad \textbf{letrec}\ f_1 = L_1 \dots f_n = L_n \ \mathsf{in}\ L \ \rightarrow \ \textbf{letrec}\ f_{j_1} = L_{j_1} \dots f_{j_{n'}} = L_{j_{n'}} \ \mathsf{in}\ L$$
$$(\text{if } f_{j_1}, \dots, f_{j_{n'}} \text{ are the recursion variables that are reachable from } L)$$

However, due to the presence of the rule scheme (exchange) in the systems we consider here, every step according to a rule of $(\varrho_{\triangledown}^{\text{reduce}})$ can be simulated by a number of $\rightarrow_{\text{ex}}$-steps followed by a step according to a rule of (reduce). Thus the syntactically easier rules of (reduce) suffice here. The availability of the rules of (exchange) also enables the use of the rules $({}_{\text{let}}{\nearrow}\,\lambda)$ and $({}^{\text{let}}{\searrow}\,@_i)$ $(i \in \{0,1\})$ in which a call graph analysis is enforced by a pattern of rather easy form.

Lemma 9. ${}^{\text{let}}{\searrow}^{\text{gc}}$ *is locally confluent modulo* $=_{\text{ex}}$*, and* ${}^{[\text{let}]}{\searrow}^{[\text{gc}]}$ *is locally confluent.*

Proof (Idea). Similarly as in the proof of Lemma 2, a critical-pair analysis is carried out for a HRS $\boldsymbol{R}^{\text{let}\searrow\text{gc}}$ with $\rightarrow_{\text{ex}} \cup =_{\text{ex}} \cdot ({}^{\text{let}}{\searrow} \cup \rightarrow_{\text{gc}})$ as its rewrite relation. Here the analysis is more laborious (two more rules), and considerably more tedious (for three schemes, $({}_{\text{let}}{\nearrow}\,@_0)$, $({}_{\text{let}}{\nearrow}\,@_1)$, and $(\textbf{let}_{\_}\,{}^{\text{let}}{\searrow})$, the rule patterns create splits of let-binding-groups, which in order to join critical steps requires a careful analysis of the possible call graphs between let-bindings in their source term). The lemma folows by the Critical Pair Theorem of [6] and Lemma 1. □

Proposition 10. ${}^{\text{let}}{\searrow}^{\text{gc}}$ *and* ${}^{[\text{let}]}{\searrow}^{[\text{gc}]}$ *are terminating.*

Theorem 11. ${}^{[\text{let}]}{\searrow}^{[\text{gc}]}$ *is confluent, terminating, and has the unique normalization property.*

The properties stated for ${}^{[\text{let}]}{\searrow}^{[\text{gc}]}$ in Thm. 11 and for ${}_{[\text{let}]}{\nearrow}$ in Thm. 6 can also be shown for the extension ${}^{[\text{let}]}{\nearrow}^{[\text{gc}]}$ of the let-lifting rewrite relation ${}_{[\text{let}]}{\nearrow}$ by incorporating $\rightarrow_{\text{gc}}$-steps. Finally, a comprehensive HRS for let-floating in both upward and downward direction, and for reducing binding-groups can be obtained by gathering all rules underlying ${}_{\text{let}}{\nearrow}$ and ${}^{\text{let}}{\searrow}^{\text{gc}}$.

Acknowledgement. We want to thank the reviewers for their valuable comments and suggestions.

References

[1] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243 – 287, 2000. PEPM'97.

[2] Clemens Grabmayer and Jan Rochel. Expressibility in the Lambda-Calculus with Letrec. Technical Report arXiv:1208.2383, `arxiv.org`, August 2012. `http://arxiv.org/abs/1208.2383`.

[3] Clemens Grabmayer and Jan Rochel. Term Graph Representations for Cyclic Lambda Terms. In *Proc. of TERMGRAPH 2013*, number 110 in EPTCS, 2013. `http://arxiv.org/abs/1302.6338v1`.

[4] Simon Peyton Jones. *The Implementation of Functional Progr. Languages.* Prentice-Hall, 1987.

[5] Jean-Pierre Jouannaud. Confluent and coherent equational term rewriting systems application to proofs in abstract data types. In Giorgio Ausiello and Marco Protasi, editors, *CAAP'83*, volume 159 of *Lecture Notes in Computer Science*, pages 269–283. Springer Berlin Heidelberg, 1983.

[6] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3–29, 1998.

[7] Gerald E. Peterson and Mark E. Stickel. Complete Sets of Reductions for Some Equational Theories. *JACM*, 28(2):233–264, 1981.

[8] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 1–12, New York, NY, USA, 1996. ACM.

[9] Jan Rochel and Clemens Grabmayer. Confluent unfolding in the λ-calculus with letrec. In *Proceedings of IWC 2013 (2nd International Workshop on Confluence)*, 2013.

[10] Terese. *Term Rewriting Systems.* Cambridge University Press, 2003.