A Single-Step Term-Graph Reduction System for Proof Assistants

Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer

maartenm@cs.ru.nl, marko@cs.ru.nl, rinus@cs.ru.nl
Department of Software Technology, Nijmegen University, The Netherlands.

Abstract. In this paper, we will define a custom term-graph reduction system for a simplified lazy functional language. Our custom system is geared towards *flexibility*, which is accomplished by leaving the choice of redex free and by making use of single-step reduction. It is therefore more suited for formal reasoning than the well-established standard reduction systems, which usually fix a single redex and realize multi-step reduction only. We will show that our custom system is correct with respect to the standard systems, by proving that it is *confluent* and allows standard lazy functional evaluation as a possible reduction path.
Our reduction system is used in the foundation of Sparkle. Sparkle is the dedicated proof assistant for Clean, a lazy functional programming language based on term-graph rewriting. An important reasoning step in Sparkle is the replacement of an expression with one of its reducts. The *flexibility* of our underlying reduction mechanism ensures that as many reduction options as possible are available for this reasoning step, which improves the ease of reasoning.
Because our reduction system is based on a simplified lazy functional language, our results can be applied to any other functional language based on term-graph rewriting as well.

1 Introduction

Clean[20] and Haskell[14] are lazy functional programming languages that have a semantics based on term-graph rewriting. In 2001, the distribution of Clean was extended with the dedicated proof assistant Sparkle[8]. With this new tool, it became much easier to reason about lazy functional programs, and to formally prove logic properties of these programs.

Industry is beginning to acknowledge the importance of formal methods for verifying safety-critical components of both hardware and software (for instance, see [5]). Due to their mathematical nature, functional programming languages are well suited for formal methods. Consequently, functional languages are being used increasingly often in industrial practice (for instance, see [17]).

Since its introduction[7], Sparkle has been used in practice for various purposes. It has been used for proving properties of I/O-programs by Dowse[10] and Butterfield[6]. An extension for dealing with temporal properties has been proposed for it by Tejfel, Horváth and Koszik[19, 13]. It has been used in education

at the Radboud University of Nijmegen. Furthermore, support for class-generic properties has been added to it by van Kesteren[15].

Building proofs with Sparkle consists of the repeated application of reasoning steps. Sparkle offers a library of about 40 reasoning steps, some of which are generic for formal reasoning, and some of which are specific for dealing with lazy functional programs. An important reasoning step in this library is 'Reduce', which makes use of the operational semantics of the underlying programming language to replace an expression with any of its reducts.

The usefulness of 'Reduce' depends on the reduction options that are made available by the underlying reduction system, which must be sufficiently *flexible*. Of course, it also has to support lazy evaluation, graphs and sharing. Normally, the natural choice would be the well-established system of Launchbury[16]. This system, however, is geared towards evaluation: it uses multi-step reduction and fixes a single redex. Therefore, both *partial* and *inner* reductions are not elements of its formal reduction relation, and are not provided as reduction options.

In this paper, we will define a custom reduction system that is flexible. Our system is based on Launchbury's, but uses single-step reduction and leaves the choice of redex free. Therefore, partial and inner reducts are elements of the formalized reduction relation, and our custom system is suited to be used as the foundation of formal reasoning. We will show that our system is confluent and that the standard lazy functional reduction path is allowed by it. This ensures that our system behaves correctly with respect to Launchbury's system.

Our reduction system will be used in the theoretical foundation of Sparkle. Without loss of generality, we have restricted ourselves to a simplified functional language. Our system is therefore applicable to other functional languages too.

This paper is structured as follows. In Section 2, we examine the desired level of flexibility. We introduce our expression language in Section 3, and describe our reduction system in Section 4. We show how to express standard reduction paths in our system in Section 5, and we prove confluence of our system in Section 6. Finally, we discuss related work in Section 7 and draw conclusions in Section 8.

2 Desired level of flexibility

Replacing expressions with reducts is a very natural and intuitive reasoning step. The flexibility of the underlying reduction system determines the number of reduction options that are available for this step. In principle, having more reduction options increases the power of reasoning. This reasoning power is only useful, however, if the options can intuitively be recognized as reducts.

In the introduction, two factors were mentioned that influence flexibility: the granularity of the reduction relation (single-step vs multi-step), and the freedom of choice of redex (fixed redex vs free redex). In the following sections, we will examine the precise effect of these factors on formal reasoning more closely.

2.1 Granularity of reduction steps

On the intuitive level, reduction is mainly considered to be defined by means of the reduction steps, and only secondary by means of the overarching reduction

relation. On the reasoning level, the reduction options that are offered to the proof builder should therefore include the results of partial reductions as well. To formalize this, a single-step reduction system is needed, in which the reduction relation is defined in terms of single applications of individual reduction steps.

Example: *(proof that requires intermediate reducts)*
Assume that the following property has been proved:
'$\forall_b[\texttt{not}\ (\texttt{not}\ b) = b]$'.
Using this property, assume that we now want to prove the following:
'$\texttt{not}\ (\texttt{id}\ (\texttt{not}\ X)) = X$' (where X is some complex computation)
On the intuitive level, this is a trivial proof: simply replace '$\texttt{id}\ (\texttt{not}\ X)$' with '$\texttt{not}\ X$', and then apply the assumed property. QED.
This intuitive proof, however, relies on single-step inner reduction. If no inner reduction is available, then '$\texttt{id}\ (\texttt{not}\ X))$' cannot be selected as redex; if no single-step reduction is available, then the reduction of '$\texttt{id}\ (\texttt{not}\ X))$' cannot be stopped after the first step and 'X' will be evaluated unnecessarily.

2.2 Choice of redex

Because lazy functional languages are referentially transparent, it is always safe to apply reduction to an inner redex. Formally, however, referential transparency has to be proved too. This proof can be constructed in two different ways:

1. Start with a reduction system that allows leftmost-outermost reduction only. Define semantic equality on top, and prove that it is referentially transparent.
2. Start with a reduction system that allows arbitrary redexes to be reduced. Prove that this system is confluent, define a semantic equality on top of it, and let referential transparency follow from the already shown confluence.

Because semantic equality needs to cope with infinite reductions (bisimulation), the second approach is much easier to carry out. Therefore, in this paper we will allow the redex to be chosen freely, and we will explicitly prove confluence.

3 The expression language

Our expression language models the core of an arbitrary lazy functional language. The basic components of our language are variables, functions, applications and let expressions. Without loss of generality, we assume that each function symbol has a fixed arity, and we abstract from constructors and cases, which can be added without difficulties. We represent function definitions in a constant external environment, and do not use lambda expressions. We consider sharing to be a basic component of any lazy functional language.

Notations: *(variables, function symbols and lists)*
Let $\mathcal{V}$ denote the set of variable names, $\mathcal{F}$ the set of function symbols, and $Arity : \mathcal{F} \to \mathbb{N}$ the function that obtains the arity of a function symbol.

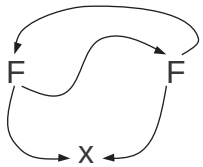
Let *Vars* and *Bound* denote the functions that obtain the free and bound variables of an expression respectively. Let '$\langle$' and '$\rangle$' denote lists, $\#xs$ the length of a list xs, and $xs!i$ the i-th element of xs, if it exists. Let $Unq(xs)$ denote that all elements in xs occur only once.

Notation: *(construction of sets)*
In this paper, sets will be denoted by means of $\{O(x_i) \mid x_i \in X_i \mid P(x_i)\}$, in which $O(x_i)$ describes the syntactical shape of the set elements, $x_i \in X_i$ describes the domains of the variable placeholders, and $P(x_i)$ describes the condition that all elements of the set must adhere to.

Definition 3.1: *(set of expressions)*
The set $\mathcal{E}$ of expressions is defined recursively by:

$$\begin{aligned}\mathcal{E} = \{&\mathsf{var}\ x && \mid x \in \mathcal{V}\}\\ \cup\ \{&\mathsf{fun}\ f\ \mathsf{on}\ xs && \mid f \in \mathcal{F}, xs \in \langle\mathcal{V}\rangle \mid \mathit{Arity}(f) \geq \#xs\}\\ \cup\ \{&\mathsf{app}\ e\ \mathsf{to}\ x && \mid e \in \mathcal{E}, x \in \mathcal{V}\}\\ \cup\ \{&\mathsf{let}\ xs = es\ \mathsf{in}\ e && \mid xs \in \langle\mathcal{V}\rangle, es \in \langle\mathcal{E}\rangle, e \in \mathcal{E} \mid \#xs = \#es \wedge \mathit{Unq}(xs)\}\end{aligned}$$

Example: *(term-graph expression with cycles)*
Our representation of expressions allows cycles to be represented by means of recursive lets. For instance, assuming the availability of a function symbol F (arity 2) and a variable x, and assuming that the leftmost occurrence of F is the root, the following graph and expression are equivalent:



$$\begin{aligned}\mathsf{let}\ \langle a, b, c\rangle = \langle\ &\mathsf{fun}\ F\ \mathsf{on}\ \langle\mathsf{var}\ c, \mathsf{var}\ b\rangle\\ ,\ &\mathsf{fun}\ F\ \mathsf{on}\ \langle\mathsf{var}\ c, \mathsf{var}\ a\rangle\\ ,\ &\mathsf{var}\ x\\ \rangle\\ \mathsf{in}\ (&\mathsf{var}\ a)\end{aligned}$$

Assumption 3.2: *(programs)*
Assume the function $\mathit{Body} : \langle\mathcal{V}\rangle \times \langle\mathcal{V}\rangle \times \mathcal{F} \to \mathcal{E}$, which models the program context and binds function symbols to fresh copies of their function bodies. Assume that $\mathit{Body}(xs, ys, f)$ denotes the body of f in which the arguments have been replaced by xs and the bound variables have been replaced by ys.

Example: *(use of the program function)*
Assume that the function `f` is defined as follows:

```
f x = let y = x+x in y+y
```

Formalized by means of the *Body*-function, this becomes:

$\mathit{Body}(f, E, z) =$ (`let z = E+E in z+z`)

The *Body*-function therefore expands a function on given arguments, using the argument variables to create a fresh instantiation of the function body.

Note that there are two different alternatives for application in our language. The '`fun`'-alternative is used for lifting function symbols to the expression level, and for gradually collecting function arguments. The '`app`'-alternative is used for applications of expressions that still have to be reduced to function symbols.

Note further that the arguments of both kinds of applications must always be variables. Because of this convention (which we borrow from [16]), expressions need to be converted before they can be represented in our language. Each application that occurs in the expression has to be transformed as follows:

$$\begin{aligned} \mathit{Transform}(\mathsf{fun}\ f\ \mathsf{on}\ es) &= \mathsf{let}\ xs = es\ \mathsf{in}\ (\mathsf{fun}\ f\ \mathsf{on}\ xs) \\ \mathit{Transform}(\mathsf{app}\ e_1\ \mathsf{to}\ e_2) &= \mathsf{let}\ \langle x \rangle = \langle e_2 \rangle\ \mathsf{in}\ (\mathsf{app}\ e_1\ \mathsf{to}\ x) \end{aligned}$$

This transformation has to be carried out recursively, and the variables that are created must be fresh. We do not lose expressiveness, because each expression can be transformed this way. The advantage of this convention is that function arguments can be duplicated without loss of sharing. This makes our function expansion rule much easier, as it is no longer necessary to create fresh variables (for sharing function arguments) within the rule itself.

Note that the transformation can never be reversed, because the result would be an expression that cannot be represented in our system. This is not a problem, because reduction never requires the transformation to be reversed.

4 Reduction System

In the following sections, we will introduce our reduction system step-by-step. First, we introduce our approach to handling sharing in Section 4.1. Then, we describe the individual rules of our system in Sections 4.2(applications), 4.3(lets) and 4.4(unsharing). By combining individual rules, *head reduction* is formalized in Section 4.5. Finally, locations are introduced in Section 4.6, and they are used to upgrade head reduction to *inner reduction* in Section 4.7.

4.1 Graphs as self-contained expressions

Sharing is handled in our reduction system in a way that is not standard. We do not use an external environment for storing graph nodes, and we do not have a reduction rule that removes let bindings from an expression and transfers them to an external environment. Instead, we store graph nodes *within* the expression by means of lets and use a *let-lifting* mechanism.

The goal of our method is get rid of external environments completely, which normally have to be dragged along continuously. By maintaining graph nodes internally, expressions become self-contained; they can be reduced and given a meaning without pairing them to an external object. This makes handling expressions more transparent, and makes subsequent definitions and proofs easier.

The disadvantage of our method is that additional functionality is needed for maintaining let definitions internally. Two tasks have to be performed:

- *If reduction requires a subexpression at a specific location to be in a certain form, then it must be possible to remove a leading let from that location.*
  Example: '$\mathsf{app}\ (\mathsf{let}\ \langle x \rangle = \langle e \rangle\ \mathsf{in}\ (\mathsf{fun}\ f\ \mathsf{on}\ \langle x \rangle))\ \mathsf{to}\ y$'. *(arity of f is 2)*
  Reduction should first join the outer app and the inner fun, adding y to the argument list $\langle x \rangle$. Then, reduction should expand f.

Unfortunately, the let expression in the middle prevents the contraction rule from matching immediately. Normally, this would not be a problem, because reduction would be able to move the inner let to an external environment. In our case, the inner let cannot be removed, and another solution is needed.

– *If reduction requires a variable to be unshared, then an explicit link has to be created to the corresponding let binding.*
Example: '$\mathsf{let}\ \langle x\rangle = \langle e\rangle\ \mathsf{in}\ (\mathsf{app}\ (\mathsf{var}\ x)\ \mathsf{to}\ y)$'. *(assume that e is in nf)*
Reduction should now replace the inner '$\mathsf{var}\ x$' with e. This requires the inner reduction of '$\mathsf{var}\ x$' to know about the external binding of x to e.
Normally, reduction of the expression as a whole would introduce $x = e$ into the external environment, by means of which the information would be made available. Because we do not use external environments, we have to find another way of passing down this information.

Fortunately, solutions to the issues above can be realized easily, see Sections 4.3 and 4.4 respectively. Overall, our reduction system remains very simple.

4.2 The reduction rules for applications

In our system, applications are contracted from initial sequences of app-nodes into fun-nodes. When sufficient arguments have been collected, the function is expanded. This process can be realized by the following two reduction rules:

– The collect-rule accumulates function arguments into a central fun-node by removing them from surrounding app-nodes. This process is repeated until the fun-node is filled and contains as many arguments as its arity describes.
– The expand-rule replaces a filled fun-node with (a fresh copy) of the body of the function (obtained with *Body*, see Assumption 3.2). Additional context information is required in the form of a list of fresh variables, which are used as instantiation for the bound variables of the body.

In this paper, we will formalize reduction by means of deterministic functions, because this makes proving confluence much easier. If additional information is required to accomplish deterministic behavior, then it is assumed to be available by means of input arguments. In the later stages of the formalization of reduction, it will be described how this information is obtained.

The reduction rules collect and expand are formalized as follows:

Definition 4.2.1: *(the realization of the* collect*-rule)*
The function $Collect : \mathcal{E} \to \mathcal{E}$ is defined by:

$$Collect(e) = \begin{cases} \mathsf{fun}\ f\ \mathsf{on}\ \langle xs : x\rangle & \textbf{if } e = (\mathsf{app}\ (\mathsf{fun}\ f\ \mathsf{on}\ xs)\ \mathsf{to}\ x) \\ & \quad \wedge\ Arity(f) > \#xs \\ e & \textbf{otherwise} \end{cases}$$

Definition 4.2.2: *(the realization of the* expand*-rule)*
The function $Expand : \langle \mathcal{V}\rangle \times \mathcal{E} \to \mathcal{E}$ is defined by:

$$Expand(ys, e) = \begin{cases} Body(xs, ys, f) & \textbf{if } e = (\mathsf{fun}\ f\ \mathsf{on}\ xs) \wedge Arity(f) = \#xs \\ e & \textbf{otherwise} \end{cases}$$

Note that, as a consequence of allowing only variables at argument positions, the reduction rules for function application do not have to take sharing into account in any way. Instead, sharing is preserved automatically.

4.3 The reduction rules for let lifting

For the administration of sharing, our reduction system maintains lets within expressions, instead of moving them into an external environment. This means that lets may get in the way of reduction: when a subexpression has to be brought into a certain form, it is possible that a let is created on its outer level. For reduction to continue, it must be possible to remove this hindering let.

Our basic idea is to move lets upwards until they are no longer in the way. This approach works, because: (1) lets at the outermost level can never be in the way; and (2) upward moves can be achieved easily at all relevant locations. We will call the upward move of a let a let lift; our alternative for external environments is therefore the process of *let lifting*.

In our system, there are two places where a let must be lifted upwards:

- *On the left-hand-side of an application.*
  The expression on the left-hand-side of an app-node must be reduced to a fun-node in order for reduction to continue by means of an application of the collect-rule. If a let expression appears at the outermost level of the left-hand-side of an application, it therefore has to be moved out of the way.
- *On the right-hand-side of a let binding.*
  An important step in the functional reduction strategy is the unsharing of a stored let binding. This is only allowed if the binding is in a certain form; in particular, it may not be a let expression. If a let expression appears at the outermost level of the right-hand-side of a let binding, it therefore has to be moved out of the way.

The two reduction rules that perform let lifting are lift app and lift let. They are formalized by means of the functions *LiftApp* and *LiftLet*. The function *LiftApp* does not require additional context information, but *LiftLet* requires the index of the let binding to be lifted for reasons of disambiguation.

Definition 4.3:1: *(the realization of the* lift-app*-rule)*
The function $LiftApp : \mathcal{E} \to \mathcal{E}$ is defined by:

$$LiftApp(e) = \begin{cases} \mathsf{let}\ xs = es\ \mathsf{in}\ (\mathsf{app}\ e''\ \mathsf{to}\ x) & \textbf{if } e = (\mathsf{app}\ e'\ \mathsf{to}\ x) \\ & \quad \wedge\ e' = (\mathsf{let}\ xs = es\ \mathsf{in}\ e'') \\ e & \textbf{otherwise} \end{cases}$$

Definition 4.3:2: *(the realization of the* lift-let*-rule)*
The function $LiftLet : \mathbb{N} \times \mathcal{E} \to \mathcal{E}$ is defined by:

$$LiftLet(i, e) = \begin{cases} \mathsf{let}\ \langle xs_1 : ys : x_i : xs_2 \rangle & \textbf{if } e = (\mathsf{let}\ \langle xs_1 : x_i : xs_2 \rangle = \\ \quad = \langle as_1 : bs : b : as_2 \rangle & \qquad\qquad \langle as_1 : a_i : as_2 \rangle\ \mathsf{in}\ a \\ \quad \mathsf{in}\ a & \quad \wedge\ \#xs_1 = \#as_1 = i - 1 \\ & \quad \wedge\ a_i = (\mathsf{let}\ ys = bs\ \mathsf{in}\ b) \\ e & \textbf{otherwise} \end{cases}$$

Note that *LiftLet* joins two let expressions into a single new one. The argument i determines which inner let should be lifted. It is required, because multiple inner bindings may be a let itself. The bindings of the inner let are inserted in the outer let just before the original binding. This ensures that the order in which inner lets are lifted does not matter; the result will always be the same.

Example: *(example of the* lift-app*-rule)*
In Section 4.1, the following example of a hindering let was given:
'app (let $\langle x \rangle = \langle e \rangle$ in (fun f on $\langle x \rangle$)) to y'. *(arity of f is 2)*
By applying *LiftApp*, this expression can now be transformed to:
'let $\langle x \rangle = \langle e \rangle$ in (app (fun f on $\langle x \rangle$) to y)'.
Reduction can now continue on the inner let by means of a collect.

Example: *(example of the* lift-let*-rule)*
In the following expression, both the inner lets can be lifted:
'let $\langle x : y \rangle = \langle$let $xs = as$ in $a :$ let $ys = bs$ in $b \rangle$ in e'.
Lifting the second inner let (using *LiftLet* on index 2) leads to:
'let $\langle x : ys : y \rangle = \langle$let $xs = as$ in $a : bs : b \rangle$ in e'.
Lifting the remaining inner let (using *LiftLet* on index 1) leads to:
'let $\langle xs : x : ys : y \rangle = \langle as : a : bs : b \rangle$ in e'.
First lifting index 1 and then index 2 would have given the same result.

4.4 The reduction rule for unsharing

The last remaining task for which a reduction rule has to be defined is the task of *unsharing*. This is the process of replacing variables with the expressions that they are associated with by means of a let binding. We will model one single unshare at a time. Note that cyclic let definitions are allowed; therefore, the process of repeated unsharing does not always terminate. A single unshare, however, always terminates.

Because efficiency is important even when building proofs, we do not allow duplication of unfinished computations. Therefore, an expression may only be unshared if it can statically be determined that it does not contain any redexes. In our language, this is only the case for *partial applications*. Chains of variables ($x = y, y = \ldots$) cannot be unshared immediately. Instead, the final binding has to be reduced to a partial application first, after which the chain can be collapsed.

The rule for unsharing is called unshare, and its function is *Unshare*. The function can only be applied to a variable, and takes the binding as additional input. It is assumed that the binding occurs in the context of the redex.

Definition 4.4.1: *(the realization of the* unshare*-rule)*
The function $Unshare : \mathcal{E} \times \mathcal{E} \to \mathcal{E}$ is defined by:

$$Unshare(x, u, e) = \begin{cases} u & \textbf{if } e = (\mathsf{var}\ x) \wedge u = (\mathsf{fun}\ f\ \mathsf{on}\ xs) \\ & \quad \wedge\ Arity(f) < \#xs \\ e & \textbf{otherwise} \end{cases}$$

Note that this unshare can replace a variable x with any expression u that it is given as additional argument. On this level, there is no verification that $x = u$ actually appears in the context of the redex. This verification is performed later, on the level of inner reduction (see Section 4.7).

4.5 Head reduction

Head reduction is the combination of the five reduction functions defined in the previous sections. It operates on a *rule selector* and an expression. Based on the rule selector, one of the five reduction functions is selected, which is then applied to the expression. A *rule selector* is an artificial identifier that denotes one of the five reduction rules. For simplicity, we incorporate the additional input arguments of the individual rules into the rule selectors defined below:

Definition 4.5:1: *(set of rule selectors)*
The set $\mathcal{R}$ of rule selectors is defined by:

$$\begin{aligned}\mathcal{R} = \{&\mathsf{collect}, \mathsf{lift\ app}\}\\ \cup\ \{&\mathsf{expand}\ xs \quad | \ xs \in \langle\mathcal{V}\rangle\}\\ \cup\ \{&\mathsf{lift\ bind}\ i \quad | \ i \in \mathbb{N}\}\\ \cup\ \{&\mathsf{unshare}\ x\ \mathsf{to}\ u \mid x \in \mathcal{V}, u \in \mathcal{E}\}\end{aligned}$$

The head reduction function is simply a case distinction on the rule selector:

Definition 4.5:2: *(head reduction)*
The function $\mathit{HeadReduce} : \mathcal{R} \times \mathcal{E} \to \mathcal{E}$ is defined by:

$$\begin{aligned}\mathit{HeadReduce}(\mathsf{collect}, &\quad e) &&= \mathit{Collect}(e)\\ \mathit{HeadReduce}(\mathsf{expand}\ xs, &\quad e) &&= \mathit{Expand}(xs, e)\\ \mathit{HeadReduce}(\mathsf{lift\ app}, &\quad e) &&= \mathit{LiftApp}(e)\\ \mathit{HeadReduce}(\mathsf{lift\ bind}\ i, &\quad e) &&= \mathit{LiftLet}(i, e)\\ \mathit{HeadReduce}(\mathsf{unshare}\ x\ \mathsf{to}\ u, &\ e) &&= \mathit{Unshare}(x, u, e)\end{aligned}$$

A summary of the total system of reduction rules is given in Table 1.

4.6 Locations

All the reduction functions that have been defined so far can only be applied to the head of an expression. In order to lift these function to inner reduction, we will use the concept of *locations*. A location is an artificial identifier that points to a specific subexpression within a compound expression. The basic operations on locations are *Get* and *Set*. For a full formalization of locations we refer to the technical report [9]. Here, we introduce locations informally only:

Notation 4.6:1: *(locations and operations on locations)*
Let $\mathcal{L}$ denote the set of available locations, $\mathit{Get} : \mathcal{L} \times \mathcal{E} \hookrightarrow \mathcal{E}$ the function that gets the subexpression from an indicated location, and $\mathit{Set} : \mathcal{L} \times \mathcal{E} \times \mathcal{E} \hookrightarrow \mathcal{E}$ the function that sets the subexpression at an indicated location.

Note that both *Get* and *Set* are partial functions; they fail when the location is not valid within the indicated expression.

name	rule	conditions
collect	$\frac{\mathsf{app}\ (\mathsf{fun}\ f\ \mathsf{on}\ xs)\ \mathsf{to}\ x}{\mathsf{fun}\ f\ \mathsf{on}\ \langle xs : x\rangle}$	$Arity(f) > \#xs$
expand ys	$\frac{\mathsf{fun}\ f\ \mathsf{on}\ xs}{Body(xs, ys, f)}$	$Arity(f) = \#xs$
lift app	$\frac{\mathsf{app}\ (\mathsf{let}\ xs = es\ \mathsf{in}\ e)\ \mathsf{to}\ x}{\mathsf{let}\ xs = es\ \mathsf{in}\ (\mathsf{app}\ e\ \mathsf{to}\ x)}$	–
lift bind i	$\frac{\mathsf{let}\ \langle x_1 \ldots x_n\rangle = \langle e_1 \ldots e_n\rangle\ \mathsf{in}\ e}{\mathsf{let}\ \langle x_1 \ldots x_{i-1} : ys : x_i : x_{i+1} \ldots x_n\rangle = \langle e_1 \ldots e_{i-1} : as : a : a_{i+1} \ldots a_n\rangle\ \mathsf{in}\ \mathrm{e}}$	$1 \leq i \leq n$, $e_i = (\mathsf{let}\ ys = as\ \mathsf{in}\ a)$
unshare x to u	$\frac{\mathsf{var}\ x}{u}$	$u = (\mathsf{fun}\ f\ \mathsf{on}\ xs)$, $Arity(f) < \#xs$

Table 1. The reduction system as a whole

4.7 Inner reduction

The final step in defining our custom reduction system is the upgrade of head reduction to *inner* reduction, which allows reduction to take place on an arbitrary redex. Inner reduction is represented by a function that operates on a location, a rule selector and an expression. It selects the redex at the indicated location, and applies head reduction to it using the given rule selector as argument.

Inner reduction performs partial verification of the incoming rule selector as well. It checks two conditions, namely: (1) whether the variables of an expand are indeed fresh with respect to the expression that is reduced; and (2) whether the binding of an unshare is indeed available in the context of the redex. These conditions are checked using a combination of the redex location and the expression as a whole. The other reduction functions operate on the redex alone, and can therefore not perform these verifications themselves.

The verification of the freshness of an expand-rule is formalized by means of the relation *Fresh*. It simply extracts the variables from the rule and checks whether there is an overlap with the bound variables of the expression.

Definition 4.7.1: *(verification of an* expand*-rule)*
The relation $Fresh \subseteq \mathcal{R} \times \mathcal{E}$ is defined by:

$$Fresh(r, e) \Leftrightarrow \forall_{xs \in \langle \mathcal{V} \rangle}[r = (\mathsf{expand}\ xs) \Rightarrow \neg\exists_{x \in \mathcal{V}}[x \in xs \wedge x \in Bound(e)]]$$

The verification of an unshare-rule is formalized in two steps. First, an auxiliary function *Defs* is defined which collects *all* let bindings within an expression. Then, the relation *Occurs* extracts the binding from an unshare-rule and checks whether it is an element of *Defs*. Because reduction is only allowed on wellformed expressions (i.e. they must be closed and they must have unique variables), being an element of *Defs* automatically ensures the validity of a let binding.

Definition 4.7:2: *(let bindings within an expression)*
The function $Defs : \mathcal{E} \to \wp(\mathcal{V} \times \mathcal{E})$ is defined recursively by:

$$\begin{aligned}
&Defs(\mathsf{var}\ x) &&= \varnothing \\
&Defs(\mathsf{fun}\ f\ \mathsf{on}\ xs) &&= \varnothing \\
&Defs(\mathsf{app}\ e\ \mathsf{to}\ x) &&= Defs(e) \\
&Defs(\mathsf{let}\langle x_1 \ldots x_n\rangle = \langle e_1 \ldots e_n\rangle\ \mathsf{in}\ e) &&= \cup_{i=1}^{n}[\{(x_i, e_i)\} \cup Defs(e_i)] \cup Defs(e)
\end{aligned}$$

Definition 4.7:3: *(verification of an* unshare*-rule)*
The relation $Occurs \subseteq \mathcal{R} \times \mathcal{E}$ is defined by:

$$Occurs(r, e) \Leftrightarrow \forall_{x \in \mathcal{V}} \forall_{u \in \mathcal{E}} [r = (\mathsf{unshare}\ x\ \mathsf{to}\ u) \Rightarrow (x, u) \in Defs(e)]$$

The verification of a rule selector can now be formalized by means of the relation *Valid*, which is simply a conjunction of *Fresh* and *Occurs*:

Definition 4.7:4: *(verification of a rule selector)*
The relation $Valid \subseteq \mathcal{R} \times \mathcal{E}$ is defined by:

$$Valid(r, e) \Leftrightarrow Fresh(r, e) \wedge Occurs(r, e)$$

Inner reduction is formalized by means of the total function *InnerReduce*. This function acts as the identity if the input arguments are not wellformed, or the reduction rule cannot be applied successfully. The input is wellformed if: (1) the location is valid; (2) the rule selector is valid; (3) the expression is closed; and (4) the bound variables within the expression are unique. The explicit conditions (3) and (4) restrict reduction to wellformed expressions only.

Definition 4.7:5: *(inner reduction)*
The function $InnerReduce : \mathcal{L} \times \mathcal{R} \times \mathcal{E} \to \mathcal{E}$ is defined by:

$$InnerReduce(l, r, e) = \begin{cases} Set(l, HeadReduce(r, e'), e) \\ \qquad \textbf{if}\ Get(l, e) = e' \wedge Valid(r, e) \\ \qquad \wedge\ Vars(e) = \varnothing \wedge Unq(Bound(e)) \\ e \quad \textbf{otherwise} \end{cases}$$

Note that the result of reduction is always a wellformed expression itself. This property can be verified easily; therefore, its proof is omitted here.

5 Correctness of let lifting

Our system is non-standard only in the handling of sharing. Other than that, it can be regarded as a simplification of a single-step version of [16]. It is easy to see, however, that our approach with let lifting is equivalent to the standard approach which makes use of external environments:

- Suppose that R is our reduction system, and that R' is obtained out of R by replacing the let-lifting mechanism with a usual external environment mechanism. That is, R' is obtained out of R by:
  - leaving out the rules lift app and lift let;
  - introducing external environments $\Gamma \subseteq \mathcal{V} \times \mathcal{E}$;

  - changing the signature of reduction from $\mathcal{E} \to \mathcal{E}$ to $\Gamma \times \mathcal{E} \to \Gamma \times \mathcal{E}$;
  - adding a rule introduce let that removes a let expression and moves the let bindings in the external environment; and
  - altering the rule unshare to use the external environment.
- Then, all reduction paths of R' can be transformed to R by:
  - leaving out external environments and all applications of introduce let;
  - inserting as many lift app's before each application of collect as there are inner lets in the application node;
  - inserting as many lift let's before each application of unshare as there are inner lets in the binding to be unshared; and
  - augmenting each unshare with the let binding used.

This simple algorithm maps any traditional reduction path into an equivalent reduction path in our system. Because R' can be considered as an extension of Launchbury's system, this means that all reduction paths of Launchbury have an equivalent in our system. The reverse does not hold, however, because our paths do not always choose the left-most outer-most redex, and do not always end with a normal form. Due to confluence (see next section), however, the paths in our system that cannot be converted to Launchbury's system are equivalent to the paths that can be converted.

6 Confluence

Confluence is a well-known property of rewrite systems. It is important for our system, because it ensures that all possible reductions preserve the meaning of an expression, and can therefore safely be applied in the context of reasoning.

In our reduction system, confluence only holds *modulo α-conversion*, because no explicit α-conversion rule is available. Therefore, if two expands are carried out on the same redex, or two expands are carried out on different redexes but there is an overlap in the variables that they introduce, then the reduction results cannot be brought together. This precondition of confluence is formalized by the relation *Joinable*. Furthermore, *Joinable* also excludes the irrelevant and trivial case that the two reductions are identical.

Definition 6.1: *(precondition of confluence)*
The relation $Joinable \subseteq \mathcal{L} \times \mathcal{R} \times \mathcal{L} \times \mathcal{R}$ is defined by:

$$\begin{aligned} Joinable(l_1, r_1, l_2, r_2) \Leftrightarrow \neg(l_1 = l_2 \wedge r_1 = r_2) \\ \wedge\ \forall_{xs,ys \in \langle \mathcal{V} \rangle}[(r_1 = \mathsf{expand}\ xs \wedge r_2 = \mathsf{expand}\ ys) \Rightarrow \\ (l_1 \neq l_2 \wedge \neg\exists_{x \in \mathcal{V}}[x \in xs \wedge x \in ys])] \end{aligned}$$

Below we present the proofs of confluence, which are built incrementally. First, we prove confluence for two single head steps, then for one head step and one inner step, and then finally for two inner steps. Without loss of generality, we present simplified proofs and abstract from wellformedness altogether.

Lemma 6.2: *(confluence - head/head version)*

$$\begin{aligned} \forall_{e \in \mathcal{E}} \forall_{r_1, r_2 \in \mathcal{R}}[Joinable(\langle\rangle, r_1, \langle\rangle, r_2) \\ \Rightarrow \exists_{r_3, r_4 \in \mathcal{R}}[HeadReduce(r_3, HeadReduce(r_1, e)) = \\ HeadReduce(r_4, HeadReduce(r_2, e))]] \end{aligned}$$

Proof:

Assume $e \in \mathcal{E}$, $r_1, r_2 \in \mathcal{R}$ and **[1]**$Joinable(\langle\rangle, r_1, \langle\rangle, r_2)$.
As can be seen in Table 1, on each kind of expression there is only one kind of reduction rule available. Therefore, r_1 and r_2 must be of the same kind.
Due to assumption [1], r_1 and r_2 cannot be the same and cannot be expand's.
Therefore, r_1 and r_2 can only be different applications of lift bind:
Assume **[2]**$r_1 = (\mathsf{lift\ bind}\ i)$, **[3]**$r_2 = (\mathsf{lift\ bind}\ j)$, **[4]**$i \neq j$.
[5]$e = (\mathsf{let}\ xs = bs\ \mathsf{in}\ e_1)$,
[6]$1 \leq i < j$ (if $i > j$ then simply swap them),
[7]$xs = \langle xs_1 : x_i : xs_2 : x_j : xs_3 \rangle$ (with $\#xs_1 = i$-1 and $\#xs_2 = j$-i-1),
[8]$bs = \langle bs_1 : b_i : bs_2 : b_j : bs_3 \rangle$ (with $\#bs_1 = i$-1 and $\#bs_2 = j$-i-1),
[9]$b_i = (\mathsf{let}\ ys = gs\ \mathsf{in}\ g)$ and **[10]**$b_j = (\mathsf{let}\ zs = hs\ \mathsf{in}\ h)$.
The basic idea is that the let lifts can simply be swapped. However, the index of the binding in r_3 has to be increased, because the let lift performed by r_1 has pushed additional bindings upwards. This is not necessary in the reverse case, because the lift of j takes place behind the lift of i.
Choose **[11]**$r_3 = (\mathsf{lift\ bind}\ j + \#ys)$ and **[12]**$r_4 = (\mathsf{lift\ bind}\ i)$.
Now, using *HR* as abbreviation for *HeadReduce*, the following holds:

$HR(r_3, HR(r_1, e))$ **{2,5}**
$= HR(r_3, HR(\mathsf{lift\ bind}\ i, \mathsf{let}\ xs = bs\ \mathsf{in}\ e_1))$ **{11,HR,7,8,9}**
$= HR(\mathsf{lift\ bind}\ j{+}\#ys, \mathsf{let}\ \langle xs_1 : ys : x_i : xs_2 : x_j : xs_3 \rangle$ **{12,HR}**
$\qquad = \langle bs_1 : gs : g : bs_2 : b_j : bs_3 \rangle \mathsf{in}\ e_1)$
$= (\mathsf{let}\ \langle xs_1 : ys : x_i : xs_2 : zs : x_j : xs_3 \rangle = \langle bs_1 : gs : g : bs_2 : hs : h : bs_3 \rangle\ \mathsf{in}\ e_1)$.

Again using *HR* as abbreviation for *HeadReduce*, the following also holds:

$HR(r_4, HR(r_2, e))$ **{3,6}**
$= HR(r_4, HR(\mathsf{lift\ bind}\ j, \mathsf{let}\ xs = bs\ \mathsf{in}\ e_1))$ **{12,HR,7,8,10}**
$= HR(\mathsf{lift\ bind}\ i, \mathsf{let}\ \langle xs_1 : x_i : xs_2 : zs : x_j : xs_3 \rangle$ **{11,HR}**
$\qquad = \langle bs_1 : b_i : bs_2 : hs : h : bs_3 \rangle\ \mathsf{in}\ e_1)$
$= (\mathsf{let}\ \langle xs_1 : ys : x_i : xs_2 : zs : x_j : xs_3 \rangle = \langle bs_1 : gs : g : bs_2 : hs : h : bs_3 \rangle\ \mathsf{in}\ e_1)$.

Therefore, $HR(r_3, HR(r_1, e)) = HR(r_4, HR(r_2, e))$. **QED.**

Lemma 6.3: *(confluence - head/inner version)*

$\forall_{e \in \mathcal{E}} \forall_{r_1, r_2 \in \mathcal{R}} \forall_{l \in \mathcal{L}} [Joinable(\langle\rangle, r_1, l, r_2)$
$\Rightarrow \exists_{r_3, r_4 \in \mathcal{R}} \exists_{l' \in \mathcal{L}} [InnerReduce(l', r_3, HeadReduce(r_1, e)) =$
$HeadReduce(r_4, InnerReduce(l, r_2, e))]]$

Proof:

Assume $e \in \mathcal{E}$, $r_1, r_2 \in \mathcal{R}$, $l \in \mathcal{L}$ and $Joinable(\langle\rangle, r_1, l, r_2)$.
If $l = \langle\rangle$, then the previous Lemma can simply be applied.
If l occurs within a free expression variable of the left-hand-side pattern of r_1 (i.e. no overlap with r_1), then the following arguments hold:

- *Rule r_2 on a modified l_2' is applicable on HeadReduce(r_1, e).*
  All expression variables that are used in the left-hand-side of a reduction rule occur unchanged in the right-hand-side. In other words: r_1 moves the redex of r_2 around, but does not change it.
- *Rule r_1 is applicable at the head of e_2.*
  The reduction r_2 only changes the contents of an expression variable in

the left-hand-side pattern of r_1. If r_1 matches on e, it therefore also syntactically matches (at the head) on e_2. Furthermore, note that it is not possible that the conditions of r_1 are falsified by r_2, or vice versa.

- *The reductions r_1 and r_2 can be swapped, without changing the result.* This follows from the two arguments above.

This only leaves a partial overlap between r_1 and r_2 to be considered. An inspection of Table 1 reveals that there are two such cases: either r_1 is a 'lift app' and r_2 is a 'lift bind'; or r_1 is a 'lift bind' and r_2 is an inner 'lift bind'. In both cases, r_1 and r_2 can be swapped, similarly to Lemma 6.2. The full proof is omitted here, but it can be found in [9]. **QED.**

Theorem 6.4: *(confluence)*

$$\forall_{e\in\mathcal{E}}\forall_{r_1,r_2\in\mathcal{R}}\forall_{l_1,l_2\in\mathcal{L}}[\mathit{Joinable}(l_1,r_1,l_2,r_2) \Rightarrow \exists_{r_3,r_4\in\mathcal{R}}\exists_{l_1',l_2'\in\mathcal{L}}[\mathit{InnerReduce}(l_1',r_3,\mathit{InnerReduce}(l_1,r_1,e)) = \mathit{InnerReduce}(l_2',r_4,\mathit{InnerReduce}(l_2,r_2,e))]]$$

Proof:

Assume $e \in \mathcal{E}$, $r_1, r_2 \in \mathcal{R}$, $l_1, l_2 \in \mathcal{L}$ and $\mathit{Joinable}(l_1, r_1, l_2, r_2)$.
Assume that l_1 is at least as close to the root of e as l_2. If otherwise, then simply swap l_1 and l_2. We distinguish two cases:

- Case 1: *l_2 is a sublocation of l_1.* Now, r_1 is a head reduction of $\mathit{Get}(l_1, e)$, and r_2 is an inner reduction of $\mathit{Get}(l_1, e)$. By applying Lemma 6.3, r_1 and r_2 can be brought together in the context of $\mathit{Get}(l_1, e)$. Because a reduction of a subexpression is always also a reduction of the expression as a whole, r_1 and r_2 can be brought together in the context of e as well.
- Case 2: *l_2 is not a sublocation of l_1.* In this case, r_1 and r_2 are completely disjoint. Their redex transformations therefore do not interfere with each other at all, and can be swapped leading to the same single result. **QED**.

7 Related work

Our reduction system is based on reduction as proposed by Launchbury in [16], which has since 1993 been used as the de facto standard for evaluating lazy functional programs. Several systems have been derived from Launchbury's, but none that we know of leaves the choice of redex free. Derived systems of interest are [4], which defines an operational semantics specifically for Clean, and [12], which defines a *single-step* reduction system for parallel evaluation. Both systems fix a single redex, however, and are therefore less suited for formal reasoning.

In [1] the authors describe a single-step reduction system based on a call-by-need extension of the lambda calculus, which fully supports lazy evaluation and sharing. It is both single-step and it leaves the choice of redex free. The disadvantage of this system, however, is the syntactical distance between the lambda calculus and (the core of) a lazy functional programming language. This distance is most apparent in the representation of functions and applications. Due to this distance, the system of [1] is not suited for *dedicated* formal reasoning on the level of the program, which is one of the trademark features of Sparkle.

Related more generally is the ρ_g-Calculus[2], which integrates term-rewriting with lambda-calculus, expressing sharing and cycles. It uses both unification and

matching constraints, leading to a term-graph representation in an equational style. This calculus is more general than classical term graph rewriting[18, 3], which can be simulated in it. We feel that our work can serve as a first basis for creating a reduction system for a proof assistant based on the ρ_g-calculus.

Another issue of future work concerns the addition of tactical support for the equivalence of cyclic graphs. The work of [11], which establishes the bisimilarity of different proof systems for equational cyclic graph specifications, is expected to be valuable input for this future work.

8 Conclusions

We have defined a term-graph reduction system for a simplified lazy functional language. Our system uses single-step reduction and leaves the choice of redex free. This offers a degree of flexibility that is not available in the commonly used reduction systems for functional languages. Due to this degree of flexibility, our system is much better suited for the foundation of formal reasoning. Our reduction system is used in the foundation of Sparkle, Clean's proof assistant.

Our system maintains sharing within expressions and does not use external environments. This offers the advantage of orthogonality: expressions can be given a meaning as they are, whereas in the common reduction systems they have to be combined with an environment first. The internal maintenance of sharing does not make the reduction system more complicated; it suffices to add two additional rules for let-lifting . All in all, our system consists of five reduction rules only, and is very simple.

All common reduction paths can be expressed in our system. Furthermore, we have proved that our system is confluent. This implies that our system is equivalent to the standard systems: there is at least one reduction path that corresponds to normal reduction, and all other paths can be converged to it.

References

1. Z. M. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler. A call-by-need lambda calculus. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246, New York, NY, USA, 1995. ACM Press.
2. P. Baldan, C. Bertolissi, H. Cirstea, and C. Kirchner. A rewriting calculus for cyclic higher-order term graphs. *Mathematical Structures in Computer Science*, 17(3):363–406, 2007.
3. H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE (2)*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer, 1987.
4. E. Barendsen and S. Smetsers. Graph rewriting aspects of functional programming. In *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 63–102. World Scientific, 1999.
5. L. Brim, B. Houverkort, M. Leucker, and J. van de Pol, editors. *Formal Methods in Industrial Critical Systems, 11th International Workshop, FMICS 2006. Bonn, Germany, August 2006, Revisited Selected Papers*, volume 4346 of *LNCS*. Springer, 2006.

6. A. Butterfield and G. Strong. Proving Correctness of Programs with I/O - a paradigm comparison. In T. Arts and M. Mohnen, editors, *Selected Papers from the 13th International Workshop on Implementation of Functional Languages, IFL 2001*, volume 2312 of *LNCS*, pages 72–88, Stockholm, Sweden, 2001. Springer.
7. M. de Mol and M. van Eekelen. A proof tool dedicated to clean - the first prototype. In *Proceedings of Applications of Graph Transformations with Industrial Relevance 1999*, volume 1779 of *LNCS*, pages 271–278. Springer Verlag, 1999. ISBN 3-540-67658-9.
8. M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In T. Arts and M. Mohnen, editors, *The 13th International Workshop on Implementation of Functional Languages, IFL 2001, Selected Papers*, volume 2312 of *LNCS*, pages 55–72, Stockholm, Sweden, 2002. Springer.
9. M. de Mol, M. van Eekelen, and R. Plasmeijer. Proving confluence of term-graph reduction for sparkle. Technical Report ICIS-R07012, Institute for Computing and Information Sciences, 2007.
10. M. Dowse, A. Butterfield, and M. C. J. D. van Eekelen. Reasoning About Deterministic Concurrent Functional I/O. In C. Grelck, F. Huch, G. Michaelson, and P. W. Trinder, editors, *IFL*, volume 3474 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2004.
11. C. Grabmayer. A duality between proof systems for cyclic term graphs. *Mathematical Structures in Computer Science*, 17(3):439–484, 2007.
12. J. G. Hall, C. A. Baker-Finch, P. W. Trinder, and D. J. King. Towards an operational semantics for a parallel non-strict functional language. In K. Hammond, A. J. T. Davie, and C. Clack, editors, *IFL*, volume 1595 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 1998.
13. Z. Horváth, T. Kozsik, and M. Tejfel. Proving invariants of functional programs. In P. Kilpeläinen and N. Päivinen, editors, *SPLST*, pages 115–126. University of Kuopio, Department of Computer Science, 2003.
14. P. Hudak, S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. *SIGPLAN Notices*, 27(5):R1–R164, 1992.
15. Kesteren, R. van and Eekelen, M. van and Mol, M. de. Proof support for general type classes. In H.-W. Loidl, editor, *Trends in Functional Programming 5: Selected papers from the Fifth International Symposium on Trends in Functional Programming, TFP04*, pages 1–16. München, Germany, Intellect, 2004.
16. J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.
17. A. Moran. Functional Programming in the Real World - Report on the First Commercial Users of Functional Programming Workshop. *SIGPLAN Not.*, 39(12), 2004.
18. M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term graph rewriting: theory and practice*. John Wiley and Sons Ltd., Chichester, UK, 1993.
19. M. Tejfel, Z. Horváth, and T. Kozsik. Extending the sparkle core language with object abstraction. *Acta Cybern.*, 17(2), 2006.
20. M. van Eekelen and R. Plasmeijer. Concurrent CLEAN language report (version 1.3). Technical Report CSI-R9816, Computing Science Institute Nijmegen, June 1998. http://www.cs.ru.nl/∼clean.