Confluent Term-Graph Reduction for Computer-Aided Formal Reasoning

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. SPARKLE is the dedicated proof assistant for the lazy functional programming language CLEAN, which is based on term-graph rewriting. Because equational reasoning is an important proof technique, SPARKLE needs to offer support for formally proving $e_1 = e_2$ in as many situations as possible. The base proof of equality is by means of reduction: if e_1 reduces to e_2, then also $e_1 = e_2$. Therefore, the underlying reduction system of SPARKLE needs to be geared towards *flexibility*, allowing reduction to take place as often as possible.
In this paper, we will define a flexible term-graph reduction system for a simplified lazy functional language. Our system leaves the choice of redex free and uses single-step reduction. It is therefore more suited for formal reasoning than the well-established standard reduction systems for lazy functional languages, which usually fix a single redex and/or realize multi-step reduction only. We will show that our 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. Because it is based on a simplified functional language, it can be applied to any other functional language based on term-graph rewriting as well.

1 Introduction

CLEAN [1] and HASKELL [2] are lazy functional programming languages that are based on (term-)graph rewriting. In 2001, the distribution of CLEAN was extended with the dedicated proof assistant SPARKLE [3]. With this new tool, it became possible to reason about lazy functional programs, and to formally prove logic properties of these programs.

Since then, SPARKLE has been used in practice for various purposes. It has been used for proving properties of I/O-programs by Dowse[4] and Butterfield[5]. An extension for dealing with temporal properties has been proposed for it by Tejfel, Horvth and Koszik[6]. 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[7].

An important proof technique in general is *equational reasoning*, which can be used in SPARKLE because functional programming languages are referentially transparent. Equational reasoning is intuitive and easy to apply, and occurs frequently in proofs about functional programs. Its usefulness, however, depends on the power of the proof techniques that are available for proving equality. In formal proofs, $e_1 = e_2$ needs to be proved explicitly before e_1 may be replaced by e_2. The most basic proof technique for showing equality is reduction: if e_1 reduces to e_2, then e_1 is equal to e_2. Consequently, the underlying reduction system has an important influence on equational reasoning. The more *flexible* it is, the more expressions reduce to each other, and the more basic equalities can be proved for free.

1.1 Other reduction systems (related work)

There are many possible ways to describe a reduction system for a lazy functional programming language. Since 1993, however, the system presented by Launchbury in [8] has been used as the de-facto standard for lazy functional reduction. Launchbury's description is geared towards HASKELL. An adaptation for CLEAN has been presented in [9].

Launchbury's system is geared towards evaluation and aims to reduce an expression to its normal form as efficiently as possible. For this purpose, the system fixes a single normalizing strategy: left-most outer-most reduction. The strategy is incorporated into the reduction relation, such that there is at most a single redex in any expression. This makes the reduction system easier, but rigid as well, because it restricts the number of allowed reductions greatly. Therefore, Launchbury's system is not really suited for the foundation of formal reasoning.

It is easy to turn the definition of a 'single-redex system' into the definition of a 'multiple-redex system'. However, to ensure correctness, confluency has to be proved for a multiple-redex system too, and this requires considerable effort. As far as we know, no reduction system that is based on [8] has been adapted to a multiple-redex system and has formally been proved confluent.

Another hindering feature is that Launchbury's system is *multi-step*. In a multi-step system, the individual reduction rules are still single steps, but the reduction relation as a whole is only defined for a collection of steps that lead to a normal form. This restricts the number of possible reductions, but is not an issue at all, because turning a multi-step system into a single-step one is a trivial formality.

Finally, note that there are many other formalisms, such as for instance lambda calculus variations, that can be used to describe lazy functional reduction in. We are interested in *dedicated* formal reasoning, however, and aim to formalize reduction on a level that is as close as possible to the programming language. We are therefore not interested in these other formalisms.

1.2 Our reduction system

In this paper, we will define a custom reduction system for a simplified lazy functional language. Our system is loosely based on [8], but leaves the choice of redex free and makes use of single-step reduction. Therefore, our system is flexible, and better suited for formal reasoning than the standard reduction systems.

We will show that our system is confluent and that the standard lazy functional reduction path is allowed by it. It is therefore equivalent to the standard systems.

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 of the reduction system more closely. We introduce our simplified expression language in Section 3, and describe our reduction system in detail in Section 4. We show how to express standard reduction paths in our system in Section 5, and we prove that our system is confluent in Section 6. Finally, we draw conclusions in Section 7.

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 identified that influence flexibility: the granularity of the reduction relation, and the freedom of choice of 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, it is therefore most natural to make use of a single-step reduction system, in which the reduction relation is defined in terms of applications of single reduction steps.

Regardless of the realization of the reduction system, an expression is always semantically equal to all of its intermediate reducts. When a single-step reduction system is used, however, it is much easier to make use of this equality in a formal proof. This is illustrated by the following example:

Example 2.1.1: *(proof that requires intermediate reducts)*

Assume that the following property has been proved earlier:

‘$\forall_{xs}[\texttt{length}\ (xs\texttt{++}xs) = (\texttt{length}\ xs)\texttt{+}(\texttt{length}\ xs)]$’.

Using this property, the task is to prove that:

‘$\forall_{x,xs}[\texttt{length}\ (([x]\texttt{++}xs)\texttt{++}[x\,{:}\,xs]) = \texttt{length}\ [x\,{:}\,xs]\texttt{+}\texttt{length}\ [x\,{:}\,xs]]'$.

On the intuitive level, this is a trivial proof: first replace $([x]\texttt{++}xs)$ with $[x\,{:}\,xs]$; then, apply the assumed property. QED.

In the single-step case, the first step of the proof can be formalized with ‘reduce $([x]\texttt{++}xs)$ 1’, where the argument ‘1’ denotes a reduction of one step.

In the multi-step case, this formalization cannot be used; the replacement in the first step can only be formalized if the user specifies the target expression '$[x : xs]$' explicitly.
Having to specify target expressions explicitly is cumbersome, because the expressions can be very big, but most of all very annoying, because no explicit identification is needed on the intuitive level. It therefore hinders reasoning.

2.2 Choice of redex

Because lazy functional languages are referentially transparent, it is always safe to apply reduction to an inner redex, regardless of the realization of the reduction system. In a formal context, however, the soundness of the proof assistant itself requires referential transparency to be proved as well! This proof can be constructed in two different ways:

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

Because semantic equality (which needs some kind of bisimulation to cope with infinite reductions) is much more complex than a reduction system, we feel 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 confluency.

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 abstract from constructors and case distinctions, which can be added to our system without difficulties. We do not include lambda expressions, but instead make use of a constant environment of function definitions. We consider sharing to be a basic component of any lazy functional language.

Notation: *(sets of variables and function symbols)*
Assume that $\mathcal{V}$ denotes the set of available variable names.
Assume that $\mathcal{F}$ denotes the set of available function symbols.
Assume that each function symbol has a fixed arity, which can be obtained by means of the function $\mathit{Arity} : \mathcal{F} \rightarrow \mathbb{N}$.

Notation: *(lists)*
Assume that '$\langle$' and '$\rangle$' are used to denote lists.
Assume that $\#xs$ denotes the length (or size) of the list (or set) xs.
Assume that $xs!i$ denotes the i-th element of xs, if it exists.
Assume that $\mathit{Unq}(xs)$ denotes that all elements in xs occur only once.
Assume that $\mathit{Overlap}(xs, ys)$ denotes that there is at least one element of xs that also occurs in ys.

Definition 3:1: *(set of expressions)*
The set $\mathcal{E}$ of expressions is defined recursively by:

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

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.

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 [8]), 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:

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

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, beginning with the individual reduction rules and ending with a formalization for reduction as a whole. Before we begin with the reduction rules, however, we will first describe our non-standard approach to handling sharing in Section 4.1. We do not make use of external environments, but instead store sharing within expressions themselves. This has a profound influence on our reduction system, and we will therefore describe it first.

Our reduction system is very simple and consists of five reduction rules only: two for handling applications (Section 4.2), two for the internal bookkeeping of sharing (Section 4.3), and one for unsharing (Section 4.4). We will formalize each rule by means of a deterministic function that transforms an isolated redex to its reduction result. Any additional information that is required for computing the result is assumed to be available by means of additional input arguments.

The additional arguments for the five reduction functions are not the same. In Section 4.5, we therefore define the union set of *rule selectors*. A rule selector is a combination of an artificial name for a rule and the additional information that it requires. From now on, additional information will always be passed to the reduction functions in terms of a rule selector.

In Section 4.6, we define *head reduction* on top of the individual rules. Head reduction is represented by a function that operates on a rule selector and an isolated redex. Based on the rule selector, it selects a single reduction rule and then applies it on the redex. Head reduction operates on the same level as the individual rules, and transforms an isolated redex only. It therefore requires the same additional information, this time represented by means of a rule selector.

To upgrade head reduction to arbitrary reduction, it must be possible to identify and transform individual subexpressions. For this purpose, we will define *locations* in Section 4.7. A location is an artificial identifier which specifies a path in an expression that leads from the root to a specific subexpression.

Finally, in Section 4.8, we will formalize *inner reduction*. Inner reduction is represented by a function that operates on a redex location, a rule selector and an expression. Based on the location, it extracts a single subexpression from the expression and then applies head reduction to it. We have chosen to provide inner reduction with the same amount of additional information as head reduction, although some of this information can be inferred from the expression as a whole. Instead, we will explicitly validate this duplicate information.

Inner reduction is our top-level representation of the reduction system. We have chosen to represent it by means of a deterministic function, because this makes reasoning about the reduction system (and therefore proving confluency) easier. Deterministic behavior, however, can only be achieved by assuming that additional information (most notably, information about which fresh variables to introduce) is created externally and is made available as input.

4.1 Graphs as self-contained expressions

Sharing is handled in our reduction system in a way that is not standard. We do not make use of 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 make use of 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 having to pairing them to an external object first. This makes handling expressions more transparent, and makes subsequent definitions and proofs much easier.

The disadvantage of our method is that reduction has to offer functionality for maintaining let definitions internally. Two specific 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: 'app (let $\langle x\rangle = \langle e\rangle$ in (fun f on $\langle x\rangle$)) to y'. *(arity of f is 2)*
  Basically, this expression is a partial application of f that itself is applied again to y. Reduction should contract the two applications, adding y to the argument list in the fun-node, after which f can be expanded.
  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, after which the required syntactic match would be achieved.
  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: 'let $\langle x\rangle = \langle e\rangle$ in (app (var x) to y)'. *(assume that e is in nf)*
  In this example, reduction should replace the inner 'var x' with e, because x is bound to e by the outer let expression. This requires the inner reduction of 'var x' to know about the external binding of x to e.
  Normally, reduction of the expression as a whole would remove the leading let, and introduce $x = e$ into an external environment. Then, the subsequent reduction of 'var x' would take this environment as additional argument, and the needed information would be available.
  Because we do not make use of external environments, we have to come up with 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

Applications are handled by means of the reduction rules collect and expand:

- 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 much 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.

These two reduction rules are realized by means of the following functions:

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

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

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

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

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, however, that lets may get in the way of reduction. When reduction requires a subexpression to be brought into a certain form, it is possible that a let is created on the outer level of that subexpression. For reduction to continue, it must be possible to remove this hindering let.

Fortunately, a solution to this problem presents itself readily. The 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 without problems on all relevant places. 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 simple system, there are only two places where it must be possible to lift a let 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 $\mathit{LiftApp} : \mathcal{E} \rightarrow \mathcal{E}$ is defined by:

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

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

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

Note that *LiftLet* actually joins two let expressions into a single new one. The argument i determines which inner let should be lifted. It is required, because more than one of the inner let 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, an example was given of an expression in which an inner let hinders the continuation of reduction:
‘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’.

The same result would be obtained when the let lifts are swapped.

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 will not allow the 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*. Note that chains of variables (lists of bindings of the form $x = y, y = z, z = a, a = \ldots$) cannot be unshared immediately. Instead, the final binding of the chain has to be reduced to a partial application first, after which the chain as a whole can be collapsed.

The reduction rule for unsharing is called unshare, and its associated function is *Unshare*. The function can only be applied to a variable expression, and takes the binding as additional context information. 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 $\mathit{Unshare} : \mathcal{E} \times \mathcal{E} \rightarrow \mathcal{E}$ is defined by:

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

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 4.8).

4.5 Rule selectors

A rule selector is an artificial identifier for a reduction rule, combined with the additional information that it requires for the purpose of realizing deterministic behavior. In the previous sections, this information has already been identified for each individual rule:

- The rules collect and lift app do not require any additional information.

- The rule expand requires a list of variables that can be used to create a fresh instantiation of the function body.
- The rule $\mathsf{lift\ let}$ requires the index of the inner let to be lifted.
- The rule $\mathsf{unshare}$ requires the full let binding that is being unshared. Note that the individual rules operate on the level of the redex only; therefore, this information has to be supplied explicitly.

The set of rule selectors can be formalized as follows:

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

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

4.6 Head reduction

The final preparation for inner reduction is head reduction, which combines the five individual reduction functions into a single function. Head reduction takes a rule selector as additional argument, and decides on the basis of this selector which individual reduction function is to be applied. The additional input that this function requires is retrieved from the selector as well. Head reduction can be formalized as follows:

Definition 4.6:1: *(head reduction)*
The function $\mathit{HeadReduce} : \mathcal{R} \times \mathcal{E} \rightarrow \mathcal{E}$ is defined by:

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

A summary of the total system of reduction rules, presented in the usual style, is given in Table 1.

4.7 Locations

A location is an artificial identifier that points to a specific subexpression within a compound expression. We will represent locations by means of lists of *directions*. Each direction describes how to travel from the outermost expression level to a subexpression that is one level deeper. Using a list of directions, an expression can be traversed recursively until the indicated subexpression is reached.

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

Table 1. The reduction system as a whole

Definition 4.7:1: *(set of directions)*
The set $\mathcal{D}$ of directions within locations is defined by:
$\mathcal{D} = \{\textsf{app node}\}$
$\cup \{\textsf{let bind } i \mid i \in \mathbb{N}\}$
$\cup \{\textsf{let node}\}$

Definition 4.7:2: *(set of locations)*
The set $\mathcal{L}$ of locations within expressions is defined by:
$\mathcal{L} = \langle \mathcal{D} \rangle$

Note that not all locations are valid within a certain expression. A location is only valid if its first direction repeatedly matches with the expression that is being traversed. We will not explicitly formalize validity of locations. Instead, the operations on locations will all represented by partial functions.

The basic operations on locations are *Get*, which gets a subexpression that is indicated by a location, and its complement *Set*, which sets the indicated subexpression. Both operations will be represented by means of partial functions that are defined by means of pattern matching (for pattern matching on lists, the constructors nil and cons are temporarily introduced). The patterns are not exhaustive; instead, it is assumed that the operations fail on those cases that have not been specified.

Definition 4.7:̂3: *(get subexpression of expression)*

The partial function $Get : \mathcal{L} \times \mathcal{E} \hookrightarrow \mathcal{E}$ is defined recursively by:

$$\begin{array}{lll}
Get(\mathsf{nil}, & e) & = e \\
Get(\mathsf{cons\ (app\ node)}\ l, & \mathsf{app}\ e\ \mathsf{to}\ x) & = Get(l, e) \\
Get(\mathsf{cons\ (let\ bind}\ i)\ l, & \mathsf{let}\ xs = es\ \mathsf{in}\ e) & = Get(l, es!i) \\
Get(\mathsf{cons\ (let\ node)}\ l, & \mathsf{let}\ xs = es\ \mathsf{in}\ e) & = Get(l, e)
\end{array}$$

Definition 4.7:̂4: *(set subexpression of expression)*

The partial function $Set : \mathcal{L} \times \mathcal{E} \times \mathcal{E} \hookrightarrow \mathcal{E}$ is defined recursively by:

$$\begin{array}{lll}
Set(\mathsf{nil}, & p,\ e) & = p \\
Set(\mathsf{cons\ (app\ node)}\ l, & p,\ \mathsf{app}\ e\ \mathsf{to}\ x) & = \mathsf{app}\ Set(l, p, e)\ \mathsf{to}\ x \\
Set(\mathsf{cons\ (let\ bind}\ i)\ l, & p,\ \mathsf{let}\ xs = es\ \mathsf{in}\ e) & = \mathsf{let}\ xs = Set'(l, i, p, es)\ \mathsf{in}\ e \\
Set(\mathsf{cons\ (let\ node)}\ l, & p,\ \mathsf{let}\ xs = es\ \mathsf{in}\ e) & = \mathsf{let}\ xs = es\ \mathsf{in}\ Set(l, p, e)
\end{array}$$

Definition 4.7:̂5: *(set subexpression of list of expressions)*

The partial function $Set' : \mathcal{L} \times \mathbb{N} \times \mathcal{E} \times \langle\mathcal{E}\rangle \hookrightarrow \langle\mathcal{E}\rangle$ is defined by:

$$Set'(l,\ i,\ p,\ \langle e_1 \dots e_i \dots e_n\rangle) = \langle e_1 \dots Set(l, p, e_i) \dots e_n\rangle$$

4.8 Inner reduction

The final step in the definition of the 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 the expression as a whole. It selects a redex by means of the location, and then applies head reduction to it with the given rule selector as argument.

Inner reduction not only realizes the selection of an inner redex as added functionality, but also performs partial verification of the incoming rule selector. It checks two conditions, namely: (1) whether the variables of an expand are indeed fresh; and (2) whether the binding of an unshare is indeed available in the context of the redex. These conditions are checked using the expression as a whole. The other reduction functions operate on the redex alone, and can therefore not perform these verifications themselves.

The partial verification is formalized by means of the predicate *Valid*, which operates on the rule selector and the expression as a whole. It assumes that the expression is wellformed (i.e. is closed and has unique variables; see later in this section). Therefore, any binding $x = u$ that can be applied to a subexpression 'var x' is automatically in scope, and it suffices to produce the set of *all* bindings (with the function *Defs*), and then to check whether $x = u$ is an element of it.

Assumption 4.8:̂1: *(variable functions)*

Assume that the function *Vars* produces the *set* of free variables within an expression, and the function *Bound* the *list* of bound variables.

Definition 4.8:2: *(let bindings within an expression)*
The function $\mathit{Defs} : \mathcal{E} \to \wp(\mathcal{V} \times \mathcal{E})$ is defined recursively by:

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

Definition 4.8:3: *(verification of a rule selector)*
The relation $\mathit{Valid} \subseteq \mathcal{R} \times \mathcal{E}$ is defined by:

$$\begin{aligned}
\mathit{Valid}(r, e) \Leftrightarrow\ &\forall_{\mathit{xs} \in \langle \mathcal{V} \rangle}\ \ [r = (\mathsf{expand}\ \mathit{xs}) \qquad \Rightarrow \neg \mathit{Overlap}(\mathit{xs}, \mathit{Bound}(e))]\\
\wedge\ &\forall_{x \in \mathcal{V}} \forall_{u \in \mathcal{E}} [r = (\mathsf{unshare}\ x\ \mathsf{to}\ u) \Rightarrow (x, u) \in \mathit{Defs}(e)]
\end{aligned}$$

The inner reduction function is formalized by means of *InnerReduce*. We have restricted inner reduction to *wellformed* expressions. An expression is wellformed if it is closed (has no free variables), and all the bound variables in it are unique. *InnerReduce* is a total function that returns its input expression if reduction cannot be applied. This is the case when: (1) the location is not valid; or (2) the rule selector is not valid; or (3) the expression is not wellformed; or (4) the rule does not match. Conditions (1),(2) and (3) are enforced explicitly; condition (4) is implied by the behavior of the inner reduction functions.

Definition 4.8:4: *(inner reduction)*
The function $\mathit{InnerReduce} : \mathcal{L} \times \mathcal{R} \times \mathcal{E} \to \mathcal{E}$ is defined by:

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

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 [8]. It is easy to see, however, that our approach with let lifting is equivalent to the standard approach which makes 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; adding a rule introduce let; and altering the rule unshare.
- Then, all reduction paths of R' can be transformed to R by:
  - leaving out all applications of introduce let;

- inserting lift app before each application of expand;
- inserting lift let before each application of unshare;
- augmenting each unshare with the used let binding, which is obtained by inspecting the environment that is input to the reduction step;
- and leaving the rest the same.

Note that it is not a problem that more lift steps are inserted than necessary, because our system returns the input expression when a rule does not match.

A full proof of this observation is unfortunately out of scope for this paper.

6 Confluency

Confluency is a well-known property of rewrite systems, stating that it must always be possible to join divergent reductions of a source expression to a common target expression. It is an important property for our reduction system, because it ensures that all possible reductions preserve the meaning of an expression, and are therefore safe to apply in the context of formal reasoning.

In this section, we will prove that our reduction system is confluent. The proof is constructed incrementally. First, confluency is proved for two single head reduction steps, then for one head reduction step and one inner reduction step, and then finally for two inner reduction steps. Without loss of generality, we present simplified proofs only. We abstract from wellformedness, and leave out reasoning that is similar to reasoning given before.

Lemma 6:1: *(confluency - head/head version)*

$$\forall_{e\in\mathcal{E}}\forall_{r_1,r_2\in\mathcal{R}}[\neg(\mathit{HeadReduce}(r_1,e) =^{\alpha} \mathit{HeadReduce}(r_2,e))$$
$$\Rightarrow \exists_{r_3,r_4\in\mathcal{R}}[\mathit{HeadReduce}(r_3,\mathit{HeadReduce}(r_1,e)) =$$
$$\mathit{HeadReduce}(r_4,\mathit{HeadReduce}(r_2,e))]]$$

Proof:

Assume $e \in \mathcal{E}$, $r_1, r_2 \in \mathcal{R}$ and $[\mathbf{1}]\neg(\mathit{HeadReduce}(r_1,e) =^{\alpha} \mathit{HeadReduce}(r_2,e))$. 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. The result of applying r_1 and r_2 may not be alpha-equal. This excludes 'collect', 'lift app', 'expand' and 'unshare', because they can only be applied in one way. Thus, r_1 and r_2 can only be different applications of 'lift bind:

Assume $[\mathbf{2}]r_1 = (\mathsf{lift\ bind}\ i)$, $[\mathbf{3}]r_2 = (\mathsf{lift\ bind}\ j)$, $[\mathbf{4}]i \neq j$.

$[\mathbf{5}]e = (\mathsf{let}\ xs = bs\ \mathsf{in}\ e_1)$,

$[\mathbf{6}]1 \leq i < j$ (if $i > j$ then simply swap them),

$[\mathbf{7}]xs = \langle xs_1 : x_i : xs_2 : x_j : xs_3\rangle$ (with $\#xs_1 = i\text{-}1$ and $\#xs_2 = j\text{-}i\text{-}1$),

$[\mathbf{8}]bs = \langle bs_1 : b_i : bs_2 : b_j : bs_3\rangle$ (with $\#bs_1 = i\text{-}1$ and $\#bs_2 = j\text{-}i\text{-}1$),

$[\mathbf{9}]b_i = (\mathsf{let}\ ys = gs\ \mathsf{in}\ g)$ and $[\mathbf{10}]b_j = (\mathsf{let}\ zs = hs\ \mathsf{in}\ h)$.

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 $[\mathbf{11}]r_3 = (\mathsf{lift\ bind}\ j + \#ys)$ and $[\mathbf{12}]r_4 = (\mathsf{lift\ bind}\ i)$.

Now, using *HR* as abbreviation for *HeadReduce*, the following holds:
$HR(r_3, HR(r_1, e))$ **{2,5}**
$= HR(r_3, HR(\textsf{lift bind } i, \textsf{let } xs = bs \textsf{ in } e_1))$ **{11,HR,7,8,9}**
$= HR(\textsf{lift bind } j{+}\#ys, \textsf{let } \langle xs_1 : ys : x_i : xs_2 : x_j : xs_3\rangle$ **{12,HR}**
$= \langle bs_1 : gs : g : bs_2 : b_j : bs_3\rangle \textsf{in } e_1)$
$= (\textsf{let } \langle xs_1 : ys : x_i : xs_2 : zs : x_j : xs_3\rangle = \langle bs_1 : gs : g : bs_2 : hs : h : bs_3\rangle \textsf{ in } e_1).$
Again using *HR* as abbreviation for *HeadReduce*, the following also holds:
$HR(r_4, HR(r_2, e))$ **{3,6}**
$= HR(r_4, HR(\textsf{lift bind } j, \textsf{let } xs = bs \textsf{ in } e_1))$ **{12,HR,7,8,10}**
$= HR(\textsf{lift bind } i, \textsf{let } \langle xs_1 : x_i : xs_2 : zs : x_j : xs_3\rangle$ **{11,HR}**
$= \langle bs_1 : b_i : bs_2 : hs : h : bs_3\rangle \textsf{in } e_1)$
$= (\textsf{let } \langle xs_1 : ys : x_i : xs_2 : zs : x_j : xs_3\rangle = \langle bs_1 : gs : g : bs_2 : hs : h : bs_3\rangle \textsf{ in } e_1).$
Therefore, $\mathit{HeadReduce}(r_3, \mathit{HeadReduce}(r_1, e)) =$
$\mathit{HeadReduce}(r_4, \mathit{HeadReduce}(r_2, e)).$

QED.

Lemma 6:2: *(confluency - head/inner version)*

$$\forall_{e \in \mathcal{E}} \forall_{r_1, r_2 \in \mathcal{R}} \forall_{l \in \mathcal{L}} [\neg(\mathit{HeadReduce}(r_1, e) =^{\alpha} \mathit{InnerReduce}(l, r_2, e)) \Rightarrow \exists_{r_3, r_4 \in \mathcal{R}} \exists_{l' \in \mathcal{L}} [\mathit{InnerReduce}(l', r_3, \mathit{HeadReduce}(r_1, e)) = \mathit{HeadReduce}(r_4, \mathit{InnerReduce}(l, r_2, e))]]$$

Proof:

Assume $e \in \mathcal{E}$, $r_1, r_2 \in \mathcal{R}$ and $l \in \mathcal{L}$.
Assume $\neg(\mathit{HeadReduce}(r_1, e) =^{\alpha} \mathit{InnerReduce}(l, r_2, e))$.
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 $\mathit{HeadReduce}(r_1, e)$.*
  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 6:1.

QED.

Theorem 6:3: *(confluency)*

$$\forall_{e \in \mathcal{E}} \forall_{r_1, r_2 \in \mathcal{R}} \forall_{l_1, l_2 \in \mathcal{L}} [\neg(\mathit{InnerReduce}(l_1, r_1, e) =^{\alpha} \mathit{InnerReduce}(l_2, r_2, e)) \Rightarrow \exists_{r_3, r_4 \in \mathcal{R}} \exists_{l'_1, l'_2 \in \mathcal{L}} [\mathit{InnerReduce}(l'_1, r_3, \mathit{InnerReduce}(l_1, r_1, e)) = \mathit{InnerReduce}(l'_2, r_4, \mathit{InnerReduce}(l_2, r_2, e))]]$$

Proof:

Assume $e \in \mathcal{E}$, $r_1, r_2 \in \mathcal{R}$ and $l_1, l_2 \in \mathcal{L}$.
Assume $\neg(\mathit{InnerReduce}(l_1, r_1, e) =^{\alpha} \mathit{InnerReduce}(l_2, r_2, e))$.
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. Now, the following two cases can be distinguished:

- 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 6:2, 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 Conclusions

In this paper, we have defined a term-graph reduction system for a generic 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. Because of this degree of flexibility, our reduction system is much better suited for the foundation of formal reasoning. Our reduction system is used in the foundation of Sparkle, the proof assistant for Clean.

Our system maintains sharing within expressions and does not make use of 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 much more complicated; it suffices to add two additional rules for let-lifting (and no rule is necessary for moving a let into the external environment). All in all, our system consists of five reduction rules only, and is very simple.

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

References

1. M. van Eekelen and R. Plasmeijer. *Concurrent Clean Language Report (version 1.3)*, Nijmegen, June 1998. CSI Technical Report, CSI-R9816. `http://www.cs.kun.nl/~clean/Manuals/manuals.html`

2. S. Peyton Jones(editor), J. Hughes(editor) et al. *Report on the programming language Haskell 98*, Yale, 1999. `http://www.haskell.org/definition/`
3. M. de Mol, M. van Eekelen and R. Plasmeijer, *Theorem Proving for functional Programmers - Sparkle: A Functional Theorem Prover*, Nijmegen, 2001. In selected papers of the 13th International Workshop on Implementation of Functional Languages (IFL 2001), Lecture Notes in Computer Science, Vol. 2312, pages 55-72.
4. M. Dowse, A. Butterfield and M. van Eekelen. *A Language for Reasoning about Concurrent Functional I/O*, Dublin, 2004. In selected papers of the 16th International Workshop on Implementation of Functional Languages (IFL 2004), Lecture Notes in Computer Science, Vol. 3074, pages 177-195.
5. A. Butterfield and G. Strong. *Proving Correctness of Programs with I/O - a paradigm comparison*, Dublin, 2001. In selected papers of the 13th International Workshop on Implementation of Functional Languages (IFL 2001), Lecture Notes in Computer Science, Vol. 2312, pages 72-88.
6. M. Tejfel, Z. Horvth and T. Kozsik, *Extending the Sparkle Core Language with Object Abstraction*, Budapest, 2005. In Acta Cybernetica, volume 17, number 2.
7. R. van Kesteren, M. van Eekelen and M. de Mol. *Proof Support for General Type Classes*, Nijmegen, 2004. In selected papers of the Fifth Symposium on Trends in Functional Programming (TFP 2004).
8. J. Launchbury. *A Natural Semantics for Lazy Evaluation*, Charleston, 1993. In proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Programming Languages, pages 144-154. `citeseer.nj.nec.com/launchbury93natural.html`.
9. E. Barendsen and S. Smetsers, *Strictness Typing*, Nijmegen, 1998. In proceedings of the 10th International Worksop on the Implementation of Functional Languages (IFL 1998), pages 101-116.