ART Memo 7 (Version 1)
A Semantics for Redex Trails

Olaf Chitil

July 14, 2000

Abstract

We define a small step operational semantics for a core of Haskell. We extend this semantics to additionally create redex trails. This small and direct semantics of redex trails shall improve our understanding of them and shall help to modify or extend them in a useful way.

1 Introduction

1.1 Redex Trails (from Memo 2)

A redex trail is a directed graph with an application at every node. An application is either a redex or a value (constant, constructor application, partial application). The redex trail shows the computation history of an application. A redex is the parent of an application, if it created that application. An application has exactly one parent (except for root redexes), but every of its subexpressions also has a parent which may be distinct. The redex trail of an application is the graph consisting of its parent and the parents of its subexpressions and recursively the redex trails of these parents and the subexpressions of these parents. Although the sequence of parents can be seen as the main redex trail of an application, the whole trail is a complex branching structure.

The trace of a program run is the redex trail of its result. This trace contains the redex trails of all intermediate values and also of redexes that were created but have never been evaluated during the computation.

Because an application consists of values and sometimes redexes, the graph structure of the trace is complex. A redex is the source of a single arrow and the target of (possibly several) arrows. A value or a subexpressions of an application is the source of a single arrow but never the target of an arrow. Note that a redex is a subexpression of an application only if this redex was never evaluated. Hence it cannot be the target of an arrow.

1.2 Definition of Redex Trails

The current definition of redex trails in [SR97b, SR97a] is insufficient. Redex trails are introduced informally only at the hand of a few examples. The only formal definition of redex traces is given through the transformation that enables the creation of the trace. However, this definition is inadequate. The fact that the description of the transformation is incomplete and contains errors is only a minor problem.

To obtain the redex trail of a program, the program has to be transformed as described and then be executed to create the trail. This has the following disadvantages:

- It is not easy to determine the redex trail of even a simple program by hand.
- The second stage of obtaining a redex trail is based on the semantics of the *transformed* program. This is rather confusing, because the purpose of a redex tail is to give an abstract view of the computation of the *original* program.
- Although it is the purpose of a redex tail to give an abstract view of a computation, a description of the computation of a program is not part of the definition. It remains unclear which assumptions about the operational semantics are made, both in that a redex trace shall be an abstraction of a computation and that the execution of the transformed program shall yield the desired redex trail.

So the transformation defines a redex trail only in a very indirect manner, as depicted here:

$$\text{computation} \overset{semantics}{\longleftarrow} \text{program 1} \overset{semantics}{\longrightarrow} \text{program 2} \overset{result\ semantics}{\longrightarrow} \text{redex trail}$$

We conclude that the transformation is a good definition of an implementation of redex trails but not a good definition of redex trails themselves.

Hence we give a direct definition of redex trails here. Because a redex trail shall give an abstract view of a computation, we start with a high-level semantics of a functional language and then extend it to describe redex trails beside the normal semantics. This semantics will not have the disadvantage listed before and we can furthermore use it for the following purposes:

- Inconsistencies between the two definitions of redex trails will improve our understanding of them. The final consistency of the two definitions of redex trails will give us confidence that we have defined what we had in mind.
- To prove the correctness of the transformation that implements redex trails.
- To extend redex trails systematically to language constructs for which they are not yet (properly) defined, for example list comprehensions and IO.
- In [SR97a] it is claimed that the redex trail of a program is complete, that "every reduction occurring in the computation can be observed in the trail". It would be nice to prove this property.

2 A High-Level Operational Call-by-Need Semantics

Although a redex trail does not fully describe a computation, it relates redexes and describes sharing properties of call-by-need evaluation. Hence we have to start with an operational semantics. However, we do not want to tie ourselves to a specific abstract machine. Hence Launchbury's natural semantics for call-by-need [Lau93] is the obvious choice for an operational semantics. However, we modify it in several aspects. Some modifications are necessary to reach our goal of defining redex trails, some are more for convenience.

A natural semantics only describes computations of expressions that have a value, that is, a weak head normal form. However, we are also interested in obtaining a redex trail for expressions that have value $\bot$, that is, whose computation leads to an error, a black hole, or

an infinite computation that has to be interrupted. To be able to obtain a computation in this case we use a small step variant of Launchbury's semantics (cf. [WP99]).

In our first definition of the semantics we still restrict ourselves to a small language. We discuss further language constructs in subsequent sections. Especially we forget about IO for the moment. In our semantics an expression just evaluates to a value.

To achieve sharing easily, Launchbury requires that arguments of functions (and later also data constructors) are variables. So a preliminary translation is needed that introduces `let` constructs to remove non-variable arguments. This translation will not effect the redex trail, because the reduction of a `let` construct is not registered in the redex trail, a `let` construct is not considered as a redex. We restrict the language further so that only a definition body in a `let` construct can be an application. Thus we assure that a redex will never be a proper subexpression of an expression in a heap cell, but that every redex will have its own heap cell. We will need this property later when we extend the semantics for redex trails. The syntax is defined in Figure 1. We give a semantics to an expression by defining to which value it evaluates.

To express sharing the semantics operates on graphs instead of expressions. For graphs a machine-oriented representation is chosen. An expression is respresented by a graph. A graph $\Gamma : M$ consists of a heap Γ and a variable. The variable points to the heap cell that represents the top of the expression. A heap, denoted by Γ or Δ, is a partial mapping from variables to expressions. We write it as a set of tuples $x \mapsto M$. By domain of a heap we mean the domain of the mapping. The operator $+$ combines two heaps under the assumption that their domains are disjunct. Computation is performed only on closed heaps, that is, heaps where the free variables of all terms in the heap are a subset of the domain of the heap. So a closed graph represents a closed term with shared subterms.

In Figure 2 we define a reduction relation on graphs, where $\Gamma : z \Rightarrow \Delta : z$ or simply $\Gamma \underset{z}{\Rightarrow} \Delta$ means that the graph $\Gamma : z$ reduces to the graph $\Delta : z$. A computation is a finite or infinite sequence of graphs $G_1, G_2, \ldots$ with $G_i \Rightarrow G_{i+1}$ for all graphs in the sequence. Note that the reduction relation is actually a function, that is, the second graph is determined by the first. The computation of a expression M is the computation beginning with the graph $\{z \mapsto M\} : z$. M evaluates to a value, if its computation is finite with the last graph $\Delta : z$ having the property $\Delta \succ_z V$. We write $\{z \mapsto M\} : z \overset{*}{\Rightarrow} \Delta : z$.

Note that the APPLICATION REDUCTION rule does not require a context $\mathbb{C}$, because a redex is always either the top level expression or the content of a heap cell (the latter being reduced with the help of the FOLLOW VARIABLE rule). It is never a proper subexpression of the content of a heap cell.

In Launchbury's semantics a graph is a tuple of a heap and a term, the latter of which represents the top of the term and has variables pointing into the heap. This top term component is used to give values upwards. So in the small step version of Launchbury's semantics given in [WP99], there is a rule like

$$\overline{\Gamma + \{x \mapsto V\} : x \Rightarrow \Gamma + [x \mapsto V] : V}$$

We cannot simply give values upwards in this way, because this would conflict with the construction of redex trails as we will see in the subsequent section. Instead of giving values upwards we use the value lookup relation which we introduced especially for this purpose..

variable	x, y, z			
expression	M, N	$:=$	$\hat{M}$	restricted expression
		$\mid$	$\hat{M}\, x$	application
		$\mid$	$\texttt{let}\ \{x_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N$	let
restricted expression	$\hat{M}, \hat{N}$	$:=$	x	variable
		$\mid$	$\lambda x.\, M$	abstraction
		$\mid$	$\texttt{let}\ \{x_i = M_i\}_{i=1}^{k}\ \texttt{in}\ \hat{N}$	let
value	V	$:=$	$\lambda x.\, M$	abstraction
leftmost outermost context	$\mathbb{C}$	$:=$	$[\cdot]$	hole
		$\mid$	$\mathbb{C}\, x$	application context

Figure 1: Syntax

$$\frac{\Gamma \succ_z (\lambda x.\, N)\, y}{\Gamma \underset{z}{\Rightarrow} \Gamma[z \mapsto N[y/x]]}\ \textsc{application reduction}$$

$$\frac{\Gamma(z) = \mathbb{C}[\texttt{let}\ \{x_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N] \quad y_i \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{z}{\Rightarrow} \Gamma[z \mapsto \mathbb{C}[N[\overline{y}/\overline{x}]]] + \{y_i \mapsto M_i[\overline{y}/\overline{x}]\}}\ \textsc{let reduction}$$

$$\frac{\Gamma(z) = \mathbb{C}[w] \quad \Gamma \underset{w}{\Rightarrow} \Delta}{\Gamma \underset{z}{\Rightarrow} \Delta}\ \textsc{follow variable}$$

Figure 2: Small step reduction rules

$$\frac{}{\Gamma \succ_x \Gamma(x)} \qquad \frac{\Gamma(x) = \mathbb{C}[y] \quad \Gamma \succ_y V}{\Gamma \succ_x \mathbb{C}[V]}$$

Figure 3: Value lookup

3 Extension for Redex Trails

We allocate the redexes which form the redex trails in the heap as well. It is not possible to completely separate redex trails from the expression that is under evaluation anyway, because

the argument of a redex may be evaluated later and the redex trail shall hold the most evaluated form.

To every heap cell we add a variable which points to the redex that created this heap cell. We also add such a variable for the top level expression. Figure 4 shows the extended semantics. To increase readability we use s and t for these special variables.

Every redex reduces to an expression with its own history. Hence every redex needs its own heap cell and we restricted the syntax to reach this goal. An alternative would be to add trail variables to subexpressions. This would, however, also require a modification of syntax and make expressions far less readable.

So for a (possible incomplete) computation $\emptyset : \text{Root} \cdot M \stackrel{*}{\Rightarrow} \Delta : t \cdot N$ the result is $\Delta : N$ and the redex trail is $\Delta : t$.

$$\frac{\Gamma \succ_z (\lambda x.\, N)\, y \quad t \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{z}{\Rightarrow} \Gamma[z \mapsto t \cdot N[y/x]] + \{t \mapsto \Gamma(z)\}} \text{ APPLICATION REDUCTION}$$

$$\frac{\Gamma(z) = t \cdot \mathbb{C}[\texttt{let}\ \ \{x_i = M_i\}_{i=1}^{k}\ \ \texttt{in}\ \ N] \quad y_i \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{z}{\Rightarrow} \Gamma[z \mapsto t \cdot \mathbb{C}[N[\overline{y}/\overline{x}]]] + \{y_i \mapsto t \cdot M_i[\overline{y}/\overline{x}]\}} \texttt{LET} \text{ REDUCTION}$$

$$\frac{\Gamma(z) = s \cdot \mathbb{C}[w] \quad \Gamma \underset{w}{\Rightarrow} \Delta}{\Gamma \underset{z}{\Rightarrow} \Delta} \text{ FOLLOW VARIABLE}$$

$$\frac{\Gamma(x) = s \cdot M}{\Gamma \succ_x M} \qquad \frac{\Gamma(x) = s \cdot \mathbb{C}[y] \quad \Gamma \succ_y V}{\Gamma \succ_x \mathbb{C}[V]}$$

Figure 4: Small step reduction rules for redex trails

The additional trail variables are reminiscence of the cost centers in [SP97].

Note that the definition of contexts $\mathbb{C}$ has no influence on how redex trails are constructed. It should be possible to make even clearer, that evaluation order has no influence on the redex trail structure, only on how far arguments in redexes are evaluated.

4 Functions with Arity

Until now a redex is an application of a λ-abstraction to one argument. This leads to a large number of redexes and reductions. We want a function value to have an arity. Only the application of a function value of arity n to n arguments is a redex. Figure 5 shows that syntax and semantics can easily be modified to cater for functions with arity. Note that application is still binary, because the arity of function component is not known statically. Note that the definition of values includes abstractions without any arguments.

$$
\begin{array}{rrcll}
\text{expression} & M, N, O, P & := & x & \text{variable} \\
& & | & \lambda x_1 \ldots x_n . M & \text{abstraction of arity } n \\
& & | & \texttt{let } \{x_i = \hat{M}_i\}_{i=1}^{k} \texttt{ in } N & \texttt{let} \\
\text{value} & V & := & (\lambda x_1 \ldots x_n . M)\, y_1 \ldots y_m \text{ with } m < n & \text{partial application}
\end{array}
$$

$$
\frac{\Gamma \succ_x (\lambda x_1 \ldots x_n . N)\, y_1 \ldots y_n \quad t \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{z}{\Rightarrow} \Gamma[z \mapsto N[\overline{y}/\overline{x}]] + \{t \mapsto \Gamma(x)\}} \text{ APPLICATION REDUCTION}
$$

Figure 5: Changes in syntax and semantics for redex trails for functions with arity

5 Data Structures

Introduction of algebraic data types and the corresponding `case` construct is straightforward. The `case` construct gives a second kind of redex. Figure 6 shows all the necessary extensions.

$$
\begin{array}{rrcll}
\text{expression} & M, N, O, P & := \ldots & | \ \texttt{case } \hat{M} \texttt{ of } \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k} & \text{data destruction} \\
\text{restricted expression} & \hat{M}, \hat{N} & := \ldots & | \ C & \text{data constructor} \\
\text{value} & V & := \ldots & | \ C\, x_1 \ldots x_n & \text{(partially) applied constructor} \\
\text{context} & \mathbb{C} & := \ldots & | \ \texttt{case } \mathbb{C} \texttt{ of } \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k} &
\end{array}
$$

$$
\frac{\Gamma \succ_z \texttt{case } C_j\, \overline{y} \texttt{ of } \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k} \quad t \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{z}{\Rightarrow} \Gamma[z \mapsto t \cdot N_j[\overline{y}/\overline{x}]] + \{t \mapsto \Gamma(z)\}}
$$

Figure 6: Extension of syntax and semantics for data structures

6 Pattern Matching

Lauchbury only gives the semantics for a core language. The idea is that standard functional language can be translated into this core language, for example by transforming pattern matching into nested case constructs. For our purposes such a preliminary translation is undesirable, because the close link to the original program would be lost. Although we will obviously not give a semantics for full Haskell, we at least outline how it could be done without requiring a translation into the core language.

Patterns can be arbitrarily nested, but we demand that they are linear, that is, a variable

occurs at most once in a pattern.

$$\begin{array}{lrll} \text{pattern} & p & := & x & \text{variable} \\ & & \mid & C\,p_1 \ldots p_n \quad n \text{ is arity of } C & \text{full constructor application} \end{array}$$

The simple λ-abstraction is replaced by an abstraction with pattern matching:

$$\lambda p_1.N_1; \ldots ; p_n.N_n$$

For simplicity we only consider single argument functions here.

A pattern p' *matches* a pattern p if there is a variable substitution σ such that $p\sigma = p'$.

We define the relation that a pattern does not match a graph Γ at a variable x as follows:

$$\frac{\Gamma \succ_x D\,p'_1 \ldots p'_m}{\Gamma \bowtie^x C\,p_1 \ldots p_n} \qquad \frac{\Gamma \succ_x C\,x_1 \ldots x_n \quad \Gamma \succ_{x_1} p_1\sigma \;\ldots\; \Gamma \succ_{x_{i-1}} p_{i-1}\sigma \quad \Gamma \bowtie^{x_i} p_i}{\Gamma \bowtie^x C\,p_1 \ldots p_n}$$

Note that we require that everything to the left of the mismatching constructor does match, because matching is performed from left to right.

For the process of evaluating an argument to test if it matches a pattern p we define a pattern contexts $\mathbb{C}_p$:

$$\begin{array}{rcl} \mathbb{C}_x & := & [\cdot] \\ \mathbb{C}_{C\,p_1 \ldots p_n} & := & [\cdot] \mid C\,\mathbb{C}_{p_1}\,x_2 \ldots x_n \mid C\,p_1\mathbb{C}_{p_2}\,x_3 \ldots x_n \mid \ldots \mid C\,p_1 \ldots p_{n-1}\mathbb{C}_{p_n} \end{array}$$

The final step: the reduction rules for an abstraction with patterns:

$$\frac{\Gamma \succ_z (\lambda p_1.N_1; \ldots ; p_n.N_n)\,y \quad \Gamma \bowtie^y p_1 \;\ldots\; \Gamma \bowtie^y p_{i-1} \quad \Gamma \succ_y \mathbb{C}_{p_i}[x] \quad \Gamma \underset{\vec{x}}{\Rightarrow} \Delta}{\Gamma \underset{z}{\Rightarrow} \Delta}$$

$$\frac{\Gamma \succ_z (\lambda p_1.N_1; \ldots ; p_n.N_n)\,y \quad \Gamma \bowtie^y p_1 \;\ldots\; \Gamma \bowtie^y p_{i-1} \quad \Gamma \succ_y p_i\sigma}{\Gamma \underset{z}{\Rightarrow} \Gamma[z \mapsto t \cdot N_i\sigma] + \{t \mapsto \Gamma(z)\}}$$

7 Projections

In the transformation that implements redex trails functions that are projections have to be handled specially, otherwise the reduction of the projection would not be registered in the redex trail of the result [SR97a]. The source of the probem might be that in the transformation system only values carry redex trails, not expressions that are under evaluation.

In our operational semantics we do not have to handle projections in any special way to make redex trail construction work correctly as the following little example demostrates:

$$\begin{array}{ll} & \{z \mapsto \text{Root} \cdot \texttt{let}\ \{head = \lambda(x : xs).x, ys = v : vs, v = True, vs = []\}\ \texttt{in}\ head\,ys\} \\ \underset{z}{\Rightarrow} & \{head \mapsto \text{Root} \cdot \lambda(x : xs).x, ys \mapsto \text{Root} \cdot v : vs, v \mapsto \text{Root} \cdot True, vs \mapsto \text{Root} \cdot [] \\ & , z \mapsto \text{Root} \cdot head\,ys\} \\ \underset{z}{\Rightarrow} & \{\ldots, z \mapsto t \cdot v, t \mapsto \text{Root} \cdot head\,ys\} \end{array}$$

8 IO

For a program that performs IO the value to which it evaluates is irrelevant (it may not even terminate) but its semantics is determined by the sequence of actions it performs. We consider two actions, `put C` can be interpreted as writing a nullary constructor `C` and `get C` can be interpreted as reading a nullary constructor `C`. A program does not determine a single action sequence. If a sequence has a `get C` action, then there exists for every nullary constructor C a sequence which is identical up to this action, followed by a `get` C.

We extend the language by two functions `put` and `get` that create the actions, a function `>>=` to sequence actions and a special function `return`. The latter is handled like a data constructor except that it cannot be destructed by a `case` construct. Only `>>=` can destruct this special value and thus we ensure that actions are sequenced correctly and referential transparency is preserved. So we do not have a static type system but we dynamically check that actions are sequenced correctly.

Our reduction relation may now carry an additional action (a missing action could also be viewed as an non-observable action τ). Note that we use `>>=` infix. For simplicity the `put` C action returns the constructor C. Figure 7 shows the extension of our semantics to IO.

$$
\begin{array}{rrcl}
\text{restricted expression} & \hat{M}, \hat{N} & := & \ldots \mid \texttt{put} \mid \texttt{get} \mid \texttt{>>=} \mid \texttt{return} \\
\text{value} & V & := & \ldots \mid \texttt{return} \mid \texttt{return}\, x \\
\text{leftmost outermost context} & \mathbb{C} & := & \ldots \mid \texttt{put}\ [\cdot] \mid [\cdot]\ \texttt{>>=}\ \texttt{x}
\end{array}
$$

$$
\frac{\Gamma(z) = \texttt{put}\ \ x \quad \Gamma \succ_x C}{\Gamma \overset{\texttt{put}\,C}{\underset{z}{\Rightarrow}} \Gamma[z \mapsto \texttt{return}\ \ x]}
\qquad\qquad
\frac{\Gamma(z) = \texttt{get} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \overset{\texttt{get}\,C}{\underset{z}{\Rightarrow}} \Gamma[z \mapsto \texttt{return}\ \ x] + \{x \mapsto C\}}
$$

$$
\frac{\Gamma \succ_z (\texttt{return}\ \ y)\ \texttt{>>=}\ \ x}{\Gamma \underset{z}{\Rightarrow} \Gamma[z \mapsto x\, y]}
$$

Figure 7: Extension of syntax and semantics for IO

Next we have to extend our IO semantics for redex trails. We do this in a way that resembles the current implementation. Thus it highlights the shortcomings of the current implementation. Figure 8 shows the extension.

So every `put` action has a parent. Conversely every constructor that was read by a `get` action has the `get` redex as parent. IO is just a dummy trace variable. We note that the newly allocated `return` heap cells do not have parents. Furthermore, the reduction of the sequencing function `>>=` is not registered in any trail. Thus computations that construct actions are not registered at all in the trail.

It is not quite clear how the Haskell function `getContents` should be modelled in our framework. It actually leads to an interleaving of independent action sequences. However, the assumption is that the environment provides `getContents` with characters independent of the interleaving sequence. This gives ideas for interesting further research. It should be

$$\frac{\Gamma(z) = s \cdot \texttt{put}\ \ x \quad \Gamma \succ_x C \quad t \notin \mathrm{dom}(\Gamma)}{\Gamma \overset{t \cdot \texttt{put}\, C}{\underset{z}{\Rightarrow}} \Gamma[z \mapsto \mathrm{IO} \cdot \texttt{return}\ \ x] + \{t \mapsto \Gamma(z)\}}$$

$$\frac{\Gamma(z) = s \cdot \texttt{get} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \overset{\texttt{get}\, C}{\underset{z}{\Rightarrow}} \Gamma[z \mapsto \mathrm{IO} \cdot \texttt{return}\ \ x] + \{x \mapsto t \cdot C\} + \{t \mapsto \Gamma(z)\}}$$

$$\frac{\Gamma \succ_z (\texttt{return}\ \ y)\ \texttt{>>=}\ \ x}{\Gamma \underset{z}{\Rightarrow} \Gamma[z \mapsto \mathrm{IO} \cdot x\, y]}$$

Figure 8: Extension of IO semantics for redex trails

possible to model `unsafePerformIO` in this way and state precisely which proof obligations or assumptions the programmer has to make to ensure referential transparency.

9 Named Functions

λ-abstractions are very difficult to read. Usually functions have names and we prefer to show these names. Although we do not support[1] the syntax $fx_1 \ldots x_n = M$ we can name a function nonetheless by binding a λ-abstraction to a variable in a `let` construct: `let` $\{f = \lambda x_1 \ldots x_n.M\}$ `in` $\ldots$. When we show a redex trail we would like to be able to show this variable instead of the λ-abstraction. We achieve this by adding yet another variable to every heap cell. Noet that this variable only serves as a name and may not be in the domain of the heap. We should probably distinguish between variables occurring in the original program and those introduced to fit our syntax (arguments must be variables). The later should not be shown in the redex trail.

10 Extraction of Redex Trail

Intermediate expressions and final values of a computation are mixed with redex trails in the heap. For a given graph we would like to extract the redex trail. A normal garbage collector with the final value or redex as root does just that. We could modify the garbage collector to replace λ-abstractions by their names as far as possible and remove the λ-abstractions.

11 CAFs

The names of CAFs are not registered in the redex trail. The parent of an instance right hand side of a CAF is the next enclosing λ-abstraction. Should we introduce nullary λ-abstractions

[1] It simplifies the semantics that a λ-abstraction is just an expression. For example, we can thus easily store it in the heap.

for CAFs or handle the `let` construct differently?

12 Are Small Steps Small Enough?

We chose a small step semantics to handle infinite and interrupted computations. However, in our semantics a single step is still complex and its derivation usually requires numerous uses of the FOLLOW VARIABLE rule. This rule basically searches for the current redex and does so for each step anew. Sestoft's Mark 1 machine [Ses97] has a stack that keeps track of redex positions. Every step is just an instance of a rule, it does not require an arbitrarily large derivation. Hence it may be more suitable, especially because it allows more fine grained interruption of a computation.

When a computation is stopped before reaching a value, the question arises which redex or value is the one that determines the redex trail of the computation. we distinguish several cases:

- blackhole: the source of the redex trail should be the variable that depends on itself. The redex trail should include this self-dependence. However, a variable is neither a redex nor a value. Example: `let x = x in x`.
- error: the source should be the `error` redex itself.
- interruption: next redex to evaluate.
- insufficient space (heap,stack): redex whose reduction caused space problem.

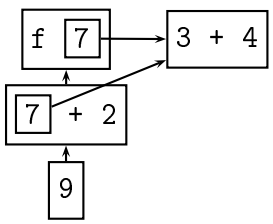
In the last two cases the redex might actually be a `let` expression. Should the last "real" redex be chosen in that case?

13 An Outdated Example

For a readable example we need some simple primitive type. We extend our system by natural numbers with addition as given in Figure 9[2].

Now we evaluate the expression `let` $\{f = \lambda x.\, x + 2, a = 3 + 4\}$ `in` $f\, a$.

$$
\begin{aligned}
& \emptyset : \mathrm{Root} \cdot \texttt{let}\ \{f = \lambda x.\, x + 2, a = 3 + 4\}\ \texttt{in}\ f\, a \\
\Rightarrow\ & \{f \mapsto \mathrm{Root} \cdot \lambda x.\, x + 2, a \mapsto \mathrm{Root} \cdot 3 + 4\} : \mathrm{Root} \cdot f\, a \\
\Rightarrow\ & \{\ldots\} + \{t \mapsto \mathrm{Root} \cdot f\, a\} : t \cdot a + 2 \\
\Rightarrow\ & \{\ldots, a \mapsto s \cdot 7, \ldots\} + \{s \mapsto \mathrm{Root} \cdot 3 + 4\} : t \cdot a + 2 \\
\Rightarrow\ & \{\ldots\} + \{r \mapsto t \cdot a + 2\} : r \cdot 9 \\
=\ & \{f \mapsto \mathrm{Root} \cdot \lambda x.\, x + 2, a \mapsto s \cdot 7, t \mapsto \mathrm{Root} \cdot f\, a, s \mapsto \mathrm{Root} \cdot 3 + 4, r \mapsto t \cdot a + 2\} : r \cdot 9
\end{aligned}
$$



Note that the parent of $3 + 4$ is Root, not a.

[2]The new context makes reduction non-deterministic; however, this context is needed for value lookup.

$$
\begin{array}{rrcll}
\text{expression} & M, N & := \ldots & | \quad M + M & \text{addition} \\
\text{restricted expression} & \hat{M}, \hat{N} & := \ldots & | \quad n & (n \in I\!N) \\
\text{value} & V & := \ldots & | \quad n & \text{numbers} \\
\text{leftmost outermost context} & \mathbb{C} & := \ldots & | \quad \mathbb{C} + \mathbb{C} &
\end{array}
$$

$$
\frac{M \equiv_\Gamma n_1 + n_2 \quad t \notin \mathrm{dom}(\Gamma)}{\Gamma : s \cdot M \Rightarrow \Gamma + \{t \mapsto s \cdot M\} : t \cdot n_1 \underline{+} n_2}
$$

Figure 9: Extension for numbers

References

[Lau93] John 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. ACM Press, 1993.

[Ses97] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[SP97] Patrick M. Sansom and Simon L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, March 1997.

[SR97a] Jan Sparud and Colin Runciman. Complete and partial redex trails of functional computations. In C. Clack, K. Hammond, and T. Davie, editors, *Selected papers from 9th Intl. Workshop on the Implementation of Functional Languages (IFL'97)*, pages 160–177. Springer LNCS Vol. 1467, September 1997.

[SR97b] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proc. 9th Intl. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS Vol. 1292, September 1997.

[WP99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Conference Record of POPL '99: The* 26$^{\text{nd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.