ART Memo 7 (Version 2)
A Semantics for Redex Trails

Olaf Chitil

December 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

A redex trail is a directed graph. Every node is labelled with an expression which 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 expression, if it created that expression. An expression 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 expression is the graph consisting of its parent and the parents of its subexpressions and recursively the redex trails of these parents. Although the sequence of parents can be seen as the main redex trail of an expression, 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 a node expression 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 any arrow. Note that a redex is a proper subexpression of a node expression 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:

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

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.

1.3 The Importance of Sharing

The notions of reduction and reducible expression (redex) are certainly operational. Hence we have to start with an operational semantics although a redex trail does not fully describe a full computation.

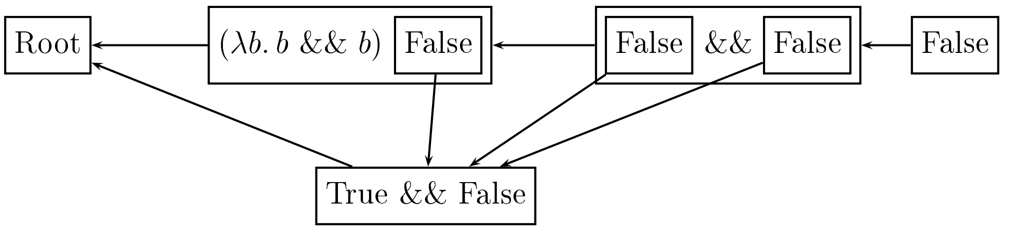
A simple term-based call-by-name reduction semantics unfortunately does not suffice, even if we ignore that the redex trail shall desribe the sharing of subexpressions correctly. The redex trail shall also hold all subexpressions in most evaluated form. The following example demonstrates this:

$$(\lambda b.\, b \;\&\&\; b)\ (\text{True} \;\&\&\; \text{False})$$

Call-by-name reduction:

$$
\begin{array}{l}
(\lambda b.\, b \mathbin{\&\&} b)\ (\mathrm{True} \mathbin{\&\&} \mathrm{False}) \\
\Rightarrow \\
(\mathrm{True} \mathbin{\&\&} \mathrm{False}) \mathbin{\&\&} (\mathrm{True} \mathbin{\&\&} \mathrm{False}) \\
\Rightarrow \\
\mathrm{False} \mathbin{\&\&} (\mathrm{True} \mathbin{\&\&} \mathrm{False}) \\
\Rightarrow \\
\mathrm{False}
\end{array}
$$

The redex trail:



The above representation of the redex trail does not show that the three occurrences of False which have the same parent are actuall the same shared expression. Nonetheless the call-by-name reduction and this representation of the redex trail differ in an essential point: In the redex trail both arguments of && in the parent of the result are False, whereas in the call-by-name reduction the second argument of && is left unevaluated. So the fact that an expression is shared becomes important for the redex trail when it is *updated*.

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

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, for our goal of defining redex trails many modifications are necessary.

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 $\perp$, 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]). To avoid confusion we postpone a detailed discussion of the similarities and differences between our semantics and variants of Launchbury's semantics to Section 9.

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. 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 rooted graphs instead of expressions. An expression is respresented by a rooted graph, that is, a graph plus a graph node or pointer which indicates the top of the expression. A graph, denoted by Γ or Δ, is a partial mapping from nodes to expressions. We write it as a set of tuples $p \mapsto M$. The nodes of the graph are its domain, that is, the domain of the mapping. The operator $+$ combines two graphs under the assumption that their domains are disjunct. Computation is performed only on closed graphs, that is, graphs where the nodes of all terms in the graph are a subset of the domain of the graph. So a closed graph represents a closed term with shared subterms.

In Figure 2 we define a reduction relation on rooted graphs, where $\Gamma : p \Rightarrow \Delta : p$ or simply $\Gamma \underset{p}{\Rightarrow} \Delta$ means that the graph $\Gamma : p$ reduces to the graph $\Delta : p$. A computation is a finite

variable	x, y, z			
graph node (pointer)	p, q, r			
expression	M, N	$:=$	x	variable
		$\mid$	p	node
		$\mid$	$\lambda x.\, M$	abstraction
		$\mid$	$M\, N$	application
value	V	$:=$	$\lambda x.\, M$	abstraction
simple evaluation context	$\mathbb{E}$	$:=$	$[\cdot]\, p$	application context

Figure 1: Syntax

$$\frac{\Gamma \succ_p (\lambda x.\, N)\, q}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto N[q/x]]}\ \textsc{application reduction}$$

$$\frac{\Gamma(p) = M\, N \quad N \text{ not a node} \quad q \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto M\, q] + \{q \mapsto N\}}\ \textsc{ensure argument sharing}$$

$$\frac{\Gamma(p) = \mathbb{E}[M] \quad M \text{ not a node} \quad \text{there is no redex } N \text{ with } \Gamma \succ_p N \quad q \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto \mathbb{E}[q]] + \{q \mapsto M\}}\ \textsc{new node for every redex}$$

$$\frac{\Gamma(p) = \mathbb{E}[q] \quad (\text{there is no redex } N \text{ with } \Gamma \succ_p N) \quad \Gamma \underset{q}{\Rrightarrow} \Delta}{\Gamma \underset{p}{\Rrightarrow} \Delta}\ \textsc{follow need / enter context}$$

$$\frac{\Gamma(p) = q \quad \Gamma \underset{q}{\Rrightarrow} \Delta}{\Gamma \underset{p}{\Rrightarrow} \Delta}\ \textsc{follow variable}$$

Figure 2: Small step reduction rules

$$\frac{}{\Gamma \succ_p \Gamma(p)} \qquad \frac{\Gamma(p) = \mathbb{E}[q] \quad \Gamma \succ_q V}{\Gamma \succ_p \mathbb{E}[V]} \qquad \frac{\Gamma(p) = q \quad \Gamma \succ_q V}{\Gamma \succ_p V}$$

Figure 3: Value lookup

or infinite sequence of rooted graphs $G_1, G_2, \ldots$ with $G_i \Rightarrow G_{i+1}$ for all rooted 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 an expression M is the computation beginning with the graph $\{p \mapsto M\} : p$. M evaluates to a value, if its computation is finite with the last graph $\Delta : p$ having the property $\Delta \succ_p V$. We write $\{p \mapsto M\} : p \overset{*}{\Rightarrow} \Delta : p$.

Note that a redex is of the form $\mathbb{E}[V]$.

The NEW NODE rule assures that a redex will never be a proper subexpression of a node expression, but that every redex will be a node expression. On the one hand this simplifies the APPLICATION REDUCTION rule; it does not require an evaluation context. However, most importantly we will need this property later when we extend the semantics for redex trails.

Note that there is no reduction inside a variable binding expression. Hence we also never break up a variable binding expression into several nodes. So we can instantiate a bound variable by standard substitution.

Lemma 2.1 (Reduction is deterministic) *For a graph* Γ *and* $p \in \mathrm{dom}(\Gamma)$ *there exists at most one graph* Δ *with* $\Gamma \underset{p}{\Rightarrow} \Delta$. □

3 Extension for Redex Trails

We allocate the redexes which form the redex trails in the graph 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. So the redex trail has to refer to unevaluated subexpressions of the evaluated expression.

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

Every redex reduces to an expression with its own history. Hence every redex needs its own heap cell and we introduced the NEW NODE rule 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 $\emptyset : \mathrm{Root} \cdot M \overset{*}{\Rightarrow} \Delta : t \cdot N$ the result is $\Delta : N$ and the redex trail is $\Delta : t$????????????? Thus the last top level redex would be the starting point.

Note that the definition of contexts $\mathbb{E}$ 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. Maybe I should define a non-deterministic semantics that may evaluate more than necessary (covering both call-by-value and call-by-need)?

4 Data Structures

Introduction of algebraic data types and the corresponding `case` construct is straightforward. The `case` construct gives a second kind of redex. Figure 5 shows all the necessary extensions.

$$\frac{\Gamma \succ_p (\lambda x.\, N)\, q \quad t \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto t \cdot N[q/x]] + \{t \mapsto \Gamma(p)\}} \text{ APPLICATION REDUCTION}$$

$$\frac{\Gamma(p) = t \cdot M\, N \quad N \text{ not a node} \quad q \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto t \cdot M\, q] + \{q \mapsto t \cdot N\}} \text{ ENSURE ARGUMENT SHARING}$$

$$\frac{\Gamma(p) = t \cdot \mathbb{E}[M] \quad M \text{ not a node} \quad \text{there is no redex } N \text{ with } \Gamma \succ_p N \quad q \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto t \cdot \mathbb{E}[q]] + \{q \mapsto t \cdot M\}} \text{ NEW NODE FOR EVERY REDEX}$$

$$\frac{\Gamma(p) = t \cdot \mathbb{E}[q] \quad (\text{there is no redex } N \text{ with } \Gamma \succ_p N) \quad \Gamma \underset{q}{\Rrightarrow} \Delta}{\Gamma \underset{p}{\Rrightarrow} \Delta} \text{ FOLLOW NEED / ENTER CONTEXT}$$

$$\frac{\Gamma(p) = t \cdot q \quad \Gamma \underset{q}{\Rrightarrow} \Delta}{\Gamma \underset{p}{\Rrightarrow} \Delta} \text{ FOLLOW VARIABLE}$$

$$\frac{\Gamma(x) = t \cdot M}{\Gamma \succ_p M} \qquad \frac{\Gamma(p) = t \cdot \mathbb{E}[q] \quad \Gamma \succ_q V}{\Gamma \succ_p \mathbb{E}[V]} \qquad \frac{\Gamma(p) = t \cdot q \quad \Gamma \succ_q V}{\Gamma \succ_p V}$$

Figure 4: Small step reduction rules for redex trails

$$\begin{array}{rrcll}
\text{expression} & M, N & := & \ldots & \\
 & & | & C & \text{data constructor} \\
 & & | & \texttt{case}\ M\ \texttt{of}\ \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k} & \text{data destruction} \\
\\
\text{value} & V & := & \ldots \mid C\, p_1 \ldots p_n \text{ with } n \geq 0 & \text{(partially) applied constructor} \\
\text{context} & \mathbb{E} & := & \ldots \mid \texttt{case}\ [\cdot]\ \texttt{of}\ \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k} &
\end{array}$$

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

Figure 5: Extension of syntax and semantics for data structures

5 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 6 shows that syntax and semantics can easily be modified to cater for functions with arity.

Note that for a one-argument function the semantics remains unchanged.

Note that it is no longer possible to statically determine the subexpressions of an expression that will become redexes. This is because the arity of function component of an application is not known statically. Consider for example the application $p\,q\,r$. It may be a single redex, the application of a binary function to two arguments. Or it may consist of two redexes, $p\,q$, which evaluates to a unary function applied to r. Or it may be a value, because the function p requires at least three arguments. Further possibilities exist. For this reason the syntax still has only binary application. Thus the reduction rule is simple and every redex is an application, not just part of an n-ary application.

expression	M, N	$:=$	$\ldots$	
		$\mid$	$\lambda x_1 \ldots x_n .\, M$ with $n \geq 1$	abstraction of arity n
value	V	$:=$	$(\lambda x_1 \ldots x_n .\, M)\, p_1 \ldots p_m$ with $0 \leq m < n$	partial application

$$\frac{\Gamma \succ_p (\lambda x_1 \ldots x_n .\, N)\, q_1 \ldots q_n \quad t \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto t \cdot N[\overline{q}/\overline{x}]] + \{t \mapsto \Gamma(p)\}} \text{ APPLICATION REDUCTION}$$

Figure 6: Changes in syntax and semantics for redex trails for functions with arity

6 Complete Redex Trails of Interrupted Computations

For a complete, successful computation the result graph (reachable parts) is the redex trail. However, for computations that were interrupted oder aborted with `error` (should be a special syntactic symbol) the redex trail is the last graph of the computation with the node that is currently under evaluation (the innermost in a sequence of evaluation contexts) as root node. Unfortunately not all parts of the redex trail that have been created are reachable from this root node. Also all nodes in the sequence of evaluation contexts, the nodes that are currently under evaluation, have not yet even been entered into the redex trail although they may be the source of the error.

Registering redexes when expressions are entered (i.e. evaluation starts).

When an expression is entered it is already itself added as parent to the redex trail. The entered expression is marked with a ˆ over the parent node.

$\Gamma : p$ is only a value if $\Gamma(p)$ is not marked as entered.

The fact that with n-ary functions function application redexes cannot be statically determined makes it impossible to add them correctly to the redex trail at the time when they are entered. When an n-ary application is entered, it is added as its own parent. (if the computation is interrupted, then the function component will have value $\bot$, which is assumed to have just the arity of any application in which it is used, returning $\bot$). However, when the function component is evaluted, then redexes may be removed or new ones added to reflect the true redexes.

Note that in the application reduction rules the application may actually consist of several applications. The syntax of application was especially chosen to make the nesting of applications invisible. However, in all other cases an application is a single n-ary application. (Not nice, but making it explicit would be really cumbersome.)

For the n-ary application we need a new ENSURE ARGUMENT SHARING rule.

$$\frac{\Gamma(p) = t \cdot \mathbb{E}[q] \quad s \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto \hat{s} \cdot \mathbb{E}[q]] + \{s \mapsto \Gamma(p)\}} \text{ REDEX TO TRAIL}$$

$$\frac{\Gamma \succ_p \hat{t} \cdot (\lambda x.\, N)\, q}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto t \cdot N[q/x]]} \text{ APPLICATION REDUCTION}$$

$$\frac{\Gamma(p) = t \cdot M\, N \quad N \text{ not a node} \quad q \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto t \cdot M\, q] + \{q \mapsto t \cdot N\}} \text{ ENSURE ARGUMENT SHARING}$$

$$\frac{\Gamma(p) = t \cdot \mathbb{E}[M] \quad M \text{ not a node} \quad q \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto t \cdot \mathbb{E}[q]] + \{q \mapsto t \cdot M\}} \text{ NEW NODE FOR EVERY REDEX}$$

$$\frac{\Gamma(p) = \hat{t} \cdot \mathbb{E}[q] \quad \Gamma \underset{q}{\Rightarrow} \Delta}{\Gamma \underset{p}{\Rightarrow} \Delta} \text{ FOLLOW NEED / ENTER CONTEXT}$$

$$\frac{\Gamma(p) = t \cdot q \quad \Gamma \underset{q}{\Rightarrow} \Delta}{\Gamma \underset{p}{\Rightarrow} \Delta} \text{ FOLLOW VARIABLE}$$

$$\frac{}{\Gamma \succ_p \Gamma(p)} \qquad \frac{\Gamma(p) = \hat{t} \cdot \mathbb{E}[q] \quad \Gamma \succ_q s \cdot V}{\Gamma \succ_p \hat{t} \cdot \mathbb{E}[V]} \qquad \frac{\Gamma(p) = \hat{t} \cdot q \quad \Gamma \succ_q s \cdot V}{\Gamma \succ_p \hat{t} \cdot \mathbb{E}[V]}$$

Figure 7: Small step reduction rules for redex trails

$$\frac{\Gamma \succ_p \hat{t} \cdot \texttt{case}\ \ C_j\,\overline{q}\ \ \texttt{of}\ \ \{C_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k}}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto t \cdot N_j[\overline{q}/\overline{x}]]}$$

Figure 8: Extension of syntax and semantics for data structures

$$\begin{array}{rrcll}
\text{expression} & M, N & := & \ldots & \\
 & & | & \lambda x_1 \ldots x_n.\, M \text{ with } n \geq 1 & \text{abstraction of arity } n \\
 & & | & N\, M_1\, \ldots\, M_n \text{ with } n \geq 1 & \text{application of arity } n \\
\\
\text{value} & V & := & \ldots \mid (\lambda x_1 \ldots x_n.\, M)\, y_1 \ldots y_m \text{ with } 0 \leq m < n & \text{partial application} \\
\\
\text{simple evaluation context} & \mathbb{E} & := & \ldots \mid [\cdot]\, p_1 \ldots p_n \text{ with } n \geq 1 & \text{application context}
\end{array}$$

$$\frac{\Gamma \succ_p \hat{t} \cdot (\lambda x_1 \ldots x_n.\, N)\, q_1 \ldots q_n}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto t \cdot N[\overline{q}/\overline{x}]]}\ \textsc{application reduction}$$

$$\frac{\Gamma \succ_p \hat{t} \cdot (\lambda x_1 \ldots x_n.\, N)\, q_1 \ldots q_m \quad m < n \quad \Gamma(t) = s \cdot M}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto s \cdot M]}\ \textsc{partial application}$$

$$\frac{\Gamma(p) = \hat{t} \cdot M\, q_i\, \ldots\, q_m \quad \Gamma \succ_p \hat{t} \cdot (\lambda x_1 \ldots x_n.\, N)\, q_1 \ldots q_m \quad i \leq n < m \quad \Gamma(t) = s \cdot M\, q_i\, \ldots\, q_m}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto \hat{t} \cdot r\, q_{n+1} \ldots q_m][t \mapsto s \cdot r\, q_{n+1} \ldots q_m] + \{r \mapsto t \cdot M\, q_i\, \ldots\, q_n\}}\ \textsc{oversaturated application}$$

$$\frac{\Gamma(p) = t \cdot N\, M_1 \ldots M_n \quad M_j \text{ not a node} \quad q \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rrightarrow} \Gamma[p \mapsto t \cdot N\, M_1 \ldots M_{j-1}\, q\, Mj+1 \ldots M_n] + \{q \mapsto t \cdot M_j\}}\ \textsc{ensure argument sharing}$$

Figure 9: Changes in syntax and semantics for redex trails for functions with arity

7 What is a Parent? The `if-then-else` Problem

The `if-then-else` construct has a special syntax and hence require special reduction rules.

$$\begin{array}{rrcl}
\text{expression} & M, N & := & \ldots \\
 & & | & \texttt{if}\ \ N\ \ \texttt{then}\ \ M_1\ \ \texttt{else}\ \ M_2
\end{array}$$

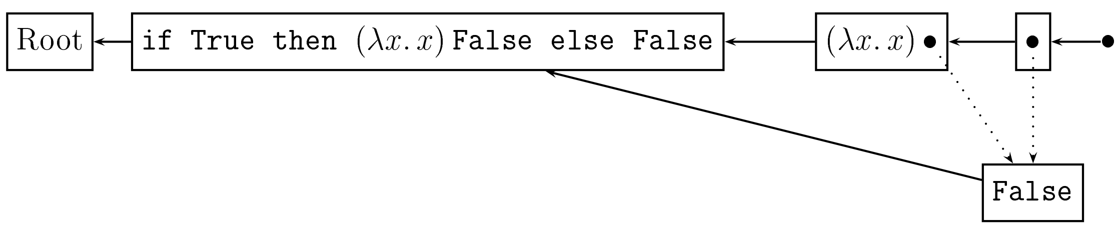
Unfortunately there are two different natural choices which give different redex trails.

The first version (which Hat currently implements) describes `if` N `then` M_1 `else` M_2 similar to `case` N `of` $\{\texttt{True} \mapsto M_1, \texttt{False} \mapsto M_2\}$:

$$\mathbb{E} := \ldots \mid \texttt{if}\ [\cdot]\ \texttt{then}\ M_1\ \texttt{else}\ M_2$$

$$\frac{\Gamma \succ_p \texttt{if True then}\ M_1\ \texttt{else}\ M_2}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto M_1]} \qquad \frac{\Gamma \succ_p \texttt{if False then}\ M_1\ \texttt{else}\ M_2}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto M_2]}$$

For the expression $\texttt{if True then}\ (\lambda x.\, x)\,\texttt{False else False}$ the redex trail augmented version of the rules gives us



The expression $\texttt{case True of}\ \{\texttt{True} \mapsto (\lambda x.\, x)\,\texttt{False}, \texttt{False} \mapsto \texttt{False}\}$ gives the same redex trail (except for one node expression of course).

The second version describes $\texttt{if}\ N\ \texttt{then}\ M_1\ \texttt{else}\ M_2$ similar to $f\ N\ M_1\ M_2$ where f is defined as

$$\begin{aligned} f\ \texttt{True}\ M_1\ M_2 &= M_1 \\ f\ \texttt{False}\ M_1\ M_2 &= M_2 \end{aligned}$$

(the semantics of such definitions has not yet been described).

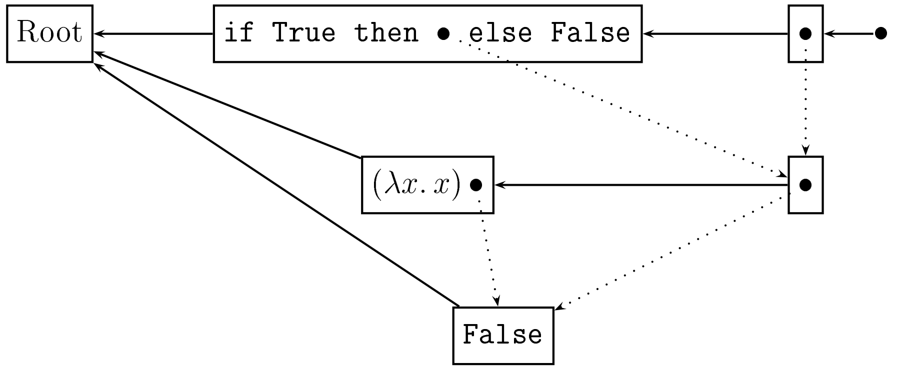
$$\mathbb{E} := \ldots \mid \texttt{if}\ [\cdot]\ \texttt{then}\ p_1\ \texttt{else}\ p_2$$

$$\frac{\Gamma \succ_p \texttt{if True then}\ p_1\ \texttt{else}\ p_2}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto p_1]} \qquad \frac{\Gamma \succ_p \texttt{if False then}\ p_1\ \texttt{else}\ p_2}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto p_2]}$$

$$\frac{\Gamma \succ_p \texttt{if True then}\ M_1\ \texttt{else}\ p_2 \quad M_1 \text{ not a node} \quad q \notin \Gamma}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto \texttt{if True then}\ q\ \texttt{else}\ p_2] + \{q \mapsto M_1\}}$$

$$\frac{\Gamma \succ_p \texttt{if False then}\ M_1\ \texttt{else}\ M_2 \quad M_2 \text{ not a node} \quad q \notin \Gamma}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto \texttt{if False then}\ M_1\ \texttt{else}\ q] + \{q \mapsto M_2\}}$$

For the expression $\texttt{if True then}\ (\lambda x.\, x)\,\texttt{False else False}$ the redex trail augmented version of the rules gives us



Note that the parent of the $(\lambda x.\, x)$ `False` node is Root, not the `if-then-else`. The expression f `True` $((\lambda x.\, x)$ `False`$)$ `False` gives the same redex trail (except for one node expression of course).

So the central question is: who are the parents of the second and third argument of `if-then-else`? The parent of the alternatives of a `case` construct has to be the `case` construct itself, because in general the alternatives may bind variables. In contrast, the two arguments of `if-then-else` are normal expressions, they do not bind variables.

8 Local Definitions, the `let` construct

Figure 10 defines the reduction of `let`. The reduction rule is straightforward. However, it is unclear what the redex trail producing rule should look like: should a `let` redex be ignored, should it be registered in the trail like other redexes (rather large), should only the defined subexpressions have a special trail which mainly declares to which defined variables they belong?

9 Comparison with other Semantics

To achieve sharing easily, Launchbury requires that arguments of functions (and later also data constructors) are variables. So a preliminary transformation is needed that introduces `let` constructs to remove non-variable arguments. For our purposes this transformation is undesirable. A redex trail shall describe the computation of the original, non-transformed program. Especially `let` redexes that were only introduced by the transformation should not be registered in the redex trail.

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

$$\frac{}{\Gamma + \{x \mapsto V\} : x \Rightarrow \Gamma + \{x \mapsto V\} : V}$$

We cannot simply give values upwards in this way, because this would conflict with the construction of redex trails. The copied values would loose their parents. Instead of giving values upwards we use the value lookup relation which we introduced especially for this purpose..

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

$$\frac{\Gamma(p) = t \cdot \texttt{let}\ \ \{x_i = M_i\}_{i=1}^{k}\ \ \texttt{in}\ \ N \quad q_i \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto t \cdot N[\overline{q}/\overline{x}]] + \{q_i \mapsto t \cdot M_i[\overline{q}/\overline{x}]\}} \ 1.\ \textsc{redex let reduction}$$

$$\frac{\Gamma(p) = t \cdot \texttt{let}\ \ \{x_i = M_i\}_{i=1}^{k}\ \ \texttt{in}\ \ N \quad q_i \notin \mathrm{dom}(\Gamma) \quad s \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto s \cdot N[\overline{q}/\overline{x}]] + \{q_i \mapsto s \cdot M_i[\overline{q}/\overline{x}]\} + \{s \mapsto t \cdot \texttt{let}\ \ \{x_i = q_i\}_{i=1}^{k}\ \ \texttt{in}\ \ p\}} \ 2.\ \textsc{redex let reduction}$$

$$\frac{\Gamma(p) = t \cdot \texttt{let}\ \ \{x_i = M_i\}_{i=1}^{k}\ \ \texttt{in}\ \ N \quad q_i \notin \mathrm{dom}(\Gamma) \quad s_i \notin \mathrm{dom}(\Gamma)}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto t \cdot N[\overline{q}/\overline{x}]] + \{q_i \mapsto s_i \cdot M_i[\overline{q}/\overline{x}]\} + \{s_i \mapsto t \cdot x_i\}} \ 3.\ \textsc{redex let reduction}$$

Figure 10: `let`

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

The major difference with respect to the graph reduction semantics of [BR00] is, that the latter explicitly models the stack whereas in the semantics given here every reduction step has to completely restart searching for the next redex. For our purposes the stack is not needed and would unnecessarily complicate the semantics.

10 Pattern Matching

This section has not been updated.

Launchbury 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.

pattern	p	$:=$	x		variable
		$\mid$	$C\,p_1 \ldots p_n$	n is arity of C	full constructor application

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:

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

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

11 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:

$$\begin{array}{ll}
 & \{1 \mapsto \text{Root} \cdot \texttt{let}\ \{head = \lambda xs.\, \texttt{case}\ xs\ \texttt{of}\ \{y : ys \mapsto y\}\}\ \texttt{in}\ head\,(\text{True} : [])\} \\
\underset{1}{\Rrightarrow} & \{1 \mapsto \text{Root} \cdot 2\,(\text{True} : []), 2 \mapsto \text{Root} \cdot \lambda xs.\, \texttt{case}\ xs\ \texttt{of}\ \{y : ys \mapsto y\}\} \\
\underset{1}{\Rrightarrow} & \{1 \mapsto \text{Root} \cdot 2\,3, 2 \mapsto \text{Root} \cdot \lambda xs.\, \texttt{case}\ xs\ \texttt{of}\ \{y : ys \mapsto y\}, 3 \mapsto \text{Root} \cdot \text{True} : []\} \\
\underset{1}{\Rrightarrow} & \{1 \mapsto 4 \cdot \texttt{case}\ 3\ \texttt{of}\ \{y : ys \mapsto y\}, 2 \mapsto \text{Root} \cdot \ldots, 3 \mapsto \text{Root} \cdot \text{True} : [] \\
 & , 4 \mapsto \text{Root} \cdot 2\,3\} \\
\underset{1}{\Rrightarrow} & \{1 \mapsto 4 \cdot \texttt{case}\ 3\ \texttt{of}\ \{y : ys \mapsto y\}, 2 \mapsto \text{Root} \cdot \ldots, 3 \mapsto \text{Root} \cdot 6 : 5 \\
 & , 4 \mapsto \text{Root} \cdot 2\,3, 5 \mapsto \text{Root} \cdot [], 6 \mapsto \text{Root} \cdot (:)\ \text{True}\} \\
\underset{1}{\Rrightarrow} & \{1 \mapsto 4 \cdot \texttt{case}\ 3\ \texttt{of}\ \{y : ys \mapsto y\}, 2 \mapsto \text{Root} \cdot \ldots, 3 \mapsto \text{Root} \cdot 6 : 5 \\
 & , 4 \mapsto \text{Root} \cdot 2\,3, 5 \mapsto \text{Root} \cdot [], 6 \mapsto \text{Root} \cdot (:)\ 7, 7 \mapsto \text{Root} \cdot \text{True}\} \\
\underset{1}{\Rrightarrow} & \{1 \mapsto 8 \cdot 7, 2 \mapsto \text{Root} \cdot \ldots, 3 \mapsto \text{Root} \cdot 6 : 5 \\
 & , 4 \mapsto \text{Root} \cdot 2\,3, 5 \mapsto \text{Root} \cdot [], 6 \mapsto \text{Root} \cdot (:)\ 7, 7 \mapsto \text{Root} \cdot \text{True} \\
 & , 8 \mapsto 4 \cdot \texttt{case}\ 3\ \texttt{of}\ \{y : ys \mapsto y\}\}
\end{array}$$

12 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 11 shows the extension of our semantics to IO.

$$\begin{array}{rrcl}
\text{expression} & M, N & := & \ldots \mid \texttt{put} \mid \texttt{get} \mid \texttt{>>=} \mid \texttt{return} \\
\text{value} & V & := & \ldots \mid \texttt{return}\, x_1 \ldots x_n \text{ with } n \geq 0 \\
\text{simple evaluation context} & \mathbb{E} & := & \ldots \mid \texttt{put}\ [\cdot] \mid [\cdot]\ \texttt{>>= p}
\end{array}$$

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

$$\frac{\Gamma \succ_p (\texttt{return}\ \ q)\ \texttt{>>=}\ r}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto r\, q]}$$

Figure 11: 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 12 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

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

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

$$\frac{\Gamma \succ_p (\texttt{return}\ q)\ \texttt{>>=}\ r}{\Gamma \underset{p}{\Rightarrow} \Gamma[p \mapsto \mathrm{IO} \cdot r\, q]}$$

Figure 12: 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.

13 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 $f x_1 \ldots x_n = M$ 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. Note that this variable only serves as a name and may not be in the domain of the heap.

14 Trusting

15 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.

16 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?

17 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?

References

[BR00] Adam Bakewell and Colin Runciman. The space usage problem: An evaluation kit for graph-reduction semantics. In *2nd Scottish Functional Programming Workshop*, September 2000.

[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.