A Semantics for Tracing

— Work in Progress —

Olaf Chitil[⋆]

University of York, UK
olaf@cs.york.ac.uk

Abstract. We define a small step operational semantics for a core of Haskell. We modify this semantics to generate traces, specifically Augmented Redex Trails. This small and direct definition of Augmented Redex Trails shall improve our understanding of them and shall help to extend them systematically.

1 Hat — The Haskell Tracer

Hat is a tool for viewing the computation of a Haskell program in various ways. Hat enables the programmer to understand how the various parts of a program cause the computation to perform the observed actions. It is useful for debugging, program comprehension and teaching [3].

Hat transforms a Haskell program into a new Haskell program. When the compiled new program is executed, it writes a trace to a file in addition to any normal I/O the original program would perform. Hat includes several separate tools for viewing the trace in various ways.

2 Aims

Hat's trace, called Augmented Redex Trail (ART), is a complex directed graph. The only formal definition of an ART is given through the transformation that enables its creation. However, this definition is inadequate. To obtain an ART of a program's computation, the program has to be transformed and then be executed to create the trail. This has the following disadvantages:

– It is hard to determine the ART of even a simple computation by hand.
– The second stage of obtaining an ART is based on the semantics of the *transformed* program. This is rather confusing, because the purpose of an ART is to give an abstract view of the computation of the *original* program.

[⋆] The work reported in this paper was supported by the Engineering and Physical Sciences Research Council of the United Kingdom under grant number GR/M81953.

– Although it is the purpose of an ART 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 an ART shall be an abstraction of a computation and that the execution of the transformed program shall yield the desired ART.

So the transformation defines an ART only in a very indirect manner, as depicted here:

$$\text{computation} \overset{semantics}{\longleftarrow} \text{program } 1 \overset{transformation}{\longrightarrow} \text{program } 2 \overset{semantics}{\longrightarrow} \text{ART}$$

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

Hence we are working on a direct definition of ARTS. Because an ART shall give an abstract view of a computation, we start with a high-level semantics of a functional language and then modify it to describe ARTs 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 ARTs will improve our understanding of them. The final consistency of the two definitions of ARTs will give us confidence that we have defined what we had in mind.
– To prove the correctness of the transformation that implements ART generation.
– To extend ARTs systematically to language constructs for which they are not yet (properly) defined, for example list comprehensions and IO.
– In [2] 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 and similar properties.

3 Design Decisions

An ART is a graph that describes the reductions of a computation and their relationships. Hence our basis has to be an operational semantics.

We have to use a call-by-need semantics — a call-by-name semantics is not sufficient — because an ART describes how expressions are shared. Especially, it is visible in an ART that when a shared expression is evaluated, it is replaced by the result at all its usage points.

We do not want to tie ourselves to a specific abstract machine. Hence Launchbury's big-step semantics for call-by-need [1] is an obvious starting point.

A big-step semantics only describes computations of expressions that have a value, that is, a weak head normal form. However, we are also interested in obtaining an ART for expressions that have value $\perp$, that is, whose evaluation 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 switch to a small step variant of Launchbury's semantics.

An ART is basically independent of the order in which the reductions of a computation are performed.[1] An ART for a terminating eager and a lazy computation of the same program may differ only in so far that the eager computation evaluates all arguments completely, whereas the lazy computation may leave some partially unevaluated. To clearly express this property of ARTs we separate our operational semantics into a definition of non-deterministic rewriting steps and an arbitrary function that chooses the next rewriting step. This evaluation strategy function may implement eager, lazy or any other evaluation order.

4 An Operational Semantics

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 comma (,) extends a graph by a node under the assumption that the node does not occur in the domain of the graph. 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.

The value lookup function defined in Figure 2 just follow chains of pointers (graph nodes). The function is undefined, if the expression points to a cycle of pointers.

In Figure 3 we define a reduction relation on graphs, where $\Gamma \Rightarrow \Delta$ means that the graph Γ reduces to the graph Δ. A computation is a finite or infinite sequence of graphs $\Gamma_1, \Gamma_2, \ldots$ with $\Gamma_i \Rightarrow \Gamma_{i+1}$ for all graphs in the sequence. The reduction relation is non-deterministic. We do not discuss the evaluation strategy here.

The computation of an expression M is the computation beginning with the graph $\{p \mapsto M\}$. M evaluates to a value, if its computation is finite with the last graph Δ having the property $\Delta(p) = V$.

The sharing rules assure 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 other rules; we do not require an evaluation context. However, most importantly we will need this property later when we modify the semantics for defining ARTs.

[1] We can easily extend ARTs to record the order of reductions. However, for most purposes it is best to ignore this order.

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
	$\mid C$	data constructor
	$\mid \texttt{case}\ M\ \texttt{of}\ \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k}$	data destruction
	$\mid \texttt{let}\ \{x_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N$	local definition
value V	$:= \lambda x.\, M$	abstraction
	$\mid C\, p_1 \ldots p_n$ with $n \geq 0$	(part.) applied constructor

Fig. 1. Syntax

$$\langle M \rangle_{\Gamma} = \begin{cases} M & \text{, if } M \text{ is not a graph node} \\ \langle \Gamma(p) \rangle_{\Gamma} & \text{, if } M = p \text{ and } p \in \mathrm{dom}(\Gamma) \end{cases}$$

Fig. 2. Value Lookup Function

$$\frac{\langle M \rangle_{\Gamma} = \lambda x.N}{\Gamma, p \mapsto M\, q \;\Rightarrow\; \Gamma, p \mapsto N[q/x]} \text{ APPLICATION REDUCTION}$$

$$\frac{\langle M \rangle_{\Gamma} = C_j\, \overline{q}}{\Gamma, p \mapsto \texttt{case}\ M\ \texttt{of}\ \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k} \;\Rightarrow\; \Gamma, p \mapsto N_j[\overline{q}/\overline{x}_j]} \text{ CASE REDUCTION}$$

$$\Gamma, p \mapsto \texttt{let}\ \{x_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N \;\Rightarrow\; \Gamma, p \mapsto N[\overline{q}/\overline{x}], \overline{q} \mapsto \overline{M[\overline{q}/\overline{x}]} \text{ LET REDUCTION}$$

SHARING RULES:

$$\Gamma, p \mapsto M\, N \;\Rightarrow\; \Gamma, p \mapsto M\, q, q \mapsto N$$

$$\Gamma, p \mapsto M\, N \;\Rightarrow\; \Gamma, p \mapsto q\, N, q \mapsto M$$

$$\Gamma, p \mapsto \texttt{case}\ M\ \texttt{of}\ \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k} \;\Rightarrow\; \Gamma, p \mapsto \texttt{case}\ q\ \texttt{of}\ \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k}, q \mapsto M$$

Fig. 3. Small Step Reduction Rules

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.

$$\langle M\rangle_\Gamma = \begin{cases} M & \text{, if } M \text{ is not a graph node} \\ \langle r\rangle_\Gamma & \text{, if } M = p \text{ and } \Gamma(p) = {}^{s}_{r}N \text{ and } r \neq \Box \\ \langle \Gamma(p)\rangle_\Gamma & \text{, otherwise} \end{cases}$$

Fig. 4. Value Lookup Function for ARTs

$$\frac{\langle M\rangle_\Gamma = \lambda x.N}{\Gamma, p \mapsto^{s}_{\Box} M\, q \;\Rightarrow\; \Gamma, p \mapsto^{s}_{r} M\, q, r \mapsto^{p}_{\Box} N[q/x]}\ \text{APPLICATION REDUCTION}$$

$$\frac{\langle M\rangle_\Gamma = C_j\, \overline{q}}{\begin{array}{c}\Gamma, p \mapsto^{s}_{\Box} \texttt{case}\ M\ \texttt{of}\ \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k} \\ \Rightarrow \Gamma, p \mapsto^{s}_{r} \texttt{case}\ M\ \texttt{of}\ \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k}, r \mapsto^{p}_{\Box} N_j[\overline{q}/\overline{x}_j]\end{array}}\ \text{CASE REDUCTION}$$

$$\frac{}{\begin{array}{c}\Gamma, p \mapsto^{s}_{\Box} \texttt{let}\ \{x_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N \\ \Rightarrow \Gamma, p \mapsto^{s}_{r} \texttt{let}\ \{x_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N, r \mapsto^{s}_{\Box} N[\overline{q}/\overline{x}], \overline{q \mapsto^{p}_{\Box} M[\overline{q}/\overline{x}]}\end{array}}\ \text{LET REDUCTION}$$

SHARING RULES:

$$\Gamma, p \mapsto^{s}_{r} M\, N \;\Rightarrow\; \Gamma, p \mapsto^{s}_{r} M\, q, q \mapsto^{s}_{\Box} N$$

$$\Gamma, p \mapsto^{s}_{r} M\, N \;\Rightarrow\; \Gamma, p \mapsto^{s}_{r} q\, N, q \mapsto^{s}_{\Box} M$$

$$\Gamma, p \mapsto^{s}_{r} \texttt{case}\ M\ \texttt{of}\ \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k} \;\Rightarrow\; \Gamma, p \mapsto^{s}_{r} \texttt{case}\ q\ \texttt{of}\ \{C_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k}, q \mapsto^{s}_{\Box} M$$

Fig. 5. Small Step Reduction Rules for Defining ARTs

5 An Operational Semantics Defining ARTs

An ART shall describe a complete computation. However, the reduction rules update expressions in the graph and thus information is lost. To preserve the information we do not update any expression but allocate a new graph node for a reduction result. We extend each graph node by a pointer pointing to the reduction result of the graph node, if any. Hence we also introduce the null pointer $\Box$ for graph nodes that do not have a result. This is the case if the expression of the node is a value or has not (yet) been reduced.

Furthermore, each node has a second pointer to its parent, that is, the node whose reduction caused the creation of the node.

We write $p \mapsto^{s}_{r} M$ for a node p with expression M, parent s and result r. Figure 5 gives the modified semantics that defines ARTs. The value lookup function has to be modified to follow result pointers, see Figure 4.

6 Conclusions

The definition does not yet cover all aspects of ARTs and much remains to be done.

References

1. 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.
2. 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.
3. Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, UU-CS-2001-23. Universiteit Utrecht, 2001.