Structure and Properties of Traces for Functional Programs

Olaf Chitil and Yong Luo

Computing Laboratory
University of Kent
Canterbury, United Kingdom

Abstract

The tracer Hat records in a detailed trace the computation of a program written in the lazy functional language Haskell. The trace can then be viewed in various ways to support program comprehension and debugging. The trace was named the *augmented redex trail*. Its structure was inspired by standard graph rewriting implementations of functional languages. Here we describe a model of the trace that captures its essential properties and allows formal reasoning. The trace is a graph constructed by graph rewriting but goes beyond simple term graphs. Although the trace is a graph whose structure is independent of any rewriting strategy, we define the trace inductively, thus giving us a powerful method for proving its properties.

Keywords: Tracing, debugging, augmented redex trail, Haskell.

1 Introduction

Usually, a computation is treated as a black box that performs input and output actions. However, we have to look into the black box when we want to see how the different parts of the program cause the computation to perform the input/output actions. The most common need for doing this is debugging: When there is a disparity between the actual and the intended semantics of a program, we need to locate the part of the program that causes the disparity. Other reasons for observing how a program works are checking internal consistency properties (cf. assertions), reverse-engineering of an insufficiently documented program, and learning to program. Tracing is the process of obtaining additional information about the internal workings of a computation.

1.1 Tracing Functional Programs

Conventional forms of tracing are the introduction of specific statements in a program for logging information about the computation progress and the use of debuggers for stepping through a computation and inspecting the computation state. For lazy functional languages different tracing methods have been developed for two reasons. First, conventional methods are unsuitable, because intermediate terms with large unevaluated subterms are hard to read and the lazy reduction strategy is too complex for human programmers to follow. Second, conventional tracing methods reflect only the computational model of imperative programs: a long sequence of state transformations. In contrast, functional programmers want to ignore low-level operational details, in particular evaluation order, but take advantage of properties such as explicit data flow and absence of side effects.

1.2 What is a Trace?

A trace is a structure that consists of information about a computation. It (partially) describes how the computation of a program obtained its outputs from its inputs. If computation is deterministic (as we assume here), then the complete computation is already determined by the program and its inputs. So is a trace hence superfluous? No. Any detail of a computation can be reconstructed from program and inputs by rerunning the computation, but that would be expensive. The purpose of a trace is to provide quick and easy access to any desired information. Most viewing tools are interactive; hence they have to provide demanded bits of information quickly, in particular independent of the length of the computation.

Operational semantics provide descriptions of computations. Both a sequence of terms $M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow \ldots$ of a small-step operational semantics and the proof tree of a big-step natural semantics are descriptions of computations. However, they are unsuitable as traces not just because they do not provide quick access to all desired information, but more obviously because they are full of redundancies. In both these descriptions most parts of terms are replicated many times. A trace has to be as compact as possible to describe long computations.

1.3 Existing Tracing Systems

Several tracing systems for lazy functional languages are available, all for Haskell [26,19,12,27,22]. All systems take a two-phase approach to tracing:

(i) During the computation information about the computation is recorded in a data structure, the trace.

(ii) After termination of the computation the trace is used to view the computation. Usually an interactive tool displays fragments of the computation on demand. The programmer uses their knowledge of the intended behaviour of the program to locate faults.

A trace is a complex graph of term components; most trace structures also incorporate links to the source program. The trace as concrete data structure liberates the viewer from time and the sequential evaluation strategy.

Each tracing method gives a different view of a computation; in practice, the views are complementary and can productively be used together [7]. Hence the Haskell tracer Hat integrates several methods [27]. During a computation a single unified trace, the *augmented redex trail (ART)*, is generated. Separate tools provide different views of the ART, for example algorithmic debugging [23,19,22], following redex trails [26] and observing functions [12].

1.4 The Aim: A Theory of Tracing

Hat transforms a Haskell program into a new Haskell program. When the compiled new program is executed, it writes the ART to a file in addition to any normal I/O the original program would perform. This indirect definition of the ART through program transformation makes it hard to determine the ART of even a simple computation by hand. Because two programs are involved, the original and the transformed one, it is hard to disentangle which assumptions about the semantics of each are made. The ART also includes many special constructs, because Haskell is a large and complex language.

Therefore our aim is to give a direct and simple definition of the ART that captures its essential properties and will enable us to formally relate a view to the semantics of a program. Thus viewed information has a clear meaning and can be used correctly to understand a program and locate program faults.

The need for formalising tracing systems has been recognised:

> "Programmers rely on the correctness of the tools in their programming environments. In the past, semanticists have studied the correctness of compilers and compiler analyses, which are the most important tools. In this paper, we make the case that other tools, such as debuggers and steppers, deserve semantic models, too, and that using these models can help in developing these tools." [9]

We concentrate on the ART because it was already distilled as a unified trace from several other traces. This focus on the ART does not preclude revisions of its definition in the light of new insights. We are aware of several shortcomings (lack of information) that we intend to remove. Although the ART is only used for Haskell, it is suitable for both strict and non-strict pure functional languages, as our definition shall clarify.

1.5 Structure of the Paper

We first motivate our choice of programming language in Section 2. Then we give in Section 3 an informal introduction to the ART based on its origin in graph rewriting implementations of functional languages. Sections 4 and 5 define ARTs. In Section 6 we prove some simple properties of ARTs to get a better understanding and to demonstrate that our definitions allow relatively simple proofs. We prove in Section 7 that ARTs are *acyclic* directed graphs and in Section 9 that their structure is independent of any particular evaluation order. Section 8 establishes a property central to using ARTs for producing views of computations. Section 10 shows how ARTs can express runtime errors. We discuss related work in Section 11 and conclude in Section conclusion.

2 Programs

Operational semantics for functional languages that describe sharing, in particular the widely used semantics of Launchbury [15] and its variations, are defined for a subset of the λ-calculus plus a `case` construct for basic pattern matching. The constructs of this core calculus were chosen to optimise the simplicity of the operational semantics. Any functional program can be translated into this core calculus and hence the core calculus is suitable for compilers. However, programs written in the core calculus look rather different from programs written by a programmer as the core calculus does not contain programming constructs mostly used in programs. Because a trace supports a programmer in understanding how a program works, it is essential that the trace relates to the program as originally written. Hence we choose a calculus that contains the most frequently used program language features: named functions and pattern matching. Our programs are applicative term rewriting systems [14].

function symbol	f, g, h			
data constructor	C, D			
atom	a, b	$:=$	f	function symbol
		$\mid$	C	data constructor
variable	x, y, z			
program term	M, N	$:=$	a	atom
		$\mid$	x	variable
		$\mid$	$M\,N$	application
pattern	P	$:=$	x	variable
		$\mid$	Q	constructor pattern
constructor pattern	Q	$:=$	C	constructor
		$\mid$	$Q\,P$	constructor application

We write $M\,N_1 \ldots N_n$ for the program term $(...((M\,N_1)\,N_2)\ldots)N_n$ and say that it is an application of M to n arguments. Each atom is associated with a natural number, its *arity*. Function symbols of arity 0 are sometimes called constant symbols.

Definition 2.1 If f is a function symbol of arity n and $P_1 \ldots P_n$ are patterns and R is a program term such that the variables of R are a subset of the variables of $f P_1 \ldots P_n$, then $f P_1 \ldots P_n = R$ is a *rewrite rule*. A *program* is a finite set of rewrite rules.

The following example program implements insertion sort. We write applications of function symbols in infix or mixfix notation where this is common practice.

```
sort [] = []
sort (x:xs) = insert x (sort xs)

insert x [] = [x]
insert x (y:ys) = if x <= y then x:y:ys else y : insert x ys

if True then x else y = x
if False then x else y = y
```

We do admit higher-order programs. For example, we could use the following alternative definition of `sort`:

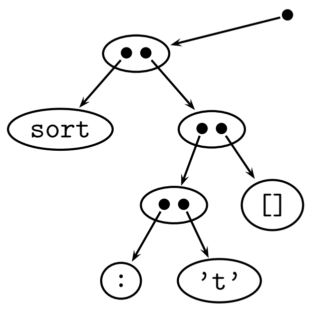
```
sort = foldr insert []
```

The definition of programs lists all the constraints we need for the properties we prove. It is more general than required for writing functional programs. The definition does not require that all data constructors in patterns are fully applied, that is, the number of their arguments is equal to their arity. This constraint would only be needed for establishing a completeness of the operational rewriting semantics with respect to a denotational semantics. Our programs are *not* orthogonal rewriting systems [14]. We do not require rules to be left-linear; a variable may occur repeatedly in the left-hand side of the rule. In term rewriting left-linearity is already desirable to avoid an equality check of arbitrarily large terms, but in term graph rewriting there is no such expensive equality check anyway, because a variable is matched with a graph node. An orthogonal term rewriting system is confluent. We explicitly allow the left-hand sides of two rules to have a common instance and thus overlap. Thus we can leave the choice between these rules either to a separate evaluation strategy whose definition we leave open or we can model non-deterministic functions as they appear in functional logic programs. However, a left-hand side of a rule cannot overlap with a proper subterm of the left-hand side of a rule. Hence our programs are not subcommutative but almost subcommutative (Corollary 9.3).

3 The Origin of the Augmented Redex Trail

Term graph rewriting [21] provides an operational semantics for functional programs that is abstract and closely related to standard term rewriting semantics. In contrast to terms, graphs allow the sharing of common subterms as it happens in real implementations of functional languages, such as the G-machine [13] or the more efficient STG-machine [20]. Term graph rewriting can correctly model the asymptotic time and space complexity of real implementations [1]. For us sharing is the key for a space efficient trace structure and closeness to the implementation also promises easy creation of a trace.

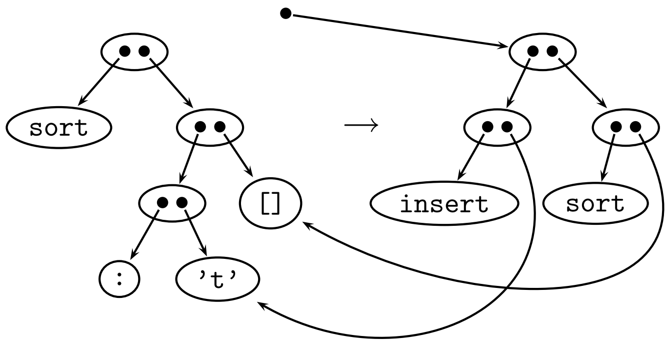
Consider the representation of the Haskell term `sort ['t']` as term graph (`['t']` is syntactic sugar for `(:) 't' []`):



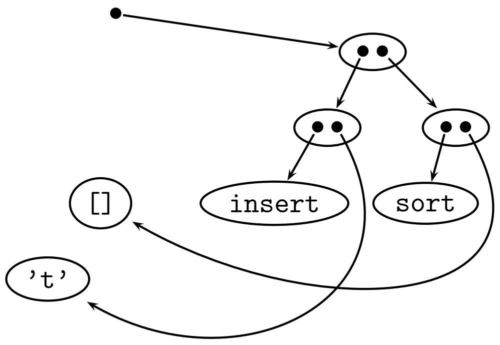
The function `sort` is defined by the following two rewrite rules:

```
sort [] = []
sort (x:xs) = insert x (sort xs)
```

We perform a reduction step with the second equation:



For this reduction step we have added new graph nodes that represent the right-hand side of the used rewrite rule. We have a new root node denoting the top of the term. To ensure that the new top node is correctly shared (when the redex is only part of a larger term), implementations actually *overwrite* the top node of the redex with the top node of the reduct. Some graph nodes become unreachable from the root and are removed by a garbage collector:



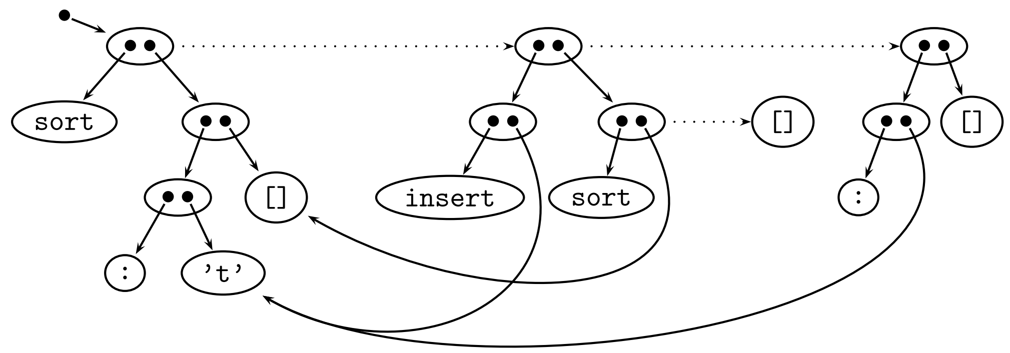
It is vital for the efficiency of graph rewriting that each reduction step only adds a small number of new nodes; subterms that are bound to the variables of the rewrite rule are shared with the redex graph and hence need not be added.

For tracing a computation we simply do not overwrite the redex with the reduct, but construct the reduct separately (sharing subterms with the redex as above) and connect the top nodes of redex and reduct by a special reduction edge.

Using also the rewrite rules

```
sort [] = []
insert x [] = [x]
```

we obtain the following trace of a complete computation (dotted arrows are reduction edges):



This is the augmented redex trail (ART) of the computation. It is a compact but detailed representation of the computation; in particular, it directly relates each redex with its reduct. Its creation in both practise and theoretical model is greatly simplified by the fact that each reduction step only *adds* to the trace but *never modifies* the already existing parts.

As described here the ART has no information about the order of reduction steps. It does not say that `sort []` $\rightarrow$ `[]` happened before `insert 't' []` $\rightarrow$ `['t']`. This information is irrelevant for most views provided by Hat. This observation agrees with our idea that functional programmers abstract from time. Because Hat writes the nodes of the ART into file in the order of their creation, timing information is actually available. If it later proves necessary, we can easily add it to our model.

4 Defining Term Graphs

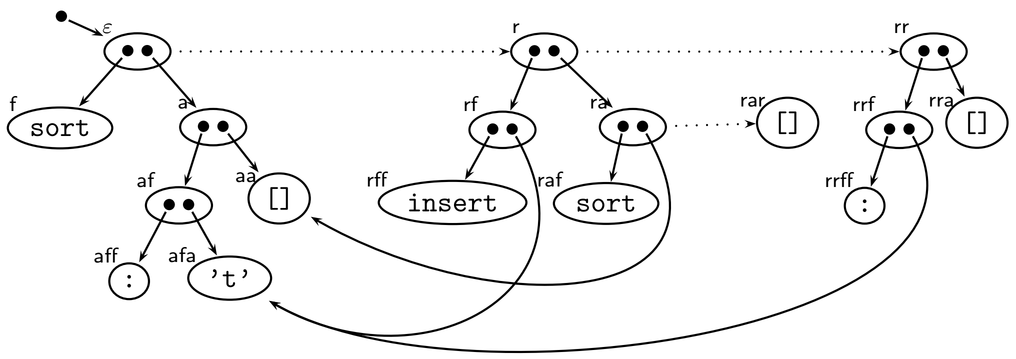
Graphs have the disadvantage that the choice of node and edge elements is usually irrelevant because we are mostly interested in the labels. The notion of graph morphism becomes central, because we do not want to distinguish isomorphic graphs. Isomorphism classes of graphs are inconvenient to handle. Hence we choose a standard representation of graphs where a node describes the path from the root of the graph to the node. A node is a (possibly empty) string of the letters f, a and r, where f means going to the function component of an application, a means going to the argument component of an application, and r means following a reduction edge to the reduct.

$$\text{node letter} \qquad i, j := \mathsf{f} \mid \mathsf{a} \mid \mathsf{r}$$

$$\text{node } n, m, o \in \{\mathsf{f}, \mathsf{a}, \mathsf{r}\}^*$$

Because of sharing, several paths may lead to the same node in the graph. However, in our term graphs there exist canonical paths. Every node is first introduced as part of an unshared representation of a term, either the initial term of the computation or a right-hand side of a rewriting rule. In the first case the canonical path of the node is the unique path within the initial term. In the second case the canonical path of the node consists of the canonical path of the redex node plus the unique path from the redex node through the right-hand side to the node itself. So a node contains useful information in that it describes its position within a reduct (or initial term) and the sequence of redex nodes that lead to its creation.

Here our previous example with node identities:



Each node of a term graph has a label, which may point to further nodes. We define a term graph such that it is as simple as possible. Then we discuss some design decisions in detail.

Definition 4.1

$$\begin{array}{rll} \text{label } T := & a & \text{atom} \\ | & n\,m & \text{application} \\ | & n & \text{indirection} \end{array}$$

A *term graph* is a partial function from nodes to labels, $\mathcal{G} : n \mapsto T$. The domain $\text{dom}(\mathcal{G})$ of term graph $\mathcal{G}$ is the set of nodes for which the function is defined. We sometimes regard a term graph $\mathcal{G}$ as a set of tuples $\{(n, \mathcal{G}(n)) \mid n \in \text{dom}(\mathcal{G})\}$.

Let the *free nodes of a term graph* be the nodes that occur in its labels but not its domain. A term graph is *closed* if it has no free nodes.

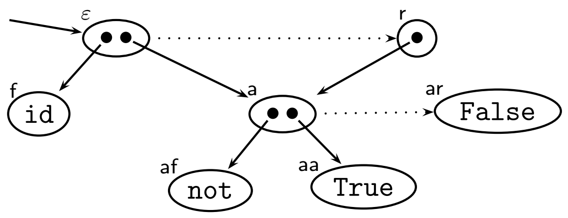
Our example graph pictured before is formally defined by

$$\begin{array}{llll} \mathcal{G}(\varepsilon) = \mathsf{f}\ \mathsf{a} & \mathcal{G}(\mathsf{f}) = \texttt{sort} & \mathcal{G}(\mathsf{a}) = \mathsf{af}\ \mathsf{aa} & \mathcal{G}(\mathsf{af}) = \mathsf{aff}\ \mathsf{afa} \\ \mathcal{G}(\mathsf{aff}) = \texttt{:} & \mathcal{G}(\mathsf{afa}) = \texttt{'t'} & \mathcal{G}(\mathsf{aa}) = \texttt{[]} & \mathcal{G}(\mathsf{r}) = \mathsf{rf}\ \mathsf{ra} \\ \mathcal{G}(\mathsf{rf}) = \mathsf{rff}\ \mathsf{afa} & \mathcal{G}(\mathsf{rff}) = \texttt{insert} & \mathcal{G}(\mathsf{ra}) = \mathsf{raf}\ \mathsf{aa} & \mathcal{G}(\mathsf{raf}) = \texttt{sort} \\ \mathcal{G}(\mathsf{rar}) = \texttt{[]} & \mathcal{G}(\mathsf{rr}) = \mathsf{rrf}\ \mathsf{rra} & \mathcal{G}(\mathsf{rrf}) = \mathsf{rrff}\ \mathsf{afa} & \mathcal{G}(\mathsf{rrff}) = \texttt{:} \\ \mathcal{G}(\mathsf{rra}) = \texttt{[]} & & & \end{array}$$

So $\text{dom}(\mathcal{G}) = \{\varepsilon, \mathsf{f}, \mathsf{a}, \mathsf{af}, \mathsf{aff}, \mathsf{afa}, \mathsf{aa}, \mathsf{r}, \mathsf{rf}, \mathsf{rff}, \mathsf{ra}, \mathsf{raf}, \mathsf{rar}, \mathsf{rr}, \mathsf{rrf}, \mathsf{rrff}, \mathsf{rra}\}$ and the set of labels of $\mathcal{G}$ is $\{\mathsf{f}\ \mathsf{a}, \texttt{sort}, \mathsf{af}\ \mathsf{aa}, \mathsf{aff}\ \mathsf{afa}, \texttt{:}, \texttt{'t'}, \texttt{[]}, \ldots\}$.

We do not explicitly include reduction edges in the graph, because they are already implicitly given through our choice of nodes. A reduction edge always points from a node n to the node $n\mathsf{r}$. If and only if node $n\mathsf{r}$ exists in the graph, there is a reduction edge from node n to node $n\mathsf{r}$.

So to record a reduction we have to add at least one new node. This requirement causes a problem when we reduce with a projection, such as `id x = x`, because we do not want to copy any node. For this very purpose we have the indirection label; it appears only as reduct of a projection.



The solution of using an indirection node was inspired by graph machines for functional languages that use indirection nodes for similar purposes. The existence of an indirection node is also vital for traversing a term graph backwards, from (parts of) reducts to redexes [25].

Later we will often need the last node of a chain of edges where each edge is either a reduction edge or an edge from an indirection node:

Definition 4.2 Let $\mathcal{G}$ be a term graph and $n \in \mathrm{dom}(\mathcal{G})$. Then

$$\lceil n \rceil_{\mathcal{G}} = \text{if } n\mathsf{r} \in \mathrm{dom}(\mathcal{G}) \text{ then } \lceil n\mathsf{r} \rceil_{\mathcal{G}} \text{ else if } \exists m.\mathcal{G}(n) = m \text{ then } \lceil m \rceil_{\mathcal{G}} \text{ else } n$$

For example, in the term graph $\mathcal{G}$ above $\lceil \varepsilon \rceil_{\mathcal{G}} = \mathsf{ar}$ and $\lceil \mathsf{f} \rceil_{\mathcal{G}} = \mathsf{f}$.

5 Defining Augmented Redex Trails

Labels are not nested. Labels do not include variables, because in rewriting variables will be replaced by nodes. Program terms do not include nodes. To describe rewriting we will need terms that include nodes.

Definition 5.1

$$\begin{array}{rcll} \text{term } M, N & := & a & \text{atom} \\ & \mid & n & \text{node} \\ & \mid & x & \text{variable} \\ & \mid & M\ N & \text{application} \end{array}$$

Terms contain both nodes and variables. A *label term* is a term that does not contain variables. A *program term* is a term that does not contain nodes (as defined before). A *computation term* is a term that contains neither variables nor nodes.

Not every term graph represents a computation. Augmented redex trails (ARTs) are defined inductively. The graph representation of an initial term M, graph(ε, M), is an ART. If $\mathcal{G}$ is an ART and $\mathcal{G}$ reduces in one step with program P to $\mathcal{G}'$, that is, $\mathcal{G} \rightarrow_P \mathcal{G}'$, then $\mathcal{G}'$ is an ART. Even shorter:

Definition 5.2 Let P be a program and M a computation term. A term graph $\mathcal{G}$ with $\text{graph}(\varepsilon, M) \rightarrow_P^* \mathcal{G}$ is an *augmented redex trail (ART) for initial term M and program P.*

An ART is constructed by a finite number of reduction steps. The definition of ARTs could be extended to cover infinite, non-terminating computations, but then it would no longer be an inductive definition that enables simple proofs. In practice we can produce a trace only for a finite number of steps; we cannot produce some limit of an infinite number of steps.

To correctly describe sharing of subterms between the redex and the reduct of a reduction step, we have to substitute nodes (instead of terms) for the variables of a rewrite rule. Hence several subsequent definitions handle label terms.

First we have to define what the graph for a given label term is.

Definition 5.3

$$\begin{aligned}
\text{graph}(n, a) &= \{(n, a)\} \\
\text{graph}(n, m) &= \{(n, m)\} \\
\text{graph}(n, M\,N) &= \begin{cases} \{(n, M\,N)\} & \text{, if } M,\ N \text{ are nodes} \\ \{(n, M\,n\mathsf{a})\} \cup \text{graph}(n\mathsf{a}, N) & \text{, if only } M \text{ is a node} \\ \{(n, n\mathsf{f}\ N)\} \cup \text{graph}(n\mathsf{f}, M) & \text{, if only } N \text{ is a node} \\ \{(n, n\mathsf{f}\ n\mathsf{a})\} \cup & \text{, otherwise} \\ \qquad \text{graph}(n\mathsf{f}, M) \cup \text{graph}(n\mathsf{a}, N) & \end{cases}
\end{aligned}$$

There is no optimisation to share common subterms of the given label term; for example, we do not share the common subterm `f 42` of the term `if g fa then f 42 else (f 42) + 1`.

Lemma 5.4 *Let n, m and o be any nodes, i a letter and M a label term. Let $\mathcal{G} = \mathit{graph}(n, M)$.*

(i) *$\mathcal{G}$ is a term graph.*

(ii) *$\mathit{dom}(\mathcal{G}) \subset n\{\mathsf{f}, \mathsf{a}\}^*$.*

(iii) *$mi \in \mathit{dom}(\mathcal{G})$ implies $m \in \mathit{dom}(\mathcal{G})$ or $mi = n$.*

(iv) *The free nodes of $\mathcal{G}$ are a subset of the nodes occurring in M.*

(v) *If M contains no node ending in r, then the labels of $\mathcal{G}$ contain no node ending in r.*

(vi) *$o \in \mathit{dom}(\mathcal{G}) \wedge \mathcal{G}(o) = m \implies o = n \wedge M = m$.*

(vii) *$n\mathsf{f} \in \mathit{dom}(\mathcal{G}) \implies \exists m.\mathcal{G}(n) = n\mathsf{f}\,m$ and*
$n\mathsf{a} \in \mathit{dom}(\mathcal{G}) \implies \exists m.\mathcal{G}(n) = m\,n\mathsf{a}$.

Proof. By induction on M. To show that $\text{graph}(n, M)$ is a term graph we show that there is only one tuple for each node in the set representation. □

The central part of a reduction step is matching a node in the graph with the left-hand side of a rewrite rule. The predicate $\text{match}_{\mathcal{G}}(n, M)$ checks whether node n matches label term M in term graph $\mathcal{G}$.

Definition 5.5 Matching is defined inductively over the structure of the matched label term:

$$
\begin{aligned}
\text{match}_{\mathcal{G}}(o, a) &= (\mathcal{G}(o) = a)\\
\text{match}_{\mathcal{G}}(o, M\,N) &= \exists m, n.(\mathcal{G}(o) = m\,n) \wedge\\
&\qquad\qquad \text{match}_{\mathcal{G}}(\text{if } M \text{ is a node then } m \text{ else } \lceil m \rceil_{\mathcal{G}}, M) \wedge\\
&\qquad\qquad \text{match}_{\mathcal{G}}(\text{if } N \text{ is a node then } n \text{ else } \lceil n \rceil_{\mathcal{G}}, N)\\
\text{match}_{\mathcal{G}}(o, m) &= (o = m)
\end{aligned}
$$

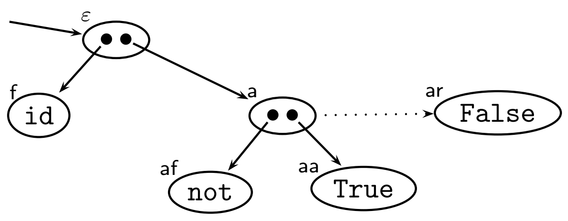
Before we discuss this definition of matching we define also what a reduction step is:

Definition 5.6 The *reduction relation* $\rightarrow_P$ on term graphs for program P is defined as follows. If

- $\mathcal{G}$ is a term graph with $n \in \text{dom}(\mathcal{G})$ and $n\mathsf{t} \notin \text{dom}(\mathcal{G})$,
- $L = R$ is a rewrite rule of the program P,
- σ is a substitution replacing variables by nodes,
- $\text{match}_{\mathcal{G}}(n, L\sigma)$,

then $\mathcal{G} \rightarrow_{P,n} \mathcal{G} \cup \text{graph}_{\mathcal{G}}(n\mathsf{r}, R\sigma)$ with rewrite rule $L = R$ and substitution σ.

Matching has been defined with care to ensure that a node in the label term matches a node in the graph only if it appears directly as part of an application node in the graph. Consider the ART



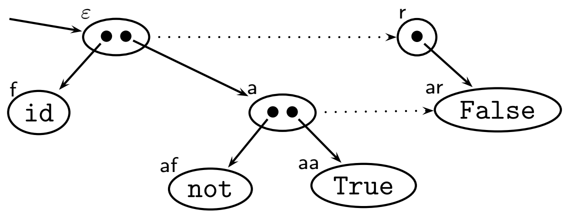
and the rewrite rule `id x = x`. We find that the label term $\texttt{id x}[\mathsf{a}/\mathsf{x}] = \texttt{id}\,\mathsf{a}$ matches at node ε:

$$
\begin{aligned}
&\text{match}_{\mathcal{G}}(\varepsilon, \texttt{id}\,\mathsf{a})\\
=&(\mathcal{G}(\varepsilon) = \mathsf{f}\ \mathsf{a}) \wedge \text{match}_{\mathcal{G}}(\text{if } \texttt{id} \text{ is a node then } \mathsf{f} \text{ else } \lceil \mathsf{f} \rceil_{\mathcal{G}}, \texttt{id}) \wedge\\
&\qquad\qquad\qquad \text{match}_{\mathcal{G}}(\text{if } \mathsf{a} \text{ is a node then } \mathsf{a} \text{ else } \lceil \mathsf{a} \rceil_{\mathcal{G}}, \mathsf{a})\\
=&\text{true} \wedge \text{match}_{\mathcal{G}}(\lceil \mathsf{f} \rceil_{\mathcal{G}}, \texttt{id}) \wedge \text{match}_{\mathcal{G}}(\mathsf{a}, \mathsf{a})\\
=&\text{match}_{\mathcal{G}}(\mathsf{f}, \texttt{id}) \wedge \text{match}_{\mathcal{G}}(\mathsf{a}, \mathsf{a})\\
=&\text{true}
\end{aligned}
$$

So in one step the ART can be rewritten to the ART shown in the previous section. However, label term $\texttt{id x}[\mathsf{ar}/\mathsf{x}] = \texttt{id}\,\mathsf{ar}$ does *not* match at node ε:

$$\begin{aligned}
&\text{match}_{\mathcal{G}}(\varepsilon, \texttt{id}\,\mathsf{ar})\\
=&(\mathcal{G}(\varepsilon) = \mathsf{f}\ \mathsf{a}) \wedge \text{match}_{\mathcal{G}}(\text{if } \texttt{id} \text{ is a node then } \mathsf{f} \text{ else } \lceil \mathsf{f} \rceil_{\mathcal{G}}, \texttt{id}) \wedge\\
&\qquad\qquad \text{match}_{\mathcal{G}}(\text{if } \mathsf{ar} \text{ is a node then } \mathsf{a} \text{ else } \lceil \mathsf{a} \rceil_{\mathcal{G}}, \mathsf{ar})\\
=&\text{true} \wedge \text{match}_{\mathcal{G}}(\lceil \mathsf{f} \rceil_{\mathcal{G}}, \texttt{id}) \wedge \text{match}_{\mathcal{G}}(\mathsf{a}, \mathsf{ar})\\
=&\text{match}_{\mathcal{G}}(\mathsf{f}, \texttt{id}) \wedge \text{match}_{\mathcal{G}}(\mathsf{a}, \mathsf{ar})\\
=&\text{true} \wedge \text{false}\\
=&\text{false}
\end{aligned}$$

So our graph *cannot* be rewritten to the term graph



Indeed this term graph is *not* an ART. Why do we explicitly exclude it? Well, otherwise we would have several structurally different graphs representing the same reduction step. Multiple representations just cause confusion and would later lead us to give a complex definition of an equivalence class of graphs. Instead we choose a canonical representation: in a reduction step we only replace a variable by a node that is the start node of a possibly long sequence of reduction and indirection edges. Thus a substitution for matching a program term is basically uniquely defined:

Lemma 5.7 *Let $\mathcal{G}$ be a term graph with match$_{\mathcal{G}}(m, M\sigma)$ and match$_{\mathcal{G}}(m, M\sigma')$ for a program term M and substitutions σ and σ' replacing variables by nodes. Then $\sigma(x) = \sigma'(x)$ for all variables x occurring in M.*

Proof. Induction on M.

(i) $M = x$: match$_{\mathcal{G}}(m, x\sigma) \Rightarrow m = \sigma(x)$. match$_{\mathcal{G}}(m, x\sigma') \Rightarrow m = \sigma'(x)$. Hence $\sigma(x) = \sigma'(x)$.

(ii) $M = a$: Trivial, because there are no variables occurring in a.

(iii) $M = N\,O$: match$_{\mathcal{G}}(m, (N\,O)\sigma)$ and match$_{\mathcal{G}}(m, (N\,O)\sigma')$ imply that there exist n,o, n' and o' such that $\mathcal{G}(m) = n\,o$ and match$_{\mathcal{G}}(n', N\sigma)$, match$_{\mathcal{G}}(o', O\sigma)$, match$_{\mathcal{G}}(n', N\sigma')$ and match$_{\mathcal{G}}(o', O\sigma')$. With the induction hypothesis follows that $\sigma(x) = \sigma'(x)$ for all variables in N and O, that is, all variables in $N\,O$.

□

Why did we choose the start node of a sequence of reduction and indirection edges instead of, for example, the last one? If we chose a different node the graph would contain some partial (but not complete) information about the evaluation order and term graph reduction would no longer be almost subcommutative. These properties will be expressed in Proposition 9.2 and Corollary 9.3. In the practical implementation of Hat it also turned out that the choice of the start node is the easiest to implement.

Lemma 5.8 *Let $\mathcal{G}$ be a closed term graph with $match_{\mathcal{G}}(n, L\sigma)$ for a left-hand side L of a rewrite rule and a substitution σ replacing variables by nodes. Then:*

(i) *n is an application or a constant symbol node.*

(ii) *The nodes occurring in $L\sigma$ are a subset of the nodes appearing in labels of $\mathcal{G}$.*

(iii) *The nodes occurring in $L\sigma$ are a subset of $dom(\mathcal{G})$.*

Proof.

(i) L is an application or a constant symbol, thus so is $L\sigma$. Then the property follows straight from the definition of match.

(ii) By induction on the definition of match.

(iii) Follows from ii because $\mathcal{G}$ is closed.

□

Definition 5.9 Let $\mathcal{G}$ be an ART with $n \in \text{dom}(\mathcal{G})$ and $n\mathsf{r} \in \text{dom}(\mathcal{G})$. Then we call n a *redex node* and $n\mathsf{r}$ a *reduct node* of $\mathcal{G}$.

6 Simple Properties of the ART

In this section we list a number of properties of the ART that give a feeling for the structure of an ART and the constraints it meets compared to an arbitrary term graph. Such simple properties can also be of practical value: Hat comprises a tool called `hat-check` that performs a number of sanity checks on an ART generated by Hat. This testing tool proved very valuable for discovering faults in the Hat tracer. The checks of `hat-check` are relatively low-level and implementation dependent, but should be extended to include most of the properties listed here. The proof of each property takes advantage of the inductive definition of an ART and uses some related properties of matching and graph construction.

Definition 6.1 A set of nodes S is prefix-closed if $nm \in S \implies n \in S$.

Lemma 6.2 *A node set S is prefix-closed if and only if for all nodes n and letters i, $ni \in S \implies n \in S$.*

Proof. By induction on the node length. □

Proposition 6.3 *The domain of an ART $\mathcal{G}$ is* prefix-closed.

Proof. We prove by induction on the definition of ARTs that for all nodes n and letters i, $ni \in \text{dom}(\mathcal{G}) \implies n \in \text{dom}(\mathcal{G})$. In both base and inductive case we need Lemma 5.4.iii. Finally prefix-closedness follows with Lemma 6.2. □

Proposition 6.4 *An ART is* closed.

Proof. Induction on the definition of ARTs. Either $\mathcal{G} = \text{graph}(\varepsilon, M)$ for an initial term M; then, according to Lemma 5.4.iv, $\mathcal{G}$ has no free nodes. Otherwise we assume $\mathcal{G}$ is closed and have to prove that $\mathcal{G} \cup \text{graph}(n\mathsf{r}, R\sigma)$ is closed. We know from $\text{match}_{\mathcal{G}}(n, L\sigma)$ and Lemma 5.8.iii that the nodes occurring in $L\sigma$ are a subset of $\text{dom}(\mathcal{G})$. Because R contains no nodes and the variables of R are a subset of the variables of L, the nodes occurring in $R\sigma$ are a subset of $\text{dom}(\mathcal{G})$. With Lemma 5.4.iv follows that the free nodes of $\text{graph}(n, R\sigma)$ are a subset of $\text{dom}(\mathcal{G})$. So $\mathcal{G} \cup \text{graph}(n\mathsf{r}, R\sigma)$ is closed. □

Proposition 6.5 *No label of an ART contains a node ending in* r*.*

Proof. Induction on the definition of ARTs. Consider $\mathcal{G} = \text{graph}(\varepsilon, M)$. The initial term M contains no nodes and thus according to Lemma 5.4.v $\mathcal{G}$ contains no node ending in r. Otherwise let $\mathcal{G}$ be an ART such that no label contains a node ending in r. With $\text{match}_{\mathcal{G}}(n, L\sigma)$ follows according to Lemma 5.8.ii that $L\sigma$ contains no node ending in r. Because the variables of R are a subset of the variables of L and R contains no nodes, $R\sigma$ contains no node ending in r. With Lemma 5.4.v follows that the labels of $\text{graph}(n, R\sigma)$ contain no node ending in r. So the labels of $\mathcal{G} \cup \text{graph}(n, R\sigma)$ contain no node ending in r. □

Proposition 6.6 *In an ART only a node ending in* r *can be an indirection. That is, if* $\mathcal{G}(n) = m$*, then* n *ends in* r*.*

Proof. Induction on the definition of ARTs. Case $\mathcal{G} = \text{graph}(\varepsilon, M)$: The initial term M is not a node, so according to Lemma 5.4.vi there is no n with $\mathcal{G}(n) = m$. Case $\mathcal{G} = \mathcal{G}' \cup \text{graph}(o\mathsf{r}, R\sigma)$: Let $n \in \text{dom}(\mathcal{G})$ such that $\mathcal{G}(n) = m$. Either $n \in \text{dom}(\mathcal{G}')$ and then according to the induction hypothesis n ends in r. Or $n \in \text{dom}(\text{graph}(o\mathsf{r}, R\sigma))$. Then according to Lemma 5.4.vi $n = o\mathsf{r}$, so n ends in r. □

Proposition 6.7 *If* n *is a redex node of an ART* $\mathcal{G}$*, then it is either an application or a constant symbol node, that is,* $\mathcal{G}(n) = m\,o$ *or* $\mathcal{G}(n) = f$*.*

Proof. Induction on the definition of ARTs. Case $\mathcal{G} = \text{graph}(\varepsilon, M)$: Because of Lemma 5.4.ii $\mathcal{G}$ contains no node ending in r, hence no redex node. Case $\mathcal{G} = \mathcal{G}' \cup \text{graph}(m\mathsf{r}, R\sigma)$: Let n be a redex node of $\mathcal{G}$. Because $m\mathsf{r}$ is the only node ending in r in $\text{dom}(\text{graph}(m\mathsf{r}, R\sigma))$ according to Lemma 5.4.ii, $n \in \text{dom}(\mathcal{G}')$. If $n = m$, then according to Lemma 5.8.i n is an application or a constant symbol node. Otherwise n is a redex node of $\mathcal{G}'$ and according to the induction hypotheses it is an application or a constant symbol node. □

Proposition 6.8 *Let* $\mathcal{G}$ *and* $\mathcal{G}'$ *be ARTs with* $\mathcal{G} \to_P^* \mathcal{G}'$ *and* $n \in \text{dom}(\mathcal{G})$*. Then* $\lceil n \rceil_{\mathcal{G}'} = \lceil \lceil n \rceil_{\mathcal{G}} \rceil_{\mathcal{G}'}$*.*

Proof. By induction on the definition of $\lceil n \rceil_{\mathcal{G}}$.

- $n\mathsf{r} \in \text{dom}(\mathcal{G})$: So $n\mathsf{r} \in \text{dom}(G')$. Hence $\lceil n \rceil_{\mathcal{G}'} = \lceil n\mathsf{r} \rceil_{\mathcal{G}'}$ and $\lceil n \rceil_{\mathcal{G}} = \lceil n\mathsf{r} \rceil_{\mathcal{G}}$ and with the induction hypothesis $\lceil n\mathsf{r} \rceil_{\mathcal{G}'} = \lceil \lceil n\mathsf{r} \rceil_{\mathcal{G}} \rceil_{\mathcal{G}'}$ the property follows.
- $n\mathsf{r} \notin \text{dom}(\mathcal{G})$ and $\mathcal{G}(n) = m$: Then $\mathcal{G}'(n) = m$ and because of Proposition 6.7 $n\mathsf{r} \notin \text{dom}(\mathcal{G}')$. So $\lceil n \rceil_{\mathcal{G}'} = \lceil m \rceil_{\mathcal{G}'}$ and $\lceil n \rceil_{\mathcal{G}} = \lceil m \rceil_{\mathcal{G}}$ and with the induction hypothesis $\lceil m \rceil_{\mathcal{G}'} = \lceil \lceil m \rceil_{\mathcal{G}} \rceil_{\mathcal{G}'}$ the property follows.

- $n\mathsf{r} \notin \mathrm{dom}(\mathcal{G})$ and $\mathcal{G}(n) \neq m$: Then $\lceil n \rceil_{\mathcal{G}} = n$ and the property holds trivially.

□

Proposition 6.9 *Let $\mathcal{G}$ be an ART. If $n\mathsf{f} \in dom(\mathcal{G})$, then $\mathcal{G}(n) = n\mathsf{f}\, m$ for some node m. If $n\mathsf{a} \in dom(\mathcal{G})$, then $\mathcal{G}(n) = m\, n\mathsf{a}$ for some node m.*

Proof. Induction on the definition of ARTs, using Lemma 5.4.vii. □

7 No Cycles in an ART

For rewriting a constant symbol our definition deviates from real implementations and the real Hat ART. There a constant symbol is only reduced once during a whole computation and the reduct is shared by all use occurrences of the constant symbol. Modelling this behaviour would add substantial complexity; in particular, because constant symbols can be recursive, it would lead to cyclic ARTs and infinite (regular) terms. Hence we leave this extension and a comparison with the simpler model defined here for future work.

To express that an ART is acyclic we first have to define reachability of nodes:

Definition 7.1 The direct reachability relation for a term graph $\mathcal{G}$ is the least binary relation on nodes $\rightsquigarrow_{\mathcal{G}}$ with

$$\begin{aligned} \mathcal{G}(n) = m &\implies n \rightsquigarrow_{\mathcal{G}} m \\ \mathcal{G}(n) = m\, o &\implies n \rightsquigarrow_{\mathcal{G}} m \text{ and } n \rightsquigarrow_{\mathcal{G}} o \\ n, n\mathsf{r} \in \mathrm{dom}(G) &\implies n \rightsquigarrow_{\mathcal{G}} n\mathsf{r} \end{aligned}$$

Proposition 7.2 *An Art $\mathcal{G}$ is acyclic, that is, for all $n \in dom(\mathcal{G})$ we have $n \not\rightsquigarrow^{+}_{\mathcal{G}} n$.*

Proof. Induction on the definition of ARTs using Lemma 7.3. □

Lemma 7.3 *Let $\mathcal{G}$ be an ART with node m and M a label term that is not a node with $match_{\mathcal{G}}(m, M)$. For all nodes o in M we have $m \rightsquigarrow^{+}_{\mathcal{G}} o$.*

Proof. Induction on the definition of match. □

Because any ART has only a finite number of nodes we immediately get:

Corollary 7.4 *Reachability is terminating in an ART $\mathcal{G}$, that is, there is no infinite sequence $n_0 \rightsquigarrow_{\mathcal{G}} n_1 \rightsquigarrow_{\mathcal{G}} n_2 \rightsquigarrow_{\mathcal{G}} \ldots$.*

Because an ART $\mathcal{G}$ is terminating, the last node of a chain, $\lceil m \rceil_{\mathcal{G}}$, is defined for all nodes m, and so is $\mathrm{match}_{\mathcal{G}}(m, M)$.

8 Reconstructing Reduction Steps

A reduction step producing a new ART from an old ART uses a rewrite rule of the program, similar to any graph rewriting based abstract machine for implementing a functional language. This property does not yet indicate that the ART is an expressive representation of the whole computation history from initial term to final result (or abortion), suitable for fault localisation and program comprehension. It is a central property of the ART that every reduction step performed in its construction can easily be reconstructed from it, without having to rerun the computation, simply by traversing a small part of the graph.

Because of reduction edges, a single node of a term graph usually represents many computation terms. We obtain the most evaluated form of node n of graph $\mathcal{G}$, $\mathrm{mef}_{\mathcal{G}}(n)$, by always following reduction and indirection edges:

Definition 8.1 Let $\mathcal{G}$ be an ART with redex node n.

$$\mathrm{mef}_{\mathcal{G}}(n) = \mathrm{mefT}_{\mathcal{G}}(\mathcal{G}(\lceil n\rceil_{\mathcal{G}}))$$

$$\begin{aligned}\mathrm{mefT}_{\mathcal{G}}(a) &= a\\ \mathrm{mefT}_{\mathcal{G}}(m\,n) &= \mathrm{mef}_{\mathcal{G}}(m)\,\mathrm{mef}_{\mathcal{G}}(n)\end{aligned}$$

In the following we apply $\mathrm{mef}_{\mathcal{G}}$ not just to a node but use its homomorphic extension for a label term. We write such an application in postfix notation, for example $M\,\mathrm{mef}_{\mathcal{G}}$, like the application of a substitution.

We can also reconstruct other computation terms from a node. From a redex node we can reconstruct a redex:

Definition 8.2 Let $\mathcal{G}$ be an ART with redex node n.

$$\mathrm{redex}_{\mathcal{G}}(n) = \begin{cases}\mathrm{mef}_{\mathcal{G}}(m)\,\mathrm{mef}_{\mathcal{G}}(o) & \text{, if } \mathcal{G}(n) = m\,o\\ a & \text{, if } \mathcal{G}(n) = a\end{cases}$$

Redex, matching and most evaluated form are closely related. Note though that the definitions of $\mathrm{redex}_{\mathcal{G}}$ and $\mathrm{mef}_{\mathcal{G}}$ follow the graph structure whereas $\mathrm{match}_{\mathcal{G}}$ is defined on the structure of the matched label term.

Proposition 8.3 *Let $\mathcal{G}$ be an ART with node m and M a label term that is not just a node.*

$$\mathit{match}_{\mathcal{G}}(m, M) \implies \mathit{redex}_{\mathcal{G}}(m) = M\,\mathit{mef}_{\mathcal{G}}$$

Proof. Case analysis.

- $\mathcal{G}(m) = a$: $\mathrm{match}_{\mathcal{G}}(m, M) \Rightarrow M = \mathcal{G}(m) = a \Rightarrow \mathrm{redex}_{\mathcal{G}}(m) = a = M\,\mathrm{mef}_{\mathcal{G}}$.
- $\mathcal{G}(m) = n\,o$: If M is an atom or node, then $\mathrm{match}_{\mathcal{G}}(m, M) = \text{false}$. So let $M = N\,O$. $\mathrm{match}_{\mathcal{G}}(m, N\,O) \implies \mathrm{match}_{\mathcal{G}}(\text{if } N \text{ is a node then } n \text{ else } \lceil n\rceil_{\mathcal{G}}, N) \wedge \mathrm{match}_{\mathcal{G}}(\text{if } O \text{ is a node then } o \text{ else } \lceil o\rceil_{\mathcal{G}}, O)$. With Lemma 8.4 follows $\mathrm{mef}_{\mathcal{G}}(n) = N\,\mathrm{mef}_{\mathcal{G}} \wedge \mathrm{mef}_{\mathcal{G}}(o) = O\,\mathrm{mef}_{\mathcal{G}}$. Hence $\mathrm{mef}_{\mathcal{G}}(m) = \mathrm{mef}_{\mathcal{G}}(n)\,\mathrm{mef}_{\mathcal{G}}(o) = (N\,O)\,\mathrm{mef}_{\mathcal{G}}$.

□

Lemma 8.4 *Let $\mathcal{G}$ be an ART with node m and M a label term.*

$$\mathit{match}_{\mathcal{G}}(\mathit{if}\ M\ \mathit{is\ a\ node\ then}\ m\ \mathit{else}\ \lceil m\rceil_{\mathcal{G}}, M) \implies \mathit{mef}_{\mathcal{G}}(m) = M\,\mathit{mef}_{\mathcal{G}}$$

Proof. Induction on M.

- $M = a$: $\mathrm{match}_{\mathcal{G}}(\lceil m\rceil_{\mathcal{G}}, a) \Rightarrow \mathcal{G}(\lceil m\rceil_{\mathcal{G}}) = a \Rightarrow \mathrm{mef}_{\mathcal{G}}(m) = a = a\,\mathrm{mef}_{\mathcal{G}}$.
- $M = n$: $\mathrm{match}_{\mathcal{G}}(m, n) \Rightarrow (m = n) \Rightarrow \mathrm{mef}_{\mathcal{G}}(m) = \mathrm{mef}_{\mathcal{G}}(n) = n\,\mathrm{mef}_{\mathcal{G}}$.

- $M = N\,O$: $\mathrm{match}_{\mathcal{G}}(\lceil m \rceil_{\mathcal{G}}, N\,O) \Rightarrow \exists n, o.(\mathcal{G}(\lceil m \rceil_{\mathcal{G}}) = n\,o) \wedge \mathrm{match}_{\mathcal{G}}(\text{if } N \text{ is a node then } n \text{ else } \lceil n \rceil_{\mathcal{G}}, N) \wedge \mathrm{match}_{\mathcal{G}}(\text{if } O \text{ is a node then } o \text{ else } \lceil o \rceil_{\mathcal{G}}, O)$. With the induction hypothesis follows $\exists n, o.(\mathcal{G}(\lceil m \rceil_{\mathcal{G}}) = n\,o) \wedge (\mathrm{mef}_{\mathcal{G}}(n) = N\,\mathrm{mef}_{\mathcal{G}}) \wedge (\mathrm{mef}_{\mathcal{G}}(o) = O\,\mathrm{mef}_{\mathcal{G}})$. So $\mathrm{mef}_{\mathcal{G}}(n) = (N\,O)\,\mathrm{mef}_{\mathcal{G}}$.

□

The opposite direction of Proposition 8.3 does not hold for reasons already discussed in Section 5. In the example given there $\mathrm{mef}_{\mathcal{G}}(\varepsilon) = (\mathsf{id}\,\mathsf{ar})\,\mathrm{mef}_{\mathcal{G}}$, but $\mathrm{match}_{\mathcal{G}}(\varepsilon, \mathsf{id}\,\mathsf{ar}) = \text{false}$.

The real Hat ART also includes so-called *parent edges* that we have not yet mentioned. Each node has a parent edge that points to the top node of the redex that caused its creation. Parent edges are the key ingredient for the redex trail view of locating program faults [26,25]. In this paper we will only use them to reconstruct a reduct from an ART. In our ART model parents are encoded in the nodes:

Definition 8.5

$$\begin{aligned}
\mathrm{parent}(n\mathsf{r}) &= n \\
\mathrm{parent}(n\mathsf{f}) &= \mathrm{parent}(n) \\
\mathrm{parent}(n\mathsf{a}) &= \mathrm{parent}(n) \\
\mathrm{parent}(\varepsilon) &= \text{undefined}
\end{aligned}$$

For example, the parent of node t is node ε. The parent of node rt is node r.

It is easy to recognise in a term graph all nodes that form a right-hand side of a rule: they all have the same parent. So using parent edges we can easily reconstruct a reduct from a reduct node:

Definition 8.6 Let $\mathcal{G}$ be an ART with redex node n.

$$\begin{aligned}
\mathrm{reduct}_{\mathcal{G}}(n) &= \mathrm{reductP}_{\mathcal{G}}(n, n\mathsf{r}) \\
\mathrm{reductP}_{\mathcal{G}}(p, n) &= \text{if parent}(n) = p \text{ then reductT}_{\mathcal{G}}(p, \mathcal{G}(n)) \text{ else mef}_{\mathcal{G}}(n) \\
\mathrm{reductT}_{\mathcal{G}}(p, a) &= a \\
\mathrm{reductT}_{\mathcal{G}}(p, m) &= \mathrm{mef}_{\mathcal{G}}(m) \\
\mathrm{reductT}_{\mathcal{G}}(p, n\,o) &= \mathrm{reductP}_{\mathcal{G}}(p, n)\,\mathrm{reductP}_{\mathcal{G}}(p, o)
\end{aligned}$$

Lemma 8.7 *Let $\mathcal{G}$ be an ART with node m and M a label term with nodes none of whose parents is m. Then $graph(m\mathsf{r}, M) \subseteq \mathcal{G} \implies reduct_{\mathcal{G}}(m) = M\,mef_{\mathcal{G}}$.*

Proof. Follows from the following more general lemma. □

Lemma 8.8 *Let $\mathcal{G}$ be an ART with nodes m and p and M a label term with nodes none of whose parents is p.*

$$parent(m) = p \wedge graph(m, M) \subseteq \mathcal{G} \implies reductP_{\mathcal{G}}(p, m) = M\ mef_{\mathcal{G}}$$

Proof. Induction on M.

- $M = a$: $\text{graph}(m, a) \subseteq \mathcal{G} \Rightarrow \mathcal{G}(m) = a \Rightarrow \text{reductP}_{\mathcal{G}}(p, m) = \text{reductT}_{\mathcal{G}}(p, a) = a = a\,\text{mef}_{\mathcal{G}}$.
- $M = n$: $\text{graph}(m, n) \subseteq \mathcal{G} \Rightarrow \mathcal{G}(m) = n \Rightarrow \text{reductP}_{\mathcal{G}}(p, m) = \text{reductT}_{\mathcal{G}}(p, n) = \text{mef}_{\mathcal{G}}(n) = n\,\text{mef}_{\mathcal{G}}(n)$.
- $M = N\,O$: W.l.o.g. let $N = n$ be a node and O not a node. $\text{graph}(m, n\,O) \subseteq \mathcal{G} \Rightarrow \mathcal{G}(m) = N\,m\mathsf{a} \wedge \text{graph}(m\mathsf{a}, O) \subseteq \mathcal{G}$. With the induction hypothesis follows $\text{reductP}_{\mathcal{G}}(p, m\mathsf{a}) = O\,\text{mef}_{\mathcal{G}}$. Thus $\text{reductP}_{\mathcal{G}}(p, m) = \text{reductT}_{\mathcal{G}}(p, n\,m\mathsf{a}) = \text{reductP}_{\mathcal{G}}(p, n)\,\text{reductP}_{\mathcal{G}}(p, m\mathsf{a}) = \text{mef}_{\mathcal{G}}(n)\,(O\,\text{mef}_{\mathcal{G}}) = (N\,O)\,\text{mef}_{\mathcal{G}}$.

□

The preceding definitions of mef, redex and reduct are all total on their specified domain, because reachability in ARTs is terminating.

Proposition 8.9 *Let $\mathcal{G}' \to_{P,n} \mathcal{G}''$ with rewrite rule $L = R \in P$ and node substitution σ. For all $\mathcal{G}$ with $\mathcal{G}'' \to_P^* \mathcal{G}$*

$$\textit{redex}_{\mathcal{G}}(n) = L\sigma\,\textit{mef}_{\mathcal{G}} \quad \textit{and} \quad \textit{reduct}_{\mathcal{G}}(n) = R\sigma\,\textit{mef}_{\mathcal{G}}.$$

Proof. This follows from Lemmas 9.2, 8.3 and 8.7 and the fact that $\text{graph}(n\mathsf{r}, R\sigma) \subseteq \mathcal{G}$. □

9 Evaluation Order

Our rewriting is non-deterministic. We have not defined any reduction strategy. All properties given in this paper hold without reference to any particular reduction strategy. This generality proves that the ART can be used for different reduction strategies, in particular for optimising implementations of non-strict functional languages that mix eager and lazy evaluation [11,18,10]. For a language with strict semantics we would need to disallow some reduction steps, but otherwise all definitions and properties carry over as well. For using an ART to locate a fault it is only important that every reduction step is *correct* with respect to the language semantics. [1]

Our programs are almost non-overlapping. For generality we do allow for non-deterministic functions; we do permit the left-hand sides of two rewrite rules for the same function symbol to have common instance. There may be different computations for the same program and initial term. Hence normal rewriting and thus also ART rewriting cannot be confluent. However, rewriting on ARTs is almost subcommutative, which means that whenever an ART has several unreduced redexes, these can be reduced in any order always yielding the same ART.

We first need a lemma that says that no rewriting can take place in the part of the term graph representing a proper subterm of a left-hand side of a rewrite rule.

Lemma 9.1 *Let $\mathcal{G}$ be a term graph with node n, L the left-hand side of a rewrite rule of program P and σ a substitution replacing all variables of L by nodes. Let V*

[1] For analysing the ART of a non-terminating computation we will have to make some assumptions about the *completeness* of the reduction strategy.

be a proper subterm of $L\sigma$ that is not just a node. Then

$$\mathit{match}_{\mathcal{G}}(n, V) \wedge \mathcal{G} \rightarrow_P \mathcal{G}' \implies \lceil n \rceil_{\mathcal{G}'} = n$$

Proof. Suppose $n\mathsf{r} \in \mathcal{G}'$. Because $\mathrm{match}_{\mathcal{G}}(n, V)$, $n\mathsf{r} \notin \mathcal{G}$. So there exists a left-hand side L' of a rewrite rule and a substitution σ' with $\mathrm{match}_{\mathcal{G}}(n, L'\sigma')$. With Proposition 8.3 follows both $\mathrm{redex}_{\mathcal{G}}(n) = L'\sigma'\mathrm{mef}_{\mathcal{G}}$ and $\mathrm{redex}_{\mathcal{G}}(n) = V\mathrm{mef}_{\mathcal{G}}$. However, $L'\sigma'\mathrm{mef}_{\mathcal{G}} = V\mathrm{mef}_{\mathcal{G}}$ contradicts the definition of rewrite rules and in particular the definition of patterns.

Suppose $\mathcal{G}'(n) = m$ for a node m. Then $\mathcal{G}(n) = m$. However, according to the definition of matching this contradicts with $\mathrm{match}_{\mathcal{G}}(n, V)$.

Altogether $n\mathsf{r} \notin \mathcal{G}'$ and $\mathcal{G}'(n)$ is not a node. So $\lceil n \rceil_{\mathcal{G}'} = n$. □

When a left-hand side of a rewrite rule or a proper subterm of it matches a term graph, then it will do so in the future:

Proposition 9.2 *Let $\mathcal{G}$ be a term graph, L the left-hand side of a rewrite rule of program P and σ a substitution replacing all variables of L by nodes. Let N be a subterm of $L\sigma$ (including $N = L\sigma$).*

$$\mathit{match}_{\mathcal{G}}(n, N) \wedge \mathcal{G} \rightarrow_P \mathcal{G}' \implies \mathit{match}_{\mathcal{G}'}(n, N)$$

Proof. Induction on N.

- $N = m$ for some node m.
  $\mathrm{match}_{\mathcal{G}}(n, m) \Rightarrow n = m \Rightarrow \mathrm{match}_{\mathcal{G}'}(n, m)$.
- $N = a$ for some atom a.
  Then $\mathcal{G}(n) = a$. Because of $\mathcal{G} \subseteq \mathcal{G}'$, $\mathcal{G}'(n) = a$. So $\mathrm{match}_{\mathcal{G}'}(n, a)$.
- $N = M\,O$ for some proper subterms M and O of a left-hand side instance.
  Then there exist m and o with $\mathcal{G}(n) = m\,o \wedge \mathrm{match}_{\mathcal{G}}(\text{if } M \text{ is a node then } m \text{ else } \lceil m \rceil_{\mathcal{G}}, M) \wedge \mathrm{match}_{\mathcal{G}}(\text{if } O \text{ is a node then } o \text{ else } \lceil o \rceil_{\mathcal{G}}, O)$. With the induction hypothesis follows $\mathrm{match}_{\mathcal{G}'}(\text{if } M \text{ is a node then } m \text{ else } \lceil m \rceil_{\mathcal{G}}, M) \wedge \mathrm{match}_{\mathcal{G}'}(\text{if } O \text{ is a node then } o \text{ else } \lceil o \rceil_{\mathcal{G}}, O)$. According to Lemmas 9.1 and 6.8, if M is a node, then $\lceil m \rceil_{\mathcal{G}'} = \lceil \lceil m \rceil_{\mathcal{G}} \rceil_{\mathcal{G}'} = \lceil m \rceil_{\mathcal{G}}$. Similarly, if O is a node, then $\lceil o \rceil_{\mathcal{G}'} = \lceil \lceil o \rceil_{\mathcal{G}} \rceil_{\mathcal{G}'} = \lceil o \rceil_{\mathcal{G}}$. So $\mathrm{match}_{\mathcal{G}'}(\text{if } M \text{ is a node then } m \text{ else } \lceil m \rceil_{\mathcal{G}'}, M) \wedge \mathrm{match}_{\mathcal{G}'}(\text{if } O \text{ is a node then } o \text{ else } \lceil o \rceil_{\mathcal{G}'}, O)$. Because of $\mathcal{G} \subseteq \mathcal{G}'$, $\mathcal{G}'(n) = m\,o$. Altogether $\mathrm{match}_{\mathcal{G}'}(n, M\,O)$.

□

Corollary 9.3 *Rewriting on ARTs is almost subcommutative, that is, if $\mathcal{G} \rightarrow_{P,n_1} \mathcal{G}_1$ and $\mathcal{G} \rightarrow_{P,n_2} \mathcal{G}_2$ with $n_1 \neq n_2$, then there exists an ART $\mathcal{G}'$ with $\mathcal{G}_1 \rightarrow_{P,n_2} \mathcal{G}'$ and $\mathcal{G}_2 \rightarrow_{P,n_1} \mathcal{G}'$.*

Proof. $\mathcal{G} \rightarrow_{P,n_1} \mathcal{G}_1$ and $\mathcal{G} \rightarrow_{P,n_2} \mathcal{G}_2$ implies by definition of rewriting that there exist rewrite rules $L_1 = R_1$ and $L_2 = R_2$ and substitutions σ_1 and σ_2 with $\mathrm{match}_{\mathcal{G}}(n_1, L_1\sigma_1)$ and $\mathrm{match}_{\mathcal{G}}(n_2, L_2\sigma_2)$ and $n_1\mathsf{r}, n_2\mathsf{r} \notin \mathrm{dom}(\mathcal{G})$. With Proposition 9.2 follows $\mathrm{match}_{\mathcal{G}_2}(n_1, L_1\sigma_1)$ and $\mathrm{match}_{\mathcal{G}_1}(n_2, L_2\sigma_2)$. With $n_1 \neq n_2$ follows $n_1\mathsf{r} \notin \mathrm{dom}(\mathcal{G}_2)$ and $n_1\mathsf{r} \notin \mathrm{dom}(\mathcal{G}_1)$. Hence $\mathcal{G}_2 \rightarrow_{P,n_1} \mathcal{G}'$ and $\mathcal{G}_1 \rightarrow_{P,n_2} \mathcal{G}'$ where $\mathcal{G}' = \mathcal{G} \cup \mathrm{graph}(n_1, L_1\sigma_1) \cup \mathrm{graph}(n_2, L_2\sigma_2)$. □

If a program is actually non-overlapping, not just almost non-overlapping, then rewriting on ARTs is subcommutative and thus also confluent.

10 Runtime Errors

The main aim of tracing is to support locating the cause of an observed faulty behaviour. For functional programs there are three types of faulty behaviour: an incorrect result, a runtime error or non-termination. As currently defined an ART is useful for locating the cause of an incorrect result and it may also be used to consider an interrupted non-terminating computation. However, we have not yet modelled any runtime error. A runtime error may be raised by a built-in function, but for our programs the most natural source of a runtime error is pattern match failure. The left-hand sides of rewrite rules for a function symbol may not be complete, for example for a function determining the head element of a (non-empty) list.

We extend our data constructors by a special one called `error` that does not appear in the pattern of any rewrite rule. We extend our term graph reduction relation:

Definition 10.1 We extend Definition 5.6 of the *reduction relation* $\to_P$ on term graphs for program P as follows: If

- f is a function symbol of arity n and $P_1, \ldots, P_n$ patterns,
- for all substitutions of computation terms for variables $\hat{\sigma}$ and all rewrite rules $L = R \in P$ we have $L\hat{\sigma} \neq (f\, P_1 \ldots P_n)\hat{\sigma}$,
- $\mathcal{G}$ is a term graph with $n \in \mathrm{dom}(\mathcal{G})$ and $n\mathsf{r} \notin \mathrm{dom}(\mathcal{G})$,
- σ is a substitution replacing variables by nodes,
- $\mathrm{match}_{\mathcal{G}}(n, (f\, P_1 \ldots P_n)\sigma)$,

then $\mathcal{G} \to_P \mathcal{G} \cup \{(n\mathsf{r}, \texttt{error})\}$.

So if in a term graph all arguments of a full application are fully evaluated, then this full application can be reduced.

Because `error` does not appear in the pattern of any rewrite rule, a reduction step yielding `error` will usually start a chain of pattern match failure reduction steps that promote the runtime error to the outermost reduction.

11 Related Work

The first papers on semantical foundations for tracing [3,2,9] describe methods where stepping through a computation and viewing information are firmly linked. In contrast, we study here a tracing method that relies on the decoupling of computation and viewing through the generation of a trace as self-contained data structure. Existing indirect definitions of traces through descriptions of their generation [12,22,8] and even the simplified definition in [19] are too complex to prove theorems and make it hard to distinguish incidental implementation features from essential links to the semantics of the traced program.

The *redex trail* trace was introduced in [26,25] for a particular viewing method.

The programmer explores a computation backwards, from an effect — such as output or a runtime error — to its cause. Hence the most important component of a redex trail are the parent edges, which point from each part of a reduct to its redex. The redex trail has no reduction edges. These were added in [27] to obtain the *augmented redex trail* (ART) which thus supports several different views of a computation.

The first author sketched a semantics for ARTs in [6]. This semantics is only for a small core language into which the full language has to be translated and it is still too complex for proving interesting properties. A formal definition of an ART-like trace for a functional-logic language has been given by extending an abstract machine to generate the trace [4]. This paper also uses a small core language and proofs are hard. Although the trace is very similar to the ART, it differs in two respects (besides small extensions for logical variables and non-determinism): First, parent edges have a different meaning; the parent edge of an expression points to the dynamic redex that forced its evaluation, not the static redex that produced it. Second, the trace only contains expressions that were needed for the computation, that is, that were matched against in a reduction or that are part of the final result. In contrast we defined the ART here so that a reduction step always adds the full right-hand side of the rewrite rule to the trace, even if parts of that will never be used. In [24] we consider a variant of our definition that allows for non-deterministically adding only a part of a right-hand side. We intend to explore this variant further.

Algorithmic debugging is a semi-automatic method for locating a faulty rewrite rule in a program [23,19,22]. [5] prove that algorithmic debugging is correct for functional logic programs, that is, if a computation yields an incorrect result, then algorithmic debugging correctly locates the faulty rewrite rule. The paper does not use a trace data structure but describes algorithmic debugging abstractly through a non-deterministic big-step semantics. However, algorithmic debugging is based on a tree-based representation of the computation, the evaluation dependency tree (EDT). We showed in [16] how the EDT can easily be constructed from the ART and using the ART we proved the correctness of algorithmic debugging. Recently we extended this correctness result to ARTs from which some parts that represent unevaluated expressions have been removed [17]. With our definition of runtime errors the correctness result also applies to computations that abort with a runtime error whereas [5] only handles computations that produce incorrect results, because unevaluated expressions (replaced by $\perp$) and runtime errors are not distinguished.

12 Conclusions and Further Work

We have motivated and presented a theory of tracing for functional programs. We have given a formal definition of the augmented redex trail (ART). The ART is a complex structure that encodes a wealth of information about a computation such that it can easily be retrieved but the ART is still compact. Nonetheless our definition is relatively simple and can be used for formal proof as numerous examples demonstrate. Although the ART is a graph that takes advantage of sharing, we have given an inductive definition, thus making the powerful inductive proof method available for proofs about ARTs. Despite this inductive definition, the ART is independent of any particular evaluation order, demonstrating that we really abstract from such low-level details. It was non-trivial to prove that the ART is acyclic. Most importantly we have shown how each reduction step of a computation can easily be reconstructed from the ART. These reconstructed reductions are used in nearly all views of computations based on the ART.

Future work goes in two orthogonal directions. First, we have to determine and prove essential properties of various views based on ARTs. We have already proved the correctness of the algorithmic debugging method based on ARTs and intend to prove similar results for other views, in particular the redex trailing method for locating faults.

Second, we intend to extend the programming language and make our model of the real ART even more realistic. Currently we are studying two extensions. We add local rewrite rules to the programming language. Information about the value of a free variable, such as variable `x` of function `f` in the definition

```
g x y z = f y + f z
  where
  f v = v + x
```

needs to be available in the trace. The current ART is acyclic, but implementations of lazy functional languages actually create cyclic graphs for constants such as

```
ones = 1 : ones
```

The real ART produced by Hat also contains cycles in such cases and it is well known that these cause problems for example for algorithmic debugging. We intend to compare definitions of ARTs with and without cycles to determine which information is lost.

Acknowledgements

The work reported in this paper was supported by the Engineering and Physical Sciences Research Council of the United Kingdom under grant EP/C516605/1.

References

[1] Adam Bakewell. Using term-graph rewriting models to analyse relative space efficiency. In *TERMGRAPH 2002 International Workshop on Term Graph Rewriting*, volume 72 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[2] K. Bernstein and E. W. Stark. On formally defining debuggers: A comparison of three approaches. In *2nd International Workshop on Automated and Algorithmic Debugging*, St. Malo, France, May 1995.

[3] Karen L. Bernstein and Eugene W. Stark. Operational semantics of a focusing debugger. In *Proceedings of the Eleventh Conference on the Mathematical Foundations of Programming Semantics*, ENTCS 1. Elsevier, 1995.

[4] Bernd Braßel, Michael Hanus, Frank Huch, and German Vidal. A semantics for tracing declarative multi-paradigm programs. In *Proceedings of the 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 179–190. ACM Press, 2004.

[5] Rafael Caballero, Francisco J. López-Fraguas, and Mario Rodríguez-Artalejo. Theoretical foundations for the declarative debugging of lazy functional logic programs. In *Functional and Logic Programming, FLOPS 2001*, LNCS 2024. Springer, 2001.

[6] Olaf Chitil. A semantics for tracing. In Thomas Arts and Markus Mohnen, editors, *Draft Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL 2001*, pages 249–254, Älvsjö, Sweden, September 2001. Ericsson Computer Science Laboratory.

[7] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Freja, Hat and Hood — A comparative evaluation of three systems for tracing and debugging lazy functional programs. In Markus Mohnen and Pieter Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000*, LNCS 2011, pages 176–193. Springer, 2001.

[8] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for tracing. In *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL 2002)*, LNCS 2670, pages 165–181, 2003.

[9] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In David Sands, editor, *10th European Symposium on Programming, ESOP 2001*, LNCS 2028, pages 320–334. Springer, 2001.

[10] Robert Ennals and Simon Peyton Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 287–298, New York, NY, USA, 2003. ACM Press.

[11] Karl-Filip Faxén. Cheap eagerness: speculative evaluation in a lazy functional language. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN International Conference on Functional programming*, pages 150–161, New York, NY, USA, 2000. ACM Press.

[12] Andy Gill. Debugging Haskell by observing intermediate data structures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001. 2000 ACM SIGPLAN Haskell Workshop.

[13] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69. ACM Press, 1984.

[14] J.W. Klop. Term rewriting systems. In S. Abramsky, M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[15] 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.

[16] Yong Luo and Olaf Chitil. Proving the correctness of algorithmic debugging for functional programs. In *Draft Proceedings of the Seventh Symposium on Trends in Functional Programming, TFP 2006*, Nottingham, UK, April 2006.

[17] Yong Luo and Olaf Chitil. Replacing unevaluated parts in the traces of functional programs. In *Draft Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL 2006*, pages 304–325, Budapest, Hungary, 2006. Eötvös Loránd University. Technical Report No: 2006-S01.

[18] Jan-Willem Maessen. Eager Haskell: resource-bounded execution yields efficient iteration. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 38–50, New York, NY, USA, 2002. ACM Press.

[19] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, May 1998.

[20] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[21] Detlef Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, chapter 1, pages 3–61. World Scientific, 1999. Volume 2: Applications, Languages and Tools.

[22] B. Pope and Lee Naish. Practical aspects of declarative debugging in Haskell-98. In *Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 230–240, 2003.

[23] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[24] Josep Silva and Olaf Chitil. Combining algorithmic debugging and program slicing. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 157–166, New York, NY, USA, 2006. ACM Press.

[25] 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.

[26] 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.

[27] 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. Final proceedings to appear in ENTCS 59(2).