Tracing Lazy Functional Computations Using Redex Trails

Jan Sparud and Colin Runciman

Department of Computer Science, University of York,
Heslington, York, YO1 5DD, UK
(e-mail: {`sparud,colin`}`@cs.york.ac.uk`)

Abstract. We describe the design and implementation of a system for tracing computations in a lazy functional language. The basis of our tracing method is a program transformation carried out by the compiler: transformed programs compute the same values as the original, but embedded in functional data structures that also include *redex trails* showing how the values were obtained. A special-purpose display program enables detailed but selective exploration of the redex trails, with cross-links to the source program.

Keywords: debugging, graph reduction, Haskell, program transformation.

1 Introduction

1.1 Why trace functional computations?

Functional programming languages have many advantages over the conventional alternative of procedural languages. For example, program construction is more rapid, more modular and less error-prone. Programs themselves are more concise.

Yet functional programming systems are not very widely used. There are various reasons for this, but one that crops up time and again is the *lack of tracing facilities.* Yes, there is less scope for making mistakes in a functional language; but programmers do still make them! And when their programs go wrong they need to trace the cause. Unfortunately, implementors of functional languages are hard-pressed to provide equivalents of the 'debugging tools' routinely used to investigate faults in procedural programs. Tracing evaluation by normal order graph reduction is more subtle than following a sequence of commands already explicit at source level.

There have been various attempts to tackle the problem of tracing lazy functional programs. We discuss some of them in §6. But so far as we know there is as yet no really effective solution — a state of affairs we'd like to change.

1.2 How? Some design goals and assumptions

Functional language We concentrate on the tracing problem for purely functional languages such as Haskell. Despite the absence of side-effects, lazy evaluation and higher-order functions in languages like Haskell make the problem difficult: there is a big gap between high-level declarative programs and the low-level sequences of events in their computations.

Graph reduction We assume an implementation based on graph-reduction. In essence, the objective of computation is to evaluate an expression represented by a graph. This is achieved by repeatedly replacing one subgraph by another, where the reduction rules used to define replacements are derived from the equations given in the program. At each reduction step, a *redex* matching the left hand side of an equation is replaced by a the corresponding instance of the right hand side. Computation by graph reduction is made efficient by compilation to code for a G-machine, or similar.

Backward traces We need to provide backward traces from results or from run-time errors, because the most pressing need for traces arises in the context of an unexpected output or failure.

Redex trails We use the idea of a *redex trail* to provide the overall framework for answering the question 'How has this value/failure come about?'. At each reduction, parts of the redex no longer attached to the main graph are normally discarded. If we instead make a link from each newly created node of the graph to its *parent redex*, the computation builds its own trail as reduction proceeds.

Non-invasive traces The transformation to introduce redex trails should not change the course of the underlying computation in any way. For example, unevaluated expressions should remain unevaluated.

Complete traces Until we have a very strong reason to discard parts of the information in redex trails, and a clear argument which parts should go, we want to construct traces in full. There must be a representation of every reduction step for definitions and expressions of every kind.

Selective display A full trace of even a modest computation contains a great deal of information — too much for the programmer to absorb in its entirety. Programmers need fine control over what trace information is actually displayed to them, down to the level of interactive link-by-link examination of the trails leading from a run-time fault or selected fragment of output.

Traces linked to source However good the tracing system, source programs are likely to remain the primary reference for programmers. Not only should expressions in traces be displayed just as they might be written in a source program; trace text should be also be linked directly to source text.

A portable implementation Although a prototype tracer must have some specific host implementation (ours will be the Haskell compiler `nhc` [Röj95]), we aim to produce a portable tracing scheme that could be adopted in other implementations of a functional language. For this reason, we shall prefer to use source-to-source transformation rather than to modify the run-time system.

An efficient implementation Finally, the implementation must be efficient — or at least, not so inefficient that using the tracer is infeasible or unattractive. If execution slows by no more than a factor of ten, we can hope that the value of tracing information will make the speed tolerable; but factors of a hundred or a thousand are unlikely to be acceptable. Also, memory requirements must be such that the tracer can be used on an ordinary workstation.

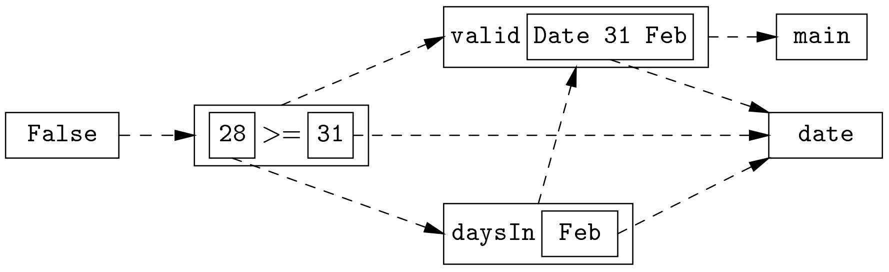
1.3 Details that follow

§2 explains what we mean by a redex trail, and shows how trails can be represented as functional data structures. §3 gives rules for transforming a program so that all values it computes are wrapped in a construction including a redex trail; it also explains briefly how we implement these rules in the nhc compiler. §4 explains the design of the interactive display program we use to examine redex trails. §5 evaluates aspects of the trace system we have built, including some performance figures. §6 discusses related work on other tracing systems for functional programs. §7 concludes and also gives some of our plans for future work.

2 The design of a trace structure

To appreciate what we mean by a redex trail consider the following example of a simple program testing the validity of a date.

```
daysIn Jan = 31
daysIn Feb = 28
     ...
daysIn Dec = 31
valid (Date d m) = daysIn m >= d
date = Date 31 Feb
main = valid date
```



The diagram shows the complete redex trail from the value `False` of the main expression. The immediate parent redex (whose reduction caused the creation of this instance of `False`) was `28 >= 31`. The parent redex of this application of `>=` was `valid (Date 31 Feb)`, but the parent redex of the left operand `28` was `daysIn Feb` and that of the right operand `31` was `date`. And so on. There is a link from every part of every expression to its parent redex.

The idea is to construct such a trace giving comprehensive information about a functional computation. The trace should be built as the computation proceeds. Once the computation is over the trace is just a static data structure.

Such redex trails can be represented as functional data structures of type:

```
data Trace = Ap Trace [Trace] | Nm Trace Name | Root
```

An application is represented by `Ap` t_{ap} `[`t_f`,` t_{x_1}`,` `...`, t_{x_n}`]`, where t_{ap} is the trace for the application itself, t_f is the trace of the function part of the application, and $t_{x_1} \ldots t_{x_n}$ are the traces of the arguments. A name (a function, variable, constructor or literal) is represented by `Nm` t_{nm} *name* where t_{nm} is the trace of the name and *name* is a textual representation of the name. The application and name nodes should also have *source code references*, but since we are only interested in the structure of the trace at this stage, we have omitted them here. `Root` is the null trace attached to top-level names.

3 Creating traces

In this section we define transformations to derive a self-tracing version of any given program. We will introduce a number of combinators that actually perform the trace creation, but first we will describe how the *types* of a program will change under transformation.

3.1 Type transformation rules

Every value in the program will be wrapped in the `R` datatype, defined as:

```
data R a = R a Trace
```

The intuition is that every value in the original program should have a wrapped value in the transformed program containing the original value and a trace for that value.

Figure 1 gives the rules determining how types are transformed. Motivation and examples follow.

Traces for structured data values For a structured data value, we want a trace not only for the outermost construction but also for all components. The $\mathcal{D}$ scheme is responsible for transforming datatypes into this form. We don't give the definition of the $\mathcal{D}$ scheme, but here is an example of the transformation of

$$\mathcal{R}[\![t]\!] \Rightarrow \texttt{R}\ (\mathcal{T}[\![t]\!])$$

$$\mathcal{T}[\![\mathit{tcon}\ t_1\ \ldots t_n]\!] \Rightarrow \mathit{tcon}\ \mathcal{T}[\![t_1]\!]\ \ldots\ \mathcal{T}[\![t_n]\!]\ (n \geq 0,\ \mathit{tcon} \neq \text{'}\!\rightarrow\!\text{'})$$
$$\mathcal{T}[\![t_1 \rightarrow t_2]\!] \Rightarrow \texttt{Trace} \rightarrow \mathcal{R}[\![t_1]\!] \rightarrow \mathcal{R}[\![t_2]\!]$$
$$\mathcal{T}[\![\alpha]\!] \Rightarrow \alpha$$

Fig. 1. Type rules for transformed expressions.

a simple tree datatype:

$\mathcal{D}[\![\texttt{data Tree a = Node (Tree a) Int (Tree a) | Leaf a}]\!]$
$\Rightarrow$
$\texttt{data Tree a = Node } \mathcal{R}[\![\texttt{Tree a}]\!]\ \mathcal{R}[\![\texttt{Int}]\!]\ \mathcal{R}[\![\texttt{Tree a}]\!] \texttt{ | Leaf } \mathcal{R}[\![\texttt{a}]\!]$
$\Rightarrow$
`data Tree a = Node (R (Tree a)) (R Int) (R (Tree a)) | Leaf (R a)`

Traces for values of function type Perhaps a value of function type $\alpha \rightarrow \beta$ can be treated as a structured value with constructor $\rightarrow$ and components α and β? Then the type $\alpha \ \rightarrow \ \beta$ would be transformed into $\texttt{R}\ (\mathcal{R}[\![\alpha]\!]\rightarrow\mathcal{R}[\![\beta]\!])$. Is this adequate? No! A function of type $\alpha \ \rightarrow \ \beta$ has a guarantee to fulfill: given an argument of type α it must return a value of type β. But transformed functions have a further obligation; they must also return a *trace* for that value of type β. A function of type $\mathcal{R}[\![\alpha]\!]\rightarrow\mathcal{R}[\![\beta]\!]$ *cannot* fulfill that obligation, since when the function is applied it does not know the current trace (or evaluation context) so it cannot build a full trace for the return value. Transformed functions need the trace of the application site as an extra argument. As an example, here is how the type of the standard *map* function is transformed:

$\mathcal{R}[\![\ (\texttt{a}\rightarrow \texttt{b}) \rightarrow [\texttt{a}] \rightarrow [\texttt{b}]\ ]\!]$
$\Rightarrow$
$\texttt{R}\ (\texttt{Trace} \rightarrow \mathcal{R}[\![\texttt{a}\rightarrow \texttt{b}]\!]\rightarrow \mathcal{R}[\![\ [\texttt{a}] \rightarrow [\texttt{b}]\ ]\!])$
$\Rightarrow$
$\texttt{R}\ (\texttt{Trace} \rightarrow \texttt{R}\ (\texttt{Trace} \rightarrow \texttt{R a}\rightarrow \texttt{R b}) \rightarrow \texttt{R}\ (\texttt{Trace} \rightarrow \texttt{R}\ [\texttt{a}] \rightarrow \texttt{R}\ [\texttt{b}]))$

Tracing *partial applications* will only be possible if a transformed function returns an R-value for each argument it is applied to. Note that this requirement is indeed fulfilled.

3.2 Creating traces for expressions

The trace for an expression depends on the *evaluation context* in which it is computed. The evaluation context of an expression is simply the trace of the redex in which the expression occurs. Given an evaluation context t, we will now define the transformation scheme for expressions, $\mathcal{E}[\![\mathrm{e}]\!]_t$.

Tracing identifiers Identifiers can be either `let`-bound or λ-bound. Identifiers in patterns are λ-bound, and they already have traces, so the transformation of such identifiers is the identity transformation. Definitions of `let`-bound identifiers are transformed to expect a trace as argument; given a trace they produce a `Nm` node for this particular instance of the identifier.

$$\mathcal{E}[\![\mathit{ident}_\lambda]\!]_t \Rightarrow \mathit{ident}$$
$$\mathcal{E}[\![\mathit{ident}_{\texttt{let}}]\!]_t \Rightarrow \mathit{ident}\ t$$

Fig. 2. Transforming identifiers.

Tracing constructed values Figure 3 shows the transformation rules for constructed values. The trace for a constructed value with no components is just the `Nm` node of the constructor. If the constructed value has components, its trace is an `Ap` node containing traces for the constructor and for each component.

$$\mathcal{E}[\![\mathit{conid}\ e_1\ \ e_2\ \dots\ e_n]\!]_t \Rightarrow \texttt{con}_n\ \ t\,\mathcal{N}[\![\mathit{conid}]\!]\ \ \mathit{conid}\ \mathcal{E}[\![e_1]\!]_t\ \ \mathcal{E}[\![e_2]\!]_t\ \dots\ \mathcal{E}[\![e_n]\!]_t$$

```
con_0 t nm conid = R conid (Nm t nm)
con_n t nm conid e_1@(R _ t_{e_1}) e_2@(R _ t_{e_2}) ...e_n@(R _ t_{e_n}) =
      R (conid e_1 e_2 ... e_n) (Ap t [Nm t nm, t_{e_1}, t_{e_2}, ..., t_{e_n}])
```

Fig. 3. Transforming constructed values.

Tracing function applications Figure 4 shows the rule for transforming function applications to make them create traces as well as results. The auxiliary function vap_n builds the trace node and then applies the function to one of the arguments, leaving it to another auxiliary function ap_n to apply it to the rest of the arguments. (Functions need to take arguments one at a time, so that partial applications have traces).

Case expressions are transformed in much the same way as function applications, with `case` as the function name and the scrutinised expression as the argument.

$$\mathcal{E}[\![f\ e_1\ \ e_2\ \ldots\ e_n]\!]_t \Rightarrow \texttt{vap}_n\ t\ \mathcal{E}[\![f]\!]_t\ \mathcal{E}[\![e_1]\!]_t\ \mathcal{E}[\![e_2]\!]_t\ \ldots\ \mathcal{E}[\![e_n]\!]_t$$

$$\begin{array}{l}
\texttt{vap}_n\ \texttt{t (R f t}_f\texttt{) e}_1\texttt{@(R \_ t}_{e_1}\texttt{) e}_2\texttt{@(R \_ t}_{e_2}\texttt{)}\ \ldots\ \texttt{e}_n\texttt{@(R \_ t}_{e_n}\texttt{) =} \\
\qquad \texttt{ap}_{n-1}\ \texttt{t}_1\ \texttt{(f t}_1\ \texttt{e}_1\texttt{) e}_2\ \ldots\ \texttt{e}_n \\
\qquad \texttt{where t}_1\ \texttt{= Ap t [t}_f\texttt{, t}_{e_1}\texttt{, t}_{e_2}\texttt{, ..., t}_{e_n}\texttt{]} \\
\\
\texttt{ap}_n\ \texttt{t (R f t}_f\texttt{) e}_1\texttt{@(R \_ t}_{e_1}\texttt{) e}_2\texttt{@(R \_ t}_{e_2}\texttt{)}\ \ldots\ \texttt{e}_n\texttt{@(R \_ t}_{e_n}\texttt{) =} \\
\qquad \texttt{ap}_{n-1}\ \texttt{t (f t e}_1\texttt{) e}_2\ \ldots\ \texttt{e}_n \\
\texttt{ap}_0\ \texttt{t e = e}
\end{array}$$

Fig. 4. Transforming function applications.

As a simple example application of these rules, consider the transformation of the expression *f* `True` *(g x)* in the evaluation context *t*. Assume that *f* and *g* are *let*-bound functions and *x* is a λ-bound variable.

$$\begin{array}{l}
\mathcal{E}[\![\texttt{f True (g x)}]\!]_t \\
\Rightarrow \\
\texttt{vap}_2\ \texttt{t}\ \mathcal{E}[\![\texttt{f }]\!]_t\ \mathcal{E}[\![\texttt{True}]\!]_t\ \mathcal{E}[\![\texttt{g x}]\!]_t \\
\Rightarrow \\
\texttt{vap}_2\ \texttt{t (f t) (con}_0\ \texttt{t "True" True) (vap}_1\ \texttt{t (g t) x)}
\end{array}$$

Tracing `let`-expressions The transformation rule for `let`-expressions is shown in Figure 5. It uses the $\mathcal{F}$ scheme (defined in §3.3) to transform the local definitions.

$$\mathcal{E}[\![\texttt{let}\ \{d_1\texttt{;}\ d_2\texttt{;}\ \ldots\texttt{;}\ d_n\}\ \texttt{in}\ e]\!]_t \Rightarrow \texttt{let}\ \{\mathcal{F}[\![d_1]\!]\texttt{;}\ \mathcal{F}[\![d_2]\!]\texttt{;}\ \ldots\texttt{;}\ \mathcal{F}[\![d_n]\!]\}\ \texttt{in}\ \mathcal{E}[\![e]\!]_t$$

Fig. 5. Transforming `let`-expressions.

Tracing other types of expressions Case expressions are transformed in much the same way as function applications, with `case` as the function name and the scrutinised expression as the argument. This is useful when browsing the trace: one can see in the source code which branch was used in a case expression.

Lambda expressions are treated as functions with the function name `\`. This works surprisingly well, since source links from the trace make it easy to examine the full lambda abstraction, if necessary.

High level syntactic constructs such as list comprehensions and sequence generators are currently traced as if replaced by their standard translations using list-processing functions.

3.3 Transforming function definitions to create traces

Figure 6 shows the transformation scheme $\mathcal{F}$ for function definitions. A traced function accepts a trace as its argument (see Figure 2) and returns an `R`-value containing a translated version of the function and a trace for the function identifier. The translated function takes one argument (and application-site trace) at a time, and for each one produces a new `R`-value, representing the partially applied function. We use an auxiliary function fun_n to build the trace node for the identifier and to accept arguments one at a time. The translated function f itself is not called until enough arguments are available for a full (saturated) application.

The $\mathcal{P}$ transformation scheme transforms patterns. Currently we only have limited support for using guards. The computation of guards of a function are traced, but as soon as one of them is true and the corresponding function body is evaluated, the traces of the guards are discarded.

$$\mathcal{F}\llbracket f\ p_1\ \ldots\ p_n\ \texttt{=}\ e \rrbracket \quad \Rightarrow f = \mathrm{fun}_n\ f'\ \mathcal{N}\llbracket f \rrbracket \text{ where } f'\ t\ \mathcal{P}\llbracket p_1 \rrbracket \ldots \mathcal{P}\llbracket p_n \rrbracket = \mathcal{E}\llbracket e \rrbracket_t$$

$$\begin{array}{ll}
\mathcal{P}\llbracket \_ \rrbracket & \Rightarrow \_ \\
\mathcal{P}\llbracket ident \rrbracket & \Rightarrow ident \\
\mathcal{P}\llbracket conid\ p_1\ \ldots\ p_n \rrbracket & \Rightarrow \texttt{R}\ (conid\ \mathcal{P}\llbracket p_1 \rrbracket \ldots \mathcal{P}\llbracket p_n \rrbracket)\ \_
\end{array}$$

```
funn f fs t =
   R (λt1 e1→
      R (λt2 e2→
         ...
           R (λtn en→ f tn e1 ... en)
             tn-1)
         ...
        t1)
      [Nm t fs]
```

Fig. 6. Transformation scheme for function definitions.

We illustrate the $\mathcal{F}$ scheme with an example, in which two simple functions are transformed.

```
f x = g x x True + x
g x y True = x*y
g x y False = 1
      ⇒
f = fun1 f' "f"
    where f' t x = vap2 t ((+) t)
                        (vap3 t (g t) x x (con0 t "True" True)) x
g = fun3 g' "g"
    where g' t x y (R True _) = vap2 t ((*) t) x y
          g' t x y (R False _) = con0 t "1" 1
```

3.4 Two problems and their remedies

Over-saturated applications Trace-construction as we have described it so far works well for both partially and fully saturated applications. But it fails to build correct traces for function applications that *over-saturate* functions (i.e. the function in the application is given more arguments than are shown in the function's definition). In the following example, `f 2 3` is an over-saturated application, since `f` is given two arguments, although by definition `f` has arity 1.

```
let f x = g x
    g x y = x+y
in  f 2 3
```

Unfortunately, traces for calls made from within an over-saturated function are lost. The reason can be seen in the definition of ap_n. If an application of arity m over-saturates a function of arity n, the functional argument f to ap_{m-n} will represent the result of a saturated call complete with a trace t_f, *which ap ignores!*

Including the trace for the function part in *every* call to an *ap* would be excessive. Suppose we are tracing the function call *f 1 2 3*: the result would be a trace with separate nodes for each of *f 1 2 3*, *(f 1) 2 3*, and *((f 1) 2) 3*.

We only want to include the trace of the function part *if* it is the result of a call to a fully saturated function. But how do we know that? Examining the definition of fun_n we see that an unsaturated function is applied to its own trace. When ap_n applies a function, it can check if the trace in the result *is the same* as the one passed to the function as first argument. If the traces are the same, the application is partial; otherwise it is saturated.

It is not enough to test if the trace objects denote equal values, we must check that the trace objects really are one and the same. This sounds simple, but it is not possible in a pure functional language to test equality of *objects*, one can only test equality of *values*. We overcome the problem by introducing an impure primitive for *pointer equality* which we call `sameAs` — it is the *only* impure function we need at run-time. The modified version of ap_n is presented in Figure 7.

Unevaluated applications When the computation has finished, exploration of the final trace can start, either from a faulting expression or some part of

```
ap_n t (R f t_f) e_1@(R _ t_e1) e_2@(R _ t_e2) ... e_n@(R _ t_en) =
      if t `sameAs` t_f then
         ap_{n-1} t (f t e_1) e_2 ... e_n
      else
         let t' = Ap t [t_f, t_e1, t_e2, ..., t_en]
         in ap_{n-1} t' (f t' e_1) e_2 ... e_n
ap_0 t e = e
```

Fig. 7. The modified version of ap_n.

the output. But what if we are interested in some saturated but unevaluated function application?

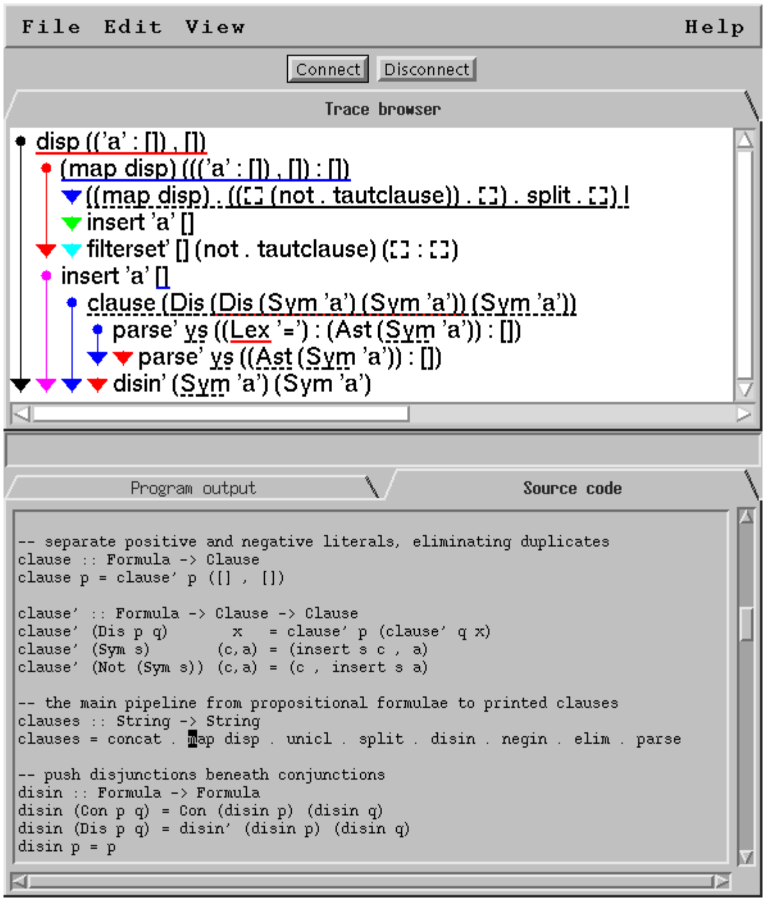
There is a conflict here between needing extra evaluation to obtain a trace, and wishing to avoid the extra evaluation to preserve the usual behaviour. If we evaluate the application to obtain the trace, the computation may diverge, giving us different behaviour for traced computated than for non-traced computations, which is clearly not acceptable.

We had a similar problem with partially applied functions, which we solved by forcing functions to return a new function and a trace for each argument. But when the function is fully saturated, the result may or may not be evaluated.

We solve this problem by introducing a new constructor `Sat` in the `Trace` datatype. A `Sat` node is introduced when a function application becomes fully saturated, and contains two parts: the trace of the fully saturated function call, and the trace of the result of the function call, which may be unevaluated. When the display program encounters a `Sat` node in the trace, it checks whether the result is evaluated. If so, the trace of that result is used; if not, the saturated function call and its trace is used instead. Note that `Sat` can be seen as an exceptional *forward* pointer in the trace, and that it is only useful as long as the result part is unevaluated. The revised definition of $\texttt{fun}_n$ is given in Figure 8.

```
fun_n f fs t =
   R (λt_1 e_1→
      R (λt_2 e_2→
         ...
            R (λt_n e_n→ let R r t' = f t_n e_1 ... e_n in R r (Sat t_n t'))
              t_{n-1})
         ...
        t_1)
     [Nm t fs]
```

Fig. 8. The new definition of fun_n used in the function definition transformation.



```
disp (('a' : []) , [])
  (map disp) ((('a' : []) , []) : [])
    ((map disp) . ((⬚ (not . tautclause)) . ⬚) . split . ⬚) !
    insert 'a' []
    filterset' [] (not . tautclause) (⬚ : ⬚)
  insert 'a' []
    clause (Dis (Dis (Sym 'a') (Sym 'a')) (Sym 'a'))
      parse' ys ((Lex '=') : (Ast (Sym 'a')) : [])
      parse' ys ((Ast (Sym 'a')) : [])
      disin' (Sym 'a') (Sym 'a')
```

```
-- separate positive and negative literals, eliminating duplicates
clause :: Formula -> Clause
clause p = clause' p ([] , [])

clause' :: Formula -> Clause -> Clause
clause' (Dis p q)       x   = clause' p (clause' q x)
clause' (Sym s)       (c,a) = (insert s c , a)
clause' (Not (Sym s)) (c,a) = (c , insert s a)

-- the main pipeline from propositional formulae to printed clauses
clauses :: String -> String
clauses = concat . map disp . unicl . split . disin . negin . elim . parse

-- push disjunctions beneath conjunctions
disin :: Formula -> Formula
disin (Con p q) = Con (disin p) (disin q)
disin (Dis p q) = disin' (disin p) (disin q)
disin p = p
```

Fig. 9. A session snapshot of the interactive trace display program.

4 Displaying trace information

In this section we describe the interactive display component of our tracer. Information is presented to the user in three panels: (1) redex trails; (2) program sources; (3) program output[1] (see Figure 9 for an example). Though the user is free to browse any of these panels by scrolling, links to and from the redex trails provide the most important form of access. The programmer has access by mouse-click from any identifier in the trace display to both the corresponding definition and the relevant applied occurrence of the identifier in the source program.

---

[1] The current tracer restricts I/O in the traced program to a single textual input and a single textual output.

The number of nodes in a full trace exceeds the number of reductions in the traced computation, so typical traces are large structures containing a great deal of information. How much of this information can sensibly be presented is limited by both the *capacity of the user* and the *capacity of the screen* used for display.

4.1 How the user controls the display

At the start of a tracing session the redex-trail panel may contain an undefined function application detected and reported as a run-time error. If there was no such error the panel is initially empty; by selecting some fragment of text from the output panel, the user requests an initial display of the parent redex for that text.

The user now acts as a source of demand, controlling the extent of 'display by need'. Clicking over an already-displayed subexpression in a trace node requests the display of its parent redex. Moving the mouse to a different part of the display not only changes the currently selected expression; it also causes highlighting of three classes of *related expressions.* If E is the currently selected expression:

- all *shared occurrences* of E currently on display are highlighted in the same colour as E itself;
- all *expressions with the same parent redex* as E are highlighted in a different colour;
- if the *parent redex* of E is already on display, it is highlighted using a third colour.

4.2 Making the most of screen space

Screen-space is an all-too-scarce resource when displaying a complex structure. We have used a combination of techniques to reduce the amount of display space needed to show part of a trace.

Displaying expressions on one line Even a single expression may be large, yet the user must be able to view the relationships between several of them. We therefore confine each expression to a single line of the display. To make this possible:

- for all expressions, details below a specified depth in the parse-tree are suppressed by default – clicking on a place-holder for any unprinted subexpression requests more detail if needed;
- when tracing to discover the cause of a fault, subexpressions that are *never evaluated* can be suppressed with a distinguished marker – their details *cannot be relevant*;
- to make space for extra detail in one part of an expression the user can click on other parts to collapse them to place-holders;
- when all else fails, horizontal scrolling is available!

Display by need revisited We have already described how parent redexes are displayed only as and when the user demands them. Continuing the analogy with a lazy evaluator:

- it is important to have shared references to a common parent redex rather than displaying it several times — hence the colour-coded information about shared, or already-displayed, parent redexes as the mouse-cursor traverses the display;
- also, re-clicking on a subexpression with a currently displayed parent redex removes the parent (and any displayed parents of its subexpressions, recursively) from the display.
- when all else fails, vertical scrolling is available!

5 Evaluation

Although our tracing system is still being refined, in this section we offer some observations and measurements by way of a preliminary evaluation. We address the following questions:

- What class of functional programs can the tracing system deal with?
- How much extra time and space does its use require?
- To what extent is the tracing scheme portable to other implementations, or even to other functional languages?
- How well does the tracer achieve its purpose as a useful tool for understanding (possibly faulty) functional computations?

5.1 Class of traceable programs

Our current implementation of the tracer is by extension of the nhc13 compiler. Programs to be traced can make use of a very large subset of Haskell 1.3, including type and constructor classes. We have traced a variety of programs, not all of them written by ourselves. The largest programs extend to several modules and some 2000 lines, yet few changes were needed to the original sources.

The most significant of the necessary changes restrict the I/O components of programs to a single output only. We are unsure as yet how best to trace I/O more comprehensively.

Some other things are not yet handled as we eventually intend. For example, comprehensions are currently traced as if replaced by their standard translations into compound applications of list-processing functions; we'd prefer something closer to the source program. Guard expressions are faithfully evaluated, but unless a fault occurs during a guard computation the redex trail for the Boolean value is not currently incorporated in the final trace; we'd like to make this information available to the programmer. We see no fundamental difficulties here; it's just a matter of a bit more work attending to details.

Table 1. The compile-time costs of generating redex trails. Only the `cichelli` program has more than one module.

	compilation time (s)		size of object code (kb)	
	normal	tracing	normal	tracing
`cichelli`	18.79	50.58	51.21	170.25
`clausify`	8.32	20.06	39.26	104.17
`primes`	6.51	14.39	13.15	37.33

Table 2. The run-time costs of generating redex-trails. Garbage-collection time is not included because it varies with heap size.

	reduction time (s)		max live heap (b)		number of
	normal	tracing	normal	tracing	reductions
`cichelli`	1.05	21.47	34 k	13 M	354 636
`clausify`	2.05	73.32	7 k	28 M	1 405 539
`primes`	5.09	33.50	60 k	54 M	520 073

5.2 Time and space costs

Using the tracer incurs extra costs both at compile-time and at run-time. We give here some figures for the construction of *complete* redex trails. We take three example programs: `cichelli` uses a brute-force search to construct a perfect hash function for a set of 16 keywords; `clausify` simplifies a given proposition to clausal form; `primes` generates the first 2,500 primes using a wheel-sieve. All three programs are included in the NoFib benchmark suite [Par93].

Table 1 shows the added costs apparent at *compile-time*. Much the same results are obtained for all three examples: both compile-time and the size of resulting object-code increase by a factor of two or three when we apply the program transformation to build redex trails.

Table 2 illustrates the *run-time* costs of generating redex trails; figures for the traced computations include the costs of evaluating both the result and the trace in full. Recording the details of the parent redex for every subexpression in every instance of every function body is not cheap! A trace-constructing computation takes 6–36 times longer than a normal one. The trace structure itself occupies from 20–100 bytes per application: so to construct traces with a million recorded reductions we need plenty of memory on our workstations! The wide variations in the performance figures for the different programs are accounted for by factors such as the arity of functions involved.

A final aspect of performance is less easily measured. The interactive trace-display program must respond rapidly to the user's requests. For all the examples we have tried to date, response-time has never been a problem.

5.3 Portability

Our current tracing system is implemented in a version of Röjemo's nhc compiler [Röj95]. The transformation to introduce trace values is performed on an intermediate representation of programs internal to nhc, and nhc's run-time system has been modified to support traced computations.

However, our tracing scheme is not closely tied to this one compiler. The results of the program transformation could be expressed in source form and supplied to a different Haskell compiler. Only one special primitive is needed at run-time: `sameAs` tests whether two traces are the same in a referentially opaque sense as explained in section §3.4. There are just two other additions to the run-time system: one retains access to output; the other provides a link to the display program.

We see no reason why our tracing scheme could not be applied to other lazy functional languages.

5.4 Will it solve 'The Tracing Problem'?

This is both the most important and the most difficult question to answer satisfactorily. A tracer might cope efficiently with a full range of programs and port easily to new compilers, yet fail the acid test: do programmers use it in practice to solve their problems?

At this stage, apart from our own experience trying out the tracer, we have only the results from an extended student exercise with an earlier prototype. Providing direct links between the trace and the source program turns out to be even more important than we expected; as do the links to textual input and output. But from these early trials we are quietly optimistic.

Currently we cannot handle non-terminating programs, unless a *black hole* [Jon92] is detected. A possible solution to this problem is to let the user interrupt the computation (e.g. by pressing `control-C`), and start browsing from the trace of the interrupted application, which must form a part of the cycle that caused the program to loop.

We would welcome enquiries from readers who teach functional programming and would be interested in 'class-testing' a suitable version of the tracer.

6 Related work

Some previous systems for tracing functional programs systems have been based on monitoring the *series of events* in a computation [KHC91], perhaps with the ability to examine events immediately preceding or following a suspected fault [TA90]. This approach is viable for a strict functional language with eager evaluation but breaks down for non-strict languages with a lazy evaluation strategy. In a Haskell computation closures for function applications can lie dormant for many reductions. If no equation matches when a closure comes to be evaluated it may be irrelevant to ask 'What happened just before this?' because preceding events may have nothing to do with the faulty closure. Simple traces of event sequences have been tried for lazy languages, but found wanting.

Our tracing based on *redex trails* has in common with Nilsson and Fritzson's system [NF94] the construction of a computational history tracing the origin of expressions back through ancestral redexes. However, the structure of their computational history is rather different. A node in their trace tree contains a saturated function application, along with its result and a list of histories for the function applications evaluated in the body of the saturated function. They also suggest using the algorithmic debugging technique [Sha82], which is a method of navigating in the trace by answering questions about the equivalence of (possibly complex) expressions. Naish and Barbour [NB95] use a source-to-source transformation to produce a computational history (which they call an evaluation tree) similar to Nilsson's and Fritzson's for a simple lazy functional language. Nilsson and Sparud [Spa96,NS97] further explores the creation and browsing of such evaluation trees, and address the problem of high memory consumption by constructing the history incrementally.

Hazan and Morgan [HM93] suggests a tracing technique that could be seen as a simplified abstraction of ours. Their *distinguished paths* are like our redex trails but contain *only function names from fully saturated applications*. For example, if `main` needs the value of `f 1`, and `f` needs the value of `head []` the consequent run-time fault will be assigned the distinguished path `main` → `f` → `head`.

7 Conclusions and Future Work

So what have we achieved? We have developed a new scheme for constructing a complete trace of a lazy functional computation, based on redex trails. We have expressed this scheme in rules for program transformation, and we have implemented it in a Haskell compiler: the trace is itself represented as a functional data structure and the compiler transforms programs so that they build traces of their own execution. We have also developed a display program that allows the interactive exploration of redex trails, fully linked to the source program.

Previous attempts at similar schemes have often put severe limits on the language used to express traced programs, or on the amount of detail recorded in traces. We aim to build fully-detailed traces for full Haskell computations. Though our present tracer handles most of Haskell, we are keen to lift some of the current restrictions on I/O; special forms such as comprehensions and guards also need more attention as explained in §5.

The usefulness of such a tracer for functional programmers at large critically depends on its performance and capacity. Our ultimate goal is a tracer that can be applied to large computations – such as a functional language compiler. Though applications to date have been modest, as illustrated in §5 they do include fully-traced computations of over a million reductions, derived from multi-module programs several pages in length.

However, the speed penalty of tracing is still too high. We hope to reduce it so that in the *worst case* slow-down is no more than a factor of ten. There is certainly scope for speeding up the trace-constructing machinery — for example, by moving key auxiliary functions into the run-time system.

Space costs are even more critical. The important observation here is that even the most enquiring programmer cannot really want to follow a trail through hundreds of thousands of expressions! Rather they will be interested in just a few selected paths, though there is no way of knowing in advance just which paths these will be. We have just begun work on *trace-pruning* techniques that should substantially reduce the cost of building and storing traces, by confining them to *partial* redex trails. For example, we can bound the length of ancestral paths, truncating longer trails at each garbage collection: even bounding the length to zero there is often a useful trail from an error, since trails are still constructed between collections. Another approach is to discard details of computation inside specified modules: some early results here are very promising — for the `primes` computation, the peak amount of memory needed shrinks by a factor of 50 when we eliminate traces inside the prelude. We also have another line of attack on the space problem: we plan to write trace structures to file in a *compressed binary format* [WR97]. Storing traces in files also has the advantage that the trace of a lengthy run can be re-examined many times without re-incurring the cost of the trace-building computation.

Building a fast and space-efficient tracing system is only one side of the problem. Once the above refinements are made, we are keen to put the tracer to the test in the hands of other users. We'd like to run a series of controlled experiments measuring the speed and accuracy with which users of the tracer can find faults in programs. Only by such experiments and the outcomes of wider use can we hope to confirm that constructing and examining redex trails can be an effective part of functional programming.

Acknowledgements

Niklas Röjemo built the `nhc` compiler that we have adapted for our experiments in tracing. Anonymous PLILP referees provided helpful comments, questions and suggestions. Our work is funded by the Engineering and Physical Sciences Research Council.

References

[HM93] Jonathan E. Hazan and Richard G. Morgan. The location of errors in functional programs. In Peter Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 135–152, Linköping, Sweden, May 1993.

[Jon92] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[KHC91] A. Kishon, P. Hudak, and C. Consel. Monitoring semantics: a formal framework for specifying, implementing and reasoning about execution monitors. In *ACM Conference on Programming Language Design and Implementation (PLDI'91)*, pages 338–52, June 1991.

[NB95] Lee Naish and Tim Barbour. Towards a portable lazy functional declarative debugger. Technical Report 95/27, Department of Computer Science, University of Melbourne, Australia, 1995.

[NF94] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3), 1994.

[NS97] Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Journal of Automated Software Engineering*, 4(2):152–205, April 1997.

[Par93] W. Partain. The **nofib** benchmark suite of Haskell programs. In J. Launchbury and P. Sansom, editors, *Proc. 1992 Glasgow Workshop on Functional Programming*, pages 195–202. Springer Verlag, Workshops in Computing, 1993.

[Röj95] N. Röjemo. Highlights from nhc – a space efficient haskell compiler. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 282–292, La Jolla, June 1995. ACM Press.

[Sha82] Ehud Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, May 1982.

[Spa96] Jan Sparud. A transformational approach to debugging lazy functional programs. Licentiate Thesis, Department of Computing Science, Chalmers University of Technology, S-412 96, Göteborg, Sweden, February 1996.

[TA90] A. P. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. In *Proc. ACM conf. on Lisp and functional programming.* ACM Press, 1990.

[WR97] M. Wallace and C. Runciman. Heap compression and binary I/O in Haskell. In *Proc. 2nd ACM SIGPLAN Workshop on Haskell*, Amsterdam, June 1997.