

Tracing the Evaluation of Lazy Functional Languages: a Model and its Implementation

Richard Watson¹ and Eric Salzman²

¹ Department of Mathematics and Computing,
University of Southern Queensland,
Toowoomba, QLD 4350, Australia
`rwatson@usq.edu.au`

² Department of Computer Science,
The University of Queensland, QLD 4072, Australia
`eric@it.uq.edu.au`

Abstract. We address the problem of producing a trace of the evaluation of a program written in a lazy functional language. To avoid ambiguities and possible misunderstandings it is essential that the trace structure is defined with respect to a formally described model of program evaluation.

We provide such a formal semantics for lazy evaluation of a simple lazy language, based closely on the work of Launchbury. The trace corresponds to the sequence of expression reductions defined by the evaluation model.

We also present a scheme to generate a concrete trace of the evaluation of programs written in the target language, based on its semantic rules. We employ a two-step transformational approach: first transform the program so that, on execution, it generates a call-by-name trace as result, then further transform this trace to a call-by-need trace.

Keywords: functional programming, debugging, lazy evaluation, program transformation

1 Introduction

While there has been much progress over recent years in the design and implementation of lazy functional languages, the development of tools to enable debugging of programs written in these languages is not as well advanced. To debug a program, one must be able to observe its behaviour. In this paper we describe a technique for generating an *evaluation trace* for a program written in a lazy functional language. The trace is a history of the computational steps carried out in evaluating (executing) the program, so can form the basis of a debugging system.

Although the current work is motivated by the need for debugging tools, our focus is only on trace generation. We describe elsewhere (Watson and Salzman, 1997) an example of such a debugging tool, which we call a trace browser.

We also recognise that execution tracing provides but one possible path to understanding program behaviour; for instance, the use of execution profilers (see e.g. Runciman and Wakeling, 1993; Sansom and Peyton Jones, 1995) can reveal the time and space consumption of the functions in a program.

Before building a trace generation system, two questions must be addressed: (1) what form should the trace take, and (2) how can the trace be generated? The answers to these questions define the current work, and serve to contrast it with previous work. We have chosen to generate a *reduction* trace by *transforming* a program so that, on evaluation of the transformed program, a trace is generated as the result of the program. (Actually we generate a pair — the result of the original program together with its evaluation trace.) While others have described how a trace of reductions can be generated (see e.g. Snyder, 1990; Goldson, 1994) and how a transformed (instrumented) program can be used to produce a trace (O’Donnell and Hall, 1988; Kishon, 1992; Sparud, 1996), our work is unique in that we combine these two features together with a *formal semantic model* of lazy evaluation which serves to define the reduction steps that the trace reports. Our transform system, for which we provide a set of simple rules, uniformly handles difficult-to-implement language features such as higher-order functions; other transformational approaches have employed special rules depending on the type of a function argument (Sparud, 1996) or failed to satisfactorily address how curried application is traced (Naish and Barbour, 1996).

Our model is a big step operational semantics based very closely on the work of Launchbury (1993). To our knowledge, only Gibbons and Wansbrough (1996) have built a tracing system based on a formally defined reduction model (in their case, the lazy lambda calculus of Ariola et al. (1995)); we believe the Launchbury-style of operational model is superior to that of Ariola et al. as a means of trace definition because it is simpler, with fewer rules, and just one rule for each kind of expression in the language.

The remainder of the paper is organised as follows. In section 2 we provide a background to the problems of tracing non-strict higher-order functional programs, and refer to past attempts at their solution. In section 3 we introduce a simple lazy functional language, its semantic model, and a trace structure based on the model. In section 4 we describe the transformational tracing scheme that we have developed. In section 5 we evaluate the significance of the work reported, and suggest future research directions.

2 The tracing problem

In this section we identify the challenges associated with tracing lazy functional programs, then address the questions raised in the introduction, namely “what should a trace look like and how can it be created?”.

The properties of pure, higher-order, non-strict functional languages impose a number of interesting constraints on the task of trace generation:

- *purity* precludes the production of a trace as a side-effect.

- *higher-orderness* requires that we must be able to trace the partial application of a function, and identify the resulting abstraction.
- the *non-strict* property prevents us from reporting an argument’s value at application time. However we assume that the reader of the trace will wish to know the argument value, and so must devise a technique of associating the possible eventual evaluation of the argument with its corresponding application instance.

Our tracing system solves all these problems; it can present a user-oriented view of the evaluation trace within the constraints of the language paradigm.

Two kinds of trace have been described in the literature (the terminology is our own): an *operational* trace, which describes a sequence of actions in the order that they were performed to evaluate a program, and a *declarative* trace, which describes the result of function application (argument and result values) but places the information in a trace structure corresponding to the declarative structure of the program.

We have adopted an operational style of trace, because it supports the style of debugging which has proved effective in the imperative language domain. The programmer compares expected program behaviour with that reported in the trace — a divergence between the two indicates a potential error. The declarative trace is appealing mainly because it supports the use of the *algorithmic debugging technique* (Shapiro, 1982) which can detect program errors in a semi-automatic manner. See the work of Nilsson and Fritzson (1994), Sparud (1996), and Naish and Barbour (1996) for application of the technique to lazy functional languages. We reject the declarative trace because we believe that it does not fit well with the programmer’s mental model of lazy functional computation; reduction models are usually used to teach and explain the way functional programs work (see e.g. Bird and Wadler, 1988), so an operational trace should closely match a programmer’s concept of how functional languages are evaluated.

Three approaches to trace generation have been reported (representative work is cited):

1. modify the standard program translator to produce compiled code which, on execution, will generate a trace (Snyder, 1990; Nilsson and Fritzson, 1994).
2. build a special purpose interpreter which reports its evaluation steps (Goldson, 1994; Augustson, 1995; Gibbons and Wansbrough 1996).
3. transform the original program so that evaluation of the instrumented program generates a trace (O’Donnell and Hall, 1988; Sparud, 1996; Naish and Barbour 1996).

While all these schemes have their merits, we have chosen to employ the program instrumentation approach. This yields a portable scheme, applicable to different languages and implementations.

The program instrumentation aims to *simulate* an evaluation sequence as defined by some abstract model of evaluation. That is, rather than monitoring the activity of an evaluation machine, the original program is augmented so that the evaluation of this instrumented program will generate both a resulting value

$$\begin{aligned}
k &\in Const \\
x &\in Var \\
e \in Expr & ::= \lambda x.e \mid e_1 e_2 \mid x \mid k \mid let\ d_1 \dots d_n\ in\ e \\
d \in Decl & ::= x = e
\end{aligned}$$

Fig. 1. Abstract syntax of normalised language

and a simulated trace of its evaluation. We must be absolutely clear about what we are simulating; hence we adopt a formal operational model to describe the semantics of the language being traced.

3 A model for lazy evaluation

The lazy language we describe exhibits *call-by-need* semantics. That is, the language is non-strict (as defined by call-by-name semantics), and the evaluation of expressions is shared where possible. In our model, only those expressions bound to variables are shared: once a variable is evaluated to a whnf value, further references to that variable yield the value directly. More extensive sharing is possible *via* program transformation (Peyton Jones, 1987, Chapter 15), but is beyond the scope of this work.

We use a formal evaluation model based on Launchbury’s operational semantics for lazy evaluation (Launchbury, 1993). Our model is very similar, so in the following we will present the model with little explanation but focus on the differences between our model and Launchbury’s. Looking ahead, the trace structure that we will generate includes one object for each judgement present in the semantic rules, so our justification for departure from the original model is to provide a better basis for trace description.

We describe here a very simple lazy language. We have developed a model and tracing system for a larger language including algebraic data types, `case` expressions, and primitive operators (see Watson (1997) for a description of the extended language and semantics), but don’t present it for lack of space, and because tracing the extended language does not greatly extend the technique presented here.

3.1 A simple language

Figure 1 shows the abstract syntax of the language. A program is a list of declarations. Programs in this language differ from the concrete programs in two respects: (1) variables appearing in the original program are systematically renamed so that all names are unique (thus lexical scope need not be modelled by the semantic rules) and (2) function definitions in the original program take the form $x\ x_1 \dots x_n = e$ which is normalised to $x = \lambda x_1 \dots \lambda x_n.e$.

This simple language is identical to Launchbury’s with the important exception that we allow the general form of application expressions $e_1\ e_2$, rather than the normalised form $let\ x = e_2\ in\ e_1\ x$.

Constant	$(\Gamma, n) : k \Downarrow (\Gamma, n) : k$
Abstraction	$(\Gamma, n) : \lambda x.e \Downarrow (\Gamma, n) : \lambda x.e$
Application	$\frac{(\Gamma, k) : e_1 \Downarrow (\Delta, m) : \lambda x.e \quad (\Phi, n) : \hat{e}[x^n/x] \Downarrow (\Theta, o) : z}{(\Gamma, k) : e_1 e_2 \Downarrow (\Theta, o) : z}$ <p style="text-align: center;">where $n = m + 1$ $\Phi = \Delta \cup \{x^n \mapsto e_2\}$</p>
Variable	$\frac{(\Gamma, k) : e \Downarrow (\Delta, l) : z}{((\Gamma, x^n \mapsto e), k) : x^n \Downarrow ((\Delta, x^n \mapsto z), l) : z}$
Let	$\frac{(\Delta, k) : e \Downarrow (\Phi, l) : z}{(\Gamma, k) : \text{let } x_1^r = e_1 \cdots x_n^r = e_n \text{ in } e \Downarrow (\Phi, l) : z}$ <p style="text-align: center;">where $\Delta = \Gamma \cup \{x_1^r \mapsto e_1, \dots, x_n^r \mapsto e_n\}$</p>

Fig. 2. Call-by-need semantic rules

3.2 Dynamic rules

The rules appear in figure 2. The following naming conventions are used:

$$\begin{aligned} \Gamma, \Delta, \Phi, \Theta &\in \text{Heap} = \text{Var} \mapsto \text{Expr} \\ z \in \text{Val} & ::= \lambda x.e \mid k \end{aligned}$$

Judgements are of the form $(\Gamma, k) : e \Downarrow (\Delta, l) : z$, which means: “After k applications have occurred, the expression e , in the context of the set of bindings in Γ , evaluates to the value z . Following the evaluation, l applications will have occurred, and the new heap is Δ .”

Variables in the rules appear either as x , a statically declared program variable name (see the application rule), or as x^n , a dynamically created instance of the program variable x . The instance x^n is created due to the n^{th} invocation of the application rule. This explicit dynamic renaming, which subsumes the limited renaming required under Launchbury’s rules, is a key element of our rules; it also explains the presence of the application counter that is paired with the heap in the rules.

The \hat{e} notation is conditional α -conversion: it denotes renaming of all **let**-bound variable *names* defined within expression e , except those defined in the body of an abstraction, to specific variable *instances*. Note that, while Launchbury only performs renaming of variables when required to avoid name capture, we rename *all* variables. The \hat{e} operation, and the $[x^n/x]$ substitution, are *dynamic variable renaming*, but an equivalent and more appropriate term is *instance creation*. The use of \hat{e} captures the intuitive notion that a **let**-bound variable within the body of a lambda abstraction is a dynamic object in that its value is (potentially) dependent upon the value of a formal parameter, which is itself dynamically bound by function application.

Apart from the adoption of an explicit and universal variable renaming scheme, we differ from Launchbury primarily in the application and variable rules. We bind a renamed formal parameter on the heap rather than perform a β -substitution. This corresponds to the operational semantics described briefly by Ariola et al. (1995). Launchbury himself also proposes it as a viable alternative. We carry out dynamic renaming of variables (instance creation) in the application rule while Launchbury does so in the variable rule. We believe our approach is intuitively more appealing as it reflects the notion that instances are created due to application, not as a consequence of variable evaluation. Launchbury uses β -substitution, effectively removing any reference to formal parameters from a reduction proof, yet his use of a normalisation step to replace argument expressions by an extra variable effectively re-introduces it; our scheme uses formal variables directly, hence our claim to a nicer rule set because we don't need to introduce the extra pseudo-parameter to an application. Our argument binding approach allows the use of the general form of application expressions (in Launchbury's semantics, argument expressions must be variables), though at the cost of dynamic parameter renaming, a larger heap, and an extra reduction step for each evaluated argument.

3.3 An abstract trace

An abstract trace is a representation of the sequence of semantic rules applied to reduce an expression to normal form. There are a number of possible forms that this trace can take; for instance, Launchbury shows an example which depicts the trace as a linear textual form of a natural deduction proof tree. We adopt a *sequential* trace structure, which mirrors the sequence of rule applications, and corresponds to the concrete trace structure described in the next section.

The rules for generating the sequential reduction trace, called rt_{seq} , appear in figure 3. There are three rules; the choice of rule depends upon the number of precedent judgements in the semantic rule. The abstract trace is a sequence of *objects*, each of which is either an expression or, in the case of the application rule, an (expression,trace) pair. Note that the care with which we have constructed the semantics is reflected in the simplicity of the trace. Because the terminal value of the rightmost judgements for each rule is the same (for each rule, examine the rightmost expression above and below the horizontal line), we only need report initial expressions of a rule. This simplification is not possible using Launchbury's rules.

As a simple example, consider the sequential trace of a let expression.

$$\begin{array}{l}
 [\text{let } a = 1, f = \lambda x.x \text{ in } f a, \\
 (f a, [f, \lambda x.x]), \\
 x, \\
 a, \\
 1 \]
 \end{array}$$

In the second line, the trace object for the application $f a$ includes a trace of the evaluation of its function f to an abstraction. This is a single step here but may

$$\begin{array}{ll}
rt_{seq} ((\Gamma, k) : e \Downarrow (\Delta, l) : z) & = [e] \quad \text{Const, Abs} \\
rt_{seq} \left(\frac{J_1}{(\Gamma, k) : e \Downarrow (\Delta, l) : z} \right) & = e : rt_{seq}(J_1) \quad \text{Var, Let} \\
rt_{seq} \left(\frac{J_1 \ J_2}{(\Gamma, k) : e \Downarrow (\Delta, l) : z} \right) & = (e, rt_{seq}(J_1)) : rt_{seq}(J_2) \quad \text{App}
\end{array}$$

Fig. 3. Sequential trace rules

in general consist of many reductions (see the example in figure 5). The final object in a function reduction is always an abstraction, and the object following an application (here line three) is the body of that abstraction, with one or more lambda variables bound to arguments.

3.4 A concrete trace

A concrete trace, which provides a basis for the trace generation scheme described in the following section, can be derived from the rt_{seq} trace, and is described by the following Haskell data definitions:

```

type Trace = [Step]
data Step = Ks String           -- constant
          | Abs String         -- abstraction
          | App Trace          -- application
          | VP String | VL String -- variable
          | LExp String [String] -- let expression

```

Each trace object is “tagged” by a data constructor to indicate its expression class, and the constructor arguments describe which particular expression is being described. Apart from the `Ks` object, which simply reports a constant value, all other objects rely on the systematic generation of a unique “name” for all variables and abstractions. These names are derived from the lexical structure of the original program and take the form $level_1.level_2.\dots.name$ where $level_1.\dots.level_n$ describes the declaration level at which $name$ appears. New levels are “created” by declarations and application expressions.

This unique naming, used only in the trace objects, is how we solve the problem of reporting lambda abstractions in a trace, thus enabling higher order functions to be traced.

The application object does not directly indicate the expression (say $e_1 \ e_2$) it denotes, but it does include a trace of evaluation of e_1 , from which e_1 can be determined. To preserve non-strict semantics, the evaluation of e_2 cannot be generated at application time, though it may well appear later in the trace if the argument is evaluated. Associating this argument evaluation with its binding application is a major challenge — see section 4.4.

The `let` expression object `LExp` takes two arguments: the first is the name of the abstraction in which it occurs, and the second is the list of variables defined within the expression. This information is vital in performing the trace transform from call-by-name to call-by-need, which we sketch in the section 4.4.

As a final note on trace objects, we observe that the distinction between `let`-bound and formal parameter variables, traced respectively by `VL` and `VP`, is unnecessary from a trace generation perspective, but serves only to aid in trace interpretation by the reader.

4 A tracing transform

In this section we describe how a call-by-need evaluation trace can be produced by firstly generating a raw call-by-name trace as the result of executing an instrumented program, and then how the raw trace can be transformed to call-by-need. We consider the general approach to program instrumentation in sections 4.1 and 4.2, before presenting the instrumentation (§4.3) and the trace transform (§4.4). Example traces appear in section 4.5.

4.1 Approaches to program instrumentation

Simply stated, our aim is to transform some original program (written in the simple language just described) into a Haskell program that on execution will produce a reduction trace. This *concrete* reduction trace should be equivalent to the rt_{seq} *abstract* trace that would be produced by application of the call-by-need semantic rules to the original program.

The one-to-one correspondence between the expression types of our language and the reduction rules we wish to simulate suggests an expression-based transform. For all expressions e in the original program:

$$\mathcal{TE}[e] = (e', \text{object: trace})$$

where e' is the tracing analogue to e , *object* describes e , and *trace* is the trace of evaluation of e . For whnf expressions, *trace* is empty, and for applications, *object* includes the trace of evaluation of the function which is applied. This mirrors precisely the rt_{seq} trace format.

This approach is *bottom-up* in the sense that a complete trace is assembled by building sub-traces and gluing them together. An alternative *top-down* scheme looks like:

$$\mathcal{TE}[e] = \lambda \text{trace}.(e', \text{trace} \# [\text{object}])$$

where $\#$ is the list concatenation operator. Here the trace is built by adding to the end of the sequence; of necessity a state (the trace so far) must be carried through the computation. The encapsulation of e' within an abstraction results in a less efficient instrumented program, as the amount of sharing carried out by the implementation language system is reduced. For this reason we have adopted the bottom-up approach.

4.2 Call-by-name, call-by-need, and transform state

We show in this section that a stateless, bottom-up transform cannot provide a call-by-need trace — it can only manage call-by-name. On the other hand, inclusion of suitable state (specifically: the evaluation status of variables) can enable the creation of a call-by-need trace. This is an important result as it justifies the two-step call-by-need trace generation scheme we will describe.

We sketched above a simple, stateless, bottom-up transform. Consider the use of this kind of transform to instrument the declaration

$$e = \dots x \dots x \dots$$

where we assume both the x variables occur in the same scope. The instrumented code becomes

$$\mathcal{TE}[e] = \dots \mathcal{TE}[x] \dots \mathcal{TE}[x] \dots$$

Now assuming both variables x are evaluated, then by referential transparency both $\mathcal{TE}[x]$ terms must evaluate to the same (result,trace) pair. That is, two identical traces of evaluation of x will be produced. This does not reflect the notion of sharing inherent in call-by-need semantics, whereby we would expect one of the variable evaluation traces to be shorter as it would show reduction directly to a normal form because the evaluation of the other x variable had already taken place.

So how can we implement call-by-need instrumentation? We could use some form of static strictness analysis to determine the ordering of variable evaluations and then apply a different transform to the two sub-expressions, one resulting in a complete trace and the other generating a trace of just the resulting value. There is a more straightforward and reliable method. The idea is to carry information about the evaluation status of variables and then decide *at evaluation time* to generate an abbreviated trace if necessary. The transformed version of our earlier example might be

$$\begin{aligned} \mathcal{TE}[\dots x \dots x \dots] = & \text{let } (z, t) = \mathcal{TE}[x] \\ & x' = \text{if } x \text{ has been evaluated then } (z, [z]) \\ & \quad \quad \quad \text{else } (z, t) \\ & \text{in } \dots x' \dots x' \dots \end{aligned}$$

Clearly our transformed program must maintain a *runtime* state to be able to determine if a variable (x in our example) has been previously evaluated.

4.3 Call-by-name program instrumentation

Figure 4 shows the transformation rules. Before these transformations can be applied, all function declarations are normalised to create explicit lambda abstractions. Transforms are written as

$$\mathcal{TE}[e] = \text{'a Haskell expression'}$$

Each transform maps from the abstract syntax of the original program (inside `[]` brackets) to concrete Haskell syntax (shown in `typewriter` font). In practice,

$$\begin{aligned}
\mathcal{TD} \llbracket x = e \rrbracket &= x = \mathcal{TE} \llbracket e \rrbracket \\
\mathcal{TE} \llbracket k \rrbracket &= (k, [\text{Ks } "k"]) \\
\mathcal{TE} \llbracket \lambda x. e \rrbracket &= (\lambda x \rightarrow \mathcal{TE} \llbracket e \rrbracket, [\text{Abs } \textit{absName}]) \\
\mathcal{TE} \llbracket e_1 e_2 \rrbracket &= \text{let } (\text{fun}, \text{funTr}) = \mathcal{TE} \llbracket e_1 \rrbracket \\
&\quad (\text{app}, \text{appTr}) = \text{fun } \mathcal{TE} \llbracket e_2 \rrbracket \\
&\quad \text{in } (\text{app}, \text{App funTr: appTr}) \\
\mathcal{TE} \llbracket x \rrbracket &= \text{let } (\text{var}, \text{varTr}) = x \\
&\quad \text{in } (\text{var}, \{\text{VP|VL}\} \textit{varName}: \text{varTr}) \\
\mathcal{TE} \llbracket \textit{let decls in } e \rrbracket &= \text{let map } (\mathcal{TD}) \textit{ decls} \\
&\quad \text{in let } (\text{letE}, \text{letTr}) = \mathcal{TE} \llbracket e \rrbracket \\
&\quad \quad \text{in } (\text{letE}, \text{LExp } \textit{abstraction bv}: \text{letTr})
\end{aligned}$$

Fig. 4. Minimal, bottom-up, call-by-name instrumentation

the transform functions also take as argument an environment which is used to calculate the names of abstraction and variable trace objects, and which also holds a symbol table to track variable names and their class.

The transform rules are straightforward so deserve little comment. Note that the application expression transform respects non-strict argument evaluation: transformed argument expression $\mathcal{TE} \llbracket e_2 \rrbracket$ is not evaluated but rather passed as argument to the (non-strict) function `fun`. The terms *varName*, *absName*, and *abstraction* are unique manufactured names, and *bv* is the list of `let`-bound variables. Their values are determined by consulting the environment, which is also updated as the original program is traversed. Details of the environment, rules for its maintenance, and the precise form of the naming convention used in the concrete trace objects can be found in Watson (1997).

4.4 Conversion to call-by-need

The raw trace created by the instrumented program is not quite what we want. There are two problems. The non-strict property of the language results in variable evaluation appearing at some possibly distant point in the trace from where it was bound by function application. This leads to problem one: the raw trace provides no link between function application and corresponding variable evaluation, so it is very difficult to interpret the trace. Secondly, the simple, stateless program transform can only provide a call-by-name view of the trace (§4.2).

We remedy this situation by transforming the raw call-by-name trace to a call-by-need trace. This trace transform performs three tasks as it traverses the trace:

1. application objects are numbered in the order dictated by the semantic rules.
2. variable trace objects are numbered with the number of the application that caused their instantiation.
3. when the evaluation of a variable instance is repeated (generating identical duplicate sub-traces), all but the initial evaluation sub-trace are replaced by a single-element sub-trace showing direct reduction to `whnf`.

The first two actions allow unambiguous identification of variable instances, and the final one converts the trace to call-by-need. Application numbering and duplicate trace removal are straightforward, with the caveat that identification of the duplicate traces relies on instance identification (step 2 above), which is not so simply achieved.

The structure of the trace alone cannot be used to guide the instance identification process as there is no inherent relationship between application order and the order of variable evaluation. We seek to do this: on traversing the raw trace, when we encounter a variable trace object we must be able to determine its instance number (the application number of its binding application, as defined by the semantic rules). We resort to consideration of the scope of variable instances to do so. We maintain a transform state as we traverse the raw trace, which lists the variable instances currently in scope; hence determining the correct instance requires a simple lookup operation. Maintaining the in-scope list is the key to this procedure. We present elsewhere (Watson, 1997) a scopeful semantics which formally describes the way in-scope lists are created.

Notice that, after eschewing state considerations in developing a simple call-by-name trace, we are now resorting to a state-based scheme to create a call-by-need trace. This is consistent with our argument in section 4.2, but we must answer the question: why did we not produce a call-by-need trace directly, using a runtime state? The answer is twofold. Firstly the two-step approach is more efficient, and secondly it allows more flexibility in that a call-by-name trace may in fact be sufficient for some purposes.

4.5 An example

Figure 5 shows three traces of evaluation of the expression

```

let  f x y = x (x y)
     id a  = a
     g     = f id
in   g g 1

```

The first (leftmost) trace is the raw call-by-name trace produced by evaluating the transformed expression. The tracing names are seen in the formal parameter and abstraction trace objects: for example the second argument to `f` is labelled as `f'` as is its binding abstraction. Alongside is the call-by-need version whose variable and application objects have been annotated with an application number. The vertical space in this trace shows where repeated evaluations (for `g` [object VL 0 `g`] and parameter `x` [object VP 1 `f'`]) have been removed.

The remaining column shows an *expression trace*, which is a sequence of successively simplified expressions. Excepting the initial expression, each expression is formed from its predecessor by reducing a sub-expression (the redex) as defined by the call-by-need trace of reductions. Unevaluated (argument) expressions appear italicised. This expression trace can be automatically generated from the

Raw call-by-name	Call-by-need	Expression
[Let z f id g, App [App [VL g, App [VL f, Abs f'], Abs f''], App [VP f', VL id, Abs id'], VP id', App [VP f', VL id, Abs id'], VP id', VP f'', VL g, App [VL f, Abs f'], Abs f''], App [VP f', VL id, Abs id'], VP id', App [VP f', VL id, Abs id'], VP id', VP f'', Ks 1]	[Let z f id g, App 5 [App 2 [VL 0 g, App 1 [VL 0 f, Abs f'], Abs f''], App 3 [VP 1 f', VL 0 id, Abs id'], VP 3 id', App 4 [VP 1 f', Abs id'], VP 4 id', VP 2 f'', VL 0 g, Abs f''], App 6 [VP 1 f', Abs id'], VP 6 id', App 7 [VP 1 f', Abs id'], VP 7 id', VP 5 f'', Ks 1]	let ... g g 1 f id g 1 (\x y -> x(x y)) id g 1 (\y -> x(x y)) g 1 x (x y) 1 id (x y) 1 (\a -> a) (x y) 1 a 1 x y 1 (\a -> a) y 1 a 1 y 1 g 1 (\y -> x(x y)) 1 x (x y) (\a -> a) (x y) a x y (\a -> a) y a y 1

Fig. 5. Trace example

call-by-need trace and is used as the basis of a trace browser described in Watson and Salzman (1997).

To relate the expression trace to the reduction trace consider the reduction trace and expression trace for the application expression $e_1 \text{ arg}$:

$$\begin{array}{ll}
 \text{App } [o_1, & e_1 \text{ arg} \\
 & o_2, & e_2 \text{ arg} \\
 & \dots & \dots \\
 & o_m], & e_m \text{ arg} \\
 o_n & e_n \\
 \dots & \dots
 \end{array}$$

The trace objects $o_1, o_2 \dots o_n$ denote expressions $e_1, e_2 \dots e_n$. Observe how a single **App** object represents a sequence of successively simplified expressions. The expression e_n is the body part of the lambda abstraction e_m , with argument arg bound to the lambda variable in e_m .

We show just one possible representation here. The expression trace can be further simplified by removing the abstractions when they obviously derive from a named function, and by removing from the trace the evaluation of “trusted” functions. Or information can be added: we could show instance numbers for variables and also include heap binding information.

4.6 Implementation

A prototype implementation of the transformation system has been carried out using the Gofer language. A series of performance tests were performed which compared, for a set of test programs, the cost of evaluating the original and transformed versions of the programs.

The relative cost of producing the call-by-name trace is approximately 60 times the cost of running the original program. If only the result of the computation is required (but not the trace), then the cost ratio is about 45. While these figures indicate a significant but perhaps manageable overhead, especially given that we are dealing with a prototype implementation, the cost of generating call-by-need traces is much worse, with a slowdown factor of between two and three orders of magnitude. Much of the cost of producing a call-by-need trace arises from the requirement to maintain a runtime state (the evaluation status of each variable); in our simple prototype the heap was implemented as a linear list, which is clearly not an efficient scheme.

5 Discussion and conclusions

In summary we consider that the approach we have outlined has a number of advantages over comparable schemes:

- + it is based on a simple, explicit, and intuitive model for lazy evaluation
- + it employs a simple trace structure, which allows straightforward transformation into other user views of the evaluation history
- + the transform scheme is very simple and uniform, yet handles curried function application and higher order functions.

These advantages must be partially offset against the major disadvantage of the cost of trace generation. In defence, we offer the following comments:

- While our measurements indicate poor absolute performance, we are unable to compare our approach to others, due to the lack of available data. How efficient are other schemes? Could it be that tracing (without resort to meddling with the internals of a compiler) is an inherently costly undertaking? We note that very recent work by Sparud and Runciman (1997) has reported relatively low cost of trace generation but does rely on support from the language translator.
- Our implementation is a prototype, and its design was dictated by speed of development, not speed of execution. We anticipate significant reductions in

execution cost if alternate implementations based on the existing transform scheme are developed.

- The production of a call-by-need trace is much more costly than a call-by-name trace. Perhaps programmers would be content with the call-by-name view of computation, which is semantically equivalent? If so then we achieve an enormous saving!
- If traces could be selectively generated, only for functions which the user considers of interest, then the other parts of the program could execute at full speed. This seems an important avenue for future investigation, as suggested by Hall et al. 1990.

There are a number of possible directions of future research based on the work reported here.

While the program instrumentation is seen to offer advantages of simplicity and portability, an alternative trace generation scheme based on a lazy abstract machine for the language semantics could be a viable approach. Initial investigations have shown that a straightforward implementation of an interpreter for the semantic rules yields a trace generator which is comparable in speed to our instrumentation approach, though it exhibited very poor space utilisation. Implementation of an abstract machine for the semantics in the style of Sestoft (1997) could be a much more promising proposition.

Although we have demonstrated how the fundamental features of lazy languages can be traced, a usable debugging system would require that advanced language features such as pattern matching, input/output, and list comprehensions be traced also. We have already extended the semantics and tracing mechanism to encompass eager primitive functions, algebraic data types, and simple pattern matching (Watson, 1997). We believe that the transformation scheme can be further extended to the other features, especially if these language features are amenable to description using an extended version of our language semantics.

In conclusion, we summarise our work as follows. We have presented a transformational approach to trace generation for lazy functional programs. The key features of the scheme are its simplicity, that it handles adequately the fundamental problems of tracing non-strict higher-order languages, and that it is based on an intuitive formal model for lazy evaluation. The transformational scheme we have presented to produce a concrete trace can be considered as a first step toward a “industrial strength” tracing system; while exhibiting poor absolute performance it may well form the basis of a more advanced and efficient trace generator.

References

- Z. Ariola, M. Felleisen, J. Mariast, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 233–246, 1995.

- M. Augustson and J. Reinfelds. A visual Miranda machine. In *Proceedings, Software Education Conference SRIG-ET'94*, Ed. M. Purvis, IEEE Computer Society Press, Los Alamitos California, pages 198–203, 1995.
- R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.
- J. Gibbons and K. Wansbrough. Tracing lazy functional languages. In *Proceedings of Computing: The Australasian Theory Symposium*, 1996.
- D. Goldson. A symbolic calculator for non-strict functional languages. *The Computer Journal*, 37(3):177–187, 1994.
- C. Hall, K. Hammond, and J. O'Donnell. An algorithmic and semantic approach to debugging. In *Proceedings of the 1990 Glasgow Workshop on Functional Programming*, pages 44–53, 1990.
- A. Kishon. *Theory and Art of Semantics-Directed Program Execution Monitoring*. PhD thesis, Department of Computer Science, Yale University, 1992.
- J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 144–154, 1993.
- L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. In *Proceedings of the 19th Australian Computer Science Conference*, 1996. Also available as Technical Report 95/27, Department of Computer Science, University of Melbourne.
- H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–369, 1994.
- J. T. O'Donnell and C. V. Hall. Debugging in applicative languages. *Lisp and Symbolic Computation*, 1(2):113–145, 1988.
- S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- C. Runciman and D. Wakeling. Heap profiling for lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, 1993.
- P. M. Sansom and S. L. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 355–366, 1995.
- P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3), 1997. To appear.
- E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.
- R. M. Snyder. Lazy debugging of lazy functional programs. *New Generation Computing*, 8:139–161, 1990.
- J. Sparud. *A Transformational Approach to Debugging Lazy Functional Programs*. Licentate thesis, Department of Computer Science, Chalmers University of Technology, 1996.
- J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proceedings of the Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP97)*, 1997.
- R. Watson. *Tracing Lazy Evaluation by Program Transformation*. PhD thesis, School of Multimedia and Information Technology, Southern Cross University, 1997.
- R. Watson and E. Salzman. A trace browser for a lazy functional language. In *Proceedings of the Twentieth Australian Computer Science Conference*, pages 356–363, 1997.