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.