Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 72 (2007) 3–16
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Using Term-Graph Rewriting Models to Analyse Relative Space Efficiency

Adam Bakewell

Department of Computer Science, University of York, York YO10 5DD, UK
ajb@cs.york.ac.uk

Abstract

Space leaks are a common operational problem in programming languages with automated memory management. Graph rewriting is a natural model of operational behaviour. This paper summarises a PhD thesis which gives a graph-rewriting framework suitable for modelling language implementations and proof techniques for determining the presence or absence of leaks. The approach is to model implementations as graph evaluators with garbage collectors. An evaluator may leak relative to another evaluator, with respect to a translation between their states. Leaks are classified according to their cause and the behaviour which exposes them.
Graphs naturally model state size, but we argue that this is too concrete. Accurate evaluators are introduced which allow for a more abstract model in which initial program size is ignored. Evaluators are compared by defining a translation between graphs. Space-safe translations, and non-standard garbage collectors, are defined as another kind of term-graph rewrite system. Leaky evaluators are detected by a proof method which searches for graphs whose evaluation trace is a self-feeding rule sequence.

Keywords: Term-graph rewriting, automated memory management, space leak

1 Introduction

Space leaks are a common operational problem in programming languages with automated memory management. Typically, space leak refers to the situation where objects persist in memory after they are known to be unwanted. Leaks are a notoriously difficult problem in lazy functional languages like Haskell [10]: programs do not fully specify their operational behaviour, and compilers are expected to optimise the performance of programs. It is hard to predict the space behaviour of the simplest implementations — they can differ wildly from the programmer's expectations. Compilers cannot predict exactly when an object becomes unwanted and garbage collectors cannot tell exactly which objects are unwanted, owing to the undecidability of garbage [8]. Ignoring the programmer's concern of how to predict space usage, the problem is: how to specify a leak-free standard of space usage for a language; how to decide whether an implementation has a leak; how to decide if an optimisation introduces or eliminates a leak.

1571-0661
doi:10.1016/j.entcs.2002.09.002

Graph rewriting is a natural model of operational behaviour: the runtime memory (heap and stack) with its complex sharing patterns is modelled as a graph; an evaluation strategy is modelled as a collection of graph rewrite rules; a garbage collector may be included in the model, perhaps defined by another rewrite system.

This paper introduces a thesis [2] which provides theoretical tools for analysing the relative space efficiency of different evaluation strategies (evaluators) with garbage collection. This solves the space leak problem by providing a formal framework in which the *space semantics* of a language can be specified as a term-graph evaluator. The space semantics of different implementation techniques can also be modelled as evaluators. Evaluators can then be compared to decide whether they are leakier than each other. The graph framework, leak classifications and leak detection methods we present are intended to be applicable to any language. But their design is influenced by the application used in our examples — that comparison of lazy evaluators.

Section 2 introduces the term-graph rewriting framework with some example evaluators. Section 3 is about leaks: what counts as a leak and why; what causes leaks and how can we classify them. Section 4 discusses why evaluators need to be *accurate* to be used with the leak definition. Comparing different evaluators is based on graph translation, Section 5 shows that translation is another kind of graph rewriting with a space property. Section 6 is about detecting leaks by searching through sequences of evaluator rules. There is not room for definitions and proofs here: see [2] for full details.

2 Three Call-by-Need Evaluators

Before discussing space we introduce the term-graph model by way of three example lazy evaluators. These evaluators do not represent real compilers, rather they serve as abstract definitions of some different lazy evaluation techniques. All three evaluate programs which are represented as graphs of terms built from the expression grammar $X ::= \lambda x.X \mid X\ x \mid x \mid \text{let } x = X \text{ in } X \mid \bot$. This is a fairly standard, simple, core functional language — we do not consider a complete real language for simplicity. Note that function symbols like λ and let are higher-order, each binding one variable whose scope is restricted to their sub-terms. Free variables in a term are arcs to other graph nodes. Note that we follow the standard notation in our presentation, rendering the 'apply-to' function symbol as an infix space and so on. The $\bot$ function is a place-holder used during evaluation.

Lazy Graph Evaluation

The first evaluator lazy in Fig. 1 is a term-graph version of Sestoft's Mk.1 machine for lazy evaluation [11], which closely models the STG-machine used by some Haskell compilers [9]. It is also a simplification of our space semantics for Core Haskell [3].

Evaluators are higher-order term-graph rewriting models of implementations (Ch. 4 of [2]). Briefly, graphs model state as a set of nodes which are mapped to