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 © 2007 Elsevier B.V. Open access under CC BY-NC-ND license.
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

$$\{a \mapsto F\ x\}a, s \longrightarrow \{a \mapsto F, t \mapsto x : s\}a, t \qquad (\mathit{Push})$$
$$\{a \mapsto \lambda y.E, s \mapsto x : t\}a, s \longrightarrow \{a \mapsto E[x/y]\}a, t \qquad (\mathit{Reduce})$$
$$\{a \mapsto x\}a, s \longrightarrow \{a \mapsto \bot, t \mapsto \#a\ s\}x, t \qquad (\mathit{Lookup})$$
$$\{a \mapsto \lambda x.E, y \mapsto \bot, s \mapsto \#y\ t\}a, s \longrightarrow \{y \mapsto \lambda x.E, a \mapsto \lambda x.E\}y, t \qquad (\mathit{Update})$$
$$\{a \mapsto \text{let } y = E \text{ in } X\}a, s \longrightarrow \{a \mapsto X[b/y], b \mapsto E[b/y]\}a, s \qquad (\mathit{Let})$$

$$\begin{array}{ll} & \{a \mapsto \text{let } y = c \text{ in } y\ y, c \mapsto d, d \mapsto \lambda x.x\}a, \epsilon \\ \longrightarrow^{5}_{\text{lazy}} & \{a \mapsto \bot, b \mapsto \bot, c \mapsto \bot, d \mapsto \lambda x.x, s \mapsto \#c\ t, t \mapsto \#b\ u, u \mapsto \#a\ v, v \mapsto b : \epsilon\}d, s \\ \longrightarrow^{4}_{\text{lazy}} & \{a \mapsto b, b \mapsto \lambda x.x, c \mapsto \lambda x.x, d \mapsto \lambda x.x\}a, \epsilon \\ \xrightarrow[gc]{}{}^{2}_{\text{lazy}} & \{a \mapsto \lambda x.x\}a, \epsilon \end{array}$$

Fig. 1. A call-by-need evaluator lazy and its evaluation of a graph.

terms — so our graphs are graphs of terms — followed by a sequence of root nodes. For example, the graph $\{a \mapsto \text{let } y = c \text{ in } y\ y, c \mapsto d, d \mapsto \lambda x.x\}a, \epsilon$ in Fig. 1 has three nodes a, c and d; each contains a term built from The graph root nodes are a and ϵ. In our examples the first root indicates the node to evaluate next and the second is the top of a chain of stack nodes. A missing, or null, arc is written ϵ, so our example graph has an empty stack.

The evaluator lazy has five rules. Each rule has a left *graph pattern* and a right graph pattern, both are rooted. Patterns are incomplete graphs in which terms or sub-terms may be *holes* such as F in *(Push)*; holes are always uppercase. In right patterns, variable-for-variable substitutions may be attached to a hole, see (*Let*) and (*Reduce*) for example.

A rule can be applied to a graph G if there is a *graph context* such that the left pattern in the context equals G. A context replaces the holes of a pattern by terms, replaces the free variables by arcs, renames nodes, renames bound variables and adds extra nodes. The context must not equate distinct nodes (in [2] this is achieved by including disequality constraints in graph patterns. A rewrite step replaces G by the right pattern in the matching context. So the first rewrite step in Fig. 1 matches the left pattern of (*Let*) to the example graph under the context which maps: $\{a \mapsto a, y \mapsto y, E \mapsto c, X \mapsto y\ y, s \mapsto \epsilon\}$ and adds the remaining nodes $\{c \mapsto d, d \mapsto \lambda x.x\}$.

Rules do not delete expression nodes (they do delete stack nodes); we use a garbage collector. The default collector *gc* removes all nodes unreachable from the graph roots. Evaluation on evaluator A with maximal garbage collection (a collection after every rewrite) is written $\xrightarrow[gc]{}{}_A$.

The trace in Fig. 1 evaluates our example graph as follows. (*Let*) allocates a shared expression in new node b. Then (*Push*) begins the evaluation of the application, storing the argument in stack node v meanwhile (the infix : denotes a pushed argument stack-term). Then three (*Lookup*) steps move the control root down the variable chain from b to d, adding an *update marker* (the prefix # denotes an update marker stack-term) to the stack chain at each step. The next three steps

$$\boxed{\text{ind}} = \boxed{\text{lazy}} - \{(Update)\} \cup \{(VUpdate), (IUpdate), (IReduce)\}$$

$$\{a \mapsto \lambda x.E, y \mapsto \bot, s \mapsto \#y\ t\}a, s \longrightarrow \{a \mapsto \lambda x.E, y \mapsto I\ a\}y, t \quad (VUpdate)$$

$$\{a \mapsto I\ b, y \mapsto \bot, s \mapsto \#y\ t\}a, s \longrightarrow \{a \mapsto I\ b, y \mapsto I\ b\}y, t \quad (IUpdate)$$

$$\{a \mapsto I\ c, c \mapsto \lambda y.E, s \mapsto x : t\}a, s \longrightarrow \{a \mapsto E[x/y], c \mapsto \lambda y.E\}a, t \quad (IReduce)$$

$$\begin{aligned}
& \{a \mapsto \text{let } y = c \text{ in } y\ y, c \mapsto d, d \mapsto \lambda x.x\}a, \epsilon \\
\longrightarrow^{5}_{\text{ind}} & \{a \mapsto \bot, b \mapsto \bot, c \mapsto \bot, d \mapsto \lambda x.x, s \mapsto \#c\ t, t \mapsto \#b\ u, u \mapsto \#a\ v, v \mapsto b : \epsilon\}d, s \\
\longrightarrow^{4}_{\text{ind}} & \{a \mapsto b, b \mapsto I\ d, c \mapsto I\ d, d \mapsto \lambda x.x\}a, \epsilon \\
\xrightarrow[gc]{}\ ^{3}_{\text{ind}} & \{a \mapsto \lambda x.x\}a, \epsilon
\end{aligned}$$

Fig. 2. A call-by-need evaluator with value indirections ind and its evaluation of a graph.

(*Update*) nodes c, b and a with the value of d. Then the copy at a is specialised by (*Reduce*). As b has been updated on its first use the second use is much faster, one (*Lookup*) and (*Update*) copy the value of b into a.

Value Indirections

The lazy rule (*Update*) inefficiently makes a new copy of a λ-value each time it is looked up. Real compilers minimise the work of copying by using *indirections* to such values [12]. Our second evaluator, ind in Fig. 2, models this situation.

Instead of updating values by copying, (*VUpdate*) creates an indirection, any further updates just duplicate the indirection by (*IUpdate*) (The expression grammar is extended with the function symbol I). Indirections are only dereferenced in (*IReduce*) reduction steps. In the example, the first five steps are the same as Fig. 1. Then there is one (*VUpdate*), two (*IUpdate*) steps and an (*IReduce*) which make b and c indirections to d (c is collectible by *gc*). Finally, a (*Lookup*), an (*IUpdate*), an (*IReduce*) and a garbage collection produce the result. Value indirections seem to save some space. We look at whether this saving makes lazy leakier than ind. While they never form chains they can extend the lifetime of some graph nodes compared with lazy as a single copied λ-value node in lazy may correspond to an indirection in ind. We look at whether this increase makes ind leakier than lazy.

Update-Marker Indirections

Our third evaluator, #ind in Fig. 3, models a form of *update-marker compaction* [5]. This strategy inserts indirections during lookup steps rather than waiting until the updates. In the example, after (*Let*) and (*Push*), the first lookup of b uses (*Lookup*ϵ) which adds an update marker as normal. The next two lookups use (*Lookup*#) which directly insert indirections, forming an indirection chain from b to d. Next, the (*Update*) and (*Reduce*) at a follow immediately. Finally, the second request for b must traverse the indirection chain using the rules (*Lookup*#), (*Indirect*), (*Indirect*), (*Update*).

Clearly update-marker indirections save some space, not only in stack nodes but also in expression nodes where indirections can be collected before they are even

$$\boxed{\#\text{ind}} = \boxed{\text{lazy}} - \{(Lookup)\} \cup \{(Indirect), (Lookup\epsilon), (Lookup\ :), (Lookup\#)\}$$

$$\{a \mapsto I\ x\}a, s \longrightarrow \{a \mapsto I\ x\}x, s \qquad (Indirect)$$

$$\{a \mapsto x\}a, \epsilon \longrightarrow \{a \mapsto \bot, s \mapsto \#a\ \epsilon\}x, s \qquad (Lookup\epsilon)$$

$$\{a \mapsto x, s \mapsto y : u\}a, s \longrightarrow \{a \mapsto \bot, t \mapsto \#a\ s\}x, t \qquad (Lookup\ :)$$

$$\{a \mapsto x, s \mapsto \#y\ u\}a, s \longrightarrow \{a \mapsto I\ x, s \mapsto \#y\ u\}x, s \qquad (Lookup\#)$$

$$\begin{aligned}
& \{a \mapsto \text{let } y = c \text{ in } y\ y, c \mapsto d, d \mapsto \lambda x.x\}a, \epsilon \\
\longrightarrow^{5}_{\#\text{ind}}\ & \{a \mapsto \bot, b \mapsto I\ c, c \mapsto I\ d, d \mapsto \lambda x.x, s \mapsto \#a\ t, t \mapsto b : \epsilon\}d, s \\
\longrightarrow^{2}_{\#\text{ind}}\ & \{a \mapsto b, b \mapsto I\ c, c \mapsto I\ d, d \mapsto \lambda x.x\}a, \epsilon \\
\xrightarrow[gc]{}^{4}_{\#\text{ind}}\ & \{a \mapsto \lambda x.x\}a, \epsilon
\end{aligned}$$

Fig. 3. A call-by-need evaluator with update-marker indirections $\boxed{\#\text{ind}}$ and its evaluation of a graph.

created in $\boxed{\text{ind}}$. The downside is that $\boxed{\#\text{ind}}$ can make chains of indirections, as demonstrated by the evaluation traces in the figures, so it sometimes uses more space than $\boxed{\#\text{ind}}$. We shall see which of these better/worse relationships count as space leaks.

3 Relative Space Leaks

Programs which use more space than expected are said to have space leaks. In compiled languages with automated memory management, the source of leaks is often unclear. The choice of algorithms and compiler optimisations, the compiler's evaluation strategy and the garbage collector and the language specification may all contribute to the fault. In lazy functional languages leaks are common because it is difficult to predict the operational behaviour of programs and compilers may make changes which distort the relationship between a declared program and the behaviour of its object code.

Typically *space leak* refers to the situation where objects persist in memory after they are known to be unwanted. Failing to recycle a heap object, or losing all pointers to it, is an example caused by poor programming. However, if the language specification demands a garbage collector the compiler is at fault. In both cases the presence of a leak means that some part of the implementation has done worse than the specification demands. We cannot deduce the presence of a leak from the program specification alone or the program alone or the compiler alone or the language specification alone.

In functional languages like Haskell, programs are unable to specify their operational behaviour, and compilers are expected to optimise the performance of programs. This leaves programmers with just the rule that when structures become garbage collectible they will not need any space, so an estimation of space usage can be based on an analysis of *transient* and *persistent* structures — transient structures being those amenable to deforestation [14]. Certain space-saving optimisations, such as *proper tail recursion* [6], are expected as standard, but programmers can hardly be expected to understand the effect of all the optimisations used by

a compiler, especially those which act on non-program structures like indirections. We advocate the inclusion of a space semantics as part of a language specification. This provides an arbitrary standard model against which implementations can be compared to decide whether they leak. Of course, the space semantics may need to be changed so our comparison method should not assume a fixed standard.

The inclusion of more complex space-saving optimisations like *projection shorting* [13] in compilers may give the impression that the aim of compiler design is a space-optimal, garbage-free evaluator and such an evaluator would be leak-free. Unfortunately this is impossible as it is undecidable whether a graph node is semantically garbage [8], and garbage-free evaluation does not prevent inefficient encoding of programs.

As there is no perfect standard we should call leak-free and inefficiency is relative we define leaks relatively: there is an implementation model A and an expectation model B which may be either occasionally or consistently either better or worse. A and B may be expressed in different languages so we introduce a translation relation $\Longmapsto$ between their languages. Leakiness is a ternary relation: A is leakier than B with respect to $\Longmapsto$, $A \rightsquigarrow B$ *wrt* $\Longmapsto$ for short, if there are graphs $G \Longmapsto H$ such that evaluating G on A is less space-efficient than evaluating H on B. As A and B are abstract models, less space-efficient means increased space complexity. The formal definition of $\rightsquigarrow$ is: $\forall k \in \mathbb{N} \cdot \exists G, H \cdot G \Longmapsto H \wedge \mathit{space}\ G\ A > k \times \mathit{space}\ H\ B$.

This definition only considers space: we do not rule out ill-typed or non-terminating programs, though such restrictions could be made by restricting the translation.

Interpretations of Leakiness

In this general form leakiness can be interpreted usefully in many ways: A and B could be two implementations of a function where $\Longmapsto$ is a correspondence between initial states; A and B could have the same evaluation strategy and different garbage collectors with $\Longmapsto$ an identity on the programs of interest; setting B to be empty we can ask whether evaluating a program may require unbounded space. Here the standard interpretation is that A and B are different evaluators, usually with the same garbage collector, and $\Longmapsto$ defines the corresponding initial states. We ask questions like $\boxed{\text{lazy}} \rightsquigarrow \boxed{\#\text{ind}}$ *wrt* $\overset{idX}{\Longmapsto}$ where $\overset{idX}{\Longmapsto}$ is an identity on graphs of expression terms.

Leak Classes

Classifying leaks by their cause or behaviour is useful and interesting. The $\rightsquigarrow$ definition tells us that leaks can be caused purely by inefficiency in A (an *evaluation leak*), or by inefficient translation which A does not rectify or B does not match (a *translation leak*).

Some leaks only occur when A follows a certain sequence of steps, regardless of the parts of the graph not involved in these steps. We call such context-free leaks *active*: any graph which demands the bad sequence of steps, regardless of the context, suffers. $\boxed{\#\text{ind}}$ prevents an *active* leak relative to $\boxed{\text{lazy}}$ where a sequence of

variable lookups causes a chain of update markers to accumulate on the stack (as in Fig. 1 where the first use of the value in node d builds an update-marker chain whereas in Fig. 3 no such chain is built). Improper tail recursion [6] is another example.

The opposite of active is a *passive* leak where the context of an evaluation sequence decides whether a graph exposes the leak. This roughly means that A may expand a structure and if the structure is shared by a term in the context then its expansion causes a leak; whereas expansion of an unshared structure can be garbage collected, preventing any growth. #ind leaks passively relative to lazy because in some contexts it can build chains of indirection nodes (In Fig. 3 the second use of d has to traverse a chain of indirections but the copying strategies in Fig. 1 and Fig. 2 do not). Projection shorting [13] is another example of preventing a passive leak.

The definition of $\rightsquigarrow$ is not the only possible characterisation of space inefficiency. There are other standards which may be considered more appropriate in some circumstances. $\rightsquigarrow$ is just one *space-fault criterion*. We can define other standards by modifying the definition of *space* (discussed in the next section), or by modifying the degree of space worsening we wish to count as a fault. For example, the *unsafe* space-fault criterion is defined (note the range of *space* is $\mathbb{N} \cup \{\infty\}$):
$\mathit{unsafe}\, AB \Longmapsto$ iff $\exists G, H \cdot (G \Longmapsto H) \wedge (\mathit{space}\ G\ A = \infty) \wedge (\mathit{space}\ H\ B \in \mathbb{N})$.

The negation of *unsafe* is the standard described by Appel [1] as space-safety. The leaks discussed in this paper are all *unsafe* faults. For an example of leak which is not *unsafe*, consider an evaluator which converts a graph representing a binary number into unary: this leaks relative to the empty evaluator because there is no constant limiting the expansion, but there is no finite binary number whose unary representation is infinite, so the leak is not *unsafe*. Admittedly, *unsafe* faults only occur when the space inefficiency affects a non-terminating graph, but an *unsafe* fault usually indicates inefficiency in terminating graphs — we can think of this fault as the limit of a series of inefficient terminating graphs (the non-black holing evaluator badbh in [2] leaks only on non-terminating graphs though). The *unsafe* criteria is a proper subset of $\rightsquigarrow$ and its definition demands a worse kind of fault than $\rightsquigarrow$. Considering the position of a space fault in the hierarchy of possible criteria in this way gives a useful indication of its severity.

4 Accurate Evaluators

Graphs offer an obvious definition of *size*: the sum of the *size*s of the terms in their nodes. Combining this and evaluation with maximal garbage collection, we define the *size usage* of a graph G on A as $\max\{\mathit{size}\ G' | G \xrightarrow{gc}{}^*_A G'\}$. This measure ignores the evaluator and collector code size so it is suitable for talking about the dynamic (heap and stack) space usage of a program. If we assume that our graph evaluators will be implemented faithfully, then *space* ought to mean size usage. However.

An Abstract Measure of Space Usage

To measure only the dynamic space usage we should also ignore the initial graph size — the initial graph counts as part of the fixed-space compiled code, which we know there will be enough space for. Giving consideration to our aim of modelling functional language implementations, the *space* measure should be applicable to *environment*-based evaluators. These represent a graph as a heap of *closures* where closures contain a code pointer and some argument pointers. For practical reasons, closures are kept small, and certainly cannot grow beyond some program-determined limit. So when space reasoning is based on abstract versions of environment-based machines space is usually measured by counting closures, e.g. [5].

The term-graph model provides a natural abstract alternative to size usage for this purpose: space usage is defined as the maximum graph *cardinality* during evaluation: $space\ G\ A = \max\{\#G' | G \xrightarrow{gc} {}^*_A G'\}$.

In general, however, *space* is inappropriate: there is nothing to prevent graph evaluators from building terms of unlimited size in a single node. Imagine $\boxed{\text{stack}}$ (defined in [2]), a version of $\boxed{\text{lazy}}$ which stores its stack in a single node: using *space* we could not detect any leak in $\boxed{\text{stack}}$ whose effect is limited to the stack. In fact we would conclude $\boxed{\text{lazy}} \rightsquigarrow \boxed{\text{stack}}$ because there is no limit to the length of the stack chain.

Accurate Evaluators Cannot Hide Leaks

Accurate evaluators overcome this problem. We call an evaluator *accurate* if its space usage multiplied by initial graph size bounds its size usage. If its space usage bounds its size usage it is *uniformly accurate*. Accuracy means that evaluators can use more size on bigger graphs without being called leaky. In some sense this contradicts our claim to have a meaningful model of space and leaks. Clearly uniformly accurate evaluators, or measuring size usage, lead to a valid notion of relative leak; clearly general evaluators and measuring space usage do not; but is accuracy enough to validate space usage? The answer determines whether $\boxed{\text{lazy}}$ is leakier than $\boxed{\text{ind}}$ because they are accurate but not uniformly accurate.

Consider a translation on graphs which encode numbers: let $u(n)$ be a graph containing a single node encoding n in unary and $b(n)$ be a graph containing a single node encoding n in binary; so $\overset{bun}{\Longmapsto} = \{(u(n), b(n)) | n \in \mathbb{N}\}$. Let $\boxdot$ be the empty evaluator. Now, $\boxdot \rightsquigarrow \boxdot\ wrt\ \overset{bun}{\Longmapsto}$, even though $\overset{bun}{\Longmapsto}$ allows an increase in size complexity. We might expect a translation leak but there is none: all graphs in this comparison have one node. This is acceptable as the initial graph in our model is part of the compiled code in reality and both versions use no space dynamically.

Suppose that A computes the unary square of a unary number and B compute the binary square of a binary number, both in a stored in a single node. Now $A \rightsquigarrow B\ wrt\ \overset{bun}{\Longmapsto}$, but A and B are not accurate. To become accurate they must store numbers as chains of digits, so there would be a leak. In this case *accuracy* has the desired effect and exposes the inefficiency of using a unary encoding for dynamic storage.

Now suppose that A creates a graph which contains m copies of the node in

the initial graph. It is accurate and $A \not\backsimeq A$ *wrt* $\overset{bun}{\Longmapsto}$. This situation is very similar to the lazy vs ind problem: lazy can make m copies of values whereas ind will introduce m constant-size indirections. There is no limit to m and the size of copies is proportional to the initial program whereas indirection size is a small constant. Both evaluators are accurate and there is no leak.

We justify this position by separating the initial graph from the input. Space complexity is a function of the *input size*, not the initial graph size, otherwise a sub-linear complexity is impossible. Therefore if the value of n in our discussion is determined by the input the evaluators could not be *accurate*. If the value of n is part of the program then the evaluators can be accurate and preserve or hide a size discrepancy through the constant factor allowed by the initial graph size.

Another consideration is that $\backsimeq$ is a large criterion. Evaluator comparisons that are *unsafe* by the space usage measure are also *unsafe* by the size usage measure, and vice-versa, if the evaluators are accurate. There are also useful intermediate criteria such as *strong inefficiency* (Ch. 7 of [2]) which retain the equivalence of cardinality and size-based faults. Other researchers have made use of similar definitions, for example *weak efficiency* in [7].

To summarise, if evaluators are accurate we can count graph cardinality rather than graph size and still detect all *unsafe* faults. If we are interested in the wider class of $\backsimeq$ faults then accurate evaluators and measuring graph cardinality are still good enough, provided that we are willing to allow an initial program-size-dependent inefficiency.

5 Space-Safe Translation

Translation is a key part of the leakiness equation. We must analyse a translation to detect *translation* leaks (such as the conversion from binary numbers into unary). To prove no leak we use translations which are guaranteed not to decrease space complexity. Garbage collection is another kind of translation which must not increase space complexity.

We define *space relations* for these purposes. A translation $\Longmapsto$ is a space relation if: $\exists k \in \mathbb{N} \cdot \forall G, H \cdot (G \Longmapsto H) \Rightarrow (k \times \#(gc\ G) \geq \#(gc\ H))$. So translating a graph with $\Longmapsto$ preserves cardinality within some $\Longmapsto$-determined constant factor k, assuming the graphs are garbage collected. If we are comparing A and B *wrt* a space relation $\Longmapsto$ then there is no translation leak: any space fault must be caused by the action of A or B. Chs. 9 and 10 of [2] develop a *lockstep simulation* proof technique for showing there is no evaluation leak. The rest of this section considers the properties of space relations and their application as garbage collectors.

Space-Relation Translators

Space-relation translations are defined as another kind of term-graph rewrite system. As with accurate evaluators, restrictions guarantee their space property. It is convenient to define translation by a more abstract kind of rewrite system than evaluators as the order in which parts of a graph are translated and the operational

behaviour of translation are not of direct interest.

Rather than rewriting G into H piecewise, we define *configuration rewrite systems* which rewrite *configurations* encoding G and H together. The configuration of G and H is a quadruple $\langle\{(a,b),(s,t)\},\{\},G',H'\rangle$, where a and s are the roots of G; b and t are the roots of H; G' are the node mappings of G and H' are the node mappings of H. Configurations contain two enumerated relations: the first records node pairs which are reachable from parts of the graphs translated so far and which must be related by the translation, the second records node pairs which have been related by translation. The third and fourth components are the nodes yet to be translated. Each rewrite step removes part of G and the part of H to which it corresponds. If the configuration can be rewritten to an *end* configuration then H is a translation of G. This approach combines well with the simulation proof method.

For example, we define a space relation $\overset{ind}{\Longmapsto}$ between the states of ind and the states of lazy . The third graphs shown in the evaluation traces in Figs. 2 and 1, also shown below, are in $\overset{ind}{\Longmapsto}$. Their garbage collections are also in $\overset{ind}{\Longmapsto}$ because translation ignores unreachable nodes:

$\{a\mapsto b,b\mapsto I\ d,c\mapsto I\ d,d\mapsto\lambda x.x\}a,\epsilon \overset{ind}{\Longmapsto} \{a\mapsto b,b\mapsto\lambda x.x,c\mapsto\lambda x.x,d\mapsto\lambda x.x\}a,\epsilon$

$\{a\mapsto b,b\mapsto I\ d,d\mapsto\lambda x.x\}a,\epsilon \overset{ind}{\Longmapsto} \{a\mapsto b,b\mapsto\lambda x.x\}a,\epsilon$

The definition of $\overset{ind}{\Longmapsto}$ is a collection of configuration-rewrite rules including:

$$\langle\{\},\{\},\{\overline{a}\mapsto I\ b,b\mapsto\lambda x.E\},\{\overline{a'}\mapsto\lambda x'.E',b'\mapsto\lambda x'.E'\}\rangle \quad \rightarrowtriangle$$
$$\langle\{(e,e')\},\{\overline{(a,a')},(b,b'),(y,y')\},\{e\mapsto E[y/x]\},\{e'\mapsto E'[y'/x']\}\rangle \quad (I^n\lambda\lambda^n\lambda)$$

This translates all n indirections to the same λ-term into $n+1$ copies of the λ-term (it uses a shorthand vector notation). A similar rule translates n indirections to a λ-term into n copies for the case when node b' is garbage in lazy , as in the example translations above. So the proof of the example translation above rewrites the following configuration:
$\langle\{(a,a)\},\{\},\{a\mapsto b,b\mapsto I\ d,d\mapsto\lambda x.x\},\{a\mapsto b,b\mapsto\lambda x.x\}\rangle$.

Configuration-rewrite rules are unrooted — there is nothing to say where in a configuration the next redex is to be found. A restricted form guarantees that they define a space relation.

If $G \overset{ind}{\Longmapsto} H$ then we are guaranteed the following: $\#G \leq 2\times\#H$, because an indirection to a λ-term may translate to a single, as in the example above; conversely, $\#H \leq \#G$; $size\ G \leq size\ H + \#H \times S$ where S is the size of an indirection node; $size\ H \leq size\ G \times m$ where m is the maximum node size in G. These quantities are all encoded in the translation $\overset{ind}{\Longmapsto}$ and readily calculable from its definition. They tell us that indirections offer no saving in cardinality and a graph-dependent saving in size.

The proof that ind and lazy are mutually not leakier, ind $\backsim$ lazy , in [2] is based on $\overset{ind}{\Longmapsto}$. Therefore, $G \overset{ind}{\Longmapsto} H \Rightarrow space G\,\text{ind} \leq 2\times space\,H\,\text{lazy} \leq space\,G\,\text{ind}$.

As the evaluators are accurate we can also say that The size usage of H on $\boxed{\text{lazy}}$ is less than the size usage of G on $\boxed{\text{ind}}$ multiplied by *size* H.

Space-Relation Garbage Collectors

Garbage collection can be modelled abstractly as a graph rewrite system.
It should terminate (confluence may also be important). Its main property, however, is that it should decrease the graph size. Abstractly we can say that garbage collectors are terminating rewrite systems which do not increase graph cardinality by more than some collector-dependent constant factor! This is precisely the guarantee of a space relation. The significance is that to investigate space-safe translation is to investigate garbage collection. The thesis exploits this idea to define non-standard collectors and uses them in some leakiness proofs.

Space-relations defined by configuration rewriting ignore unreachable nodes, i.e. $G \Longmapsto H \Rightarrow gc\, G \Longmapsto gc\, H$ where gc is the reachability-based collector. A consequence is that the identity space relation $\overset{id}{\Longmapsto}$ is exactly gc ; that is, $G \overset{id}{\Longmapsto} gc\, G$, and $gc\, G$ is the smallest $\overset{id}{\Longmapsto}$-translation of G. By adding extra translation rules we can define non-standard garbage collectors which produce smaller graphs than gc. For example, we warned that $\boxed{\#\text{ind}}$ can create chains of indirection nodes. The garbage-collector translation *Igc* defined by the identity plus (*DeadInd*) translates any arcs pointing to an indirection node a to arcs which point directly to b.

$$\langle \{\}, \{\}, \{a \mapsto I\ b\}, \{\} \rangle \twoheadrightarrow \langle \{(b,c)\}, \{(a,c)\}, \{\}, \{\} \rangle \quad (\mathit{DeadInd})$$

The smallest *Igc*-translation of a graph is indirection free. For example, the third graph in the evaluation sequence in Fig. 3 has the following *Igc* translation: $\{a \mapsto b, b \mapsto I\ c, c \mapsto I\ d, d \mapsto \lambda x.x\}a, \epsilon \overset{Igc}{\Longmapsto} \{a \mapsto d, d \mapsto \lambda x.x\}a, \epsilon$. So using the indirection collector can reduce the number of evaluation steps in Fig. 3 by two and removes nodes b and c much sooner than the other evaluators. We conjecture that $\boxed{\#\text{ind}}$ with *Igc* is not leakier than $\boxed{\text{ind}}$, while $\boxed{\text{ind}}$ is leakier than $\boxed{\#\text{ind}}$ with *Igc*. Further, $\boxed{\text{lazy}}$ is not leakier than $\boxed{\text{ind}}$ with *Igc* because $\boxed{\text{ind}}$ does not build indirection chains.

6 Finding Leaks in Evaluators

To prove $A \rightsquigarrow B$ we must find graphs to witness the leak. For leaks in a certain class a mechanised search is possible.

First we restrict our attention to *unsafe* faults. We need to find a G with infinite space usage on A, translate it to some H and show that H evaluates on B in bounded space. Within this class, we only wish to consider evaluation leaks — as our translations are constructed to prevent translation leaks it is appropriate to concentrate leak detection on evaluator faults. Finally, to make a systematic search more practicable, we only look at active space faults.

Now the search procedure can be stated: construct every possible infinite evaluation *trace* t of A (a trace is a sequence of evaluator rules); decide whether any graph G whose trace on A is t uses infinite space; if so, translate it to give H and decide whether H runs in bounded space on B.

Searching for a Leak Witness

Rather than attempting to construct every trace, we restrict the search to a subclass where the trace repeats the same rule sequence. We call any graph pattern whose trace has this behaviour an *expanding self-feeding loop*:

$xsfloops\ A = \{G | \exists \mathbb{G}, n \cdot G \xrightarrow{gc} {}^{n}_{A} \mathbb{G}(G), \#\mathbb{G}(G) > \#G\}$

So after n evaluation steps G becomes a larger version of itself: the *graph context* $\mathbb{G}$ adds some extra nodes to G. The new nodes are reachable. Any instance of G will have this n-step evaluation trace on A (by the definition of evaluation). Therefore the behaviour repeats, so G is non-terminating and $space\ G\ A = \infty$.

To test whether a rule sequence $t \in \boxed{\text{A}}^{\,n}$ forms the trace of an expanding self-feeding loop we construct the *super rules* of t (super rules are a graph analogue of *forward closures* [4], used to show termination in term rewriting) then test whether any are self-feeding and expanding.

The super rules of t are a set of graph rewrite rules which form an exact replacement for t. That is, if G rewrites to H using t then there is a super rule r of t such that G rewrites directly to H using r. A trace usually has many super rules because graphs patterns are higher-order and the nodes in successive rules may overlap in different ways. For example, a super rule of the $\boxed{\text{lazy}}$ trace $(Push), (Lookup), (Update), (Reduce), (Let), (Lookup)$ is:

$$\{a \mapsto f\ c, f \mapsto \lambda h.\text{let } i = B \text{ in } i\}a, b \longrightarrow$$
$$\{a \mapsto \bot, l \mapsto B[l/i, c/h], f \mapsto \lambda h.\text{let } i = B \text{ in } i, m \mapsto \#a\ b\}l, m$$

Next we construct the *self-feeding loops* of a super rule by finding contexts in which the rule is guaranteed to self-feed. We create a graph instance of the self-feeding graph pattern by substituting ϵ for all the free variables. Then we test whether the resulting graph expands after the super rule and garbage collector are applied. In our example garbage collection cannot correct the expansion so the graph $\{a \mapsto f\ \epsilon, f \mapsto \lambda h.\text{let } i = f\ \epsilon \text{ in } i\}a, \epsilon$ is a *candidate leak witness*: it is in $xsfloops\ \boxed{\text{lazy}}$.

Next we translate the candidate (to itself in this case), and determine its behaviour on $\boxed{\#\text{ind}}$. After six steps on $\boxed{\#\text{ind}}$ it reaches the following graph: $\{a \mapsto \bot, b \mapsto I\ c, c \mapsto f\ \epsilon, f \mapsto \lambda h.\text{let } i = f\ \epsilon \text{ in } i, s \mapsto \#a\ \epsilon\}c, s$. This is a non-expanding, self-feeding loop graph — a form of non-termination we can detect in the same way that we detect expanding self-feeding loops. This confirms that $\boxed{\text{lazy}} \rightsquigarrow \boxed{\#\text{ind}}$ $wrt\ \overset{idX}{\Longmapsto}$ is an active, unsafe, evaluation leak.

Detecting Other Leaks

As there is a mutual leak, $\boxed{\#\text{ind}} \eqsim \boxed{\text{lazy}}$, we can show $\boxed{\#\text{ind}} \rightsquigarrow \boxed{\text{lazy}}$ $wrt\ \overset{idX}{\Longmapsto}$ by discovering a witness graph and demonstrating its behaviour. However, in this direction the leak is passive so our search method is fruitless. A hand-made witness graph is: $\{a \mapsto g\ g, g \mapsto f, f \mapsto \lambda x.let\ y = x\ in\ y\ y\}a, \epsilon$, its trace on $\boxed{\#\text{ind}}$ is the limit of the irregular sequence
$\Sigma_{i=1}^{n} \langle (Push), (Lookup\ :), (Lookup\#), (Indirect)^{i}, (Update), (Reduce), (Let) \rangle$.

This would be much harder to search for than a regular self-feeding witness.

The generality of the term-graph rewriting model is a benefit to this leak search problem. Both the super-rule and self-feeding loop construction procedures are simple variants on equation solving — unification of graph patterns. No special consideration needs to be given to particular structures, as it might in an abstract-machine framework, and the method applies to any evaluator.

7 Summary

Graph rewriting is a natural way to model size usage. Our higher-order term-graph variant has proved flexible enough to model a range of evaluation strategies. Of course, similar developments would have been possible in an abstract-machine framework such as the $\lambda_{gc}^{\rightarrow\forall}$ framework of [8]. The advantages of the graph approach come from the development of accurate evaluators which are guaranteed to be meaningful abstract models of space usage. This means that if an accurate evaluator is leakier than another in our model, its implementation is guaranteed to have a space fault. The difficulty of graphs is that they are further removed from implementations than conventional abstract machines so it can be difficult to model the space behaviour of some aspects of real implementations, or to argue that the model retains the essential behaviour of the implementation. An example in [2] is the modelling of environment-based machines which do not implement proper *trimming*. Despite this distance, having some framework which guarantees any evaluator the properties of interest seems a useful approach. Graphs are primitive in that they do not distinguishing the characteristics or purpose of different structures. Graph rewriting is a good evaluator framework because its generality encourages the development of widely-applicable proof techniques.

Graph garbage collectors and translators are term-graph rewrite systems too. It is appropriate to distinguish them from our rooted evaluators because we can define their space behaviour more abstractly. Garbage collectors and translators have important common properties. Again, the approach is to decide on the properties of interest and to use restrictions on a general rewriting framework to guarantee them.

To conclude, a summary of the leakiness relations between the evaluators discussed here: $\boxed{\text{lazy}} \leadsto \boxed{\text{ind}} \leadsto \boxed{\#\text{ind}}$. Value indirections are safe but update-marker indirections while often beneficial can be dangerous. Combining $\boxed{\#\text{ind}}$ with the indirection collector should repair its leak relative to $\boxed{\text{ind}}$.

References

[1] A W Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[2] A Bakewell. *An Operational Theory of Relative Space Efficiency*. PhD thesis, Department of Computer Science, University of York, December 2001. http://www.cs.york.ac.uk/ftpdir/reports/YCST-2002-08.ps.Z.

[3] A Bakewell and C Runciman. A space semantics for core Haskell. In *2000 ACM SIGPLAN Haskell Workshop, Montreal, Canada*, volume 41(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001. http://www.elsevier.nl/locate/entcs/volume41.html.

[4] N Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1-2):69–116, 1987.

[5] J Gustavsson. *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Göteborg, 2001.

[6] R Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–80, January 1992.

[7] Y Minamide. A new criterion for safe program transformations. In *HOOTS 2000: 4th International Workshop on Higher Order Operational Techniques in Semantics, Montreal, Canada*, volume 41(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001. http://www.elsevier.nl/locate/entcs/volume41.html.

[8] G Morrisett, M Felleisen, and R Harper. Abstract models of memory management. In *Conf. on Functional Programming Languages and Computer Architecture, La Jolla, California*, pages 66–77. ACM Press, June 1995.

[9] S L Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[10] S L Peyton Jones and J Hughes (editors). *Haskell 98: A Non-Strict, Purely Functional Language*, February 1999.

[11] P Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[12] D Turner. A new implementation technique for applicative languages. *Software — Practice & Experience*, 9:31–49, 1979.

[13] P Wadler. Fixing some space leaks with a garbage collector. *Software — Practice & Experience*, 17(9):595–608, 1987.

[14] P Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.