A Model for Comparing the Space Usage of Lazy Evaluators

Adam Bakewell and Colin Runciman
Department of Computer Science, University of York, UK.
{ajb,colin}@cs.york.ac.uk

ABSTRACT

Identifying the source of space faults in functional programs is hard. The problem is compounded as space usage can vary enormously from one implementation to another. We use a term-graph rewriting model to describe evaluators with explicit space usage. Given descriptions for two evaluators E1 and E2, if E1 never has asymptotically worse space usage than E2, we can use a bisimulation-like proof method to prove it. Conversely, if E1 is leakier than E2, we characterise a class of computations that expose the difference between them.

Keywords Space Leak, Heap Usage, Operational Semantics, Functional Programming, Term-graph Rewriting, Bisimulation, Garbage.

1. INTRODUCTION

Functional programmers often complain that their programs have *space leaks*, usually meaning that the program is taking much more memory than expected, perhaps running out of space when run with only a modest amount of input. Worse still, the space usage of functional programs is rarely consistent from one implementation to another, making it difficult to tell whether the implementation or the program is faulty. There is no agreed evaluation model for Haskell, but assuming there were, we would like to be sure that our implementation conforms to that model and that its space usage is never worse than the standard model. Then we can be sure that any optimisations in its evaluation strategy do not accidentally increase space usage.

Example 1.

```
f xs = last xs == head xs
tf b = b : tf $! (not b)
```

The Haskell function `f` compares the first and last elements of a list. We test it on a long list of a million alternating boolean values, running: `f (take 1000000 (tf True))` (we have taken care not to cause a space leak in the definition of `tf` by using strict function application!). Evaluation soon runs out of memory; a program using only two elements of a list is taking a lot of space.

The solution is to swap the arguments to `==`, redefining `f` as: `f xs = head xs == last xs`, now the test runs in constant space. The implementation chooses to evaluate the arguments to `==` left to right, so when we find the last element first, the rest of the list cannot be garbage collected because the unevaluated `head xs` is holding onto it. This fault has another known cure: the garbage collector can evaluate `(head xs)` when the first cons cell of `xs` has been produced [25]. □

Space leaks can often be fixed by improving the evaluator (e.g. [24]). In this paper we define one evaluator as *leakier* than another if it has asymptotically worse space usage for some programs. We develop a proof technique, a *proportional lockstep bisimulation*, to show that one evaluator is not leakier than another. This technique can be used to compare evaluators that always give the same time complexity, so it is ideal for investigating the effect that modifying an evaluator has on space usage. If two evaluators are mutually not leakier we call them *space-equivalent*. To prove an evaluator definitely is leakier than another, we give equations for characterising programs that have asymptotically different space usage on the two evaluators. This helps us understand *why* there is a space leak.

We use a term-graph rewriting formalism for operational semantics to make aspects of reduction relevant to space analysis explicit. Our examples include comparisons of Core Haskell evaluators. The techniques are more widely applicable. We use some simpler graph evaluators to introduce the ideas.

Section 2 introduces a term-graph formalism, evaluators and functions for describing space use. Section 3 shows how to ensure that an equivalence relation between two graph languages can be written so it also guarantees a bound on the sizes of the graphs it relates. Section 4 defines a *leakier* evaluator and develops a bisimulation-like proof technique for showing an evaluator is not leakier than another or that two evaluators are space-equivalent. This is applied to variants of a Core Haskell evaluator. Section 5 is about showing an evaluator is leakier than another, developing equations to characterise space leaks. Section 6 reviews examples of space leaks and other related work. Section 7 concludes. Proofs are sketched or omitted to save space.

$ListBool = \{LBTerm\}$		*ListBool* language
$LBTerm ::=$	$F\ x$	list, False:list at x
	$T\ x$	list, True:list at x
	$\Box$	end of list

$ListNot = \{LNTerm\}$		*ListNot* language
$LNTerm ::=$	$x : y$	list, elem at x:list at y
	$[\,]$	end of list
	0	False value
	$\neg\ x$	Not value at x

Figure 1: *ListBool* and *ListNot*: term languages for writing graphs representing list-of-boolean data structures. *ListNot* is more general, with separate list and boolean constructors.

2. A TERM-GRAPH MODEL FOR OPERATIONAL SEMANTICS

Operational semantics are often defined as relations on structures which model heaps, stacks, environments and so on (e.g. [23]). To avoid becoming restricted to any particular abstract machine design — and any particular interpretation of state size in that design — we use *term graphs* [7] to model evaluator state. Terms can be used for operational semantics — with address annotations to indicate sharing [21] — but graphs naturally model sharing and their size directly describes space usage. Ariola and Arvind [3] also use term graphs in their study of languages with sharing. Other benefits of this general framework will become apparent: garbage collection and relationships between semantics can be defined in a language-independent way. Section 2.1 defines term graphs. Sections 2.2 and 2.3 define garbage collection and evaluation. Section 2.4 discusses measuring space and includes a Core Haskell evaluator example.

2.1 Term Graphs

We define term graphs as distributed terms. Each node contains a term; its free variables are the graph arcs — the *addresses* of other graph nodes.

Definition 1. (**Grammar, term**)
A Term Grammar $\mathcal{G}$ is a set of syntactic *categories*, each of which defines a set of *function symbols*. The function symbol definition $F\ x_1 \cdots x_b . A_1 \cdots A_n$ tells us that F binds b variables, and takes n *arguments*. Each A_i is a category name; if it is upper-case the ith argument of F must be a term of category A_i, if it is lower-case the ith argument must be a variable of category A_i. Each x_i specifies the category of the ith bound variable.
$Term(\mathcal{G})$ is the set of terms constructed from $\mathcal{G}$.
For convenience we often write terms in a more familiar notation, e.g. $(h : t)$ for $(Cons\ h\ t)$. □

Definition 2. (**Term graph, rooted graph**)
A term graph $G \in Graph(\mathcal{G})$ is a mapping from variables to terms in $Term(\mathcal{G})$. That is, G is a set of *nodes* which have an *address* and contain a term.
A rooted graph G_V has an ordered set of root variables, $V = \langle v_1, \ldots, v_n \rangle$. It may be written as $G_{v_1,\ldots,v_n}$. □

Example 2. Figure 1 gives the node term grammars for two graph languages. They are not typed but we assume that only graphs representing $[Bool]$ data structures are written. Figure 2 depicts *ListBool* and *ListNot* graphs tf_b and tf_n representing the infinite list `(tf True)` of Example 1. Equations (1) and (2) are precise definitions of tf_b and tf_n.

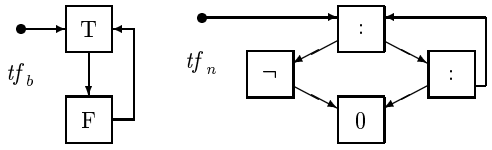

Figure 2: The *ListBool* graph tf_b and a *ListNot* graph tf_n, both representing $tf = True : False : tf$. Each has one root which addresses the value of tf. The *ListNot* representation shares the 0 node, inverting it to make the *True* element.

$$tf_b = \{a \mapsto T\ b, b \mapsto F\ a\}_a \quad (1)$$
$$tf_n = \{w \mapsto x : y, x \mapsto \neg\ z, y \mapsto z : w, z \mapsto 0\}_w \quad (2)$$

Free variables in node terms correspond to arcs in the diagram. The graph roots are access points for an evaluator or a garbage collector. □

Definition 3. (**Free variables, substitution**)
We extend the usual term definitions for free variables (FV) and substitution to term graphs as follows.
$FV(\{x_i \mapsto N_i\}_{i=1}^n) = (\bigcup\{FV(N_i)\}_{i=1}^n) \setminus \{x_i\}_{i=1}^n$
$G\theta = (\{x_i \mapsto N_i\}_{i=1}^n)\theta = \{x_i \mapsto N_i(\theta|_{FV(G)})\}_{i=1}^n$ □

Definition 4. (**Well-formed graph**)
G_V is *well-formed* iff $V \subseteq dom(G)$ and $FV(G) = \emptyset$. □

Well-formed graphs have no unconnected arcs. This ensures that evaluators operate on a closed state which includes all the information needed during reduction. Therefore we can measuring space usage as graph size. As tf_b and tf_n have no free variables, they are well-formed graphs. In our notation, juxtaposition indicates disjoint graph union: $GH \equiv G \cup H, dom(G) \cap dom(H) = \emptyset$. $G \equiv H$ means G and H are identical whereas $G = H$ denotes equality by definition or equality up to bound variable renaming.

2.2 Garbage Collection and Graph Size

Only nodes on a path from the roots of a graph can ever be used by an evaluator as defined in Section 2.3. A separate garbage collector gc (Definition 5) removes any unreachable nodes. It applies to any rooted graph. Unreachability is only an approximation to garbage; which nodes are never needed again is undecidable [16].

Definition 5. (**Reachable nodes,garbage collection**)
The set $reach(G, X)$ contains the node addresses reachable in G starting from those in X. The garbage collector gc removes all nodes unreachable from the roots of a graph.
$reach(G, X) = fix\ (\lambda V.X \cup reach(G, FV(G|_V)))$
$gc(G_V) = (G|_{reach(G,V)})_V$ □

Definition 6 gives the *size* of terms and graphs in the obvious way. The space needed by G_V is $size(gc(G_V))$.

Definition 6. (**Term size, graph size**)
$size(x) = 1$
$size(F\ x_1 \cdots x_b . N_1 \cdots N_a) = 1 + b + \sum_{i=1}^a size(N_i)$
$size(\{x_i = N_i\}_{i=1}^n) = \sum_{i=1}^n size(N_i)$
$size(G_V) = size(G)$ □

2.3 Graph Operational Semantics

We use structural operational semantics [18] for describing graph evaluators. A higher-level natural semantics can hide details relevant to space analysis. For example, the stack is not explicit in Launchbury's semantics [14]. Our definition guarantees the time and space needed at each evaluation step are bounded to make the semantics realistic.

Graph Patterns

Evaluators are defined as sets of rules, each rule is two *patterns*. A graph matching the left pattern evaluates to a graph matching the right pattern. Patterns are *graph contexts*.

Definition 7. (**Pattern**)
A pattern $P \in Pattern(\mathcal{G})$ is built from the same grammar as a graph in $Term(\mathcal{G})$. It may also have special free variables, written in uppercase, which are interpreted as *labelled holes*. Holes in a pattern match arbitrary terms; variables only match variables. Holes in the right pattern of a rule may have a substitution attached to replace variables that were bound around them in the left pattern. □

Example 3. $L = \{a \mapsto (\lambda y.N)\ x\}_a$ is left pattern, x, y are variables and N is a hole. $R = \{a \mapsto N[x/y]\}_a$ is right pattern and hole N has a substitution to replace y with a x, a variable which is still in scope. □

Definition 8. (**Pattern matching**)
A function *match* takes two mappings: $\psi : Var \to Var$ and $\phi : Hole \to Term(\mathcal{G})$ and a pattern P. It produces a graph by applying ψ to all the variables in P and ϕ to all holes in P. If a hole has a substitution θ then θ is mapped to a substitution by ψ and applied to the filled hole.
$match\ \psi\ \phi\ x = \psi(x)$
$match\ \psi\ \phi\ (H[x_i/y_i]_{i=1}^n) = [\psi(x_i)/\psi(y_i)]_{i=1}^n(\phi(H))$
$match\ \psi\ \phi\ (F\ x_1 \cdots x_b.P_1 \cdots P_a) = F\ \psi(x_1) \cdots \psi(x_b).$
$(match\ \psi\ \phi\ P_1) \cdots (match\ \psi\ \phi\ P_a)$
$match\ \psi\ \phi\ \{a_i \mapsto P_i\}_{i=1}^n = \{\psi(a_i) \mapsto match\ \psi\ \phi\ P_i\}_{i=1}^n$
$match\ \psi\ \phi\ G_{v_1,\ldots,v_n} = (match\ \psi\ \phi\ G)_{\psi(v_1),\ldots,\psi(v_n)}$ □

Example 4. Let $\psi = [a/x, x/y]$ and $\phi = [(f\ x)/N]$. Using L and R from Example 3: $match\ \psi\ \phi\ L = \{a \mapsto \lambda x.f\ x\}_a$ and $match\ \psi\ \phi\ R = \{a \mapsto f\ a\}_a$ □

Matching is the basis of our evaluation mechanism. Example 4 demonstrates the delayed substitution on holes which helps to preserve well-formedness. FV (Definition 3) extends to contexts by the rule $FV\ (H\theta) = \emptyset$ and *size* (Definition 6) by the rule: $size(H\theta) = 0$.

Evaluators

Definition 9. (**Order-k evaluator**)
An order-k evaluator $\boxed{\text{A}}$ is a finite set of rules $(L_V \longrightarrow R_W)$ where L_V and R_W are graph contexts with root sets V and W. Holes in R may have substitutions attached to them, holes in L may not.
For preservation of well-formedness we require:
1. $\forall \psi, phi\ FV(match\ \psi\ \phi\ R) \subseteq FV(match\ \psi\ \phi\ L)$
2. $\#W = \#V$, $W \subseteq dom(R) \cup FV(L)$.
If $dom(L) \setminus dom(R) \neq \emptyset$ then we are obliged to prove these removed nodes are unreachable.
To bound the work at each step we require:
3. $size(L) \leq k, size(R) \leq k$.

$$
\begin{array}{ll}
tf_b = & \{a \mapsto T\ b, b \mapsto F\ a\}_a \\
\longrightarrow_b & \{a \mapsto \underline{F}\ b, b \mapsto F\ a\}_b \\
\longrightarrow_b & \{a \mapsto F\ b, b \mapsto \underline{T}\ a\}_a \\
\longrightarrow_b & \{a \mapsto \underline{T}\ b, b \mapsto T\ a\}_b \\
\longrightarrow_b & tf_b
\end{array}
$$

$$
\begin{array}{ll}
tf_n = & \{w \mapsto x : y, x \mapsto \neg\ z, y \mapsto z : w, z \mapsto 0\}_w \\
\longrightarrow_n & \{w \mapsto \underline{z} : y, y \mapsto z : w, z \mapsto 0\}_y \\
\longrightarrow_n & \{w \mapsto z : y, y \mapsto \underline{v} : w, z \mapsto 0, \underline{v \mapsto \neg\ z}\}_w \\
\longrightarrow_n & \{w \mapsto \underline{u} : y, y \mapsto v : w, z \mapsto 0, v \mapsto \neg\ z, \underline{u \mapsto \neg\ z}\}_y \\
\longrightarrow_n & \{w \mapsto u : y, y \mapsto \underline{z} : w, z \mapsto 0, u \mapsto \neg\ z\}_w
\end{array}
$$

Figure 3: Evaluation traces for tf_b on $\boxed{\text{b}}$ and tf_n on $\boxed{\text{n}}$. Parts of the graph that change are underlined. The garbage collector removes x after the first evaluation step of tf_n and v after the fourth step, when it becomes a renaming of tf_n.

---

4. $rng(L)$ is linear in all its variables and holes.
5. $L_V = gc(L_V)$. □

Definition 10. (**Evaluation, redex**)
The evaluation relation $\longrightarrow_A$ defined by $\boxed{\text{A}}$ is:

$$\longrightarrow_A = \left\{ (GG', gc(GG'')) \;\middle|\; \begin{array}{l} \exists \psi, \phi, (L \longrightarrow R) \in \boxed{\text{A}}\ s.t. \\ G' = match\ \psi\ \phi\ L \\ G'' = match \psi\ \phi\ R \end{array} \right\}$$

Any graph $G \in dom(\longrightarrow_A)$ is an $\boxed{\text{A}}$-redex. □

Example 5.

$$\boxed{\text{b}} = \left\{ \begin{array}{l} \{x \mapsto F\ y\}_x \longrightarrow \{x \mapsto T\ y\}_y \\ \{x \mapsto T\ y\}_x \longrightarrow \{x \mapsto F\ y\}_y \end{array} \right\} \tag{3}$$

$\boxed{\text{b}}$ is an evaluator for *ListBool* graphs. It inverts the element at the root in-place then moves the root x onto the next element at y. Evaluation terminates if no rule applies, so there is no rule for the list-end case.

$$\boxed{\text{n}} = \left\{ \begin{array}{l} \left\{ \begin{array}{l} x \mapsto y : z, \\ y \mapsto 0 \end{array} \right\}_x \longrightarrow \left\{ \begin{array}{l} x \mapsto y' : z, \\ y' \mapsto \neg y, y \mapsto 0 \end{array} \right\}_z \\ \left\{ \begin{array}{l} x \mapsto y' : z, \\ y' \mapsto \neg y, y \mapsto 0 \end{array} \right\}_x \longrightarrow \left\{ \begin{array}{l} x \mapsto y : z, \\ y' \mapsto \neg y, y \mapsto 0 \end{array} \right\}_z \end{array} \right\} \tag{4}$$

$\boxed{\text{n}}$ is a *ListNot* analogue of $\boxed{\text{b}}$. The first rule inverts false elements by inserting a negation node. The second rule inverts true elements by missing out negation node y'; it cannot remove y' because there may be other references to it from elsewhere in the graph. Figure 3 shows the evaluation of tf_b (1) using $\boxed{\text{b}}$ and tf_n (2) using $\boxed{\text{n}}$. □

Because our rules can match, copy and substitute into arbitrary term via holes, we need a way to guarantee each step only does a bounded amount of work. A *space-valid* evaluator (Definition 11) has this property and from now on we assume all evaluators are space-valid.

Definition 11. (**Space-valid evaluator**)
An order-k evaluator $\boxed{\text{A}}$ is *space-valid* if there is a maximum node size for every evaluation. That size depends on the grammar of the graph language:
1. Program-independent space-valid:
$G \longrightarrow_A G' \Rightarrow \max\{size(N) \mid N \in G'\} \leq k$

When there is some fixed bound k on the size of any possible term, an evaluator will be order-k and space-valid.
2. Program-dependent space-valid:
$G \longrightarrow_A G' \Rightarrow$
$\max\{size(N) \mid N \in G'\} \leq \max\{k, \max\{size(N) \mid N \in G\}\}$
With a recursive grammar, if evaluation cannot increase the maximum node size or allocate a node bigger than k it is space-valid. □

Evaluators b and n are both Program-independent space-valid: Their graph grammars are non-recursive, for b the maximum term size is 2, for n it is 3.
Some useful space properties are explicit in this term-graph formalism. Allocation, explicit deallocation and updating are all directly observable in the rules. Proposition 1 below confirms that an evaluation step corresponds to a unit of computation. Proposition 2 is another property we need: each step must only allocate a bounded amount of space — a useful observation is that time use bounds space use. The number of unreachable nodes generated at each step is unknown; this makes reasoning about space usage difficult in functional languages.

PROPOSITION 1. *When evaluating graph G on space-valid evaluator $A = \{L_i \longrightarrow R_i\}_{i=1}^{r}$, each step is $O(max\{size(N) \mid N \in G\} \times \Sigma_{i=1}^{r} \#(L_i R_i))$.*
Proof: The time to determine if a rule applies to a graph and the match information needed depends on the left pattern size by Definition 9.3–5. To apply the match to a right pattern depends on the pattern size and the maximum node size, which are bounded by Definition 9.3 and Definition 11. □

PROPOSITION 2. *The number of nodes allocated at each step is bounded, but the number of nodes that become garbage at each step is unbounded.*
Proof: Definition 9.3 and space-validity guarantee allocation is bounded. The free variable property (Definition 9.1) allows for an unlimited reduction in reachable nodes. □

2.4 Measuring Space Usage

Definition 12 gives the *space* needed to evaluate a graph. It uses the graph cardinality rather than *size*. With this simplification, the space allocated by each step is directly observable from the operational semantics rules.

Definition 12. (**Evaluation space**)
$space(G, A) = \max\{\#G' | G \longrightarrow_A^* G'\}$ □

space is only useful if it is within a constant factor of the maximum graph *size* during evaluation. A space-valid semantics has this property by Proposition 3.

PROPOSITION 3.
If A is Program-independent space-valid then:
$\exists k, \forall G, \max\{size(G') \mid G \longrightarrow_A^* G'\} \leq k \times space(G, A)$
If A is Program-dependent space-valid then:
$\forall G, \exists k, \max\{size(G') \mid G \longrightarrow_A^* G'\} \leq k \times space(G, A)$
Proof: Induction on length of evaluation sequence. □

As *space* is only accurate to within a constant factor of the *size* usage, it makes sense to talk about asymptotic space usage. Then we can investigate space behaviour as the number of inputs read goes to the limit. First we introduce a simple IO model by way of a notation for effects.

Definition 13. (**Effect**)
An atomic effect is *get c* or *put c* where c is a function symbol with no bound variables or arguments. An effect is a sequence of atomic effects. A rules is decorated an effect which is executed in order when its left pattern matches a graph. $G \xrightarrow{E} G'$ means the evaluation step changing G to G' has effect E. $G \longrightarrow G'$ is shorthand for $G \xrightarrow{\langle\rangle} G'$. □

Example 6. Figure 4 defines a Core Haskell grammar. Figure 5 defines h, the operational semantics of Core. This formulation is a graph version of Sestoft's mk.1 machine for lazy evaluation [23] (its correctness follows from a simple inductive proof). The use of reduction context and value categories to reduce the number of rules is taken from Gustavsson and Sands [10]. The machine has its origins in Launchbury's natural semantics for laziness [14]. We have omitted the numeric extension used in those papers (integers and arithmetic operators) because the choice of *Int* or *Integer* type affects space usage in Haskell — unbounded integers do not use bounded space. h includes a simple monadic IO mechanism based on Gordon's definition [8]. We assume that IO is restricted to characters and that there are constructors *IO* and unit () to support the translation of the monadic syntax into Core.
h is program-dependent space-valid: the maximum node size during evaluation never exceeds the initial node size. A Haskell-like syntax rather than terms is used; an equivalent term formulation is easily written by treating *case* and *let* as standing for families of terms. □

Now we define asymptotic space usage as a function of the number of inputs read. $Fspace(A)$ (Definition 14) includes all graphs with space usage bounded by F on A. For example, $(\lambda x.x^2)space(A)$ includes any graph whose *space* is at worst quadratic in the number of inputs.

Definition 14. (**Asymptotic space classes**)

$$Fspace(A) = \left\{ G \;\middle|\; \begin{array}{l} \exists k, G \xrightarrow{E}_A^* G' \Rightarrow \\ \#G' \leq k \times F(\#\langle c | (get\ c) \in E\rangle + 1) \end{array} \right\}$$

□

3. SPACE-EQUIVALENCE OF GRAPHS

This section shows how a proof of equivalence of two graphs can be made to say something about their relative sizes. Section 4 uses this as the basis for showing an evaluator does not have a space leak with respect to another evaluator.
An equivalence $\sim$ between two graph languages can be defined inductively by a set of rules. Typically each rule has the form of *(Rule)*, relating the terms at addresses a and b assuming the terms they refer to are also related. We use these rules to construct a proof tree, beginning at the root nodes.

$$(Rule)\frac{G \sim^{(a_1,b_1)} H \cdots G \sim^{(a_n,b_n)} H}{G \sim^{(a,b)} H}$$

A graph might contain cycles, resulting in an infinite proof tree. Recording addresses as they are related solves this problem: instead of repeatedly re-relating the same subgraphs we can apply the *(Leaf)* axiom. To support this, the premises of an equivalence rule apply a substitution ϕ which identifies addresses related at that step, typically $\phi = [b/a]$.

Expression Nodes	Core graphs contain X and S nodes	Reduction Contexts	
$X ::= \mathbb{R}[X]$	reduction context, X fills hole	$\mathbb{R} ::= \Box\ x$	apply to variable
$\mid \mathbb{V}$	value	$\mid$ *case* $\Box$ *of*	case expression
$\mid x$	variable	$\{c^{a_i,i,n}\ x_1 \cdots x_{a_i} \to X_i\}_{i=1}^{n}$	
$\mid$ *let* $\{x_i = X_i\}_{i=1}^{b}$ *in* X	let expression	$\mid$ *putChar* $\Box$	output a character
$\mid$ *getChar*	input a character	Values	
$\mid \bot$	undefined term	$\mathbb{V} ::= \lambda x.X$	lambda abstraction
$\mid$ *Red* $\mathbb{R}[\mathbb{V}]$	reduce $\mathbb{R}[\mathbb{V}]$	$\mid c^{a,r,n}\ x_1 \cdots x_a$	saturated constructor with arity a, rank r of n
Stack Nodes		Initial Graph: $\{a \mapsto N\}_{a,\epsilon}$ typically $N =$ *let* $\{\cdots\}$ *in main*	
$S ::= \#x\ s$	update x marker:stack at s	Terminal Graph: $G\{a \mapsto \mathbb{V}\}_{a,\epsilon}$ value at a, null stack	
$\mid \mathbb{R} : s$	pushed context:stack at s		

Figure 4: Core Haskell grammar. Expression nodes form a graph and stack nodes form a chain whose elements point to the expression graph. Initially, the Haskell program is represented by a single node pointed to by the control root a; the stack root is null. *Red* and $\bot$ are not part of Haskell, they are generated during evaluation.

$$
\begin{array}{rcll}
\{a \mapsto \mathit{getChar}\}_{a,s} & \overset{\langle \mathit{get}\ c\rangle}{\longrightarrow} & \{a \mapsto \mathit{IO}\ b, b \mapsto c\}_{a,s} & (\mathit{Read}) \\
\{a \mapsto \mathit{let}\ \{x_i = N_i\}_{i=1}^{b}\ \mathit{in}\ N\}_{a,s} & \longrightarrow & \{a_i \mapsto N_i[a_i/x_i]_{i=1}^{b}\}_{i=1}^{b}\{a \mapsto N[a_i/x_i]_{i=1}^{b}\} & (\mathit{Let}) \\
\{a \mapsto x, x \mapsto N\}_{a,s} & \longrightarrow & \{a \mapsto N, x \mapsto \bot, t \mapsto \#x\ s\}_{a,t} & (\mathit{Lookup}) \\
\{a \mapsto \mathbb{V}, x \mapsto \bot, s \mapsto \#x\ t\}_{a,s} & \longrightarrow & \{a \mapsto \mathbb{V}, x \mapsto \mathbb{V}\}_{a,t} & (\mathit{Update}) \\
\{a \mapsto \mathbb{R}[N]\}_{a,s} & \longrightarrow & \{a \mapsto N, t \mapsto \mathbb{R} : s\}_{a,t} & (\mathit{Push}) \\
\{a \mapsto \mathbb{V}, s \mapsto \mathbb{R} : t\}_{a,s} & \longrightarrow & \{a \mapsto \mathit{Red}\ \mathbb{R}[\mathbb{V}]\}_{a,t} & (\mathit{Reduce}) \\
\{a \mapsto \mathit{Red}\ ((\lambda x.N)\ y)\}_{a,s} & \longrightarrow & \{a \mapsto N[y/x]\}_{a,s} & (\mathit{Apply}) \\
\{a \mapsto \mathit{Red}\ (\mathit{case}\ c\ y_1 \cdots y_a\ \mathit{of}\ \{c\ x_1 \cdots x_a \to N\} \cup \mathit{AS})\}_{a,s} & \longrightarrow & \{a \mapsto N[y_i/x_i]_{i=1}^{a}\}_{a,s} & (\mathit{Case}) \\
\{a \mapsto \mathit{Red}\ (\mathit{putChar}\ c)\}_{a,s} & \overset{\langle \mathit{put}\ c\rangle}{\longrightarrow} & \{a \mapsto \mathit{IO}\ b, b \mapsto ()\}_{a,s} & (\mathit{Write})
\end{array}
$$

Figure 5: Core Haskell Evaluator h Definition. *(Read)* gets character c from the input file; *(Let)* adds a set of new nodes to the graph; *(Lookup)* begins finding the value of x, it creates an update marker stack node for x, eventually *(Update)* copies this value into x; *(Push)* puts a reduction context into a stack node and then reduces the expression in its hole, *(Reduce)* combines the value of that expression with the context, forming a structure reduced by *(Apply)*,*(Case)* or *(Write)*.

$$(\mathit{Leaf})\frac{}{G \sim^{(a,a)} H} \quad (\mathit{Rule}')\frac{(G \sim^{(a_1,b_1)} H \cdots G \sim^{(a_n,b_n)} H)\phi}{GG' \sim^{(a,b)} HH'}$$

Expanding the rule template to (Rule'): G and H are graph variables and G' and H' are graph patterns (they are not rooted or even connected) where G' includes a node at address a and H' a node at address b. The premise removes G' and H' as a and b are not needed again.

Because the proof tree is finite, if all nodes of the related graphs are removed at some node of the proof then the sizes of the related graphs are related by some function. But we want to guarantee that the size of G bounds the size of H, $k \times \#G \geq \#H$, whenever $G \sim H$. This follows if the proof relates every node in G to no more than k nodes in H. We may also want H to bound G. We achieve this bound by arranging that the proof tree never branches. Once the nodes at a and b are related, they never relate to any other nodes elsewhere in the proof tree.

$$(\mathit{Rule}'')\frac{(G \sim^{\theta\sigma} H'')\phi}{GG' \sim^{\theta\{(a,b)\}} HH'}$$

Now the equivalence rule template is (Rule''). The relation θ carries the addresses to be related. Instead of backtracking at the leaves of the proof tree and picking a new branch, we just keep taking new addresses to relate from θ and there are no leaves. As before, a and b are in the patterns G' and H'. New addresses to relate are added to σ in the premise, so $\sigma = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ if we think of (Rule'') as a new version of (Rule'). The substitution ϕ applies to all parts of the premise: G,H and $\theta\sigma$. Axioms in a proof system in the style of *(Rule)* are now inference rules with an empty σ. Removing some nodes at each step guarantees the space bound.

Definition 15 describes how to define a *proportional-space* relation $\gtrsim$ in this style which guarantees that if $G \gtrsim H$ then $\mathit{size}(G) \geq k \times \mathit{size}(H)$. By symmetry, a *space-equivalence* guarantees that the size of each related graph bounds the other.

Definition 15. (**Proportional-space relation**)
A proportional-space relation $\gtrsim \subseteq (\mathit{Graph}(\mathcal{L}_1) \times \mathit{Graph}(\mathcal{L}_2))$ is defined by a set of rules of the form: $\dfrac{(GG'' \gtrsim^{\theta\sigma} HH'')\phi}{GG' \gtrsim^{\theta\{(a,b)\}} HH'}$

1. G and H are variables that match arbitrary graphs.
2. G' and H' are graph patterns of bounded size that match a sub-graph.
3. G'' and H'' are graph patterns, $\mathit{size}(G') > \mathit{size}(G'')$.
4. Once related, at least one node from G' must never be related again: $a \in \mathit{domG}', b \in \mathit{domH}', a \notin \mathit{domG}''$.
5. Substitution ϕ identifies any addresses in G' and H' which are related at this step (typically $\phi = [b/a]$).
6. Relation σ contains new addresses to relate: $\sigma : (FV(G') \cup \mathit{dom}(G'')) \to (FV(H') \cup \mathit{dom}(H''))$.
7. To guarantee proportional space: $\mathit{reach}(HH'', \mathit{rng}(\theta\sigma)) \setminus \mathit{dom}(H'') = \mathit{reach}(HH', \mathit{rng}(\theta\{(a,b)\})) \setminus \mathit{dom}(H')$. □

$$(True)\frac{(G \approx^{\theta\{(x,y)\}} H\{c \mapsto \neg d, d \mapsto 0\})[b/a]}{G\{a \mapsto Tx\} \approx^{\theta\{(a,b)\}} H\{b \mapsto c : y, c \mapsto \neg d, d \mapsto 0\}}$$

$$(Nil)\frac{(G \approx^{\theta} H)[b/a]}{G\{a \mapsto \square\} \approx^{\theta\{(a,b)\}} H\{b \mapsto [\,]\}} \quad (False)\frac{(G \approx^{\theta\{(x,y)\}} H\{c \mapsto 0\})[b/a]}{G\{a \mapsto Fx\} \approx^{\theta\{(a,b)\}} H\{b \mapsto c : y, c \mapsto 0\}}$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$(End)\frac{}{\emptyset \approx^{\{(w,w)\}} \{x \mapsto \neg z, z \mapsto 0\}}$$

$$(False)\frac{}{\{b \mapsto F\ w\} \approx^{\{(b,y)\}} \{x \mapsto \neg z, y \mapsto z : w, z \mapsto 0\}}$$

$$(True)\frac{}{\{a \mapsto T\ b, b \mapsto F\ a\} \approx^{\{(a,w)\}} \{w \mapsto x : y, x \mapsto \neg z, y \mapsto z : w, z \mapsto 0\}}$$

$$(Start)\frac{}{tf_b = \{a \mapsto T\ b, b \mapsto F\ a\}_a \approx \{w \mapsto x : y, x \mapsto \neg z, y \mapsto z : w, z \mapsto 0\}_w = tf_n}$$

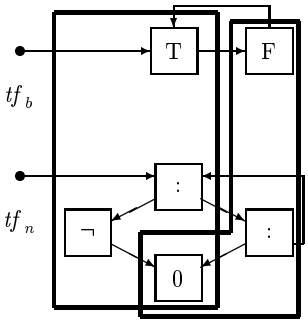

Figure 6: Example space-equivalence proof. $\approx$ is a space-equivalence relating *ListBool* and *ListNot* graphs. It is defined by *(Nil),(False)* and *(True)*. The example proof below the dotted line shows $tf_b \approx tf_n$. The diagram illustrates the proof: the *(True)* and *(False)* steps each relate the nodes in a thick box; the number of boxes is proportional to the sizes of both graphs.

Definition 16. (**Space-equivalence relation**)
If $\approx$ and $\approx^{-1}$ are proportional-space relations then $\approx$ is a space-equivalence relation. □

Definition 15.7 ensures that every node of a graph $G \in dom(\gtrsim)$ matches a pattern node at some level of the proof tree. We can abbreviate a proof tree of depth d as $\{G_i \gtrsim^{\theta_i} H_i\}_{i=1}^{d}$ where $(G_{i+1} \gtrsim^{\theta_{i+1}} H_{i+1})$ implies $(G_i \gtrsim^{\theta_i} H_i)$. By Proposition 4, this proof guarantees the space bound.

PROPOSITION 4. *If $\gtrsim$ is a proportional space relation then there is a k s.t. if $G \gtrsim H$ then $\#H \leq k \times \#G$.*
Proof: We prove $k \times \#G \geq d$ where d is the depth of the proof tree $\{G_i \gtrsim^{\theta_i} H_i\}_{i=1}^{d}$ from Definition 15.4. Definition 15.2,15.3,15.7 implies $k \times d \geq \#H$. □

The first level of a proof tree is generated by (*Start*) which says we should show that corresponding root nodes relate. The proof succeeds when (*End*) applies — when there is nothing left to relate and nothing else we want to relate.

$$(Start)\ G_{v_1,\ldots,v_n} \gtrsim H_{u_1,\ldots,u_n} \Leftrightarrow G \gtrsim^{\{(v_i,u_i)\}_{i=1}^{n}} H$$

$$(End)\ \frac{a \in dom(\theta) \Rightarrow \theta(a) = a \quad dom(G) \cap dom(\theta) = dom(H) \cap rng(\theta) = \emptyset}{G \gtrsim^{\theta} H}$$

Example 7. The rules *(True)*, *(Nil)* and *(False)* in Figure 6 define a space-equivalence $\approx$ which relates *ListBool* graphs to equivalent *ListNot* graphs. The rules do not remove the elements of *ListNot* graphs because they may be shared. Each step relates one *ListBool* node to at most three *ListNot* nodes so: $G \approx H \Rightarrow \#G \leq \#H \leq 3 \times \#G$.
Figure 6 gives an example proof, showing that tf_b and tf_n are space-equivalent graphs. □

4. SPACE-EQUIVALENT EVALUATORS

In this section we show how to prove that evaluator A does not have a space leak with respect to evaluator B. Section 4.1 defines what it means for an evaluator to have a space leak. Section 4.2 explains that if the space-equivalence relation between two languages is a *lockstep bisimulation* then there cannot be a space leak. Section 4.3 shows how to prove a space-equivalence is a lockstep bisimulation and Section 4.4 gives some rules for simplifying the proofs.

4.1 Space Leaks in Evaluators

Given a space-equivalence relation between two graph languages, the evaluator A for the first language is *leakier* than the evaluator B for the second language if its space usage for G is much greater than the space usage of B on H where G and H are space equivalent. Evaluators are space-equivalent if neither is leakier than the other.
We must decide how much more space A can use before we call it leakier. Definition 17 says that it must be worse by more than any constant factor and Proposition 5 confirms this is an asymptotic distinction, so if A has a higher space complexity than B for *some* program then it has a leak.

Definition 17. (**Leakier evaluator** $\rightsquigarrow$)
A is leakier than B with respect to $\sim$:
A $\rightsquigarrow$ B exactly if
$\exists G, H s.t. G \sim H \wedge \forall k, space(G, \text{A}) \not\leq k \times space(H, \text{B})$ □

Definition 18. (**Space-equivalent evaluators** $\leadsto$)
A and B are space-equivalent if they are mutually not leakier: A $\leadsto$ B $\Leftrightarrow$ A $\not\rightsquigarrow$ B $\wedge$ A $\not\leftsquigarrow$ B □

PROPOSITION 5.
Space-equivalence is asymptotic equivalence: A $\leadsto$ B $\Leftrightarrow$
$G \sim H \Rightarrow \forall F, (G \in Fspace(\text{A}) \Leftrightarrow H \in Fspace(\text{B}))$ □

If this definition seems too liberal, classifying A as leakier if it used more than some fixed k times as much space as B would demand an evaluation model accurate within constant factors and a justification for k. A weaker definition calling A is leakier if it uses unlimited space when B uses bounded space cannot distinguish an exponential-space program from a logarithmic one. Our asymptotic leak definition is meaningful in that the results in the abstract hold for real implementations. It is worth noting that leak examples are non-terminating programs. The space usage of any terminating program is bounded.

4.2 Lockstep Evaluators and Bisimulation

A bisimulation demonstrates an operational equivalence. Milner developed the theory to show that the observable

actions of two communicating systems are the same [15]. We want to check that the space usage of two evaluations remain proportional. We use a variation on the idea of *applicative bisimulation* [1] to show that a proportional-space relation is preserved by evaluation.

Cast in our graph terminology, $\sim$ is a *simulation* if whenever $G \sim H$ and $H \longrightarrow_B^* H'$ there is a G' such that $G \longrightarrow_A^* G'$ and $G' \sim H'$. So A simulates everything that B does. If $\sim^{-1}$ is also a simulation then it is a *bisimulation*. Normally bisimulation is defined on a language to define operational equivalence in that language. Then *congruence* is a desirable result, so replacing sub-programs by operational equivalents yields an operationally equivalent program (Howe's method [11] is one approach to proving congruence). We are using bisimulation in a more general sense, to show that a graph of one language and its equivalent in another language have the same operational behaviour. We do not discuss the corresponding generalised form of congruence but if the sub-graphs we replace are closed, our bisimulation is a congruence.

Lockstep Bisimulation

To show space-equivalence of A and B with respect to a space-equivalence relation $\approx$, we need to show that $\approx$ is preserved by evaluation in a bisimulation-like manner. Bisimulation as stated above is too generous for our purposes. We can only allow a fixed number of steps between states where the relation $\approx$ holds so that any large increase in graph size cannot go undetected. We know a space-valid evaluator only allocates a bounded amount of space at each step, so limiting the number of steps between relation points is adequate. Similarly, both evaluators must take some steps between relation points to ensure both traces are followed. We call this kind of bisimulation w-lockstep where w is the number of steps allowed between relation points.

Definition 19. (w-**lockstep simulation**)
$\sim$ is a w-lockstep simulation between A and B iff

$$\begin{array}{lll} G & & G \longrightarrow_A^i G' \\ \sim & \wedge\, H \longrightarrow_B H'' \Rightarrow \exists i, j \in [1, w] s.t. & \sim \\ H & & H \longrightarrow_B^j H' \end{array}$$

That is, graphs related by $\sim$ are related again after no more than w evaluation steps.

Corollary: If $\sim$ and $\sim^{-1}$ are w-lockstep simulations then $\sim$ is a w-lockstep bisimulation. □

PROPOSITION 6. *If $\stackrel{\geq}{\sim}$ is a proportional bisimulation between A and B and $G \stackrel{\geq}{\sim} H$ then*
$\exists k, space(H, B) \leq k \times space(G, A) \Leftrightarrow B \not\asymp A$.
Proof: Induction on length of evaluation sequences.
Corollary: If $\approx$ is a space-equivalent bisimulation between A and B then $A \bowtie B$ □

If a proportional-space relation $\stackrel{\geq}{\sim}$ is a w-lockstep simulation between A and B then by Proposition 6 it guarantees $A \not\asymp B$. Intuitively, at every pair of states $G \stackrel{\geq}{\sim} H$, we know $\#H \leq k \times \#G$ so $\#G$ bounds the space occupied by H until the next point where $\stackrel{\geq}{\sim}$ holds.

In this paper, the evaluators we compare operate in exact lockstep — they are 1-lockstep evaluators. To prove a space-equivalence $\sim$ between A states and B states is a bisimulation we must show (5).

$$\begin{array}{lll} G & & G \longrightarrow_A G' \\ \sim & \wedge\, (G \longrightarrow_A G' \vee H \longrightarrow_B H') \Rightarrow & \sim \\ H & & H \longrightarrow_B H' \end{array} \quad (5)$$

4.3 Proving a Relation is a Bisimulation

To show a space-equivalence relation is a bisimulation, we show that evaluating equivalent graphs produces equivalent graphs (5). Suppose $\stackrel{\geq}{\sim}$ relates A-states to B-states. To show $\stackrel{\geq}{\sim}$ is a simulation we show that for each B-redex, every related A-state is also a redex and that after the evaluation step the graphs remain related. So first we find the pairs of rules whose redexes relate under $\stackrel{\geq}{\sim}$. Then we show that that matching graphs always — whatever nodes become garbage after the step — still relate after an evaluation step.

The space-bound property of the relations we prove are bisimulations is incidental to the bisimulation proof process. We cannot construct these proofs from the space-equivalence definition alone. In this paper, the proofs also use the rules defined in Figure 7. They deduce a valid proof tree from a known proof tree. They are correct by the following arguments. *(Gc)* introduces or removes nodes without affecting reachable space. *(Sub)* removes or adds a substitution that will not alter the contents of the graphs. *(Ren)* exchanges the addresses of nodes where the nodes at exchanged addresses are equivalent. Another way to look at it is that it moves to a different part of the proof tree. *(Rel)* lets us add addresses to the relation which we know will successfully relate, again allowing us to move to other parts of the proof tree.

Example 8. Recall evaluators b and n from Section 2.3. To prove the space-equivalence $\approx$ of Figure 6 is a lockstep bisimulation and therefore $b \bowtie n$, we show that whenever $\approx$ relates an b-redex to an n-redex, the graphs still relate after an evaluation step. Figure 8 details the proof. □

4.4 Identity Space-Equivalence

When one evaluator is a variant of another, we can use the Identity space-equivalence, $\stackrel{I}{\approx}$ (Figure 9), as a starting point for defining a space-equivalence relation. It relates a graph to a renaming of itself. Including $\stackrel{I}{\approx}$ in a relation also greatly simplifies its bisimulation proof as we only need to show the lockstep property (5) for related redexes that are evaluated by different rules.

PROPOSITION 7. *If A and B use the same rule to reduce G and H then the resultant graphs remain related by $\sim$. Formally, suppose $\sim \supseteq \stackrel{I}{\approx}$ relates A states to B states and $(L \longrightarrow R) \in A \cap B$ and there are variable substitutions θ, σ and hole substitutions ϕ, ψ s.t.*

$$\begin{array}{lll} G \equiv G'(match\ \theta\ \phi\ L) & & G \longrightarrow_A G'(match\ \theta\ \phi\ R) \\ \sim & then & \sim \\ H \equiv H'(match\ \sigma\ \psi\ L) & & H \longrightarrow_B H'(match\ \sigma\ \psi\ R) \end{array}$$ □

Example 9. Implementations can use *indirections* to avoid copying expressions [19]. Evaluator h copies values in its (*Update*) rule (Figure 5). Figure 10 defines i, a variant of h that uses an indirection node instead. But can we be sure that such indirections cannot form chains, causing a space leak? To prove $i \bowtie h$ we define a space-equivalence between h and i graphs and show it is a bisimulation. We

(Rel) $G \sim^{\sigma} H,\ \sigma \subseteq \bigcup \theta_i \wedge dom(\sigma) \subseteq dom(G) \wedge rng(\sigma) \subseteq dom(H)$
where $\{G_i \sim^{\theta_i} H_i\}_{i=1}^{d}$ is the proof tree of $G \sim^{\theta} H$

(Gc) $G \sim^{\theta} H \Leftrightarrow GG' \sim^{\theta} HH'$, if $dom(H') \cap reach(H, rng(\theta)) = dom(G') \cap reach(G, dom(\theta)) = \emptyset$

(Sub) $(G \sim^{\theta} H)\sigma \Leftrightarrow (G \sim^{\theta} H)$, if $(dom(\sigma) \cup rng(\sigma)) \cap (reach(G, dom(\theta)) \cup reach(H, rng(\theta))) = \emptyset$

(Ren) $(G\{a \mapsto M\} \sim^{\theta\{(a,b)\}} H\{b \mapsto N\})[y/x] \Leftrightarrow (G\{x \mapsto M\} \sim^{\theta\{(x,y)\}} H\{y \mapsto N\})[b/a]$

Figure 7: Rules used in proportional and space-equivalence bisimulation proofs. *(Rel)* says that a sub-relation σ of relation θ can be proved, it may include parts of the relation discovered further up the proof tree provided the nodes involved are in the graphs; *(Gc)* adds or removes unreachable nodes to the graphs; *(Sub)* adds or removes variables not occuring in a graph to the substitution; *(Ren)* renames addresses in the relation to addresses known to be related.

$(False : \cdots)$ redexes: $G\{a \mapsto F\ b\}_a \approx H\{x \mapsto y : z, y \mapsto 0\}_x \Rightarrow G\{a \mapsto T\ b\}_b \approx H\{x \mapsto y' : z, y \mapsto 0, y' \mapsto \neg\ y\}_z$

Proof: LHS	$\Rightarrow G\{a \mapsto F\ b\} \approx^{\{(a,x)\}} H\{x \mapsto y : z, y \mapsto 0\}$	apply *(Start)*
	$\Rightarrow G \approx^{\{(b,z)\}} H\{y \mapsto 0\}[a/x]$	decompose with *(False)*
a,x,y' garbage	$\Rightarrow$ RHS by *(Start)*	
otherwise	$\Rightarrow G \approx^{\{(b,z)\}} H\{y \mapsto 0, y' \mapsto \neg y\}[a/x]$	*(Gc)* introduces y'
	$\Rightarrow G\{a \mapsto T\ b\} \approx^{\{(a,x)\}} H\{x \mapsto y' : z, y \mapsto 0, y' \mapsto \neg y\}$	apply *(True)*
	$\Rightarrow G\{a \mapsto T\ b\} \approx^{\{(b,z)\}} H\{x \mapsto y' : z, y \mapsto 0, y' \mapsto \neg y\}$	*(Rel)* moves roots $\Rightarrow$ RHS by *(Start)*

$(True : \cdots)$ redexes: $G\{a \mapsto T\ b\}_a \approx H\{x \mapsto y' : z, y' \mapsto \neg\ y, y \mapsto 0\}_x \Rightarrow G\{a \mapsto F\ b\}_b \approx H\{x \mapsto y : z, y' \mapsto \neg\ y, y \mapsto 0\}_z$
Proof: follows the same steps as the proof above except no *(Gc)* step is needed.

Figure 8: Bisimulation proof showing b is space-equivalent to n. Whenever $\approx$ (Figure 6) relates an b-redex to an n-redex, the graphs remain related after one evaluation step. The proofs use the $\approx$ rules and laws from Figure 7. The proof says address a is garbage in the RHS iff x and y' are as well. No other redex combinations are related by $\approx$.

---

can define graphs containing indirection chains but they will not evaluate and have no h counterpart.
There are a lot of redex combinations to prove. Fortunately, Proposition 7 tells us we only need to consider those that use different evaluation rules. Figure 11 summarises the proof for these cases which involve the new rules of i. □

5. EVALUATORS WITH LEAKS

To prove A is leakier than B, A $\rightsquigarrow$ B, by Definition 17 we require a bisimulation $\gtrsim$ and graphs G and H such that $G \gtrsim H$ and $space(G, \boxed{A}) \not\leq k \times space(H, \boxed{B})$.
Section 5.1 develops a characterisation of these leak examples to help us identify potential sources of leaks. Section 5.2 looks at leaks caused by *allocation chains* and Section 5.3 looks at leaks caused by *reference duplication*.

5.1 Finding and Characterising Leaks

We can prove A $\rightsquigarrow$ B with respect to the bisimulation $\sim$ by finding a sequence of related graphs which form an *unbounded-bounded* sequence.

Definition 20. (**Unbounded-bounded sequence**)
(G, H, G', H') form an unbounded-bounded evaluation sequence of A and B if G grows without limit but H has bounded growth. Formally, if there is a lockstep bisimulation $\sim$ s.t.

$$\begin{array}{ccc} G & \longrightarrow_A^* & G' \\ \sim & & \sim \\ H & \longrightarrow_B^* & H' \end{array} \quad \begin{array}{l} \forall k, \#G' \not\leq \#G + k \\ \wedge \\ \exists k, \#H' \leq \#H + k \end{array}$$

□

If A $\rightsquigarrow$ B there will always be such a sequence (Proposition 8) and if G and H are finite we have found the leak (Proposition 9).

PROPOSITION 8. $\boxed{A} \rightsquigarrow \boxed{B} \Rightarrow \exists (G, H, G', H')$ *which form an unbounded-bounded sequence.*
Proof: we know $\exists G, H, \forall k, space(G, \boxed{A}) \not\leq k \times space(H, \boxed{B})$. *$G$ and H start an evaluation sequence (G, H, G', H') of unbounded length. If this is not an unbounded-bounded sequence then either some subsequence of it is an unbounded-bounded sequence or* $space(G', \boxed{A}) \not\leq k \times space(H', \boxed{B})$. □

PROPOSITION 9. *If (G, H, G', H') is an unbounded-bounded sequence of* A *and* B *and* $\exists k, \#G \leq k$ *and* $\#H \leq k$ *then* $space(G, \boxed{A}) \not\leq k \times space(H, \boxed{B})$. □

Now we need a way of finding these sequences. Definition 21 below is a co-inductive characterisation of all unbounded-growth graphs of an evaluator. Such evaluations never terminate, so this definition is a subset of Gordon's definition of all divergent programs [9]. In the following sections we identify subsets of *InfGrowth* which we can use to prove a program has unbounded growth.

Definition 21. (*InfGrowth*)
InfGrowth(A) contains graphs with unbounded growth on A, defined as the greatest fixpoint of $\mathcal{I}$.
InfGrowth(A) $= \nu X.\mathcal{I}(X)$
where $\mathcal{I}(X) = \{G \mid \exists G' \in X, G \longrightarrow^* G' \wedge \#G' > \#G\}$ □

$$(Var)\frac{(G \overset{I}{\approx}^{\theta\{(x,y)\}} H)[b/a]}{G\{a \mapsto x\} \overset{I}{\approx}^{\theta\{(a,b)\}} H\{b \mapsto y\}} \qquad (Func)\frac{(G\{a_i \mapsto M_i\}_{i=1}^{y} \overset{I}{\approx}^{\theta\{(a_i,b_i)\}_{i=1}^{y}} H\{b_i \mapsto N_i\}_{i=1}^{n})[b/a, v_1/u_1, \ldots, v_x/u_x]}{G\{a \mapsto F\, u_1 \cdots u_x.M_1 \cdots M_y\} \overset{I}{\approx}^{\theta\{(a,b)\}} H\{b \mapsto F\, v_1 \cdots v_x.N_1 \cdots N_y\}}$$

Figure 9: Space-equivalence rules defining $\overset{I}{\approx}$, a relation relating a graph to a renaming of itself. *(Var)* relates variables assuming they are the addresses of nodes that can be related. *(Func)* relates nodes whose terms are constructed from the same function symbol, assuming the corresponding sub-terms relate.

$$\boxed{\text{i}} = \boxed{\text{h}} \setminus \{(Update)\} \cup \left\{ \begin{array}{lll} \{a \mapsto \mathbb{V}, x \mapsto \bot, s \mapsto \#x\, t\}_{a,s} & \longrightarrow & \{a \mapsto I\, x, x \mapsto \mathbb{V}\}_{a,t} \quad (VUpdate) \\ \{a \mapsto I\, x, y \mapsto \bot, s \mapsto \#y\, t\}_{a,s} & \longrightarrow & \{a \mapsto I\, x, y \mapsto I\, x\}_{a,t} \quad (IUpdate) \\ \{a \mapsto I\, x, x \mapsto \mathbb{V}, s \mapsto \mathbb{R} : t\}_{a,s} & \longrightarrow & \{a \mapsto Red\, \mathbb{R}[\mathbb{V}], x \mapsto \mathbb{V}\}_{a,t} \quad (IReduce) \end{array} \right\}$$

Figure 10: A Core Haskell evaluator with value indirections. The grammar is that of Figure 4 extended with indirections: $X ::= \cdots \mid I\, x$. Instead of duplicating values, *(VUpdate)* creates an indirection. Where the value is an indirection, the indirection is duplicated *(IUpdate)*. For brevity, indirected values are reduced by copying them into the hole of the reduction context as before.

5.2 Allocation-Chain Leaks

This section focuses on a particular kind of unbounded-bounded sequence where one evaluator builds an unlimited chain of nodes in the graph while the other does not. In Haskell implementations, certain uses of indirection nodes can form chains in this way. Unwinding cyclic data structures could also cause this kind of leak. Lazy memo-functions [12] can prevent some of these leaks.

Definition 22. (**Allocation-Chain Leak**)
Graphs (GL, HK) exhibit an allocation-chain leak of $\boxed{\text{A}}$ and $\boxed{\text{B}}$ if there is a lockstep bisimulation $\sim$, a substitution θ and an n s.t.

$$\begin{array}{ccc} GL & \longrightarrow_A^n & GG'(L\theta) \\ \sim & & \sim \\ HK & \longrightarrow_B^n & HK \end{array}$$

where $|gc(G'(L\theta))| > |L|$. GL continually grows as a chain of G' sub-graphs build up but HK remains the same. □

Definition 22 describes perhaps the simplest kind of leak example: $\boxed{\text{B}}$ keeps repeating the same sequence of states while $\boxed{\text{A}}$ also repeats states but allocates some extra structure which is kept live. For example, an implementation that buffered all input might have this behaviour. The leak would be shown up by a program that continually read then processed some input. It would be an allocation-chain leak if the input buffer was some list-like structure.

PROPOSITION 10. *If (G, H) exhibit an allocation-chain leak of $\boxed{A}$ and $\boxed{B}$ then they begin an unbounded-bounded sequence.*
Proof: $G \in InfGrowth(\boxed{A})$ *and* $H \notin InfGrowth(\boxed{B})$. □

Example 10. Consider $\boxed{\text{v}}$ (6), a simple list inverter for *ListNot* graphs, defined by a single rule.

$$\boxed{\text{v}} = \left\{ \{x \mapsto y : z\}_x \longrightarrow \{x \mapsto y' : z, y' \mapsto \neg\, y\}_z \right\} \qquad (6)$$

We suspect $\boxed{\text{v}} \backsimeq \boxed{\text{b}}$ (3) because the lazy inversion method of just adding a $\neg$ to every element might somehow cause a chain to build up. $(False')$ and $(True')$ define a proportional-space relation $\gtrsim$ which relates *ListNot* (:) nodes to *ListBool* nodes, ignoring the list elements.

$$(False')\frac{(G \gtrsim^{\theta\{(c,y)\}} H)[x/a]}{G\{a \mapsto b : c\} \gtrsim^{\theta\{(a,x)\}} H\{x \mapsto F\, y\}}$$

$$(True')\frac{(G \gtrsim^{\theta\{(c,y)\}} H)[x/a]}{G\{a \mapsto b : c\} \gtrsim^{\theta\{(a,x)\}} H\{x \mapsto T\, y\}}$$

To prove $\boxed{\text{v}} \backsimeq \boxed{\text{b}}$, we use Definition 22 to construct a suitable example (7). After two evaluation steps on $\boxed{\text{v}}$ the graph grows, on $\boxed{\text{b}}$ it stays the same size. As $\{x \mapsto y'' : x\}_x \equiv \{x \mapsto (y : x)[y''/y]\}_x$, this growth will continue indefinitely, causing a leak.

$$\begin{array}{ccc} \left\{ \begin{array}{l} x \mapsto y : x, \\ y \mapsto 0 \end{array} \right\}_x & \longrightarrow_v^2 & \left\{ \begin{array}{l} x \mapsto y'' : x, y \mapsto 0, \\ y'' \mapsto \neg\, y', y' \mapsto \neg y \end{array} \right\} \\ \gtrsim & & \gtrsim \\ \{a \mapsto F\, a\}_a & \longrightarrow_b^2 & \{a \mapsto F\, a\}_a \end{array} \qquad (7)$$

We should also check that $\boxed{\text{b}} \not\backsimeq \boxed{\text{v}}$. The proof that $\gtrsim$ is a proportional bisimulation follows the same steps as the $\boxed{\text{b}} \backsimeq \boxed{\text{n}}$ proof in Figure 8, using $(False')$ and $(True')$ in place of $(False)$ and $(True)$. □

5.3 Reference Duplication Leaks

Another kind of leak is caused by the context of an evaluation holding a dead reference to a growing structure. This section characterises some leaks that arise in this way.

Definition 23. (**Reference duplication leak**)
Graphs (G, H) exhibit a reference duplication leak of $\boxed{\text{A}}$ and $\boxed{\text{B}}$ if there is a bisimulation $\sim$ and an n s.t.

$$\begin{array}{ccc} G \equiv C\{a \mapsto M\}L & \longrightarrow_A^n & C\{a \mapsto M', a' \mapsto M\}L[a'/a] \\ \sim & & \sim \\ H \equiv D\{b \mapsto N\}K & \longrightarrow_B^n & D\{b \mapsto N', b' \mapsto N\}K[b'/b] \end{array}$$

where:
$a \in FV(gc(L)), a \in FV(gc(CL) \setminus L)$
$b \in FV(gc(K)), b \notin FV(gc(DK) \setminus K)$
Sub-graphs C and D are unchanged by evaluation; C refers to a but D does not refer to b. After n evaluation steps, a is still reachable but b is not; the structure at a starts to grow but b is garbage. □

$$(I)\frac{(G\{d \mapsto \mathbb{V}\} \approx^{\theta\{(d,e)\}} H\{e \mapsto \mathbb{V}'\})[b/a]}{G\{a \mapsto V\ \mathbb{V}\} \approx^{\theta\{(a,b)\}} H\{b \mapsto I\ c, c \mapsto V\ \mathbb{V}'\}} \qquad (VI)\frac{(G\{x \mapsto V\ \mathbb{V}\} \approx^{\theta\{(x,y)\}} H\{y \mapsto V\ \mathbb{V}'\})[b/a]}{G\{a \mapsto V\ \mathbb{V}, x \mapsto V\ \mathbb{V}\} \approx^{\theta\{(a,b)\}} H\{y \mapsto V\ \mathbb{V}', b \mapsto I\ y\}}$$

$$(II)\frac{(G\{x \mapsto V\ \mathbb{V}\} \approx^{\theta\{(x,y)\}} H\{y \mapsto I\ z\})[b/a]}{G\{a \mapsto V\ \mathbb{V}, x \mapsto V\ \mathbb{V}\} \approx^{\theta\{(a,b)\}} H\{b \mapsto I\ z, y \mapsto I\ z\}} \qquad (IV)\frac{(G\{a \mapsto V\ \mathbb{V}\} \approx^{\theta\{(a,b)\}} H\{b \mapsto V\ \mathbb{V}'\})[y/x]}{G\{a \mapsto V\ \mathbb{V}, x \mapsto V\ \mathbb{V}\} \approx^{\theta\{(a,b)\}} H\{b \mapsto V\ \mathbb{V}', y \mapsto I\ b\}}$$

$$(VV)\frac{(G\{a \mapsto V\ \mathbb{V}\} \approx^{\theta\{(a,b)\}} H\{b \mapsto V\ \mathbb{V}'\})[y/x]}{G\{a \mapsto V\ \mathbb{V}, x \mapsto V\ \mathbb{V}\} \approx^{\theta\{(a,b)\}} H\{b \mapsto V\ \mathbb{V}', y \mapsto V\ \mathbb{V}'\}}$$

Bisimulation proof for $\approx$ for *(Update)* and *(VUpdate)* redexes:
$G\{a \mapsto V\ \mathbb{V}, x \mapsto \bot, s \mapsto \#x\ t\}_{a,s} \approx H\{b \mapsto V\ \mathbb{V}', y \mapsto \bot, u \mapsto \#y\ v\}_{b,u}$
$\Rightarrow G\{a \mapsto V\ \mathbb{V}, x \mapsto V\ \mathbb{V}\}_{a,t} \approx H\{b \mapsto I\ y, y \mapsto V\ \mathbb{V}'\}_{b,v}$
Proof: *(Start),(Func)* on # and $\bot$, *(Sub)* on $[u/s]$ decompose LHS into $(G\{a \mapsto V\ \mathbb{V}\} \approx^{\{(a,b),(t,v)\}} H\{b \mapsto V\ \mathbb{V}'\})[y/x]$.
(a) x not reachable in RHS: *(Func)* decomposes values, *(I)* builds RHS. (b) x is reachable: *(Ren)* and *(VI)* generate RHS.

Figure 11: The rules above the dotted line define the space-equivalence $\approx$ where $\approx \cup \overset{I}{\approx}$ relates plain Core Haskell graphs to ones with indirections. This relation assumes a more explicit formulation of the grammar: to handle the $\mathbb{V}$ and $\mathbb{R}$ categories in our term-graph framework, values are prefixed with the function symbol V and reduction contexts with R. Rule *(I)* matches a value to an equivalent indirected value, it applies when the value at c is reachable only via indirections. Any other indirections to such a value are related to (duplicated) values in the h graph with *(II)*. Similarly, *(VI)* relates an indirection to a value in the case where y is reachable from non-indirections and *(IV)* relates additional indirections to such a value. *(VV)* is like *(II)* or *(IV)* for duplicated values. The bisimulation proof summary for $\approx$ below the dotted line shows one of the cases not covered by the identity equivalence. x may or may not be garbage in the right-hand side depending on the contents of the graph matching G, so we show that the graphs will remain related by $\approx$ in both cases. We omit the proof for *(Update)* and *(IUpdate)* redexes and for *(Reduce)* and *(IReduce)* redexes which are similar.

$$\boxed{\text{k}} = \boxed{\text{h}} \setminus \{(Lookup), (Update)\} \cup \left\{ \begin{array}{rcll} \{a \mapsto x, x \mapsto N\}_{a,s} & \longrightarrow & \{a \mapsto N, x \mapsto N, t \mapsto \#x\ s\}_{a,t} & (Lookup') \\ \{a \mapsto \mathbb{V}, x \mapsto N, s \mapsto \#x\ t\}_{a,s} & \longrightarrow & \{a \mapsto \mathbb{V}, x \mapsto \mathbb{V}\}_{a,t} & (Update') \end{array} \right\}$$

Figure 12: Non-black holing Core Haskell evaluator k, a variant of h (Figure 5) which leaves the binding for a variable untouched while finding its value (the grammar is unchanged from Figure 4).

PROPOSITION 11. *If (G, H) exhibit a reference duplication leak of A and B then they begin an unbounded-bounded sequence.*
Proof: $G \in InfGrowth(\boxed{A})$ *and* $H \notin InfGrowth(\boxed{B})$. □

Example 11. Our Haskell evaluator h includes *black holing*: in the *(Lookup)* rule, x is re-bound to $\bot$ while its value is found. Corresponding techniques are used in real implementations to avoid leaks ([13]. Figure 12 defines k, a variant of h without black holing. It leaves the binding for x untouched while finding its value.
We can prove k $\rightsquigarrow$ h with a reference duplication leak example. The Core Haskell program (8) tries to determine if the infinite list $a = \bot : \bot : \bot : \cdots$ is finite. It can be translated to the graph G (9).

$$\begin{array}{l} \mathit{let}\ \ \mathit{finite}\ (h : t) = \mathit{finite}\ t \\ \quad\ \ \mathit{finite}\ [\,] = \mathit{True} \\ \quad\ \ \mathit{list}\ x = \mathit{let}\ a' = \mathit{list}\ x\ \mathit{in}\ x : a' \\ \quad\ \ a = \mathit{list}\ \bot \\ \quad\ \ b = \mathit{finite}\ a \\ \mathit{in}\ b \end{array} \tag{8}$$

$$G = F\{a \mapsto \mathit{list}\ \epsilon, b \mapsto \mathit{finite}\ a, \mathit{main} \mapsto b\}_{\mathit{main},\epsilon}$$
$$F = \left\{ \begin{array}{l} \mathit{finite} \mapsto\ \ \lambda l.\mathit{case}\ l\ \mathit{of} \\ \qquad\qquad \{[\,] \to \mathit{True}, (h : t) \to \mathit{finite}\ t\} \\ \mathit{list} = \lambda x.\mathit{let}\ \{a' = \mathit{list}\ x\}\ \mathit{in}\ x : a' \end{array} \right\} \tag{9}$$

The evaluation traces of G on k and h are shown below.
$G \longrightarrow_k C\{a \mapsto \mathit{list}\ \epsilon\}L \longrightarrow_k^{13} C\{a \mapsto \epsilon : a', a' \mapsto \mathit{list}\ \epsilon\}R$
$G \longrightarrow_h D\{a \mapsto \mathit{list}\ \epsilon\}L \longrightarrow_h^{13} D\{a \mapsto \epsilon : a', a' \mapsto \mathit{list}\ \epsilon\}R$
where:
$C = F\{b \mapsto \mathit{finite}\ a, s \mapsto \#b\ \epsilon\}$
$D = F\{b \mapsto \bot\ a, s \mapsto \#b\ \epsilon\}$
$L = \{\mathit{main} \mapsto \mathit{finite}\ a\}_{\mathit{main},s}$
$R = \{\mathit{main} \mapsto \mathit{finite}\ a'\}_{\mathit{main},s}$
After *(Lookup)* creates the update marker node for b at address s, the nodes in C and D are never altered again. C keeps a reference to a but D has no such reference. The evaluation of *finite* a updates a with the first element of the list and the tail of the list is in the new node a'. The root of the new graph R is just $L[a'/a]$. On h, a is now garbage and the program runs in constant space. On k, C holds onto the list at a as it grows which causes a leak.

$$(TermBot)\frac{(G \gtrsim^{\theta} H)[b/a]}{G\{a \mapsto N\} \gtrsim^{\theta\{(a,b)\}} H\{b \mapsto \bot\}}$$

To prove that k $\not\rightsquigarrow$ h (black holing is never leakier than not black holing) we define the proportional-space relation $\gtrsim$ by *(VV)* from Figure 11 and *(TermBot)* and $\overset{I}{\approx}$. A simulation proof showing h $\not\rightsquigarrow$ k can be constructed. The simulation $\gtrsim$ cannot be a bisimulation as k may continue evaluating after h stops. □

Example 1 was another instance of this kind of leak, where *head* held a reference to a growing list. In that case a small change in evaluation order — getting the garbage collector to evaluate *head* of $a : b$ as soon as possible [25] fixes the leak.

6. RELATED WORK

As lazy functional language implementations developed there was a move away from directly implementing graph reduction or SKI reduction towards a compiled code style better suited to modern computer architectures. Some machine designs have been found to be leakier than expected. The literature documents failure to perform *black holing* [13; 20] and certain uses of indirections [19] as sources of space faults, examples that we have dealt with in this paper. Careless implementation of environment-based machines causes another space fault mended by *environment trimming* or *stack stubbing* [19].

Improved garbage collection can eliminate leaks. Type-based [16], *projection-shorting* [25] and *indirection-shorting* [19] collectors remove more semantically dead heap cells than our reachability-based collector. Some of these fixes can also be done in the evaluator [24], while others use the garbage collector to change the evaluation order.

Profiling to Detect Leaks

It is not easy to tell which parts of a Haskell program cause structures to build up. Lazy evaluation can process potentially huge structures in constant space; sharing can save work at the expenses of increasing space complexity. As we saw in Example 1, small changes to a program can have a huge effect on its space use.

Heap profiling [22] produces various diagrams of the live heap contents from which we can work out which functions are allocating structures, which are holding onto structures and what structures are growing. A biographical profile [20] detects whether heap cells have been or will be used, and can point directly to wasted space. As a heap profile only gives information on one run, we cannot tell how serious the problem is or if we have definitely fixed it later. We can compare profiles for equivalent programs or different compilers by hand — profiling is to our space-equivalence analysis as testing is to proof.

Space-Safety in Call-by-Value Languages

The ideal of a standard space-use model for evaluation has been investigated for call-by-value languages. Many optimisations were designed to replace heap use by stack use, so the issue of space safety soon arose. Some optimisations can extend the lifetime of data objects, causing leaks [5]. Appel defines safe-for-space evaluation of SML [2]. Clinger [6] gives a hierarchy of space-safety levels for implementations of Scheme. A stack-based implementation is bettered by machines that implement *tail recursion* properly and use *environment trimming* which are shown to give an asymptotic improvement in space use.

Operational Semantics for Call-by-Need

We needed an operational semantics to reason about the space usage of evaluation. Operational semantics suitable for languages like Haskell have been studied only relatively recently. Abramsky's call-by-name semantics [1] for the lazy lambda calculus was one of the first. Launchbury [14] gives a natural semantics for call-by-need with a syntax similar to Core Haskell. Sestoft [23] used this to define abstract machines (structural operational semantics) for lazy evaluation and Mountjoy [17] used it to develop a natural semantics which models the STG-machine used in Haskell implementations. Others using a term-graph formalism for studying sharing include Ariola and Arvind [3], though we are not aware of any work that deals with space issues as in this paper. Our term-graph framework enables operational semantics to be written which directly and fairly model space and time usage [4]. We are also using it to define a space-semantics for Core Haskell which is close to h, though less similar to [23].

Gustavsson and Sands [10] give another theoretical analysis of space. They present a collection of *space improvement* laws. These show programs have the same asymptotic space complexity in any context. Their laws only hold for one particular Core Haskell implementation model. They give an algebra for manipulating programs without worsening their space complexity. It is suitable for reasoning about the space-safety of small transformations but cannot handle larger changes or reason about recursive programs[1]. Our comparisons of whole evaluators do show 'space-safety' of recursive programs; we have used recursive programs to show an evaluator is leakier than another.

7. CONCLUSIONS

We have defined and compared the space usage of several Core Haskell evaluators in a term-graph rewriting framework. We defined a proportional-space relation to guarantee that there is a space bound between related graphs. We then adapted a variation on bisimulation to show that such a relation is preserved by evaluation and therefore there can be no asymptotic space leak. This technique is powerful enough to handle complex evaluators, providing they are *lockstep* — they have the same time complexity. We saw a certain use of *indirections* does not change asymptotic space complexity, as well as comparing some simpler graph evaluators. The example proofs provide a collection of laws for proving space-equivalence of any evaluators that fit the framework. We have made progress in classifying and finding space leaks, identifying programs which exhibit *infinite growth* as a characterisation of an asymptotic space leak. We saw examples where unnecessary allocation-chains or reference duplications cause leaks. We showed value indirections are safe but *black holing* is necessary in Haskell evaluators to prevent a reference duplication leak.

As this is an initial investigation of the proof technique, further work includes: improving and automating parts of the proof system and more detailed work on detecting and classifying leaks and their causes. We would also like to reason about constant-factor space usage improvements to increase the kinds of fault we can detect. The proportional-space relation proofs are linked to the garbage collector definition, so another useful extension would be to include a comparison of different garbage collectors in the proofs by linking the relation construction rules to the garbage collector definition. The lockstep restriction on our method is undesirable; finding ways to compare evaluators with different time com-

[1]Gustavsson and Sands have recently extended their algebra to include *letrec*.

plexities would be useful. The question of how to decide a suitable level of space-safety for lazy languages remains, but we have made progress by investigating techniques for deciding whether one evaluator matches the space efficiency of another.

8. ACKNOWLEDGEMENTS

Thanks to David Wakeling and Olaf Chitil for their comments to the anonymous reviewers for their suggestions.

9. REFERENCES

[1] S. Abramsky. *Research Topics in Functional Programming*, chapter 4: The lazy lambda calculus, pages 65–116. Addison-Wesley, 1990.

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

[3] Z. M. Ariola and Arvind. Properties of a first-order functional language with sharing. *Theoretical Computer Science*, 146:69–108, 1995.

[4] A. Bakewell and C. Runciman. The space usage problem: An evaluation kit for graph-reduction semantics. In *Draft Proc. 2nd Scottish Functional Programming Workshop, School of Computer Science, University of St. Andrews*, July 2000.

[5] D. R. Chase. Safety considerations for storage allocation optimizations. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–9. ACM Press, 1988.

[6] W. D. Clinger. Proper tail recursion and space efficiency. In *Proc. 3rd ACM International Conference on Functional Programming, Baltimore, Maryland*, pages 174–185. ACM Press, September 1998.

[7] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenburg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1999.

[8] A. D. Gordon. *Functional Programming and Input/Output.* PhD thesis, University of Cambridge, August 1992.

[9] A. D. Gordon. A tutorial on co-induction and functional programming. In *Glasgow Workshop on Functional Programming*, pages 78–95. Springer, September 1994.

[10] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In *The 3rd International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

[11] D. J. Howe. Equality in lazy computation systems. In *Proc. 4th Annual Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society, Los Alamitos, CA., 1989.

[12] J. Hughes. Lazy memo-functions. Technical Report 21, Chalmers Programming Methodology Group, September 1985.

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

[14] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages, Charleston*, pages 144–154, January 1993.

[15] R. Milner. *Communication and concurrency.* Prentice-Hall International, 1989.

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

[17] J. Mountjoy. The spineless tagless G-machine, naturally. In *Proc. Third International Conference on Functional Programming. Baltimore, Maryland*, pages 163–173. ACM Press, September 1998.

[18] H. R. Nielson and F. Nielson. *Semantics with applications: a formal introduction.* Wiley, 1992.

[19] S. L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, 1987.

[20] N. Röjemo and C. Runciman. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In *Proc. International Conference on Functional Programming. Philadelphia, Pennsylvania*, pages 34–41. ACM Press, May 1996.

[21] K. H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, University of Copenhagen, February 1996.

[22] C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–246, April 1993.

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

[24] J. Sparud. Fixing some space leaks without a garbage collector. In *Proc. Conference on Functional Programming Languages and Computer Architecture, Copenhagen*, pages 117–124. ACM Press, June 1993.

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