Looking for Leaks
Incomplete Draft for the SPACE2001 Workshop

Adam Bakewell[*]

January 8, 2001

Abstract

Implementations of programming languages that assist the programmer by providing automatic memory management can contain *space leaks*. We define a leaky implementation as one with asymptotically worse space usage than the language standard for some program — the leak witness.

This paper is about proving an implementation — or rather its operational semantics, expressed as a term-graph rewriting system — has a space leak. We do this by conducting an automated search for candidate leak witnesses. These are programs that do not terminate and keep on allocating more memory without limit.

To make the problem decideable we restrict ourselves to finding witnesses from a class of looping programs which are evaluated by repeatedly applying the same sequence of rules from the operational semantics. A non-standard unification procedure is used to construct a *super-rule* — the repeating rule sequence. A matching procedure tests whether a super-rule can self feed, producing its own redex and so representing a set of non-terminating programs. An approximation is applied to test if the super-rule will also allocate at each iteration, thus selecting candidate witnesses.

This brute force search is slow, so we employ the idea of *proof planning* to reduce the search space. We also use an approximation to the exact super-rule construction procedure to avoid generating multiple solutions.

The search technique is applied to variants of a simple call by name operational semantics for Core Haskell.

1 Space Leaks

We aim to find a proof that an implementation $\boxed{\mathrm{I}}$ of a programming language has a space leak with respect to the reference implementation $\boxed{\mathrm{R}}$. Our definition of a space leak is (1): $\boxed{\mathrm{I}}$ has asymptotically greater space usage on some program P; it will keep on demanding more and more memory whereas $\boxed{\mathrm{R}}$ runs P in constant space.

$$\boxed{\mathrm{I}} \precapprox \boxed{\mathrm{R}} \Leftrightarrow \exists P \cdot \forall k \in \mathbb{N}, \mathit{space}(P, \boxed{\mathrm{I}}) > k \times \mathit{space}(P, \boxed{\mathrm{R}}) \tag{1}$$

So $\boxed{\mathrm{I}}$ could have asymptotically lower space usage than the reference $\boxed{\mathrm{R}}$ for many programs, but it does not meet the reference standard in every case ($\precapprox$ is a quasi-ordering). These notes are about searching for P, the leak witness program.

2 Operational Semantics

We use the operational semantics of an implementation, expressed as a set of term-graph rewrite rules, as the input to the search procedure. Our term-graph framework [BR00] models programs as rooted graphs containing terms built from a user-defined grammar. The operational semantics is defined as a graph *evaluator* — the rewrite rules. In combination with a garbage collector, which removes graph nodes unreachable from the roots, this specifies the space behaviour of the operational semantics.

[*]Department of Computer Science, University of York, UK. ajb@cs.york.ac.uk

Example 1 (A semantics for call-by-need evaluation)
The λ-calculus with *let* expressions is defined by the higher-order term grammar below. Category X defines expression terms, including *let* and λ which both bind a variable. Category S defines stack terms which hold the context during evaluation.

Syntactic category	*Term definitions*	*Usual notation*	
X ::=	$LAM\ x.X$	$\lambda x.X$	*lambda abstraction*
|	$APP\ X\ x$	$X\ x$	*apply expression to variable*
|	$VAR\ x$	x	*variable*
|	$LET\ x.X\ X'$	$let\ x = X\ in\ X'$	*(recursive) local definition*
|	BOT	$\perp$	*constant for black holing*
S ::=	$PSH\ x\ s$	$x; s$	*stacked application argument x*
|	$UDM\ x\ s$	$\#x\ s$	*update marker for variable x*

Graphs built from this grammar are sets of bindings mapping address variables to terms. Free variables in a term are the addresses of other bindings in the graph. Graphs also have some root variables, two in this language. As a simple example, consider the graph $\{a \mapsto let\ id = \lambda x.x\ in\ id\ id\}a, \epsilon$. This has one expression node addressed by the first root which contains a simple program term. The second root variable addresses the current stack node, initially there are no stack nodes so it is null.

An evaluator for this language is specified below. Each rule has a left pattern and a right pattern, both patterns have roots. A rule replaces a sub-graph matching its left pattern with the sub-graph matching its right pattern under the same substitution. A substitution $S = (\theta, \phi)$ has two parts, θ maps all the variables in the pattern to those in the graph (including node addresses and bound variables) and ϕ maps the variables written in upper-case — the *holes* — in the pattern to terms in the graph. The right pattern may apply a substitution to a hole to replace any freed occurences of variables that were bound in the left pattern.

$$\begin{array}{rcll}
\{a \mapsto \lambda x.E, y \mapsto X, s \mapsto \#y\ t\}a, s & \longrightarrow & \{y \mapsto \lambda x.E, a \mapsto \lambda x.E\}y, t & (\textit{Update}) \\
\{a \mapsto x\}a, s & \longrightarrow & \{a \mapsto \perp, t \mapsto \#a\ s\}x, t & (\textit{LookupGood}) \\
\{a \mapsto \lambda x.E, s \mapsto b; t, b \mapsto F\ y\}a, s & \longrightarrow & \{b \mapsto E[y/x], a \mapsto \lambda x.E\}b, t & (\textit{ReduceGood}) \\
\{a \mapsto F\ x\}a, s & \longrightarrow & \{b \mapsto F, t \mapsto a; s, a \mapsto \perp x\}b, t & (\textit{Push}) \\
\{a \mapsto let\ y = E\ in\ X\}a, s & \longrightarrow & \{a \mapsto X[b/y], b \mapsto E[b/y]\}a, s & (\textit{Let})
\end{array}$$

The evaluation trace for the example graph is shown below. *(Let)* allocates a new node b to hold the definition of *id*. The application is decomposed by *(Push)*; the function part is evaluated first: the value of the variable b is retrieved by a *(Lookup)* followed by an *(Update)*. *(Reduce)* replaces parameter x with argument address b and evaluates the function body, so b is looked up again.

$$\begin{array}{ll}
\{a \mapsto let\ id = \lambda x.x\ in\ id\ id\}a, \epsilon & \\
\longrightarrow \{a \mapsto b\ b, b \mapsto \lambda x.x\}a, \epsilon & (\textit{Let}) \\
\longrightarrow \{c \mapsto b, d \mapsto a; \epsilon, a \mapsto \perp b, b \mapsto \lambda x.x\}c, d & (\textit{Push}) \\
\longrightarrow \{c \mapsto \perp, e \mapsto \#c\ d, d \mapsto a; \epsilon, a \mapsto \perp b, b \mapsto \lambda x.x\}b, e & (\textit{LookupGood}) \\
\longrightarrow \{c \mapsto \lambda x.x, b \mapsto \lambda x.x, d \mapsto a; \epsilon, a \mapsto \perp b\}c, d & (\textit{Update}) \\
\longrightarrow \{a \mapsto b, b \mapsto \lambda x.x\}a, \epsilon & (\textit{ReduceGood}) \\
\longrightarrow \{a \mapsto \perp, f \mapsto \#a\ \epsilon, b \mapsto \lambda x.x\}b, f & (\textit{LookupGood}) \\
\longrightarrow \{a \mapsto \lambda x.x\}a, \epsilon & (\textit{Update}) \\
\textit{space} = 5,\ \textit{time} = 7 &
\end{array}$$

The space usage of a graph, *space*, is defined as the maximum number of nodes needed during its evaluation, 5 in the example. This is our reference evaluator $\boxed{R}$. Some of its rules are labelled *Good*, we will use *Bad* variants to construct leaky evaluators. □

3 Self-Feeding Loops

To find a leak witness candidate by hand we might try to construct a simple non-terminating program, filling in details so that it repeatedly allocates on $\boxed{I}$ but does not on $\boxed{R}$.

For our search we will look for candidates whose execution follows a regular pattern, these self-feeding loops are defined by (2): they are graphs whose evaluation repeatedly follows the same n steps, resulting in a bigger graph which will follow the same steps (its variables may be named differently, hence the substitution θ in the equation).

$$\mathit{sfls}\,\boxed{\mathrm{A}} = \{G \mid \exists \theta, n \in \mathbb{N} \cdot G \longrightarrow^{n}_{A} H \wedge H \supset \theta(G)\} \tag{2}$$

This class of leak witnesses is not complete, it is easy to construct examples where irregular repetition must be detected to find the leak, but in practice it seems sufficient for many examples.

Now we can formalise the property we are searching for. We are looking for a sequence of rules $\langle r_1, \ldots, r_n \rangle \in \boxed{\mathrm{I}}^n$ such that an instance of their most general result matches a renaming of an instance of their most general redex. If we define a function *superRule* to find the most general redex and result of a rule sequence (the *super-rule* of that rule sequence) then we can define the leak candidates as the set (3).

$$\{S(L) \mid \langle r_1, \ldots, r_n \rangle \in \boxed{\mathrm{I}}^n \wedge \exists (L \longrightarrow R) = \mathit{superRule}\langle r_1, \ldots, r_n \rangle \wedge \exists S, \theta \cdot S(\theta(R)) \supset S(L)\} \tag{3}$$

Example 2 (A Self-Feeding Rule)
This example shows how a single rule can form a self-feeding loop. Replacing *(LookupGood)* in $\boxed{\mathrm{R}}$ with *(LookupBad)* gives us a model of lazy evaluation without *black holing* during variable lookup. Before, while the value of x was being found, the variable a was overwritten with $\perp$. *(LookupBad)* does not bother with this and introduces a space leak.

$$\{a \mapsto x\}_{a,s} \quad \longrightarrow \quad \{t \mapsto \#a\ s, a \mapsto x\}_{x,t} \quad (\mathit{LookupBad})$$

Renaming the left pattern with $[t/s]$ and applying the substitution $[a/x]$ to both sides we get; $\{a \mapsto a\}a, t \subset \{t \mapsto \#a\ s, a \mapsto a\}a, t$. This tells us that the graphs matching $\{a \mapsto a\}a, t$ will evaluate by repeatedly looking up a, allocating a new stack node at each iteration, resulting in unbounded space usage. This does not happen on $\boxed{\mathrm{R}}$ as black holing prevents the loop being followed. □

The programs recognised by this self-feed test have no value so it is perhaps difficult to say how dangerous the leaky evaluator is in practice (on programs that do have values). In general it seems unlikely that a space fault could only affect non-terminating programs. It is well known that black holing is necessary to gain the expected space complexity of many programs [Jon92].

We are searching for the smallest leak witness because that should pinpoint the source of the space fault most directly. Also it makes the search procedure, which will itself be non-terminating (and doomed to failure where the smallest witness is quite large), more likely to come up with an answer. Alternatively, we could count the number of leak witnesses per n super-rules as an absolute measure of leakiness: if there is a self-feeding leak but we cannot find it after searching millions of super-rules then the leak may not be very serious in practice.

Sections 4 to 6 describe the components of the brute-force search procedure, Sections 7 and 8 look at some refinements which improve its effectiveness.

4 Super-Rule Construction

We need to find the most general redex and result of a sequence of two rewrite rules r_1 and r_2. That is, we want a rule which has the same effect as applying r_1 then r_2: we write this as $\mathit{super}(r_1, r_2)$. Super-rules corresponding to longer rule sequences can then be constructed by repeatedly applying *super*, so $\mathit{superRule}\langle r_1, r_2, r_3 \rangle = \mathit{super}(\mathit{super}(r_1, r_2), r_3) = \mathit{super}(r_1, \mathit{super}(r_2, r_3))$.

super may have no solutions because the right pattern of r_1 might never match the left pattern of r_2. It may also have many solutions, as discussed in section 5. So *super* must provide a complete set of super-rules which replace r_1 followed by r_2 exactly.

Roughly, super-rule construction works as follows. Given $r_1 = L_1 \longrightarrow R_1$ and $r_2 = L_2 \longrightarrow R_2$, first we form a most general mid-point by unifying R_1 and L_2. We get a substitution S such that any address in both $S(R_1)$ and $S(R_1)$ maps to the same term pattern and the pattern roots are identical.

Example 3 (*super*(*Let*, *Let*))
Unifying the right pattern of *(Let)* with a renaming of its left pattern gives the graph pattern $MID = \{a \mapsto let\ z = A[b/y]\ in\ B[b/y], b \mapsto E[b/y]\}a, s$. The unifier is $S = (\theta, \phi)$ where $\phi = [\,]$ and $\theta = [(let\ z = A\ in\ B)/X, A[b/y]/A, B[b/y]/B]$.

Now the unifier S is applied to L_1 and L_2. Any nodes in $S(L_2)$ that are not already in, $S(R_1)$ are added to $S(L_1)$ to form the super-rule left pattern. Similarly, the super-rule right pattern is formed from $S(R_2)$ plus any nodes in $S(R_1)$ that are not already in $S(L_2)$. Any nodes allocated by R_1 which cannot be live (in any context) in the new right pattern are removed. For (Let, Let) we get the rule below.

$$\{a \mapsto let\,y = E\ in\ (let\ z = A\ in\ B)\}a, s \quad \longrightarrow \quad \{a \mapsto B[c/z, b/y], c \mapsto A[c/z, b/y], b \mapsto E[b/y]\}a, s$$

□

super preserves the following properties of graph rewrite rules. The left pattern is linear in holes, no hole has a substitution and all its nodes are reachable from its roots. The right pattern may be non-linear, holes may have substitutions and it can be disconnected. All nodes in the right pattern but not in the left are allocated by the rule: they are distinct from all existing nodes and from each other. Any nodes in the left pattern but not in the right are deallocated by the rule. Further, all nodes in the left pattern are assumed to be distinct (though we will have cause to relax this restriction later). There are two changes we must make to the original rewrite rule specification [BR00]. Left patterns may now be non-linear in variables, and patterns are extended with constraint sets which express required inequalities amoungst their variables, so *(Update)* can be written as below. These constraint sets can be quadratic in the number of nodes in the pattern but in practice they do not get quite so large because nodes that belong to different syntactic categories can never be equal: $a \neq s$ and $y \neq s$ are implicit in *(Update)*.

$$\{a \mapsto \lambda x.E, y \mapsto X, s \mapsto \#y\ t\}a, s||\{a \neq y\} \quad \longrightarrow \quad \{y \mapsto \lambda x.E, a \mapsto \lambda x.E\}y, t||\{a \neq y\} \quad (Update)$$

5 Multiple Super-Rules

As well as taking the variable disequality constraints into consideration, the super-rule problem can produce multiple solutions. The pattern unification (or rather, *disunification* [BB94] — unification with disequalities) procedure must generate a set of solutions. Examples 4 and 5 illustrate the cases that cause multiple solutions.

Example 4 (Holes with substitutions)
Unifying the patterns $H[x/y]$ and z demands multiple solutions because either the substitution has an effect making hole $H = y$ and the variable $x = z$ or the substitution has no effect and $H = z, z \neq y$. To understand why both solutions are needed, consider constructing the *super*(*Let*, *Lookup*). We must disunify the patterns $\{a \mapsto X[b/y], b \mapsto E[b/y]\}a, s$ and $\{a \mapsto x\}a, s$. This requires a solution to the equation $X[b/y] = x$. So either $X = y$ where the variable looked up is the one just allocated or the variable is one already present elsewhere in the graph and $X = x$. Therefore we generate the two super-rules below.

$$\{a \mapsto let\ c = A\ in\ (g)\}a, b||\{g \neq c\} \longrightarrow \{a \mapsto \bot, h \mapsto \#a\,b, d \mapsto A[d/c]\}g, h||\{g \neq c, a \neq h, a \neq d, h \neq d\}$$
$$\{a \mapsto let\ c = A\ in\ (c)\}a, b \longrightarrow \{a \mapsto \bot, h \mapsto \#a\,b, d \mapsto A[d/c]\}d, h||\{a \neq h, a \neq d, h \neq d\}$$

□

In general, when a disunification requires a solution to $H\theta = P$, there are as many solutions as there are ways of partitioning the free variables of P into $|\theta| + 1$ sets. Fortunately, we never have to solve the unification problem $H[x/y] = H'[u/v]$ because we always disunify a left pattern with a right pattern.

Example 5 (Equalities between left-pattern nodes)
The basic super-rule construction procedure for *super*(*Push*, *LookupBad*) gives us (4). This is not quite right because it leaves $a = f$ or $a \neq f$ unspecified, and yet it does not quite represent both possibilities properly.

$$\{a \mapsto f\,b, f \mapsto X\}a, s||\{\} \longrightarrow \{a \mapsto \bot\,b, c \mapsto f, f \mapsto X, u \mapsto \#c\,t, t \mapsto a : s\}f, u||\{a \neq c, c \neq f, u \neq t\} \quad (4)$$

If $a = f$ then $X = (f\ b)$ but if we substitute this into the right pattern of (4) then there are two nodes with address a which map to different terms. But it is possible to evaluate *(Push)* then *(Lookup)* in the case where $a = f$, we just need to generate both solutions explicitly by continuing the disunification with either $a = f$ or $a \neq f$. This gives the correct result, shown below. This modification can generate a number of solutions exponential in the pattern size.

$$\begin{array}{l} \{a \mapsto f\ b, f \mapsto X\}a, s||\{a \neq f\} \\ \longrightarrow \{a \mapsto \perp b, c \mapsto f, f \mapsto X, u \mapsto \#c\ t, t \mapsto a : s\}f, u||\{a \neq c, c \neq f, a \neq f, u \neq t\} \\ \{a \mapsto a\ b\}a, s||\{\} \longrightarrow \{a \mapsto \perp b, c \mapsto a, u \mapsto \#c\ t, t \mapsto a : s\}f, u||\{a \neq c, u \neq t\} \end{array} \quad (5)$$

□

6 Simple Search Strategy

Generating all rule sequences then finding their super-rules then testing if they can self-feed and finally testing whether they allocate produces a list of candidate leak witnesses.

The allocation test is only an approximation because we cannot always tell whether the increase in space usage will survive a garbage collection — it depends on the context. So we define the change in space caused by a rule as a range and then we can take the lower limit as a safe approximation, or the upper limit if more solutions are preferred.

Example 6 (Finding a leak in a stack-like scoping evaluator)
Defining $\boxed{\text{I}}$ as $\boxed{\text{R}}$ but replacing *(ReduceGood)* with *(ReduceBad)* and adding *(Return)* gives us a semantics that has a space leak. Now, when a function body is entered by *(Reduce)*, it pushes a *Ret* stack term which is not popped until the value of that function body is found (category S is extended with these *Ret x s* function symbols).

$$\begin{array}{rcll} \{a \mapsto \lambda x.E, s \mapsto b; t, b \mapsto F\ y\}a, s & \longrightarrow & \{b \mapsto E[y/x], s \mapsto \mathit{Ret}\ b\ t, a \mapsto \lambda x.E\}b, s & (\mathit{ReduceBad}) \\ \{a \mapsto \lambda x.E, s \mapsto \mathit{Ret}\ f\ t\}a, s & \longrightarrow & \{a \mapsto \lambda x.E\}a, t & (\mathit{Return}) \end{array}$$

The smallest candidate witness produced is: $\{f \mapsto \lambda x.f\ \epsilon, a \mapsto f\ \epsilon\}a, \epsilon$, from the rule sequence $(\mathit{Push}, \mathit{Lookup}, \mathit{Update}, \mathit{ReduceBad})$. This accumulates a *Ret* node at each iteration. Coincidentally, this rule sequence is the smallest self-feeding loop on $\boxed{\text{R}}$, which runs in constant space. □

7 Going Faster

To search more quickly we use the idea of *proof planning* [Bun88] to restrict the search space without losing possible solutions. There are a number of improvements we can make to the simple rule sequence generation process before the (much more complex) super-rule generator is applied.

Firstly, we only need to consider rule sequences that could occur. The super-rule generator will only succeed if the whole rule sequence can occur, but we can use a *next rule table* to avoid generating any rule sequences which is obviously impossible because rule x can never follow rule y.

Example 7 (Next rule table of $\boxed{\text{R}}$)
For our example evaluator $\boxed{\text{R}}$ (and its variants) the next rule table is shown below. It is generated simply by disunifying the right pattern of the first rule with the left pattern of the second. It is quite dense, 20 out of a possible 25 rule sequences may occur.

First Rule	*Second Rule*				
	Update	*Lookup*	*Reduce*	*Push*	*Let*
Update	*1*		*1*		
Lookup	*1*	*1*		*1*	*1*
Reduce	*1*	*1*	*1*	*1*	*1*
Push		*1*	*1*	*1*	*1*
Let	*1*	*1*	*1*	*1*	*1*

We only generate rule sequences $\langle r_1, \ldots, r_n \rangle$ such that r_{i+1} may follow r_i and r_1 may follow r_n. If we search all sequences up to length 7 on $\boxed{\text{I}}$, this cuts the number of sequences from 97,655 to 21,851. □

Further processing of the next rule table may be appropriate to remove any tributaries and leave only strongly-connected cycles, since we are only interested in rules that are reachable from themselves. The brute force search will still generate all permutation loops of a super-rule. For example, if $\langle a, b, c \rangle$ form an allocating and self-feeding super-rule then so do $\langle b, c, a \rangle$ and $\langle c, a, b \rangle$. These variants can be eliminated at the sequence generation stage, the disadvantage being that we will not always get the loop variant which is easiest to read.

Example 8 (Removing permutation loops in $\boxed{\text{I}}$)
Adding this filter to our search up to length 7 in $\boxed{\text{R}}$ cuts the number of sequences down to 3,367. For $\boxed{\text{I}}$ in Example 6, instead of getting all 4 versions of the witness (10 taking multiple solutions into account, and 58 if we also take all results of the self-feed test) we only get the 2 super-rules for the (alphabetically least) permutation of those rules, shown below.

$$\begin{array}{ll}
\multicolumn{2}{l}{\{a \mapsto i, i \mapsto \lambda c.i\ g, b \mapsto w; z, w \mapsto \perp g\}a, b \| \{w \neq a, a \neq i, w \neq i, g \neq c\}} \\
\longrightarrow & \{s \mapsto i, t \mapsto w; b, w \mapsto \perp g, b \mapsto \mathit{Ret}\ w\ z, a \mapsto \lambda c.i\ g, i \mapsto \lambda c.i\ g\}s, t \\
& \| \{s \neq w, w \neq a, a \neq i, w \neq i, g \neq c, s \neq a, s \neq i, t \neq b\}(\mathit{LookupGood}, \mathit{Update}, \mathit{ReduceBad}, \mathit{Push}) \\
\multicolumn{2}{l}{\{a \mapsto h, h \mapsto \lambda s.h\ s, b \mapsto u; w, u \mapsto \perp d\}a, b \| \{u \neq a, a \neq h, u \neq h\}} \\
\longrightarrow & \{q \mapsto h, r \mapsto u; b, u \mapsto \perp d, b \mapsto \mathit{Ret}\ u\ w, a \mapsto \lambda s.h\ s, h \mapsto \lambda s.h\ s\}q, r \\
& \| \{q \neq u, u \neq a, a \neq h, u \neq h, q \neq a, q \neq h, r \neq b\}(\mathit{LookupGood}, \mathit{Update}, \mathit{ReduceBad}, \mathit{Push})
\end{array}$$

□

Finally, we are only interested in super-rules which allocate. Another simple approximation rules out any rule sequence that cannot ever allocate. This brings the search space to depth 7 in $\boxed{\text{R}}$ down to 2,649 sequences.

8 Avoiding Multiple Solutions

The exact super-rule construction algorithm can generate a lot of solutions for certain combinations of rules, particularly where there are many hole substitutions or disconnected right-hand patterns. An approximate version of the disunification algorithm which generalises at the points where the exact version produces multiple solutions gives only one super-rule for each rule sequence. Furthermore, if the approximate rule is not self-feeding then the exact verion cannot be self-feeding either. Adding the approximate versions as filters before generating the exact solution cuts the search space down much further. In our search to depth 7 on $\boxed{\text{R}}$, there are 2,179 sequences which have an approximate super-rule out of which only 31 can self-feed. The exact super-rule generator produces 357 rules from these sequences.

9 Conclusion and Further Work

The primary shortcoming of the work presented here is that it only generates *candidate* leak witnesses: it produces programs that are allocating self-feeding loops on $\boxed{\text{I}}$, but does not say anything about their behaviour on $\boxed{\text{R}}$. To solve this problem we need to decide if any instance of our candidate witnesses can run in constant space on $\boxed{\text{R}}$. For examples like the ones in this paper, it will be sufficient to prove that either: the candidate cannot begin a self-feeding loop on $\boxed{\text{R}}$, as in Example 2; or that the cadidate forms a self-feeding but non-allocating loop, as in Example 6.

There is also plenty of scope for improvement in the search procedure: sharing disunifications and using next rule tables with a lookahead of more than one step could give constant-factor time improvements, at the cost of greater space usage.

References

[BB94] W L Buntine and H-J Bürckert. On solving equations and disequations. *Journal of the ACM*, 41(4):591–629, July 1994.

[BR00] A Bakewell and C Runciman. The space usage problem: An evaluation kit for graph-reduction semantics. In S Gilmore, editor, *Proc. 2nd Scottish Functional Programming Workshop, School of Computer Science, University of St. Andrews*, pages 1–18, July 2000.

[Bun88] A Bundy. The use of explicit plans to guide inductive proofs. In *CADE-9*, volume 310 of *LNCS*, pages 111–120, 1988.

[Jon92] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–80, January 1992.