The Space Usage Problem: An Evaluation Kit for Graph-Reduction Semantics

Adam Bakewell and Colin Runciman*

October 2, 2000

Abstract

We describe a software tool for specifying operational semantics as a term-graph reduction system. The semantics are guaranteed to model the asymptotic space and time usage of an implementation accurately yet are abstract enough to support reasoning at the program level.

Term graphs make all the aspects of reduction relating to space usage explicit as they naturally encode size and address information. The semantics are constrained to be small-step, deterministic rules and each rewrite step uses bounded resources. The result is a system suitable for describing and analysing the space behaviour of functional programming languages.

1 Introduction

Functional programmers often have difficulty in understanding how their programs will be evaluated. Under-specification of languages may mean little help is available. There might only be a guide to the general strategy, e.g. "Haskell is a non-strict, purely functional programming language" or there might be a detailed, albeit non-deterministic, specification as for Scheme [Cli98]. A reference *space semantics* would be a useful part of any language specification, offering programmers an explanation of how their programs are executed and a guide to how much space they need. It would also be

*Dept. of Computer Science, University of York, UK.Email: {ajb,colin}@cs.york.ac.uk

a reference space model for implementors, though they do not have to implement the semantics directly. This would remove some of the uncertainty about space usage in functional languages: when there appears to be a *space leak* we can compare reality with the reference model to decide whether the program, implementation or language is at fault. It also helps us reason in the abstract about the behaviour of programs and the effect changing them has on resource usage.

Our approach uses term graphs to model space usage in the most obvious way. We restrict semantics to the small-step, structural style. This does not have to be as low-level as an abstract machine specification, so we can understand evaluation works without learning the details of an implementation. A more abstract big-step natural semantics tends to hide information about space usage.

Example 1 (Disadvantages of natural semantics)

Consider the usual operational semantics rules, *(Lam)* and *(App)*, which define the relation $\Downarrow$ for call-by-value evaluation of λ-calculus terms.

$$(Lam)\ \lambda x.E \Downarrow \lambda x.E \quad (App)\ \frac{E \Downarrow \lambda x.E'' \quad E' \Downarrow V \quad E''[V/x] \Downarrow V'}{E\ E' \Downarrow V'}$$

These rules tell us how to construct a proof tree to find the value of a term. But they do not really tell us how much space we need to do the proof. In *(App)* the order of premise evaluation is not specified; a variety of substitution methods could be used and various degrees of sharing would fit the specification. A real implementation would also need continuations or a stack; it is not clear how the space for this relates to the semantics. □

This paper describes a tool kit for defining and analysing a space semantics. The kit takes a description of a language, a graph *evaluator* (operational semantics) and an initial graph, checks various properties of these components and then evaluates the graph. This paper is an abstract presentation of the techniques used in the kit, rather than a manual, as the kit is primarily a research tool.

Section 2 develops a term-graph model for evaluator state which allows sharing and copying to be expressed easily. The measurement of size and a language-independent garbage collector are introduced. Section 3 describes the construction of evaluators using a style which allows the convenience of substitution into sub-terms. This is supported by a basic type system to aid checking and analysis. The semantics are constrained to be deterministic and we ensure that they only generate valid graphs. Section 4 discusses an analysis that ensures the semantics are a valid model of space and time

usage by bounding the work at each step to guarantee a *space valid* model. Other issues relating to reducing the garbage collection overhead and giving the semantics more control over space usage are considered. Section 5 looks at related work and Section 6 concludes. Throughout the paper proofs are sketched or omitted to save space.

2 Term Grammars and Graphs

2.1 Term Grammars

A programming language is described by a term grammar $\mathcal{G}$ from which program terms are built. Each term belongs to a category in the grammar. We will use the category information later as a simple type-system to aid automatic checking.

Definition 1 (Term grammar)
A term grammar $\mathcal{G}$ is a set of syntactic category definitions. Each category C is a set of function symbol definitions. A function symbol F may bind b variables and have n arguments. Each argument of F must belong to a specified category. Either it must be a term and we write its category in upper-case or it must be a variable (addressing a term of that category, an arc in the graph) and we write its category in lower-case. A bound variable stands for an arc; its category must also be specified. □

Example 2 (Grammar)

Category	Term notation	Shorthand notation
X ::=	$LAM\ x.X$	$\lambda x.X$
|	$APP\ X\ X'$	$X\ X'$
|	$VAR\ x$	x

We define the λ-calculus as category X. A dot separates the bound variables from the arguments of a function symbol, unless it binds no variables like APP. The usual shorthand notation is also given, but the term definitions are important for defining properties like *size* later. □

Terms are constructed from a grammar in the obvious way. Variables must have the correct categories at all occurrences and sub-terms must either be variables or terms according to the grammar.

$$\frac{C \in \mathcal{G} \quad (F\ c_1 \cdots c_b.A_1 \cdots A_a) \in C \quad \Delta = \{x_i \mapsto c_i\}_{i=1}^{b} \cup (\Gamma|_{\overline{\{x_1,\ldots,x_b\}}}) \quad (A_i = c') \Rightarrow \Delta(T_i) = c' \quad (A_i = C') \Rightarrow \mathcal{G}, \Delta \vdash T_i : C'}{\mathcal{G}, \Gamma \vdash (F\ x_1 \cdots x_b.T_1 \cdots T_a) : C} \quad (1)$$

More formally, T is a term of category $C \in \mathcal{G}$, written $T : C$, if it can be assigned a category by (1). So there must be a mapping $\Gamma : \mathit{Var} \to \mathcal{G}$ which includes all the free variables of T and $\mathcal{G}, \Gamma \vdash T : C$. Any variables bound at a function symbol must be uniquely named but they may have the same name as variables bound elsewhere.

Example 3 (Term)
The category X term $(\lambda x.x\ x)\ (\lambda x.x\ x)$, written in the shorthand notation, is a simple example of a non-terminating program. It cannot be written in a conventional language with a Milner-style type system but we can investigate its operational behaviour in our weakly typed system. □

Definition 2 (Free variables, substitution)
$FV(T)$ are the free variables of T. A substitution $\theta : \mathit{Var} \to \mathit{Var}$ must be category-preserving: $\theta(x) : C \Leftrightarrow x : C$ so that $T : C \Leftrightarrow T\theta : C$.
$FV(F\ x_1 \cdots x_b.T_1 \cdots T_a) = \bigcup_{i=1}^{a} FV(T_i) \setminus \{x_i\}_{i=1}^{b}$
$FV(x) = \{x\}$
$(F\ x_1 \cdots x_b.T_1 \cdots T_a)\theta = (F\ x_1 \cdots x_b.T_1\theta' \cdots T_a\theta')$ where $\theta' = \theta|_{\overline{\{x_1,\ldots,x_b\}}}$
$x\theta = \theta(x)$, if $x \in \mathit{dom}\ \theta$; $x\theta = x$, otherwise. □

2.2 Term Graphs

A term graph is a set of addressed nodes, a mapping from variables to terms. A node $N = (x \mapsto T)$ in graph G has address x and term T and $G(x) = T$. Graphs are our model of state — a set of addressed locations of unspecified size — and we will define evaluators as functions on them. To guide an evaluator, graphs must have roots: a set of variables which address the current evaluation position. The categories of the roots are specified by $\mathcal{G}(\mathit{ROOT})$.

Example 4 (Grammar with root)
$\mathcal{G}_\lambda(\mathit{ROOT}) = \langle x, s \rangle$
S ::= $\mathit{ARG}\ x\ s$ (written $x : s$) | $\mathit{FUN}\ x\ s$ (written $x; s$)

To evaluate terms built from X using the call-by-value strategy, we define the graph grammar $\mathcal{G}_\lambda = \{X, S\}$ where S defines stack terms which are used to preserve the context while a sub-expression is evaluated. $\mathcal{G}_\lambda$ graphs have a current expression root and a current stack node root. □

Definition 3 (Rooted graph)

$$\frac{\begin{array}{c}\mathcal{G}(\mathit{ROOT}) = \langle c_1, \ldots, c_r \rangle \\ \exists \Delta \cdot \{v_i \mapsto c_i\}_{i=1}^{r} \subseteq \Delta \wedge G(x) = T \Rightarrow \mathcal{G}, \Delta \cup \Gamma|_{\overline{dom\ G}} \vdash T : \Delta(x)\end{array}}{\Gamma \vdash G_{v_1,\ldots,v_r} \in \mathit{Graph}(\mathcal{G})} \quad (2)$$

The rooted graphs of $\mathcal{G}$, $Graph(\mathcal{G})$ includes any graph whose terms can be consistently categorised according to the grammar. So there is a mapping $\Gamma : Var \to \mathcal{G}$ such that (2) holds. The root sequence is written to the right of a graph and roots must address graph nodes. □

The special constant ϵ may appear anywhere in a term graph where a free variable or root can. It stands for a null arc in the graph. It belongs to every category and cannot be bound by any term or be used as an address.

Example 5 (Rooted Graph)
To evaluate the non-terminating term from Example 3 we begin with the graph $\{a \mapsto (\lambda x.x\, x)\, (\lambda x.x\, x)\}_{a,\epsilon}$. It has one node with address a containing the term, pointed to by the expression root. The stack root is null. □

Definition 4 (Free variables and well-formed graphs)
$FV(G) = \Gamma$, where Γ is the smallest mapping such that $\Gamma \vdash G \in Graph(\mathcal{G})$.
$G \in Graph(\mathcal{G})$ is well-formed if $FV(G) = \emptyset$. □

We extend the definitions of free variables to graphs, following the graph literature [AA95]. Sometimes heap-node addresses are treated as free variables in operational semantics [Ses97]. Both styles have their advantages. Well-formed (closed) graphs are important because the absence of dangling arcs means a graph includes all the (non-constant) space needed by an evaluation system. Also, an incomplete graph might fail to evaluate properly.

2.3 Garbage Collection and Graph Size

Only nodes on a path from the roots of a graph can ever be used by an evaluator. A language-independent garbage collector *gc* removes any unreachable nodes from a rooted graph. This is limiting in that we cannot yet reason about different garbage collection strategies, but there is scope for providing a framework for specifying graph collectors too. If our model is applied to define the asymptotic space usage of a language then we are not excluding implementations that work in a different way but we expect them to have the same asymptotic space and time usage for all programs. An example is *regions* [TT94] where a global analysis eliminates all garbage collection.

Definition 5 (Reachable nodes,Garbage collection)
The set $reach(G, X)$ contains node addresses reachable in G starting from those in X. All nodes unreachable from the roots are removed by gc.
$reach(G, X) = fix\ (\lambda V.X \cup V \cup FV(G|_V))$
$gc(G_V) = (G|_{reach(G,V)})_V$ □

Proposition 1 (Garbage collection preserves well-formedness)
If G_V is well-formed and $G'_V = gc(G_V)$ then G'_V is well-formed.
Further, if $G'' \subset G'$ then G''_V is not well-formed. □

The *size* of terms and graphs are defined in the obvious way; the space needed by G is $size(gc(G))$. This definition is a valid asymptotic measure which assumes the graphs are well-formed, the grammar is finite (so the function symbol size is limited) and we have an infinite set of variables which each occupy a unit of space. Node addresses in the graph domain do not take space, they are like the addresses of computer memory cells. A language-independent size measure is important for making a fair comparison of different language encodings.

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

2.4 Term grammar checking

The kit takes an input file containing grammar, evaluator and initial graph definitions. First it checks that the grammar is valid and then that the graph is a well-formed graph according to the grammar. The validated versions are displayed using formatting strings supplied in the input file, we give one for Latex documentation a second for ASCII display. The definition of S in Example 4 was generated from the lines below. When a term is displayed, the strings `#1` and `#2` are replaced by the corresponding sub-term.

```
S ::= ARG x s      "#1 : #2"      "Arg #1 #2"
    | FUN x s      "#1 ; #2"      "Fun #1 #2"
```

Checking a grammar is valid — that it has no free category names and that categories and functions are uniquely named — is very similar to checking a graph is well-formed. Hence we convert the grammar to a graph which has a node for each category and each function symbol. A node's address is the category or function name; its term is a list of the functions in the category or the categories that parametrise the function. For $\mathcal{G}_\lambda$ this gives us GG_λ.

$$GG_\lambda = \left\{ \begin{array}{c} ROOT \mapsto [X, S], \\ S \mapsto [ARG, FUN], X \mapsto [LAM, APP, VAR], \\ ARG \mapsto [X, S], FUN \mapsto [X, S], \\ LAM \mapsto [X, X], APP \mapsto [X, X], VAR \mapsto [X] \end{array} \right\}_{ROOT}$$

Garbage collecting these grammar graphs is another useful check; we have discovered redundant categories (indicating a mistake in the input) this way.

3 Term-graph Evaluation

3.1 Patterns

Evaluators are defined as sets of rewrite rules, where each rule is a pair of *patterns*. Patterns are *graph contexts*: they are built from graph grammars extended with labelled *holes*. The rules for checking patterns are more complicated than the term rule because of the differing constraints on left and right patterns and the presence of holes.

Finding the category of a hole means we must introduce a category variable α into our type system. A hole environment H records the category of each occurrence of each hole. When two sub-term environments are combined at a term, the categories of the same hole must have a greatest common instance in the ordering $C \sqsubset \alpha$ where α is a category variable and C is any category — they must *unify* as in normal type checking. The operator $\oplus$ defined by (3) combines two hole environments in this way. Hole environments also record β, the set of variables bound in the term pattern around the hole, and θ, the optional substitution, for checking later.

$$\begin{array}{l} H \oplus H' = H|_{\overline{dom\ H'}} \left\langle \begin{array}{l|l} x \mapsto (c \sqcap c', \beta, \theta), & H(x) = (c, \beta, \theta) \wedge \\ x \mapsto (c \sqcap c', \beta', \theta') & H'(x) = (c', \beta', \theta') \end{array} \right\rangle \\ \qquad + H'|_{\overline{dom\ H}} + \\ F \oplus F' = F + F' \text{ where } F(x) = (C, \beta) \wedge F'(x) = (C', \beta') \Rightarrow C = C' \\ B \oplus B' = B + B' \text{ where } B(x) = C \wedge B'(x) = C' \Rightarrow C = C' \end{array} \tag{3}$$

Pattern checking also records the categories of bound and free variables in environments B and F. Surrounding bound variable information is also held in F. An analogous $\oplus$ function combines these environments though no unification is needed. For convenience we define $\bigoplus_1^n S_i = S_1 \oplus \cdots \oplus S_n$.

Definition 7 (Term Pattern)
Patterns are checked by judgements of the form $\mathcal{G}, \beta \vdash pat\ P : C, B, F, H$ which says P is a term pattern in the context of grammar $\mathcal{G}$ and a set of variables β bound in the surrounding pattern. P matches terms of category C. B, F and H are bound, free and hole variable environments; the sequences mapping variables and holes to their categories and other details. Bound, free and hole variables must be distinctly named. By convention we write variables in lower-case and holes in upper-case to match the grammar definition style.

A hole (4) *is a pattern which matches any term. Its category is a variable* α*, recorded along with* β *and substitution* θ *which will be applied to a term matching* H*.*

$$\mathcal{G}, \beta \vdash pat\ H\theta : \alpha, \langle\rangle, \langle\rangle, \langle H \mapsto (\alpha, \beta, \theta)\rangle \tag{4}$$

A constructed pattern (5) *of category* C *binds the appropriate number of distinctly-named variables. If the* i*th argument of* F *must be a variable then pattern* P_i *must be a variable, it is recorded in the sequence* F_i*. If it must be a term then* P_i *must be a pattern of the appropriate category; this is checked in the context of the new bound variables* Δ*.*

$$\frac{\begin{array}{c} C \in \mathcal{G} \wedge (F\ c_1 \cdots c_b.A_1 \cdots A_a) \in C \\ \Delta = \langle x_i \mapsto c_i \rangle_{i=1}^{b} \quad dom\ \beta \cap dom\ \Delta = \emptyset \\ (A_i = c') \Rightarrow P_i = x \wedge F_i = \langle x \mapsto (C', \beta + \Delta)\rangle \wedge B_i = H_i = \langle\rangle \\ (A_i = C') \Rightarrow \mathcal{G}, \beta + \Delta \vdash P_i : C', B_i, F_i, H_i \end{array}}{\mathcal{G}, \beta \vdash pat(F\ x_1 \cdots x_b.P_1 \cdots P_a) : C, \Delta \oplus \bigoplus_1^a B_i, \bigoplus_1^a F_i, \bigoplus_1^a H_i} \tag{5}$$

□

Example 6 (Term pattern)
$\lambda y.B$ is a term pattern where y matches a variable and B is a hole matching any term, which may include occurrences of the variable matching y. $B[x/y]$ is simply a hole; it matches a term in which all occurrences of the variable matching y are replaced by the variable matching x. □

Now we can define the left and right patterns that comprise a rule. Left patterns must be fully connected and linear in all the holes and variables in their terms to ensure that pattern matching takes bounded time. Non-linear patterns would actually be workable in our framework without completely losing the time bound but they are an unnecessary complication. Unconnected patterns are useful in proof systems but they would make matching time depend on graph size. A right pattern is constrained by its left pattern definition; it must not free any previously bound variable which is why holes often need substitutions. It does not need the linearity constraints of a left pattern — copying and sharing are common in rewrite steps.

Definition 8 (Left and right patterns)
$L = (\{x_i \mapsto P_i\}_{i=1}^{n})_{r_1,\ldots,r_m}$ *is a pattern with environments* B, F, H *if:*
$\mathcal{G}, \emptyset \vdash P_i : C_i', B_i, F_i, H_i$. *Each* P_i *is a term pattern.*
$B = \bigoplus_1^n B_i, H = \bigoplus_1^n H_i$. *Variable and hole environments combine.*
$F = \langle r_i \mapsto (c_i, \emptyset)\rangle_{i=1}^{m} \oplus \bigoplus_1^n F_i$ *where* $\mathcal{G}(ROOT) = \langle c_1, \ldots, c_m\rangle$.
$F(x_i) = (C, \beta) \Rightarrow C \sqsubseteq C_i'$. *Pattern and address categories match.*

Pattern L with environments B, F, H is a left pattern, $L \in LhsPat(\mathcal{G})$ if:

$L = gc(L)$.	*L must be fully connected.*
$H(x) = (C, \beta, \theta) \Rightarrow \theta = \emptyset$.	*No hole has a substitution.*
$\lvert(B + F + H)\rvert_{\{x\}} \leq 1$.	*L is linear in all variables and holes.*

R is a right pattern of L, $R \in RhsPat(\mathcal{G}, L)$ if:

R is a pattern with environments B', F', H', L has environments B, F, H.

$B'(x) = C \Rightarrow B(x) = C$	*no new variables are bound in R.*
$F'(x) = (C, \beta') \Rightarrow x \in dom\ R \setminus dom\ L$	*Either x is allocated by R*
$\vee F(x) = (C, \beta) \wedge \beta' \supseteq \beta$.	*or x occured in L.*
$dom\ R \supseteq dom\ L$.	*No nodes are deallocated by R.*

$H'(h) = (C', \beta', \theta) \Rightarrow H'(h) = (C, \beta, \emptyset) \wedge C \sqsubseteq C'$.Holes retain their category and θ maps variables in $\beta \setminus \beta'$ (those freed in R) to variables in F or F' of the same category which cannot themselves be freed in the context of β'. □

Example 7 (Left and right patterns)
$L = \{a \mapsto \lambda x.E, f \mapsto \lambda y.B, s \mapsto f; t\}_{a,s} \in LhsPat(\mathcal{G}_\lambda)$. All nodes in L are reachable from the roots a and s. E and B are holes and a term matching B could contain the variable y.
$R = \{a \mapsto \lambda x.E, f \mapsto B[a/y], s \mapsto f; t\}_{f,t} \in RhsPat(\mathcal{G}_\lambda, L)$. Nodes s and a are unchanged and f contains B with a substitution to replace the bound variable y with a (the address of a category X node).

Together the patterns make a rule which applies the function value at f, pointed to by the stack node s, to the argument value at a. The argument is shared amongst all occurrences because only its address is substituted. The rule leaves the expression root pointing at the specialised function body and the stack root pointing at the next node in the stack chain. Normally the garbage collector will remove s. □

The definitions of free variables and *size* extend to patterns according to the rules $FV(h\theta) = \emptyset$ and $size(h\theta) = 0$. Analogous to FV, $Holes(P)$ gives the sequence of holes occuring in pattern P — the domain of the H environment of P. A pattern is mapped to a graph by *match*, a special substitution which replaces all variables and holes.

Definition 9 (Pattern matching)
Let ψ be a substitution, ϕ be a mapping $\phi : Hole \to Term(\mathcal{G})$ and P be a pattern. Then match produces a graph by applying ψ to all the variables in P and ϕ to the holes. If a hole has a substitution vector θ, it is mapped to a substitution by ψ and applied to the filled hole.

$$
\begin{array}{lcl}
\mathit{match}\ \psi\ \phi\ x & = & \psi(x) \\
\mathit{match}\ \psi\ \phi\ (H[x_i/y_i]_{i=1}^n) & = & [\psi(x_i)/\psi(y_i)]_{i=1}^n(\phi(H)) \\
\mathit{match}\ \psi\ \phi\ (F\ x_1 \cdots x_b.P_1 \cdots P_a) & = & F\ \psi(x_1) \cdots \psi(x_b). \\
 & & (\mathit{match}\ \psi\ \phi\ P_1) \cdots (\mathit{match}\ \psi\ \phi\ P_a) \\
\mathit{match}\ \psi\ \phi\ \{a_i \mapsto P_i\}_{i=1}^n & = & \{\psi(a_i) \mapsto \mathit{match}\ \psi\ \phi\ P_i\}_{i=1}^n \\
\mathit{match}\ \psi\ \phi\ G_{v_1,\ldots,v_n} & = & (\mathit{match}\ \psi\ \phi\ G)_{\psi(v_1),\ldots,\psi(v_n)}
\end{array}
\qquad \square
$$

Note that *match* is a kind of graph isomorphism (a homomorphism is usual in graph rewriting). The pattern $\{x \mapsto \mathit{VAR}\ y, y \mapsto T\}_{x,z}$ does not match the graph $\{a \mapsto \mathit{VAR}\ a\}_{a,\epsilon}$. The mappings $\psi = \{a/x, a/y, \epsilon/z\}$ and $\phi = \{\mathit{VAR}\ a/T\}$ could give a match under a different definition but then a right pattern might try to update the same node with two different terms.

Proposition 2 (Pattern matching gives valid graphs)
$L \in \mathit{LhsPat}(\mathcal{G}) \wedge \mathit{match}\ \psi\ \phi\ L = G \Rightarrow \exists \Gamma \cdot \Gamma \vdash G \in \mathit{Graph}(\mathcal{G})$.
Further, $R \in \mathit{RhsPat}(\mathcal{G}, L) \wedge \mathit{match}\ \psi\ \phi\ R = G' \Rightarrow \Gamma \vdash G' \in \mathit{Graph}(\mathcal{G})$. $\square$

Example 8 (Pattern matching)
Let $\psi = [a/a, x/x, c/f, x/y, b/s, \epsilon/t]$ and $\phi = [(x\ x)/E, (x\ x)/B]$.
Using L and R from Example 7:
$\mathit{match}\ \psi\ \phi\ L = \{a = \lambda x.(x\ x), c = \lambda x.(x\ x), b = c; \epsilon\}_{a,b}$
$\mathit{match}\ \psi\ \phi\ R = \{a = c\ c, c = \lambda x.(x\ x), b = c; \epsilon\}_{a,\epsilon}$ $\square$

3.2 Evaluators

Definition 10 (Evaluator)
An evaluator $\boxed{A}$ *is a finite set of graph rewrite rules* $\{L_i \longrightarrow R_i\}_{i=1}^n$ *where* $L_i \in \mathit{LhsPat}(\mathcal{G})$ *and* $R_i \in \mathit{RhsPat}(\mathcal{G}, L_i)$. $\square$

Example 9 ($\mathcal{G}_\lambda$ evaluator)
$\boxed{\lambda}$ is a simple call-by-value evaluator for $\mathcal{G}_\lambda$ graphs defined by the rules below.

$$
\begin{array}{rcll}
\{a \mapsto F\ X\}_{a,s} & \longrightarrow & \{a \mapsto F, b \mapsto X, t \mapsto b : s\}_{a,t} & (\mathit{App1}) \\
\{a \mapsto \lambda x.E, s \mapsto y : t\}_{a,s} & \longrightarrow & \{a \mapsto \lambda x.E, s \mapsto a; t\}_{y,s} & (\mathit{App2}) \\
\left\{ \begin{array}{c} a \mapsto \lambda x.E, \\ f \mapsto \lambda y.B, s \mapsto f; t \end{array} \right\}_{a,s} & \longrightarrow & \left\{ \begin{array}{c} a \mapsto \lambda x.E, \\ f \mapsto B[a/y], s \mapsto f; t \end{array} \right\}_{f,t} & (\mathit{App3}) \\
\{a \mapsto x, x \mapsto E\}_{a,s} & \longrightarrow & \{a \mapsto E, x \mapsto E\}_{a,s} & (\mathit{Var})
\end{array}
$$

(App1) begins evaluating an application by pointing the expression root to the function part F and saving the argument part X in a new node b pointed to by new stack node s. When F reaches a λ-value, the argument part is evaluated and the stack node changed to point at the function by *(App2)*.

When the argument also reaches a value, *(App3)* the rule from Example 7 specialises and then evaluates the function body. Whenever parameter y is needed it is *copied* by *(Var)*. We do not give a formal argument for correctness but this is a fairly standard small-step interpretation of the rules in Example 1. Unlike those natural semantics, our rules specify the order in which the function and argument are evaluated (an arbitrary choice); *(Var)* makes it clear that there is no sharing under a lambda; the stack is explicit and easily measured by counting S-nodes. □

An evaluator defines an evaluation relation. Each step updates a rooted sub-graph; it is followed by a garbage collection to ensure the graph remains a true reflection of space usage. Juxtaposition denotes disjoint union for appending graphs and $G \equiv H$ means G and H are identical.

Definition 11 (Evaluation, redex)
The evaluation relation $\longrightarrow_A$ *defined by* $\boxed{A}$ *is:*

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

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

We know each rule generates valid sub-graphs. It also never changes the category of a node, never introduces new free variables and never removes nodes so evaluation preserves well-formedness.

Proposition 3 (Preservation of well-formedness)
If G *is well-formed and* $G \longrightarrow H$ *then* H *is well-formed.* □

By Proposition 4 the work at each step (ignoring the garbage collection) is bounded. Unfortunately, updating holes makes the work depend on node size, which can grow. We return to this in Section 4.1.

Proposition 4 (Complexity of a Rewrite Step)
If $\boxed{A} = \{L_i \longrightarrow R_i\}_{i=1}^n$ *then* $G \longrightarrow_A H$ *is* $O(t \times a \times f \times \Sigma_{i=1}^n (size(L_i R_i)))$.
where t *is the maximum term size in* G*,* a *is the size of an address (which we assume is constant) and* f *is the size of a function symbol. Thus we cannot reduce the workload simply by increasing the size of the grammar.* □

We currently have no notion of initial or final states. A graph $G \notin dom(\longrightarrow_A)$ could be a *value* (a final state) or an error state. In either case we write $G \not\longrightarrow_A$. A *divergent* graph has no value. It could evaluate to an error state (a detectably divergent graph) or fail to terminate.

3.3 Determinism

We are primarily interested in modelling the space behaviour of *sequential* functional languages. Non-deterministic semantics are useful for some applications such as parallel evaluation, but we insist on determinism, requiring that
$G \longrightarrow_A G' \wedge G \longrightarrow_A G'' \Rightarrow G' \equiv G''$. This implies confluence and means that space and time usage do not depend on which rule is applied.

The question of how to decide which rule should be applied remains. We could use a prioritised model where the first rule to match a graph is applied. But then rule order becomes significant. The implementator's solution is to use the graph structure at the root to determine which rule to apply. In graph rewriting a similar idea, *orthogonality* [EEKR99], guarantees confluence. We already have left-linear rules so if an evaluator is *non-overlapping* only one rule can be applied. A relevant aside here is that our framework is less prone to the non-confluence problems of general graph rewriting because we have explicit addressing. The term-graph rewriting system $F(x) \longrightarrow G(x,x)$ and $F(A) \longrightarrow G(A,A)$ is not confluent because the first rule shares the argument x whereas the second duplicates the argument term [EEKR99]. In our system the equivalent rules would not overlap because x is an address and A is a term. If x is a hole then both rules cause duplication.

Definition 12 (Non-overlapping evaluator)
An evaluator is non-overlapping if no left pattern does overlaps with any other left pattern. L and M overlap if they can match the same graph, that is if $\exists G, G', \psi, \phi, \sigma, \zeta \cdot G(match\ \psi\ \phi\ L) \equiv G'(match\ \sigma\ \zeta\ M)$. □

3.4 Evaluator checking and graph evaluation

The kit accepts an operational semantics definition in its input and checks that each rule preserves well-formedness according to the rules in this section. The line below is the kit input defining (*App1*). After processing the rules are displayed according to the formatting strings given with the grammar, producing the definitions in Example 9 for $\boxed{\lambda}$. The checking can infer whether a symbol is a variable, hole or function so there is no need for any special notation. Both `f` and `x` below are identified as holes.

```
{a = APP f x}a,s -> {a = f, b = x, t = ARG b s}a,t "App1"
```

After checking, the specified graph is evaluated. For the graph from Example 5 the trace shown below is produced. We have only made a few cosmetic changes to the trace — removing brackets, changing node order a

little and adding λ to the arrows. As this is a non-terminating example the last 5 steps repeat indefinitely, only the names of the allocated nodes change: b and c after 2 steps are equivalent to d and e after 7 steps in the trace. The example will run in bounded space.

$$
\begin{array}{ll}
\{a \mapsto (\lambda x.(x\ x))\ (\lambda x.(x\ x))\}_{a,\epsilon} & \\
\longrightarrow_\lambda \{a \mapsto \lambda x.(x\ x), b \mapsto \lambda x.(x\ x), c \mapsto b : \epsilon\}_{a,c} & (App1) \\
\longrightarrow_\lambda \{a \mapsto \lambda x.(x\ x), b \mapsto \lambda x.(x\ x), c \mapsto a; \epsilon\}_{b,c} & (App2) \\
\longrightarrow_\lambda \{a \mapsto b\ b, b \mapsto \lambda x.(x\ x)\}_{a,\epsilon} & (App3) \\
\longrightarrow_\lambda \{a \mapsto b, b \mapsto \lambda x.(x\ x), d \mapsto b, e \mapsto d : \epsilon\}_{a,e} & (App1) \\
\longrightarrow_\lambda \{a \mapsto \lambda x.(x\ x), b \mapsto \lambda x.(x\ x), d \mapsto b, e \mapsto d : \epsilon\}_{a,e} & (Var) \\
\longrightarrow_\lambda \{a \mapsto \lambda x.(x\ x), b \mapsto \lambda x.(x\ x), d \mapsto b, e \mapsto a; \epsilon\}_{d,e} & (App2) \\
\longrightarrow_\lambda \{a \mapsto \lambda x.(x\ x), d \mapsto \lambda x.(x\ x), e \mapsto a; \epsilon\}_{d,e} & (Var)
\end{array}
$$

4 Space Usage

4.1 Space-Valid Evaluators

We want evaluators to provide a direct, simple model of space and time usage. We will use the following functions. The *space* needed to evaluate a graph is the number of nodes in the largest graph encountered during evaluation. The *time* of an evaluation is the number of steps.

Definition 13 (Evaluation *space* and *time*)
$space(G, \boxed{A}) = \max\{\#G' | G \longrightarrow_A^* G'\}$
$time(G, \boxed{A}) = t,\ where\ G \longrightarrow_A^t G' \wedge G' \not\longrightarrow_A$ □

For *space*, the problem is that an evaluator can cause nodes to grow. For *time*, copying and substituting into the terms that match holes is problematic since nodes can grow, so there is no bound on the work at each step. Both measures become valid if evaluation does not cause graph nodes to grow without limit. Such an evaluator is *space valid*.

Definition 14 (Space Valid Evaluator)
$\boxed{A}$ *is space valid if* $\forall G \exists k \cdot \{size(G') | G \longrightarrow_A^* G'\} \leq k \times space(G, \boxed{A})$ □

We use Proposition 5 to check the space validity of evaluators, it demands a kind of linearity in the holes in each right pattern node: they must all occur in the same left pattern node to ensure no node grows. For nodes without holes there must be some maximum size. This check is sufficient for space validity but not necessary. Evaluator $\boxed{\lambda}$ is space valid and satisfies the check, but had we stored the saved expressions at b and a inside stack nodes in

rules (*App1*) and (*App2*) then the modified $\boxed{\lambda}$ would still be space valid but (*App2*) fails the check.

Proposition 5 (Space Validity Check)
If for all rules $(L \longrightarrow R) = ((\{x_i \mapsto P_i\}_{i=1}^n)_V \longrightarrow (\{y_i \mapsto Q_i\}_{i=1}^m)_W) \in \boxed{A}$,
$1 \leq i \leq m \Rightarrow (\exists P_i \cdot Holes(Q_i) \subseteq Holes(P_j) \wedge size(Q_i) \leq size(P_j))$
or $Holes(Q_i) = \langle\rangle$ *then* $\boxed{A}$ *is space valid.*
Proof. $size(match\ \phi\ \psi\ P) = size(P) + \Sigma\langle size(\psi(h)) | h \in Holes(P)\rangle$ *by induction on* P. *If* $k = \max\{size(Q_i)\}_{i=1}^m$ *and* $G(match \phi\psi L) \longrightarrow_A G(match \phi\psi R)$ *then:* $\max\{size(match\ \phi\ \psi\ Q_i)\}_{i=1}^m \leq \max\{k, \max\{size(match\ \phi\ \psi\ P_i)\}_{i=1}^n\}$. *By induction on the length of the evaluation, no node bigger than* k *or the largest node in the initial graph is ever allocated.* □

4.2 Explicit Deallocation

In general, rules are not allowed to remove nodes from the graph as they might then become ill-formed. A simple analysis labels a rule *garbage generating* if there could be some garbage to collect after it is applied. Any nodes in $R \setminus reach(R, W \cup dom\ L)$ where $(L_V \longrightarrow R_W)$ are always garbage and the kit removes them from the rule. Any node in $gc(R_W)$ is never garbage after the rewrite, so we can label rules *garbage generating* by Definition 15.

Definition 15 (Garbage generating rule)
$(L_V \longrightarrow R_W)$ *is garbage generating if* $gc(R_W) \not\equiv R_W$. □

The *(App3)* and *(Var)* rules are the only garbage generating rules in $\boxed{\lambda}$. We could further reduce the garbage collection load: variables known to be used exactly once could be explicitly deallocated by a variant of *(Var)*; functions that never use their argument could be labelled, allowing *(App3)* to remove argument node a sometimes. We also know that s in *(App3)* should be garbage, so *(App3)* could be allowed to explicitly deallocate it, this is the idea we develop here.

Definition 16 (Stack category)
$S \in \mathcal{G}$ *is a stack category if only one root variable and* S*-nodes may point to* S*-nodes. No function symbol* $(F\ x_1 \cdots x_b.A_1 \cdots A_a) \in S$ *can refer to more than one* S*-node, so* $A_i = s \wedge j \neq i \Rightarrow A_j \neq S \wedge A_j \neq s$. □

If S is a stack category then (after garbage collection) S-nodes must form a chain. They could still be a cycle or become a cycle after an evaluation step, but if we are sure this never happens then it is safe to let a rule explicitly deallocate S-nodes when their reference count is decreased by Proposition 6.

Proposition 6 (Stack node deallocation)
If: S is a stack category; in initial graph G_V, $V \cup rng\ G$ is linear in S-variables; and in the evaluator $\{L_i \longrightarrow R_i\}$, each $gc(R_i)$ is linear in S-variables and S-holes. Then any S-node not in $gc(R_i)$ may be removed in the evaluation of G_V. □

4.3 Checking evaluator space properties

The kit can check for space validity as described above and calculates in advance the maximum node size k for the evaluation of a graph. We should make the distinction made in [BR00]: in the special case where the grammar is non-recursive we can find k without knowing the graph or evaluator, then we call $\boxed{\text{A}}$ *program-independent space valid*. If we have to apply Proposition 5 it is *program-dependant space valid*. For example, $\boxed{\lambda}$ is program-dependant space valid, the maximum node size when evaluating Example 5 is the initial node size, 15.

To help guarantee well-formedness, the kit extends rules according to the rule $extend(L_V \longrightarrow R_W) = (L_V \longrightarrow R(L \setminus R)_W)$. This permits a shorthand notation for semantics but it still leaves all the deallocation work to gc. We cannot force a deallocation. The stack node deallocation analysis does result in node s being removed from the right hand side of *(App3)* since S is a stack category.

Example 10 (Shorthand description of $\boxed{\lambda}$)
We can take advantage of *extend* to simplify *(App2)*,*(App3)* and *(Var)* in the definition of $\boxed{\lambda}$.

$$\begin{array}{rcll} \{a \mapsto \lambda x.E, s \mapsto y : t\}_{a,s} & \longrightarrow & \{s \mapsto a; t\}_{y,s} & (App2) \\ \{a \mapsto \lambda x.E, f \mapsto \lambda y.B, s \mapsto f; t\}_{a,s} & \longrightarrow & \{f \mapsto B[a/y]\}_{f,t} & (App3) \\ \{a \mapsto x, x \mapsto E\}_{a,s} & \longrightarrow & \{a \mapsto E\}_{a,s} & (Var) \end{array}$$

□

We have not covered the IO facility of the kit in this paper. Essentially, function symbols of *base categories* (whose functions all have no parameters) may be read from an input string and written to the output. Thus our models can include all the space under direct control of an implementation. The IO mechanism adds side effects to rules: sequences of *put* and *get* actions which are executed in order after the left pattern match succeeds. The *get* actions introduce new variables but they cannot overwrite existing ones. This IO system allows us to define asymptotic space usage as a function of the number of inputs read [BR00], since it does not really make sense to consider input as part of the initial graph.

5 Related Work

Another style of small-step semantics designed for reasoning about space usage has been researched for Core Scheme, a much richer language than our call-by-value system having data types, assignment, recursion and choice [MH97]. Clinger [Cli98] used that work for comparing the behaviour of different Scheme abstract machines. It is perhaps less suitable for reasoning at the program level, being a complex abstract machine with heap, stack and environment components.

A *profiling semantics* for NESL [BG96] is another implementation-oriented abstract machine. Minamide's proof that its continuation-passing style translation preserves space usage [Min99] uncovers a space leak in the original definition resulting from not counting the stack space (obscured by the use of a big-step semantics) properly.

We have highlighted some of the differences of our approach from traditional term-graph rewriting as described by [AA95, EEKR99]. Essentially these stem from our need for greater control over copying and sharing than in plain graph rewriting. Our framework is more similar to *equational term-graph rewriting* [AK96]. Term rewriting has also been investigated as a basis for operational semantics [Ros96]. Annotations to indicate sharing are needed, so it is not such an immediate model of space usage.

Other work aimed at controlling the space usage of a language includes region analysis [TT94], an implementation of ML that entirely removes garbage collection without damaging space usage. The *sized types* system [Par98] accepts a subset of Haskell which is guaranteed to run in bounded space.

6 Conclusions

We have defined a style of term-graph which describes the state of an evaluation system as a set of addressed terms. This formalises a graphical notation so it should be a helpful way to understand the evaluation of functional programs as well as being a precise notation for reasoning about space phenomena. We gave a framework for specifying operational semantics as graph evaluators, making sure they are deterministic and that they model space and time usage accurately. Though this is not yet a formal argument, we are trying to use a style of semantics that is as abstract as possible without losing information pertinent to space usage, to give a resource-usage meaning to source-level programs. We also began to explore some other ideas relating to controlling garbage collection, though we cannot yet specify different garbage collectors.

The kit is primarily a research tool, intended to support our work on investigating and comparing different operational semantics. Current work includes developing a space-semantics for Core Haskell. We are interested in developing proofs that show programs have similar space behaviour, or that different semantics have asymptotically different space behaviour for some programs [BR00]. The kit is a possible starting point for machine checking parts of such proofs. There is also scope for developing the kit in its own right as a graph rewriting system for prototyping systems where operational behaviour is an important consideration. It would benefit from a notation for handling families of functions (or variadic functions), the type system might be improved and a stronger notion of initial, final and error states could be developed. Support for a natural semantics style which does not lose the precise operational meaning of our small-step style would be a useful addition.

References

[AA95] Z M Ariola and Arvind. Properties of a first-order functional language with sharing. *Theoretical Computer Science*, 146:69–108, 1995.

[AK96] Z M Ariola and K W Klop. Equational term graph rewriting. *Fundamenta informaticae*, 26:207–240, 1996.

[BG96] G E Blelloch and J Greiner. A provably time and space efficient implementation of NESL. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, 1996.

[BR00] A Bakewell and C Runciman. A model for comparing the space usage of lazy evaluators. In *Proc. 2nd ACM Conference on Principles and Practice of Declarative Programming Langauges, Montreal*. ACM Press, September 2000.

[Cli98] W D Clinger. Proper tail recursion and space efficiency. In *Proc. 3rd ACM Internataional Conference on Functional Programming, Baltimore, Maryland*, pages 174–185. ACM Press, September 1998.

[EEKR99] 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.

[MH97] G Morrisett and R Harper. Semantics of memory management for polymorphic languages. In A D Gordon and A M Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 175–226. Cambridge University Press, 1997.

[Min99] Y Minamide. Space-profiling semantics of the call-by-value lambda calculus and the CPS transformation. In *The 3rd International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

[Par98] L Pareto. Sized types. Master's thesis, Dept. of Computing Science, Chalmers University of Technology, February 1998.

[Ros96] K H Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, University of Copenhagen, Febuary 1996.

[Ses97] P Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[TT94] M Tofte and J-P Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. *Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

Appendix: Kit Example

The listing below is the kit input defining a simple functional programming language which includes the constructs of the λ-calculus with function arguments restricted to variables and extended with boolean constructors and If expressions. Category S terms are used to hold the context while the value of an applied expression, the selector of an If or a variable is being found. The evaluator specifies call-by-need evaluation for this language: arguments are not evaluated before the Reduce rule enters a function body, instead they are evaluated when they are first needed by Lookup.

```
-- Lambda calculus plus Boolean data type and If
ROOT ::= x s

X ::= LAM x . X   "\lambda #1.#2"        "\#1.#2"
    | APP X x     "#1 \: #2"             "#1 #2"
    | VAR x       "#1"                   "#1"
    | BOT         "\bot"                 "_|_"
    | CTR BOOL    "#1"                   "#1"
    | IF X X X    "{\it if}\:#1\:#2\:#3" "#1 ? #2 : #3"
BOOL ::= TRUE     "{\it True}"           "True"
       | FALSE    "{\it False}"          "False"

S ::= PSH x s     "#1 ; #2"              "#1:#2"
    | UDM x s     "\# #1 \: #2"          "# #1 #2"
    | PSHIF X X s "\{#1:#2\} ; #3"       "{#1;#2}:#3"

-- Call-by-need operational semantics
{a = APP F X}a,s -> {a = F, t = PSH X s}a,t       "Push"
{a = LAM x.E, s = PSH y t}a,s -> {a = E[y/x]}a,t"Reduce"
{a = VAR x}a,s -> {a = BOT, t = UDM a s}x,t     "Lookup"
{a = LAM x.E, y = BOT, s = UDM y t}a,s
 -> {y = LAM x.E}y,t                            "Update"
{a = IF e x y}a,s -> {a = e, t = PSHIF x y s}a,t"PushIf"
{a = CTR TRUE, s = PSHIF x y t}a,s -> {a = x}a,t   "IfT"
{a = CTR FALSE, s = PSHIF x y t}a,s -> {a = y}a,t  "IfF"
{a = CTR c, y = BOT, s = UDM y t}a,s
 -> {y = CTR c}y,t                           "UpdateCtr"

-- Graph to calculate TRUE && TRUE
and = LAM x . (LAM y . (IF (VAR x) (VAR y) (CTR FALSE)))
true = CTR TRUE
main = APP (APP (VAR and) true) true

root = main,null
```

The kit output, formatting for Latex display, is shown below. The grammar is accepted, S is recognised as a stack category so node s is not included in the extended right-hand sides of many rules. The graph is well-formed and it is evaluated resulting in the final graph which correctly maps address *main* to *True*.

$$
\begin{array}{rcll}
ROOT & ::= & \langle x, s \rangle & \\
 & & & \\
X & ::= & LAM\ x.X & \lambda x.X \\
 & | & APP\ X\ x & X\ x \\
 & | & VAR\ x & x \\
 & | & BOT & - \\
 & | & CTR\ BOOL & BOOL \\
 & | & IF\ X\ X'\ X'' & if\ X\ X'\ X''
\end{array}
\qquad
\begin{array}{rcll}
BOOL & ::= & TRUE & True \\
 & | & FALSE & False \\
 & & & \\
S & ::= & PSH\ x\ s & x; s \\
 & | & UDM\ x\ s & \#x\ s \\
 & | & PSHIF\ X\ X'\ s & \{X : X'\}; s
\end{array}
$$

$$
\begin{array}{rcll}
\{a \mapsto F\ X\}_{a,s} & \longrightarrow & \{a \mapsto F, t \mapsto X; s\}_{a,t} & (Push) \\
\{a \mapsto \lambda x.E, s \mapsto y; t\}_{a,s} & \longrightarrow & \{a \mapsto E[y/x]\}_{a,t} & (Reduce) \\
\{a \mapsto x\}_{a,s} & \longrightarrow & \{a \mapsto -, t \mapsto \#a\ s\}_{x,t} & (Lookup) \\
\{a \mapsto \lambda x.E, y \mapsto -, s \mapsto \#y\ t\}_{a,s} & \longrightarrow & \{y \mapsto \lambda x.E, a \mapsto \lambda x.E\}_{y,t} & (Update) \\
\{a \mapsto if\ e\ x\ y\}_{a,s} & \longrightarrow & \{a \mapsto e, t \mapsto \{x : y\}; s\}_{a,t} & (PushIf) \\
\{a \mapsto True, s \mapsto \{x : y\}; t\}_{a,s} & \longrightarrow & \{a \mapsto x\}_{a,t} & (IfT) \\
\{a \mapsto False, s \mapsto \{x : y\}; t\}_{a,s} & \longrightarrow & \{a \mapsto y\}_{a,t} & (IfF) \\
\{a \mapsto c, y \mapsto -, s \mapsto \#y\ t\}_{a,s} & \longrightarrow & \{y \mapsto c, a \mapsto c\}_{y,t} & (UpdateCtr)
\end{array}
$$

$$
\begin{array}{ll}
\{and \mapsto \lambda x.\lambda y.if\ x\ y\ False, true \mapsto True, main \mapsto and\ true\ true\}_{main,\epsilon} & \\
\longrightarrow \quad \{main \mapsto and\ true, a \mapsto true; \epsilon, & \\
\qquad and \mapsto \lambda x.\lambda y.if\ x\ y\ False, true \mapsto True\}_{main,a} & (Push) \\
\longrightarrow \quad \{main \mapsto and, b \mapsto true; a, a \mapsto true; \epsilon, & \\
\qquad and \mapsto \lambda x.\lambda y.if\ x\ y\ False, true \mapsto True\}_{main,b} & (Push) \\
\longrightarrow \quad \{main \mapsto -, c \mapsto \#main\ b, b \mapsto true; a, a \mapsto true; \epsilon, & \\
\qquad and \mapsto \lambda x.\lambda y.if\ x\ y\ False, true \mapsto True\}_{and,c} & (Lookup) \\
\longrightarrow \quad \{main \mapsto \lambda x.\lambda y.if\ x\ y\ False, b \mapsto true; a, & \\
\qquad a \mapsto true; \epsilon, true \mapsto True\}_{main,b} & (Update) \\
\longrightarrow \quad \{main \mapsto \lambda y.if\ true\ y\ False, a \mapsto true; \epsilon, true \mapsto True\}_{main,a} & (Reduce) \\
\longrightarrow \quad \{main \mapsto if\ true\ true\ False, true \mapsto True\}_{main,\epsilon} & (Reduce) \\
\longrightarrow \quad \{main \mapsto true, d \mapsto \{true : False\}; \epsilon, true \mapsto True\}_{main,d} & (PushIf) \\
\longrightarrow \quad \{main \mapsto -, e \mapsto \#main\ d, & \\
\qquad d \mapsto \{true : False\}; \epsilon, true \mapsto True\}_{true,e} & (Lookup) \\
\longrightarrow \quad \{main \mapsto True, true \mapsto True, d \mapsto \{true : False\}; \epsilon\}_{main,d} & (UpdateCtr) \\
\longrightarrow \quad \{main \mapsto true, true \mapsto True\}_{main,\epsilon} & (IfTrue) \\
\longrightarrow \quad \{main \mapsto -, f \mapsto \#main\ \epsilon, true \mapsto True\}_{true,f} & (Lookup) \\
\longrightarrow \quad \{main \mapsto True\}_{main,\epsilon} & (UpdateCtr) \\
space = 6, time = 12 &
\end{array}
$$