A Space Semantics for Core Haskell

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

August 11, 2000

Abstract

Haskell currently lacks a standard operational semantics. We argue that such a semantics should be provided to enable reasoning about operational properties of programs, to ensure that implementations guarantee certain space and time behaviour and to help determine the source of space faults. We present a small-step deterministic semantics for the sequential evaluation of Core Haskell programs and show that it is an accurate model of asymptotic space and time usage. The semantics is a formalisation of a graphical notation so it provides a useful mental model as well as a precise mathematical notation. We discuss its implications for education, programming and implementation. The basic semantics is extended with a monadic IO mechanism so that all the space under the control of an implementation is included.

1 Introduction

Space usage is an area of uncertainty for Haskell programmers. Recent comments on the Haskell mailing list include "It is difficult to understand how my program relates to memory management" and "The space behaviour of Haskell programs is often very subtle". So the average user is almost totally in the dark and even the more experienced can be mystified by space usage. Over 10 years ago, Wadler remarked "A theory that allows one to reason about the space and time efficiency of functional programs is long overdue" [Wad87]. Some work has been done in this area [Lau93, MS98] but the Haskell standard has done nothing to address the problem.

To a certain extent this difficulty is an inevitable consequence of the functional style, which eliminates memory management from programs. However, eager languages like Scheme and SML have operational standards that include a worst-case space behaviour [ICS91, App92]. Such a standard should exist for Haskell, so that we can reason about the space and time properties of programs and implementations, determine which program transformations might worsen operational behaviour and so on. Of course a standard semantics does not immediately solve all the problems of space prediction and control, but at least programmers can begin to investigate without dismantling a compiler.

Even those who understand an implementation in detail do not have portable knowledge. Different implementations have different space behaviour. A conversation (based on remarks on the Haskell list) might proceed: "This program has a space leak on Implementation X"; "Oh, there's no problem on Implementation Y"; "Implementation Z also leaks, but in a different way to Implementation X"; "Is the implementation leaking? It could be the program". A standard semantics removes uncertainty in this situation by determining whether the implementation or the program should be improved.

We present a small-step, structural operational semantics for the sequential, deterministic evaluation of Core Haskell programs. The semantics are written in a term-graph formalism that pays particular attention to accurately describing space usage, making sharing and addressing explicit. This provides an accurate asymptotic model, by representing a state as a set of bounded-sized nodes. The presentation is at a higher level than most abstract machine designs so it represents a variety of implementation styles: analysing the space usage of a program, or analysing the effect of changing the evaluation strategy on space usage, is not tied to any particular implementation.

Graphical representation The term-graph semantics formalise of a graphical notation. This in itself is a useful aid; the rules are easily visualised and they offer a precise method for drawing out a Haskell evaluation. This way we can be sure that our graph sketches are accurate in the degree of evaluation, the degree of sharing and the effect of the context on an evaluation. The only gap a user has to overcome is the translation from Haskell to the Core Haskell language used by the semantics. Fortunately, Core has few non-program constructs so programmers do not need to understand abstract machine design to use the semantics in this way.

The translation gap The semantics of the full language is specified by the semantics of the Core language and the translation of other constructs into Core. If the translation assumed by a user is not the translation of the compiler, that misunderstanding could lead to unexpected operational behaviour.

Implementors also have to overcome the gap between the graph semantics and their abstract machine if they wish to comply with the standard. The term-graph formalism is quite flexible, the standard semantics should easily adapt to model different kinds of abstract machine.

The term-graph framework Every evaluation step represents a unit of computation, we cannot use side-conditions or mathematical tricks to hide complex steps. A Core Haskell graph and the semantics are a closed system, they include all the space needed for reduction. The interpretation of space as graph size is straightforward.

This model does not include the space for code and libraries, but that is usually regarded as a large constant for space analysis. The relation of type classes to the semantics may be an issue because sometimes using a class instance adds a dictionary argument to a function. For now we are assuming without evidence that dictionaries also present a constant space overhead. This may break down in the presence of polymorphic recursion if a dictionary parameter must be added to a function.

Reasoning about garbage Our model comes with a simple reachability-based collector. The combination of graph evaluator and garbage collector comprise a minimum standard of space efficiency.

Contents Section 2 reviews some related work. Section 3 introduces the term-graph grammar for Core Haskell programs, a measure of state size and a garbage collector. Section 4 defines the semantics. Section 5 briefly discusses its applications as a graphical notation for education, a standard for programmers and implementors and as a basis for analyses. Section 6 explains the properties that guarantee accurate space and time modelling and correctness. Section 7 is a monadic IO extension. Section 8 concludes and the Appendix gives the correctness proof.

2 Related Work

Launchbury [Lau93] gave a natural semantics for lazy evaluation which models the STG-machine [Pey92]. Sestoft [Ses97] developed various small-step semantics from the natural semantics.

Sestoft's semantics can be used as the basis for a space analysis (e.g. [GS99]) but the interpretation is awkward because not all of the space needed for evaluation is directly present in the model: to count stack space and heap space separately it is necessary to count stack and heap space for *update markers* on the stack. The natural semantics could be used for space analysis too, but it seems to be at too high a level of abstraction: it hides the stack component needed by a real implementation and it needs annotating with information about live variables. The STG-machine [Pey92], on the other hand, is clearly too low-level.

Another style of semantics that has been used to reason about space usage is the Scheme abstract machine of Morrisett and Harper [MH97], again that has rather more detailed than we require.

Rose [Ros96] developed a term rewriting framework for operational semantics which is extended to allow sharing and cycles to be represented. This results in an alternative kind of graph reduction that can model time and space usage accurately.

Sestoft's semantics is at the right level of abstraction for our purposes, and our semantics is based closely on it, but we are concentrating on giving a clear description of space usage by using a general term-graph framework.

3 Core Haskell Graphs

This section discusses the Core Haskell language. Section 3.1 introduces the basic language. Section 3.2 explains the term-graph formulation of the language we use for describing space properties. Sections 3.3 and 3.4 discuss measuring size and garbage collection in this framework.

3.1 Core Haskell

Core Haskell expressions are terms built from the grammar (1) where x stands for a variable and X stands for an arbitrary expression.

$$X ::= \lambda x.X \mid X\ x \mid x \mid \bot \\ \mid\ let\ \{x_i = X_i\}_{i=1}^{n}\ in\ X \mid c^{a,r,n}\ x_1 \cdots x_a \\ \mid\ case\ X\ of\ \{c^{a_r,r,n}\ x_1 \cdots x_{a_r} \to X_r\}_{r=1}^{n} \qquad (1)$$

There are many issues in the choice of this language. We are trying to be consistent with the Haskell Report [Hask99] and the core languages of implementations such as the 2nd-order λ-calculus [Pey92]. We also want a language that enables a clear formulation of the semantics.

Core is a λ-calculus extended with algebraic data types and *let* for sharing and recursion. The argument of an application must be a variable. A data constructor $c^{a,r,n}$ has arity a and rank r of n, it must always have a arguments. This saturation ensures that we never need to apply constructors in the semantics. The *trans* rules (2) enforce these language restrictions before evaluation. Essentially *trans* is Launchbury's normalisation step [Lau93].

$$\begin{array}{l} trans\ (X\ x) = X\ x \\ trans\ (X\ X') = let\ \{y = X'\}\ in\ X\ y \\ \quad \text{where } X' \text{ is not a variable, } y \text{ is fresh} \\ trans\ (c^{a,r,n}\ X_1 \cdots X_m) = \\ \quad let\ \{x_i = X_i \mid 1 \le i \le m, X_i \text{ not a variable}\} \\ \quad in\ \lambda x_{m+1}.\cdots.\lambda x_a.c^{a,r,n}\ x_1 \cdots x_a \\ \quad \text{where } x_i = X_i \text{ if } X_i \text{ is a variable} \end{array} \qquad (2)$$

The *trans* step makes Core Haskell closer to implementations [Pey92]. Restricting arguments to be variables is said to ensures sharing [Ses97]. We could easily ensure sharing in the semantics, but the restriction is easy to apply, it makes sharing more obvious to the reader and it makes the semantics simpler because *let* is the only construct whose evaluation allocates expressions. An implementation might also restrict applied expressions or case subject-expressions to variables, but those changes would complicate our presentation.

Variables may be bound by λ, *let* or *case* alternatives. They may also be free. We do not need to assume they are uniquely named within an expression. The undefined term $\bot$ is not really a standard part of Core; it is included primarily for modelling *black holing* (see Section 4).

A *let*-expression introduces n distinctly named, mutually recursive expressions for use by X. It is permissible for n to be zero or for some of the bindings to be semantically garbage.

An algebraic data type has n constructors. Each constructor has a rank r between 1 and n to identify it (in examples we can name constructors so there will be no need for this notation). *case* stands for a family of expressions: each data type has a *case* with a subject expression and an alternative for each constructor.

We have not included numbers or primitives because the choice of *Int* or *Integer* type affects space and time complexity: *Int* is like a very large data type comprising a bounded number of constants whereas *Integer* numbers are constructed in a list-like manner so bigger *Integer*s take more space. This omission means that our semantics does not specify the order in which primitive binary operations should evaluate their arguments, which is often an important consideration for programming.

Initially, we assume that a Haskell program is translated into a single Core expression using the standard techniques [Pey87]. This introduces a space issue. Translation replaces programs with their simplified equivalents, but there is no guarantee that the translation will have the operational behaviour expected of the original. A standard semantics helps here. Given a program and its translation we can determine its operational behaviour. Thus, to give a semantics to the full language we only need the Core semantics and a standard translation scheme.

Example 1 (Core translation)

```
main = null fs
where null [] = True
      null _ = False
      fs = False : fs
```
(3)

The program (3) translates to the Core expression (4). The function `null` becomes a λ-expression with the pattern matching done by a case expression. Constructor arguments must be variables, so the *False* in *fs* is defined separately as f.

$$\begin{array}{ll} let\ \{ & null = \lambda l.case\ l\ of \\ & \qquad \{[\,] \to True, (h:t) \to False\} \\ & f = False \\ & fs = f : fs\ \}\ in\ null\ fs \end{array} \qquad (4)$$

□

3.2 Term-Graph Core Language

The semantics are expressed in a term-graph rewriting formalism (described in [BR00b]). We model a state as a *term graph*, which is a mapping from variables to terms — a set of addressed terms. Free variables in a term are the addresses of other terms in the graph — the graph arcs. A variable may also be the empty address, ϵ, if there is no arc.

Category	Term definition	Usual notation	
X ::=	$LAM\ x.X$	$\lambda x.X$	λ-abstraction
|	$APP\ X\ x$	$X\ x$	expression application
|	$VAR\ x$	x	variable (bound or a node address)
|	BOT	$\perp$	undefined term symbol
|	$LETn\ x_1 \cdots x_n.X_1 \cdots X_n X$	$let\ \{x_i = X_i\}_{i=1}^{n}\ in\ X$	let expression for n bindings
|	$CTR\tau_r\ x_1 \cdots x_a$	$c^{a,r,n}\ x_1 \cdots x_a$	rth constructor of type τ, saturated
|	$CAS\tau\ X\ A\tau_1 \cdots A\tau_n$	$case\ X\ of\ \{A\tau_r\}_{r=1}^{n}$	case expression for type τ
$A\tau_r$ ::=	$ALT\tau_r\ x_1 \cdots x_a.X$	$c^{a,r,n}\ x_1 \cdots x_a \to X$	ith alternative of case for type τ
S ::=	$PSH\ x\ s$	$x : s$	pushed argument x
|	$UDM\ x\ s$	$\#x\ s$	update marker for x
|	$PSH\tau\ A\tau_1 \cdots A\tau_n\ s$	$\{A\tau_r\}_{r=1}^{n} : s$	pushed case alternatives

Figure 1: Core Haskell Term Grammar. X defines expression terms as described in Section 3.1. *let* and *case* stand for families of function symbols, the initial program dictates which are needed. Each algebraic data type definition $data\ \tau = \{c_r\ T_{r,1} \cdots T_{r,a_r}\}_{r=1}^{n}$ introduces a set of constructors and a *case*. A *case* has a subject expression then an alternative for each constructor. An alternative binds the appropriate number of variables for its constructor. S defines stack terms which are generated during evaluation, they form a chain in the graph. The variable s in each stack term addresses the next S-node in the chain.

In the term-graph framework, the expression grammar (1) is a syntactic *category* X (a higher-order *sort*) defining the *function symbols* that build expressions. A function symbol like λ or *let* binds some variables and takes some arguments, which may be variables or terms of a specified category. $T \in Term(c)$ means T is a term of category c. Similarly, $x \in Var(c)$ means x stands for a term of category c, either it is bound or it is an arc pointing to some other node in the graph. For example, $f \in Var(X)$ in (4). Figure 1 summarises the term formulation of the grammar. Now we define Core Graphs, the distributed-term state model.

Definition 1 (Core Graph)
$^x_sG = {}^x_s\{v_i \mapsto T_i\}_{i=1}^{n}$ is a Core Graph if:
$\forall T \in rng\, G, (T \in Term(X) \cup Term(S)) \wedge$
$(\exists c \cdot v_i \in (FV(T) \cap Var(c)) \Rightarrow T_i \in Term(c))$.
The graph has two root variables written immediately before the set of node definitions: x addresses the current evaluation position (the control), s addresses the current stack node (the context for the current evaluation). We must have $(x \mapsto T_x) \in G$ with $T_x \in Term(X)$ and either $(s \mapsto T_s) \in G$ with $T_s \in Term(S)$ or s is empty. □

Core Graphs must be *well-formed* — closed and self-consistent in that variables which should address expression nodes according to the grammar do so, similarly for stack nodes. The semantic rules locate nodes by following arcs from the roots. An open graph might fail to evaluate or might not capture the space requirements of a program properly. The simplistic category type-system eliminates some potential errors in graph construction and semantic rules (e.g. applying an S-node).

Example 2 (Core Graph)
To evaluate (4) we build the graph (5) which has one node containing the expression at address a. The control root is a and the stack root is empty. For convenience some sub-terms are abbreviated.

$$\begin{aligned} {}^a_\epsilon\{a \mapsto let\ \{\ & null = \mathbf{Null}, \\ & f = False, \\ & fs = f : fs\}\ in\ null\ fs\} \\ \text{where}\ \ \mathbf{Null} &= \lambda l.case\ l\ of\ \mathbf{Alts} \\ \mathbf{Alts} &= \{[\,] \to True, (h : t) \to False\} \end{aligned} \tag{5}$$

This definition is just a formalisation of a graphical notation, we can visualise the graph as (6). The graph is a box containing node boxes which house terms. Free variables in the terms are arcs to other nodes. The roots are drawn as points at the left of the graph with the control above the stack. An empty arc is drawn as an unconnected dot.

$$\boxed{\begin{array}{ll} \bullet\!\longrightarrow & \boxed{\begin{aligned} let\ \ \{&null = \mathbf{Null}, \\ &f = False, \\ &fs = f : fs\}\ in\ null\ fs \end{aligned}} \\ \bullet & \end{array}} \tag{6}$$

□

3.3 Graph Size

A Core Graph includes all the information needed during reduction by the semantics so we can use its *size* to model space usage. Terms have a natural notion of size and the *size* of a term graph is the sum of the sizes of its nodes. The graph domain takes no space; it is just labels for the terms in the graph range.

Definition 2 (Term and graph size)
$size\ x = 1$
$size\ (F\ x_1 \cdots x_b.T_1 \cdots T_n) = 1 + b + \Sigma_{i=1}^{n} size\ T_i$
$size\ ({}^{x}_{s}\{x_i \mapsto T_i\}_{i=1}^{n}) = \Sigma_{i=1}^{n} size\ T_i$ □

For example, according to the term encoding of the grammar in Figure 1 the size of (5) is 23. There are some hidden assumptions here: in practice the size of a variable depends on the space available for a graph (a name size is logarithmic in the size of the name space) but we abstract away from this by assuming an infinite set of variables. Similarly, the size of a term depends on the grammar size (in a bigger grammar a symbol will take more space) but we also abstract away from this. So *size* is reasonably accurate but we should not read too much into it. As we are presenting an asymptotic space model, rather than considering detailed abstract machine design, an even more abstract measure may be preferable — the graph cardinality, $\#G$. The validity of this measure is discussed in Section 6.

3.4 Garbage Collection

The size of a graph containing nodes that a garbage collector could remove would not be a very useful measure of space usage. The term-graph system includes a simple reachability-based collector *gc* which finds the smallest sub-graph of a Core Graph that is also a Core Graph. It uses *reach* to find the set of addresses reachable from the root addresses then restricts the graph domain to those addresses.

Definition 3 (Garbage Collector *gc*)
$gc({}^{x}_{s}G) = {}^{x}_{s}(G|_{reach(G,\{x,s\})})$
where $reach(G, X) = fix\ (\lambda V.X \cup V \cup FV(G|_V))$ □

For space analysis we assume that *gc* is after any rule that could generate garbage and before we count the graph size. An important question for Haskell is what level of garbage collection should we expect? Garbage collectors can often remove a lot more than *gc* does. For example, they can short-out projections of data structures to prune parts that will never be needed again [Wad87, MFH95]. There is no obvious basis for deciding what kinds of semantically dead nodes should be collected, we certainly cannot remove them all, but we definitely cannot justify leaving unconnected nodes in a graph. Some implementations move some of the work we treat as evaluation into the collector (by using indirections instead of copying, for example). In such cases we expect the overall asymptotic time and space behaviour to be the same as our model.

4 Operational Semantics

The operational semantics of Core in Figure 2 are a set of graph rewrite rules. They are based on Sestoft's mk.1 machine for lazy evaluation [Ses97] (repeated in the Appendix). Each rule ($L \longrightarrow R$) replaces a rooted sub-graph matching L with a rooted sub-graph matching R. The terms in L are patterns: they are built from the same grammar as Core terms except they may include *holes* — variables written in upper-case which match arbitrary expressions. Normal variables in a pattern only match variables in a graph. The terms in R are also patterns; their holes can have substitutions attached to replace any variables freed during the rewrite step with another variable which is still in scope.

Where a node occurs in R but not L it is *allocated* by the rule. Some rules do *explicit deallocation*: *(Reduce)*,*(Update)*,*(UpdateCtr)* and *(ReduceCase)* all remove a node s from the graph. We could have left them for the garbage collector of course, but the removal is safe because S-nodes form an acyclic chain: only S-nodes and the second graph root contain arcs to S-nodes, they can only have one such arc; none of the semantic rules duplicates a reference to an S-node or creates an S-node with more than one reference; initially there are no S-nodes.

Example 3 (Evaluation trace)
The evaluation trace of (5) using the semantics is shown in Figure 3. □

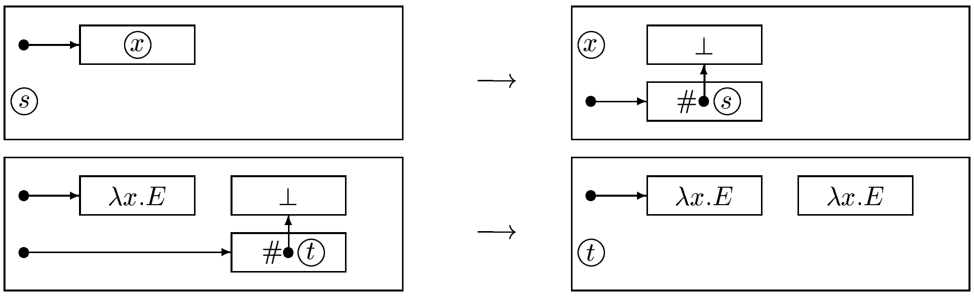
Now we can define the *space* and *time* required to evaluate a Core Graph as the maximum number of nodes needed and the number of steps needed respectively. In Example 3, *space* is 5 and *time* is 9. These simple measures are intended for asymptotic analysis, their validity is discussed in Section 6. Essentially, knowing the initial program sets a maximum node size for its evaluation. This means we can reduce *time* by using larger data types or let-blocks. Studying *size* usage instead, though superficially more accurate, would still only be an asymptotic model of space usage.

$$
\begin{array}{rcll}
{}^{a}_{s}\{a \mapsto E\,x\} & \longrightarrow & {}^{a}_{t}\{a \mapsto E, t \mapsto x : s\} & (\mathit{Push}) \\
{}^{a}_{s}\{a \mapsto \lambda y.E, s \mapsto x : t\} & \longrightarrow & {}^{a}_{t}\{a \mapsto E[x/y]\} & (\mathit{Reduce}) \\
{}^{a}_{s}\{a \mapsto x\} & \longrightarrow & {}^{x}_{t}\{a \mapsto \bot, t \mapsto \#a\,s\} & (\mathit{Lookup}) \\
{}^{a}_{s}\{a \mapsto \lambda x.E, y \mapsto \bot, s \mapsto \#y\,t\} & \longrightarrow & {}^{y}_{t}\{a \mapsto \lambda x.E, y \mapsto \lambda x.E\} & (\mathit{Update}) \\
{}^{a}_{s}\{a \mapsto c\,x_1 \cdots x_a, y \mapsto \bot, s \mapsto \#y\,t\} & \longrightarrow & {}^{y}_{t}\{a \mapsto c\,x_1 \cdots x_a, y \mapsto c\,x_1 \cdots x_a\} & (\mathit{UpdateCtr}) \\
{}^{a}_{s}\{a \mapsto \mathit{let}\ \{x_i = E_i\}_{i=1}^{n}\ \mathit{in}\ E\} & \longrightarrow & {}^{a}_{s}\{y_i \mapsto E_i[y_j/x_j]_{j=1}^{n}\}_{i=1}^{n} \cup \{a \mapsto E[y_i/x_i]_{j=1}^{n}\} & (\mathit{Let}) \\
{}^{a}_{s}\{a \mapsto \mathit{case}\ E\ \mathit{of}\ AS\} & \longrightarrow & {}^{a}_{t}\{a \mapsto E, t \mapsto AS : s\} & (\mathit{PushCase}) \\
{}^{a}_{s}\{a \mapsto c\,x_1 \cdots x_a, s \mapsto \{c\,y_1 \cdots y_a \to E\} \cup AS : t\} & \longrightarrow & {}^{a}_{t}\{a \mapsto E[x_i/y_i]_{i=1}^{a}\} & (\mathit{ReduceCase})
\end{array}
$$

(Push) With an application $E\,x$ at the control address we first evaluate E; the argument x is saved in a new stack node t which points back to the old stack top at s. Eventually, if E becomes a λ-value, the applied expression we started evaluating in *(Push)* has become a function. *(Reduce)* will apply: x replaces y in the function body which is then evaluated with t as the new stack top. This reduction is an in-place update, destroying the λ-value at a.
(Lookup) To evaluate a variable x we move the control root to find the value of the expression at node x. If found, this value will be copied into a, so we allocate stack node t which contains an update marker for a. Node a is overwritten with $\bot$; this is how we model *black holing*: if there is a direct self-dependency (e.g. in the graph ${}^{a}_{\epsilon}\{a \mapsto a\}$) it is detected because no rule evaluates $\bot$.
(Update),(UpdateCtr) Reaching a λ-value or a constructor value, with an update marker for y at s, we need to *copy* the value into y and continue from there. Any further uses of a will just *(Update)* immediately after *(Lookup)*. Free subexpressions within λ-bodies are not shared.
(Let) Each binding becomes a new graph node, then E is evaluated.
(PushCase) Analogous to *(Push)*, to evaluate a case expression we first evaluate its subject expression, saving the alternatives in a new stack node. *(ReduceCase)* When there is a constructor value at a, with a set of case alternatives at s, we reduce the case expression to the right-hand side of the alternative corresponding to that constructor, substituting any arguments for the pattern-bound variables. Like *(Reduce)*, this is an in-place update. The stack top moves to t and evaluation continues.
The garbage generating rules are (Reduce),(Update),$(\mathit{UpdateCtr})$,(Let) and $(\mathit{ReduceCase})$.

An *initial state* is a graph ${}^{a}_{\epsilon}\{a \mapsto T\}$ where $T \in \mathit{Term}(X)$. A *final state* is a graph ${}^{a}_{\epsilon}G$ to which no rule applies and which encodes a value as a distributed term, so $G(a)$ is a λ-value or a constructor value.

We can draw the rules as shown below. In the graphical notation, roots appear at the left of the graph-pattern boxes, with the control root above the stack root. Arcs internal to a pattern are drawn, variables that match arcs are circled, holes are upper-case variables and bound variables are lower-case. Position indicates address, so *(Lookup)* allocates a new node and updates the existing node to $\bot$, *(Update)* deallocates the # node.



(Lookup)

(Update)

Figure 2: Operational Semantics of Core.

$$
\begin{array}{lll}
 & {}^{a}_{\epsilon}\{a \mapsto \mathit{let}\ \{\mathit{null} = \mathbf{Null}, f = \mathit{False}, \mathit{fs} = f : \mathit{fs}\}\ \mathit{in}\ \mathit{null}\ \mathit{fs}\} & \\
\longrightarrow & {}^{a}_{\epsilon}\{a \mapsto b\, d, b \mapsto \mathbf{Null}, c \mapsto \mathit{False}, d \mapsto c : d\} & (\mathit{Let}) \\
\longrightarrow & {}^{a}_{s}\{a \mapsto b, b \mapsto \mathbf{Null}, c \mapsto \mathit{False}, d \mapsto c : d, s \mapsto d : \epsilon\} & (\mathit{Push}) \\
\longrightarrow & {}^{b}_{t}\{a \mapsto \bot, b \mapsto \mathbf{Null}, c \mapsto \mathit{False}, d \mapsto c : d, t \mapsto \#a\ s, s \mapsto d : \epsilon\} & (\mathit{Lookup}) \\
\longrightarrow & {}^{a}_{s}\{a \mapsto \mathbf{Null}, c \mapsto \mathit{False}, d \mapsto c : d, s \mapsto d : \epsilon\} & (\mathit{Update}) \\
\longrightarrow & {}^{a}_{\epsilon}\{a \mapsto \mathit{case}\ d\ \mathit{of}\ \mathbf{Alts}, c \mapsto \mathit{False}, d \mapsto c : d\} & (\mathit{Reduce}) \\
\longrightarrow & {}^{a}_{u}\{a \mapsto d, c \mapsto \mathit{False}, d \mapsto c : d, u \mapsto AS : \epsilon\} & (\mathit{PushBool}) \\
\longrightarrow & {}^{d}_{v}\{a \mapsto \bot, c \mapsto \mathit{False}, d \mapsto c : d, v \mapsto \#a\ u, u \mapsto AS : \epsilon\} & (\mathit{Lookup}) \\
\longrightarrow & {}^{a}_{u}\{a \mapsto c : d, c \mapsto \mathit{False}, d \mapsto c : d, u \mapsto AS : \epsilon\} & (\mathit{UpdateCtr}) \\
\longrightarrow & {}^{a}_{\epsilon}\{a \mapsto \mathit{False}\} & (\mathit{ReduceBool})
\end{array}
$$

Figure 3: Evaluation trace of (5), demonstrating the action of each rule in the semantics. The abbreviations introduced in (5) are used. Node b which contains the definition of *null* is removed by *gc* at the *(Update)* step; c and d which encode the infinite list *fs* are not collected until after the last step.

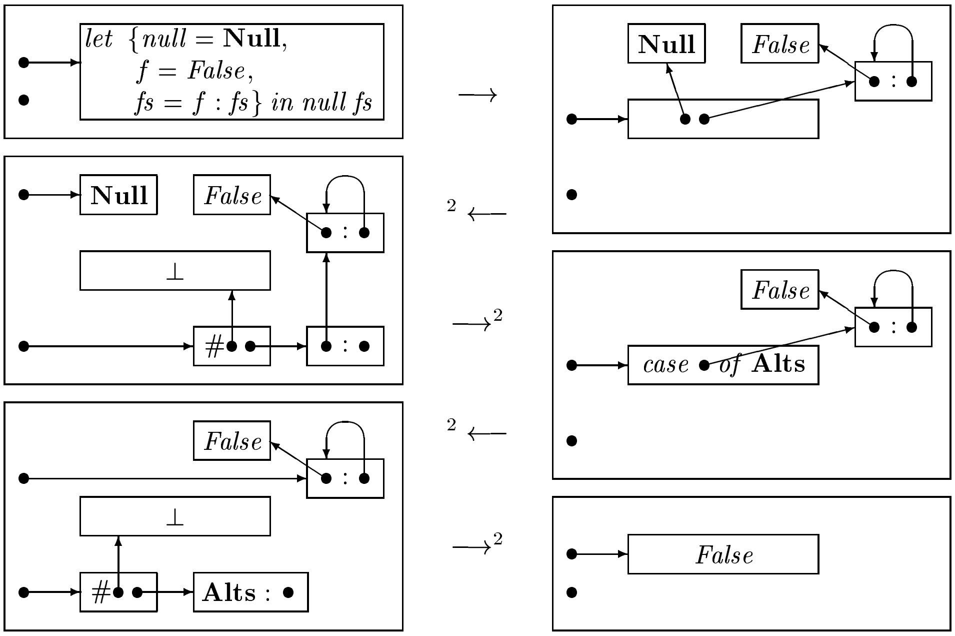


Figure 4: Pictorial representation of the evaluation trace in Figure 3, showing (top to bottom) the initial graph and the graph after 1,3,5,7 and 9 evaluation steps. In this representation it is much easier to see how the graph is changed during evaluation. Using node position to indicate address works well for identifying which nodes are allocated, updated or deallocated — though note that the S-nodes after the 7th step may or may not have the same location as the S-nodes after the 3rd step.

Definition 4 (Space and time usage)
$space(G) = \max\{\#G' | G \longrightarrow^* G'\}$
$$time(G) = \begin{cases} 1 + time(G'), \ if\ G \longrightarrow G' \\ 0, \ otherwise \end{cases} \qquad \square$$

An evaluation sequence can stop in a non-final state. We write $G \not\longrightarrow$ if this happens. For example, because *black holing* lets us detect direct self-dependencies, ${}^a_\epsilon\{a \mapsto a\} \longrightarrow {}^a_s\{a \mapsto \bot, s \mapsto \#a\ \epsilon\} \not\longrightarrow$. At the operational level we can distinguish these *detectably divergent* programs from non-terminating ones.

5 Discussion and Applications

This section looks at the benefits and implications of the semantics for different users.

5.1 Education

Lazy evaluation is often explained as a kind of graph reduction (though without a stack component) and this is probably the favoured mental model of programmers. Our semantics are easily re-cast in graphical form, making them suitable for describing and investigating how evaluation works. Figure 4 illustrates the idea.

5.2 Application Programming

For someone using Haskell for a serious application (and therefore willing to work at fine-tuning the code for performance) a standard semantics offers some help. A implementation that complies with the standard guarantees a worst-case space and time complexity for a program. A non-compliant implementation could have completely different behaviour, so having a standard improves portability. If a program seems to be using too much space on a compliant implementation then the programmer can be sure it is his responsibility to improve the code. In practice things may not be this straight forward. What appears to be a leak may actually be a bad, but constant factor, inefficiency caused by a time/space tradeoff in the implementation. For locating sources of inefficiency, the semantics does not offer any great leap forward, the usual profiling tools are probably still the only practicable way to interrogate the operational behaviour of a complex program. The problem of predicting space requirements and optimising a program for better space efficiency by altering strictness, sharing and parameter order are still tricky tasks, the semantics can only offer a sound basis for attempting such tasks.

5.3 Implementing the Language

For an implementation to be compliant with the semantics we really require a proof showing that its abstract machine has the same asymptotic space behaviour as the semantics. This is difficult in general, though we have made some progress in comparing term-graph evaluators [BR00a]. Alternatively, implementors could follow Sestoft's approach, developing abstract machines from the semantics, this should also be supported with a proof that the changes do not affect space usage. Sestoft does not do that as such but he does introduce environment trimming to fix a particular fault. Mountjoy [Mou98] developed Launchbury's semantics until they became the STG-machine, he does not discuss space issues, though that seems more difficult at the natural semantics level.

In practical terms, the level of space-safety described by the semantics demands that the closure representation of an implementation does not extend the lifetime of variables beyond the point where the semantics can prove they are garbage. Efficient closure representation is discussed in Appel and Shao [AS96].

The effect of translation and transformation is a very important consideration for compiler writers and those investigating language extensions. Modern compilers include many complex transformations which often affect operational properties. For example, a *full laziness* transformation is often used to gain a higher degree of sharing. Some people argue that no transformation should be allowed to damage the time or space performance of a program, this may be too harsh and inhibit useful optimisations like full laziness. To be fair to programmers, compilers should allow programs to be viewed at a stage in compilation where they have their run-time operational properties and they are still easily related to the source code. Beyond this point, any transformations should preserve operational behaviour. Others are researching theories of operational equivalence for time and space behaviour [MS98, GS99].

6 Properties of the Semantics

Apart from using term graphs with their clear modelling of addressing, sharing and size, our rules differ from Sestoft's (reproduced in the Appendix) as follows: When we want the value of a variable x, *(Lookup)* moves the control root directly to x whereas (var_1) moves the expression at x into the control. Also in *(Lookup)*, we model *black holing* by overwriting a with $\perp$. If there is a direct self-dependency evaluation will stop in a non-final state. Sestoft's *(var)* rules achieve the same effect by removing x from the heap until its value is reached. That approach would be problematic in the term-graph framework because it leaves the graph open. *(Update)* would have to search the entire graph to restore any dangling pointers to x, so we could not claim that the semantics is a realistic model of execution time. Another consequence is that if we want to measure stack space separately (by counting S-nodes and X-nodes separately), Sestoft's model does not give an asymptotically correct X-node count. Gustavsson and Sands [GS99] encountered this problem, they solved it by counting both stack and heap space for update markers.

Proposition 1 guarantees that when a Core Graph is evaluated it remains closed and self-consistent in the sense of Definition 1. This validates our claim that *size* is a true reflection of space requirements and simplifies the argument for correctness.

Proposition 1 ($\longrightarrow$ preserves well-formedness)
If G is a Core Graph and $G \longrightarrow H$ then H is a Core Graph.
Proof: in all rules $L \longrightarrow R$, the free variables of R are a subset of the free variables of L and only nodes known to be garbage are removed. □

Launchbury [Lau93] showed his natural semantics correct with respect to a denotational one and Sestoft [Ses97] showed correctness with respect to Launchbury, so we can prove correctness by showing operational equivalence with Sestoft [Ses97].

Proposition 2 (Correctness)
If e has a value then there is a final state ${}^{a}_{\epsilon}G$ s.t. ${}^{a}_{\epsilon}\{a \mapsto e\} \longrightarrow^* {}^{a}_{\epsilon}G$ and ${}^{a}_{\epsilon}G$ encodes the value of e.
If e does not have a value then ${}^{a}_{\epsilon}\{a \mapsto e\}$ does not terminate or ${}^{a}_{\epsilon}\{a \mapsto e\} \longrightarrow^* {}^{x}_{s}H \not\longrightarrow$.
Proof: See Appendix. □

Another important property of the semantics is determinism. The rules define sequential reduction for Haskell. Non-deterministic or parallel evaluation rules may complicate space usage by creating or eliminating structures ahead of time. For a parallel call-by-need semantics see [HBFTK98].

Proposition 3 (Determinism)
$G \longrightarrow H \wedge G \longrightarrow H' \Rightarrow H \equiv H'$.
Proof: By rule left-pattern unification: no graph can match more than one pattern. □

Now we will look at the space and time usage model of the semantics. *space* (Definition 4) is an accurate asymptotic model, even though it ignores node *size*, by Proposition 4. This says that given a Core Graph, the graphs in its evaluation trace will never contain a bigger node than the nodes in the initial graph, which we assume are finite.

Proposition 4 (Space validity)
$\forall G, \exists k, \max\{size(G') \mid G \longrightarrow^* G'\} \leq k \times space(G)$.
Proof: No rule increases the maximum node size in the graph: If ${}^{x}_{s}G \longrightarrow^{y}_{t} H$ then $\max\{size(T) \mid (a \mapsto T) \in H\} \leq \max\{3, \max\{size(T) \mid (a \mapsto T) \in G\}\}$. By induction, the maximum node size of a graph is never increased by evaluation. □

We also claim that *time* is an accurate model of asymptotic time usage. This is is true in the sense described by Proposition 5. Once the initial graph is known the complexity of each step in its evaluation trace is bounded. Again, there is a time-space tradeoff. The inaccuracy here may not be as severe because real implementations typically use environments to delay substitution, so increasing the initial program size does not increase the constant overhead on each substitution step by as much.

Proposition 5 (Time complexity)
$\forall G, \exists k, G \longrightarrow^* H \longrightarrow H' \Rightarrow (H \longrightarrow H')$ *is* $O(k)$.
Proof: The complexity of a step is governed by the product of the number of rules, the maximum number of nodes in any rule pattern and the maximum node size during reduction. Once an initial graph is known all three are fixed for its evaluation by space validity. For this result we also needed every node in each rule left-pattern to be reachable from a root. □

7 Monadic IO

The semantics of IO are an important consideration for space analysis because excessive buffering can cause a space leak. Figure 5 is a simple extension of Core Haskell with monadic character IO, assuming single input and output files, inspired by Gordon's definition [Gor92]. The *effect* of a rule is written as a sequence of actions above the arrow. Our assumption that the library code presents only a constant overhead means that, by these rules, the IO mechanism is not allowed to buffer an unlimited amount of input (or output). Any such behaviour must be explicit in the program. With this definition we can also make a more useful definition of asymptotic space usage. Until now a program would either need bounded or unbounded space. With a model for input we can talk about space usage as a function of the number

$$
\begin{array}{rcll}
{}^{a}_{s}\{a \mapsto getChar\} & \stackrel{\langle get\ Ch\rangle}{\longrightarrow} & {}^{a}_{s}\{a \mapsto IO\ b, b \mapsto Ch\} & (GetChar) \\
{}^{a}_{s}\{a \mapsto putChar\ E\} & \longrightarrow & {}^{a}_{t}\{a \mapsto E, t \mapsto putChar : s\} & (PushPutChar) \\
{}^{a}_{s}\{a \mapsto Ch, s \mapsto putChar : t\} & \stackrel{\langle put\ Ch\rangle}{\longrightarrow} & {}^{a}_{t}\{a \mapsto IO\ b, b \mapsto ()\} & (ReducePutChar)
\end{array}
$$

Figure 5: Monadic IO Extension. We assume a unit data type with the constructor () and an $(IO\ \alpha)$ data type with an IO constructor. The expression grammar is extended with *putChar* and *getChar* according to: $X ::= \cdots \mid getChar \mid putChar\ X$. A *putChar* stack node is created while its argument is evaluated: $S ::= \cdots \mid putChar : s$. The semantics are extended with (*GetChar*) which reads a character Ch into a new node. (*PushPutChar*) evaluates the argument of a *putChar*; assuming it reaches a character value, (*ReducePutChar*) outputs it, returning IO ().

$$
\begin{array}{rll}
(*) & {}^{a}_{\epsilon}\{a \mapsto skips\ \epsilon, skips \mapsto \mathbf{Skips}\} & \\
\longrightarrow^4 & {}^{a}_{\epsilon}\{a \mapsto case\ getChar\ of\ \mathbf{Ias}, skips \mapsto \mathbf{Skips}\} & (Push), (Lookup), (Update), (Reduce) \\
\longrightarrow & {}^{a}_{s}\{a \mapsto getChar, skips \mapsto \mathbf{Skips}, s \mapsto \mathbf{Ias} : \epsilon\} & (PushCase) \\
\stackrel{\langle get\ C\rangle}{\longrightarrow} & {}^{a}_{s}\{a \mapsto IO\ b, b \mapsto C, skips \mapsto \mathbf{Skips}, e \mapsto \mathbf{Ias} : \epsilon\} & (GetCtr) \\
\longrightarrow & {}^{a}_{\epsilon}\{a \mapsto case\ b\ of\ \mathbf{Cas}, b \mapsto C, skips \mapsto \mathbf{Skips}\} & (ReduceCase) \\
\longrightarrow & {}^{a}_{t}\{a \mapsto b, b \mapsto C, skips \mapsto \mathbf{Skips}, t \mapsto \mathbf{Cas} : \epsilon\} & (PushCase) \\
\longrightarrow^2 & {}^{a}_{t}\{a \mapsto C, b \mapsto C, skips \mapsto \mathbf{Skips}, t \mapsto \mathbf{Cas} : \epsilon\} & (Lookup), (UpdateCtr) \\
\longrightarrow & \begin{cases} (*),\ if\ C = '\ ' \\ {}^{a}_{\epsilon}\{a \mapsto putChar\ b, b \mapsto C\},\ otherwise \end{cases} & (ReduceCase) \\
\longrightarrow & {}^{a}_{u}\{a \mapsto b, b \mapsto C, u \mapsto putChar : \epsilon\} & (PushPutChar) \\
\longrightarrow^2 & {}^{a}_{u}\{a \mapsto C, u \mapsto putChar : \epsilon\} & (Lookup), (UpdateCtr) \\
\stackrel{\langle put\ C\rangle}{\longrightarrow} & {}^{a}_{\epsilon}\{a \mapsto IO\ d, d \mapsto ()\} & (ReducePutChar)
\end{array}
$$

Figure 6: Evaluation trace of (7). The first 11 steps repeat while space characters are read. When a non-space character is read it is written out and the program terminates. Thus the program has $O(1)$ space complexity, never using more than 5 nodes and $O(n)$ time complexity where n is the number of spaces read.

of inputs read, i.e. the number of *get* effects in the evaluation trace.

Definition 5 (Asymptotic space usage) *The space usage of Core Graph G is $O(F)$ if* $\exists k, \forall H, G \stackrel{E}{\longrightarrow}{}^* H \Rightarrow$
$\#H \leq k \times (\lambda n.F)(\#\langle c | get\ c \in E\rangle + 1)$ □

Example 4 (Evaluation with IO)

```
let skips s = getChar >>= \c ->
 if c==' ' then skips s else putChar c
in skips undefined
```

The Haskell program above reads characters from the input until a non-space character is entered. It can be translated to the Core Graph (7). Figure 6 shows its evaluation trace.

$$ {}^{a}_{\epsilon}\{a \mapsto skips\ \epsilon, skips \mapsto \mathbf{Skips}\} \quad (7) $$

$\mathbf{Skips} = \lambda x. case\ getChar\ of\ \mathbf{Ias}$,
$\mathbf{Ias} = \{(IO\ c) \rightarrow case\ c\ of\ \mathbf{Cas}\}$,
$\mathbf{Cas} = \{'\ ' \rightarrow skips\ \epsilon\} \cup \{c \rightarrow putChar\ c \mid c \neq '\ '\}$.

□

8 Conclusions and Further Work

We have presented a graph-based operational semantics for Core Haskell. The semantics are based on the work of Sestoft, but reformulated in the term-graph framework to provide a more explicit model of space and time, suitable for asymptotic analysis. The semantics offer a starting point for learning about lazy evaluation since the graph approach lends itself to a diagrammatic presentation. The semantics give a minimum standard for the space and time usage of programs and a framework for investigating how programs behave. This helps in locating space faults, designing implementations and validating the operational behaviour of implementations.

We believe that most current implementations meet the standard of these semantics. Many include optimisations that will give better asymptotic space usage for some programs, so this formulation may actually be too cautious. Programs may still behave very differently on different implementations.

Several issues need further study. For example, the translation to Core. A standard translation would give a semantics to the full language, making it more useful to programmers. We have assumed that the space-semantics of class instances reduce to a constant overhead, but this may well not be adequate. The assumption that libraries only occupy a constant amount of space should also be verified.

We cannot claim that the semantics make operational analysis of programs easy. But as an accurate model which includes all the information pertaining to space and time usage, they are a good basis on which such techniques can be built. A full theory of space usage is still some way off but this semantics seems to be a useful starting point.

References

[App92] A W Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[AS96] A W Appel and Z Shao. Empirical and analytical study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, January 1996.

[BR00a] 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 Languages, Montreal*. ACM Press, September 2000.

[BR00b] A Bakewell and C Runciman. The space usage problem: An evaluation kit for graph-reduction semantics. In S Gilmore, editor, *Draft Proc. 2nd Scottish Functional Programming Workshop, School of Computer Science, University of St. Andrews*, pages 1–18, July 2000.

[Gor92] A D Gordon. *Functional Programming and Input/Output*. PhD thesis, University of Cambridge, August 1992.

[GS99] 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.

[Hask99] S L Peyton Jones and J Hughes (editors). *Haskell 98: A Non-Strict, Purely Functional Language*, February 1999.

[HBFTK98] J G Hall, C Baker-Finch, P Trinder, and D J King. An operational semantics for parallel lazy evaluation. In *Proc. 10th International Workshop on the Implementation of Functional Languages, University College London, UK. (IFL '98)*, volume 1595 of *LNCS*, pages 171–182. Springer-Verlag, September 1998.

[ICS91] New York IEEE Computer Society. IEEE Standard for the Scheme Programming Language. Technical report, IEEE standard 1178, 1991.

[Lau93] J Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages, Charleston*, pages 144–154, January 1993.

[MFH95] 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.

[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.

[Mou98] 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.

$$
\begin{array}{rcll}
\Gamma, e\,p, S & \Longrightarrow & \Gamma, e, p : S & (\mathit{app_1}) \\
\Gamma, \lambda y.e, p : S & \Longrightarrow & \Gamma, e[p/y], S & (\mathit{app_2}) \\
\Gamma[p \mapsto e], p, S & \Longrightarrow & \Gamma, e, \#p : S & (\mathit{var_1}) \\
\Gamma, \lambda y.e, \#p : S & \Longrightarrow & \Gamma[p \mapsto \lambda y.e], \lambda y.e, S & (\mathit{var_2}) \\
\Gamma, \mathit{let}\{x_i = e_i\}_{i=1}^n \ \mathit{in}\ e, S & \Longrightarrow & \Gamma[p_i \mapsto e_i[p_j/x_j]_{j=1}^n]_{i=1}^n, e[p_i/x_i]_{i=1}^n, S & (\mathit{let}) \\
\Gamma, c\,x_1 \cdots x_a, \#p : S & \Longrightarrow & \Gamma[p \mapsto c\,x_1 \cdots x_a], c\,x_1 \cdots x_a, S & (\mathit{var_3}) \\
\Gamma, \mathit{case}\ e\ \mathit{of}\ AS, S & \Longrightarrow & \Gamma, e, AS : S & (\mathit{case_1}) \\
\Gamma, c\,x_1 \cdots x_a, \{c\,y_1 \cdots y_a \to e\} \cup AS : S & \Longrightarrow & \Gamma, e[x_i/y_i]_{i=1}^a, S & (\mathit{case_2})
\end{array}
$$

Figure 7: Sestoft's mark 1 machine for lazy evaluation. The relation $\Longrightarrow$ maps a configuration comprising heap, control expression and stack components to a new configuration. A heap maps variables to expressions. Expressions $e \in \mathit{Term}(E)$ are built from the Core grammar (1) excluding $\perp$. The stack is a list which contains pushed arguments, alternative sets and update markers. The initial configuration for evaluating e is $(\emptyset, e, \epsilon)$.

[MS98] A Moran and D Sands. Improvement in a lazy context: An operational theory for call-by-need. Draft version, September 1998.

[Pey87] S L Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, 1987.

[Pey92] S L Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

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

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

[Wad87] P Wadler. Fixing some space leaks with a garbage collector. *Software — Practice & Experience*, 17(9):595–608, 1987.

Appendix: Correctness

We prove the semantics correct by showing they are operationally equivalent to Sestoft's in Figure 7.

Proposition 2 (Correctness)
If e has a value then there is a final state ${}^a_\epsilon G$ s.t. ${}^a_\epsilon\{a \mapsto e\} \longrightarrow^* {}^a_\epsilon G$ and ${}^a_\epsilon G$ encodes the value of e.
If e does not have a value then ${}^a_\epsilon\{a \mapsto e\}$ does not terminate or ${}^a_\epsilon\{a \mapsto e\} \longrightarrow^* {}^x_s H \not\longrightarrow$.
Proof: The relation $\leftrightarrow$ below relates every initial configuration of Sestoft's machine to an initial graph. By Proposition 6, related graphs remain related by evaluation or they both detectably diverge. □

$$\frac{s \notin \mathit{dom}\,G \quad {}^a_t G \leftrightarrow \Gamma, e, S}{{}^a_s G\{s \mapsto x : t\} \leftrightarrow \Gamma, e, x : S}(\leftrightarrow:)$$

$$\frac{a, s \notin \mathit{dom}\,G \quad {}^x_t G\{x \mapsto e\} \leftrightarrow \Gamma, e, S}{{}^a_s G\{a \mapsto e, s \mapsto \#x\,t, x \mapsto \perp\} \leftrightarrow \Gamma, e, \#a : S}(\leftrightarrow \#)$$

$$\frac{\{e, e_1, \ldots, e_n\} \subseteq \mathit{Term}(E)}{{}^a_\epsilon\{a \mapsto e\}\{x_i \mapsto e_i\}_{i=1}^n \leftrightarrow \{x_i \mapsto e_i\}_{i=1}^n, e, \epsilon}(\leftrightarrow \epsilon)$$

$\leftrightarrow$ relates a Core Graph to the equivalent configuration in Sestoft's semantics. The rules generate Sestoft's stack from the S-nodes and $\perp$ nodes in our graph, the remaining nodes form the heap. The only twist is in $(\leftrightarrow \#)$ where we swap x with a because we move the evaluation address into the graph in the *(Lookup)* rule whereas $(\mathit{var_1})$ copies the expression out of the heap. $(\leftrightarrow:)$ covers pushed arguments and alternative sets.

Proposition 6 (Evaluation preserves $\leftrightarrow$)
Whenever a graph relates to a configuration, both: are final states, evaluate in one step to related states or detectably diverge.
Proof: by cases on T and e using Lemmas 1–3, where $G = {}^x_s G'\{x \mapsto T\} \leftrightarrow (\Gamma, e, S) = C$. □

Lemma 1 *if $e, e' \in \mathit{Term}(E)$ then:*
${}^a_s G\{a \mapsto e\} \leftrightarrow \Gamma, e, S \Leftrightarrow {}^a_s G\{a \mapsto e'\} \leftrightarrow \Gamma, e', S$

Lemma 2 ${}^a_s G\{x \mapsto e\} \leftrightarrow \Gamma[x \mapsto e], e', S \Leftrightarrow$
$x \notin \mathit{dom}\,G \wedge {}^a_s G \leftrightarrow \Gamma, e', S \wedge e \in \mathit{Term}(E)$.

Lemma 3 $x \in \mathit{Term}(X) \cap \mathit{dom}\,G \wedge {}^a_s G \leftrightarrow \Gamma, e, S \Rightarrow$
$x \in \{a\} \cup \mathit{dom}\,\Gamma \vee (x \mapsto \perp) \in G$.

Proofs: by induction on the stack. □