J. Functional Programming **7** (3): 231–264, May 1997. Printed in the United Kingdom

Deriving a lazy abstract machine

PETER SESTOFT

Department of Mathematics and Physics,
Royal Veterinary and Agricultural University, Thorvaldsensvej 40,
DK-1871 Frederiksberg C, Denmark
(*e-mail:* `sestoft@dina.kvl.dk`)

Abstract

We derive a simple abstract machine for lazy evaluation of the lambda calculus, starting from Launchbury's natural semantics. Lazy evaluation here means non-strict evaluation with sharing of argument evaluation, i.e. call-by-need. The machine we derive is a lazy version of Krivine's abstract machine, which was originally designed for call-by-name evaluation. We extend it with datatype constructors and base values, so the final machine implements all dynamic aspects of a lazy functional language.

Capsule Review

As a kid I was fascinated by the drifting continents theory. It is obvious that there is some connection between the shapes of Africa and South America, and it was very satisfying to hear that there is indeed a plausible explanation. Abstract machines for functional languages are similar to the drifting continents. When you look at all the different machines around, you soon get the suspicion that there must be more than meets the eye. This paper is gratifying because it explains that, indeed, there is a coherent theory underlying these isolated islands. Another pleasing thing about this paper is that the particular abstract machine that it derives is small, simple, yet efficient.

1 Introduction

The development of an efficient abstract machine for lazy evaluation usually starts from either a graph reduction machine or an environment machine.

Graph reduction machines perform substitution by rewriting the term graph, that is, the program itself. Sharing of argument evaluation in an application is implemented by sharing the subgraph representing the argument term; this is 'obviously correct'. However, the rewriting of program subterms precludes efficient implementation on sequential imperative computers, so graph reduction machines must be refined to give efficient implementations. The machines become more complicated and less obviously correct; cf. the G-machine (Augustsson, 1984; Johnsson, 1984).

Environment machines perform substitution by updating the environment, mapping variables to terms instead of modifying the program. They resemble sequential imperative computers and therefore are 'obviously efficient'. However, they implement call-by-name evaluation, and so must be modified to implement sharing of

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

argument evaluation. The machines become more complicated and less obviously efficient; cf. the Three Instruction Machine TIM (Fairbairn and Wray, 1987).

Ingenious and efficient *hybrids* of graph reduction machines and environment machines exist, but since they are improvements of the others, they do not have their initial simplicity; cf. the Spineless Tagless G-machine (Peyton Jones, 1992).

In this paper we take a different approach. We develop an abstract machine for lazy evaluation from the natural semantics published by Launchbury (1993). Like the graph reduction schemes, this semantics accounts for sharing from the outset, yet an abstract machine can be derived from it easily.

1.1 Contributions

We show that known implementation techniques can be developed from a published natural semantics for lazy evaluation. We do so in a number of refinement steps, proving the correctness of the non-trivial ones.

Initially, we consider a simple language: the lambda calculus augmented with mutually recursive let-bindings. From a natural semantics for this language, we obtain an abstract machine, which is a naturally lazy version of the Krivine machine (Curien, 1988). The machine has four instructions, is simple, close to a sequential computer, and potentially efficient.

Extending the language and its natural semantics to include datatype constructors and base values, we show that well-known and efficient implementations can be derived for these too. The latter steps may be seen as reverse engineering of techniques used in the Spineless Tagless G-machine (Peyton Jones, 1992).

Thus, the development covers all dynamic aspects of a lazy functional language, but not the static aspects, such as type inference. We hope this demonstrates that a lazy functional language implementation can be at the same time clearly correct, small, understandable and efficient.

1.2 Motivation

One goal of this paper is pedagogical: to understand lazy evaluation. The extensional properties of lazy functional languages are simple and elegant; lazy languages satisfy more laws than strict ones. Conversely, the intensional aspects, such as space consumption and evaluation order, are harder to understand. In practice, intensional aspects are as important as extensional ones; an engineer should be able to estimate the properties, including time and space requirements, of programs, just as he or she understands the properties of other artefacts (bridges, power lines, computer hardware, etc.). The study of a model implementation, as developed in this paper, may help students understand the intensional aspects of lazy languages.

Similarly, a model implementation is useful for describing and experimenting with implementation design choices, in particular the representation of environments.

Another goal of the paper is to provide a foundation for intensional program analyses. For instance, an evaluation order analysis may determine at compile-time (an approximation to) the evaluation order of subexpressions. Any semantic

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

foundation for such an analysis must incorporate a notion of evaluation order, and should be true to a real implementation. An abstract machine with close links to an operational semantics seems a good candidate for a foundation.

The present work has already been put to such use. Sansom and Peyton Jones (1995) devised a *cost semantics*, an instrumentation of Launchbury's semantics with time and space costs. Using a modified version of our derivation and proof they show that an abstract machine with profiling, close to the actual implementation, is correct with respect to the cost semantics. Working at the level of the semantics, rather than that of the abstract machine, enables a precise discussion of alternative cost attributions.

1.3 Outline

In section 2 we present a small lazy functional language and its operational semantics, following Launchbury. In section 3 we first derive a simple abstract machine with an evaluation stack and prove its correctness with respect to the semantics. We then introduce an environment to avoid modifying the program being evaluated, and replace variable names by de Bruijn indices. In section 4 we identify and solve a problem with space consumption in the abstract machine. We then extend the language, the semantics, and the abstract machine with algebraic datatypes in section 5 and with base values in section 6. In section 7 we evaluate the resulting abstract machine, and in section 8 we discuss related work.

2 A lazy language and its natural semantics

By *lazy evaluation* we understand normal order reduction to weak head normal form (whnf), with sharing of argument evaluation.

2.1 Syntax

Launchbury (1993) presents a natural semantics for lazy evaluation of so-called *normalized* lambda expressions:

$$e \quad ::= \quad \lambda x.e \;\mid\; e\,x \;\mid\; x \;\mid\; let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e$$

The argument in an application must be a variable x; this ensures sharing of argument evaluation by requiring that non-trivial argument expressions are let-bound (and by treating let-bound expressions appropriately). General lambda expressions may be transformed into 'normalized' ones by the introduction of new let-bindings. Let-bindings are simultaneous and recursive, and all x_i in a let-binding must be distinct. We write $let\ \{x_i = e_i\}\ e$ for the let-binding $let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e$.

Throughout, $\equiv$ denotes syntactical identity of expressions, and $e[e'/x]$ denotes naïve simultaneous substitution of e' for all free occurrences of x in e:

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textit{Lam}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \qquad \Delta : e'[x/y] \Downarrow \Theta : w}{\Gamma : e\,x \Downarrow \Theta : w} \qquad \textit{App}$$

$$\frac{\Gamma : e \Downarrow \Delta : w}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto w] : \widehat{w}} \qquad \textit{Var}$$

$$\frac{\Gamma[x_i \mapsto e_i] : e \Downarrow \Delta : w}{\Gamma : \textit{let}\ \{x_i = e_i\}\ e \Downarrow \Delta : w} \qquad \textit{Let}$$

Fig. 1. Launchbury's natural semantics for lazy evaluation.

$$\begin{array}{lcll}
x[e'/x] & \equiv & e' & \\
y[e'/x] & \equiv & y & \text{if } x \not\equiv y \\
(\lambda x.e)[e'/x] & \equiv & \lambda x.e & \\
(\lambda y.e)[e'/x] & \equiv & \lambda y.e[e'/x] & \text{if } x \not\equiv y \\
(e_1\,e_2)[e'/x] & \equiv & (e_1[e'/x])\,(e_2[e'/x]) & \\
(\textit{let}\ \{x_i = e_i\}\ e)[e'/x] & \equiv & \textit{let}\ \{x_i = e_i\}\ e & \text{if } \exists i.x \equiv x_i \\
(\textit{let}\ \{x_i = e_i\}\ e)[e'/x] & \equiv & \textit{let}\ \{x_i = e_i[e'/x]\}\ e[e'/x] & \text{if } \forall i.x \not\equiv x_i
\end{array}$$

No implicit renaming of bound variables is assumed. We write $e[p_i/x_i]$ for the simultaneous substitution $e[p_1/x_1, \ldots, p_n/x_n]$, where the $x_1, \ldots, x_n$ must be distinct.

2.2 Launchbury's semantics

A *heap* or *store* $\Gamma = \{\ldots, x \mapsto e, \ldots\}$ is a mapping from variables x to expressions e. By *dom* Γ we denote the heap's domain (the set of variables x bound by Γ), by *rng* Γ its range (the set of expressions e bound in Γ), and by $\Gamma[x \mapsto e]$ the heap which maps x to e and any other variable y to $\Gamma[y]$. We write $\Gamma[p_i \mapsto e_i]$ for $\Gamma[p_1 \mapsto e_1, \ldots, p_n \mapsto e_n]$.

A *configuration* $\Gamma : e$ consists of a heap Γ and an expression e to be evaluated. A *judgement* $\Gamma : e \Downarrow \Theta : w$ says that in the heap Γ, the expression e will evaluate to the value w, producing the new heap Θ. Launchbury defines the relation $\Downarrow$ on configurations by a set of inference rules, that is, an operational semantics, reproduced in Figure 1. He proved this semantics correct with respect to a denotational one.

Rule *Lam*: a lambda abstraction $\lambda x.e$ is already a value (in whnf) and therefore evaluates to itself; the heap Γ is unmodified.

Rule *App*: an application $(e\,x)$ is evaluated in heap Γ by evaluating e to a lambda abstraction $\lambda y.e'$, producing a new heap Δ. The argument x is substituted for the formal parameter y in e', and $e'[x/y]$ is evaluated in the heap Δ to obtain the final result w and final heap Θ.

Rule *Var*: a variable x which is bound to expression e in heap $\Gamma[x \mapsto e]$ is evaluated by evaluating e in Γ to obtain a value w and new heap Δ. To achieve laziness (that is, avoid re-evaluation of e), the heap must be updated with the reduced value w, giving the heap $\Delta[x \mapsto w]$. This duplicates the expression w and might cause name clashes later, so all bound variables in the other copy of w are replaced by fresh variables, as indicated by the renaming notation $\widehat{w}$. Observe that e is evaluated in heap Γ which has no binding for x. If the evaluation of e to whnf requires x, then the evaluation fails, indicating a 'black hole', a direct self-dependency in the program, as in $let\ x = x\ in\ x$ or $let\ x = y,\ \ y = x\ in\ x$.

Rule *Let*: a let-binding $let\ \{x_i = e_i\}\ in\ e$ is evaluated by binding the expressions e_i to the variables x_i, and then evaluating e in the resulting heap $\Gamma[x_i \mapsto e_i]$.

A program is a closed expression e in which all bound variables are distinct. The value w of a program e is computed by finding a derivation of $\{\} : e \Downarrow \Delta : w$. An easy induction shows that the result w, if any, will be a lambda abstraction $\lambda y.e'$.

The semantics rules are deterministic modulo variable renaming, and essentially sequential: to build a derivation tree, one must determine the final heap of any left-hand premise before proceeding to any right-hand premise.

2.3 Properties of Launchbury's semantics

The semantics rules adequately model lazy evaluation: no function argument is evaluated more than once, and no unevaluated expression gets duplicated.

Namely, in a normalized expression (see section 2.1), any non-trivial function argument e must be bound to a let-variable x, and no such let-bound expression is evaluated to whnf twice. At its first use, the let-bound x is removed from the heap Γ, evaluated, and then rebound to the whnf w of e. An attempt to refer to x again before its rebinding will fail; rule *Var* is not applicable unless x is bound in the heap. After the rebinding, every subsequent use of x will just retrieve the whnf from the heap, via rules *Var* and *Lam*.

Furthermore, only expressions which are in whnf are ever duplicated. In the *App* rule, $e'[x/y]$ substitutes a variable x for another variable y, and therefore can duplicate only variables. In the *Var* rule, the duplicated expression w is a whnf of form $\lambda y.e'$.

Thus the rules model lazy evaluation. However, the duplication of $w \equiv \lambda y.e'$ in rule *Var* violates *full laziness* (Peyton Jones, 1987, Chapter 15), since e' may contain a redex with no free occurrences of y. Such a redex may be wastefully re-evaluated at each application of $w \equiv \lambda y.e'$. If desired, full laziness can be obtained by introducing a new let-binding for every maximal free expression: recursively replace $\lambda y.(\ldots e \ldots)$, where e is a maximal free expression, by $let\ x = e\ in\ \lambda y.(\ldots x \ldots)$, where x is a fresh variable.

The semantics must avoid variable capture in the naïve substitution $e'[x/y]$ in the *App* rule, and must avoid rebinding of any variable x already bound in heap Δ in the *Var* rule. Launchbury ensures this by requiring all bound variables in a program

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

e to be distinct, and by renaming all bound variables at the only point where an expression w containing bound variables may be duplicated: the *Var* rule.

However, this renaming strategy is unsuitable as basis for an abstract machine design. One problem is that variable freshness is not locally checkable in the *Var* rule of Figure 1, because a binding $x \mapsto e$ may have been deleted from Γ in a previous application of the *Var* rule. To see this, evaluate the expression

$$let\ s = \lambda z.z\ in\ let\ p = (q\, s),\ q = (\lambda y.let\ r = y\ in\ r)\ in\ p$$

in an empty heap. When applying the *Var* rule to evaluate and then re-bind q during the evaluation of p, it seems possible to rename r to p, because p occurs in no local expression or heap. But this would cause trouble later when re-binding the evaluated p in the heap Δ, invalidating Launchbury's (1993, p. 149) Theorem 1 about distinct naming.

Hence, to check freshness, one must inspect all of the derivation tree built so far, which is undesirable. Furthermore, it is unnecessary to rename *all* bound variables of w in the *Var* rule; a more economical approach is adequate, and preferable in an implementation.

2.4 Revising the semantics

We therefore change the renaming machinery, obtaining the revised semantics in Figure 2. It is identical to Launchbury's in all respects other than renaming. In the next section we show that it avoids variable capture.

First, we make freshness locally checkable by extending each judgement with the set A of the variables x left out of Γ in the premise of the *Var* rule. Intuitively, A is the set of variables whose values are currently being computed.

Secondly, we move the renaming from the *Var* rule to the *Let* rule, and rename only let-bound variables. Renaming $let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e$ with fresh variables $p_1, \ldots, p_n$ gives the expression $let\ p_1 = \hat{e}_1, \ldots, p_n = \hat{e}_n\ in\ \hat{e}$, where $\hat{e}$ denotes the result of the naïve substitution $e[p_1/x_1, \ldots, p_n/x_n]$. Since all x_i are distinct, this is well-defined. A variable p is *fresh* if it does not occur in A or Γ or $let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e$. Intuitively, the introduction of fresh variables corresponds to allocation of unused heap addresses $p_1, \ldots, p_n$, and is therefore naturally done in the *Let* rule rather than the *Var* rule.

The value (whnf) w of a closed expression e is computed by finding a derivation of $\{\} : e \Downarrow_{\{\}} \Delta : w$ using the revised semantics rules.

Henceforth we distinguish the *heap pointers* p introduced by the *Let* rule from the *program variables* x originating from the expression to be evaluated. The soundness of this distinction follows from the following observation: in a derivation tree for evaluation of a closed expression, all occurrences of heap pointers p in expressions are free (with p possibly bound in the heap), and all occurrences of program variables x are bound by let or lambda. This is proved below.

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

$$\Gamma : \lambda x.e \Downarrow_A \Gamma : \lambda x.e \qquad \textit{Lam}$$

$$\frac{\Gamma : e \Downarrow_A \Delta : \lambda y.e' \qquad \Delta : e'[p/y] \Downarrow_A \Theta : w}{\Gamma : e\,p \Downarrow_A \Theta : w} \qquad \textit{App}$$

$$\frac{\Gamma : e \Downarrow_{A \cup \{p\}} \Delta : w}{\Gamma[p \mapsto e] : p \Downarrow_A \Delta[p \mapsto w] : w} \qquad \textit{Var}$$

$$\frac{\Gamma[p_i \mapsto \hat{e}_i] : \hat{e} \Downarrow_A \Delta : w}{\Gamma : \mathit{let}\ \{x_i = e_i\}\ e \Downarrow_A \Delta : w}\,(*) \qquad \textit{Let}$$

(∗) In the *Let* rule, the variables $p_1, \ldots, p_n$ must be distinct and fresh: they must not occur in A or Γ or $\mathit{let}\ \{x_i = e_i\}\ e$. The notation $\hat{e}$ means $e[p_1/x_1, \ldots, p_n/x_n]$.

Fig. 2. Revised natural semantics for lazy evaluation.

2.5 Properties of the revised semantics

We must show that there is no variable capture in the substitution $e'[p/y]$ in rule *App*, and no rebinding of a variable p already bound in heap Δ in rule *Var*.

For a given expression e, let $Bv(e)$ denote its let- or lambda-bound variables and $Fv(e)$ its free variables, and extend this notation to configurations:

$$\begin{array}{rcl} Bv(\Gamma : e) & = & Bv(e) \ \cup \ \bigcup\{\ Bv(e') \mid e' \in \mathit{rng}\,\Gamma\ \} \\ Fv(\Gamma : e) & = & Fv(e) \ \cup \ \bigcup\{\ Fv(e') \mid e' \in \mathit{rng}\,\Gamma\ \} \end{array}$$

The revised semantics satisfies: In a derivation of $\{\} : e \Downarrow_{\{\}} \Delta : w$ where e is closed, all bound variables are program variables x from e, and all free variables are heap pointers p introduced by the *Let* rule. We now formalize this.

Definition 1
Let A be a set of variables. The configuration $\Gamma_0 : e_0$ is A-*good* if

(1) A and $\mathit{dom}\,\Gamma_0$ are disjoint;
(2) $Fv(\Gamma_0 : e_0) \subseteq A \cup \mathit{dom}\,\Gamma_0$; and
(3) $Bv(\Gamma_0 : e_0)$ and $A \cup \mathit{dom}\,\Gamma_0$ are disjoint.

In the context of a judgement $\Gamma_0 : e_0 \Downarrow_A \Gamma_1 : e_1$, these requirements say: (1) no variable whose value is being computed is also bound in Γ_0; (2) every free variable of e_0 is either being computed or is bound in Γ_0: there are no dangling pointers; and (3) program variables and pointers are distinct in e_0.

Definition 2
The judgement $\Gamma_0 : e_0 \Downarrow_A \Gamma_1 : e_1$ is *promising* if $\Gamma_0 : e_0$ is A-good.

In a derivation of a promising judgement, the distinction between program variables and pointers is maintained everywhere:

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

Lemma 1
Assume the judgement $\Gamma_0 : e_0 \Downarrow_A \Gamma_1 : e_1$ has a derivation. If $\Gamma_0 : e_0$ is A-good, then $\Gamma_1 : e_1$ is A-good, $dom\,\Gamma_0 \subseteq dom\,\Gamma_1$, and every judgement in the derivation is promising.

Proof
By induction on the structure of the derivation. □

Proposition 1
Let e be a closed expression and consider a derivation of $\{\} : e \Downarrow_{\{\}} \Theta : w$. In no instance of rule *App* can there be any variable capture in $e'[p/y]$, and in no instance of rule *Var* is p already bound in Δ.

Proof
Since e is closed, $\{\} : e$ is $\{\}$-good by definition. By Lemma 1, every judgement $\Gamma : e_0 \Downarrow_A \Delta : e_1$ in the derivation is promising, and in every such judgement, the configurations $\Gamma : e_0$ and $\Delta : e_1$ are A-good. It follows that:

In every instance of the *App* rule, $p \in Fv(\Gamma : e\,p) \subseteq A \cup dom\,\Gamma \subseteq A \cup dom\,\Delta$ because $\Gamma : e\,p$ is A-good. Since $\Delta : \lambda y.e'$ is A-good, $Bv(\Delta : \lambda y.e')$ is disjoint from $A \cup dom\,\Delta$, so $p \notin Bv(\lambda y.e')$: variable p is not bound in $\lambda y.e'$. Hence p cannot be captured in the substitution $e'[p/y]$.

In every instance of the *Var* rule, $A \cup \{p\}$ and $dom\,\Delta$ are disjoint since $\Delta : w$ is $(A \cup \{p\})$-good. Hence p is not already bound in Δ. □

Proposition 1 says that if expression e is closed, then there can be no variable capture or rebinding in a derivation of $\{\} : e \Downarrow_{\{\}} \Delta : w$ which computes the value w of e. The revised renaming strategy is adequate. Note that the bound variables of e need not be distinctly named.

To illustrate renaming, let us evaluate the expression $let\ x = x\ in\ (\lambda y.\lambda x.y)\,x$, in which one should not naïvely substitute x for y in the beta-reduction:

$$\dfrac{\dfrac{}{\{p\mapsto p\} : \lambda y.\lambda x.y \ \Downarrow_{\{\}}\ \{p\mapsto p\} : \lambda y.\lambda x.y}\,\mathit{Lam} \qquad \dfrac{}{\{p\mapsto p\} : \lambda x.p \ \Downarrow_{\{\}}\ \{p\mapsto p\} : \lambda x.p}\,\mathit{Lam}}{\dfrac{\{p\mapsto p\} : (\lambda y.\lambda x.y)\,p \ \Downarrow_{\{\}}\ \{p\mapsto p\} : \lambda x.p}{\{\} : let\ x = x\ in\ (\lambda y.\lambda x.y)\,x \ \Downarrow_{\{\}}\ \{p\mapsto p\} : \lambda x.p}\,\mathit{Let}}\,\mathit{App}$$

In the premise of the *Let* rule, a fresh variable p has been substituted for x in the expressions x and $(\lambda y.\lambda x.y)\,x$. In the second premise of the *App* rule, the argument p has been substituted for y in $\lambda x.y$; there is no variable capture.

3 Deriving a simple abstract machine

Operationally speaking, evaluation in the natural semantics builds a derivation tree for $\{\} : e \Downarrow_{\{\}} \Delta : w$ from the bottom up, whereas computation by an abstract machine builds a state sequence. We must turn the recipe for tree construction (Figure 2) into a recipe for state sequence construction (Figure 3).

The challenge lies in the representation of the context of subtrees. For instance, when applying the *Var* rule, we build a subtree for the premise, and the context (the rule) tells us that we must update the heap Δ at p with the computed value w

afterwards to create $\Delta[p \mapsto w]$. This context must be made explicit in the abstract machine: after the state sequence corresponding to the subtree, we must remember to update the heap at p.

Only the *App* and *Var* rules change the context of subtrees. In the *App* rule, we must remember to build a second subtree after building the first one, and in the *Var* rule, we must remember to update the heap after building the subtree. Although a subtree is built in the *Let* rule, the result $\Delta : w$ of the subtree is also the result of the entire tree, so no new context is needed.

The first step towards an abstract machine is to make the context (or continuation) of each subtree explicit in the state. When the context is extensible as here (e.g. if t' is a subtree of t, then the context of t' is an extension of the context of t), it is convenient to use a stack for representing the context. This is a long-standing tradition: the control context of a procedure invocation in Algol or Pascal is represented by a stack of return addresses.

3.1 First step: introduce a stack

A state of the first abstract machine is a triple (Γ, e, S) where Γ and e are just the heap and the expression of a configuration $\Gamma : e$, and S is a stack which represents the context of this configuration: part of a surrounding derivation tree. Traditionally, the e component is called the *control* of the abstract machine.

- The *Lam* rule does not give rise to any machine rules; no machine action is required to leave the heap and expression as they are.
- The *App* rule from the natural semantics gives rise to two machine rules: app_1 which begins the computation corresponding to the left subtree, and app_2 which begins the computation corresponding to the right subtree.
- The *Var* rule gives rise to two machine rules: var_1 which begins the computation corresponding to the subtree, and var_2 which updates the heap with the computed result.
- The *Let* rule gives rise to the machine rule *let* which allocates new heap addresses and begins the computation corresponding to the subtree.

The stack S is a list of *arguments* p and *update markers* $\#p$. A reduced value w must be a lambda abstraction $\lambda y.e$. Whenever the control is a lambda abstraction $\lambda y.e$, we must examine the stack top element, and act accordingly:

- an argument p on the stack top is a reminder that $\lambda y.e$ must be applied to p, that is, $e[p/y]$ must be evaluated. This is done by the app_2 rule.
- an update marker $\#p$ on the stack top is a reminder that the heap must be updated with $[p \mapsto \lambda y.e]$. This is done by the var_2 rule. Intuitively, $\lambda y.e$ is the value of some expression that was previously bound to p in the heap; to achieve sharing, the binding of p must be updated with this reduced value.

Values other than lambda abstractions will be considered later; they must check for update markers $\#p$ also, to correctly implement the *Var* rule. The first version of the

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

	Heap	Control	Stack	rule
	Γ	$(e\,p)$	S	app_1
$\Longrightarrow$	Γ	e	$p : S$	
	Γ	$\lambda y.e$	$p : S$	app_2
$\Longrightarrow$	Γ	$e[p/y]$	S	
	$\Gamma[p \mapsto e]$	p	S	var_1
$\Longrightarrow$	Γ	e	$\#p : S$	
	Γ	$\lambda y.e$	$\#p : S$	var_2
$\Longrightarrow$	$\Gamma[p \mapsto \lambda y.e]$	$\lambda y.e$	S	
	Γ	$let\{x_i = e_i\}e$	S	let (*)
$\Longrightarrow$	$\Gamma[p_i \mapsto \hat{e}_i]$	$\hat{e}$	S	

(*) In the *let* rule, the notation $\hat{e}$ means $e[p_1/x_1, \dots, p_n/x_n]$, where variables $p_1, \dots, p_n$ must be distinct and fresh: they must not occur in Γ or $let\ \{x_i = e_i\}\ e$ or S.

Fig. 3. Abstract machine mark 1, with stack.

abstract machine is shown in Figure 3. The set of update markers $\#p$ in the stack corresponds closely to the set A used in the revised natural semantics rules.

The initial heap and the stack are empty, so the initial state is $(\{\}, e, [\,])$. The machine terminates when no rule applies. Hence a terminal state either has form $(\Gamma, \lambda y.e, [\,])$, representing successful termination with result $\lambda y.e$, or form (Γ, p, S) where $p \notin dom\ \Gamma$, representing the discovery of a black hole in the program. The rules are deterministic modulo the choice of fresh variables.

In the terminology of Peyton Jones (1992, Section 3.2), the natural semantics in Figure 2 is an *eval/apply* model, whereas the abstract machine in Figure 3 is a *push/enter* model. The former evaluates an application $(e\,p)$ by first evaluating e, then applying the result to p. The latter evaluates $(e\,p)$ by first pushing the argument p, then entering the function e. The derivation above and the correctness proof below show that these models are closely related.

To see the machine in action, and to show how it achieves sharing of subcomputations, consider the evaluation of $let\ y = \lambda x.x,\ v = (\lambda z.z)\,y\ in\ v\,v$ in Figure 4. The final result is $\lambda x.x$, as expected. In the first step, fresh heap pointers p and q are allocated and substituted for free occurrences of y and v. The redex $(\lambda z.z)\,y$ is reduced only once, although it is used twice in the let-body. The last application of rule var_2 needlessly overwrites an expression which is already in whnf. Such *identical updates* could be avoided by not pushing an update marker when accessing a whnf (by rule var_1). This refinement of the machine may be introduced at a later stage, if desired.

3.2 Correctness of the first abstract machine

The correctness of the abstract machine with respect to the revised semantics is established by Proposition 2 below, but first we need some auxiliary properties. Let

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

	Heap Γ	Control e	Stack S	(rule)
	$[\,]$	$let\ y = \lambda x.x,\ v = (\lambda z.z)\, y\ in\ v\, v$	$[\,]$	
$\Rightarrow$	$\{p \mapsto \lambda x.x, q \mapsto (\lambda z.z)\, p\}$	$q\, q$	$[\,]$	(let)
$\Rightarrow$	$\{p \mapsto \lambda x.x, q \mapsto (\lambda z.z)\, p\}$	q	$[q]$	(app_1)
$\Rightarrow$	$\{p \mapsto \lambda x.x\}$	$(\lambda z.z)\, p$	$[\#q, q]$	(var_1)
$\Rightarrow$	$\{p \mapsto \lambda x.x\}$	$\lambda z.z$	$[p, \#q, q]$	(app_1)
$\Rightarrow$	$\{p \mapsto \lambda x.x\}$	p	$[\#q, q]$	(app_2)
$\Rightarrow$	$\{\}$	$\lambda x.x$	$[\#p, \#q, q]$	(var_1)
$\Rightarrow$	$\{p \mapsto \lambda x.x\}$	$\lambda x.x$	$[\#q, q]$	(var_2)
$\Rightarrow$	$\{p \mapsto \lambda x.x, q \mapsto \lambda x.x\}$	$\lambda x.x$	$[q]$	(var_2)
$\Rightarrow$	$\{p \mapsto \lambda x.x, q \mapsto \lambda x.x\}$	q	$[\,]$	(app_2)
$\Rightarrow$	$\{p \mapsto \lambda x.x\}$	$\lambda x.x$	$[\#q]$	(var_1)
$\Rightarrow$	$\{p \mapsto \lambda x.x, q \mapsto \lambda x.x\}$	$\lambda x.x$	$[\,]$	(var_2)

Fig. 4. Example evaluation with abstract machine mark 1.

$ap(S) = \{q \mid q \text{ is in } S\}$ stand for the set of argument pointers on the stack S, and let similarly $\#(S) = \{q \mid \#q \text{ is in } S\}$ stand for the set of update markers on the stack S.

Lemma 2

For all Γ, e, A, S, Δ, and e' such that $\Gamma : e$ is A-good, $A = \#(S)$, and $ap(S) \subseteq \#(S) \cup dom\,\Gamma$, if $\Gamma : e \Downarrow_A \Delta : e'$ is derivable then $(\Gamma, e, S) \Rightarrow^* (\Delta, e', S)$.

Proof

By induction on the derivation of $\Gamma : e \Downarrow_A \Delta : e'$. Lemma 1 ensures that all premises are promising, and that $ap(S) \subseteq \#(S) \cup dom\,\Delta$.

Case *Lam*: The *Lam* rule says that $\Gamma : \lambda y.e \Downarrow_A \Gamma : \lambda y.e$; but clearly $(\Gamma, \lambda y.e, S) \Rightarrow^* (\Gamma, \lambda y.e, S)$ by the empty sequence of computation steps.

Case *App*: Assume $\Gamma : e\,p \Downarrow_A \Theta : w$ by rule *App*. Note that $ap(p : S) = \{p\} \cup ap(S) \subseteq Fv(\Gamma : (e\,p)) \cup ap(S) \subseteq \#(S) \cup dom\,\Gamma$ by Lemma 1, so the induction hypothesis applies in line three below.

$$
\begin{array}{lll}
 & (\Gamma, (e\,p), S) & \\
\Rightarrow & (\Gamma, e, p : S) & \text{by rule } app_1 \\
\Rightarrow^* & (\Delta, \lambda y.e', p : S) & \text{by left premise and ind. hyp.} \\
\Rightarrow & (\Delta, e'[p/y], S) & \text{by rule } app_2 \\
\Rightarrow^* & (\Theta, w, S) & \text{by right premise and ind. hyp.}
\end{array}
$$

Case *Var*: Assume $\Gamma[p \mapsto e] : p \Downarrow_A \Delta[p \mapsto w] : w$ by rule *Var*. Observe that w is necessarily a lambda abstraction $\lambda y.e'$. Moreover, for line three below, note that $A \cup \{p\} = \#(\#p : S)$, so the induction hypothesis applies.

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

$$\begin{array}{lll} & (\Gamma[p \mapsto e], p, S) & \\ \Rightarrow & (\Gamma, e, \#p : S) & \text{by rule } var_1 \\ \Rightarrow^* & (\Gamma, \lambda y.e', \#p : S) & \text{by the premise and ind. hyp.} \\ \Rightarrow & (\Gamma[p \mapsto \lambda y.e'], \lambda y.e', S) & \text{by rule } var_2 \end{array}$$

Case *Let*: Assume $\Gamma : let\ \{x_i = e_i\}\ e \Downarrow_A \Delta : w$ by rule *Let*, and that $p_1, \ldots, p_n$ do not occur in A or Γ or $let\ \{x_i = e_i\}\ e$. Then they do not occur in S, that is, in $\#(S)$ or $ap(S)$, so

$$\begin{array}{lll} & (\Gamma, let\ \{x_i = e_i\}\ e, S) & \\ \Rightarrow & (\Gamma[p_i \mapsto \hat{e}_i], \hat{e}, S) & \text{by rule } let \\ \Rightarrow^* & (\Delta, w, S) & \text{by the premise and ind. hyp.} \end{array}$$

□

The lemma shows that the abstract machine can simulate derivations by the natural semantics. To show the converse, that it computes no more results than the natural semantics, we introduce the concept of balanced computation. The intention is that a balanced computation corresponds to a derivation (sub)tree.

We say that stack S' *extends* stack S if $S' = r_1 : \ldots : r_n : S$ for some stack objects $r_1, \ldots, r_n$, where $n \geq 0$.

Definition 3
A *balanced computation* is a computation $(\Gamma, e, S) \Rightarrow^* (\Delta, e', S)$ in which the initial and final stacks are the same, and in which every intermediate stack extends the initial one.

Every successful computation $(\{\}, e_0, []) \Rightarrow^* (\Gamma, \lambda y.e', [])$ is balanced. We want to prove that for every balanced computation (satisfying some further restrictions) there is a corresponding derivation tree. First, we show that a balanced computation has a simple structure.

Definition 4
The *trace* of a computation $(\Gamma_0, e_0, S_0) \overset{tr_1}{\Longrightarrow} (\Gamma_1, e_1, S_1) \overset{tr_2}{\Longrightarrow} \cdots \overset{tr_n}{\Longrightarrow} (\Gamma_n, e_n, S_n)$ where $n \geq 0$, is the sequence $tr_1, tr_2, \ldots, tr_n$ of transition rules used. A *balanced trace* is the trace of a balanced computation.

What are the possible forms of balanced traces? The empty trace, having one state and no transitions, is balanced. Assume the initial stack is S. Any non-empty balanced trace must begin with app_1, var_1, or let, since app_2 or var_2 would produce an intermediate stack which is not an extension of S.

If the trace begins with app_1, producing an intermediate stack of form $p : S$, then eventually an app_2 transition must occur which restores the stack to S; no other transition can do this. The subtrace between app_1 and the first occurrence of app_2 is balanced (with stacks which are extensions of $p : S$), and the subtrace following it is balanced (with stacks which are extensions of S). Hence the trace has the form $app_1\ bal\ app_2\ bal$, where bal stands for arbitrary balanced traces.

If the trace begins with var_1, producing an intermediate stack of form $\#p : S$, then eventually a var_2 transition must occur which restores the stack to S. The

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

subtrace between var_1 and the first occurrence of var_2 is balanced. Furthermore, since the control (before and) after var_2 must be $\lambda y.e$, only an app_2 or var_2 transition could follow, but either would remove an element from the stack and contradict the balancedness of the trace. Hence the occurrence of var_2 is the last element of the trace, which must have the form $var_1\ bal\ var_2$.

If the trace begins with let, then the subtrace after let must be balanced, so the trace has form $let\ bal$.

In summary, all balanced traces can be derived from the following grammar:

$$bal \ ::= \ \epsilon \ \mid \ app_1\ bal\ app_2\ bal \ \mid \ var_1\ bal\ var_2 \ \mid \ let\ bal$$

The four possibilities correspond to the four natural semantics rules *Lam*, *App*, *Var*, and *Let* in Figure 2. Thus potentially, a balanced trace has the same structure as a derivation tree; the following lemma gives a formal proof, which shows that the natural semantics can simulate any balanced computation of the machine.

Lemma 3
For all Γ_0, e_0, S, Γ_1, w, and A it holds that if $(\Gamma_0, e_0, S) \Rightarrow^* (\Gamma_1, w, S)$ is balanced, and $w \equiv \lambda v.e_1$, and $A = \#(S)$ then $\Gamma_0 : e_0 \Downarrow_A \Gamma_1 : w$ is derivable.

Proof
By induction on the structure of balanced traces, following the grammar.

Case ϵ: follows by rule *Lam* because we have $\Gamma_0 = \Gamma_1$ and $e_0 \equiv w \equiv \lambda v.e_1$.

Case $app_1\ bal\ app_2\ bal$: We must have $e_0 \equiv (e\,p)$. The state after app_1 must be $(\Gamma_0, e, p : S)$ and the state before app_2 must be $(\Delta, \lambda y.e', p : S)$. Since the trace between these is balanced, $\Gamma_0 : e \Downarrow_A \Delta : \lambda y.e'$ is derivable by the induction hypothesis. The state after app_2 is $(\Delta, e'[p/y], S)$, and the trace of $(\Delta, e'[p/y], S) \Rightarrow^* (\Gamma_1, w, S)$ is balanced, so $\Delta : e'[p/y] \Downarrow_A \Gamma_1 : w$ is derivable by the induction hypothesis. Using the *App* rule, we conclude that $\Gamma_0 : e\,p \Downarrow_A \Gamma_1 : w$ is derivable.

Case $var_1\ bal\ var_2$: We must have $e_0 \equiv p$ and $\Gamma_0 = \Gamma[p \mapsto e]$ and $\Gamma_1 = \Delta[p \mapsto \lambda v.e_1]$ for some p, Γ, e, and Δ. The state after var_1 is $(\Gamma, e, \#p : S)$, and the state before var_2 is $(\Delta, w, \#p : S)$, with $w \equiv \lambda v.e_1$. Since the subtrace between these is balanced, $\Gamma : e \Downarrow_{A \cup \{p\}} \Delta : w$ is derivable by the induction hypothesis, which is applicable because $A \cup \{p\} = \#(\#p : S)$. Using the *Var* rule we find that $\Gamma[p \mapsto e] : p \Downarrow_A \Delta[p \mapsto w] : w$ is derivable.

Case $let\ bal$: We must have $e_0 \equiv let\ \{x_i = e_i\}\ e$, and for renaming of $\hat{e}$ some $p_1, \ldots, p_n$ have been chosen that do not occur in Γ_0 or $let\ \{x_i = e_i\}\ e$ or S. The state after let is $(\Gamma_0[p_i \mapsto \hat{e}_i], \hat{e}, S)$, and the trace after let is balanced, so $\Gamma_0[p_i \mapsto \hat{e}_i] : \hat{e} \Downarrow_A \Gamma_1 : w$ is derivable by the induction hypothesis. Since the $p_1, \ldots, p_n$ do not occur in Γ_0 or A or $let\{x_i = e_i\}e$, we can use rule *Let* to conclude that $\Gamma_0 : let\{x_i = e_i\}e \Downarrow_A \Gamma_1 : w$ is derivable. □

Proposition 2
Assume $w \equiv \lambda y.e_1$. Then $(\{\}, e_0, []) \Rightarrow^* (\Delta, w, [])$ if and only if $\{\} : e_0 \Downarrow_{\{\}} \Delta : w$ is derivable.

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

Proof
If: follows from Lemma 2 because $\{\} : e_0 \Downarrow_{\{\}} \Delta : w$ is derivable and promising. Only if: observe that the computation $(\{\}, e_0, [\,]) \Rightarrow^* (\Delta, w, [\,])$ is necessarily balanced; then the result follows from Lemma 3. □

Proposition 2 says that the abstract machine terminates with a value $\lambda v.e_1$ in the control if and only if the natural semantics successfully derives this value. Both systems are deterministic, modulo the choice of fresh variables in the *Let* and *let* rules. As a second possibility, the machine may terminate with a heap pointer p in the control, indicating a 'black hole', in which case the natural semantics permits no derivations at all. As a third possibility, the machine may embark on an infinite computation, corresponding to an infinite derivation tree in the natural semantics. Determinism implies that these possibilities are mutually exclusive.

In retrospect it is not surprising that proving Lemma 2 requires less machinery than Lemma 3. In the former proof we must create a sequence from a tree, which is just a matter of recursive flattening. In the latter proof we must create a tree from a sequence, which is harder: the notion of balanced trace is a device to recover the tree structure from the sequence.

3.3 Second step: introduce environments and closures

The abstract machine above uses substitution in the expression e to model application (in the app_2 rule) and renaming (in the *let* rule). This is unsatisfactory from an implementation point of view because it modifies the expression at run-time. To avoid this, we introduce an *environment* E, which is a mapping from program variables x to heap pointers p. The environment can be thought of as a delayed substitution, which is not applied until we meet a program variable x in the control.

When we would previously perform the substitution $e[p/y]$, obtaining a new expression e', we will now build the pair $(e, \{y \mapsto p\})$ of the expression e and the environment $\{y \mapsto p\}$. In general, where we previously had an expression e', we shall now have a pair (e, E) of an expression e and an environment E such that e' is eE, the result of applying the substitution E to the expression e. The pair (e, E) is traditionally called a *closure*.

Henceforth the heap maps pointers to closures (e, E) instead of expressions e. Similarly, to bind the free variables of the control e, we add an environment E to the machine state, which becomes a four-tuple (Γ, e, E, S). Now an expression e in the control or the heap may have free program variables, but these will be bound in the environment E associated with e.

The resulting abstract machine is shown in Figure 5. Each of the revised app_1 and var_1 rules performs two tasks: first it finds the heap pointer p bound to x in E, and then it behaves as the old app_1 or var_1 rule.

The correctness of this modification is clear, since the computation sequences of the mark 1 and mark 2 machines are closely related. Every state of a mark 2 computation can be mapped to a corresponding state of the mark 1 machine, by replacing every closure (e, E) with the expression eE.

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

	Heap	Control	Environment	Stack	rule
	Γ	$(e\,x)$	$E[x{\mapsto}p]$	S	app_1
$\Longrightarrow$	Γ	e	$E[x{\mapsto}p]$	$p : S$	
	Γ	$\lambda y.e$	E	$p : S$	app_2
$\Longrightarrow$	Γ	e	$E[y{\mapsto}p]$	S	
	$\Gamma[p{\mapsto}(e', E')]$	x	$E[x{\mapsto}p]$	S	var_1
$\Longrightarrow$	Γ	e'	E'	$\#p : S$	
	Γ	$\lambda y.e$	E	$\#p : S$	var_2
$\Longrightarrow$	$\Gamma[p{\mapsto}(\lambda y.e, E)]$	$\lambda y.e$	E	S	
	Γ	$let\{x_i{=}e_i\}e$	E	S	let $(*)$
$\Longrightarrow$	$\Gamma[p_i{\mapsto}(e_i, E')]$	e	E'	S	

$(*)$ In the *let* rule, the variables $p_1, \ldots, p_n$ must be distinct and fresh: they must not occur in Γ or $let\{x_i{=}e_i\}e$ or S. The new environment E' is $E[x_1{\mapsto}p_1, \ldots, x_n{\mapsto}p_n]$.

Fig. 5. Abstract machine mark 2, with stack and environment.

Thus the introduction of environments does not change the *result* of a computation. However, it increases the set of heap addresses p transitively reachable from the machine components e, E, and S. Namely, in the mark 1 rule app_2, the substitution $e[p/y]$ would embed p in the new control only if y occurs free in e. In the mark 2 rule app_2, p will be retained in the new environment $E[y \mapsto p]$ regardless whether y occurs free in e or not. This can make a considerable difference, since many heap addresses may be transitively reachable from the closure pointed to by p in the heap Γ. In terms of lazy language implementations, we have introduced a *space leak*. The same problem appears in the mark 2 treatment of let-bindings. We shall return to this issue in section 4.

3.4 Third step: introduce variable indices

In this step, we get rid of program variable names, replacing them with de Bruijn indices, which are numbers similar to variable offsets in conventional compiler terminology[†]. The syntax of normalized lambda expressions with de Bruijn indices is:

$$e \quad ::= \quad \lambda e \;\mid\; e\,u \;\mid\; u \;\mid\; let\ e_1, \ldots, e_n\ in\ e$$

where u is a *de Bruijn index* (a positive integer). An environment E is now a mapping from positive integers to heap pointers, and can be represented as a list $[p_1, \ldots, p_k]$ of heap pointers, such that E maps index u to $E[u] = p_u$ when $1 \leq u \leq k$. More efficient representations of E exist.

[†] The introduction of de Bruijn indices at this point may seem premature, but it permits comparison with the Krivine machine, and it permits the derivation of a well-known and simple representation of data constructors in section 5.5, which in turn leads to a well-known implementation of base values in section 6.2.

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

	Heap	Control	Environment	Stack	rule
	Γ	$(e\,u)$	$E=[p_1,\ldots,p_k]$	S	app_1
$\Longrightarrow$	Γ	e	E	$p_u : S$	
	Γ	λe	E	$p : S$	app_2
$\Longrightarrow$	Γ	e	$p : E$	S	
	$\Gamma[p_u \mapsto (e', E')]$	u	$E=[p_1,\ldots,p_k]$	S	var_1
$\Longrightarrow$	Γ	e'	E'	$\#p_u : S$	
	Γ	λe	E	$\#p : S$	var_2
$\Longrightarrow$	$\Gamma[p \mapsto (\lambda e, E)]$	λe	E	S	
	Γ	$let\ \{e_i\}\ e$	E	S	let (*)
$\Longrightarrow$	$\Gamma[p_i \mapsto (e_i, E')]$	e	E'	S	

In the app_1 and var_1 rules, p_u is the u'th element of the environment $E = [p_1, \ldots, p_k]$.
(*) In the *let* rule, the addresses $p_1, \ldots, p_n$ must be fresh: they must not occur in Γ or S. The new environment E' is $p_1 : \ldots : p_n : E$.

Fig. 6. Abstract machine mark 3, with stack, environment, and de Bruijn indices.

The transition rules of the final abstract machine are shown in Figure 6. Rules app_2 and *let* ensure that the environment $E = [p_1, \ldots, p_k]$ contains bindings for all variables in scope, with p_1 most recently bound and p_k least recently bound. Thus an occurrence of variable x must be replaced by a de Bruijn index equal to the number of lambdas between the binding lambda $\lambda x \ldots$ and the occurrence, plus one; this is standard (Barendregt, 1984, Appendix C).

3.5 Related abstract machines

The lazified Krivine machines of Crégut (1991, p. 30) and Sestoft (1991, p. 31) may be obtained from the present one by combining the rules *let* and app_1 into a single rule which requires every application $(e_1\,e_2)$ to take the restricted form $let\ x = e_2\ in\ (e_1\,x)$. That is, those machines always allocate a closure for the argument to ensure sharing. When evaluating an application $(e_1\,y)$ in which the argument expression is already a variable y, this creates a superfluous indirection and requires an extra update. In a recursive function, a chain of indirections may build up, causing a space leak.

Hence one advantage of the present machine is that sharing (the *let* rule) has been separated from application (the app_1 rule). We introduce let-bindings only where necessary, by transforming the expressions to normalized form. In addition, *let* can define recursive functions and cyclic data; the other lazified Krivine machines need a separate mechanism for that.

To study our machine's relation to the Three Instruction Machine TIM (Fairbairn and Wray, 1987), we first observe that normalized lambda expressions are closely related to ordinary sequential code:

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

$(e\,u)$	reads	`Push` u`;` e	Push pointer onto stack
λe	reads	`Take;` e	Take one argument from stack
u	reads	`Enter` u	Enter closure
$let\ \{e_i\}\ e$	reads	`Let` $\{e_i\}$`;` e	Make recursive bindings

Inspection of the rules in Figure 6 shows that each of the four instructions `Push`, `Take`, `Enter`, and `Let` performs a bounded amount of computation. The three first instructions are those of the TIM, except that TIM's `Take` n instruction can take $n \geq 1$ arguments from the stack, rather than a single one. Lazy versions of the TIM usually tie sharing to argument passing, as do the lazified Krivine machines, and with the same consequences: chains of indirections will build up unless special precautions are made. Again, using let-bindings to separate sharing from argument passing is preferable.

The TIM's `Take` n instruction is more efficient than our single-argument `Take` instruction in the call-by-name case, but causes complications in the call-by-need case, when there may be update markers in between the n arguments to be taken off the stack.

The abstract machine in Figure 6 is a *Four Instruction Machine*. Experimental implementations of the machine are easily written in Scheme or Standard ML, which provide destructive update as well as an underlying garbage collector.

4 Space considerations

The above abstract machine is quite efficient, but uses too much memory; it has a *space leak*. The environments hold on to useless closures, as hinted in section 3.3. Here we solve this problem.

4.1 A leaky program

To see that the machine as presented so far leaks space, consider the following example program, where I abbreviates $\lambda y.y$:

$$\begin{array}{lcl} let\ f & = & (\lambda n.\ let\ x = I\ in\ (f\ x)) \\ in\ (f\ f) & & \end{array}$$

Evaluation of this expression will not terminate. However, the real problem is that evaluation requires an unbounded amount of space. We shall evaluate the expression using the mark 2 machine in Figure 5; using the mark 3 machine with de Bruijn indices would obscure the presentation.

Evaluation of the outer *let* allocates a closure $((\lambda n.\ let\ x = I\ in\ (f\ x)), \{f \mapsto p_f\})$ for f in the heap, at address p_f say. Evaluation of the call $(f\ f)$ in the let-body binds n to the pointer p_f, and enters the body of f with environment $\{n \mapsto p_f, f \mapsto p_f\}$.

The first evaluation of the inner *let* allocates a closure (I, E_1) for x, at address $p_{x,1}$ say. The environment part $E_1 = \{x \mapsto p_{x,1}, n \mapsto p_f, f \mapsto p_f\}$ contains bindings for x,

n, and f, none of which is needed for evaluating I. The recursive call $(f\ x)$ binds n to the pointer $p_{x,1}$ and enters the body of f with environment $\{n \mapsto p_{x,1}, f \mapsto p_f\}$.

The second evaluation of the inner *let* allocates a new closure (I, E_2) for x, at address $p_{x,2}$ say, where the environment part is $E_2 = \{x \mapsto p_{x,2}, n \mapsto p_{x,1}, f \mapsto p_f\}$. The recursive call $(f\ x)$ binds n to $p_{x,2}$ and enters the body of f with environment $\{n \mapsto p_{x,2}, f \mapsto p_f\}$.

The third evaluation of the inner *let* allocates a new closure (I, E_3) for x, at address $p_{x,3}$ say, where $E_3 = \{x \mapsto p_{x,3}, n \mapsto p_{x,2}, f \mapsto p_f\}$. Then it evaluates the recursive call, and so on.

The undesirable overall effect is to build up an ever-growing chain of closures for x, in which $p_{x,n}$ points to a closure containing $p_{x,n-1}$, which points to a closure containing $p_{x,n-2}$, and so on, down to $p_{x,1}$. At any point of execution, all these closures are reachable from the abstract machine's environment or stack, so none of them can be garbage-collected. Consequently, the space consumption grows linearly with the execution time of the program.

4.2 Practical consequences

The above example is contrived, but similar space leaks would appear in most recursive programs. To show that the problem is significant in practice, we implemented an extended version of the leaky abstract machine and ran it using Standard ML of New Jersey. The data given below are based on the heap sizes reported by that implementation. Printing a prefix of the list of natural numbers, defined by

$$let\ nats\ =\ cons\ 0\ (map\ add1\ nats)\ in\ nats$$

requires heap space proportional to the length of the prefix printed. In the improved non-leaky machine presented below, an arbitrary prefix can be printed in a bounded (and small) amount of space. In a similar experiment, computing and printing the first 200 prime numbers (using Eratosthenes's sieve) requires more than 3000 KB heap space with the leaky machine, and less than 8 KB with the non-leaky one. It is important to solve this problem.

4.3 Environment trimming

The solution is to refine the use of environments in the mark 2 and 3 machines, so they model more closely the substitutions performed in the mark 1 machine (Figure 3). Ideally, the environment E in the machine, and the environments stored in closures in the heap, should not bind any superfluous variables.

Following an idea from the Spineless Tagless G-machine (Peyton Jones, 1992, Section 5.3), we shall *trim* the environments E of let-bound expressions and of let-bodies: only their free variables should be included in E. To perform trimming, we annotate every let-bound expression and the let-body with the set t of its free variables. This annotation is called a *trimmer*.

In the mark 2 machine, an environment E is a mapping from variable names to pointers, a trimmer t is just a set of names, and the trimming $E|t$ of E with respect

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

	Heap	Control	Environment	Stack	rule
	Γ	$let\{x_i = (e_i, t_i)\}(e, t)$	E	S	let' (*)
$\Longrightarrow$	$\Gamma[p_i \mapsto (e_i, E' \mid t_i)]$	e	$E' \mid t$	S	

(*) The variables $p_1, \ldots, p_n$ must be fresh: they must not occur in Γ or $let \ldots$ or S. The new environment E' (before trimming) is $E[x_1 \mapsto p_1, \ldots, x_n \mapsto p_n]$.

Fig. 7. Revised let-rule with environment trimming for abstract machine mark 2.

to t is simply the restriction of the mapping E to the domain t. Figure 7 shows the revised mark 2 *let* rule with trimming. Adding mark 2 trimmers to the example from the previous section, we find that since I has no free variables, its trimmer should be the empty set $\{\}$:

$$\begin{aligned} let\ f \;\; &= \;\; ((\lambda n.\ let\ x = (I, \{\})\ in\ ((f\ x), \{x, f\})), \{f\}) \\ in\ &((f\ f), \{f\}) \end{aligned}$$

We see that the environment in closures allocated for x will be empty, and no space leak will occur. Although many closures are allocated for x during execution, they quickly become unreachable and therefore subject to garbage collection.

In the mark 3 machine, an environment $E = [p_1, \ldots, p_k]$ is a list of pointers, a trimmer $t = [i_1, \ldots, i_n]$ is a sub-sequence of the de Bruijn indices $1, \ldots, k$, and the trimming $E|t$ of E with t is the sub-list $[p_{i_1}, \ldots, p_{i_n}]$. The de Bruijn indices of variable occurrences must be adjusted when the environment gets trimmed. The adjustment depends on the trimmer, that is, the set of free variables of the expression being compiled. In general two passes over the expression are required at compile time, but otherwise this adjustment poses no problems.

Closures for let-bound expressions are written to the heap, potentially have a long life-time, and therefore are prone to cause space leaks; this is the reason for explicitly trimming their environments.

Similarly, when a lambda abstraction is the whnf of a let-bound expression, then a closure for the lambda abstraction will be written to the heap. If the lambda abstraction ignores its argument as in $let\ y = (\lambda y.\lambda z.z)\ x\ in\ \ldots$, the inclusion of x in the environment may cause a space leak. However, a lambda abstraction must occur either in the right-hand side of a let-binding or in a let-body, both of which have their environments trimmed, so although its environment may contain superfluous variables, they are likely to be few.

Trimming the environment of a lambda abstraction would entail the run-time adjustment of the de Bruijn indexes of variable occurrences in its body.

4.4 Related approaches

In the intermediate language of the Spineless Tagless G-machine STG, every let-bound expression is annotated with the set of its free variables. When making a closure for a let-bound expression, the STG let-instruction trims the environment.

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

It does not trim the environment of the let-body, but this does not harm lambda abstractions, which in the STG-machine can appear only as right-hand sides in let-bindings. The STG-machine seems to suffer the same potential space leak as our machine when a lambda abstraction ignores its argument, but in the STG-machine this can be fixed easily.

Lambda lifting (Johnsson, 1985) as used in the G-machine automatically performs environment trimming by turning local function bindings into top-level function bindings, passing any free variables as arguments. This may explain why lambda lifting works so well in lazy languages, in spite of its destroying scope information and introducing more parameter passing: the pointer copying required for parameter passing is required anyway for environment trimming in a lazy language. The G-machine does not seem to trim the environment of let-bound expressions.

Trimming incurs a run-time overhead, but it also reduces the time to access variables when environments are represented as linked lists. A trimmed environment could be represented by a vector, or by a linked list of argument values (and *case*-bound values, see below) followed by a vector of the values of the free variables. For simplicity we use linked lists throughout this paper. After introducing constructors and case-expressions we shall return to environment trimming again.

A lazy approach to environment trimming was proposed by the designers of the Three Instruction Machine (Fairbairn and Wray, 1987, p. 42). Every heap-allocated environment is equipped with a trimmer in the form of a bit vector, indicating which entries in the environment are live. This bit vector is generated a compile time. At garbage collection, only the live pointers in the environment are followed in the copy- or mark-phase. This scheme probably spends less time trimming, but requires more space. It has the advantage that variable indexes need not be adjusted.

Efficient environment representations which are safe for space complexity have been studied for strict functional languages (Shao and Appel, 1994). This work should be relevant when designing environment representations for lazy languages also, although their space/efficiency trade-offs are different: closures are created at a much higher rate, and the lifetime of values is harder to predict.

5 Algebraic datatypes

Machine instructions for handling algebraic datatypes can be derived from natural semantics rules also. Algebraic datatypes, as known from Standard ML or Haskell, introduce two new forms of expressions: constructor applications $c\,x_1 \ldots x_a$ and *case* expressions:

$$e \quad ::= \quad \ldots \quad | \quad c\,x_1 \ldots x_a \quad | \quad case\ e\ of\ \{c_j\,y_{j1} \ldots y_{ja_j} \rightarrow e_j\}_{j=1}^{n}$$

Constructor arguments must be variables to ensure sharing of components; additional let-bindings may be introduced to satisfy this requirement. We assume a typed language, so that constructors from different datatypes cannot be confused at run-time, and so that every constructor c_j has a fixed arity $a_j \geq 0$. Constructors must be fully applied. In a *case* expression, e is the *case object*, and $c_j\,y_{j1} \ldots y_{ja_j} \rightarrow e_j$ is an *alternative*. There must be exactly one alternative for each constructor of the

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

$$\Gamma : c\,p_1 \ldots p_a \Downarrow_A \Gamma : c\,p_1 \ldots p_a \qquad \textit{Cons}$$

$$\frac{\Gamma : e \Downarrow_A \Delta : c_k\,p_1 \ldots p_{a_k} \qquad \Delta : e_k[p_1/y_{k1}, \ldots, p_{a_k}/y_{ka_k}] \Downarrow_A \Theta : w}{\Gamma : \mathit{case}\ e\ \mathit{of}\ \{c_j\,\bar{y}_j \to e_j\} \Downarrow_A \Theta : w} \qquad \textit{Case}$$

Fig. 8. Natural semantics rules for constructors.

datatype. For brevity we write $\{c_j\,\bar{y}_j \to e_j\}$ for the list of alternatives, where $\bar{y}_j$ is a vector $y_{j1} \ldots y_{ja_j}$ of variables.

Launchbury gives the natural rules for evaluation of constructor application and case (see Figure 8, where we have added the A component as in Figure 2). It follows from the rules that a reduced value w may now be a constructor application $c\,p_1 \ldots p_a$ as well as a lambda abstraction $\lambda y.e$.

5.1 Constructors in the mark 1 machine

What changes are needed to handle constructors in the mark 1 abstract machine, shown in Figure 3? Recall that the machine stack represents the continuation of a computation; it says how to join subcomputations, corresponding to the flattening of the derivation tree.

- The new *Cons* rule is similar to the *Lam* rule, and gives rise to no new machine rules: a constructor application is a value already, and requires no machine action.
- The new *Case* rule is similar to the *App* rule, and gives rise to two new machine rules: case_1 which begins the computation corresponding to the left subtree, and case_2 which begins the computation corresponding to the right subtree.
- A new version var_3 of the var_2 rule is needed, since the result w in the *Var* rule of Figure 2 may now be a constructor application $c\,p_1 \ldots p_a$, not just a lambda abstraction $\lambda y.e$. When a constructor application $c\,p_1 \ldots p_a$ finds an update marker $\#p$ on the stack, it must update the heap at p with itself.

The new rules are shown in Figure 9. A new kind of stack object, similar in purpose to an argument p, is needed to store the case alternatives while evaluating the case object:

- a *case marker* is a list $\{c_j\,\bar{y}_j \to e_j\}$ of alternatives.

Rule case_1 pushes the case marker, then starts evaluating the case object e. When an application $c_k\,p_1 \ldots p_{a_k}$ of constructor c_k encounters a case marker, rule case_2 starts evaluating the k'th alternative.

5.2 Correctness of the constructor rules

The proof of Lemma 2 is easily extended to account for the new rules:

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

	Heap	Control	Stack	rule
	Γ	$case\ e\ of\ \ alts$	S	$case_1$
$\Longrightarrow$	Γ	e	$alts : S$	
	Γ	$c_k\, p_1 \dots p_{a_k}$	$alts : S$	$case_2$
$\Longrightarrow$	Γ	$e_k[p_i/y_{ki}]$	S	
	Γ	$c_k\, p_1 \dots p_{a_k}$	$\#p : S$	var_3
$\Longrightarrow$	$\Gamma[p \mapsto c_k\, p_1 \dots p_{a_k}]$	$c_k\, p_1 \dots p_{a_k}$	S	

The notation *alts* stands for the list $\{c_j\, \bar{y}_j \rightarrow e_j\}$ of alternatives, and $\bar{y}_k$ stands for the vector $y_{k1}, \dots, y_{ka_k}$ of variables. In the $case_2$ rule, e_k is the right-hand side of the k'th alternative, and $e_k[p_i/y_{ki}]$ abbreviates $e_k[p_1/y_{k1}, \dots, p_{a_k}/y_{ka_k}]$.

Fig. 9. Constructor rules for abstract machine mark 1.

Case *Cons* is trivial, by the empty sequence of computation steps (as for *Lam*).

Case *Case*: Assume $\Gamma : case\ e\ of\ \{c_j\, \bar{y}_j \rightarrow e_j\} \Downarrow_A \Theta : w$ by rule *Case*, then

$$\begin{array}{lll}
 & (\Gamma, case\ e\ of\ \{c_j\, \bar{y}_j \rightarrow e_j\}, S) & \\
\Rightarrow & (\Gamma, e, \{c_j\, \bar{y}_j \rightarrow e_j\} : S) & \text{by rule } case_1 \\
\Rightarrow^* & (\Delta, c_k\, p_1 \dots p_{a_k}, \{c_j\, \bar{y}_j \rightarrow e_j\} : S) & \text{left premise and ind. hyp.} \\
\Rightarrow & (\Delta, e_k[p_i/y_{ki}], S) & \text{by rule } case_2 \\
\Rightarrow^* & (\Theta, w, S) & \text{right premise and ind. hyp.}
\end{array}$$

□

The other direction is more work. First, there are more balanced traces. Arguing as in Section 3.2, we find that all balanced traces of the new machine can be derived from the following grammar:

$$\begin{array}{lll}
bal & ::= & \epsilon \mid app_1\ bal\ app_2\ bal \mid var_1\ bal\ var_2 \mid let\ bal \mid \\
 & & var_1\ bal\ var_3 \mid case_1\ bal\ case_2\ bal
\end{array}$$

The ϵ trace now corresponds to the *Cons* rule as well as the *Lam* rule, and the two last possibilities correspond to the *Var* rule and the *Case* rule, respectively.

Secondly, Lemma 3 must be amended slightly, because the result w of a computation may now be a constructor application $c_k\, p_1 \dots p_{a_k}$:

Lemma 4

For all Γ_0, e_0, S, Γ_1, and w it holds that if $(\Gamma_0, e_0, S) \Rightarrow^* (\Gamma_1, w, S)$ is balanced, and $w \equiv \lambda v.e_1$ or $w \equiv c_k\, p_1 \dots p_{a_k}$, and $A = \#(S)$ then $\Gamma_0 : e_0 \Downarrow_A \Gamma_1 : w$ is derivable.

Proof

By induction on the structure of balanced traces. Case ϵ. Follows by rule *Lam* or *Cons* according as $w \equiv \lambda v.e_1$ or $w \equiv c_k\, p_1 \dots p_{a_k}$.

Case $app_1\ bal\ app_2\ bal$, case $var_1\ bal\ var_2$, and case $let\ bal$ are as in Lemma 3.

	Heap	Control	Environment	Stack	rule
	Γ	*case e of alts*	E	S	$case_1$
$\Longrightarrow$	Γ	e	E	$(alts, E) : S$	
	Γ	$c_k\, x_1 \dots x_{a_k}$	E'	$(alts, E) : S$	$case_2$
$\Longrightarrow$	Γ	e_k	$E[y_{ki} \mapsto p_i]$	S	
	Γ	$c_k\, x_1 \dots x_{a_k}$	E'	$\#p : S$	var_3
$\Longrightarrow$	$\Gamma[p \mapsto (c_k\, \bar{x}, E')]$	$c_k\, \bar{x}$	E'	S	

The notation *alts* stands for the list $\{c_j\, \bar{y}_j \rightarrow e_j\}$ of alternatives, and $\bar{y}_k$ stands for the vector $y_{k1}, \dots, y_{ka_k}$ of variables. In the $case_2$ rule, e_k is the right-hand side of the k'th alternative, and $p_i = E'[x_i]$ for $i = 1, \dots, a_k$. In the var_3 rule, $\bar{x}$ stands for the vector $x_1 \dots x_{a_k}$ of variables.

Fig. 10. Constructor rules for abstract machine mark 2.

Case var_1 *bal* var_3. We must have $e_0 \equiv p$ and $\Gamma_0 = \Gamma[p \mapsto e]$ and $\Gamma_1 = \Delta[p \mapsto c_k\, p_1 \dots p_{a_k}]$ for some p, Γ, e, and Δ. The state after var_1 is $(\Gamma, e, \#p : S)$ and the state before var_3 is $(\Delta, c_k\, p_1 \dots p_{a_k}, \#p : S)$. Since the subtrace between these is balanced, $\Gamma : e \Downarrow_{A \cup \{p\}} \Delta : c_k\, p_1 \dots p_{a_k}$ is derivable by the induction hypothesis, which is applicable because $A \cup \{p\} = \#(\#p : S)$. Using the *Var* rule, we find that $\Gamma[p \mapsto e] : p \Downarrow_A \Delta[p \mapsto c_k\, p_1 \dots p_{a_k}] : c_k\, p_1 \dots p_{a_k}$ is derivable.

Case $case_1$ *bal* $case_2$ *bal*: We must have $e_0 \equiv$ *case e of* $\{c_j\, \bar{y}_j \rightarrow e_j\}$. The state after $case_1$ must be $(\Gamma_0, e, \{c_j\, \bar{y}_j \rightarrow e_j\} : S)$ and the state before $case_2$ must be $(\Delta, c_k\, p_1 \dots p_{a_k}, \{c_j\, \bar{y}_j \rightarrow e_j\} : S)$. The trace between these is balanced, so that $\Gamma_0 : e \Downarrow_A \Delta : c_k\, p_1 \dots p_{a_k}$ is derivable by the induction hypothesis. The state after $case_2$ is $(\Delta, e_k[p_i/y_{ki}], S)$, and the trace of $(\Delta, e_k[p_i/y_{ki}], S) \Rightarrow^* (\Gamma_1, w, S)$ is balanced, so $\Delta : e_k[p_i/y_{ki}] \Downarrow_A \Gamma_1 : w$ is derivable by the induction hypothesis. Using the *Case* rule, we find that Γ_0 : *case e of* $\{c_j\, \bar{y}_j \rightarrow e_j\} \Downarrow_A \Gamma_1 : w$ is derivable. □

Proposition 2 must be amended similarly and its proof must appeal to the extended lemmas.

5.3 Constructors in the mark 2 machine

We introduce environments to avoid doing substitutions in the code, as in section 3.3 when we derived the mark 2 machine. Because of possible free variables in the right-hand sides of alternatives, the $case_1$ rule must store the environment E along with the alternatives in the case marker (on the stack), and rule $case_2$ must extract it again. Hence a case marker now has form $(\{c_j\, \bar{y}_j \rightarrow e_j\}, E)$. Also, the substitution in rule $case_2$ must be replaced by environment extension. The new rules are shown in Figure 10.

5.4 Constructors in the mark 3 machine

Replacing variable names by de Bruijn indices as in section 3.4, the syntax for an alternative $c_j\, \bar{y}_j \rightarrow e_j$ becomes $c_j \rightarrow e_j$, simply. The pattern variables $y_{j1} \dots y_{ja_j}$ have

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

	Heap	Control	Environment	Stack	rule
	Γ	$case\ e\ of\ alts$	E	S	$case_1$
$\Longrightarrow$	Γ	e	E	$(alts, E) : S$	
	Γ	$c_k\, u_1 \dots u_{a_k}$	E'	$(alts, E) : S$	$case_2$
$\Longrightarrow$	Γ	e_k	E''	S	
	Γ	$c_k\, u_1 \dots u_{a_k}$	E'	$\#p : S$	var_3
$\Longrightarrow$	$\Gamma[p \mapsto (c_k\, \bar{u}, E')]$	$c_k\, \bar{u}$	E'	S	

The notation *alts* stands for the list $\{c_j \to e_j\}$ of alternatives. In the $case_2$ rule, e_k is the right-hand side of the k'th alternative, and $E'' = E'[u_{a_k}] : \dots : E'[u_1] : E$ is the environment for e_k. In the var_3 rule, $\bar{u}$ stands for the vector $u_1 \dots u_{a_k}$ of de Bruijn indices.

Fig. 11. Naïve constructor rules for abstract machine mark 3.

been replaced by the indices $a_j, \dots, 1$ in the right-hand side e_j. The seemingly inverse order of the indices is natural; it corresponds to the indices in e_j of the parameters in the lambda abstraction $\lambda y_{j1}. \dots \lambda y_{ja_j}.e_j$. The resulting abstract machine is shown in Figure 11.

5.5 Improving constructors in the mark 3 machine

The handling of constructors and *case* expressions can still be improved, by requiring that every constructor application has the restricted form

$$c_k\ a_k\ \dots\ 1$$

in which the arguments $u_1 \dots u_{a_k}$ are exactly the indices of the a_k most recently bound variables. In that case the new environment E'' in the second rule of Figure 11 is always $E'' = E'[1] : \dots : E'[a_k] : E$, which is just the environment E appended to the first a_k variables from E'. Then there is no need to represent the arguments explicitly in a constructor application; an application of a constructor c_k can be represented just by $c_{k,a}$, showing its tag k and arity a. We have arrived at the `Pack`$\{tag\texttt{,}arity\}$ representation for constructors used by Peyton Jones and Lester (1992), with $tag = k$ and $arity = a$.

The restriction on constructor application can be enforced by defining named constructors, as illustrated by the standard list-constructors `nil` and `cons` (with arities 0 and 2):

$$\begin{array}{lcl} \texttt{nil} & = & c_{1,0} \\ \texttt{cons} & = & \lambda\lambda c_{2,2} \end{array}$$

An application `cons` $x\ y$ of a named constructor is an ordinary application, in which the arguments x and y must be variables to ensure sharing. To illustrate the use of this encoding, consider evaluating `cons` $x\ y$ with an update marker $\#p$ on the stack top; assume that $p_x = E[x]$ and $p_y = E[y]$:

	Heap	Control	Environment	Stack	rule
	Γ	$case\ e\ of\ alts$	E	S	$case_1$
$\Longrightarrow$	Γ	e	E	$(alts, E) : S$	
	Γ	$c_{k,a}$	E'	$(alts, E) : S$	$case_2$
$\Longrightarrow$	Γ	e_k	E''	S	
	Γ	$c_{k,a}$	E'	$\#p : S$	var_3
$\Longrightarrow$	$\Gamma[p \mapsto (c_{k,a}, E'[1 \ldots a])]$	$c_{k,a}$	E'	S	

The notation *alts* stands for the list $\{c_j \rightarrow e_j\}$ of alternatives. In the $case_2$ rule, e_k is the right-hand side of the k'th alternative, and $E'' = E'[1] : E'[2] : \ldots : E'[a] : E$. In the var_3 rule, $E'[1 \ldots a]$ abbreviates $[E'[1], \ldots, E'[a]]$.

Fig. 12. Improved constructor rules for abstract machine mark 3.

	Γ	$(\lambda\lambda c_{2,2})\, x\, y$	E	$\#p : S$	
$\Rightarrow$	Γ	$(\lambda\lambda c_{2,2})\, x$	E	$p_y : \#p : S$	(app_1)
$\Rightarrow$	Γ	$(\lambda\lambda c_{2,2})$	E	$p_x : p_y : \#p : S$	(app_1)
$\Rightarrow$	Γ	$\lambda c_{2,2}$	$p_x : E$	$p_y : \#p : S$	(app_2)
$\Rightarrow$	Γ	$c_{2,2}$	$p_y : p_x : E$	$\#p : S$	(app_2)
$\Rightarrow$	$\Gamma[p \mapsto (c_{2,2}, [p_y, p_x])]$	$c_{2,2}$	$p_y : p_x : E$	S	(var_3)

Moving cons's arguments into the environment via the stack may seem cumbersome, but this scheme is simple and permits partial application of constructor encodings such as cons. Moreover, in a realistic implementation, the frequent case of applying a known constructor to all of its arguments can be detected and optimized at compile-time. The improved constructor rules are shown in Figure 12.

Note that we ever only use the first a_k elements of the constructor application's environment, where a_k is the constructor's arity. Hence we need to store only the first a_k elements of E in the heap when meeting an update marker. This is particularly efficient if an environment is a linked list of vectors (of heap pointers); then the constructor arguments can be represented by a fixed size vector of heap pointers, and the append operation to build E'' in rule $case_2$ of Figure 12 can be done in constant time, independently of the size of E' and E.

5.6 Environment trimming for constructors

The space behaviour of the rules in Figure 12 need to be improved in two ways. In rule $case_1$, storing the entire environment E in the case marker on the stack may cause a space leak. Namely, it may hold on to a large value while the case object e is being evaluated. Hence we should trim the environment E before storing it on the stack.

In rule $case_2$, even if an alternative has no free variables at all, its environment E'' will contain all the variables free in other alternatives. If the alternative is a lambda

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

	Heap	Control	Environment	Stack	rule
	Γ	$case\ e\ of\ alts, t$	E	S	$case_1$
$\Longrightarrow$	Γ	e	E	$(alts, E\mid t):S$	
	Γ	$c_{k,a}$	E'	$(alts, E):S$	$case_2$
$\Longrightarrow$	Γ	e_k	$E''\mid t_k$	S	
	Γ	$c_{k,a}$	E'	$\#p : S$	var_3
$\Longrightarrow$	$\Gamma[p\mapsto(c_{k,a}, E'[1\dots a])]$	$c_{k,a}$	E'	S	

The notation *alts* stands for the list $\{c_j \rightarrow (e_j, t_j)\}$ of alternatives with trimmers. In the $case_2$ rule, (e_k, t_k) is the right-hand side of the k'th alternative, and $E'' = E'[1] : \dots E'[a] : E$. In the var_3 rule, $E'[1\dots a]$ abbreviates $[E'[1], \dots, E'[a]]$.

Fig. 13. Final constructor rules with environment trimming, abstract machine mark 3.

expression $\lambda z.z$, a closure for it might be written to the heap, causing a space leak because of these free variables. Hence we should trim the environment E'' of an alternative, just as we trim the environment of a let-body.

In rule var_3 we can do no better than including all the arguments when writing a constructed value $(c_{k,a}, E'[1\dots a])$ to the heap.

We therefore add a separate trimmer t_k for each alternative $c_k \rightarrow e_k$, and a joint trimmer t for the list *alts* of all alternatives. The trimmer t_k is the set of variables free in e_k, including the variables bound on the left-hand side, and t is the set of variables free in one or more of $\{c_j \rightarrow e_j\}$, excluding the variables bound on the left-hand sides. We arrive at the improved rules shown in Figure 13.

The Spineless Tagless G-machine trims the joint environment of the alternatives of *case* (Peyton Jones, 1992, Section 9.4.1). The G-machine does not seem to.

6 Base values

Additional machine instructions for handling base values, such as integers, can be introduced in several ways. In this section we derive mechanisms for base value handling from the representations suggested in Peyton Jones and Launchbury (1991). Alternatively, one may derive nearly the same base value operations from Launchbury's operational semantics (Sestoft, 1994, Appendix B).

6.1 Deriving base value handling

We use integers with addition to illustrate base values:

$$e \quad ::= \quad \dots \quad \mid \quad n \quad \mid \quad e_1 + e_2$$

Note that the operands of the addition need not be variables, as they cannot be shared. We shall assume that expressions are well-typed, so '+' is applied only to expressions that evaluate to integers. Launchbury's evaluation rules are shown in Figure 14. An integer n reduces to itself immediately by rule *Int*. An addition is

$$\Gamma : n \Downarrow_A \Gamma : n \qquad \textit{Int}$$

$$\frac{\Gamma : e_1 \Downarrow_A \Delta : n_1 \qquad \Delta : e_2 \Downarrow_A \Theta : n_2}{\Gamma : e_1 + e_2 \Downarrow_A \Theta : n_1 + n_2} \qquad \textit{Add}$$

Fig. 14. Natural semantics rules for integers.

evaluated by evaluating the operands in turn and adding their results by rule *Add*. This reflects that '+' and other base value operators are *strict*: the arguments must be evaluated before the operation can be applied.

We exploit two ideas due to Peyton Jones and Launchbury (1991): using constructors to represent boxed integers, and using *case* expressions to evaluate arguments before the application of '+'. Recall that a *boxed* base value v is represented by a closure which must be evaluated to obtain v, whereas an *unboxed* value is just the bit-pattern representing the value. A boxed value may turn out to be undefined, since an attempt to evaluate the closure may loop, whereas an unboxed value is always defined.

First, we represent a boxed integer by a constructor application $\textit{Int}\,n$. Here $\textit{Int}$ is the only constructor in the datatype of boxed integers and has arity one, corresponding to the generic constructor $c_{1,1}$. Using the final constructor representation developed in section 5.5, a boxed integer $\textit{Int}\,n$ would be represented by a pair $(\textit{Int}, E)$ where E is a one-element vector containing a pointer to a closure somehow representing the unboxed integer n. But the unboxed n is just a bit-pattern, so we shall let E be a one-element vector containing n directly, giving the boxed representation $(\textit{Int}, [n])$.

Secondly, we use *case* expressions to force argument evaluation. For instance, the addition $e_1 + e_2$ of two integer-valued expressions is defined following Peyton Jones and Launchbury (1991, p. 644), here slightly simplified:

$$\begin{array}{lll} \textit{case}\ e_1\ \textit{of} & & \\ \textit{Int}\,n_1 & \rightarrow & \textit{case}\ e_2\ \textit{of} \\ & & \textit{Int}\,n_2 \quad \rightarrow \quad \textit{Int}\,(n_1 + n_2) \end{array}$$

Using de Bruijn indices as in section 5.5, the *case* expression translates to the following one (where we ignore environment trimming):

$$\textit{case}\ e_1\ \textit{of}\ \{\textit{case}\ e_2\ \textit{of}\ \{\textit{Int}\ (2+1)\}\}$$

Here 2 and 1 are the indices of n_1 and n_2, not base value constants, so $(2+1)$ means 'add the two first values in the environment'. The rules already given in Figure 12 say how to evaluate the *case* expression, except for the unboxed addition $(2+1)$. It suffices to postulate a new machine instruction $op+$, which computes the (unboxed) sum n' of the two first values in the current environment, and creates the boxed result $(\textit{Int}, [n'])$. Thus we replace the innermost case alternative $\textit{Int}\ (2+1)$ by $op+$.

In conclusion, a single new instruction is needed to perform addition. Another

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

	Heap	Control	Environment	Stack	rule
	Γ	*Cst n*	E	S	*cst*
$\Longrightarrow$	Γ	*Int*	$[n]$	S	
	Γ	*op+*	$n_2 : n_1 : E$	S	*add*
$\Longrightarrow$	Γ	*Int*	$[n_1 + n_2]$	S	

Fig. 15. Rules for base values in abstract machine mark 3.

new instruction *Cst n* is needed to introduce integer constants; it must evaluate to $(Int, [n])$. The required new rules are shown in Figure 15.

For illustration, consider the evaluation of $e_1 + e_2$, which is compiled as

$$case\ e_1\ of\ \{case\ e_2\ of\ \{op{+}\}\}$$

Using the abbreviation $e' \equiv case\ e_2\ of\ \{op{+}\}$ for the inner *case*, we have

$$\begin{array}{lllrrl}
 & \Gamma & case\ e_1\ of\ \{e'\} & E & S & \\
\Rightarrow & \Gamma & e_1 & E & (\{e'\}, E) : S & (case_1) \\
\Rightarrow^* & \Gamma & Int & [n_1] & (\{e'\}, E) : S & (\text{evaluate } e_1) \\
\Rightarrow & \Gamma & case\ e_2\ of\ op{+} & n_1 : E & S & (case_2) \\
\Rightarrow & \Gamma & e_2 & n_1 : E & (\{op{+}\}, n_1 : E) : S & (case_1) \\
\Rightarrow^* & \Gamma & Int & [n_2] & (\{op{+}\}, n_1 : E) : S & (\text{evaluate } e_2) \\
\Rightarrow & \Gamma & op{+} & n_2 : n_1 : E & S & (case_2) \\
\Rightarrow & \Gamma & Int & [n_1 + n_2] & S & (add)
\end{array}$$

There is a close correspondence between this computation sequence and a tree derived from the *Add* rule in the natural semantics of Figure 14. The two subcomputations starting with $case_1$ and ending just before $case_2$ correspond to the left and right subtree, respectively. The first occurrence of $case_2$ initiates the evaluation of the right-hand subtree, and the second occurrence of $case_2$ activates the *add* step, which performs the addition in the conclusion of the rule.

When a let-bound expression evaluates to an integer, the result $(Int, [n])$ will encounter an update marker $\#p$ on the machine stack. Since *Int* is just the constructor $c_{1,1}$, this situation is handled by rule var_3 in Figure 12, which will update the heap at p with the closure $(Int, [n])$, as desired.

6.2 Introducing a base value stack

The base value handling developed above is simple but inefficient. Intermediate base values are held in the environment component of closures on the stack during the computation. This may prevent optimization of environment representations, and may complicate garbage collection by mixing base values and pointers in the stack. It is better to have a separate value stack V for intermediate base values.

To introduce a separate value stack V, we replace the general *Int* constructor (which is really $c_{1,1}$) by a special constructor called *Ret*, which will be used only to

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

	Heap	Control	Env	Values	Stack	rule
	Γ	$Cst\,n$	E	V	S	cst
$\Longrightarrow$	Γ	Ret	n	V	S	
	Γ	$op{+}$	E	$n_2 : n_1 : V$	S	add
$\Longrightarrow$	Γ	Ret	$n_1 + n_2$	V	S	
	Γ	Ret	n	V	$(\{e\}, E) : S$	ret_1
$\Longrightarrow$	Γ	e	E	$n : V$	S	
	Γ	Ret	n	V	$\#p : S$	ret_2
$\Longrightarrow$	$\Gamma[p \mapsto (Ret, n)]$	Ret	n	V	S	

Fig. 16. Rules for base values in abstract machine mark 3, with value stack.

handle boxed base values. A boxed value will be a closure (Ret, n), and we still use the environment component to hold the unboxed value, since this simplifies updating of the heap. Only the new Ret and $op{+}$ instructions will access the unboxed value, so we can use the representation (Ret, n) instead of $(Ret, [n])$.

The new instruction Ret must push the unboxed integer n from the environment onto the value stack. In contrast to Int it should not extend the environment, so it must behave as an argument-less constructor $c_{1,0}$ rather than $c_{1,1}$. The behavior of $c_{1,0}$ is prescribed by the $case_2$ and var_3 rules of Figure 12. By rule $case_2$, when Ret encounters a case marker $(\{e\}, E)$ on the stack, it must activate (e, E), and push the base value held in E onto the value stack V. By rule var_3, when Ret encounters an update marker $\#p$ on the stack, it must update the heap at p with (Ret, n).

The rules for $Cst\,n$ and $op{+}$ must be modified too. Clearly $Cst\,n$ must evaluate to (Ret, n). The addition operator $op{+}$ will find its arguments n_2 and n_1 on the value stack V, and must evaluate to $(Ret, n_1 + n_2)$. The new rules are shown in Figure 16.

The action performed by Ret is to 'return' to the 'address' kept on the stack top: it activates the single closure in the case marker on the stack, hence its name. Originally, Ret was introduced as a code trick by Fairbairn and Wray (1987) to handle base values in the Three Instruction Machine; they called it $Self$ rather than Ret. It was adopted also by Peyton Jones and Lester (1992), who called it $Return$. What is new here is the derivation of Ret from an ordinary constructor $c_{1,1}$.

Consider again the evaluation of $e_1 + e_2$ by the new rules. The expression is still compiled as $case\ e_1\ of\ \{e'\}$, where e' abbreviates $case\ e_2\ of\ \{op{+}\}$:

	Γ	$case\ e_1\ of\ \{e'\}$	E	V	S
$\Rightarrow$	Γ	e_1	E	V	$(\{e'\}, E) : S$
$\Rightarrow^*$	Γ	Ret	n_1	V	$(\{e'\}, E) : S$
$\Rightarrow$	Γ	$case\ e_2\ of\ \{op{+}\}$	E	$n_1 : V$	S
$\Rightarrow$	Γ	e_2	E	$n_1 : V$	$(\{op{+}\}, E) : S$
$\Rightarrow^*$	Γ	Ret	n_2	$n_1 : V$	$(\{op{+}\}, E) : S$
$\Rightarrow$	Γ	$op{+}$	E	$n_2 : n_1 : V$	S
$\Rightarrow$	Γ	Ret	$n_1 + n_2$	V	S

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

Type correctness would ensure that the only stack top objects encountered by the *Ret* instruction are update markers $\#p$ or case markers $(\{e\}, E)$; never an argument pointer p.

Consider introducing the abbreviation e *seq* e' for *case* e *of* $\{c_{1,0} \to e'\}$, where *seq* associates to the right. The compilation of $e_1 + e_2$ is e_1 *seq* e_2 *seq* $op{+}$, the reverse Polish form of the expression. The net effect of evaluating a base value expression, such as e_1, is to push its own reduced value onto the value stack.

6.3 Boolean values, conditionals, and printing

Boolean values may be represented by two argument-less constructors $\mathit{False} \equiv c_{1,0}$ and $\mathit{True} \equiv c_{2,0}$, which may be tested by *case* expressions. These constructors must be generated by the basic comparison instructions such as $op{<}$, $op{\leq}$, $op{=}$, $op{\wedge}$, $op{\vee}$ which operate on the value stack. Alternatively, Boolean values may be represented by the integers 0 and 1, and one may introduce a new conditional such as $\mathit{Cond}(e_1, e_2)$ from Peyton Jones and Lester (1992, Section 4.3.2) to test them, but this would require a duplication of the machinery for environment trimming.

Printing of data structures can be done systematically as well – see Peyton Jones and Lester (1992, Section 4.6.4) and Sestoft (1991, Section 3.4). One new machine instruction suffices. It must print a string (as a side effect), and then activate the closure in the case marker on the stack S. The activation of this closure corresponds to the demand for a new substructure to print; thus the print instruction forces the evaluation of composite data structures, as usual in lazy language implementations. For this reason printing is not a trivial concern: If one wants to argue the correctness of e.g. an evaluation order analysis with respect to the abstract machine, then one must know how printing drives evaluation.

7 Assessment and further improvements

The final abstract machine, consisting of Figures 6, 7, 13 and 16, plus operations for Booleans and printing (not shown), has twelve instructions and implements all dynamic aspects of a lazy functional language. We wrote a straightforward experimental implementation in Standard ML for the example language used by Peyton Jones and Lester (1992). We used Standard ML references to implement the heap Γ, and rely on the Standard ML system for garbage collection. Computing and printing the first 300 prime numbers using Eratosthenes's sieve takes approximately 21 seconds in this implementation running under Standard ML of New Jersey version 108.13 and Linux with an Intel 486DX4/100MHz processor. This is only 5.3 times slower than Mark Jones's Gofer system (version 2.30b, written in C).

Let us list some positive and negative properties of our abstract machine:

+ it performs tail calls in constant space;
+ it does not build indirection chains;
− it performs many identical updates (but this is easily avoided);
− all values must test the stack top;

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

+ it can exploit strictness information to eliminate some identical updates (by not pushing update markers for values which are already evaluated);
– even with strictness information it is hard to eliminate the tests on the stack top element;
– variable access is not constant time, because the linked environment structure must be traversed.

Surprisingly, the overhead incurred by identical updates in the experimental implementation is negligible. In a more realistic implementation, eliminating identical updates and some stack top tests would probably give a significant speed-up.

The environment trimming performed in let- and case-expressions may appear expensive, but this cost is present also in the G-machine and the Three Instruction Machine TIM (when applying super-combinators) and in the Spineless Tagless G-machine (where environment trimming is explicit in let-bindings and case-expressions), and is necessary to avoid space leaks.

In our experimental implementation the environment is a linked list of pointers. The other extreme in the design space is the TIM's representation of the environment as a vector of pointers. Our representation gives fast application and slow variable access, and TIM's representation gives slow application and fast variable access. A linked list of vectors is an obvious compromise, which should work well with trimming.

Compile-time optimizations may be used to improve the implementation of known, fully applied constructors. Similarly, one may improve base value handling as in the $\mathscr{B}$ compilation scheme of Peyton Jones and Lester (1992, p. 160). By maintaining a symbolic value stack at compile-time one may detect when a constant or an $op+$ instruction will necessarily find another arithmetic operation in the case marker on the stack top (at run-time). In such cases the *Ret* step may be skipped, and the base value operations can be performed directly on the value stack.

In short, we believe that a realistic implementation, perhaps the Spineless Tagless G-machine, could be derived by further refinements.

8 Related work

The literature on derivation of abstract machines is rich. Closest to the present work is Hannan and Miller's formal derivation of the call-by-name Krivine machine from a natural semantics for call-by-name evaluation, and Hannan's (1991) further concretizations of abstract machines. Their derivation differs from ours in that environments and de Bruijn indices are introduced already in the natural semantics. Hannan and Miller (1990) present a general transformation from branching natural semantics rules to non-branching ones, essentially by introducing an explicit stack of premises still to be proved. The further transformation from non-branching natural semantics rules to flat abstract machine rules proceeds by several steps, more *ad hoc*. Altogether they achieve, in smaller steps, the same as our somewhat monolithical correctness proof (Proposition 2). On the other hand, our notion of balanced traces

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

has intuitive significance and provided for an easy extension of the correctness proof when augmenting the language with data structures (section 5).

Cast in a logical framework, Hannan and Miller's derivation lends itself well to formalization; Hannan and Pfenning (1992) mechanically verified such a translation proof in Elf. We have not attempted machine verification, but believe that the supporting concepts are formalized well enough that it would be feasible.

Wand (1992a, b) derived a compiler and an abstract machine from a denotational semantics in continuation passing style. Josephs (1989) gave a denotational semantics for a lazy functional languages. Presumably Wand's approach could be used to derive a lazy abstract machine from that.

Fairbairn and Wray (1987) developed the Three Instruction Machine TIM for call-by-name evaluation of super-combinators in 1986, and made it lazy by using update markers $\#p$ on the stack. Argo studied inefficiencies and improvements of the TIM, in particular the handling of shared partial applications and the representation of environments (Argo, 1989, 1990).

Krivine developed his machine for call-by-name evaluation around 1985; a description is given by Curien (1988). Borrowing the update marker technique from the TIM, several people have made the Krivine machine lazy, including Crégut (1990, 1991) and Sestoft (1991). The close relation between the Krivine machine and the TIM has been studied by Crégut (1991, p. 41) and Mogensen (1992). Although simple, the Krivine machine is practically useful: a strict version is at the core of the efficient interpretive Caml Light system due to Leroy (1990).

Developing the Spineless Tagless G-machine, Peyton Jones (1992) studies all aspects of lazy language evaluation. The point of departure is graph reduction, and a solid operational understanding of lazy languages, rather than a formal description. The necessary intuition can be gained by studying Peyton Jones (1987) and Peyton Jones and Lester (1992).

9 Conclusion

We developed a simple and well-known abstract machine from a natural semantics for lazy evaluation. We proceeded in a number of refinement steps, proving the correctness of the non-trivial steps, and demonstrated that well-known implementation techniques can be derived from a semantics.

The resulting abstract machine is less sophisticated than the G-machine (Augustsson, 1984; Johnsson, 1984), the Spineless Tagless G-machine (Peyton Jones, 1992), and the refined Three Instruction Machine (Argo, 1990; Peyton Jones and Lester, 1992), but it is simple and its correctness requires no separate proof. This shows that a lazy functional language implementation can be at the same time demonstrably correct, understandable, small, and quite efficient.

In addition to having pedagogical value, the present abstract machine and development provide a foundation for program analyses which are concerned with the operational properties of lazy languages.

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

Acknowledgements

A visit to Andy Moran and John Hughes at Chalmers University of Technology inspired this work. The anonymous referees provided much helpful advice on the presentation and the subject matter. Thanks to Carsten Kehler Holst, Torben Mogensen and Simon Peyton Jones for comments and suggestions.

References

Argo, G. (1989) Improving the Three Instruction Machine. *Fourth International Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, pp. 100–112. Addison-Wesley.

Argo, G. (1990) *Efficient laziness.* PhD thesis, Department of Computing Science, University of Glasgow. Draft Manuscript.

Augustsson, L. (1984) A compiler for Lazy ML. *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, pp. 218–227. ACM.

Barendregt, H. P. (1984) *The Lambda Calculus. Its Syntax and Semantics.* (Revised edn.) North-Holland.

Crégut, P. (1990) An abstract machine for the normalization of λ-terms. *ACM Conference on Lisp and Functional Programming*, Nice, France, pp. 333–340. ACM.

Crégut, P. (1991) *Machines à environnement pour la réduction symbolique et l'évaluation partielle.* PhD thesis, Université Paris VII, France.

Curien, P. L. (1988) The $\lambda\rho$-calculus: an abstract framework for environment machines. *Rapport de Recherche LIENS-88-10*, Ecole Normale Supérieure, Paris, France.

Fairbairn, J. and Wray, S. C. (1987) TIM: A simple, lazy abstract machine to execute supercombinators. In: Kahn, G. (ed), *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 274*, Portland, Oregon, pp. 34–45. Springer-Verlag.

Hannan, J. (1991) Making abstract machines less abstract. In: Hughes, J. (ed), *Functional Programming Languages and Computer Architecture: 5th ACM Conference: Lecture Notes in Computer Science 523*, Cambridge, MA, pp. 618–635. Springer-Verlag.

Hannan, J. and Miller, D. (1990) From operational semantics to abstract machines: Preliminary results. *ACM Conference on Lisp and Functional Programming*, Nice, France, pp. 323–332. ACM.

Hannan, J. and Pfenning, F. (1992) Compiler verification in LF. In: Scedrov, A. (ed), *Seventh Annual IEEE Symposium on Logic in Computer Science*, pp. 407–418. IEEE Press.

Johnsson, T. (1984) Efficient compilation of lazy evaluation. *ACM Sigplan Symposium on Compiler Construction*, pp. 58–69. (*SIGPLAN Notices* **19**(6)). ACM.

Johnsson, T. (1985) Lambda lifting: Transforming programs to recursive equations. In: Jouannaud, J.-P. (ed), *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 201*, Nancy, France, pp.190–205. Springer-Verlag.

Josephs, M. B. (1989) The semantics of lazy functional languages. *Theoretical Computer Science* **68**: 105–111.

Launchbury, J. (1993) A natural semantics for lazy evaluation. *Twentieth ACM Symposium on Principles of Programming Languages*, Charleston, SC, pp. 144–154. ACM.

Leroy, X. (1990) The Zinc experiment: An economical implementation of the ML language. *Rapport Technique 117*, INRIA Rocquencourt, France.

Mogensen, T. (1992) Re: Is lambda lifting always necessary? Personal communication.

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall.

Peyton Jones, S. L. (1992). Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *J. Functional Programming* **2**(2): 127–202.

Peyton Jones, S. L. and Launchbury, J. (1991) Unboxed values as first class citizens in a non-strict functional language. In: Hughes, J. (ed), *Functional Programming Languages and Computer Architecture, 5th ACM Conference: Lecture Notes in Computer Science 523*, Cambridge, MA, pp. 636–666. Springer-Verlag.

Peyton Jones, S. L. and Lester, D. (1992) *Implementing Functional Languages*. Prentice-Hall.

Sansom, P. M. and Peyton Jones, S. L. (1995) Time and space profiling for non-strict, higher-order functional languages. *POPL'95: 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, CA, pp. 355–366. ACM.

Sestoft, P. (1991) *Analysis and efficient implementation of functional programs*. PhD thesis, DIKU, University of Copenhagen, Denmark. (DIKU Research Report 92/6.)

Sestoft, P. (1994) Deriving a lazy abstract machine. *Technical Report ID-TR 1994-146*, Department of Computer Science, Technical University of Denmark.

Shao, Z. and Appel, A. (1994) Space-efficient closure representations. *ACM Conference on Lisp and Functional Programming*, Orlando, FL, pp. 150–161.

Wand, M. (1982a) Deriving target code as a representation of continuation semantics. *ACM Trans. Programming Languages and Systems* **4**(3): 496–517.

Wand, M. (1982b) Semantics-directed machine architecture. *Ninth ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, pp. 234–241. ACM.

https://doi.org/10.1017/S0956796897002712 Published online by Cambridge University Press