The Spineless Tagless G-machine, naturally

Jon Mountjoy

Department of Computer Science
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam
The Netherlands
email: ~~jon@wins.uva.nl~~
research@jonmountjoy.com

Abstract

The application of natural semantic specifications of lazy evaluation to areas such as usage analysis, formal profiling and abstract machine construction has shown it to be a useful formalism. This paper introduces several variants and extensions of this specification.

The first variant is derived from observations of the Spineless Tagless G-machine (STG), used in the Glasgow Haskell compiler. We present a modified natural semantic specification which can be formally manipulated to derive an STG-like machine.

The second variant is the development of a natural semantic specification which allows functions to be applied to more than one argument at once. The STG and TIM abstract machines both allow this kind of behaviour, and we illustrate a use of this semantics by again modifying this semantics following observations of the STG machine. The resulting semantics can be used to formally derive the STG machine. This effectively proves the STG machine correct with respect to Launchbury's semantics.

En route, we also show that update markers in the STG machine are necessary for termination, and show how well-known abstract machine instructions, such as the *squeeze* operation, appear quite naturally as optimisations of the derived abstract machine.

1 Introduction

The STG machine has proved itself as being capable of efficiently executing lazy functional languages. This machine lies at the heart of the Glasgow Haskell Compiler (Peyton Jones 1996), which compiles by translating a program written in Haskell through intermediate, simpler, languages until it finally generates programs in the STG language. This final language is then compiled into code which mimics the operational semantics given to the language in the form of an abstract machine. However, it is yet to be shown that the STG abstract machine is correct. Intuitively the machine does what we expect it to do, being a refined model of a graph reducer which, operationally, seems sound.

The first natural semantic specification of lazy evaluation for lazy functional languages was presented in Launchbury (1993), which was proved correct with respect to a call-by-name denotational semantics of the same language. We regard this as *the* specification of what is meant by a lazy functional language. Sestoft (1997) has taken the natural semantic specification and derived from it various abstract machines, proving that these abstract machines are correct with respect to each other and the natural semantic specification. It is not quite clear, however, how these abstract machines can be related to the STG machine.

This paper introduces several variants of natural semantics specifications. The first variant is derived from observations of the STG, which seems to treat variables pointing to lambda abstractions as values, as opposed to just lambda abstractions. A modified natural semantic specification is suggested which does just this, and we show how it can be formally manipulated to derive an STG-like abstract machine.

The STG machine, however, just like the TIM abstract machine, is capable of handling multiple arguments in its applications and abstractions. Unfortunately, Launchbury's semantics handles only single arguments. Our second variation extends Launchbury's semantics to multiple arguments. The STG machine can then be formally derived from this semantics, yielding a proof that the STG machine is correct.

En route, we also show that update markers in the STG machine are necessary for termination, and show how well-known abstract machine instructions, such as the *squeeze* operation, appear quite naturally as optimisations of the derived abstract machine.

The structure of this paper is as follows:

- The following section recalls the natural semantics of lazy evaluation which we will be using as a reference semantics.
- Section 3 makes observations about the STG which suggest changes to the natural semantic specification. A new semantics is then developed and shown correct with respect to the reference semantics.
- Section 4 illustrates how an STG-like abstract machine can be derived from the semantics.
- Section 5 extends the reference semantics to handle multiple arguments during abstraction and application.
- Section 6 parallels sections 3 and 4 and derives a semantics for the STG machine based on multiple arguments, and subsequently the STG machine.

$$\Gamma : \lambda x.e \ \Downarrow \ \Gamma : \lambda x.e \qquad Lam$$

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

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

$$\frac{\Gamma[x_i \mapsto e_i] : e \ \Downarrow \ \Delta : w}{\Gamma : \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \ \Downarrow \ \Delta : w} \qquad Let$$

$$\Gamma : c \ x_1 \ldots x_j \ \Downarrow \ \Gamma : c \ x_1 \ldots x_j \qquad Cons$$

$$\frac{\Gamma : e \ \Downarrow \ \Delta : c_k \ x_1 \ldots x_{a_k} \quad \Delta : e_k[x_1/y_{k1}, \ldots, x_{a_k}/y_{ka_k}] \ \Downarrow \ \Theta : w}{\Gamma : \text{case } e \text{ of } \{c_j \ \vec{y_j} \rightarrow e_j\}_{j=1}^n \ \Downarrow \ \Theta : w} \qquad Case$$

Figure 1: Natural Semantic specification of Lazy Evaluation

Unfortunately the STG machine is quite large and complex, and to describe it here would require too much space. We thus assume some knowledge of the STG machine, as presented in (Peyton Jones 1992). Proofs of all of the propositions will be made available in a technical report.

2 The Natural Semantics of Lazy Evaluation

We begin by (briefly) reviewing the language and semantics based on Launchbury (1993).

The abstract syntax of the language, which we christen the *basic* language, is given by:

expressions:	e	::=	$\lambda x.e$
		|	let $\{x_i = e_i\}_{i=1}^n$ in e
		|	$e \ x$
		|	x
		|	$c \ x_1 \ldots x_j$
		|	case e of $\{c_j \ \vec{y_j} \rightarrow e_j\}_{j=1}^n$

This syntax guarantees that expressions are in a so-called *normalised* form. The process of normalisation ensures that:

- application of a function can only be made to a single variable
- all constructors (c) have only variables as arguments
- all constructors are saturated (fully applied)
- all bound variables are *distinct*

The first three properties are ensured by the syntax. The last is implemented by an α-conversion which renames all bound variables using completely fresh variables, which we write as $\hat{e}$. Launchbury (1993) provides a simple scheme for normalising arbitrary expressions into the basic language. Intuitively, the restriction of application and constructor arguments to variables makes the sharing of these arguments explicit. Note that Launchbury only allows application and abstraction of a single argument. A natural semantic specification of this language is shown in Figure 1.

In the above, a heap, $\Gamma = [\ldots, x \mapsto e, \ldots]$ represents a mapping from variables x to expressions e. We write $\Gamma[x_i \mapsto e_i]$ for $\Gamma[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$, $dom \ \Gamma$ for the domain of Γ, and $rng \ \Gamma$ for the range (the set of expressions bound in the heap) of Γ. The list of alternatives of a case statement is written as $alts$(below) or $\{c_j \ \vec{y_j} \rightarrow e_j\}_{j=1}^n$, where $\vec{y_j}$ represents the vector $y_{j_1} \ldots y_{j_{a_j}}$ of variables, a_j being the arity of the constructor c_j. A *configuration* $\Gamma : e$ is an expression and a heap in which the expression is to be evaluated. A *judgement* $\Gamma : e \ \Downarrow \ \Delta : w$ says that expression e in heap Γ evaluates to a value w and heap Δ.

In Figure 1 we see that rules *Lam* and *Cons* say that the value of the lambda abstraction or constructor is itself – indeed, these are what lazy functional languages consider to be values!

Rule *Let* evaluates a let expression by evaluating the qualified expression e in a heap in which all of the bindings are present. Note that this is the only rule which adds bindings to the heap, and that the bindings themselves are not evaluated.

Rule *App* says that to evaluate an application $e \ x$, we need evaluate e to a lambda abstraction $\lambda y.e'$, and evaluate the e' with x substituted for y in the resulting heap to produce a final value and heap.

An integral part of lazy evaluation is that when something is evaluated, the result is rebound in the heap so that further requests for the value yield the already evaluated value. Rule *Var* captures precisely this. This rule says that to evaluate a variable bound to an expression e in the heap, we can evaluate this expression in a heap without this binding to produce a value. This value can then be *rebound* to the variable in the final heap – thus ensuring that further demands of the variable will yield the new value. This is the only place where a heap *update* occurs. The STG language of section 3 provides support for sometimes avoiding this expensive operation, if it will lead to no change in the final value produced. Since we duplicate value w, this may cause name clashes (recall that otherwise all bound variables are distinct) and we have to rename all of the bound variables of w to fresh variables, indicated by $\hat{w}$. See Launchbury

$$\Gamma[x \mapsto \lambda y.e] : x \;\downarrow\; \Gamma[x \mapsto \lambda y.e] : x \qquad Lam_S$$

$$\frac{\Gamma : f \;\downarrow\; \Theta[v \mapsto \lambda y.e] : v \quad \Theta[v \mapsto \lambda y.e] : e[x/y] \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma : f\ x \;\downarrow\; \Delta[u \mapsto w] : u} \qquad App_S$$

$$\frac{\Gamma : e \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma[x \mapsto e] : x \;\downarrow\; \Delta[u \mapsto w, x \mapsto \hat{w}] : x} \; w \text{ a lambda abstraction, } e \text{ not} \qquad Var_S$$

$$\frac{\Gamma : e \;\downarrow\; \Delta : c\ y_1 \ldots y_j}{\Gamma[x \mapsto e] : x \;\downarrow\; \Delta[x \mapsto c\ y_1 \ldots y_j] : c\ y_1 \ldots y_j} \; e \text{ not a lambda abstraction} \qquad VarC_S$$

$$\frac{\Gamma[x_i \mapsto e_i] : e \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma : \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \;\downarrow\; \Delta[u \mapsto w] : u} \qquad Let_S$$

$$\Gamma : c\ x_1 \ldots x_j \;\downarrow\; \Gamma : c\ x_1 \ldots x_j \qquad Cons_S$$

$$\frac{\Gamma : e \;\downarrow\; \Delta : c_k\ x_1 \ldots x_{a_k} \quad \Delta : e_k[x_1/y_{k1}, \ldots, x_{a_k}/y_{ka_k}] \;\downarrow\; \Theta[u \mapsto w] : u}{\Gamma : \text{case } e \text{ of } \{c_j\ \vec{y_j} \to e_j\}_{j=1}^n \;\downarrow\; \Theta[u \mapsto w] : u} \qquad Case_S$$

Figure 2: Natural Semantic specification of Lazy Evaluation for the λ-normalised language

(1993) and Sestoft (1997) for more on this operation and its relation to name clashes.
Rule *Case* evaluates an expression to a constructor, and then selects the appropriate expression from the list of alternatives, substituting the arguments of the constructor for the formal variables in the alternative.

Launchbury proves that the natural semantic specification is sound and complete with respect to a denotational semantics of the language:

Proposition 1 (Denotational Correctness for $\Downarrow$)
If $\Gamma : e \;\Downarrow\; \Delta : z$ *then for all environments* ρ *we have:* $[\![e]\!]\rho\Gamma = [\![z]\!]\rho\Delta$

Sestoft (1997) has shown how an abstract machines can be formally derived from this natural semantics specification. He thus derives an abstract machine $\mathcal{A}$ which simulates $\Downarrow$ in the sense that the abstraction machine will yield a value for an expression iff the natural semantics proves that the value of the expression e is w. Writing $\xrightarrow{\mathcal{A}}$ for a step in the abstract machine $\mathcal{A}$, and $\xrightarrow{\mathcal{A}}{}^*$ for its transitive closure, we have:

Proposition 2 (Correctness of Sestoft's Machine)
If w *is a value then* $e \;\Downarrow\; w$ *iff* $e \xrightarrow{\mathcal{A}}{}^* w$.

If we can use these techniques to formally derive the STG machine, then we will have provided a proof of correctness of the STG machine, and related it to Sestoft's machine. Unfortunately, it is not quite clear how this should be done. The following two sections show a way – we first highlight characteristic features of the STG machine and modify our semantics to incorporate them, and we then extend the entire semantic framework to handle multiple arguments. As a result, we produce a semantics from which the STG machine can be derived.

3 The STG Machine

The syntax of the STG machine base language, as presented in Peyton Jones (1992), is quite different from that of the basic language considered in the previous section. These syntactic differences contribute significantly to the differences in the derived abstract machines. This section highlights the key difference, that of the restricted locations of lambda abstractions, and introduces a new natural semantic specification embodying this restriction.

To focus on the key difference, and in keeping with the previous section, we use a restricted STG language where application and abstraction are only allowed on one argument. We call this language the STG^S language, and its syntax is:

expressions:	e	::=	let $\{x_i = lf_i\}_{i=1}^n$ in e	
		|	$x\ y$	
		|	x	
		|	c *vars*	
		|	case e of *alts*	
lambda-forms:	*lf*	::=	$vars_f \setminus \pi\ x_a.e$	
variables:	*vars*	::=	$\{x_1, \ldots, x_n\}$	$(n \geq 0)$
alternatives:	*alts*	::=	$\{c_j\ \vec{y_j} \to e_j\}_{j=1}^n$	

We have introduced two new syntactic categories, one for convenience (*vars*, representing a possibly empty sequence of variables), and one which is peculiar to the STG machine, a lambda-form (*lf*). A lambda-form replaces the notion of a lambda abstraction, and as can be seen from the syntax, restricts the position of lambda abstractions. We write $\{\}$ if x_a is empty. In a lambda-form, if x_a is non-empty, then the lambda-form represents a lambda abstraction, x_a being the variable which is abstracted. $vars_f$ represents the free variables of the abstraction. Either x_a or $vars_f$ (or both) may be absent. These free variables play no rôle in the denota-

tional semantics of the language, and are only of operational significance. In addition, a lambda-form has an update flag, π, which plays a rôle when we consider the language operationally. The flag may be set to either u indicating that the bound expression will be updatable, or n indicating that it will not be updatable. We will discover that this is *not* just an optimisation but also necessary for termination, as is discussed in section 4.1.

We may also have a lambda-form which has no argument and only free variables. This corresponds to an ordinary let bound expression (albeit that it cannot be a lambda abstraction) of the basic language. Again, we assume that constructors are saturated.

The most notable differences enforced by the syntax of this language, compared to the basic language, are that:

- The function body of an application must itself must be a simple variable, and not an arbitrary expression.
- Lambda expressions cannot occur in arbitrary places. Instead, lambda expression can only occur let-bound.

Consequences of the restricted syntax

The basic language allows arbitrary expressions in the body of an application:

$$e ::= e\ x \mid \ldots$$

whereas the STG^S language only allows applications of variables:

$$e ::= f\ y \mid \ldots$$

where f and y are variables. A function body other than a variable is not allowed, and so has to be heap allocated by a let binding. This restriction goes hand in hand with the allowed location of lambda abstractions. Since lambda abstractions can only be let bound, and let bindings are heap allocated, this implies that all of the function bodies will be heap allocated. Recall that lambda abstractions are also values – the previous observation implies that if a variable is to be evaluated to a lambda abstraction (signifying a value), then it will instead be evaluated to a variable pointing to the lambda abstraction. That is, we 'detect a value via a variable'. An important intuition is that if an expression is to be evaluated to a lambda abstraction, then it can be evaluated to a variable pointing to the lambda abstraction *since all the lambda abstractions have to be let-bound* and thus will appear in the heap. This hints at the graph reduction ancestry of the STG machine.

We postulate then:

> The STG machine considers a *variable bound to a lambda abstraction* as a value, as opposed to the notion of a *lambda abstraction* being a value. If this concept is incorporated into the natural semantics, an STG machine can be derived.

The rest of this section shows that this is indeed true.

The above notion 'detects' a value sooner, in the sense that we do not wait until a value is rebound in the heap and start executing the value (as in rule Var), but rather stop when we see that we are pointing to it. Note however, that the STG machine does not employ the same concept in its handling of constructors (see (Peyton Jones 1992) for details), and manipulates constructors just as in our basic language semantics.

Thus, we propose a new semantics which instead of evaluating an expression like this:

$$\dfrac{\dfrac{\dfrac{}{\Gamma : \lambda q.q \ \Downarrow\ \Gamma : \lambda q.q}\ Lam}{\Gamma[x \mapsto \lambda q.q] : x \ \Downarrow\ \Gamma[x \mapsto \lambda q.q] : \lambda z.z}\ Var}{\Gamma : \text{let } x \ = \ \lambda q.q \text{ in } x \ \Downarrow\ \Gamma[x \mapsto \lambda q.q] : \lambda z.z}\ Let$$

evaluates it like this:

$$\dfrac{\dfrac{}{\Gamma[x \mapsto \lambda q.q] : x \ \downarrow\ \Gamma[x \mapsto \lambda q.q] : x}\ Lam_S}{\Gamma : \text{let } x \ = \ \lambda q.q \text{ in } x \ \downarrow\ \Gamma[x \mapsto \lambda q.q] : x}\ Let_S$$

where we use the notation $\downarrow$ to denote the new relation.

That is, the new derivation does not use a Var rule because of the early detection of the lambda abstraction which we have moved to the new Lam rule. We thus propose that we collapse the rules Var and Lam for the case when a variable is pointing to a lambda abstraction, giving us the behaviour illustrated above. Since we handle the two types of values(constructors and abstractions) differently, we propose the introduction of two Var rules – one for each. Figure 2 suggests a new natural semantics for the extended language based on these observations. The rules follow those in Figure 1 closely, except that in many places we halt when we have a variable bound to a value. The two ways of handling values is reflected in rules Var_S and $VarC_S$.

Aside: We should duplicate rules App_S, Let_S and $Case_S$, so that they too may return constructors as results. As it stands, they can only return pointers to abstractions. This duplication just complicates our presentation, and presents no technical difficulties(see the technical report). It also have no impact on the derived abstract machine.

The semantics presented in Figure 2 needs expressions to be in the form described by the syntax of the extended language. We call a language which has these restrictions λ-normalised. We write the λ-normalisation of an expression e as e^*, and it is defined as follows:

$$
\begin{array}{lcl}
x^* & = & x \\
(\lambda x.e)^* & = & \text{let } y \ = \ \lambda x.e^* \text{ in } y \\
(\text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e)^* & = & \text{let } \{x_i = \eta_i\}_{i=1}^n \text{ in } e^* \\
 & & \quad where \begin{cases} \eta_i = \lambda y.e_i'^* & \text{if } e_i \equiv \lambda y.e_i' \\ \eta_i = e_i^* & \text{otherwise} \end{cases} \\
(e\ x)^* & = & \begin{cases} e\ x & e \text{ a variable} \\ \text{let } y \ = \ e^* \text{ in } y\ x & \text{otherwise} \end{cases} \\
(c\ x_1 \ldots x_j)^* & = & c\ x_1 \ldots x_j \\
(\text{case } e \text{ of } \{c_j\ \vec{y_j} \to e_j\}_{j=1}^n)^* & = & \text{case } e^* \text{ of } \{c_j\ \vec{y_j} \to e_j^*\}_{j=1}^n
\end{array}
$$

Note that all introduced variables must be fresh.

Values

Using the natural semantics, we can now state that the meaning of a closed expression e is given by either:

$$\{\} \ : e \ \downarrow\ \Delta[u \mapsto \lambda y.e'] : u$$

or

$$\{\} \ : e \ \downarrow\ \Delta : c\ p_1 \ldots p_j$$

In the first case we can consider the lambda abstraction as the actual result, and in the second the constructor. Indeed, the semantics in Figure 1 will yield one of these (as shown below).

Although the new semantics is less elegant than the old, it does hint of a more operational nature and some conscious design decisions have been made:

- The old semantics cannot 'early-detect' a lambda expressions as a value, as illustrated in the examples above; but this is just a tradeoff: on the one hand the new semantics does not need a 'lambda' instruction, as we will never hit a lambda in the control; on the other, we have introduced an extra check on the value in the heap (is it a lambda abstraction or not).
- The old semantics needs less heap allocation as normalising invariably leads to a greater number of let bindings and thus heap allocations. However, the normalising allows for the early detection of values. A machine capable of application and abstraction of multiple arguments would decrease the heap allocation, and motivates extending our semantics as presented in section 5.
- The new semantics increases the number of times we have to check the type of a closure in the heap. For instance, rule Var_S needs to check that e is not a lambda abstraction.

In the rest of the paper we will assume that the result of evaluating the root expression is a lambda abstraction and not a constructor. All of the propositions below hold in both cases.

Correctness

We can prove that λ-normalisation does not change the meaning of an expression. That is:

Proposition 3 (Correctness of λ-normalisation)

$$\{\} : e \Downarrow \Delta : w \textit{ iff } \{\} : e^* \Downarrow \Theta : w$$

Finally, we can state the correctness lemma. The new semantics is correct in the sense that if the original semantics evaluated an expression to a value, then the new one will evaluate the expression to a variable bound to the same value. For λ-normalised terms then, we prove:

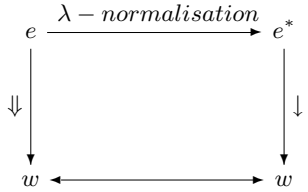
Proposition 4 (Correctness of $\downarrow$)

$\Gamma : m \Downarrow \Delta : w$ *iff* $\Gamma : m \downarrow \Delta'[u \mapsto w] : u$ *where* $\Delta'[u \mapsto w] \stackrel{\alpha}{=} \Delta$

(Note that u may point to an α-renaming of w)

We write $\Delta \stackrel{\alpha}{=} \Lambda$ to indicate that both heaps map the same range to α-equivalent expressions.

Given a closed expression e, proposition 3 tells us that the value of e is unchanged by normalisation. Proposition 4 tells us that for normalised expressions, the value of an expression is the same under both reduction systems (beyond the fact that we result in a pointer to the value instead of the value itself). We have thus shown:



Since propositions 3 and 4 provides a soundness and completeness proof for normalised terms, we have indirectly a proof of correctness with respect to the denotational semantics as well, using proposition 1.

As indicated by proposition 2, Sestoft derives an abstract machine $\mathcal{A}$ which simulates $\Downarrow$. The following section (informally) derives an abstract machine $\mathcal{N}$ which can be proved correct in a similar manner with respect to our natural semantics: That is $e \downarrow w$ iff $e \stackrel{\mathcal{N}}{\rightarrow}{}^* w$.

All together, we then have that if the meaning of a closed expression e is w under $\Downarrow$, then the abstract machine in the following section will derive this value (actually a pointer to it). That is, our basic STG^S machine is correct.

4 Deriving a STG-like machine

A machine can be formally derived from a natural semantic specification by "flattening" the specification: operationally the natural semantics builds a derivation tree relating an expression to its value from the bottom up, whereas an abstract machine computes the value by building a state sequence. The technique used to make an abstract machine is to represent the context of subtrees by a stack.

Consider the *App* rule:

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

This says that to prove that $e\ x$ evaluates to w, we must first find a proof that e evaluates to a lambda abstraction. Then, we must evaluate the expression found by substituting the argument in the abstraction. This final value is the value for the entire application.

Flattening this specification will yield two abstract machine instructions. The first will start the computation of e – and push the argument on the stack to remember to start the computation of the substitution when a lambda abstraction occurs. The second will perform the substitution if a lambda abstraction is found with an argument on the stack.

We now, rather informally, use this technique to derive an abstract machine from the semantic specification in Figure 2. A formal proof of correctness using this technique can be found by following (Sestoft 1997) or (Sansom 1994). The first step to deriving such a machine, however, is the incorporation of a slight change to the semantics. In the current semantics, renaming takes place in the variable rule. A much more natural place for this to take place is in the Let_S rule, allowing the renaming to be mimicked by the allocation of fresh heap locations. This development, and the corresponding proof that the semantics is correct with respect to renaming (that is, that no name clashes occur), can be found in the technical report.

We begin by flattening the natural semantics, as described above. A stack is introduced to represent the flattened tree structure, and we write $\Gamma\ e\ S$ to represent the state of the

	Heap	*Control*	*Environment*	*Stack*	*Rule*
	Γ	(let $\{x_i = e_i\}_{i=1}^n$ in e)	E	S	*let*
$\Longrightarrow$	$\Gamma[p_i \mapsto (e_i, E')]$	e	E'	S	
	$\Gamma[p \mapsto (e', E')]$	x	$E[x \mapsto p]$	S	var_1
$\Longrightarrow$	Γ	e'	E'	$\#p : S$	
	$\Gamma[p \mapsto (\lambda y.e', E')]$	x	$E[x \mapsto p]$	$\#p_u : S$	var_2
$\Longrightarrow$	$\Gamma[p_u \mapsto (\lambda y.e', E')]$	x	$E[x \mapsto p]$	S	
	Γ	$f\ x$	$E[x \mapsto p_x]$	S	app_1
$\Longrightarrow$	Γ	f	$E[x \mapsto p_x]$	$p_x : S$	
	$\Gamma[p_x \mapsto (\lambda y.e, E')]$	x	$E[x \mapsto p_x]$	$p : S$	app_2
$\Longrightarrow$	Γ	e	$E'[y \mapsto p]$	S	
	Γ	case e of *alts*	E	S	$case_1$
$\Longrightarrow$	Γ	e	E	$(alts, E) : S$	
	Γ	$c_k\ x_1 \ldots x_{a_k}$	E'	$(alts, E) : S$	$case_2$
$\Longrightarrow$	Γ	e_k	$E[y_{ki} \mapsto p_i]$	S	
	Γ	$c_k\ x_1 \ldots x_{a_k}$	E'	$\#p : S$	var_3
$\Longrightarrow$	$\Gamma[p \mapsto (c_k\ \vec{x}, E')]$	$c_k\ \vec{x}$	E'	S	

In the *let* rule, the variables p_i that replace x_i must be distinct and fresh and not occur in Γ or S. The new environment $E' = E[x_i \mapsto p_i]$. In the $case_2$ rule, e_k is the right hand side of the k'th alternative, and $p_i = E'[x_i]$ for $i = 1, \ldots, a_k$. See section 4.1 for a corrected var_1 rule.

Figure 3: The New Abstract Machine ($\mathcal{N}$) derived from the new natural semantics.

abstract machine in which we are evaluating expression e in heap Γ with stack S.

The rules Lam_S and $Cons_S$ give rise to no abstract machine instructions. Intuitively, no machine operations are needed to leave the heap and expression unchanged.
The Let_S rule gives rise to one abstract machine instruction which binds the values in the heap and executes the expression e:

$$\Gamma\ (\text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e)\ S \Longrightarrow \Gamma[p_i \mapsto e_i]\ (e)\ S$$

The Var_S rule gives rise to two instructions, one to initiate the evaluation of e (and push a marker x onto the stack) and the other to rebind the resulting value (if we find a marker on the stack):

$$\Gamma[x \mapsto e]\ (x)\ S \Longrightarrow \Gamma\ (e)\ (\#x : S)$$

$$\Gamma[x \mapsto \lambda y.e]\ (x)\ (\#z : S) \Longrightarrow \Gamma[z \mapsto \lambda y.e]\ (z)\ S$$

The $VarC_S$ rule also gives rise to two instructions. The first, to initiate the evaluation of e, is identical to that generated by the Var_S rule. The second tells us to perform an update if we have a marker on the stack and *constructor* in the control:

$$\Gamma\ (c\ x_1 \ldots x_j)\ (\#z : S) \Longrightarrow \Gamma[z \mapsto c\ x_1 \ldots x_j]\ (c\ x_1 \ldots x_j)\ S$$

The App_S rule gives rise to two instructions. The first should evaluate the body of the function (and place the argument on the stack), and the second should perform the substitution (when it finds a lambda abstraction and an argument on the stack).

$$\Gamma\ (f\ x)\ S \Longrightarrow \Gamma\ f\ (x : S)$$

$$\Gamma[v \mapsto \lambda y.e]\ (v)\ (x : S) \Longrightarrow \Gamma\ (e[x/y])\ S$$

The $Case_S$ gives rise to two instructions. The first evaluates the expression to a constructor, and the second chooses the correct alternative:

$$\Gamma\ (\text{case } e \text{ of } \{c_j\ \vec{y_j} \to e_j\})\ S \Longrightarrow \Gamma\ e\ (\{c_j\ \vec{y_j} \to e_j\} : S)$$

$$\Gamma\ (c_k\ x_1 \ldots x_{a_k})\ (\{c_j\ \vec{y_j} \to e_j\} : S) \Longrightarrow \Gamma\ e_k[x_k/y_k]\ S$$

The above machine can be further refined by introducing an environment to model the substitutions taking place. An environment, E, maps variables to heap locations, and can be thought of as a delayed substitution which is only applied when we meet a variable in the control. We thus replace substitutions $e[p/y]$ by a pair $(e, [y \mapsto p])$ representing the fact that within e, y actually points to p. Sestoft (1997) discusses this transformation in more detail. We write $E[x]$ for the value of x under E.

The resulting machine appears in Figure 3.

Even though we are using a restricted STG language, this machine looks much less like the machine described by Sestoft, and very much more like the STG machine. The three rules for the variables (var_1,var_2 and app_2) correspond to the famous STG *Enter* instruction, which enters a closure. Rules *let*, app_1 and $case_1$ correspond to the STG *Eval* instruction and var_2 to the *ReturnCon* rule. The machine described by Sestoft operated on our basic language, which has unrestricted lambda abstractions. As a consequence, his machine has instructions which fire if a lambda abstraction occurs in the instruction stream which make it appear quite different to ours. We have of course, just shown that they essentially do the same thing, though somewhat differently.

4.1 The Neglected Side-Conditions

Recall that the STG language in section 3 annotates lambda-forms with either u or n, indicating whether they are updatable or not. Peyton Jones (1992) states: "It is clearly safe to

set the update flag of every lambda-form to u, thereby updating every closure", and goes on further to explain that an obvious optimisation would be to set the flag to n for 'manifest functions' (which we read here as lambda abstractions), as they are already in head normal form. As support for this hypothesis, the STG machine itself has no rule for an updatable lambda abstraction. This is just as well: if it wasn't for this reasoning and action the STG machine would not work. This optimisation is a prerequisite for correct termination! We argue this point below.

The machine illustrated in figure 3 does not work, as we have forgotten to implement the side conditions of the natural semantics rules Var_S and $VarC_S$. These state that action should only occur if the variable is bound to a non-lambda term. The abstract machine instruction var_1 above does not take this into account, and as such, the machine stops in an erroneous state with a non-empty stack[1]. The corrected abstract machine $\mathcal{N}'$ should only fire rule var_1 if a non-lambda expression is bound. This corresponds to the STG rule described in Peyton Jones (1992) which fires only if the variable is bound to an updatable lambda form and lambda-abstractions are never marked updatable.

We argue that the resulting machine *is* the STG machine, albeit restricted to one argument. Here, we refer to the STG machine as described in (Peyton Jones 1992):

1. An apparent difference is the handling of the environment. All closures need to hold a mapping of variables to heap addresses. In our machine this is represented with the ever present E. In the STG machine, this is represented by the combination of the free variables, vs, and their values, ws_f. Indeed, the STG machine *trims* the environment to contain just those data necessary (the values of the free variables). The trimming is an optimisation, and a similar strategy can be implemented in our machine, as described by Sestoft.

2. The STG machine often looks up the values of the variables in the environment before using them in a later instruction, as opposed to passing the environment through to the later instruction. See for instance STG rules 1 and 2, where p_f is first looked up in the environment. Semantically, rule 1 could have just as well pass the entire environment together with f when executing the *Enter*, and look up f's value in the *Enter* rule. Thus both approaches are equivalent.

3. The STG machine has updatable and non-updatable closures. In our terminology, the updatable closures correspond to expressions which are not lambda-abstractions, and the non-updatable to those which are.

4. The final difference is in the tagging of the code component with *Enter*, *Eval* or *ReturnCon*. This provides no problems, except for the constructor case (we may need to *Eval* a constructor instead of just executing *ReturnCon*). This, together with the premature lookup in the environment explained in (2) above, explains the extra STG rule 5.

We thus have that our rules let, $case_1$, $case_2$ and var_3 correspond with STG rules $3, 4, 6$ and 16, while rules app_1, var_1, var_2 and app_2 correspond with the STG rules $1, 15, 17$ and 2 given in (Peyton Jones 1992). The full STG machine also has rules for handling integers, which explains the missing numbers. Sestoft describes a very similar way of handling numbers for his abstract machine (inspired by the unboxed representations which the STG machine uses), and so this development presents little difficulty and sheds no light on the STG machine itself.

Following Sestoft it can be shown that:

Proposition 5 (Correctness of the $\mathcal{N}'$ Machine) *Assume w is a value. Then:*

$$(\Gamma, e, [], []) \overset{\mathcal{N}'}{\rightarrow}{}^{*} (\Delta[p \mapsto w], p, [], []) \text{ iff } \Gamma : e \ \downarrow \ \Delta[u \mapsto w] : u$$

That is, the single-stack STG machine is correct with respect to the natural semantic specification given in Figure 1.

5 Allowing multiple arguments in abstractions and applications

We now propose an extension of Launchbury's semantics to allow the abstraction and application of multiple arguments. At the moment, if we wish to write the expression $\lambda xy.y$ in our basic language, then we have to write instead the expression let $t = \lambda y.y$ in $\lambda x.t$. As a consequence of this, any abstract machine derived from the semantics will perform a heap allocation (due to the let statement) when executing the above statement. In this section we propose an extension of the semantics given in figure 1 to solve this.

The abstract syntax of the language, which we christen the *extended* language, is given by:

$$\begin{array}{lrcl} \text{expressions:} & e & ::= & \lambda \vec{x}^n.e \\ & & | & e\ \vec{x}^n \\ & & | & x \\ & & | & \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \\ & & | & c\ x_1 \ldots x_j \\ & & | & \text{case } e \text{ of } \{c_j\ \vec{y_j} \rightarrow e_j\}_{j=1}^n \end{array}$$

We use the notation, $\vec{x}^n$, to indicate a vector of n arguments.

A few examples

To guide the development of the new semantics, we look at a few examples. Of course, we only expect the semantics to change for the *Lam* and *App* rules, which use the extension. In the following, we assume that w is some arbitrary value, and ignore the contents of the heap.

Having multiple arguments has as consequence that we can now pass too many or too few arguments to a lambda abstraction. In our first example, we look at an under-applied application (an application which is not given enough arguments), given by expression $e \equiv f\ w$ executed in a heap in which f is bound to the expression $\lambda ab.a$. We would expect e to evaluate to a lambda abstraction expecting one argument as the final value, as f will evaluate to a lambda abstraction of arity two, which, after substituting the only argument, will end up with the required left over abstraction. That is, we want:

$$\frac{f \ \Downarrow \ \lambda ab.a \quad \lambda b.a[w/a] \equiv \lambda b.w \ \Downarrow \ \lambda b.w}{f\ w \ \Downarrow \ \lambda b.w}$$

[1] Consider executing let $x = \lambda z.z$ in x.

$$\Gamma : \lambda\vec{x}^n.e \;\Downarrow\; \Gamma : \lambda\vec{x}^n.e \qquad Lam_E$$

$$\frac{\Gamma : e \;\Downarrow\; \Delta : \lambda\vec{y}^n.\lambda\vec{z}^m.e' \quad \Delta : \lambda\vec{z}^m.e'[\vec{x}^n/\vec{y}^n] \;\Downarrow\; \Theta : w}{\Gamma : e\ \vec{x}^n \;\Downarrow\; \Theta : w}\ m > 0 \qquad App_E$$

$$\frac{\Gamma : e \;\Downarrow\; \Delta : \lambda\vec{y}^m.e' \quad \Delta : e'[\vec{x}^m/\vec{y}^m]\ \vec{x}^{(m+1)\ldots n} \;\Downarrow\; \Theta : w}{\Gamma : e\ \vec{x}^n \;\Downarrow\; \Theta : w}\ n \geq m \qquad App'_E$$

$$\frac{\Gamma : e \;\Downarrow\; \Delta : w}{\Gamma[x \mapsto e] : x \;\Downarrow\; \Delta[x \mapsto w] : \hat{w}} \qquad Var_E$$

$$\frac{\Gamma[x_i \mapsto e_i] : e \;\Downarrow\; \Delta : w}{\Gamma : \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \;\Downarrow\; \Delta : w} \qquad Let_E$$

$$\Gamma : c\ x_1 \ldots x_j \;\Downarrow\; \Gamma : c\ x_1 \ldots x_j \qquad Cons$$

$$\frac{\Gamma : e \;\Downarrow\; \Delta : c_k\ x_1 \ldots x_{a_k} \quad \Delta : e_k[x_1/y_{k1}, \ldots, x_{a_k}/y_{ka_k}] \;\Downarrow\; \Theta : w}{\Gamma : \text{case } e \text{ of } \{c_j\ \vec{y_j} \to e_j\}_{j=1}^n \;\Downarrow\; \Theta : w} \qquad Case_E$$

Note that in rule *App*, e' may actually be a lambda abstraction. We just require that at least n elements can be consumed. Rule *App*$'$ substitutes all available arguments, and applies the resulting expression to the arguments not thus consumed.

Figure 4: Natural Semantic specification of Lazy Evaluation

This suggests the general rule:

$$\frac{\Gamma : e \;\Downarrow\; \Delta : \lambda\vec{y}^n.\lambda\vec{z}^m.e' \quad \Delta : \lambda\vec{z}^m.e'[\vec{x}^n/\vec{y}^n] \;\Downarrow\; \Theta : w}{\Gamma : e\ \vec{x}^n \;\Downarrow\; \Theta : w}\ App_E$$

with $m > 0$.

Now for a case where there are just enough arguments. Executing the expression $f\ w\ w$ with the same heap as before, we would expect w as the resulting value as the *App* rule should substitute both arguments. That is, we want the following behaviour:

$$\frac{f \;\Downarrow\; \lambda ab.a \quad a[w/a, w/b] \equiv w \;\Downarrow\; w}{f\ w\ w \;\Downarrow\; w}$$

What if there are too many arguments? We would expect that the application rule would substitute as many as are needed for the abstraction, and then apply the resulting expression to the rest of the arguments. That is, evaluating $f\ w_1\ w_2\ w_3$ should result in the value of applying w_1 to w_3:

$$\frac{f \;\Downarrow\; \lambda ab.a \quad a[w_1/a, w_2/b]\ w_3 \equiv w_1\ w_3 \;\Downarrow\; w}{f\ w_1\ w_2\ w_3 \;\Downarrow\; w}$$

These suggests a new rule:

$$\frac{\Gamma : e \;\Downarrow\; \Delta : \lambda\vec{y}^m.e' \quad \Delta : e'[\vec{x}^m/\vec{y}^m]\ \vec{x}^{(m+1)\ldots n} \;\Downarrow\; \Theta : w}{\Gamma : e\ \vec{x}^n \;\Downarrow\; \Theta : w}\ App'_E$$

which we want to able to apply when $n \geq m$. In the case where we have just enough arguments, an application will not be formed.

The new, multiple argument, semantics

A natural semantic specification of the extended language is shown in Figure 4. These rules are exactly the same as those in Figure 1 except for the two new application rules discussed above, and the Lam_E rule which says that a lambda abstraction of any number of arguments is still a value.

Following section 3, we can define an *argument*-normalisation which maps the extended language into the basic language:

$$\begin{array}{lcll}
x^{\#} & = & x & \\
(\lambda x_1 \ldots x_n.e)^{\#} & = & \text{let} & a_n = \lambda x_n.e \\
 & & & a_{n-1} = \lambda x_{n-1}.a_n \\
 & & & \vdots \\
 & & & a_1 = \lambda x_1.a_2 \\
 & & \text{in } a_1 & \\
e\ x_1 \ldots x_n & = & \text{let} & a_1 = e\ x_1 \\
 & & & a_2 = a_1\ x_2 \\
 & & & \vdots \\
 & & & a_n = a_{n-2}\ x_{n-1} \\
 & & \text{in } a_{n-1}\ x_n & \\
(\text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e)^{\#} & = & \text{let } \{x_i = e_i^{\#}\}_{i=1}^n \text{in } Ce^{\#} & \\
(c\ x_1 \ldots x_j)^{\#} & = & c\ x_1 \ldots x_j & \\
(\text{case } e \text{ of } \{c_j\ \vec{y_j} \to e_j\}_{j=1}^n)^{\#} & = & \text{case } e^{\#} \text{of} \{c_j\ \vec{y_j} \to e_j^{\#}\}_{j=1}^n &
\end{array}$$

The correctness of the new semantics follows as in the λ-normalisation cases:

Proposition 6 (Correctness of argument-normalisation)

$$\{\} : e \;\Downarrow\; \Delta : w \textit{ iff } \{\} : e^{\#} \;\Downarrow\; \Theta : w$$

$$\Gamma[x \mapsto \lambda\vec{y}^n.e] : x \;\downarrow\; \Gamma[x \mapsto \lambda\vec{y}^n.e] : x \qquad Lam_M$$

$$\frac{\Gamma : f \;\downarrow\; \Theta[v \mapsto \lambda\vec{y}^n.\lambda\vec{z}^m.e] : v \quad \Theta : \lambda\vec{z}^m.e[\vec{x}^n/\vec{y}^n] \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma : f\ \vec{x}^n \;\downarrow\; \Delta[u \mapsto w] : u}\ m > 0 \qquad App_M$$

$$\frac{\Gamma : f \;\downarrow\; \Delta[v \mapsto \lambda\vec{y}^m.e'] : v \quad \Delta : e'[\vec{x}^m/\vec{y}^m]\ \vec{x}^{(m+1)\ldots n} \;\downarrow\; \Theta : w}{\Gamma : f\ \vec{x}^n \;\downarrow\; \Theta : w}\ n \geq m \qquad App'_M$$

$$\frac{\Gamma : e \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma[x \mapsto e] : x \;\downarrow\; \Delta[u \mapsto w, x \mapsto \hat{w}] : x}\ w \text{ a lambda abstraction, } e \text{ not} \qquad Var_M$$

$$\frac{\Gamma : e \;\downarrow\; \Delta : c\ y_1 \ldots y_j}{\Gamma[x \mapsto e] : x \;\downarrow\; \Delta[x \mapsto c\ y_1 \ldots y_j] : c\ y_1 \ldots y_j}\ e \text{ not a lambda abstraction} \qquad VarC_M$$

$$\frac{\Gamma[x_i \mapsto e_i] : e \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma : \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \;\downarrow\; \Delta[u \mapsto w] : u} \qquad Let_M$$

$$\Gamma : c\ x_1 \ldots x_j \;\downarrow\; \Gamma : c\ x_1 \ldots x_j \qquad Cons_M$$

$$\frac{\Gamma : e \;\downarrow\; \Delta : c_k\ x_1 \ldots x_{a_k} \quad \Delta : e_k[x_1/y_{k1}, \ldots, x_{a_k}/y_{ka_k}] \;\downarrow\; \Theta[u \mapsto w] : u}{\Gamma : \text{case } e \text{ of } \{c_j\ \vec{y_j} \to e_j\}_{j=1}^n \;\downarrow\; \Theta[u \mapsto w] : u} \qquad Case_M$$

Figure 5: Natural Semantic specification of Lazy Evaluation for the λ-normalised, extended, language

Deriving the Abstract Machine

We again, informally, derive the abstract machine from the semantics given in Figure 4. We concentrate only on those instructions which have changed as a result of introducing multiple arguments.

As usual $Lam_E, Cons_E$ give rise to no instructions. Rules $Var_E, Case_E, Let_E$ are unchanged by the extension.

Rule App_E gives rise to two rules. One to start the computation of the body and one to do the substitution. Remember that this rule fires when there are not enough arguments to satisfy the bound abstraction:

1)	Γ	$(e\ \vec{x}^n)$	S	$\Longrightarrow$
	Γ	e	$(\vec{x}^n : S)$	
2)	Γ	$\lambda\vec{y}^n.e$	$(\vec{x}^n : S)$	$\Longrightarrow$
	Γ	$(e[\vec{x}^n/\vec{y}^n])$	S	

Rule App'_E also gives to two rules. The first is the same as the first above of course, and the second is to form the application of the substituted expression with the remaining arguments:

1)	Γ	$(e\ \vec{x}^n)$	S	$\Longrightarrow$
	Γ	e	$(\vec{x}^n : S)$	
3)	Γ	$\lambda\vec{y}^m.e$	$(\vec{x}^n : S)$	$\Longrightarrow$
	Γ	$(e[\vec{x}^m/\vec{y}^m]\ \vec{x}^{(m+1)\ldots n})$	S	

We only want rule 3 to fire if there are too many arguments on the stack for the abstraction. We can optimise this last rule somewhat, as we already know what do with an application (rule 1) – the application produced by rule 3 will just dump the rest of the arguments back on the stack and enter the new e (see rule 1)). We could just as well have combined it with rule 1 to produce:

3′)	Γ	$\lambda\vec{y}^m.e$	$(\vec{x}^n : S)$	$\Longrightarrow$
	Γ	$e[\vec{x}^m/\vec{y}^m]$	$(\vec{x}^{(m+1)\ldots n} : S)$	

The following section duplicates this development for above semantics modified for the STG machine. We will see there how rules 2 and 3′ can be further optimised, and how the squeeze operator can be defined.

6 The STG machine, naturally

Repeating the argument of section 3, we can modify the semantics in Figure 4 to detect values via the variable. This results in the semantics given in Figure 5.

Deriving the STG machine

Following the previous section exactly, rule App_M gives rise to the same two instructions, except now rule 2 detects the lambda abstraction in the heap instead of in the control. Remember that this rule fires when there are not enough arguments to satisfy the bound abstraction.

1)	Γ	$(e\ \vec{x}^n)$	S	$\Longrightarrow$
	Γ	e	$(\vec{x}^n : S)$	
2)	$\Gamma[v \mapsto \lambda\vec{y}^n.e]$	(v)	$(\vec{x}^n : S)$	$\Longrightarrow$
	Γ	$(e[\vec{x}^n/\vec{y}^n])$	S	

Rule App'_M follows similarly.

1)	Γ	$(e\ \vec{x}^n)$	S	$\Longrightarrow$
	Γ	e	$(\vec{x}^n : S)$	
3)	$\Gamma[v \mapsto \lambda\vec{y}^m.e]$	(v)	$(\vec{x}^n : S)$	$\Longrightarrow$
	Γ	$e[\vec{x}^m/\vec{y}^m]\ \vec{x}^{(m+1)\ldots n}$	S	

Again, we can optimise the application by incorporating rule 1:

3′)	$\Gamma[v \mapsto \lambda\vec{y}^m.e]$	(v)	$(\vec{x}^n : S)$	$\Longrightarrow$
	Γ	$e[\vec{x}^m/\vec{y}^m]$	$(\vec{x}^{(m+1)\ldots n} : S)$	

This begs the question: "How do we count the arguments?". An answer would be to just look at the arguments one by one to determine whether they are update markers, end of stack markers, case markers or the variables that we want. However, *we cannot get a case marker following an insufficient number of arguments.* To see this, we would need to have a: case $(f\ g)$ of *alts* which would dump the *alts* on the stack and execute the scrutiniser which will dump the argument g on the stack and enter f. Then we have *args* followed by *alts*. However, the above program is not well-typed, since the type of $f\ g$ has to be a constructor, and not a partial application.

With this knowledge, we can go back to rule 2 and see that it really looks like this (remember, there are not enough arguments to satisfy the abstraction):

$$\begin{array}{lllll} 2') & \Gamma[v \mapsto \lambda\vec{y}^n.e] & (v) & (\vec{x}^n : \#p : S) & \Longrightarrow \\ & \Gamma & (e[\vec{x}^n/\vec{y}^n]) & \#p : S & \end{array}$$

But since $e[\vec{x}^n/\vec{y}^n]$ is actually a lambda abstraction (there weren't enough arguments), the next rule to fire will be the var_2 (see Figure 3) which will do the heap update for p and remove it and re-execute the lambda abstraction. But this is the same as reentering v with the stack frame removed:

$$\begin{array}{lllll} 2'') & \Gamma[v \mapsto \lambda\vec{y}^n.e] & (v) & (\vec{x}^n : \#p : S) & \Longrightarrow \\ & \Gamma[p \mapsto e[\vec{x}^n/\vec{y}^n]] & (v) & (\vec{x}^n : S) & \end{array}$$

This is the famous *squeeze* operation, which squeezes out the update frame.

We can go even one step further. If we consider generating code for this abstract machine, then we will have a slight problem with rule $2''$, because it forms new code – that is, we need to generate code for $e[\vec{x}^n/\vec{y}^n]$ – for each partial substitution. We can get around this by replacing it with the following rule:

$$\begin{array}{lllll} 2''') & \Gamma[v \mapsto \lambda\vec{y}^n.e] & (v) & (\vec{x}^n : \#p : S) & \Longrightarrow \\ & \Gamma[p \mapsto v\ \vec{x}^n] & (v) & (\vec{x}^n : S) & \end{array}$$

which has the same semantics – instead of doing the substitution we make the binding re-apply the body to the arguments. This too forces us to generate code, except this time all we have to do is generate a number of code blocks for an *application*, one for each possible number of arguments. If a uniform representation is used when mapping this machine to hardware, then this is not costly at all (Peyton Jones 1992). In *loc. cit*, the representation of the closure is slightly different: let f be some arbitrary variable, then bind p to the closure $(f\ xs, E[f \mapsto v, xs \mapsto \vec{x}^n])$. The code for this can be shared between all partial applications to the same number of arguments. All that is required is a family of such code-blocks, one for each possible number of arguments.

The final derived machine *is* the STG machine.

7 Conclusions and Future Work

There have been many proposals for evaluating functional languages, such as the G-machine (Johnsson 1984), the spineless tagless G-machine (Peyton Jones 1992), the CMC (Lins 1987) and TIM (Fairbairn & Wray 1987). The work of Lins, Thompson & Peyton Jones (1994) has stressed the importance of relating different abstract machines, allowing us to examine the similarities and differences between the machines. In *loc. cit*, the TIM and CMC machines are related (albeit without sharing), and Peyton Jones & Lester (1992) go some way in showing the relationship between the TIM and the G-machine.

We believe that using natural semantic specifications to compare various machines is a rewarding route to take because:

- the specification provides a common ground from which we can compare the characteristics of different machines. Indeed, all of these machines mentioned above have in common that they attempt to reduce a language in a manner adhering to the principles of lazy evaluation, and this is exactly what the semantics describes. They only each do the same thing slightly differently. For instance, it is quite clear from the development in this paper that the hallmark of the STG machine is its characterisation of an abstraction via a variable. Unfortunately we have also seen that the 'common ground' that we use had to be enlarged somewhat by the inclusion of abstractions and applications to handle multiple arguments, but this should not deter us as any extension should still be consistent with the original semantics.

- the specification can be formally mapped to an abstract machine. As we have demonstrated, this mapping effortlessly handles complex notions such as shared partial abstractions, and provides a framework in which we can examine optimisations to the basic abstract machine (such as the *squeeze* operation). The process of generating an abstract machine can even be automated (Diehl 1996).

- the semantics is *minimal*, in the sense that there is no auxiliary machinery such as stacks and environments. It should be possible to refine this minimal model to build each of the abstract machines mentioned above, thus relating each of them, and making clear the design decisions that were made in creating them and the differences between them. We leave this to our future work.

- the semantics is *useful*, as demonstrated in (Turner, Wadler & Mossin 1995). Sansom & Peyton-Jones (1997) have also used a similar semantics to formally prove a profiling tool correct. This paper has established the relationship between Sestoft's and the STG machine, and so these results can now be carried through (formally) to the STG machine.

Related research includes the works of Launchbury and Sestoft mentioned in this paper. Peyton Jones (1992) mentions that the STG machine regards addresses as values, but takes this notion no further. Lins et al. (1994) relates the TIM and CMC machines, though without sharing. Their approach does not appear to provide a basis for proving other machines correct. (Sestoft 1997) shows a relationship between a one-argument lazified TIM and his derived machine. The work of (Douence & Fradet 1996) builds a very rich framework to compare various implementations and optimisations via successive transformations of a base combinator-like language. Their approach is quite different to ours: we have a high-level semantic specification which can only be transformed into the STG machine and which is related to some base natural semantic specification,

whereas they concentrate on transforming one base language into various abstract machines. Further work is necessary to gauge whether their transformation technique could be used to guide us into different semantic specifications and thus different abstract machines. The work of (Meijer 1988) is similar in that it advocates the successive refinement of a denotational semantics to various abstract machines. (Ariola, Felleisen, Maraist, Odersky & Wadler 1995) propose a reduction semantics of lazy evaluation, and further work will concentrate on determining whether the techniques used for creating abstract machines here carry over to this formalism. Both have their own advantages, as demonstrated in (Turner et al. 1995). Other related works include (Seaman & Purushothaman Iyer 1996) and (Seaman 1993). In their syntax, sharing is not explicitly enforced (leading to a slightly more complicated semantics).

We have not concentrated on the mapping between the STG machine and hardware at all, but this had a large impact on the design of the original STG machine (Peyton Jones 1992). It would be interesting to identify important mapping principles and highlight these in the semantics: for instance, the number of stacks or taglessness. This would allow us to derive an abstract machine even closer to the intended underlying architecture. This would also allow us to quantify the tradeoffs made between the extra heap allocation in the STG as opposed to the lambda instruction in Sestoft's machine. Following the methods of (Hannan 1991) it should be possible to produce machine code in a provably correct manner from the abstract machine.

Acknowledgements

The author thanks Daan Leijen, Pieter Hartel, Simon Peyton Jones and Peter Sestoft for their many useful and insightful comments. The author is supported by the Netherlands Computer Science Research Foundation with financial support from the Netherlands Organisation for Scientific Research (NWO).

References

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. & Wadler, P. (1995), A call-by-need lambda calculus, *in* 'Proceedings of the 22nd ACM Symposium on Principles of Programming Languages'.

Diehl, S. (1996), Semantics-directed generation of compilers and abstract machines, PhD thesis, Universität des Saarlandes.

Douence, R. & Fradet, P. (1996), A taxonomy of functional languages implementations, part II: Call-by-name, call-by-need and graph reduction, Technical Report 3050, Institut National de recherche en Informatique et Automatique (INRIA).

Fairbairn, J. & Wray, S. (1987), TIM: A simple, lazy abstract machine to execute supercombinators, *in* G. Kahn, ed., '3rd Functional Programming Languages and Computer Architecture', Vol. 274 of *LNCS*, Springer-Verlag, pp. 34–45.

Hannan, J. (1991), Making abstract machines less abstract, *in* J. Hughes, ed., '1991 Conference on Functional Programming Languages and Computer Architecture (FPCA)', Vol. 523 of *LNCS*, Springer-Verlag, pp. 618–635.

Johnsson, T. (1984), Efficient compilation of lazy evaluation, *in* 'Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction', Vol. 19(6) of *SIGPLAN notices*, ACM Press, pp. 58–69.

Launchbury, J. (1993), A natural semantics for lazy evaluation, *in* '20th Symposium on Principles of Programming Languages (POPL)', ACM Press, pp. 144–154.

Lins, R. D. (1987), Categorical multi-combinators, *in* G. Kahn, ed., '3rd Functional Programming Langauges and Computer Architecture', Vol. 274 of *LNCS*, Springer-Verlag, pp. 60–79.

Lins, R. D., Thompson, S. J. & Peyton Jones, S. (1994), 'On the equivalence between CMC and TIM', *Journal of Functional Programming* **4**(1), 47–63.

Meijer, E. (1988), A taxonomy of function evaluating machines, *in* T. Johnsson, S. Peyton Jones & K. Karlsson, eds, 'Proceedings of the workshop on Implementation of Functional Languages', Chalmers University of Technology, Technical Report 53.

Peyton Jones, S. L. (1992), 'Implementing lazy functional languages on stock hardware: The STG machine', *Journal of Functional Programming* **2**(2), 127–202.

Peyton Jones, S. L. (1996), Compiling Haskell by program transformation: a report from the trenches, *in* H. R. Nielson, ed., '6th European Symposium on Programming (ESOP'96)', Vol. 1058 of *LNCS*, Springer, pp. 18–44.

Peyton Jones, S. L. & Lester, D. (1992), *Implementing Functional Languages: A Tutorial*, Prentice Hall International Series in Computer Science, Prentice Hall.

Sansom, P. M. (1994), Execution Profiling for Non-strict Functional Languages, PhD thesis, University of Glasgow.

Sansom, P. M. & Peyton-Jones, S. L. (1997), 'Formally-based profiling for higher-order functional languages', *ACM Transactions on Programming Languages and Systems* **19**(2), 334–385.

Seaman, J. (1993), An Operational Semantics of Lazy Evaluation for Analysis, PhD thesis, Pennsylvania State University, Department of Computer Science and Engineering.

Seaman, J. & Purushothaman Iyer, S. (1996), 'An operational semantics of sharing in lazy evaluation', *Science of Computer Programming* **27**, 289–322.

Sestoft, P. (1997), 'Deriving a lazy abstract machine', *Journal of Functional Programming* **7**(3), 231–264.

Turner, D., Wadler, P. & Mossin, C. (1995), Once upon a type, *in* 'International Conference on Functional Programming Languages and Computer Architecture', ACM Press, pp. 1–11.