The Spineless Tagless G-Machine, as implemented by GHC

September 7, 2013

1 Introduction

This document presents the syntax of STG, and the cost semantics utilized for profiling. While this document will be primarily useful for people looking to work on profiling in GHC, the hope is that this will eventually expanded to also have operational semantics for modern STG.

While care has been taken to adhere to the behavior in GHC, these rules have not yet been used to perform any proofs. There is some sloppiness in here that probably would have to be cleaned up, especially with respect to let-no-escape. Some details are elided from this presentation, especially extra annotated data in the STG data type itself which is useful for code generation but not strictly necessary.

2 Grammar

2.1 Metavariables

We will use the following metavariables:

f, x, y, z	Variable names
K	Data constructor names
i, j, k, n	Indices to be used in lists

2.2 Preliminaries

Literals do not play a big role, so they are kept abstract:

lit ::= Literals, *basicTypes/Literal.lhs*:`Literal`

Primitive operations and foreign calls can influence the costs of an application, but because their behavior depends on the specific operation in question, they are kept abstract for simplicity's sake.

op ::= Primitive operation or foreign call, *stgSyn/StgSyn.lhs*:`StgOp`

2.3 Arguments

Arguments in STG are restricted to be literals or variables:

$$
\begin{array}{lcll}
a,\ b,\ c & ::= & & \text{Arguments, } \textit{stgSyn/StgSyn.lhs}\text{:}\texttt{StgArg} \\
 & | & x & \text{Variable}
\end{array}
$$

|	lit	Literal

2.4 Cost centres

Cost centres are abstract labels to which costs can be attributed. They are collected into cost centre stacks. Entering a function requires us to combine two cost-centre stacks ($\bowtie$), while entering a SCC pushes a cost-centre onto a cost-centre stack ($\triangleright$); both of these functions are kept abstract in this presentation. The special current cost-centre stack ($\bullet$) occurs only in STG and not at runtime and indicates that the lexically current cost-centre should be used at runtime (see the cost semantics for details). Some times we do not care about the choice of cost centre stack, in which case we will use the don't care cost centre stack.

cc	::=		Cost-centre, *profiling/CostCentre.lhs*:`CostCentre`

ccs	::=		
	|	$\bullet$	Current cost-centre stack
	|	$\_$	Don't care cost-centre stack
	|	$\mathsf{ccs} \bowtie \mathsf{ccs}'$	Function entry, *rts/Profiling.c*:`enterFunCCS`
	|	$\mathsf{ccs} \triangleright \mathsf{cc}$	Push a cost-centre, *rts/Profiling.c*:`pushCostCentre`

2.5 Expressions

The STG datatype that represents expressions:

e	::=		Expressions, *stgSyn/StgSyn.lhs*:`StgExpr`
	|	lit	Literal
	|	$x\ args$	Function application (or variable)
	|	$K\ args$	Saturated constructor application
	|	$\mathsf{op}\ args$	Saturated primitive application
	|	$\mathbf{case}\ e\ \mathbf{as}\ x\ \mathbf{of}\ \overline{alt_i}^{\,i}$	Pattern match
	|	$\mathbf{let}\ binding\ \mathbf{in}\ e$	Let binding
	|	$\mathbf{lne}\ binding\ \mathbf{in}\ e$	Let-no-escape binding
	|	$\mathbf{scc}\ \mathsf{cc}\ e$	Set cost-centre

STG is a lot like Core, but with some differences:

- Function arguments must be literals or variables (thus, function application does not allocate),
- Constructor and primitive applications are saturated,
- Let-bindings can only have constructor applications or closures on the right-hand-side, and
- Lambdas are forbidden outside of let-bindings.

The details of bindings for let statements:

$binding$	::=		Let-bindings, *stgSyn/StgSyn.lhs*:`StgBind`
	|	$x = rhs$	Non-recursive binding
	|	$\mathbf{rec}\ \overline{x_i = rhs_i}^{\,i}$	Recursive binding

rhs	::=		Right-hand sides, *stgSyn/StgSyn.lhs*:`StgRhs`
	|	cl	Closure
	|	$K\ \mathsf{ccs}\ args$	Constructor
	|	x	Indirection (runtime only)

cl	$::=$		StgRhsClosure
	$\mid$	$\lambda upd\, \mathsf{ccs}\, xs.e$	

Closures have an update flag, which indicates whether or not they are functions or thunks:

upd	$::=$		Update flag, *stgSyn/StgSyn.lhs*:`UpdateFlag`
	$\mid$	$\mathbf{r}$	Function (re-entrant closure)
	$\mid$	$\mathbf{u}$	Thunk (updatable closure)

Details for case alternatives:

alt	$::=$		Case alternative, *stgSyn/StgSyn.lhs*:`StgAlt`
	$\mid$	$K\, \overline{x_i}^{\,i} \to e$	Constructor applied to fresh names

3 Runtime productions

In our cost semantics, we will explicitly model the heap:

$\Gamma,\ \Delta,\ \Theta$	$::=$		Heap
	$\mid$	$\Gamma[Gp]$	Heap with assignment

Assignments on the heap are from names to heap values with an associated cost-centre stack. In our model, allocation produces a fresh name which acts as a pointer to the value on the heap.

Gp	$::=$		Heap assignment
	$\mid$	$x \overset{\mathsf{ccs}}{\mapsto} heap$	

$heap$	$::=$		Values on the heap
	$\mid$	cl	Closure
	$\mid$	$K\ args$	Data constructor
	$\mid$	x	Indirection

Execution procedes until a return value (a literal or a variable, i.e. pointer to the heap) is produced. To accomodate for let-no-escape bindings, we also allow execution to terminate with a jump to a function application of a let-no-escape variable.

ret	$::=$		Return values
	$\mid$	a	Normal return
	$\mid$	$\{x\ args\}$	Jump to let-no-escape

Values v are functions (re-entrant closures) and constructors; thunks are not considered vaules. Evaluation guarantees that a value will be produced.

Profiling also records allocation costs for creating objects on the heap:

θ	$::=$	
	$\mid$	$\{\}$
	$\mid$	$\{\mathsf{ccs} \mapsto cost\}$
	$\mid$	$\theta \uplus \theta'$

$cost$	$::=$	
	$\mid$	$\mathbf{alloc}\, K$
	$\mid$	$\mathbf{alloc}\, cl$

4 Cost semantics

The judgment can be read as follows: with current cost centre ccs and current heap Γ, the expression e evaluates to ret, producing a new heap Δ and a new current cost centre ccs', performing θ allocations.

$\boxed{\mathsf{ccs}, \Gamma : e \;\Downarrow_\theta\; \Delta : ret, \mathsf{ccs}'}$

$$\frac{}{\mathsf{ccs}, \Gamma : \mathsf{lit} \;\Downarrow_{\{\}}\; \Gamma : \mathsf{lit}, \mathsf{ccs}}\ \textsc{Lit}$$

$$\frac{x \overset{\mathsf{ccs}_0}{\mapsto} v \,\mathbf{in}\, \Gamma}{\mathsf{ccs}, \Gamma : x \cdot \;\Downarrow_{\{\}}\; \Gamma : x, \mathsf{ccs}}\ \textsc{Whnf}$$

$$\frac{\mathsf{ccs}_0, \Gamma : e \;\Downarrow_\theta\; \Delta : z, \mathsf{ccs}'}{\mathsf{ccs}, \Gamma[x \overset{\mathsf{ccs}_0}{\mapsto} \lambda \mathbf{u} \,\_\, \cdot . e] : x \cdot \;\Downarrow_\theta\; \Delta[x \overset{\mathsf{ccs}_0}{\mapsto} z] : z, \mathsf{ccs}'}\ \textsc{Thunk}$$

$$\frac{\begin{array}{l} f \overset{\mathsf{ccs}_0}{\mapsto} \lambda \mathbf{r}\, \mathsf{ccs}_1\, \overline{y_i}^{\,i}\, \overline{x_j}^{\,j} . e \,\mathbf{in}\, \Gamma \\ z \,\mathbf{fresh} \end{array}}{\mathsf{ccs}, \Gamma : f\, \overline{a_i}^{\,i} \;\Downarrow_{\{\}}\; \Gamma[z \overset{\mathsf{ccs}}{\mapsto} \lambda \mathbf{r} \,\_\, \overline{x_j}^{\,j} . f\, \overline{a_i}^{\,i}\, \overline{x_j}^{\,j}] : z, \mathsf{ccs}}\ \textsc{AppUnder}$$

$$\frac{\mathsf{ccs} \bowtie \mathsf{ccs}_0, \Gamma : e \;\Downarrow_\theta\; \Delta : z, \mathsf{ccs}'}{\mathsf{ccs}, \Gamma[f \overset{\mathsf{ccs}_0}{\mapsto} \lambda \mathbf{r} \bullet \overline{x_i}^{\,i} . e] : f\, \overline{a_i}^{\,i} \;\Downarrow_\theta\; \Delta : z, \mathsf{ccs}'}\ \textsc{App}$$

$$\frac{\begin{array}{l} \mathsf{ccs}, \Gamma : e \;\Downarrow_\theta\; \Delta : z, \mathsf{ccs}' \\ \mathsf{ccs}_1 \neq \bullet \end{array}}{\mathsf{ccs}, \Gamma[f \mapsto \lambda \mathbf{r}\, \mathsf{ccs}_1\, \overline{x_i}^{\,i} . e] : f\, \overline{a_i}^{\,i} \;\Downarrow_\theta\; \Delta : z, \mathsf{ccs}'}\ \textsc{AppTop}$$

$$\frac{\begin{array}{c} \mathsf{ccs}, \Gamma : f\, \overline{a_i}^{\,i} \;\Downarrow_\theta\; \Delta : f', \mathsf{ccs}' \\ \mathsf{ccs}, \Delta : f'\, \overline{b_j}^{\,j} \;\Downarrow_{\theta'}\; \Theta : z, \mathsf{ccs}'' \end{array}}{\mathsf{ccs}, \Gamma : f\, \overline{a_i}^{\,i}\, \overline{b_j}^{\,j} \;\Downarrow_{\theta \cup \theta'}\; \Theta : z, \mathsf{ccs}''}\ \textsc{AppOver}$$

$$\frac{z \,\mathbf{fresh}}{\mathsf{ccs}, \Gamma : K\, \overline{a_i}^{\,i} \;\Downarrow_{\{\mathsf{ccs} \mapsto \mathbf{alloc}\, K\}}\; \Gamma[z \overset{\mathsf{ccs}}{\mapsto} K\, \overline{a_i}^{\,i}] : z, \mathsf{ccs}}\ \textsc{ConApp}$$

$$\frac{\begin{array}{l} alt_j = K_k\, \overline{x_i}^{\,i} \to e' \\ \mathsf{ccs}, \Gamma : e \;\Downarrow_\theta\; \Delta[y \mapsto K_k\, \overline{a_i}^{\,i}] : y, \mathsf{ccs}' \\ \mathsf{ccs}, \Delta[y \mapsto K_k\, \overline{a_i}^{\,i}] : e'\, [y/x]\, \overline{[a_i/x_i]}^{\,i} \;\Downarrow_{\theta'}\; \Theta : z, \mathsf{ccs}'' \end{array}}{\mathsf{ccs}, \Gamma : \mathbf{case}\; e\; \mathbf{as}\; x\; \mathbf{of}\; \overline{alt_j}^{\,j} \;\Downarrow_{\theta \cup \theta'}\; \Theta : z, \mathsf{ccs}''}\ \textsc{Case}$$

$$\frac{\begin{array}{l} y \,\mathbf{fresh} \\ \mathsf{ccs}, \Gamma[y \overset{\mathsf{ccs}}{\mapsto} cl] : e\, [x/y] \;\Downarrow_\theta\; \Delta : z, \mathsf{ccs}' \end{array}}{\mathsf{ccs}, \Gamma : \mathbf{let}\; x = cl \;\mathbf{in}\; e \;\Downarrow_{\{\mathsf{ccs} \mapsto \mathbf{alloc}\, cl\} \cup \theta}\; \Delta : z, \mathsf{ccs}'}\ \textsc{LetClosure}$$

$$\frac{\begin{array}{l} y\ \textbf{fresh} \\ \mathsf{ccs}, \Gamma[y \overset{\mathsf{ccs}}{\mapsto} K\ \overline{a_i}^{\,i}] : e\,[x/y] \ \Downarrow_\theta\ \Delta : z, \mathsf{ccs}' \end{array}}{\mathsf{ccs}, \Gamma : \textbf{let}\, x = K\ \bullet\ \overline{a_i}^{\,i}\ \textbf{in}\ e\ \Downarrow_{\{\mathsf{ccs} \mapsto \textbf{alloc}\, K\} \cup \theta}\ \Delta : z, \mathsf{ccs}'}\ \textsc{LetCon}$$

$$\frac{\begin{array}{c} \mathsf{ccs}, \Gamma : e\ \Downarrow_\theta\ \Delta : \{f\ \overline{a_i}^{\,i}\}, \mathsf{ccs}' \\ \mathsf{ccs}, \Delta : e'\ \overline{[a_i/x_i]}^{\,i}\ \Downarrow_{\theta'}\ \Theta : z, \mathsf{ccs}'' \end{array}}{\mathsf{ccs}, \Gamma : \textbf{lne}\, f = \lambda upd\ \_\ \overline{x}^{\,i}.e'\ \textbf{in}\ e\ \Downarrow_{\theta \cup \theta'}\ \Theta : z, \mathsf{ccs}''}\ \textsc{LneClosure}$$

$$\frac{\begin{array}{c} \mathsf{ccs}, \Gamma : e\ \Downarrow_\theta\ \Delta : x, \mathsf{ccs}' \\ \mathsf{ccs}, \Delta : K\ \overline{a_i}^{\,i}\ \Downarrow_{\theta'}\ \Theta : z, \mathsf{ccs}'' \end{array}}{\mathsf{ccs}, \Gamma : \textbf{lne}\, x = K\ \_\ \overline{a_i}^{\,i}\ \textbf{in}\ e\ \Downarrow_{\theta \cup \theta'}\ \Theta : z, \mathsf{ccs}''}\ \textsc{LneCon}$$

$$\frac{\mathsf{ccs} \triangleright \mathsf{cc}, \Gamma : e\ \Downarrow_\theta\ \Delta : z, \mathsf{ccs}'}{\mathsf{ccs}, \Gamma : \textbf{scc}\, \mathsf{cc}\ e\ \Downarrow_\theta\ \Delta : z, \mathsf{ccs}'}\ \text{SCC}$$

4.1 Notes

- In the THUNK rule, while the indirection is attributed to ccs_0, the result of the thunk itself (y) may be attributed to someone else.
- APPUNDER and APPOVER deal with under-saturated and over-saturated function application. The implementations of APP rules are spread across two different calling conventions for functions: slow calls and direct calls. Direct calls handle saturated and over-applied cases (*codeGen/StgCmmLayout.hs*:`slowArgs`), while slow calls handle all cases (*utils/genapply/GenApply.hs*); in particular, these cases ensure that the current cost-center reverts to the one originally at the call site. The lack of an allocation record for the PAP is a bug.
- The APP rule demonstrates that modern GHC profiling uses neither evaluation scoping or lexical scoping; rather, it uses a hybrid of the two (though with an appropriate definition of $\bowtie$, either can be recovered.) The presence of cost centre stacks is one of the primary differences between modern GHC and Sansom'95.
- The APPTOP rule utilizes $\bullet$ to notate when a function should influence the current cost centre stack. The data type used here could probably be simplified, since we never actually take advantage of the fact that it is a cost centre.
- As it turns out, the state of the current cost centre after evaluation is never utilized. In the original Sansom'95, this information was necessary to allow for the implementation of lexical scoping; in this presentation, all closures must live on the heap, and the cost centre is thus recorded there.
- LNECLOSURE must explicitly save and reset the ccs when the binding is evaluated, whereas LETCLOSURE takes advantage of the fact that when the closure is allocated on the heap the ccs is saved. (ToDo: But isn't there a behavior difference when the closure is re-entrant? Note that re-entrant/updatable is indistinguishable to a let-no-escape.)
- Recursive bindings have been omitted but they work in the way you would expect.