The Spineless Tagless G-machine

Pedro Vasconcelos

March 28, 2023

The Spineless Tagless G-Machine

- An abstract machine for functional languages with non-strict semantics (i.e. lazy evaluation)
- Evolution of the *G-machine* and the *Spineless G-machine*

Bibliography: *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, Version 2.5. Simon L. Peyton Jones, 1992.

What is the STG?

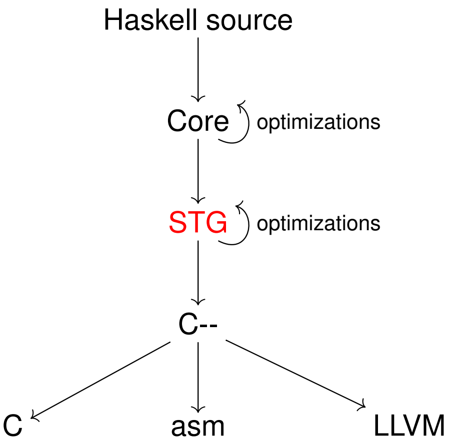
- It is a very restricted functional language
- Unlike Core and Haskell, STG is not typed
- Has a step-by-step operational semantics
- Can be directly translated into a conventional arquitecture
- Allows more optimizations as program transformations
- Used in the GHC backend (after Core):

  ```
  ghc -c -ddump-stg-from-core Foo.hs
  ```

  or

  ```
  ghc -c -ddump-stg-final Foo.hs
  ```

Global view of GHC



STG design decisions

- Lambda abstractions and applications handle multiple arguments

$$\lambda\{x_1, x_2, \ldots, x_n\}.\, e$$
$$f\{a_1, a_2, \ldots, a_n\}$$

  - More efficient than unary application
  - But partial application is allowed (*currying*)
- Arguments of applications must be atomic (variables or literals)
  - Complex expressions must be named using *let*
  - Arguments are built *before* the function call
  - Makes the evaluation order explicit

STG design decisions (cont.)

- Uniform heap representation: closures
  - Lambda-abstractions, thunks (i.e. unevaluated expressions) and constructors
  - Avoid the necessity for *tags*
- Constructors and primitive applications must be fully saturated
- Uniform treatment of data structures:
  - Decomposition using only simple case expressions
  - No special treatment for booleans, lists or tuples
  - Suports unboxed representations for primitive values (e.g. integers)

The STG language

$$
\begin{array}{rcll}
\textit{expr} & ::= & \text{let } \textit{binds} \text{ in } \textit{expr} & \text{local definitions} \\
& | & \text{letrec } \textit{binds} \text{ in } \textit{expr} & \text{recursive definitions} \\
& | & \text{case } \textit{expr} \text{ of } \textit{alts} & \text{case expressions} \\
& | & \textit{var atoms} & \text{function application} \\
& | & \textit{constr atoms} & \text{constructor application} \\
& | & \textit{prim atoms} & \text{primitive application} \\
& | & \textit{literal} & \text{primitive values} \\
\\
\textit{binds} & ::= & \textit{var}_1 = \textit{lf}_1 ; \ldots ; \textit{var}_n = \textit{lf}_n & (n \geq 1) \\
\textit{lf} & ::= & \textit{vars}_f \ \backslash\pi \ \textit{vars}_a \ \text{->} \ \textit{expr} & \text{lambda-form} \\
\pi & ::= & \text{u} \quad | \quad \text{n} & \text{update-flag} \\
\\
\textit{vars} & ::= & \{\textit{var}_1, \ldots, \textit{var}_n\} & (n \geq 0) \\
\textit{atoms} & ::= & \{\textit{atom}_1, \ldots, \textit{atom}_n\} & (n \geq 0) \\
\textit{atom} & ::= & \textit{var} \quad | \quad \textit{literal} &
\end{array}
$$

Lambda-forms

Lambda abstractions are represented by *lambda-forms*:

$$\underbrace{\{v_1, \ldots, v_m\}}_{\text{free variables}} \backslash\pi \underbrace{\{x_1,\ \ldots, x_n\}}_{\text{bound variables}} \ \texttt{->}\ \mathit{expr} \qquad \text{where } n \geq 0,\ m \geq 0$$

Denotationaly:

$$\lambda x_1. \ldots \lambda x_n. \mathit{expr}$$

Operationally:

- π indicates whether we should perform an update after the a reduction
- the free variables allow determining the closure size

Case expressions

$$
\begin{array}{l}
\text{case } \mathit{expr}_0 \text{ of} \\
\qquad \mathit{constr}_1\ \mathit{vars}_1\ \texttt{->}\ \mathit{expr}_1\,; \\
\qquad \qquad \vdots \\
\qquad \mathit{constr}_n\ \mathit{vars}_n\ \texttt{->}\ \mathit{expr}_n
\end{array}
$$

- Sole mechanism for decomposing structured values
- Restricted to simple patterns (i.e. one constructor at a time)
- Also used for primitive values (e.g. integers)

Program

- A program is a sequence of global bindings
- Entry point: the expression *main*

$$
\begin{aligned}
g_1 &= \{\ldots\} \backslash \pi_1 \; \{\ldots\} \; \texttt{->} \; \mathit{expr}_1 \; ; \\
g_2 &= \{\ldots\} \backslash \pi_2 \; \{\ldots\} \; \texttt{->} \; \mathit{expr}_2 \; ; \\
&\vdots \\
g_n &= \{\ldots\} \backslash \pi_n \; \{\ldots\} \; \texttt{->} \; \mathit{expr}_n \; ; \\
\text{main} &= \{\ldots\} \backslash \pi_{n+1} \; \{\} \; \texttt{->} \; \mathit{expr}_{n+1}
\end{aligned}
$$

Example

```
map f [] = []
map f (y:ys) = (f y) : (map f ys)
```

Translation to Core:

```
map = \a b (f::a->b) (xs::[a]) ->
    case xs of
       [] -> [] @b
       (y:ys) -> : @b (f y) (map @a @b f ys)
```

Translation to STG:

```
map = {} \n {f,xs} ->
   case xs of
     []{} -> []{};
     :{y,ys} ->
       let fy = {f,y} \u {} -> f{y};
           mfy= {f,ys} \u {} -> map{f,ys}
       in :{fy,mfy}
```

Translating Core into STG

1. Replace binary application with multi-aplication:

$$(\ldots((f\ e_1)\ e_2)\ldots e_n) \Longrightarrow f\{e_1,\ e_2,\ \ldots,\ e_n\}$$

2. Saturate constructors and primitive operations using η-expansion if necessary:

$$c\{e_1,\ \ldots,\ e_n\} \Longrightarrow \lambda y_1 \ldots y_m.\, c\{e_1,\ \ldots,\ e_n,\ y_1, \ldots, y_m\}$$

3. Introduce temporaries for atomic non-arguments:

$$f\{\ldots,\ \underbrace{expr}_{\text{complex}},\ \ldots\} \Longrightarrow \text{let } t = expr \text{ in } f\{\ldots,\ t,\ \ldots\}$$

4. Convert the right-hand sides of let bindings into *lambda-forms* by adding free variables and update flags

When to do updates?

Non-updatable:

1. Lambda abstractions:

$$vs \ \backslash\mathrm{n} \ xs \ \texttt{->} \ expr \qquad (|xs| > 0)$$

2. Partial applications:

$$vs \ \backslash\mathrm{n} \ \{\} \ \texttt{->} \ f\{x_1, \ldots, x_m\} \qquad \ldots$$

3. Constructors:

$$vs \ \backslash\mathrm{n} \ \{\} \ \texttt{->} \ c\{x_1, \ldots, x_m\} \qquad \ldots$$

Updatable: everything else (*thunks*)

$$vs \ \backslash\mathrm{u} \ \{\} \ \texttt{->} \ expr$$

But: static analysis may do better (see bibliography).

Arithmetic operations

```
foo x = let y = div 1 x
         in if x==0 then ... else y
```

- We need to construct *thunks* for integers, etc.
- Naive solution: represent integers as “boxed” values in the *heap*
- Makes arithmetic operations very inefficient

Boxed vs. unboxed integers

```
data Int = I# Int#
-- Int  :  boxed integer
-- Int# : unboxed integer

plusInt :: Int -> Int -> Int
plusInt = {} \n {bx,by} ->
    case bx of
      I# x -> case by of
                I# y -> case (x +# y) of
                          z -> I# z
```

Unboxed values in the STG

- The STG primitive operations work over unboxed values (same as Core)
- Boxed values are built using algebraic data type constructors
- Allows expressing optimizations as program transformations

Limitations:

- Unboxed values cannot be used in polymorphic functions
- Unboxed values must be bound using *case* instead of *let*

Example

```
-- Haskell
foo :: Int -> Int
foo x = 2*x + 1

-- Core
foo  = \(bx :: Int) ->
      case bx of { I# x ->  I# (+# (*# 2# x) 1#) }

-- STG
foo = {} \n {bx} ->
      case bx of {
         I# x ->  case (*# 2# x) of
                    y -> case (+# y 1#) of
                           z -> I# z
    }
```

Operational semantics

Each STG sintactical construct has an operational interpretation:

let	allocation of *thunks/closures*;
case	evaluation and selection;
function application	unconditional jump;
constructor application	return to a continuation.

State of the abstract machine

Code next action to execute

Argument stack sequence of values

Return stack sequence of continuations

Update stack sequence of *update frames*

Heap table from addresses to *closures*

Globals addres of global *closures* (immutable)

Values

Two kind of values:

Addr a	address of a closure in the *heap*
Int n	primitive integer

NB: the *Addr* e *Int* tags are only for presentation — they are not necessary during execution.

Environments

An environment ρ associates values to variables:

$$\rho = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$$

We use σ for the global environment:

$$\sigma = [g_1 \mapsto \text{Addr } a_1, \ldots\ g_m \mapsto \text{Addr } a_m]$$

Value of an atom

Auxliary function:

$$\begin{aligned} \mathsf{val}\ \rho\ \sigma\ n &= \mathsf{Int}\ n \\ \mathsf{val}\ \rho\ \sigma\ x &= \rho\, x \qquad (x \in \mathsf{dom}\ \rho) \\ \mathsf{val}\ \rho\ \sigma\ x &= \sigma\, x \qquad (x \in \mathsf{dom}\ \sigma) \end{aligned}$$

Heap

The heap associates *addresses* to *closures*:

$$h = \begin{bmatrix} a_1 \mapsto closure_1 \\ \vdots \\ a_n \mapsto closure_n \end{bmatrix}$$

Each closure is a pair of a *lambda-form* and an environment

$$closure = (\underbrace{vs \setminus\pi\ xs{-}{>}expr}_{\text{lambda-form}}, \underbrace{ws}_{\text{environment}})$$

where $|vs| = |ws|$.

Code

Four states:

Eval e ρ evalute e in environment ρ;

Enter a evaluate the *closure* in address a;

ReturnCon c ws return a constructor;

ReturnInt k return an integer.

Initial state

Code	ArgStk	ReturnStk	UpdateStk	Heap	Globs
Eval $(main\{\})$ [ ]	$\{\}$	$\{\}$	$\{\}$	h_0	σ

$$\sigma = [g_1 \mapsto \text{Addr } a_1, \ldots, g_n \mapsto \text{Addr } a_n]$$

$$h_0 = \begin{bmatrix} a_1 \mapsto (vs_1 \ \backslash\pi \ xs_1 \text{->} e_1) \ (\sigma \ vs_1) \\ \vdots \\ a_n \mapsto (vs_n \ \backslash\pi \ xs_n \text{->} e_n) \ (\sigma \ vs_n) \end{bmatrix}$$

Function application

Code	ArgStk	ReturnStk	UpdateStk	Heap
Eval $(f\ xs)\ \rho$	as	rs	us	h

such that val $\rho\ \sigma\ f = a$

$$\Downarrow$$

Enter a	$(\text{val}\ \rho\ \sigma\ xs) \mathbin{+\!\!+} as$	rs	us	h

Enter non-updatable thunk

Code	ArgStk	ReturnStk	UpdateStk	Heap
Enter a	as	rs	us	h

such that $h\ a = (vs \setminus \mathrm{n}\ xs \texttt{->} e,\ ws_f)$

$\mathrm{length}\ as \geq \mathrm{length}\ xs$

$$\Downarrow$$

Eval $e\ \rho$	as'	rs	us	h

where $ws_a \mathbin{++} as' = as$

$\mathrm{length}\ ws_a = \mathrm{length}\ xs$

$\rho = [vs \mapsto ws_f,\ xs \mapsto ws_a]$

Let expressions

$$
\begin{array}{ccc|c|c|c|c}
 & \textcolor{red}{\text{Code}} & & \textcolor{red}{\ldots} & \textcolor{red}{\ldots} & \textcolor{red}{\ldots} & \textcolor{red}{\ldots} \\
\text{Eval} & \left(\begin{array}{ll} \text{let} & x_1 = vs_1 \backslash \pi_1 ys_1 \text{->} e_1 \\ & \vdots \\ & x_n = vs_n \backslash \pi_n ys_n \text{->} e_n \\ \text{in} & e \end{array}\right) & \rho & as & rs & us & h
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{c|c|c|c|c}
\text{Eval } e\ \rho' & as & rs & us & h'
\end{array}
$$

where

$$\rho' = \rho[x_1 \mapsto \text{Addr } a_1, \ldots, x_n \mapsto \text{Addr } a_n]$$

$$
h' = h \left[\begin{array}{l} a_1 \mapsto (vs_1 \backslash \pi_1 ys_1 \text{->} e_1,\ \rho\ vs_1) \\ \vdots \\ a_n \mapsto (vs_n \backslash \pi_n ys_n \text{->} e_n,\ \rho\ vs_n) \end{array}\right]
$$

Letrec expressions

- Analogous to the previous rule
- Modification: use ρ' instead ρ in the definition of h'
- See the bibliography

Case expressions

Code	ArgStk	ReturnStk	UpdateStk	Heap
Eval (case e of $alts$) ρ	as	rs	us	h

$$\Downarrow$$

Eval e ρ	as	$(alts, \rho) : rs$	us	h

Evaluate a constructor

Code	ArgStk	ReturnStk	UpdateStk	Heap
Eval $(c\ xs)\ \rho$	as	rs	us	h

$$\Downarrow$$

ReturnCon c (val $\rho\ \sigma\ xs$)	as	rs	us	h

Return a constructor

Code	ArgStk	ReturnStk	UpdateStk	Heap
ReturnCon c ws	as	$(alts, \rho) : rs$	us	h

such that $alts = \{\ldots;\ c\ xs \texttt{->} e;\ \ldots\}$

$$\Downarrow$$

Eval $e\ \rho[xs \mapsto ws]$	as	rs	us	h

Enter an updatable thunk

$$
\begin{array}{c|c|c|c|c}
\text{Code} & \text{ArgStk} & \text{ReturnStk} & \text{UpdateStk} & \text{Heap} \\
\text{Enter } a & as & rs & us & h \\
\multicolumn{5}{c}{\text{such that } h\ a = (vs\ \backslash\text{u}\ \{\}\texttt{->}e,\ ws_f)} \\
\multicolumn{5}{c}{\Downarrow} \\
\text{Eval } e\ \rho & \{\} & \{\} & (as, rs, a) : us & h \\
\multicolumn{5}{c}{\text{where } \rho = [vs \mapsto ws_f]}
\end{array}
$$

Update a constructor

Code	ArgStk	ReturnStk	UpdateStk	Heap
ReturnCon c ws	$\{\}$	$\{\}$	$(as_u, rs_u, a_u) : us$	h

$$\Downarrow$$

ReturnCon c ws	as_u	rs_u	us	h_u

where vs arbitrary variables
length vs $=$ length ws
$h_u = h[a_u \mapsto (vs \setminus \mathrm{n}\ \{\} \texttt{->}\ c\ vs,\ ws)]$

Update (under application)

Code	ArgStk	ReturnStk	UpdateStk	Heap
Enter a	as	$\{\}$	$(as_u, rs_u, a_u) : us$	h

such that $h\ a = (vs\ \backslash\text{n}\ xs\ \text{->}\ e)\ ws_f$
$\text{length}\ as < \text{length}\ xs$

$$\Downarrow$$

Enter a	$as ++ as_u$	rs_u	us	h_u

where $xs_1 ++ xs_2 = xs$
$\text{length}\ xs_1 = \text{length}\ as$
$h_u = h[a_u \mapsto ((vs ++ xs_1)\ \backslash\text{n}\ xs_2\ \text{->}\ e)\ (ws_f ++ as)]$

Evaluate an integer

Code	ArgStk	ReturnStk	UpdateStk	Heap
Eval k ρ	*as*	*rs*	*us*	*h*

$$\Downarrow$$

ReturnInt k	*as*	*rs*	*us*	*h*

Arithmetic operations

Code	ArgStk	ReturnStk	UpdateStk	Heap
Eval $(\oplus\{x_1, x_2\})\ \rho$	as	rs	us	h

if $\rho\ x_1 = \text{Int}\ i_1,\ \rho\ x_2 = \text{Int}\ i_2$

$$\Downarrow$$

ReturnInt $(i_1 \oplus i_2)$	as	rs	us	h

Return an integer

Code	ArgStk	ReturnStk	UpdateStk	Heap
ReturnInt k	as	$(\{\ldots;\ k\text{->}e; \ldots\},\ \rho) : rs$	us	h

$$\Downarrow$$

Eval e ρ	as	rs	us	h

Integers: default return

Code	ArgStk	ReturnStk	UpdateStk	Heap
ReturnInt k	as	$\left(\begin{array}{l} k_1 \texttt{->} e_1; \\ \vdots \\ k_n \texttt{->} e_n; \\ x \texttt{->} e \end{array}, \rho\right) : rs$	us	h

if $k \neq k_i \quad (1 \leq i \leq n)$

$$\Downarrow$$

Eval $e\ \rho'$	as	rs	us	h

where $\rho' = \rho[x \mapsto \text{Int } k]$.

Reflections

- The original STG used a push-enter evaluation model for function application
- This is an elegant way of implementing lazy evaluation
- More recent GHC uses the eval-apply model because it is more efficient in practice
- The changes are described in the following paper: *How to make a fast curry*, S. Peyton-Jones, 2004. `https://www.microsoft.com/en-us/research/publication/make-fast-curry-pushenter-vs-evalapply`

More information

An STG simulator written in Haskell:
https://github.com/quchen/stgi