Implementing Lazy Functional Languages

the Spiness Tagless G-machine

Ershad Rahimikia

Functional Programming (CAS 743)

March 2006

What's Spineless Tagless G-Machine?

- An abstract G-Machine designed to support nor strict higher-order languages
- There are some other recent abstract machines l G-Machine, Spineless G-Machine, CASE, HDG

Outline

- Some general design issues
  - Closures
  - Thunks
  - Updating
- STG language
  - STG syntax
  - Operational semantics
- Mapping the abstract machine to hardware

Representation of Closures

- The heap contains of
  - Head normal forms (values)
    - Function Values
    - Data Values
  - Unevaluated suspensions (thunks)
- A term closure refers to values and thunks
  - STG Machine represents both the same (tagless)
- A closure consists of: a code pointer, free variables
- Environment pointer refers to the closure to be evalualuated.
- Access
  - Free variables : offsets from Environment Pointer
  - Arguments : Parameter passing mechanism (stacks)

Updating Strategies

The naive reduction method

- Update the graph after each reduction
- G-Machine

The call method

- The code to force the closure checks the status flag

The self-updating method

- The code to force pushes a continuation stack
- If the closure is a thunk it arranges for ar update

STG language

All function/ constructor args are sim
vars or const.

All const. and built-in ops are saturat

Pattern matching by case expressions

Binding

f = {v1, v2 ... vn} \ π {x1, x2, ... xm} → e}

Transformation Steps

Replace binary app. to multiple app.

(...((f e1)e2)) en → f {e1,e2 ... en}

Saturate const. and built-in operatior

c {e1...en} → λy1...ym.c {e1...e2,y1...ym}

Name non-atomic function argument and lambda abstractions by let expressions

Convert right-hand of each let to lam form

Free variables and update flag info

Sample function in STG

In Haskell:

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

In STG

```
map = {} \ n {f,xs} →
    case xs of
      Nil {} → Nil {}
      Cons {y,ys} → let fy = {f,y} \ u {} → f{y}
                        mfy = {f,ys} \ u {} → m
{f,ys}
                    in Cons {fy, mfy}
```

Non-Updatable and Updatal

Non-Updatable

Manifest Functions → No empty argument lis

Partial Applications

vs \ n {} → f{x1 ... xm} (f a manifest with n > m arit

Constructors

vs \ n {} → c {x1 ... xm}

Updatable

Thunks → Non of above (empty argument list

Operational Semantics

Each state consists of:

- The code
- The argument stack, as, which contains values
- The return stack, rs, which contains continuations
- The update stack, us, which contains upd frames
- The heap, h, which contains closures
- The global environment, σ, which gives t addr of all closures at top level.

Initial State

Code	Arg Stk	Return Stk	Update Stk	Heap
$Eval\,(main\{\})\{\}$	$\{\}$	$\{\}$	$\{\}$	h_{init}

$$\mathrm{h}_{init} = \begin{bmatrix} a_1 \rightarrow (vs_1 \rightarrow \backslash \pi_1 \;\; xs_1 \rightarrow e_1)(\sigma\, vs_1) \\ \ldots \\ a_n \rightarrow (vs_n \rightarrow \backslash \pi_n \;\; xs_n \rightarrow e_n)(\sigma\, vs_n) \end{bmatrix}$$

$$\delta = \begin{bmatrix} \mathrm{g}_1 \rightarrow (Addr\ \mathrm{a}_1) \\ \ldots \\ g_n \rightarrow (Addr\ \mathrm{a}_n) \end{bmatrix}$$

Application (Eval)

$$Eval\,(f\ xs)\rho \quad as \quad rs \quad us \quad h \quad \delta$$

such that $val\ \rho\ \delta\ f = Addr\ a$

$$\Rightarrow \quad Enter\ a \quad (val\ \rho\ \sigma\ xs) ++ as \quad rs \quad us \quad h \quad \sigma$$

Application (Enter)

$$Enter\ a \quad as \quad rs \quad us \quad h[a \rightarrow (vs \backslash \mathrm{n}\ xs \rightarrow e) ws_f\,]$$

such that $\mathrm{length}\ (\mathrm{as}) \geq \mathrm{length}\ (\mathrm{xs})$

$$\Rightarrow Eval\ e\ \rho \quad as' \quad rs \quad us \quad h \quad \delta$$

where $ws_{a\,++}as' = as$
$length\,(xs_a) = length\,(xs)$
$\rho = [vs \rightarrow ws_f, xs \rightarrow ws_a]$

Case Expressions

$$\begin{array}{lllllll} & Eval\,(case\ e\ of\ alts)\,\rho & as & rs & us & h & \delta \\ \Rightarrow & Eval\ e\ \rho & as & (alts,\rho):rs & us & h & \delta \end{array}$$

$$\begin{array}{lllllll} & Eval\,(c\ xs)\,\rho & as & rs & us & h & \sigma \\ \Rightarrow & \text{ReturnCon}\ c\ (val\ \rho\ \delta\ xs) & as & rs & us & h & \delta \end{array}$$

$$\begin{array}{lllllll} & \text{ReturnCon}\ c\ ws & as & (...;c\ vs \rightarrow e;...,\rho):rs & us & h & \\ \Rightarrow & Eval\ e\ \rho[vs \rightarrow ws] & as & rs & us & h & \end{array}$$

Update (ReturnCon)

$$\text{ReturnCon}\ \ c\ \ ws\ \ \{\}\ \ \{\}\ \ (as_u, rs_u, a_u):us\ \ \ h\ \ \ \sigma$$

$$\text{ReturnCon}\ \ c\ \ ws\ \ \ as_u\ \ \ rs_u\ \ \ us\ \ \ h_u\ \ \ \sigma$$

where $\text{length}\,(vs) = \text{length}\,(ws)$

$$h_u = h[a_u \rightarrow (vs \setminus n\,\{\} \rightarrow c\ vs)ws]$$

Update (Enter)

$$Enter\ a\ \ \ as\ \ \ rs\ \ \ us\ \ \ h[a \rightarrow (vs \setminus u\,\{\} \rightarrow e)ws_f]$$

$$\Rightarrow Eval\ e\ \rho\ \ \ \{\}\ \ \ \{\}\ \ \ (as, rs, a):us\ \ \ \ h\ \ \ \sigma$$

$$where\ \ \rho = [vs \rightarrow ws_f]$$

Implementation on Hardwar

- Generating ANSI- standard c
  - Portable
  - Performance problems
- Generating non- standard c (like Gnu
  - Not portable
  - Using special facilities for performance
- Generating native machine code
  - Not portable
  - Complex

Tiny interpreter in STG- Machine

while (TRUE) {cont = (*cont);}

Each structure in STG language will b compiled into a C function

Closures are implemented in data structures in heap

Arguments, Continuations and Updat Frames are kept in stacks

A sample (Application of f)

apply = {} \ n { f, x } → f{ x, x, x }

```
Node = SpA[0];  /*Grab f into Node Register*/
t = SpA [1];    /*Grab x into a local variable*/
SpA [0] = t     /*Push extra args*/
SpA [-1] = t
SpA = SpA – 1   /*Adjust stack pointer*/
ENTER (Node)    /*Enter f*/
```

References

SL Peyton Jones [July 1992], Implementing lazy functional languages on stock hardware : The Spineless Tagless G-machine Version 2.5

Thank You!

Any Questions?