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 operatio[…]
  - 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 {} → {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 upc 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}

$$h_{init} = \begin{bmatrix} a_1 \to (vs_1 \to \backslash \pi_1 \;\; xs_1 \to e_1)(\sigma \; vs_1) \\ \ldots \\ a_n \to (vs_n \to \backslash \pi_n \;\; xs_n \to e_n)(\sigma \; vs_n) \end{bmatrix}$$

$$\delta = \begin{bmatrix} g_1 \to (Addr \; a_1) \\ \ldots \\ g_n \to (Addr \; a_n) \end{bmatrix}$$

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)

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

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

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

$$\begin{aligned} where \quad & ws_a {+}{+} as' = as \\ & length(xs_a) = length(xs) \\ & \rho = [vs \to ws_f, xs \to ws_a] \end{aligned}$$

Case Expressions

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

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

$$
\begin{array}{llllll}
 & \mathrm{ReturnCon}\ c\ ws & as & (\ldots; c\ vs \to e; \ldots, \rho):rs & us & h \\
\Rightarrow & Eval\ e\ \rho[vs \to ws] & as & rs & us & h
\end{array}
$$

Update (ReturnCon)

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

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

$$
\begin{array}{ll}
where & \mathrm{length\ (vs) = length\ (ws)} \\
 & \mathrm{h_u} = \mathrm{h}[a_u \to (vs \setminus \mathrm{n}\,\{\} \to c\ vs)ws]
\end{array}
$$

Update (Enter)

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

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

$$where\quad \rho = [vs \to ws_f]$$

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

References

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

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*/
```

Thank You!

Any Questions?