Spineless Tagless G-machine

Pedro Vasconcelos

19 de Maio de 2020

Spineless Tagless G-Machine

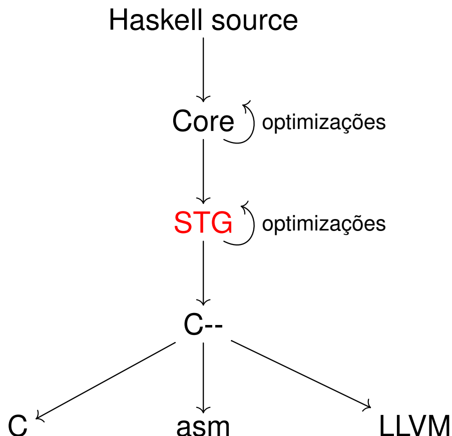
- Uma máquina abstracta para linguagens funcionais com semântica não-estrita (*lazy evaluation*)
- Evolução da *G-machine* e da *Spineless G-machine*

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

O que é a STG?

- É uma linguagem funcional muito restricta
- Com uma semântica operacional passo-a-passo
- Pode ser facilmente implementada numa máquina convencional
- Possibilita muitas optimizações como transformações de programa
- Utilizada no *backend* do Glasgow Haskell Compiler (GHC)

Visão global do GHC



Concepção geral

- Abstracções-λ e aplicações com vários argumentos

$$\lambda x_1.\lambda x_2.\ldots.\lambda x_n.e \quad \simeq \quad \lambda\{x_1, x_2, \ldots, x_n\}.e$$

  - Mais eficiente do que aplicação unária
  - Aplicação parcial é permitida (*currying*)
- Argumentos de aplicações devem ser atómicos (variáveis ou literais)
  - Expressões complexas devem ser nomeadas usando *let*
  - Os argumentos são construidos *antes* de invocar a função
  - Torna explícita a ordem de avaliação

Concepção geral (cont.)

- Representação uniforme na *heap*: closures
  - Abstracções-λ, *thunks* e dados algébricos
  - Evita a necessidade de *tags*
- Construtores e operações primitivas saturados: número fixo de argumentos
- Tratamento uniforme de estruturas de dados:
  - Não há casos especiais para booleanos, listas ou tuplos
  - Decomposição usando apenas expressões-*case* simples
  - Suporta representações optimizadas (*unboxed*)

1 Linguagem STG
2 Semântica operacional
3 Observações

Linguagem STG

$expr$	$::=$	let $binds$ in $expr$	definições locais
	$\mid$	letrec $binds$ in $expr$	definições recursivas
	$\mid$	case $expr$ of $alts$	discriminação
	$\mid$	$var\ atoms$	aplicação de função
	$\mid$	$constr\ atoms$	aplicação de construtor
	$\mid$	$prim\ atoms$	aplicação de primitiva
	$\mid$	$literal$	inteiro primitivo
$binds$	$::=$	$var_1 = lf_1; \ldots; var_n = lf_n$	$(n \geq 1)$
lf	$::=$	$vars_f \setminus \pi\ vars_a$ -> $expr$	lambda-form
π	$::=$	u $\mid$ n	update-flag
$vars$	$::=$	$\{var_1, \ldots, var_n\}$	$(n \geq 0)$
$atoms$	$::=$	$\{atom_1, \ldots, atom_n\}$	$(n \geq 0)$
$atom$	$::=$	$var \mid literal$	

Lambda-forms

Abstracções-λ são representadas por *lambda-forms*:

$$\underbrace{\{v_1, \ldots, v_m\}}_{\text{variáveis livres}} \backslash\pi \underbrace{\{x_1, \ldots, x_n\}}_{\text{variáveis ligadas}} \;\texttt{->}\; expr \qquad \text{onde } n \geq 0,\ m \geq 0$$

Denotacionalmente:

$$\lambda x_1. \ldots \lambda x_n.\, expr$$

Operacionalmente:

- π indica se deve fazer ou não *update* após redução
- variáveis livre indicam o tamanho da *closure*

Expressões-case

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

- Mecanismo único para decompor valores estruturados
- Restricto a padrões simples (i.e. um nível de construtor)
- Também usado sobre valores primitivos (e.g. inteiros)

Programa

- Programa é uma sequência de definições globais
- Ponto de entrada: expressão *main*

$$
\begin{aligned}
g_1 &= \{\ldots\} \backslash \pi_1 \; \{\ldots\} \; \texttt{->} \; expr_1 \; ; \\
g_2 &= \{\ldots\} \backslash \pi_2 \; \{\ldots\} \; \texttt{->} \; expr_2 \; ; \\
&\vdots \\
g_n &= \{\ldots\} \backslash \pi_n \; \{\ldots\} \; \texttt{->} \; expr_n \; ; \\
\mathrm{main} &= \{\ldots\} \backslash \pi_{n+1} \; \{\} \; \texttt{->} \; expr_{n+1}
\end{aligned}
$$

Exemplo

Código fonte Haskell:

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

Tradução para STG:

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

Transformações necessárias

1. Substituir aplicações binárias por multi-aplicação:

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

2. Saturar construtores e operadores primitivos usando expansão-η:

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

3. Introduzir nomes temporários para argumentos não-atómicos e abstracções-λ:

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

4. Converter os lados direitos de *lets* em *lambda-forms* acrescentando variáveis livres e *updates*

Onde fazer *updates*?

Non-updatable:

1. Abstracções-λ

$$vs \setminus n\ xs\ \text{->}\ expr \qquad (|xs| > 0)$$

2. Aplicações parciais

$$vs \setminus n\ \{\}\ \text{->}\ f\{x_1, \ldots, x_m\} \qquad \ldots$$

3. Construtores

$$vs \setminus n\ \{\}\ \text{->}\ c\{x_1, \ldots, x_m\} \qquad \ldots$$

Updatable: tudo o resto (*thunks*)

$$vs \setminus u\ \{\}\ \text{->}\ expr$$

Mas: análise estática pode fazer melhor (ver bibliografia).

Operações aritméticas

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

- Podemos necessitar de construir *thunks* para inteiros
- A redução de grafos *naive*: representar inteiros como valores na *heap* (*boxed*)
- Torna as operações aritméticas muito ineficientes. . .

Inteiros *boxed* vs. *unboxed*

```
data Int = I# Int#
-- Int : inteiro "boxed"
-- Int# : inteiro "unboxed"

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
```

Representações *boxed* e *unboxed* na STG

- A STG opera directamente apenas sobre valores primitivos (*unboxed*)
- Valores *boxed* são obtidos com construtores algébricos
- Permite expor optimizações como transformações de programa
- Permite expor valores *unboxed* ao programador

Limitações:

- Valores *unboxed* não podem ser usados em funções polimorfas
- Nomes ligados usando *case* em vez de *let*

Exemplo: factorial optimizado

```
{-# LANGUAGE MagicHash #-}
module Main where
import GHC.Exts

-- função "wrapper"
fact :: Int -> Int
fact (I# n) = I# (factAux n)

-- função "worker"
-- usa inteiros "unboxed"
factAux:: Int# -> Int#
factAux n
   = case n of 0# -> 1#
               _ -> n *# factAux (n -# 1#)

main = print (fact 10)
```

1 Linguagem STG

2 Semântica operacional

3 Observações

Semântica operacional

Cada construção sintática tem uma interpretação operacional:

let alocação de *thunks/closures*;

case avaliação e selecção;

aplicação de funções salto incondicional;

aplicação de construtores retorno a uma continuação.

Estado da máquina abstracta

- **Code** próxima ação a executar
- **Argument stack** sequência de valores
- **Return stack** sequência de continuações
- **Update stack** sequência de *update frames*
- **Heap** tabela de endereços para *closures*
- **Globals** endereços de *closures* globais (não mudam)

Valores

Dois tipos:

Addr a	endereço de um objecto na *heap*
Int n	inteiro primitivo

NB: as etiquetas *Addr* e *Int* são apenas para a exposição — não são necessárias durante a execução.

Ambientes

Um ambiente ρ associa valores a variáveis:

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

Usamos σ para o ambiente de globais:

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

Valor de um átomo

Função auxiliar:

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

Heap

A heap associa *endereços* a *closures*:

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

Cada closure é um par com uma *lambda-form* e um ambiente

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

onde $|vs| = |ws|$.

Code

Quatro estados:

Eval e ρ	avaliar e no ambiente ρ;
Enter a	avaliar a *closure* no endereço a;
ReturnCon c ws	retornar um construtor;
ReturnInt k	retornar um inteiro.

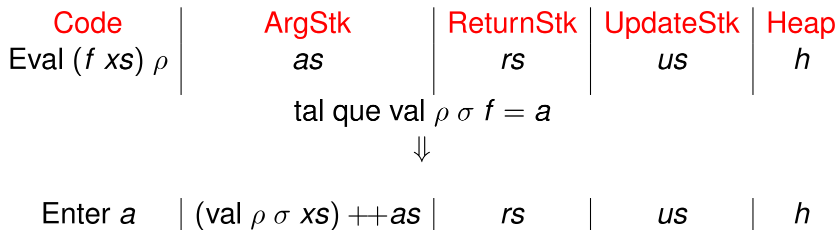
Estado inicial

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

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

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

Aplicação de funções



Enter (non-updatable thunk)

Code	ArgStk	ReturnStk	UpdateStk	Heap
Enter a	as	rs	us	h

tal que $h\ a = (vs \setminus n\ xs \mathbin{-\!>} e,\ ws_f)$

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

$$\Downarrow$$

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

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

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

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

Expressões-let

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

$$\Downarrow$$

$$\text{Eval } e\ \rho' \;\Big|\; as \;\Big|\; rs \;\Big|\; us \;\Big|\; h'$$

onde

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

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

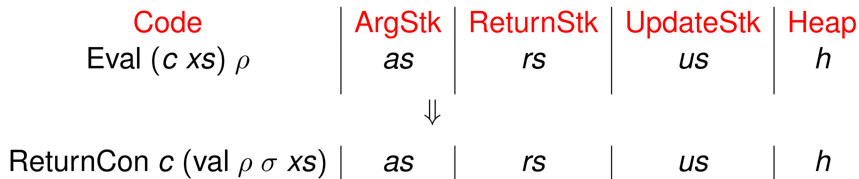
Expressões-letrec

- Análogo ao caso anterior
- ρ' em vez de ρ na definição de de h'
- Ver a bibliografia

Expressões-case

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*

Construtores



Construtores

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

tal que $alts = \{\ldots;\ c\ xs\text{->}e;\ \ldots\}$

$$\Downarrow$$

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

Enter (updatable thunk)

Code	ArgStk	ReturnStk	UpdateStk	Heap
Enter a	as	rs	us	h

tal que $h\ a = (vs \setminus u\ \{\} {-}{>} e,\ ws_f)$

$$\Downarrow$$

Code	ArgStk	ReturnStk	UpdateStk	Heap
Eval $e\ \rho$	$\{\}$	$\{\}$	$(as, rs, a) : us$	h

onde $\rho = [vs \mapsto ws_f]$

Update de um 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

onde vs variáveis arbitrárias

length vs $=$ length ws

$h_u = h[a_u \mapsto (vs \ \backslash\text{n}\ \{\}\ \text{->}\ c\ vs,\ ws)]$

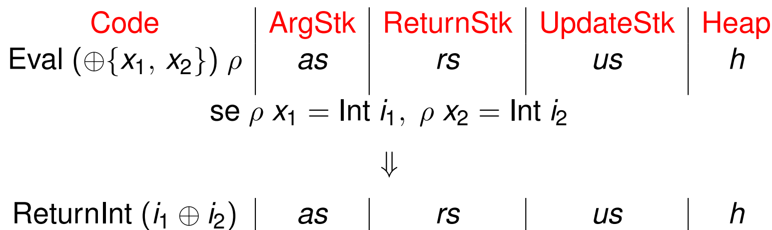
Inteiros (Eval)

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

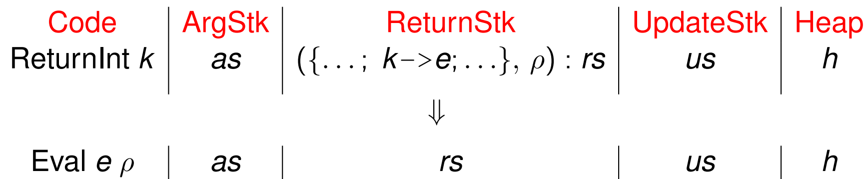
$$\Downarrow$$

ReturnInt k	as	rs	us	h

Operações aritméticas



Inteiros (return)



Inteiros (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

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

$$\Downarrow$$

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

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

Observações

- O modelo de execução de funções da STG original (1992) é "*push-enter*"
- É uma forma elegante para implementar *lazy evaluation*
- A versões mais recentes do GHC usam "*eval-apply*" convencional porque é mais eficiente na prática
- As alterações são descritas no artigo de 2004: *How to make a fast curry*

Mais informação

- Um simulador da STG em Haskell:
  `https://github.com/quchen/stgi`
- *How to make a fast curry*, S. Peyton-Jones, 2004.
  `https://www.microsoft.com/en-us/research/publication/make-fast-curry-pushenter-vs-evalapply`