Operational semantics of the STG Language

Advanced Compiler Construction and Program Analysis

Lecture 15

The topics of this lecture are covered in detail in...

Simon Peyton Jones.

Implementing Lazy Functional Languages on Stock Hardware:

The Spineless Tagless G-machine.

Journal of Functional Programming 1992

5 Operational semantics of the STG language

- 5.1 The initial state
- 5.2 Applications
- 5.3 let(rec) expressions
- 5.4 Case expressions and data constructors
- 5.5 Built in operations
- 5.6 Updating

J. Functional Programming 2 (2):127-202, April 1992 © 1992 Cambridge University Press

127

Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine

SIMON L. PEYTON JONES

Department of Computing Science, University of Glasgow G12 8QQ, UK simonpj@dcs.glasgow.ac.uk

Abstract

The Spineless Tagless G-machine is an abstract machine designed to support non-strict higher-order functional languages. This presentation of the machine falls into three parts. Firstly, we give a general discussion of the design issues involved in implementing non-strict functional languages. Next, we present the STG language, an austere but recognizably-functional language, which as well as a denotational meaning has a well-defined operational semantics. The STG language is the 'abstract machine code' for the Spineless Tagless G-machine. Lastly, we discuss the mapping of the STG language onto stock hardware. The success of an abstract machine model depends largely on how efficient this mapping can be made, though this topic is often relegated to a short section. Instead, we give a detailed discussion of the design issues and the choices we have made. Our principal target is the C language, treating the C compiler as a portable assembler.

The STG Language: syntax

```
binds
                                             X_1 = lf_1 : ... : X_n = lf_n
                                                                                                               bindings
1f
                                                                                                         lambda forms
       \{v_1, ..., v_n\} \setminus \{x_1, ..., x_m\} \rightarrow \exp r
                                                                                                updatable lambda form
                                                                                           non-updatable lambda form
       \{v_1, ..., v_n\} \setminus n \{x_1, ..., x_m\} \rightarrow expr
                                                                                                           expressions
                               ::=
expr
       let binds in expr
                                                                                              non-recursive let-binding
               letrec binds in expr
                                                                                                   recursive let-binding
                                                                                             algebraic case-expression
               case expr of constr<sub>1</sub> \{x_{11}, ..., x_{m1}\} \rightarrow expr_{1}; ...;
default
                                                                                             primitive case-expression
               case expr of literal → expr<sub>1</sub>; ...; default
                                                                                                             application
               x \{atom_1, ..., atom_n\}
                                                                                                  saturated constructor
                                                                                            saturated built-in operator
               constr {atom<sub>1</sub>, ..., atom<sub>n</sub>}
                                                                                                            literal value
               prim {atom<sub>1</sub>, ..., atom<sub>n</sub>}
               literal
                                                                                              default case alternative
default
                                                                                                    variable alternative
               ::=
                                                                                                     default alternative
               x \rightarrow expr
                                                                                                          literal values
               default → expr
literal
               ::= #0 | #1 | ...
                                                                                                   primitive operators
                              ::= +# | -# | *# | /# | ...
prim
                                                                                                                  atoms
                               ::= x | literal
atom
```

The STG Language: syntax characteristics

- 1. All function and constructor arguments are **simple variables or constants**.
- 2. All constructors and primitive operators are **saturated**.

```
x {atom<sub>1</sub>, ..., atom<sub>n</sub>} application constr {atom<sub>1</sub>, ..., atom<sub>n</sub>} saturated constructor prim {atom<sub>1</sub>, ..., atom<sub>n</sub>} saturated built-in operator
```

- 1. Pattern matching is performed **only** by a case-expression.
- 2. Lambda abstraction is special it mentions free variables and updatability:

```
\{v_1, ..., v_n\} \u \{x_1, ..., x_m\} \rightarrow expr updatable lambda form \{v_1, ..., v_n\} \u \{x_1, ..., x_m\} \rightarrow expr non-updatable lambda form
```

3. The STG supports unboxed values.

Operational semantics: state

The state consists of six components:

- 1. The *code* (in one of several forms).
- 2. The *argument stack* as, which contains *values*.
- 3. The *return stack* **rs**, which contains *continuations*.
- 4. The *update stack* us, which contains *update frames*.
- 5. The *heap* h, which contains *closures*.
- 6. The *global environment* σ, which gives the addresses of all top-level closures.

Operational semantics: values

Values are one of the following:

- 1. Addr a a heap address (of a closure)
- 2. Int n a primitive integer value

Every closure is of the following form:

```
(\{v_1, ..., v_n\} \setminus \pi \{x_1, ..., x_m\} \rightarrow expr) \{w_1, ..., w_n\}
```

Operational semantics: code

Code component is in one of the following forms:

- 1. Eval e p evaluate **e** in environment **p** and apply its value to the arguments on the argument stack.
- 2. Enter a apply the closure at address **a** to the arguments on the argument stack.
- 3. ReturnCon c ws return the constructor c applied to values ws to the continuation on the return stack.
- 4. ReturnInt k return primitive integer k to the continuation on the return stack.

Operational semantics: initial state

Code	<pre>Eval (main {}) {}</pre>
Arg stack	{}
Return stack	{}
Update stack	{}
Неар	$a_1 \mapsto (vs_1 \setminus \pi_1 xs_1 \rightarrow e_1) (\sigma vs_1)$ $a_n \mapsto (vs_n \setminus \pi_n xs_n \rightarrow e_n) (\sigma vs_n)$
Globals (σ)	$g_1 \mapsto Addr \ a_1$ $g_n \mapsto Addr \ a_n$

```
g_{1} = VS_{1} \setminus \pi_{1} XS_{1} \rightarrow e_{1}
...
g_{n} = VS_{n} \setminus \pi_{n} XS_{n} \rightarrow e_{n}
```

Operational semantics: applications (1)

Code	Eval (f xs) p
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ



Code	Enter a
Arg stack	(val p σ xs) ++ as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

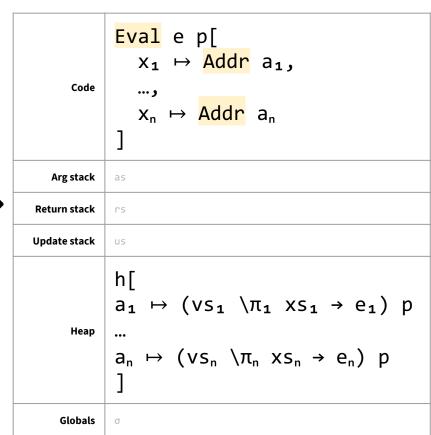
Operational semantics: applications (2)

Code	Enter a
Arg stack	as
Return stack	rs
Update stack	us
Неар	h[a → (vs \ n xs → e) ws]
Globals	σ

Code	Eval e [vs→ws, xs→take (length xs) as]
Arg stack	drop (length xs) as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Operational semantics: let-expressions

Code	Eval (let $x_1 = vs_1 \setminus \pi_1 xs_1 \rightarrow e_1$ $x_n = vs_n \setminus \pi_n xs_n \rightarrow e_n$ in e) p
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ



Operational semantics: case-expressions (1)

Code	Eval (case e of alts) p
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Code	<mark>Eval</mark> e p
Arg stack	as
Return stack	(alts, p) : rs
Update stack	us
Неар	h
Globals	σ

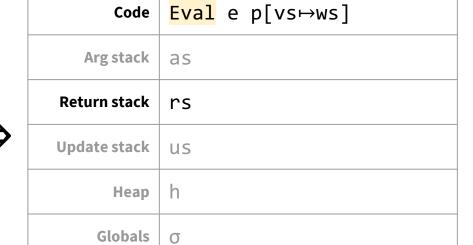
Operational semantics: case-expressions (2)

Code	Eval (c xs) p
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Code	ReturnCon c (val p σ xs)
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

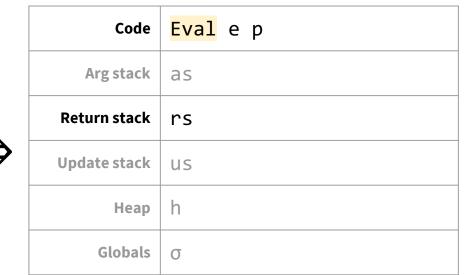
Operational semantics: case-expressions (3)

Code	ReturnCon c ws
Arg stack	as
Return stack	(; c vs → e;, p):rs
Update stack	us
Неар	h
Globals	σ



Operational semantics: case-expressions (4)

Code	ReturnCon c ws
Arg stack	as
Return stack	(; default → e, p):rs
Update stack	us
Неар	h
Globals	σ



Operational semantics: built-in operations (1)

Code	Eval k p
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Code	ReturnInt k
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Operational semantics: built-in operations (2)

Code	Eval (f {}) p[f→Int k]
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Code	<mark>ReturnInt</mark> k
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Operational semantics: built-in operations (2)

Code	ReturnInt k
Arg stack	as
Return stack	(; k → e;, p):rs
Update stack	us
Неар	h
Globals	σ

Code	<mark>Eval</mark> e p
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Operational semantics: built-in operations (3)

Code	ReturnInt k
Arg stack	as
Return stack	(; default → e, p):rs
Update stack	us
Неар	h
Globals	σ

Code	<mark>Eval</mark> e p
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Operational semantics: built-in operations (4)

Code	Eval (+# $\{x_1, x_2\}$) p[$x_1 \mapsto Int k_1$, $x_2 \mapsto Int k_2$]
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Code	ReturnInt (k ₁ +# k ₂)
Arg stack	as
Return stack	rs
Update stack	us
Неар	h
Globals	σ

Operational semantics: updating

Updates happen in two stages:

- 1. When an updatable closure is entered, it pushes an *update frame* (as, rs, a) onto the update stack, and makes the argument and return stacks empty.
- 2. When evaluation of a closure is complete an update is triggered:
 - a. If a value of the closure is a data constructor or literal, it will try to pop a continuation of the return stack, but will fail (it is empty).
 - b. If a value of the closure is a function, it will try to pop arguments off of the argument stack, but will fail (it is empty).

Operational semantics: updating (1)

Code	Enter a		
Arg stack	as		
Return stack	rs	\rightarrow	R
Update stack	us		U
Неар	$h[a \mapsto (vs \setminus u \{\} \rightarrow e) ws]$		
Globals	σ		

Code	<mark>Eval</mark> e [vs→ws]
Arg stack	{}
Return stack	{}
Update stack	(as,rs,a):us
Неар	h
Globals	σ

Operational semantics: updating (2)

Code	ReturnCon c ws			
Arg stack	{}			
Return stack	{}			
Update stack	(as,rs,a):us			
Неар	h			
Globals	σ			



Code	ReturnCon c ws			
Arg stack	as			
Return stack	rs			
Update stack	us			
Неар	$h[a \mapsto (vs \setminus u \{} \rightarrow c vs) ws]$			
Globals	σ			

Operational semantics: updating (3)

Code	Enter f		Code	Enter f
Arg stack	as		Arg stack	as ++ as _u
Return stack	{} =		Return stack	rs
Update stack	(asu,rsu,au):us		Update stack	us
Неар	$h[f \mapsto (vs \setminus n \ xs \rightarrow e) \ ws]$		Неар	$h[a_u \mapsto ((vs++xs_1) \setminus n xs_2 \rightarrow e) ws]$
Globals	σ		Globals	σ

length(as) < length(xs)</pre>

Operational semantics: updating (3')

Code	Enter f		Code	Enter f
Arg stack	as		Arg stack	as ++ as _u
Return stack	{ }		Return stack	rs
Update stack	(asu,rsu,au):us		Update stack	us
Неар	h[f ↦ (vs \n xs → e) ws]		Неар	$h[a_u \mapsto ((g:xs_1) \setminus n \{\} \rightarrow g xs_1) (a:as)]$
Globals	σ		Globals	

length(as) < length(xs)</pre>

length(xs_1) = as xs_1++xs_2 = xs

Summary

- ☐ Operational semantics of the STG language
 - > Applications
 - ➤ Let-expressions
 - ➤ Case-expressions
 - ➤ Built-in operations
 - ➤ Updating closures

See you next time!