Operational semantics of the STG Language

Advanced Compiler Construction and Program Analysis

Lecture 15

Innopolis University, Spring 2022

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

J. Functional Programming **2** (2):127–202, April 1992

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

<table>
<tr><td>binds ::= x_1=lf$_1$; … ; x_n=lf$_n$</td><td>bindings</td></tr>
<tr><td>lf ::=</td><td>lambda forms</td></tr>
<tr><td>{v_1, …, v_n} \u {x_1, …, x_m} → expr</td><td>updatable lambda form</td></tr>
<tr><td>{v_1, …, v_n} \n {x_1, …, x_m} → expr</td><td>non-updatable lambda form</td></tr>
<tr><td>expr ::=</td><td>expressions</td></tr>
<tr><td>let binds in expr</td><td>non-recursive let-binding</td></tr>
<tr><td>letrec binds in expr</td><td>recursive let-binding</td></tr>
<tr><td>case expr of constr$_1$ {x_{11}, …, x_{m1}} → expr$_1$; …; default</td><td>algebraic case-expression</td></tr>
<tr><td>case expr of literal → expr$_1$; …; default</td><td>primitive case-expression</td></tr>
<tr><td>x {atom$_1$, …, atom$_n$}</td><td>application</td></tr>
<tr><td>constr {atom$_1$, …, atom$_n$}</td><td>saturated constructor</td></tr>
<tr><td>prim {atom$_1$, …, atom$_n$}</td><td>saturated built-in operator</td></tr>
<tr><td>literal</td><td>literal value</td></tr>
<tr><td>default ::=</td><td>default case alternative</td></tr>
<tr><td>x → expr</td><td>variable alternative</td></tr>
<tr><td>default → expr</td><td>default alternative</td></tr>
<tr><td>literal ::= #0 | #1 | …</td><td>literal values</td></tr>
<tr><td>prim ::= +# | -# | *# | /# | …</td><td>primitive operators</td></tr>
<tr><td>atom ::= x | literal</td><td>atoms</td></tr>
</table>

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 {atom₁, …, atomₙ}              application
constr {atom₁, …, atomₙ}         saturated constructor
prim {atom₁, …, atomₙ}           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:

```
{v₁, …, vₙ} \u {x₁, …, xₘ} → expr        updatable lambda form
{v₁, …, vₙ} \n {x₁, …, xₘ} → 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:

$$(\{v_1, \ldots, v_n\} \backslash\pi \ \{x_1, \ldots, x_m\} \to \texttt{expr})\ \{w_1, \ldots, w_n\}$$

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	`Eval (main {}) {}`
Arg stack	`{}`
Return stack	`{}`
Update stack	`{}`
Heap	$a_1 \mapsto (vs_1\ \backslash\pi_1\ xs_1 \rightarrow e_1)\ (\sigma\ vs_1)$ … $a_n \mapsto (vs_n\ \backslash\pi_n\ xs_n \rightarrow e_n)\ (\sigma\ vs_n)$
Globals (σ)	$g_1 \mapsto \text{Addr}\ a_1$ … $g_n \mapsto \text{Addr}\ a_n$

$g_1 = vs_1\ \backslash\pi_1\ xs_1 \rightarrow e_1$

…

$g_n = vs_n\ \backslash\pi_n\ xs_n \rightarrow e_n$

Operational semantics: applications (1)

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

⇒

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

Operational semantics: applications (2)

Code	`Enter a`
Arg stack	`as`
Return stack	`rs`
Update stack	`us`
Heap	`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`
Heap	`h`
Globals	σ

Operational semantics: let-expressions

Code	`Eval (`
	`let x₁ = vs₁ \π₁ xs₁ → e₁`
	`    …`
	`    xₙ = vsₙ \πₙ xsₙ → eₙ`
	`in e) p`

Arg stack	as
Return stack	rs
Update stack	us
Heap	h
Globals	σ

$\Rightarrow$

Code	`Eval e p[`
	`  x₁ ↦ Addr a₁,`
	`  …,`
	`  xₙ ↦ Addr aₙ`
	`]`

Arg stack	as
Return stack	rs
Update stack	us

Heap	`h[`
	`a₁ ↦ (vs₁ \π₁ xs₁ → e₁) p`
	`…`
	`aₙ ↦ (vsₙ \πₙ xsₙ → eₙ) p`
	`]`

Globals	σ

Operational semantics: case-expressions (1)

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

⇒

Code	`Eval e p`
Arg stack	`as`
Return stack	`(alts, p) : rs`
Update stack	`us`
Heap	`h`
Globals	σ

Operational semantics: case-expressions (2)

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

⇒

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

$\Rightarrow$

Code	`Eval e p[vs↦ws]`
Arg stack	`as`
Return stack	`rs`
Update stack	`us`
Heap	`h`
Globals	`σ`

Operational semantics: case-expressions (4)

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

⇒

Code	`Eval e p`
Arg stack	`as`
Return stack	`rs`
Update stack	`us`
Heap	`h`
Globals	`σ`

Operational semantics: built-in operations (1)

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

⇒

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

$\Rightarrow$

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

Operational semantics: built-in operations (2)

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

$\Rightarrow$

Code	`Eval e p`
Arg stack	`as`
Return stack	`rs`
Update stack	`us`
Heap	`h`
Globals	`σ`

Operational semantics: built-in operations (3)

Code	`ReturnInt k`	⇒	**Code**	`Eval e p`
Arg stack	`as`		Arg stack	`as`
Return stack	`( …; default → e, p):rs`		**Return stack**	`rs`
Update stack	`us`		Update stack	`us`
Heap	`h`		Heap	`h`
Globals	`σ`		Globals	`σ`

Operational semantics: built-in operations (4)

Code	`Eval (+# {x₁,x₂}) p[ x₁ ↦ Int k₁, x₂ ↦ Int k₂ ]`
Arg stack	as
Return stack	rs
Update stack	us
Heap	h
Globals	σ

$\Rightarrow$

Code	`ReturnInt (k₁ +# k₂)`
Arg stack	as
Return stack	rs
Update stack	us
Heap	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`
Update stack	`us`
Heap	`h[a ↦ (vs \u {} → e) ws]`
Globals	`σ`

⇒

Code	`Eval e [vs↦ws]`
Arg stack	`{}`
Return stack	`{}`
Update stack	`(as,rs,a):us`
Heap	`h`
Globals	`σ`

Operational semantics: updating (2)

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

⇒

Code	`ReturnCon c ws`
Arg stack	`as`
Return stack	`rs`
Update stack	`us`
Heap	`h[a ↦ (vs \u {} → c vs) ws]`
Globals	σ

Operational semantics: updating (3)

Code	`Enter f`
Arg stack	`as`
Return stack	`{}`
Update stack	$(as_u, rs_u, a_u) : us$
Heap	$h[f \mapsto (vs \setminus \mathbf{n}\ xs \to e)\ ws]$
Globals	σ

$\text{length}(as) < \text{length}(xs)$

$\Rightarrow$

Code	`Enter f`
Arg stack	$as \mathbin{++} as_u$
Return stack	`rs`
Update stack	`us`
Heap	$h[a_u \mapsto ((vs \mathbin{++} xs_1) \setminus \mathbf{n}\ xs_2 \to e)\ ws]$
Globals	σ

$\text{length}(xs_1) = as$

$xs_1 \mathbin{++} xs_2 = xs$

Operational semantics: updating (3')

Code	Enter f
Arg stack	as
Return stack	{}
Update stack	$(as_u, rs_u, a_u):us$
Heap	$h[f \mapsto (vs\ \backslash\mathbf{n}\ xs \to e)\ ws]$
Globals	σ

$\text{length}(as) < \text{length}(xs)$

$\Rightarrow$

Code	Enter f
Arg stack	$as \mathbin{+\!+} as_u$
Return stack	rs
Update stack	us
Heap	$h[a_u \mapsto ((g:xs_1)\ \backslash\mathbf{n}\ \{\} \to g\ xs_1)\ (a:as)]$
Globals	σ

$\text{length}(xs_1) = as$

$xs_1 \mathbin{+\!+} xs_2 = xs$

Summary

- ❑ Operational semantics of the STG language
  - ➢ Applications
  - ➢ Let-expressions
  - ➢ Case-expressions
  - ➢ Built-in operations
  - ➢ Updating closures

See you next time!