The STG Language

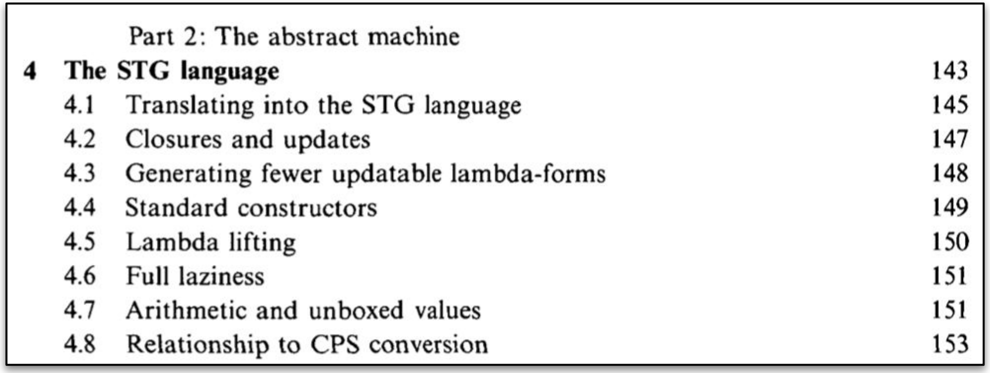
Advanced Compiler Construction and Program Analysis

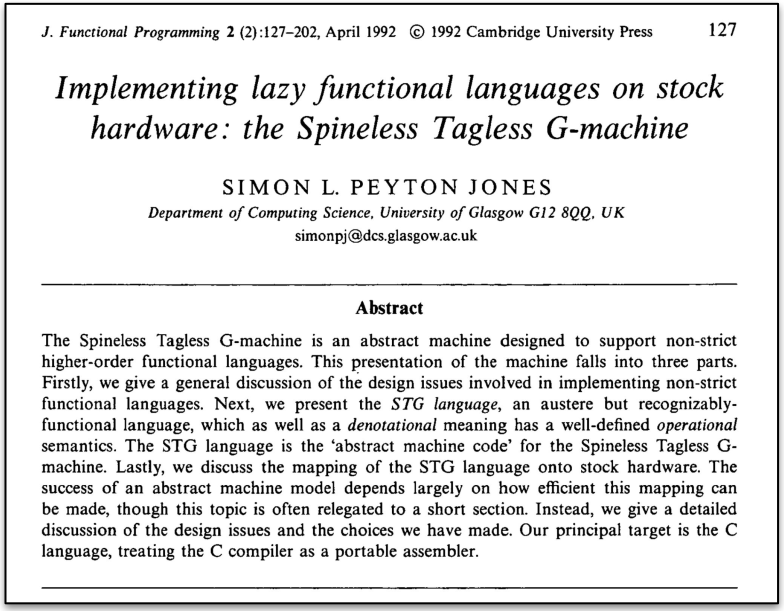
Lecture 14

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 © 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      ::=  x₁=lf₁ ; … ; x□=lf□                                              bindings
lf         ::=                                                               lambda forms
      {v₁, …, v□} \u {x₁, …, x□} → expr                               updatable lambda form
      {v₁, …, v□} \n {x₁, …, x□} → expr                           non-updatable lambda form

expr       ::=                                                                expressions
      let binds in expr                                           non-recursive let-binding
      letrec binds in expr                                            recursive let-binding
      case expr of constr₁ {x₁₁, …, x□₁} → expr₁; …; default     algebraic case-expression
      case expr of literal → expr₁; …; default                   primitive case-expression
      x {atom₁, …, atom□}                                                      application
      constr {atom₁, …, atom□}                                       saturated constructor
      prim {atom₁, …, atom□}                                     saturated built-in operator
      literal                                                                literal value

default    ::=                                                     default case alternative
      x → expr                                                        variable alternative
      default → expr                                                   default alternative
literal    ::= #0 | #1 | …                                                  literal values
prim       ::= +# | -# | *# | /# | …                                   primitive operators
atom       ::= x | literal                                                           atoms
```

The STG Language: syntax vs semantics

Syntactic construction	**Operational reading**
Function application	Tail call
Let expression	Heap allocation
Case expression	Evaluation
Constructor application	Return to continuation

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 {`$atom_1$`, …, `$atom_{□}$`}` *application*

`constr {`$atom_1$`, …, `$atom_{□}$`}` *saturated constructor*

`prim {`$atom_1$`, …, `$atom_{□}$`}` *saturated built-in operator*

3. Pattern matching is performed **only** by a `case`-expression.
4. Lambda abstraction is special — it mentions free variables and updatability:

`{`v_1`, …, `$v_{□}$`}` **`\u`** `{`x_1`, …, `$x_{□}$`} → expr` *updatable lambda form*

`{`v_1`, …, `$v_{□}$`}` **`\n`** `{`x_1`, …, `$x_{□}$`} → expr` *non-updatable lambda form*

5. The STG supports unboxed values.

Translating into the STG language: example (1)

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

```
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}
```

Translating into the STG language: example (2)

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

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

Translating into the STG language: in general

1. Replace nested binary applications with multiple application:

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

2. Saturate all constructors and built-in operators:

   $$\mathtt{constr}\ e_1\ e_2\ \ldots\ e_n \quad \Rightarrow \quad \lambda x_1.\ \ldots\ \lambda x_n.\ \mathtt{constr}\ e_1\ e_2\ \ldots\ e_n\ x_1\ \ldots\ x_n$$

3. Name every non-atomic argument and every lambda abstraction, using **`let`**.

4. Convert right hand side of let into a lambda form,
   by adding free variables and update information.

Translating into the STG language: free variables

A variable is considered free in a given lambda-form if

1. It is **mentioned** in the body of the lambda abstraction
2. It is **not bound** by the lambda
3. It is **not bound** by the top-level bindings of the program

This specifies variables that **must** appear in the free variables list.
However, any (redundant) in-scope variable **may** appear as well.

Closures and updates

Which lambda forms can safely be made non-updatable, without losing the single-evaluation property?

1. **Manifest functions** are lambda forms with a non-empty argument list.
2. **Partial applications** are lambda forms of the following form:

   $$\{v_1,\ \ldots,\ v_n\}\ \backslash n\ \{\} \rightarrow f\ \{x_1,\ \ldots,\ x_m\}$$

   where **f** is known to be a manifest function with more than **m** arguments.

3. **Constructors** are lambda forms of the following form:

   $$\{v_1,\ \ldots,\ v_n\}\ \backslash n\ \{\} \rightarrow \mathtt{constr}\ \{x_1,\ \ldots,\ x_m\}$$

4. **Thunks** are any other lambda forms.

Updates are *never required* for manifest functions, partial applications, and constructors.
Updates can *sometimes* be omitted for thunks.

Translating into the STG language: exercise

Exercise 14.1.

Translate the following Haskell program into the STG language.

```
sum :: [Int] → Int
sum [] = 0
sum (x:xs) = x + sum xs
```

Standard constructors

```
let ys = y₁ : (y₂ : (y₃ : [])) in …

let ys₃ = {} \n {} → Nil {} in
let ys₂ = {y₃, ys₃} \n {} → Cons {y₃, ys₃} in
let ys₁ = {y₂, ys₂} \n {} → Cons {y₂, ys₂} in
let ys  = {y₁, ys₁} \n {} → Cons {y₁, ys₁}
in …
```

Same shape,
can share the code pointer!

Standard constructors: top-level definitions

```
ys = [thing]
thing = …
```

```
nil = {} \n {} → Nil {}
ys = {} \n {} → Cons {thing, nil}
thing = …
```

Standard constructors: top-level definitions

```
ys = [thing]
thing = …
```

For nullary constructors, we can share the entire closure

```
nil = {} \n {} → Nil {}
ys = {thing, nil} \n {} → Cons {thing, nil}
thing = …
```

Adding top-level definitions to the free variables, so that we can reuse the same code pointer

Arithmetic and unboxed values

```
data Int = MkInt Int#


plus :: Int → Int → Int
plus e1 e2 =
  case e1 of
    MkInt x# →
      case e2 of
        MkInt y# →
          case (x# +# y#) of
            t# → MkInt t#
```

Arithmetic and unboxed values

1. Data type are divided into two kinds: ***algebraic*** data types are introduced explicitly by the user with data keyword, while ***primitive*** data types are built into the system.
2. All literal constants are of primitive types. Literals of algebraic types are given by using appropriate constructors.
3. All built-in operations operate over primitive values.
4. Values of unboxed type do not have to be of the same size as pointer. As a result, polymorphic functions can take only arguments of boxed type.
5. A **`let`** or **`letrec`** cannot bind a variable of unboxed type. Such binding instead happens instead using a primitive case expression.
6. There are two forms of **`case`** expressions (algebraic and primitive).

Summary

- ❏ The STG language

See you next time!