Compiling lazy functional languages

Advanced Compiler Construction and Program Analysis

Lecture 13

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

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.

Challenges of compiling functional languages

1. Interpretation of functional languages is not very difficult, especially if no efficiency is expected.

2. Compiling ***functional*** languages is harder, since computational model of the underlying hardware is different:
    1. Have to represent compound data (especially ***variants***)
    2. Have to represent (partially applied) functions (closures)

3. Compiling ***non-strict*** functional languages is even more challenging, since the arguments can be passed to functions as ***unevaluated expressions***, which again requires a proper representation

Approaches

1. Many approaches rely on some ***abstract machine***:
    1. A functional program is translated to a sequence of instructions.
    2. Each instruction has “imperative” operational semantics using State Transition Systems.

2. Some implementations, especially, those that rely on ***graph reduction***, are very different from conventional compiler technology. The great difference even sparked an interest in developing new hardware architectures.

3. ***Spineless tagless G-machine*** (STG machine) has its roots in lazy graph reduction, but attempts to bridge the gap with conventional compiler technology.

Exploring the design space

We are focusing now on non-strict higher-order functional languages (think Haskell).

To navigate the design space, we ask the following questions:

1. How do we represent the following?
    1. function values
    2. data values (in particular *variants*)
    3. unevaluated expressions (a.k.a. *thunks*)

2. How is function application performed?

3. How is case analysis performed on data structures?

Closures: the heap objects

We store on the heap two kinds of objects (we will call both of them ***closures***):

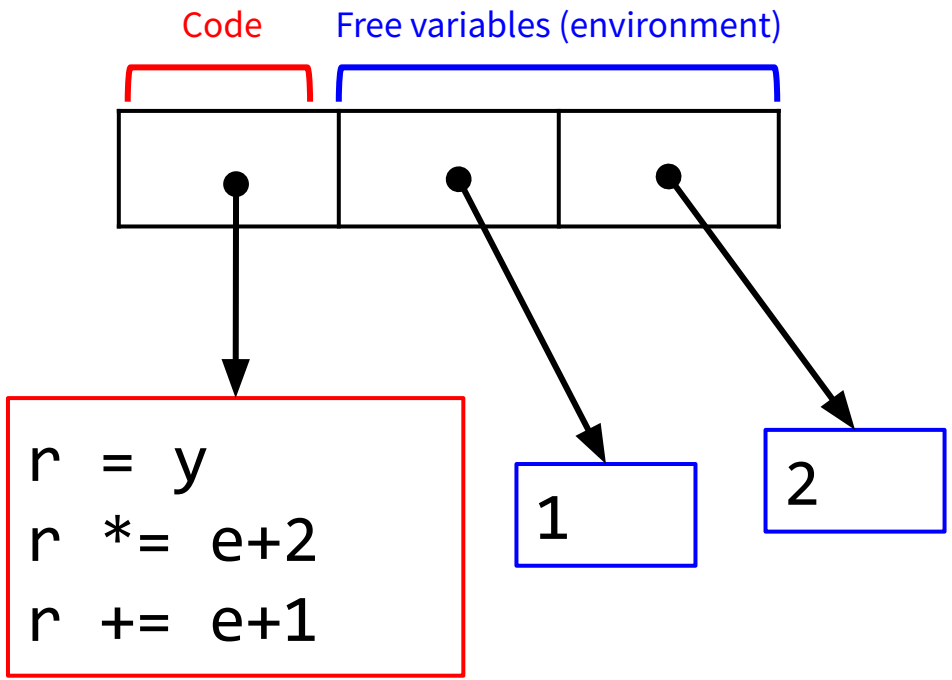
1. ***values*** (or *head normal forms*)
    1. function values
    2. data values
2. ***unevaluated expressions*** (or *thunks*)

Note that a value is allowed to contain thunks inside of it:

`{ 1, 2 + 3 }` — a tuple value with an unevaluated expression in its second component

Closures: representing functions

```
x = 1
z = 2
function f(y) {
    return (x + y*z)
}
```



Carrying free variables with **every** function may be expensive, so:

1. We can ***share blocks***, saving storage, but slowing down access.
2. We can use ***heap-allocated frames***, but in non-strict languages they have to be carefully updated to avoid duplication of computation.

Closures: thunk representation

Thunk is an unevaluated expression. When the value of thunk is required, it is ***forced***.

Lazy implementation physically *updates* the thunk when its forced, so that next time its value can be returned immediately.

The naive reduction model. Update the thunk after every reduction step. This can lead to a thunk being updated with another thunk. This is not particularly efficient.

The cell model. Each closure is provided a *status flag*. The code to force the closure checks the flag. If it is evaluated — extract value. Otherwise, enter the closure, compute value, then update the thunk with the value and flip the flag.

The self-updating model. The suspended computation code has the closure-update code embedded into it. The update overwrites the thunk with a value and also the pointer to the code (it now simply does nothing, returning the value).

Closures: uniform representation

With the self-updating model, every heap-allocated object is represented ***uniformly***:

1. Code pointer
2. Zero or more fields which give values to free variables

There are several things to note about this:

1. When thunk is updated with a large value, it is updated with the ***indirection code***.
2. When a thunk is ***entered***, its code pointer can be overwritten with a ***"black-hole"*** code pointer. This allows to detect infinite loops at runtime and produce a nicer error message, compared to stack overflow!
3. In concurrent systems, a similar mechanism could be employed to synchronize threads.

Functions: compiling function application

In strict languages (**eval-apply model**):

1. Evaluate the function
2. Evaluate the argument (a list of arguments)
3. Apply the function value to the argument

In some non-strict languages (**eval-apply model**):

1. Evaluate the function
2. Apply the function value to the argument

In compilers based on lazy graph reduction (**push-enter model**):

1. Push the argument on an evaluation stack
2. Enter the function (tail-call)

Data structures

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

```
Tree[A] = <leaf : A ; branch : {Tree[A], Tree[A]}>
```

```
case t of
   <leaf   = n>          ⇒ e1
 | <branch = {t1, t2}>   ⇒ e2
```

Summary

- ❏ Challenges of compiling lazy functional languages
- ❏ Design space for closure representation

See you next time!