# Spineless Tagless G-machine

Hannes Mehnert
mehnert@cs.tu-berlin.de

TU Berlin

21.01.2005

## Terms

- Spineless: 'does not need to build the spine of the expression being reduced'
- Tagless: no tag bits (used to distinguish between different types)

## History

- G-machine: 1987 (Augustsson, Johnsson)
- Spineless G-machine: 1988 (Burn, Peyton Jones and Robson)
- Spineless tagless G-machine:1992 (Peyton Jones)

## Overview of the STG language

- Purely-functional language
- All arguments are simple variables or constants
- All constructors and built-in operations are saturated
- Pattern matching is performed only by case expressions
- Bindings contain free variables, update flag and lamda-form, no lambda lifting needed
- Supports unboxed values

## Closures and updates

- It is safe to set the update flag to u of every lambda-form
- Updates are never required for functions, partial applications and constructors

## Lambda lifting

- All function definitions are lifted to the top level
- Their free variables become extra arguments
- Each lambda-form has no free variables or no arguments
- Local environment of the STG machine consists of two parts
    - Values in the closure just entered (its free variables)
    - Values on the stack (its arguments)
- Reduces the movement of values from the heap to the stack

## Arithmetic and unboxed values

- Variables are bound to unevaluated heap-allocated closure
- Unboxed value is the actual value (result of the evaluated closure)
- Boxed representation makes arithmetic expensive
- In the STG language, functions may take unboxed values as arguments and return them as results

## Components of the state

- Code, which is one of the following:
    - Eval e p
    - Enter a
    - ReturnCon c ws
    - ReturnInt k
- Argument stack, which contains values
- Return stack, which contains continuations
- Update stack, which contains update frames
- Heap, which contains closures
- Global environment, which gives the addresses of all closures defined at top level

## Initial state

- Code: Eval(main {}) {}
- Argument stack: empty
- Return stack: empty
- Update stack: empty
- Heap: contains a closure for each global
- Global environment: binds each global to its closure

## Let(rec) expressions

- A let and letrec expression constructs one or more closures in the heap

## Case expressions

- “case e of alts”
- Push a continuation onto return stack and evaluate e
- Continuation is a pair(alts, p)
- Alternative alts is the code which is evaluated when e finished
- Environment p is the context in which to evaluate the alternative

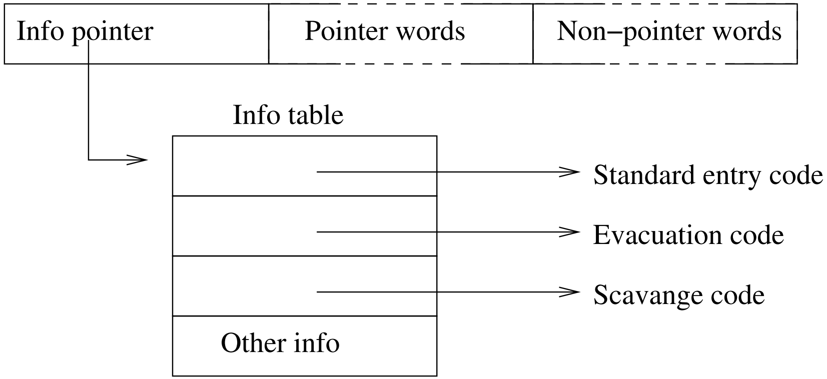
## Updating

- An update frame is pushed onto the update stack when an updatable closure is entered
  - Previous argument stack
  - Previous return stack
  - Pointer to the closure being entered
- After evaluation of a closure is complete, an update is triggered
  - Value is a data constructor or literal, pop return stack fail
  - Value is a function, bind argument(s) fail

## Heap

- The heap is a collection of closures
- Each closure is variable size and identified by a unique address
- A pointer is the address of a closure

## Layout of a closure



## Two-space stop-and-copy garbage collection

- Memory is divided into two spaces
- Each live closure must be evacuated from from-space to to-space
- To-space is scanned linearly, each closure must be scavenged

## Two-space stop-and-copy garbage collection

- Memory is divided into two spaces
- Each live closure must be evacuated from from-space to to-space
- To-space is scanned linearly, each closure must be scavenged
- Evacuation:
    - Copy closure into to-space
    - Overwrite closure in from-space with a pointer pointing to to-space
    - Return to-space address

## Two-space stop-and-copy garbage collection

- Memory is divided into two spaces
- Each live closure must be evacuated from from-space to to-space
- To-space is scanned linearly, each closure must be scavenged
- Evacuation:
  - Copy closure into to-space
  - Overwrite closure in from-space with a pointer pointing to to-space
  - Return to-space address
- Scavenging:
  - Call evacuation code
  - Replace pointer with returned to-space pointer

## Abstract machine contains three stacks

- Argument stack (closure addresses and primitive values)
- Return stack (continuations for case expressions)
- Update stack (update frames)

## One stack

- Would be possible
- Garbage collector must use all pointers in the stack as roots
- Garbage collector would need to know whether each frame is
  - Closure address
  - Code address
  - Primitive value
- Could be solved by a tag-bit, but arithmetic would be much slower

## Two stacks

- A-stack for pointers
- B-stack for non-pointers
- Nomenclature from ABC machine
  - A = argument
  - B = basic value
- Stack pointers are in special registers SpA and SpB
- Grow towards each other

## Target language

- C is used as high-level assembler to gain portability
- Argument stacks and control stack are mapped onto C arrays, bypassing usual C parameter-passing
- All "registers" are global variables
- Compiling jumps
    - Giant switch
        - Adds a layer of indirection
        - Entire program has to be gathered in a single giant C procedure and then be compiled
    - Tiny interpreter
        - Each labelled block of code is compiled to a parameter-less C function whose name is the required label
        - while (TRUE) { cont = (*cont)(); }

## Initial state

- Evaluates main
- Heap contains a closure for each global variable
- Each of these closures can be referred directly by its C label
- Linker implements the global environment

## Let(rec) expressions

- Always compile to code which allocates a closure in the heap for each definition
- Followed by code to evaluate the body
- Standard-entry code for a closure:
    - Argument satisfaction check
    - Stack overflow check
    - Heap overflow check
    - Info pointer update
    - Update frame construction

## Case expressions

- Save local environment (all live variables)
  - Already in stack, nothing to be done
  - Register or offset of heap pointer, saved to the appropriate stack
  - Closure pointed to by Node, save variable or Node
- Select alternative
  - Push pointer to return vector on stack B
  - C switch on register RTag
- Return constructor arguments
  - Node register point to closure with values
  - Return arguments in registers

## Updating

- Push update frame on B stack
- Stack base registers point to top of stacks
- Partial application
- Constructors
- Vectored returns
- Return values in registers

## References

- Used by the Glasgow Haskell Compiler
  http://www.haskell.org/ghc/
- Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine
  http://uebb.cs.tu-berlin.de/lehre/2004WScompilerbau/pap
- The Spineless G-Machine
  http://portal.acm.org/citation.cfm?id=62717