Implementing lazy functional languages on stock hardware:

the Spineless Tagless G-Machine

Joe Jevnik

July 11, 2017

Papers We Love - Boston

Background

Existing abstract machines. . .

Background

Existing abstract machines. . .

are too abstract

Background

Existing abstract machines. . .

- are too abstract
- don't say enough about data structure traversal

Background

Existing abstract machines. . .

- are too abstract
- don't say enough about data structure traversal
- leave unboxed values to the code generator

Overview

1. the design space

Overview

1. the design space
2. the abstract machine (the STG language)

Overview

1. the design space
2. the abstract machine (the STG language)
3. mapping the abstract machine onto real hardware

Design Space

What is a Functional Language?

Definition (Functional Language)

A programming language which treats computation as the evaluation of mathematical functions.

What is a Functional Language?

Definition (Functional Language)

A programming language which treats computation as the evaluation of mathematical functions.

Examples

- SML
- OCaml
- Scala
- Haskell*

Features of Functional Languages

Higher order functions

```
plusOne :: Int -> Int
plusOne x = x + 1

xs :: [Int]
xs = [1, 2, 3]

xsPlusOne :: [Int]
xsPlusOne = map plusOne xs
```

Features of Functional Languages

Algebraic data types

```
data Maybe a = Nothing | Just a
data List a = Nil | Cons a (List a)
data Tuple3 a b c = MkTuple3 a b c
data Tree a = Leaf a | Branch (Tree a) (Tree a)

safeHead :: [a] -> Maybe a
safeHead = case xs of
    Nil              -> Nothing
    Cons head tail -> Just head
```

Features of Functional Languages

Algebraic data types

```
data Bool = False | True

conditional :: Bool -> a -> a
conditional p whenFalse whenTrue = case p of
    False -> whenFalse
    True  -> whenTrue
```

Features of Functional Languages

Immutable data

```
insort value [] = [value]
insort value (x:xs)
    | value < x = value : x : xs
    | otherwise = x : insort value xs
```

Features of Functional Languages

Immutable data

```
insort value [] = [value]
insort value (x:xs)
    | value < x = value : x : xs
    | otherwise = x : insort value xs
```

```
> let xs = [1,2,4,5]
> insort 3 xs
[1,2,3,4,5]
```

Features of Functional Languages

Immutable data

```
insort value [] = [value]
insort value (x:xs)
    | value < x = value : x : xs
    | otherwise = x : insort value xs
```

```
> let xs = [1,2,4,5]
> insort 3 xs
[1,2,3,4,5]
> xs
[1,2,4,5]
```

Features of Functional Languages

Currying

```
> let f x y = x
```

Features of Functional Languages

Currying

```
> let f x y = x

> :t f
f :: a -> b -> a
```

Features of Functional Languages

Currying

```
> let f x y = x

> :t f
f :: a -> b -> a

f :: a -> (b -> a)
```

Features of Functional Languages

Currying

```
> let f x y = x

> :t f
f :: a -> b -> a

f :: a -> (b -> a)

> f 1 2
1
```

Features of Functional Languages

Currying

```
> let f x y = x

> :t f
f :: a -> b -> a

f :: a -> (b -> a)

> f 1 2
1

> (f 1) 2
1
```

Features of Functional Languages

Currying

```
> let f x y = x

> :t f
f :: a -> b -> a

f :: a -> (b -> a)

> f 1 2
1

> (f 1) 2
1

> :t f 1
f 1 :: Num a => b -> a
```

Features of Functional Languages

Currying

```
- fun f (x, y) = x;
val f = fn : 'a * 'b -> 'a
```

Features of Functional Languages

Currying

```
- fun f (x, y) = x;
val f = fn : 'a * 'b -> 'a

- f (1, 2);
val it = 1 : int
```

Features of Functional Languages

Currying

```
- fun f (x, y) = x;
val f = fn : 'a * 'b -> 'a

- f (1, 2);
val it = 1 : int

- f 1
stdIn:3.1-3.4 Error: operator and operand don't agree [
  operator domain: 'Z * 'Y
  operand:         [int ty]
  in expression:
    f 1
```

What is Lazy Evaluation?

Definition (Lazy Evaluation)

An evaluation strategy where expressions are evaluated only when their result is needed. By default, all expressions result in a deferred computation.

Lazy Evaluation is also known as “call by need”.

What is Lazy Evaluation?

Definition (Lazy Evaluation)

An evaluation strategy where expressions are evaluated only when their result is needed. By default, all expressions result in a deferred computation.

Lazy Evaluation is also known as "call by need".

Examples

- Python generators
- Scala `lazyval`
- C++ expression templates

What is Lazy Evaluation?

```
> let x = 1 : x
```

What is Lazy Evaluation?

```
> let x = 1 : x

> x
[1,1,1,1,1,1,1,1,1,1,...
```

What is Lazy Evaluation?

```
> let x = 1 : x

> x
[1,1,1,1,1,1,1,1,1,1,...

> let fibs = scanl (+) 1 $ 0 : fibs
```

What is Lazy Evaluation?

```
> let x = 1 : x

> x
[1,1,1,1,1,1,1,1,1,1,...

> let fibs = scanl (+) 1 $ 0 : fibs

> fibs
[1,1,2,3,5,8,13,21,34,55,...
```

What is Lazy Evaluation?

```
> let xs = [0, undefined, undefined]
```

What is Lazy Evaluation?

```
> let xs = [0, undefined, undefined]

> head $ map (+ 1) xs
1
```

Key Questions

- How are function values, data values, and unevaluated expressions represented?

Key Questions

- How are function values, data values, and unevaluated expressions represented?
- How is function application performed?

Key Questions

- How are function values, data values, and unevaluated expressions represented?
- How is function application performed?
- How is case analysis performed on data structures?

Representing Values

Representing Values

Types of Values

- Evaluated objects
- Currently unevaluated expressions

Representing Values

Function values and data values look the same

```
compose f g x = f (g x)

gxIsFunction :: Num a => a -> a
gxIsFunction = compose ($ 1) (+)

gxIsNumber :: Num a => a -> a
gxIsNumber = compose (+ 1) (* 2)
```

Closures

Definition (Closure)

A function paired with some bound arguments.

Definition (Free Variable)

A bound argument to a **closure**.

Representing Closures

```
typedef struct {
    void* (*code)(void** free_variables,
                  void** arguments);
    void* free_variables[];
} closure;
```

Thunks

Definition (Thunk)

A **closure** that represents a value which is currently unevaluated.

Thunks have no arguments, just **free variables**.

Cell Model Thunks

```
typedef struct {
    void* (*code)(void** free_variables);
    void* value;
    void* free_variables[];
} cell_model_thunk;
```

Cell Model Thunks

```
typedef struct {
    void* (*code)(void** free_variables);
    void* value;
    void* free_variables[];
} cell_model_thunk;

void* evaluate(cell_model_thunk* thunk) {
    if (!thunk->value) {
        thunk->value = \
            thunk->code(thunk->free_variables);
    }
    return thunk->value;
}
```

Self-Updating Thunks

```
typedef struct {
    void* (*code)(void** free_variables);
    void* free_variables[];
} self_updating_thunk;
```

Self-Updating Thunks

```
typedef struct {
    void* (*code)(void** free_variables);
    void* free_variables[];
} self_updating_thunk;

void* evaluate(self_updating_thunk* thunk) {
    return thunk->code(thunk->free_variables);
}
```

Update Frames

Definition (Update Frame)

Code injected in a **self-updating thunk** to update the thunk's closure to become the result or an **indirection** to the result.

Definition (Indirection)

A thunk whose `code` cell points to an identity function and whose `free_variables` point to a single thunk.

This is used when a **self-updating thunk**'s result is larger than the thunk itself.

Update Frames

```
typedef struct {
    void* (*code)(void** free_variable);
    void* free_variables[2];
} closure_2;
```

Update Frames

```
void update_small(closure_2* closure) {
    closure_2* result = evaluate_self_updating(closure);

    closure->code = result->code;

    closure->free_variables[0] = \
        result->free_variables[0];

    closure->free_variables[1] = \
        result->free_variables[1];
}
```

Update Frames

```
typedef struct {
    void* (*code)(void** free_variable);
    void* free_variables[2];
} closure_2;
```

Before Update

```
{code, free_var_1, free_var_2}
```

Update Frames

```
typedef struct {
    void* (*code)(void** free_variable);
    void* free_variables[2];
} closure_2;
```

Before Update

```
{code, free_var_1, free_var_2}
```

After Updating (small value)

```
{Cons, head, tail}
```

Update Frames

```
typedef struct {
    void* (*code)(void** free_variable);
    void* free_variables[2];
} closure_2;
```

Before Update

`{code, free_var_1, free_var_2}`

After Updating (small value)

`{Cons, head, tail}`

After Updating (big value)

`{indirection_code, value, NULL}`

value: `{mkTuple3, a, b, c}`

Update Frames

```
typedef struct {
    void* (*code)(void** free_variable);
    void* free_variables[2];
} closure_2;

void* indirection_code(void** free_variables) {
    return free_variables[0];
}

void update_large(closure_2* closure) {
    void* result = evaluate_self_updating(closure);
    closure->code = indirection_code;
    closure->free_variables[0] = result;
    closure->free_variables[1] = NULL;
}
```

Self-Updating Thunk Tricks

Black Hole

A code pointer which throws an exception or aborts the program.

Self-Updating Thunk Tricks

Black Hole

A code pointer which throws an exception or aborts the program.

```
let a = b
    b = a
in a
```

Self-Updating Thunk Tricks

Black Hole

A code pointer which throws an exception or aborts the program.

```
let a = b
    b = a
in a
```

Thread Synchronization

A code pointer which queues the current thread of execution to resume when the thunk is updated.

Function Application

Currying

```
> let f x y = x
```

Currying

```
> let f x y = x

> :t f
f :: a -> b -> a
```

Currying

```
> let f x y = x

> :t f
f :: a -> b -> a

f :: a -> (b -> a)
```

Currying

```
> let f x y = x

> :t f
f :: a -> b -> a

f :: a -> (b -> a)

> f 1 2
1
```

Currying

```
> let f x y = x

> :t f
f :: a -> b -> a

f :: a -> (b -> a)

> f 1 2
1

> (f 1) 2
1
```

Eval/Apply

1. Evaluate the function

Eval/Apply

1. Evaluate the function
2. Evaluate all of the arguments

Eval/Apply

1. Evaluate the function
2. Evaluate all of the arguments
3. Apply the function to the arguments

Push/Enter

1. Push arguments onto evaluation stack

Push/Enter

1. Push arguments onto evaluation stack
2. Tail call (enter) the function

Push/Enter

1. Push arguments onto evaluation stack
2. Tail call (enter) the function
3. The function enters the next function without returning

Example

```
apply3 f x y z = f x y z
```

Example

```
apply3 f x y z = f x y z
```

Eval/Apply

```
((f x) y) z
```

Example

```
apply3 f x y z = f x y z
```

Eval/Apply

```
((f x) y) z
```

Push/Enter

```
push x, y, z
enter f
```

Data Structures

Algebraic Data Types

```
data Boolean = False | True
data List a = Nil | Cons a (List a)
data Tuple3 a b c = MkTuple3 a b c
data Int = MkInt Int#
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Algebraic Data Types

```
data Boolean = False | True
data List a = Nil | Cons a (List a)
data Tuple3 a b c = MkTuple3 a b c
data Int = MkInt Int#
data Tree a = Leaf a | Branch (Tree a) (Tree a)

case xs of
  Nil             -> nilExpression
  Cons head tail -> consExpression
```

Case Expression

1. Evaluate the expression

Case Expression

1. Evaluate the expression
2. Scrutinize the value

Case Expression

1. Evaluate the expression
2. Scrutinize the value
3. Select the proper alternative

Switch on Tag

```
data List a = Cons a (List a)
            | Nil
```

Switch on Tag

```
closure* node;
void* alternative_continuation;
size_t case_tag;
#define CONS_TAG 0
void cons_code(void) {
    case_tag = CONS_TAG;
    jump(alternative_continuation);
}
#define NIL_TAG 1
void nil_code(void) {
    case_tag = NIL_TAG;
    jump(alternative_continuation);
}
```

Switch on Tag

Cons values

```
{cons_code, head, tail}
```

Nil values

```
{nil_code}
```

```
case xs of
    Cons head tail -> consAlternative
    Nil            -> nilAlternative
```

Switch on Tag

```
void some_list_case_expression(closure* c) {
    alternative_continuation = alternatives;
    node = c;
    jump(c->code);
}

void some_list_case_expression_alternatives() {
    switch(case_tag) {
    case CONS_TAG:
        /* body of `Cons` alternative */
    case NIL_TAG:
        /* body of `Nil` alternative */
    }
}
```

Vectored Return

```
void** return_vector;

void cons_code(closure* free_variables) {
    jump(return_vector[0]);
}

void nil_code(closure* free_variables) {
    jump(return_vector[1]);
}
```

Vectored Return

```
void* some_list_return_vector[] = {
    cons_alternative,
    nil_alternative,
};

void some_list_case_expression(closure* c) {
    return_vector = some_list_return_vector;
    node = c;
    jump(c->code);
}
```

Summary

Uniform representation of closures.

Summary

Uniform representation of closures.

Cheap indirections are available when performing updates.

Summary

Uniform representation of closures.

Cheap indirections are available when performing updates.

Exceptional conditions (black holes, concurrency) are handled in a uniform way.

Summary

Uniform representation of closures.

Cheap indirections are available when performing updates.

Exceptional conditions (black holes, concurrency) are handled in a uniform way.

Multiple return conventions for constructors are available.

The STG Language

Semantics

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

Characteristics

All function and constructor arguments are atoms (variables or constants).

Characteristics

All function and constructor arguments are atoms (variables or constants).

All constructor calls are saturated.

Characteristics

All function and constructor arguments are atoms (variables or constants).

All constructor calls are saturated.

Pattern matching is performed only by `case` expressions.

Characteristics

All function and constructor arguments are atoms (variables or constants).

All constructor calls are saturated.

Pattern matching is performed only by `case` expressions.

Unboxed values are supported.

Translating Haskell to STG

Haskell

```
map f []           = []
map f (head:tail) = (f head) : (map f tail)
```

Translating Haskell to STG

Haskell

```
map f []           = []
map f (head:tail) = (f head) : (map f tail)

map f xs = case xs of
    []           -> []
    (head:tail) -> f head : map f tail
```

Translating Haskell to STG

Haskell

```
map f []           = []
map f (head:tail) = (f head) : (map f tail)

map f xs = case xs of
    []           -> []
    (head:tail) -> f head : map f tail

map f xs = case xs of
    []           -> []
    (head:tail) -> let fHead    = f head
                       mapFTail = map f tail
                   in fHead : mapFTail
```

Translating Haskell to STG

STG

```
map = {} \n {f,xs} ->
  case xs of
    Nil {} -> Nil {}
    Cons {head,tail} ->
      let fHead    = {f,head} \u {} -> f {head}
          mapFTail = {f,tail} \u {} -> map {f,tail}
      in Cons {fHead,mapFTail}
```

General Translation Rules

Replace binary application with multiple application.

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

General Translation Rules

Saturate primitive operations and constructors with η-expansion.

$$C\{e_1, \ldots, e_n\} \Rightarrow \lambda y_1 \ldots y_m \to C\{e_1, \ldots, e_n, y_1, \ldots, y_m\}$$

where C is a primitive operation or constructor of arity $n + m$.

General Translation Rules

Name every non-atomic function argument, and every lambda-form, by adding a `let` expression.

```
f (a + 1) (b * 2) c
```

```
let arg1 = a + 1
    arg2 = b * 2
in f arg1 arg2 c
```

General Translation Rules

Convert the right-hand side of each `let` binding into a lambda-form by adding free-variable and update-flag information.

```
let arg1 = a + 1
    arg2 = b * 2
in f arg1 arg2 c
```

```
let arg1 = {a} \u {} -> + {a,1}
    arg2 = {b} \u {} -> * {b,2}
in f {arg1,arg2,c}
```

Identifying Free Variables

1. It is mentioned in the body of the lambda, and

Identifying Free Variables

1. It is mentioned in the body of the lambda, and
2. it is not an argument, and

Identifying Free Variables

1. It is mentioned in the body of the lambda, and
2. it is not an argument, and
3. it is not bound at the top level of the program.

Closures and updates

Manifest function

A manifest function is a lambda-form with a non-empty argument list.

Manifest functions do not require updating because they are already in head normal form.

Closures and updates

Partial application

A partial application is a lambda-form of the form:

```
{v1,...,vN} \n {} -> f {v1,...,vN}
```

where `f` is a known **manifest function** which takes more than n arguments.

Partial applications do not require updating because they are already in head normal form.

Closures and updates

Constructors

A constructor-closure is a lambda-form of the form

```
{v1,...,vN} \n {} -> C {v1,...,vN}
```

where `C` is a constructor.

Constructor code always returns to the alternative continuation so it does not require updating.

Closures and updates

Thunks

All other lambda-forms are thunks and should have their update flag set unless the compiler can prove the thunk will only be evaluated at most once.

```
f = {} \n {p,xs} ->
  let j = {p} \n {} -> factorial {p}
  in case xs of
    Nil {}       -> + {j,1}
    Cons {y,ys} -> + {j,2}
```

`j` is evaluated at most once so it doesn't need to be updated.

Translating Haskell into STG

```
map1 f = mf
  where mf []           = []
        mf (head:tail) = (f head) : (mf tail)
```

Translating Haskell into STG

```
map1 f = mf
  where mf []           = []
        mf (head:tail) = (f head) : (mf tail)

map1 = {} \n {f} ->
  letrec mf = {f,mf} \n {xs} ->
    case xs of
      Nil {}            -> Nil {}
      Cons {head,tail} ->
        let fHead    = {f,head} \u {} -> f {head}
            mapFTail = {mf,tail} \u {} -> mf {tail}
        in Cons {fHead,mapFTail}
  in mf
```

Standard constructors

Source

```
global = 1
myList = [global]
```

Standard constructors

Source

```
global = 1
myList = [global]
```

Non-standard constructor

```
global = {} \n {} -> MkInt 1#
nil = {} \n {} -> Nil {}
myList = {} \n {} -> Cons {global,nil}
```

Standard constructors

Source

```
global = 1
myList = [global]
```

Non-standard constructor

```
global = {} \n {} -> MkInt 1#
nil = {} \n {} -> Nil {}
myList = {} \n {} -> Cons {global,nil}
```

Standard constructor

```
global = {} \n {} -> MkInt 1#
nil = {} \n {} -> Nil {}
myList = {global,nil} \n {} -> Cons {global,nil}
```

Full laziness

```
f = {x} \n {y} -> let z = {x} \u {} ez
                  in ef
```

Full laziness

```
f = {x} \n {y} -> let z = {x} \u {} ez
                  in ef

z = {x} \u {} -> ez
f = {x,z} \n {y} -> ef
```

Arithmetic and unboxed Values

```
data Int = MkInt Int#
```

Arithmetic and unboxed Values

```
data Int = MkInt Int#
```

```
typedef struct {
    void* mkint_entry_code;
    int64_t value;
}
```

Arithmetic and unboxed Values

```
data Int = MkInt Int#
```

```
typedef struct {
    void* mkint_entry_code;
    int64_t value;
}
```

```
(+) = {} \n {a,b} ->
  case a of
    MkInt {a#} ->
      case b of
        MkInt {b#} -> case +# {a#,b#} of
          sum# -> MkInt sum#
```

Operational Semantics

Operational Semantics

State

1. the code

Operational Semantics

State

1. the code
2. the argument stack (values)

Operational Semantics

State

1. the code
2. the argument stack (values)
3. the return stack (continuations)

Operational Semantics

State

1. the code
2. the argument stack (values)
3. the return stack (continuations)
4. the update stack (update frames)

Operational Semantics

State

1. the code
2. the argument stack (values)
3. the return stack (continuations)
4. the update stack (update frames)
5. the heap (closures)

Operational Semantics

State

1. the code
2. the argument stack (values)
3. the return stack (continuations)
4. the update stack (update frames)
5. the heap (closures)
6. the global environment (closures)

Operational Semantics

Value

Either a heap address or a primitive integer value.

Other kinds of primitives (like float) could be supported but it would look exactly like int for the purposes of this discussion.

Operational Semantics

Global environment (σ)

A map from name of each top-level name to the address of a closure.

Operational Semantics

Local environment (ρ)

A map of each local variable name to the address of a closure.

Operational Semantics

val ρ σ v

If v is a primitive integer, return *Int* v.

If v is a variable, return its **value** in ρ.

If v is not in ρ return its **value** in σ.

Operational Semantics

Code

Eval e ρ	Evaluate expression *e* in the environment ρ and apply the value to the arguments on the argument stack.
Enter a	Apply the closure at address *a* to the arguments on the argument stack.
ReturnCon c ws	Return the constructor *c* applied to the values *ws* to the continuation on the return stack.
ReturnInt k	Return the primitive integer *k* to the continuation on the return stack.

State Transitions

Initial State

- all stacks are empty
- the heap holds all the globally bound closures
- σ holds the addresses of the globally bound
- σ holds the address of a closure called `main`

$$\mathit{Eval}\ (\mathit{main}\ \{\})\ \{\}$$

State Transitions

Function Application: $\mathit{Eval}\ (f\ xs)\ \rho$

1. push $\mathit{val}\ \rho\ \sigma\ xs$ onto the argument stack
2. let $\mathit{Addr}\ a = \mathit{val}\ \rho\ \sigma\ f$
3. move to $\mathit{Enter}\ a$

   note: $\mathit{Enter}\ a$ creates a fresh ρ

State Transitions

Entering Non-Updatable Closures: *Enter a*

when there are at least as many arguments on the stack as there are arguments to *a*...

1. *a* must be in the heap
2. update ρ the free variables in *a*
3. pop the consumed arguments from the argument stack
4. update ρ with the popped arguments
5. move to *Eval e* ρ where *e* is the body of *a*

State Transitions

`let(rec)` **Expressions:** *Eval* (`let` $x_n = v_n$ `in` e)

1. add all of the bound closures to the heap
2. add all of the entries to ρ
3. move to *Eval* e ρ

State Transitions

Case Expressions: *Eval* (`case` *e* `of` *alts*)

1. push $(alts, \rho)$ onto the return stack
2. move to *Eval* *e* ρ

State Transitions

Constructor Application: *Eval* $(C\ xs)\ \rho$

move to *ReturnCon* C $(val\ \rho\ \sigma\ xs)$

ReturnCon $C\ vs$

1. add *vs* to ρ
2. pop *alts* from the return stack
3. select the alternative whose constructor matches C
4. let $e =$ the body of the selected alternative
5. move to *Eval* $e\ \rho$

State Transitions

Unboxed Integers: *Eval k*

move to *ReturnInt k*

ReturnInt k

1. pop *alts* from the return stack
2. select the alternative whose value matches *k*
3. move to *Eval e*

Built-In Operations: *Eval* (`f# {a#,b#}`)

move to *ReturnInt* (*f# a# b#*)

State Transitions

Entering Updatable Closures: *Enter a*

1. push a tuple of the argument stack, the return stack, and *a* onto the update stack
2. clear the old argument and return stacks
3. move to *Eval e* ρ where *e* is the body of *a*

State Transitions

ReturnCon C vs **when the return stack is empty**

1. pop the top update frame from the update stack
2. restore the argument and return stacks from the update frame
3. overwrite the closure saved in the update frame to have a body of: `C {vs...}`
4. move to *ReturnCon Cvs*

State Transitions

Partial Application: *Enter a*

when there are at less arguments on the stack than there are arguments to *a*. . .

1. pop the top update frame from the update stack
2. restore the return stack from the update frame
3. update the saved closure*
4. push all of the update frame's argument stack members onto the argument stack

State Transitions

Update the saved closure

1. let `oldFreeVars` be the current free variables of the closure saved in the update frame
2. let `newFreeVars` be the old argument stack members
3. let `remainingArgs` be the arguments of *a* which were not on the stack
4. let `e` be the body of *a*
5. overwrite the closure saved in the update frame to be:

   ```
   {oldFreeVars ++ newFreeVars} \n {remainingArgs} -> e
   ```

Generating Code from STG

Choices?

Code generation options

- ANSI C
- GNU C
- machine code

Choices?

Code generation options

- ANSI C
- GNU C*
- machine code

Jumps

Requirements for code labels

- name an arbitrary sequence of code
- can be manipulated (pushed on a stack, stored in a closure, etc. . .)
- can be jumped to

Tiny Interpreter

```
typedef void* (*(*code_label)(void))(void);

void interpreter(code_label continuation) {
    while (1) {
        continuation = continuation();
    }
}
```

Tiny Interpreter

Optimizations

- eliminate register saves

Tiny Interpreter

Optimizations

- eliminate register saves
- eliminate the frame pointer (used for C debuggers)

Tiny Interpreter

Optimizations

eliminate register saves

eliminate the frame pointer (used for C debuggers)

use direct jumps instead of returning continuations:

```
#define JUMP(cont) return cont
#define JUMP(cont) \
    asm volatile ("jump %P0" : "p" (cont))
```

Representing Closures

```
typedef struct {
    code_label standard_entry_code;
    code_label evacuation_code;
    code_label scavenge_code;
    /* debugging info, misc info */
} info_table;

typedef struct {
    info_table* info;
    void* closure_free_variables[];
    int64_t primitive_free_variables[];
  };
```

Stacks

1. argument stack (closures)
2. return stack (code labels)
3. update stack (update frames)

One Stack

All three stacks work in synchrony so we could put all the values on a single stack.

This stresses the garbage collector because we would need a way to identify which values are pointers to closures.

Stacks

1. argument stack (closures)
2. return stack (code labels)
3. update stack (update frames)

Two Stacks

Use one stack for pointers to closures and another stack for everything else.

The "A" stack is the pointer stack ("A" for Argument).

The "B" stack is the Basic value stack.

The rest

- garbage collection
- black holes
- compiling `case` expressions
- different constructor return conventions
- partial application
- statically allocated closures
- optimizations everywhere
- “gory” details

Key Ideas

- `case` expressions are the only way to drive computation

Key Ideas

- `case` expressions are the only way to drive computation
- all information about the shape and type of an object are controlled through the entry code

Key Ideas

- `case` expressions are the only way to drive computation
- all information about the shape and type of an object are controlled through the entry code
- control flow moves from `case` alternative to `case` alternative

Key Ideas

- `case` expressions are the only way to drive computation
- all information about the shape and type of an object are controlled through the entry code
- control flow moves from `case` alternative to `case` alternative
- data structures are often synthetic