

Closure Implementation with Run-time Code Generation

Martin Grabmüller

Fachgebiet Übersetzerbau und Programmiersprachen
Institut für Softwaretechnik und Theoretische Informatik
Fakultät IV – Elektrotechnik und Informatik
Technische Universität Berlin

ÜBB-Kolloquium, 17. Januar 2006



Introduction to Closures and Run-time Code Generation

Run-time Code Generator Implementation

Evaluation and Performance Comparison

Future Work and Conclusion



Introduction to Closures and Run-time Code Generation

Run-time Code Generator Implementation

Evaluation and Performance Comparison

Future Work and Conclusion



- ▶ Technique for implementing free variables



Introduction to Closures

- ▶ Technique for implementing free variables

Example

add = $\lambda x.\lambda y.x+y$



Introduction to Closures

- ▶ Technique for implementing free variables

Example

add = $\lambda x. \lambda y. x+y$

Variable x is free in inner abstraction



Introduction to Closures

- ▶ Technique for implementing free variables

Example

`add = $\lambda x. \lambda y. x+y$`

Variable `x` is free in inner abstraction

- ▶ Function (representation) is paired with environment on closure creation time



Introduction to Closures

- ▶ Technique for implementing free variables

Example

$\text{add} = \lambda x. \lambda y. x+y$

Variable x is free in inner abstraction

- ▶ Function (representation) is paired with environment on closure creation time
- ▶ Environment contains one value for each free variable of the function



Introduction to Closures

- ▶ Technique for implementing free variables

Example

`add = $\lambda x. \lambda y. x+y$`

Variable `x` is free in inner abstraction

- ▶ Function (representation) is paired with environment on closure creation time
- ▶ Environment contains one value for each free variable of the function

Example

`add 2`



Introduction to Closures

- ▶ Technique for implementing free variables

Example

$\text{add} = \lambda x. \lambda y. x+y$

Variable x is free in inner abstraction

- ▶ Function (representation) is paired with environment on closure creation time
- ▶ Environment contains one value for each free variable of the function

Example

$\text{add } 2 \rightarrow (\lambda x. \lambda y. x+y) 2$



Introduction to Closures

- ▶ Technique for implementing free variables

Example

$\text{add} = \lambda x. \lambda y. x+y$

Variable x is free in inner abstraction

- ▶ Function (representation) is paired with environment on closure creation time
- ▶ Environment contains one value for each free variable of the function

Example

$\text{add } 2 \rightarrow (\lambda x. \lambda y. x+y) 2 \rightarrow [\lambda y. x+y, \{x=2\}]$



Closure Implementation Issues

- ▶ Closure implementation affects three operations:
 - ▶ closure creation
 - ▶ function calls
 - ▶ access to free variables



Closure Implementation Issues

- ▶ Closure implementation affects three operations:
 - ▶ closure creation
 - ▶ function calls
 - ▶ access to free variables
- ▶ Well-known implementation techniques:
 - ▶ Flat closures (bindings for all free variables)
 - ▶ Nested closures (omit bindings for variables in enclosing closures)



Closure Implementation Issues

- ▶ Closure implementation affects three operations:
 - ▶ closure creation
 - ▶ function calls
 - ▶ access to free variables
- ▶ Well-known implementation techniques:
 - ▶ Flat closures (bindings for all free variables)
 - ▶ Nested closures (omit bindings for variables in enclosing closures)
- ▶ Reduced performance compared to closure-free languages



Closure Implementation Issues

- ▶ Closure implementation affects three operations:
 - ▶ closure creation
 - ▶ function calls
 - ▶ access to free variables
- ▶ Well-known implementation techniques:
 - ▶ Flat closures (bindings for all free variables)
 - ▶ Nested closures (omit bindings for variables in enclosing closures)
- ▶ Reduced performance compared to closure-free languages
- ▶ Different approach: run-time code generation



Introduction to Run-time Code Generation

- ▶ Program consists of *program representation* and *code generator*



Introduction to Run-time Code Generation

- ▶ Program consists of *program representation* and *code generator*
- ▶ Prior to execution, program representation is compiled to machine code



Introduction to Run-time Code Generation

- ▶ Program consists of *program representation* and *code generator*
- ▶ Prior to execution, program representation is compiled to machine code
- ▶ Either program gets compiled completely, or incrementally



Introduction to Run-time Code Generation

- ▶ Program consists of *program representation* and *code generator*
- ▶ Prior to execution, program representation is compiled to machine code
- ▶ Either program gets compiled completely, or incrementally
- + More optimization opportunities because of more information (crucial for dynamic language features such as reflection or dynamic code loading)



Introduction to Run-time Code Generation

- ▶ Program consists of *program representation* and *code generator*
- ▶ Prior to execution, program representation is compiled to machine code
- ▶ Either program gets compiled completely, or incrementally
- + More optimization opportunities because of more information (crucial for dynamic language features such as reflection or dynamic code loading)
- + Selective code generation



Introduction to Run-time Code Generation

- ▶ Program consists of *program representation* and *code generator*
- ▶ Prior to execution, program representation is compiled to machine code
- ▶ Either program gets compiled completely, or incrementally
- + More optimization opportunities because of more information (crucial for dynamic language features such as reflection or dynamic code loading)
- + Selective code generation
- + Portability of code files



Introduction to Run-time Code Generation

- ▶ Program consists of *program representation* and *code generator*
- ▶ Prior to execution, program representation is compiled to machine code
- ▶ Either program gets compiled completely, or incrementally
- + More optimization opportunities because of more information (crucial for dynamic language features such as reflection or dynamic code loading)
- + Selective code generation
- + Portability of code files
- Code generation time adds to program run-time



Introduction to Run-time Code Generation

- ▶ Program consists of *program representation* and *code generator*
- ▶ Prior to execution, program representation is compiled to machine code
- ▶ Either program gets compiled completely, or incrementally
- + More optimization opportunities because of more information (crucial for dynamic language features such as reflection or dynamic code loading)
- + Selective code generation
- + Portability of code files
- Code generation time adds to program run-time
- More complex runtime system



Introduction to Run-time Code Generation

- ▶ Program consists of *program representation* and *code generator*
- ▶ Prior to execution, program representation is compiled to machine code
- ▶ Either program gets compiled completely, or incrementally
- + More optimization opportunities because of more information (crucial for dynamic language features such as reflection or dynamic code loading)
- + Selective code generation
- + Portability of code files
- Code generation time adds to program run-time
- More complex runtime system
- Greater memory requirements



Closure Implementation with Run-time Code Generation

- ▶ Idea: generate code on closure creation (Feeley and Lapalme)



Closure Implementation with Run-time Code Generation

- ▶ Idea: generate code on closure creation (Feeley and Lapalme)
- ▶ Embed values of free variables into generated code



Closure Implementation with Run-time Code Generation

- ▶ Idea: generate code on closure creation (Feeley and Lapalme)
- ▶ Embed values of free variables into generated code

Example

add 2



Closure Implementation with Run-time Code Generation

- ▶ Idea: generate code on closure creation (Feeley and Lapalme)
- ▶ Embed values of free variables into generated code

Example

`add 2` \rightarrow $(\lambda x. \lambda y. x+y) 2$



Closure Implementation with Run-time Code Generation

- ▶ Idea: generate code on closure creation (Feeley and Lapalme)
- ▶ Embed values of free variables into generated code

Example

$\text{add } 2 \rightarrow (\lambda x. \lambda y. x+y) \ 2 \rightarrow \lambda y. 2+y$



Closure Implementation with Run-time Code Generation

- ▶ Idea: generate code on closure creation (Feeley and Lapalme)
- ▶ Embed values of free variables into generated code

Example

add 2 \rightarrow $(\lambda x. \lambda y. x+y)$ 2 \rightarrow $\lambda y. 2+y$

- + Eliminate accesses to free variables



Closure Implementation with Run-time Code Generation

- ▶ Idea: generate code on closure creation (Feeley and Lapalme)
- ▶ Embed values of free variables into generated code

Example

add 2 \rightarrow $(\lambda x. \lambda y. x+y)$ 2 \rightarrow $\lambda y. 2+y$

- + Eliminate accesses to free variables
- + Eliminate indirection on function calls



Closure Implementation with Run-time Code Generation

- ▶ Idea: generate code on closure creation (Feeley and Lapalme)
- ▶ Embed values of free variables into generated code

Example

add 2 $\rightarrow (\lambda x. \lambda y. x+y)$ 2 $\rightarrow \lambda y. 2+y$

- + Eliminate accesses to free variables
- + Eliminate indirection on function calls
- Closure creation is expensive



Closure Implementation with Run-time Code Generation

- ▶ Idea: generate code on closure creation (Feeley and Lapalme)
- ▶ Embed values of free variables into generated code

Example

add 2 $\rightarrow (\lambda x. \lambda y. x+y)$ 2 $\rightarrow \lambda y. 2+y$

- + Eliminate accesses to free variables
- + Eliminate indirection on function calls
- Closure creation is expensive
- Memory intensive because of many specialized functions



Closure Implementation with Run-time Code Generation

- ▶ Idea: generate code on closure creation (Feeley and Lapalme)
- ▶ Embed values of free variables into generated code

Example

add 2 $\rightarrow (\lambda x. \lambda y. x+y)$ 2 $\rightarrow \lambda y. 2+y$

- + Eliminate accesses to free variables
- + Eliminate indirection on function calls
- Closure creation is expensive
- Memory intensive because of many specialized functions
- ▶ **Until now, only implemented in limited form**
(only wrapper code is generated, not complete functions)



Introduction to Closures and Run-time Code Generation

Run-time Code Generator Implementation

Evaluation and Performance Comparison

Future Work and Conclusion



- ▶ Handles programs in strict untyped λ -calculus enriched with constants, conditionals, (recursive) let, simple arithmetic



Run-time Compiler

- ▶ Handles programs in strict untyped λ -calculus enriched with constants, conditionals, (recursive) let, simple arithmetic
- ▶ Programs are stored in form of abstract syntax trees (annotated with free-variable information)



Run-time Compiler

- ▶ Handles programs in strict untyped λ -calculus enriched with constants, conditionals, (recursive) let, simple arithmetic
- ▶ Programs are stored in form of abstract syntax trees (annotated with free-variable information)
- ▶ First, program is compiled, deferring compilation of embedded λ -expressions



Run-time Compiler

- ▶ Handles programs in strict untyped λ -calculus enriched with constants, conditionals, (recursive) let, simple arithmetic
- ▶ Programs are stored in form of abstract syntax trees (annotated with free-variable information)
- ▶ First, program is compiled, deferring compilation of embedded λ -expressions
- ▶ On closure creation, control is transferred to code generator



Run-time Compiler

- ▶ Handles programs in strict untyped λ -calculus enriched with constants, conditionals, (recursive) let, simple arithmetic
- ▶ Programs are stored in form of abstract syntax trees (annotated with free-variable information)
- ▶ First, program is compiled, deferring compilation of embedded λ -expressions
- ▶ On closure creation, control is transferred to code generator
- ▶ Closures are represented by pointer to compiled code



Run-time Compiler

- ▶ Handles programs in strict untyped λ -calculus enriched with constants, conditionals, (recursive) let, simple arithmetic
- ▶ Programs are stored in form of abstract syntax trees (annotated with free-variable information)
- ▶ First, program is compiled, deferring compilation of embedded λ -expressions
- ▶ On closure creation, control is transferred to code generator
- ▶ Closures are represented by pointer to compiled code
- ▶ One-pass recursive descent compiler with simple register allocation



Run-time Compiler

- ▶ Handles programs in strict untyped λ -calculus enriched with constants, conditionals, (recursive) let, simple arithmetic
- ▶ Programs are stored in form of abstract syntax trees (annotated with free-variable information)
- ▶ First, program is compiled, deferring compilation of embedded λ -expressions
- ▶ On closure creation, control is transferred to code generator
- ▶ Closures are represented by pointer to compiled code
- ▶ One-pass recursive descent compiler with simple register allocation
- ▶ Directly generates IA-32 machine code in memory



Example: Source Code

```
letrec
  odd = \ x -> if x = 0 then 0
            else even (x - 1)
            fi
  even = \ x -> if x = 0 then 1
                else odd (x - 1)
                fi
  start = \ y -> even y
in
  start 42
end
```



Example: Source Code

```
letrec
  odd = \ x -> if x = 0 then 0
            else even (x - 1)
            fi
  even = \ x -> if x = 0 then 1
                else odd (x - 1)
                fi
  start = \ y -> even y
in
  start 42
end
```



Example: Source Code

```
letrec
  odd = \ x -> if x = 0 then 0
            else even (x - 1)
            fi
  even = \ x -> if x = 0 then 1
                else odd (x - 1)
                fi
  start = \ y -> even y
in
  start 42
end
```



Example: Source Code

```
letrec
  odd = \ x -> if x = 0 then 0
            else even (x - 1)
            fi
  even = \ x -> if x = 0 then 1
                else odd (x - 1)
                fi
  start = \ y -> even y
in
  start 42
end
```



Example: Source Code

```
letrec
  odd = \ x -> if x = 0 then 0
            else even (x - 1)
            fi
  even = \ x -> if x = 0 then 1
            else odd (x - 1)
            fi
  start = \ y -> even y
in
  start 42
end
```



Example: Source Code

```
letrec
  odd = \ x -> if x = 0 then 0
            else even (x - 1)
            fi
  even = \ x -> if x = 0 then 1
            else odd (x - 1)
            fi
  start = \ y -> even y
in
  start 42
end
```



Example: Generated Code

```
odd:
    push    %ebp
    mov     %esp,%ebp
    mov     0x8(%ebp),%eax
    cmp     $0x0,%eax
    jne     l1
    xor     %eax,%eax
    leave
    ret
l1:  mov     0x8(%ebp),%eax
    sub     $0x1,%eax
    push    %eax
    call    <even>
    leave
    ret
even:
    push    %ebp
    mov     %esp,%ebp
    mov     0x8(%ebp),%eax
    cmp     $0x0,%eax
    jne     l2
    mov     $0x1,%eax
    leave
    ret
l2:  mov     0x8(%ebp),%eax
    sub     $0x1,%eax
    push    %eax
    call    <odd>
    leave
    ret
start:
    push    %ebp
    mov     %esp,%ebp
    pushl   0x8(%ebp)
    call    <even>
    leave
    ret
```



Example: Generated Code

```
odd:
    push    %ebp
    mov     %esp,%ebp
    mov     0x8(%ebp),%eax
    cmp     $0x0,%eax
    jne     l1
    xor     %eax,%eax
    leave
    ret
l1:  mov     0x8(%ebp),%eax
    sub     $0x1,%eax
    push   %eax
    call   <even>
    leave
    ret
even:
    push   %ebp
    mov    %esp,%ebp
    mov    0x8(%ebp),%eax
    cmp    $0x0,%eax
    jne    l2
    mov    $0x1,%eax
    leave
    ret
l2:  mov     0x8(%ebp),%eax
    sub     $0x1,%eax
    push   %eax
    call   <odd>
    leave
    ret
start:
    push   %ebp
    mov    %esp,%ebp
    pushl  0x8(%ebp)
    call   <even>
    leave
    ret
```



Example: Generated Code

```
odd:
    push    %ebp
    mov     %esp,%ebp
    mov     0x8(%ebp),%eax
    cmp     $0x0,%eax
    jne     l1
    xor     %eax,%eax
    leave
    ret
l1:  mov     0x8(%ebp),%eax
    sub     $0x1,%eax
    push    %eax
    call    <even>
    leave
    ret
even:
    push    %ebp
    mov     %esp,%ebp
    mov     0x8(%ebp),%eax
    cmp     $0x0,%eax
    jne     l2
    mov     $0x1,%eax
    leave
    ret
l2:  mov     0x8(%ebp),%eax
    sub     $0x1,%eax
    push    %eax
    call    <odd>
    leave
    ret
start:
    push    %ebp
    mov     %esp,%ebp
    pushl   0x8(%ebp)
    call    <even>
    leave
    ret
```



Example: Generated Code

```
odd:
    push    %ebp
    mov     %esp,%ebp
    mov     0x8(%ebp),%eax
    cmp     $0x0,%eax
    jne     l1
    xor     %eax,%eax
    leave
    ret
l1:  mov     0x8(%ebp),%eax
    sub     $0x1,%eax
    push    %eax
    call    <even>
    leave
    ret
even:
    push    %ebp
    mov     %esp,%ebp
    mov     0x8(%ebp),%eax
    cmp     $0x0,%eax
    jne     l2
    mov     $0x1,%eax
    leave
    ret
l2:  mov     0x8(%ebp),%eax
    sub     $0x1,%eax
    push    %eax
    call    <odd>
    leave
    ret
start:
    push    %ebp
    mov     %esp,%ebp
    pushl   0x8(%ebp)
    call    <even>
    leave
    ret
```



Example: Generated Code

```
odd:
    push    %ebp
    mov     %esp,%ebp
    mov     0x8(%ebp),%eax
    cmp     $0x0,%eax
    jne     l1
    xor     %eax,%eax
    leave
    ret
l1:  mov     0x8(%ebp),%eax
    sub     $0x1,%eax
    push   %eax
    call    <even>
    leave
    ret
even:
    push   %ebp
    mov    %esp,%ebp
    mov    0x8(%ebp),%eax
    cmp    $0x0,%eax
    jne    l2
    mov    $0x1,%eax
    leave
    ret
l2:  mov     0x8(%ebp),%eax
    sub     $0x1,%eax
    push   %eax
    call    <odd>
    leave
    ret
start:
    push   %ebp
    mov    %esp,%ebp
    pushl  0x8(%ebp)
    call   <even>
    leave
    ret
```



Example: Remarks

- ▶ Optimization omitted, this particular example benefits from tail-recursion optimization



Example: Remarks

- ▶ Optimization omitted, this particular example benefits from tail-recursion optimization
- ▶ Stack frame management could easily be optimized away



Example: Remarks

- ▶ Optimization omitted, this particular example benefits from tail-recursion optimization
- ▶ Stack frame management could easily be optimized away
- ▶ More details can be found in research report (to appear)



Introduction to Closures and Run-time Code Generation

Run-time Code Generator Implementation

Evaluation and Performance Comparison

Future Work and Conclusion



Table: Summary of benchmark programs

Name	Description
iter	Loops 10^9 times
citer	Counts from 0 to 10^6
odd-even	Calculates whether 10^9 is odd or even
fib	Calculates the Fibonacci number for 42
nested	Two nested loops, inner depending on outer variable



Benchmark Results

Table: Results of benchmark programs.

Name	Total (s)	Codegen (s)	Code size (bytes)
iter	2.516	0.00001	109
citer	3.353	3.06646	44000163
odd-even	2.544	0.00002	244
fib	6.842	0.00001	132
nested	2.744	0.00255	41193



Comparison with Other Implementations

Table: Languages and language implementations

Language	Compiler	Version
C	GNU Compiler Collection	4.0.2
Scheme	Stalin	0.10
Haskell-a	GHC (no annotations)	6.4.1
Haskell-b	GHC (annotations)	6.4.1
Haskell-c	GHC (annotations+checks)	6.4.1
Standard ML	MLton	20041109
Opal	OCS	2.3i



Comparison with Other Implementations – fib

Table: Times for fib

Language	Time	Ratio
Reference	6.842	1.000
C	3.302	0.483
Scheme	3.316	0.485
Haskell-a	116.221	16.986
Haskell-b	8.994	1.315
Haskell-c	18.933	2.767
Standard ML	7.767	1.135
Opal	23.732	3.469



Comparison with Other Implementations – nested

Table: Times for nested

Language	Time	Ratio
Reference	2.744	1.000
C	0.629	0.229
Scheme	0.632	0.230
Haskell-a	0.002	–
Haskell-b	0.001	–
Haskell-c	0.001	–
Standard ML	1.669	0.608
Opal	–	–



- ▶ Code generation rate is high (up to 13.86 MB per second)



- ▶ Code generation rate is high (up to 13.86 MB per second)
- ▶ Programs with moderate generation rate are competitive to other implementations



- ▶ Code generation rate is high (up to 13.86 MB per second)
- ▶ Programs with moderate generation rate are competitive to other implementations
- ▶ More mature implementation needed for fair comparison



Introduction to Closures and Run-time Code Generation

Run-time Code Generator Implementation

Evaluation and Performance Comparison

Future Work and Conclusion



- ▶ Re-implementing the prototype (proper error handling, type checking etc.)



- ▶ Re-implementing the prototype (proper error handling, type checking etc.)
- ▶ Add data structures, input/output etc.



- ▶ Re-implementing the prototype (proper error handling, type checking etc.)
- ▶ Add data structures, input/output etc.
- ▶ Write runtime environment (garbage collector etc.)



- ▶ Re-implementing the prototype (proper error handling, type checking etc.)
- ▶ Add data structures, input/output etc.
- ▶ Write runtime environment (garbage collector etc.)
- ▶ Add profiling to avoid excessive code generation by falling back to wrapper code generation or other closure technique



- ▶ Re-implementing the prototype (proper error handling, type checking etc.)
- ▶ Add data structures, input/output etc.
- ▶ Write runtime environment (garbage collector etc.)
- ▶ Add profiling to avoid excessive code generation by falling back to wrapper code generation or other closure technique
- ▶ Use as testbed for theoretical model of dynamic compilation



Conclusion

We have seen...

- ▶ Description of implementation technique for closures in pure functional languages (only theoretical until now)



We have seen...

- ▶ Description of implementation technique for closures in pure functional languages (only theoretical until now)
- ▶ Description of working prototype implementation —
Implementation straightforward, minor difficulties



We have seen...

- ▶ Description of implementation technique for closures in pure functional languages (only theoretical until now)
- ▶ Description of working prototype implementation —
Implementation straightforward, minor difficulties
- ▶ Performance evaluation and comparison to other systems —
Compares quite well



We have seen...

- ▶ Description of implementation technique for closures in pure functional languages (only theoretical until now)
- ▶ Description of working prototype implementation —
Implementation straightforward, minor difficulties
- ▶ Performance evaluation and comparison to other systems —
Compares quite well
- ▶ Outline of future work



We have seen...

- ▶ Description of implementation technique for closures in pure functional languages (only theoretical until now)
- ▶ Description of working prototype implementation —
Implementation straightforward, minor difficulties
- ▶ Performance evaluation and comparison to other systems —
Compares quite well
- ▶ Outline of future work

Thank You!





Marc Feeley and Guy Lapalme.

Closure generation based on viewing lambda as epsilon plus compile.

Journal of Computer Languages, 17(4), 1992.



Martin Grabmüller.

Implementing Closures using Run-time Code Generation.

Research report 2006-02 in *Forschungsberichte Fakultät IV – Elektrotechnik und Informatik*, Technische Universität Berlin, February 2006.

