Code Versioning and Extremely Lazy Compilation of Scheme

Baptiste Saleil & Marc Feeley

Université de Montréal

November 19, 2014
1 Introduction

2 Extremely lazy compilation

3 Code versioning

4 Conclusion
Static vs Dynamic type checking

Static type checking
- Types are known at compile time
- Type errors are detected at compilation

Dynamic type checking
- No type information at compile time
- Type checks embedded in generated code
Static vs Dynamic type checking

Static type checking
- Types are known at compile time
- Type errors are detected at compilation

Dynamic type checking
- No type information at compile time
- Type checks embedded in generated code

→ Remove dynamic type checks
JIT compilers:

- Portability & Performance
- Lazy compilation
- Compilation time $\rightarrow$ Execution time
JIT compilers:

- Portability & Performance
- Lazy compilation
- Compilation time $\rightarrow$ Execution time

Existing solutions:

- Type inference
  - Detect types at compilation, remove type tests
  - Implies static analysis
JIT compilers:

- Portability & Performance
- Lazy compilation
- Compilation time → Execution time

Existing solutions:

- Type inference
  → Detect types at compilation, remove type tests
  → Implies static analysis
- Type annotation
  → Type hints to compiler
  → Lose expressiveness of dynamic languages
Goals

- Remove as many type checks as possible
Goals

- Remove as many type checks as possible
- Avoid expensive static analysis
Goals

- Remove as many type checks as possible
- Avoid expensive static analysis
- Keep expressiveness of dynamically typed languages
1 Introduction

2 Extremely lazy compilation

3 Code versioning

4 Conclusion
What is extremely lazy compilation?

\[(+ (- a 10) a)\]

An analysis shows that type test on \(a\) is unnecessary.
What is extremely lazy compilation?

\[ (+ \ (- \ a \ 10) \ a) \]

An analysis shows that type test on \( a \) is unnecessary.
What is extremely lazy compilation?

\[(+ (- a 10) a)\]

An analysis shows that type test on \(a\) is unnecessary.
Left operand → Right operand → Addition
What is extremely lazy compilation?

\((+ (- a 10) a)\)

An analysis shows that type test on \(a\) is unnecessary
Left operand → Right operand → Addition

Our approach

Why not use the information from execution of predecessors to optimize code generation of current node?

→ Each node of s-expression is a stub
Lazy code object

- Code generator which take a compilation context
- Successor object
- Entry point
Lazy code object chain

```lisp
(define (gen-ast ast successor)
  ...
  (if (number? ast)
      (make-lazy-code
       (lambda (ctx)
        (gen-push (encode ast))
        (jump-to successor (push-ctx 'number ctx))))
      ...
      (if (eq? (car ast) '+)
        (let* ((lazy-add
                    (make-lazy-code
                     (lambda (ctx)
                      (gen-pop r1)
                      (gen-pop r2)
                      (cond ((not (number? (stack-first ctx)))
                                 (gen-check-if-number r1 ctx))
                             ((not (number? (stack-second ctx)))
                              (gen-check-if-number r2 ctx))
                             (gen-add r1 r2)
                             (gen-push r1)
                             (jump-to successor
                              (push-ctx 'number
                               (push-ctx (pop-ctx (pop-ctx ctx ctx))))))
                      (lazy-arg1
                       (gen-ast (caddr ast) lazy-add)))
                      (gen-ast (cadr ast) lazy-arg1)))
        (lazy-arg1
         (gen-ast (caddr ast) lazy-add)))
      ...))
```
Lazy code object chain

Run expression

1 (let ((obj (gen-ast '+ (- - a 10) a)
2   (make-lazy-code
3     (lambda (ctx)
4       (gen-pop r1)
5       (gen-return)))))
6   (execute obj init-ctx))
Example: (+ (- a 10) a)
Example: \((+ (- a 10) a)\)
Example: \((+(-a10)a)\)
Example: \((+ (- a 10) a)\)
Example: \((+\ (-\ a\ 10)\ a)\)

**Introduction**

**Extremely lazy compilation**

**Code versioning**

**Conclusion**

```
Example: (+ (- a 10) a)
```

```
23 / 41
```
Example: \((+ (- a 10) a)\)
Example: (+ (- a 10) a)
Problem - How to handle join points?

(let ((a (if b
    10
    (read))))
(+ a 100))
Problem - How to handle join points?

(let ((a (if b
          10
          (read))))
 (let ((a (if b
          10
          (read))))
    (+ a 100)))
Problem - How to handle join points?

(let ((a (if b
  10
  (read))))
(+ a 100))
Each lazy code object has multiple versions
Each version associated to compilation context
Each piece of code now has multiple entry points
Lazy code object

- Code generator
- Successor object
- Context $\rightarrow$ Address table
Lazy code object

- Code generator
- Successor object
- Context → Address table

- CTX4 = '(number number)
- CTX1 = '(number unknown)
- CTX5 = '(number string)
(let ((a (if b
   10
   (read)))))
(+ a 100)
Problem 1

- Functions also have multiple entry points
- Flat closure representation is not suitable
  → New closure representation (cc-table)

```
| ... | Address | non-local 1 | ... | non-local n | ... |
```

- Add indirection
Problem 1

- Functions also have multiple entry points
- Flat closure representation is not suitable
  → New closure representation (cc-table)

Closure

<table>
<thead>
<tr>
<th>...</th>
<th>non-local 1</th>
<th>...</th>
<th>non-local n</th>
<th>...</th>
</tr>
</thead>
</table>

| Stub addr | Stub addr | Stub addr | Stub addr |

- Add indirection
Problem 1

- Functions also have multiple entry points
- Flat closure representation is not suitable
  → New closure representation (cc-table)

- Add indirection
Problem 2

- We don’t know statically which function we call
- What is the offset corresponding to calling context?
  \[\rightarrow\] Keep a global cc-table

<table>
<thead>
<tr>
<th>Global table</th>
<th>Index 0</th>
<th>Index 1</th>
<th>Index 2</th>
<th>Index 3</th>
<th>Index 4</th>
<th>Index 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTX4</td>
<td>CTX1</td>
<td>CTX8</td>
<td>CTX9</td>
<td>CTX3</td>
<td>CTX10</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>λ1 table</th>
<th>STUB λ1</th>
<th>ADDR X1</th>
<th>STUB λ1</th>
<th>STUB λ1</th>
<th>ADDR X2</th>
<th>STUB λ1</th>
<th>...</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>λ2 table</th>
<th>ADDR Y1</th>
<th>ADDR Y2</th>
<th>ADDR Y3</th>
<th>STUB λ2</th>
<th>STUB λ2</th>
<th>STUB λ2</th>
<th>...</th>
</tr>
</thead>
</table>

...
Summary

Pros

- Remove type checks if unnecessary
- Remove type checks if unnecessary in some execution
- Suitable for JIT compilation
- Keep dynamic language expressivity

Cons

- Size problem
  - Balanced by lazy compilation
- Indirection on call
  - Can avoid several type checks
- Heap overflow on pathological cases
Results:
- No extensive benchmark results yet
- Observation: A lot of type checks are removed

Remaining work:
- Benchmarking!
- Improve context propagation
- Analyze heap / memory consumption
Thanks!

Baptiste Saleil
baptiste.saleil@umontreal.ca