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 → Execution time

JIT compilers :

- Portability & Performance
- Lazy compilation
- Compilation time → 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

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

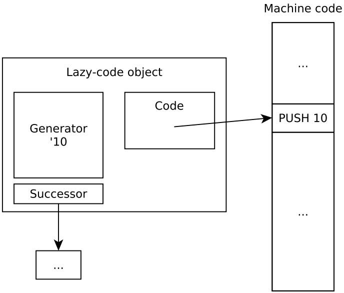
Our approach

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

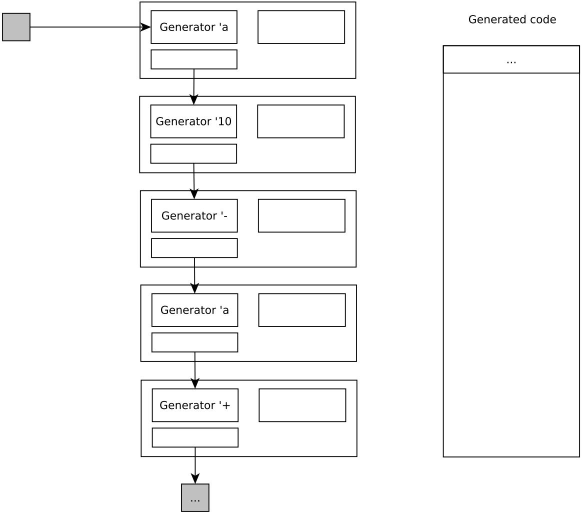
```
(define (gen-ast ast successor)
   ...
   (if (number? ast)
      (make-lazy-code
         (lambda (ctx)
            (gen-push (encode ast))
            (jump-to successor (push-ctx 'number ctx)))))
   ...
   (if (eq? (car ast) '+)
      (let* ((lazy-add
                (make-lazy-code
                   (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
                                        (pop-ctx (pop-ctx ctx)))))))
             (lazy-arg1
                (gen-ast (caddr ast) lazy-add)))
         (gen-ast (cadr ast) lazy-arg1)))
   ...)
```

Lazy code object chain

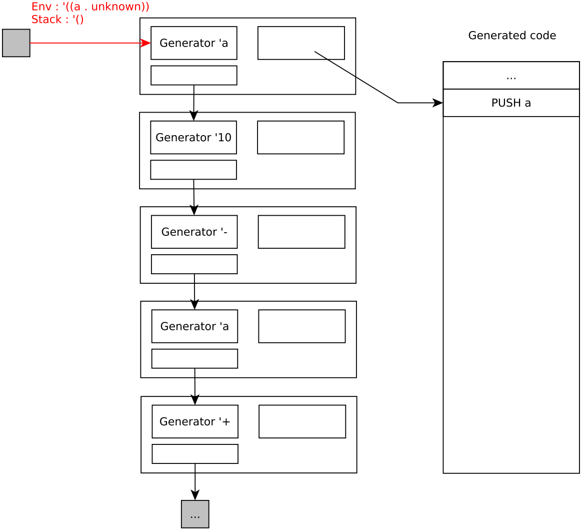
- Run expression

```
(let ((obj (gen-ast '(+ (- a 10) a)
                    (make-lazy-code
                       (lambda (ctx)
                          (gen-pop r1)
                          (gen-return))))))
   (execute obj init-ctx))
```

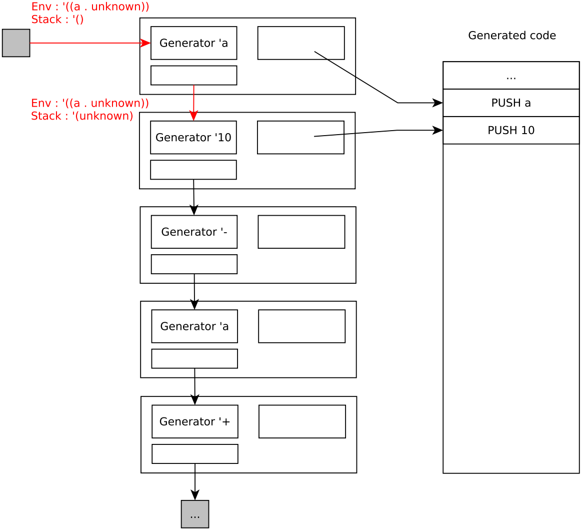
Example : (+ (- a 10) a)



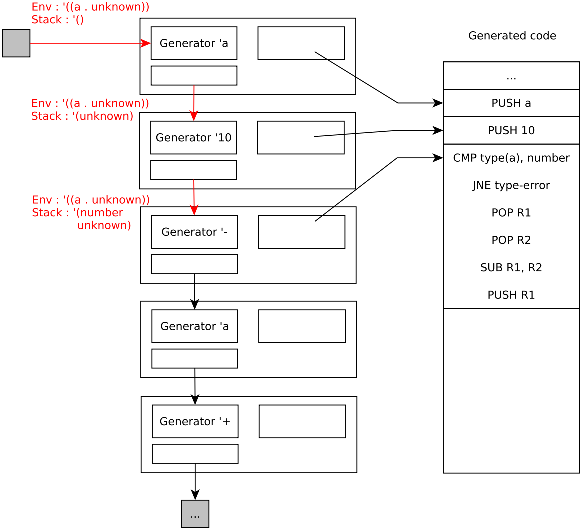
Example : (+ (- a 10) a)



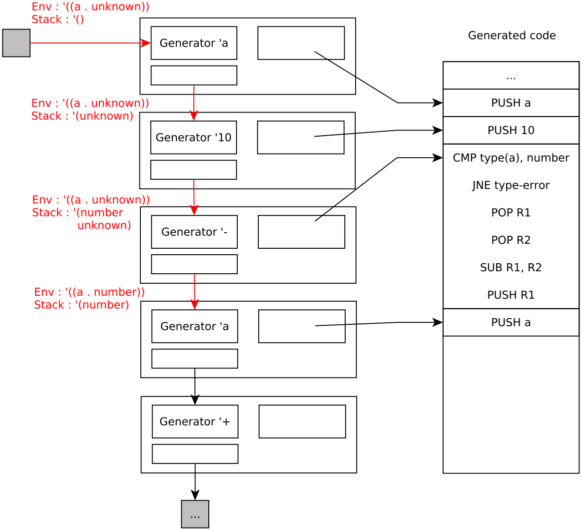
Example : (+ (- a 10) a)



Example : (+ (- a 10) a)



Example : (+ (- a 10) a)



Env : '((a . unknown))
Stack : '()

Generator 'a

Env : '((a . unknown))
Stack : '(unknown)

Generator '10

Env : '((a . unknown))
Stack : '(number
unknown)

Generator '-

Env : '((a . number))
Stack : '(number)

Generator 'a

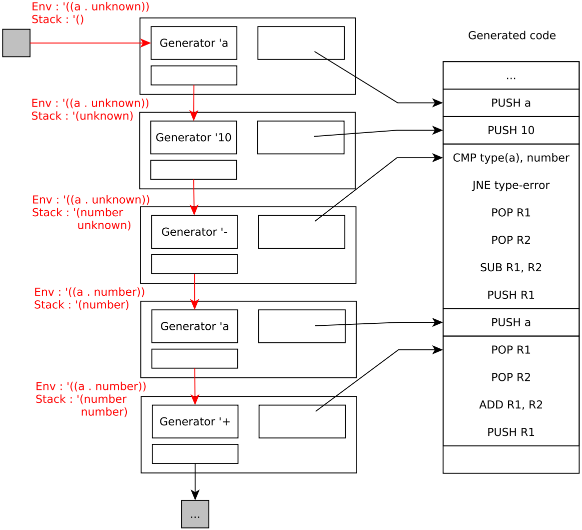
Generator '+

...

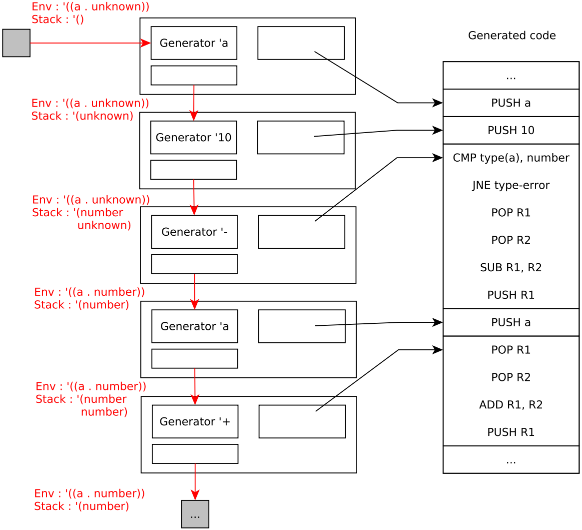
Generated code

```
...
PUSH a
PUSH 10
CMP type(a), number
JNE type-error
POP R1
POP R2
SUB R1, R2
PUSH R1
PUSH a
```

Example : (+ (- a 10) a)



Example : (+ (- a 10) a)



Env : '((a . unknown))
Stack : '()

Generator 'a

Env : '((a . unknown))
Stack : '(unknown)

Generator '10

Env : '((a . unknown))
Stack : '(number
unknown)

Generator '-

Env : '((a . number))
Stack : '(number)

Generator 'a

Env : '((a . number))
Stack : '(number
number)

Generator '+

Env : '((a . number))
Stack : '(number)

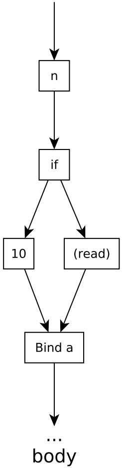
...

Generated code

```
...
PUSH a
PUSH 10
CMP type(a), number
JNE type-error
POP R1
POP R2
SUB R1, R2
PUSH R1
PUSH a
POP R1
POP R2
ADD R1, R2
PUSH R1
...
```

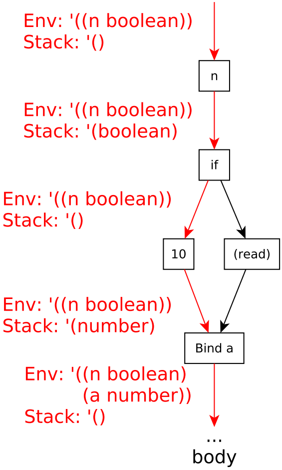
Problem - How to handle join points ?

```
(let ((a (if b
             10
             (read))))
  (+ a 100))
```



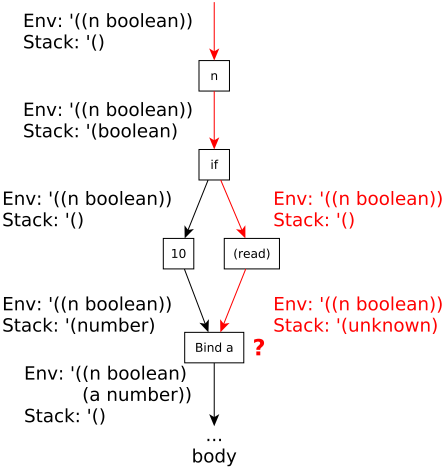
Problem - How to handle join points ?

```
(let ((a (if b
             10
             (read)))
  (+ a 100))
```



Problem - How to handle join points ?

```
(let ((a (if b
            10
            (read)))
   (+ a 100))
```



Env: '((n boolean))
Stack: '()

Env: '((n boolean))
Stack: '(boolean)

Env: '((n boolean))
Stack: '()

Env: '((n boolean))
Stack: '()

Env: '((n boolean))
Stack: '(number)

Env: '((n boolean))
Stack: '(unknown)

Env: '((n boolean)
(a number))
Stack: '()

...
body

Concept

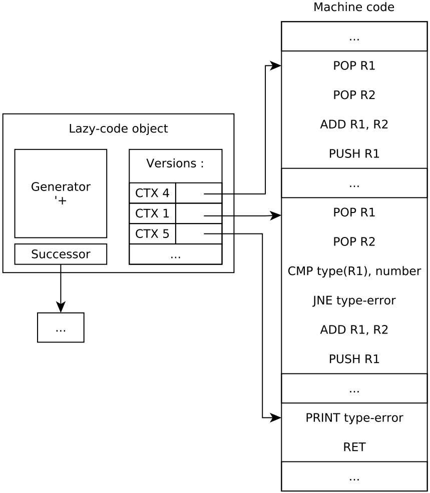
- Each lazy code object has multiple *versions*
- Each version associated to compilation context
- Each piece of code now has multiple entry points

Lazy code object

- Code generator
- Successor object
- Context→Address table

Lazy code object

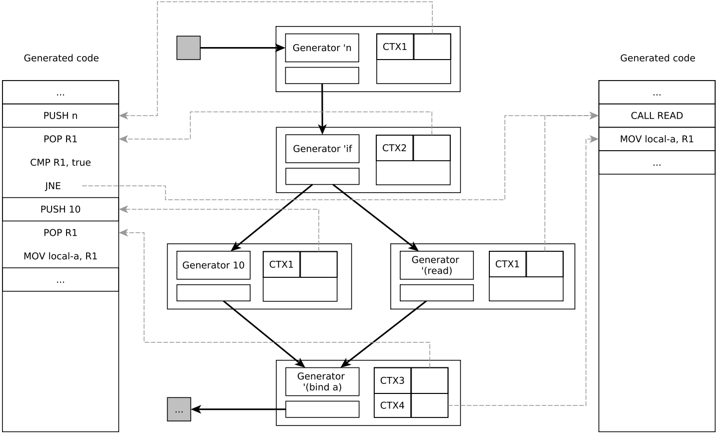
- Code generator
- Successor object
- Context→Address table



- CTX4 = '(number number)
- CTX1 = '(number unknown)
- CTX5 = '(number string)

Complete example with join point

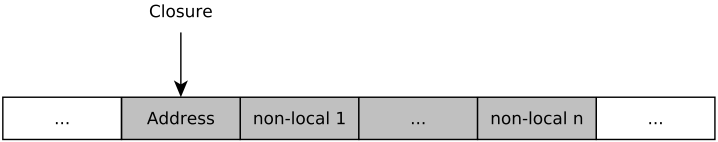
```
(let ((a (if b
             10
             (read))))
  (+ a 100))
```



Problem 1

- Functions also have multiple entry points
- Flat closure representation is not suitable

→ New closure representation (cc-table)



Closure

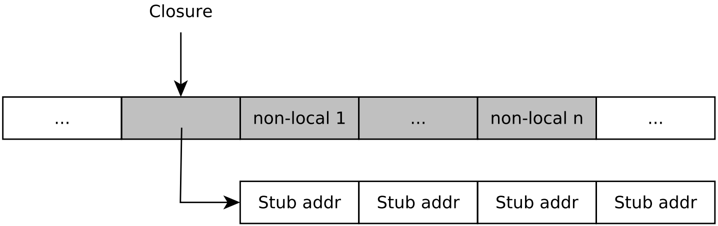
...	Address	non-local 1	...	non-local n	...

- Add indirection

Problem 1

- Functions also have multiple entry points
- Flat closure representation is not suitable

→ New closure representation (cc-table)

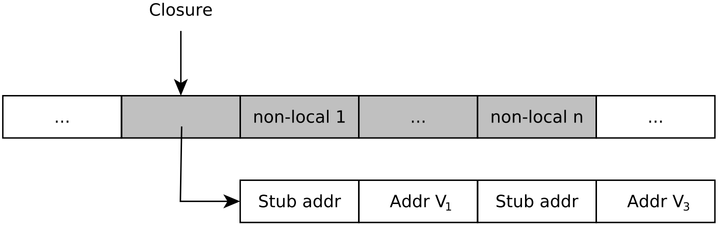


- Add indirection

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 ?

→ Keep a global cc-table

	Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	
Global table	CTX4	CTX1	CTX8	CTX9	CTX3	CTX10	...
λ1 table	STUB λ1	ADDR X1	STUB λ1	STUB λ1	ADDR X2	STUB λ1	...
λ2 table	ADDR Y1	ADDR Y2	ADDR Y3	STUB λ2	STUB λ2	STUB λ2	...
...							

- Possible combinatory explosion

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