Type Check Removal Using Lazy Interprocedural Code Versioning

Baptiste Saleil
baptiste.saleil@umontreal.ca

Marc Feeley
feeley@iro.umontreal.ca

September 4, 2015

Université de Montréal

Overview

- Dynamically typed language
  - Safety at runtime
  - Type checks → performance issue

- Just-In-Time compiler
  - Avoid static analysis if possible
  - Basic Block Versioning
    - → Remove type checks
    - → No analysis nor profiling
    - → On-the-fly code duplication

Contributions

* Simplify the compilation process
  - → Accelerate implementation
  - → Limiting the number of IR and analysis

** Extending Basic Block Versioning
  - → Allow interprocedural propagation
  - → Extend the use of BBV

What is Basic Block Versioning ?

Basic Block Versioning

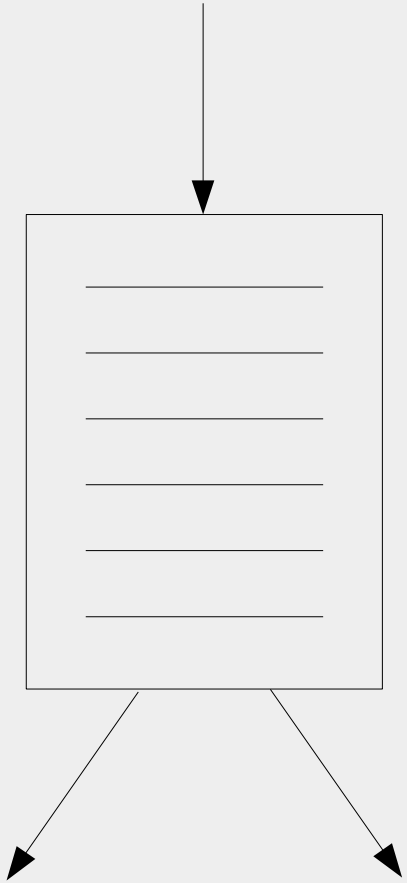
- JIT compilation technique

  Maxime Chevalier-Boisvert, Marc Feeley, ECOOP 2015

- Generate specialized versions on-the-fly
  - → Possibly several versions
  - → Only executed versions are generated
- Specialized using runtime information
  - → Example: Typing information
- Efficient at removing dynamic type checks
  - → JavaScript

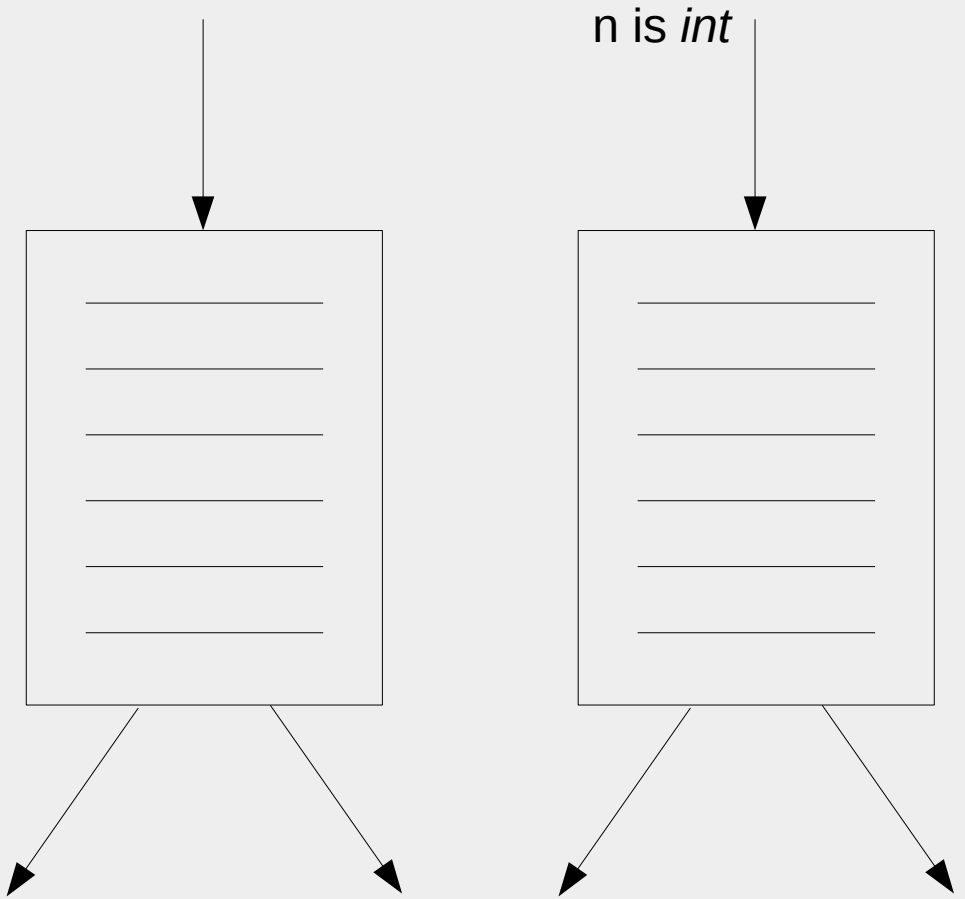
Basic Block Versioning

- Basic block
  - Instructions sequence
  - One entry point
  - One exit point



Basic Block Versioning

- Basic block
  - Instructions sequence
  - One entry point
  - One exit point



Basic Block Versioning

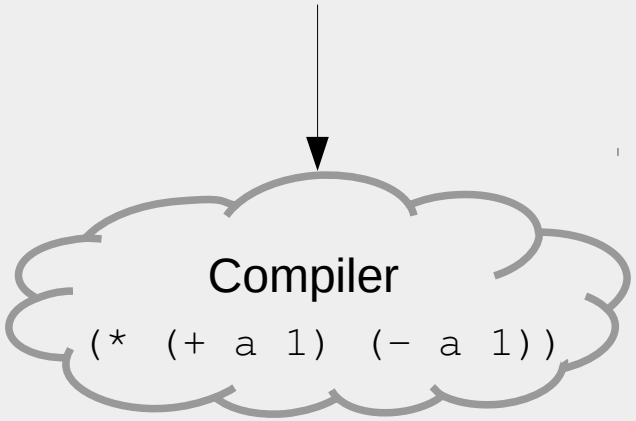
- Exemple

```
(* (+ a 1) (- a 1))
...
```

Basic Block Versioning

- Exemple

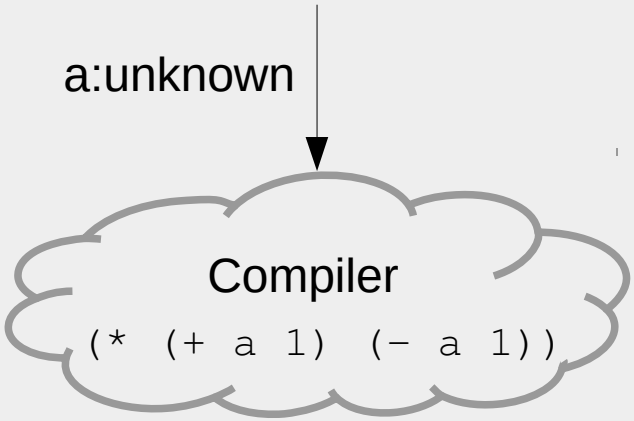
```
(* (+ a 1) (- a 1))

...
```



Basic Block Versioning

- Exemple

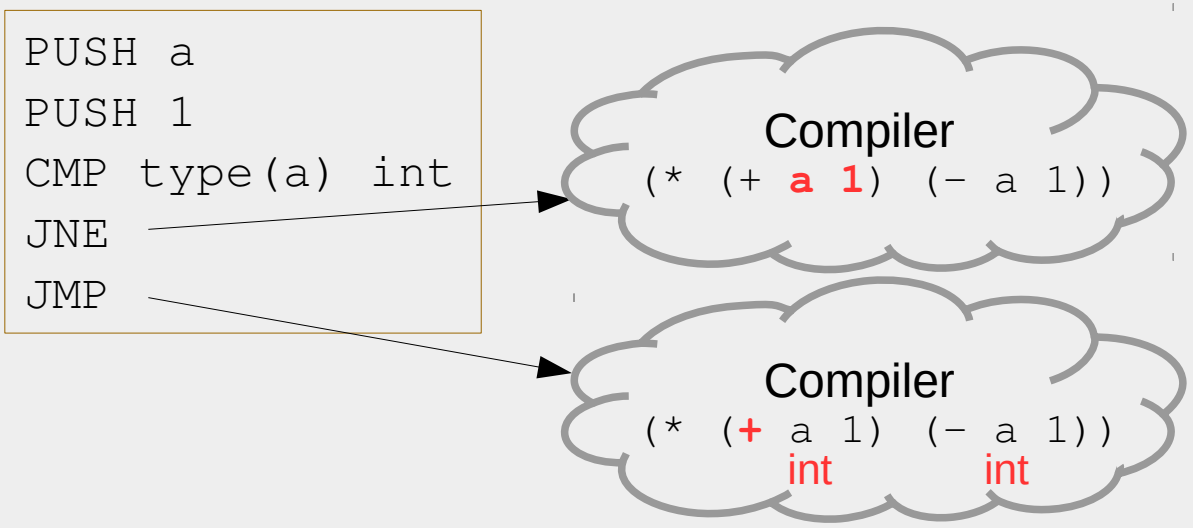
```
(* (+ a 1) (- a 1))

...
```



Basic Block Versioning

- Exemple `(* (+ a 1) (- a 1))`

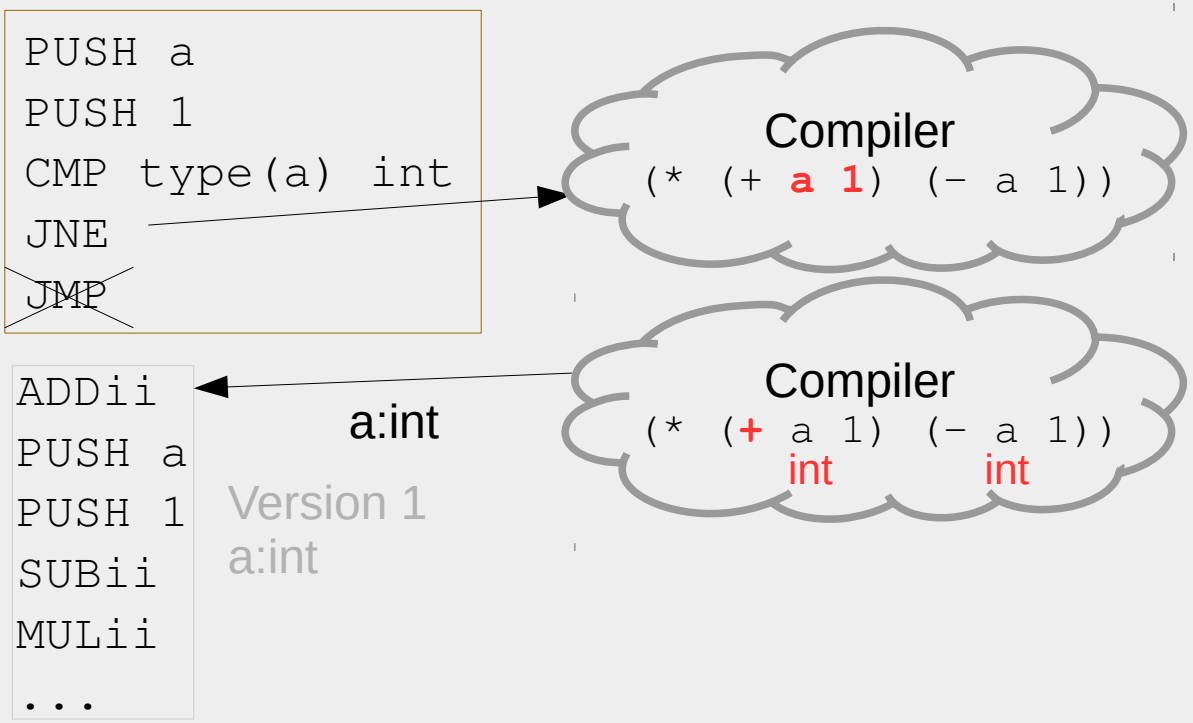
`...`



Basic Block Versioning

- Exemple `(* (+ a 1) (- a 1))`

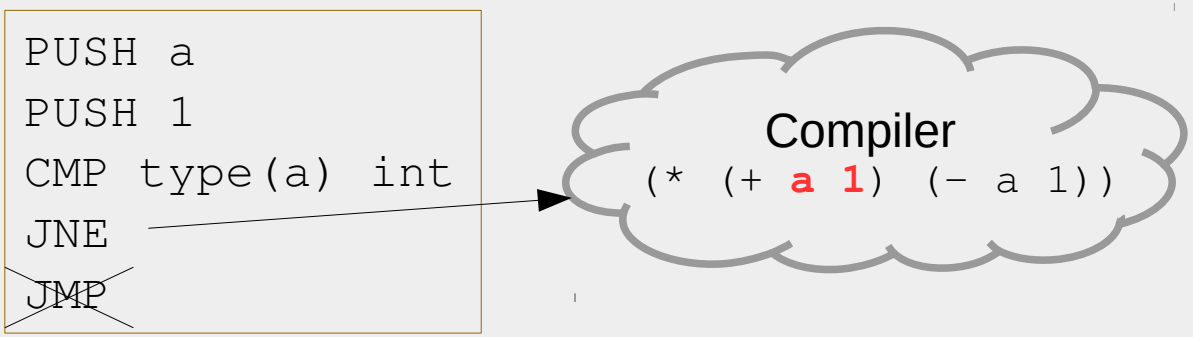
...



Basic Block Versioning

- Exemple `(* (+ a 1) (- a 1))`

`...`



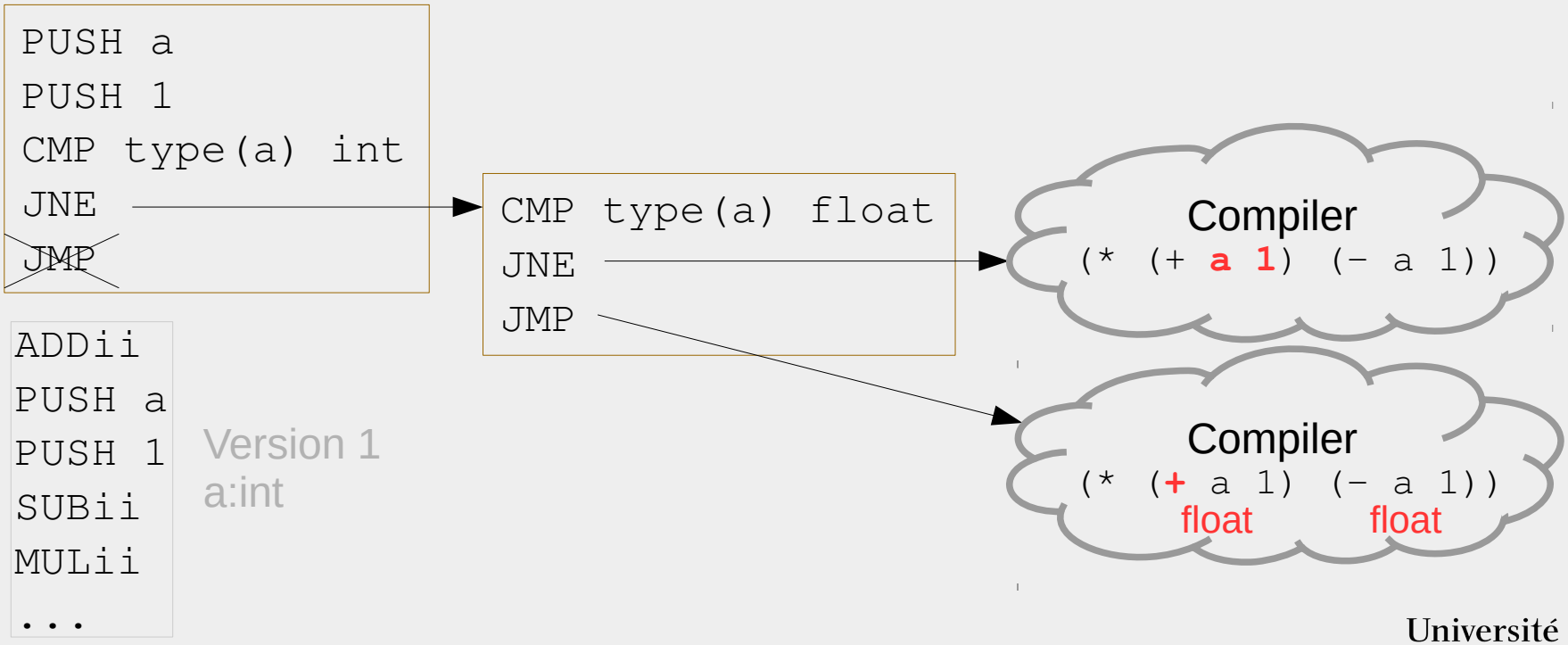

```
ADDii
PUSH a
PUSH 1
SUBii
MULii
...
```

Version 1
a:int

Basic Block Versioning

- Exemple `(* (+ a 1) (- a 1))`

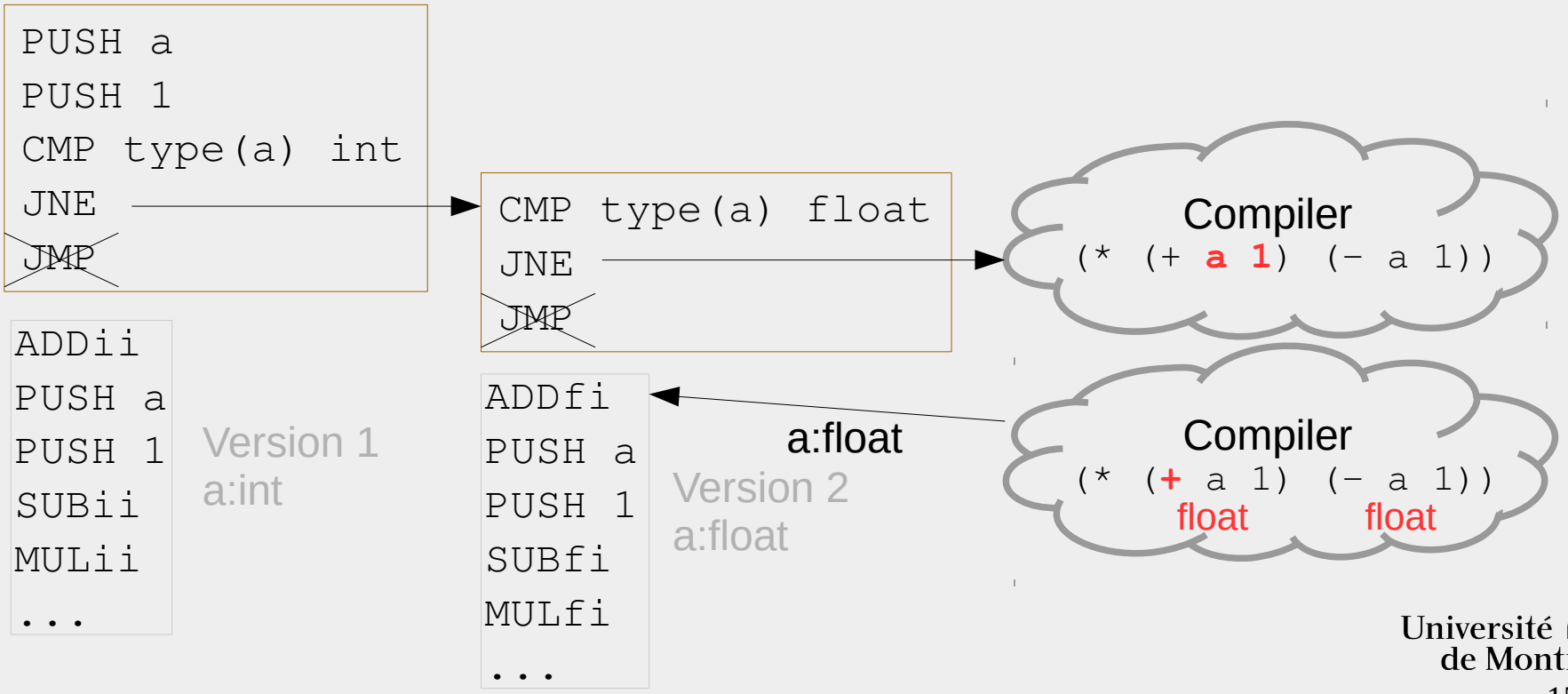
`...`



Basic Block Versioning

- Exemple `(* (+ a 1) (- a 1))`

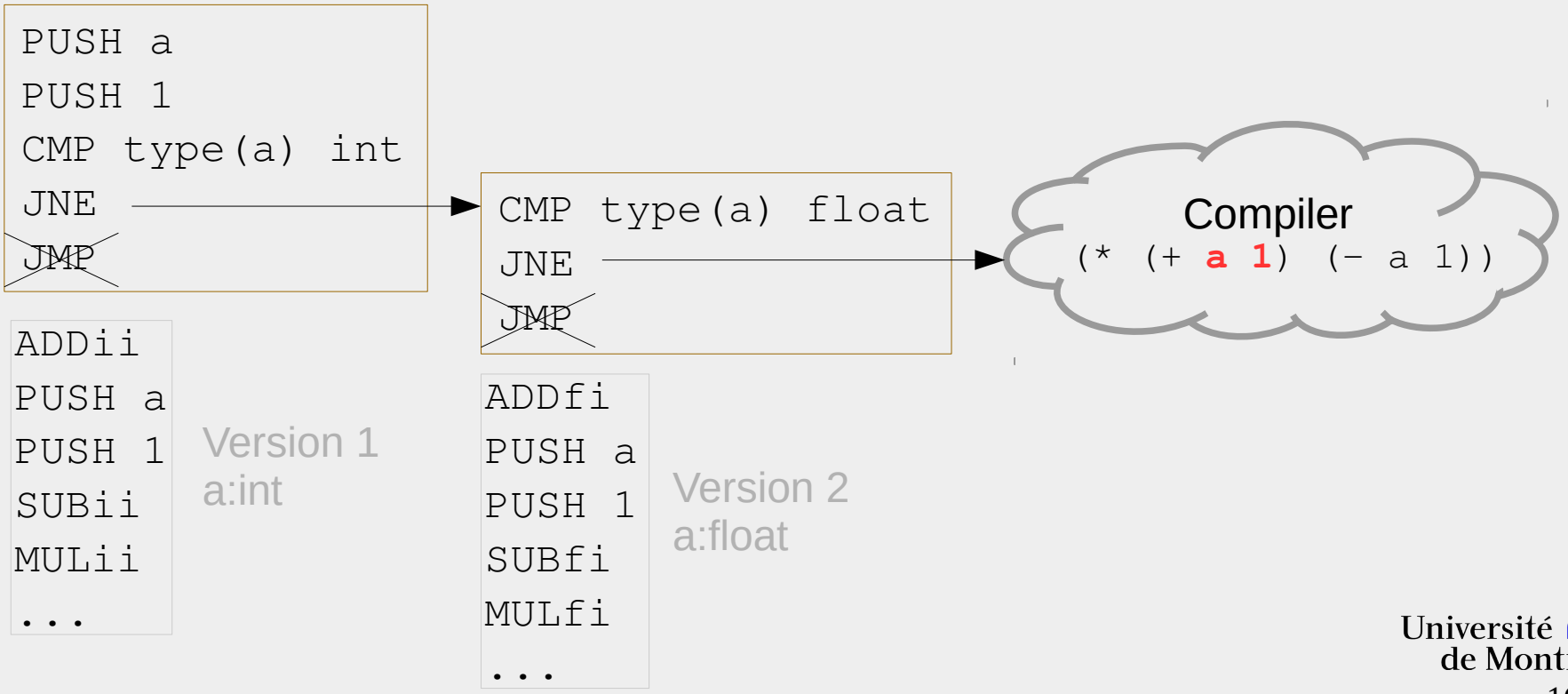
`...`



Basic Block Versioning

- Exemple `(* (+ a 1) (- a 1))`

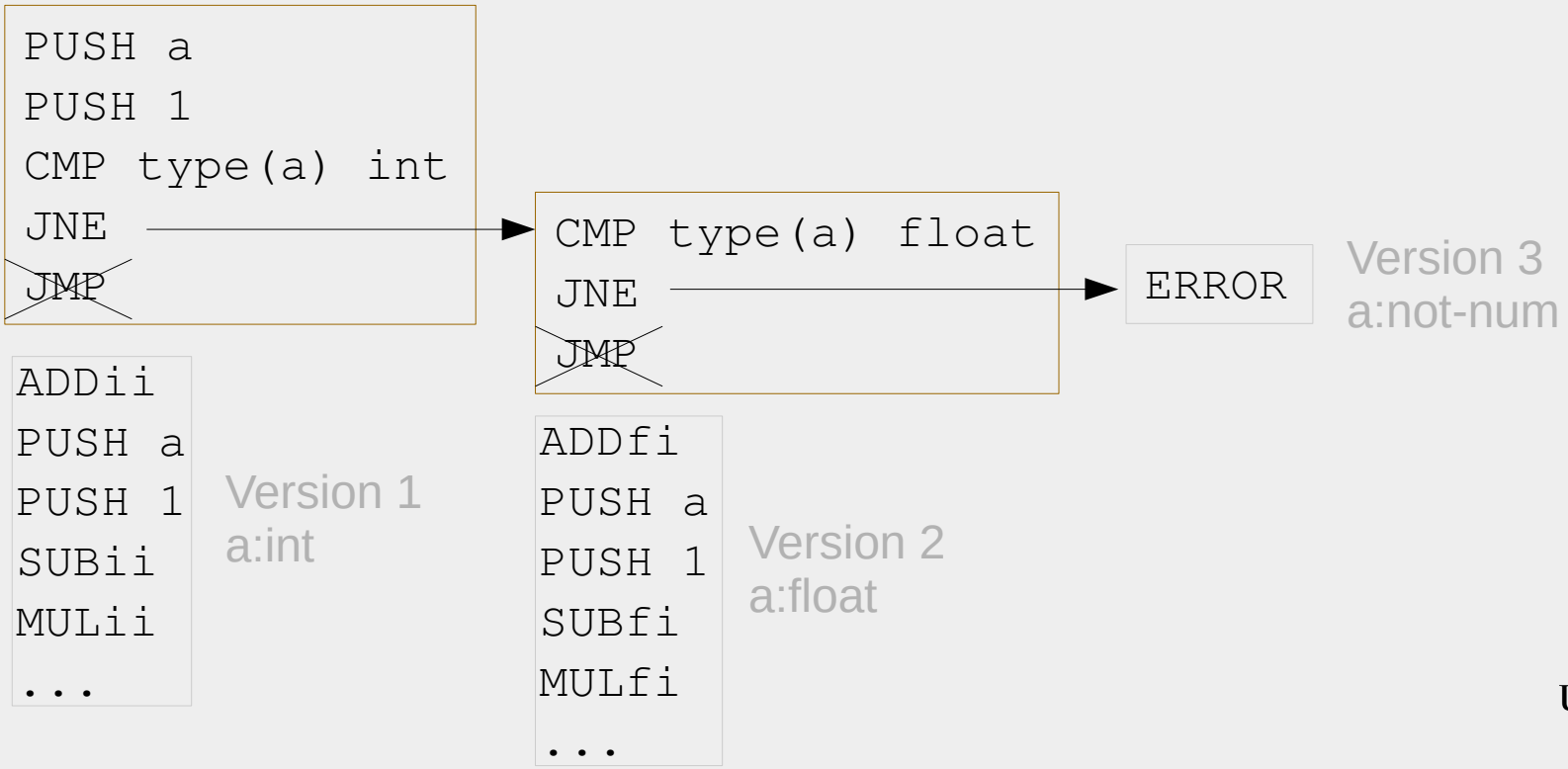
...



Basic Block Versioning

- Exemple `(* (+ a 1) (- a 1))`

`...`



* Extremely lazy compilation

Extremely lazy compilation

- Simplifying the compilation process
  - sexprs → compiler stubs
- Lazy code object
  - Generator: context → basic block version

    (make-lazy-code-object (lambda (ctx) ...))
  - Versions table: context → basic block version
  - Successor lazy code object

    (jump-to successor ctx)
- The compilation discovers new information
- Implementation
  - Stack machine
  - 0 analysis / IR

Extremely lazy compilation

- Example

```
...
(let ((c (integer->char n)))
  ...)
```

Extremely lazy compilation

- Example

```
gen-chain : sexpr x lazy-code-object → lazy-code-object
```

```
(define (gen-chain ast successor)
  (cond
    ...
    ((integer? ast)

     (make-lazy-code-object
       (lambda (ctx) ; Generator
         (x86-push ast)
         (jump-to successor (ctx-push ctx CTX_INT)))))

    ...
```

Extremely lazy compilation

- Example

```
gen-chain : sexpr x lazy-code-object → lazy-code-object
```

```
(define (gen-chain ast successor)
  (cond
    ...
    ((integer? ast)

     (make-lazy-code-object
       (lambda (ctx) ; Generator
         (x86-push ast)
         (jump-to successor (ctx-push ctx CTX_INT)))))

    ...
```

Extremely lazy compilation

- Example

```
gen-chain : sexpr x lazy-code-object → lazy-code-object
```

```
(define (gen-chain ast successor)
  (cond
    ...
    ((integer? ast)

     (make-lazy-code-object
       (lambda (ctx) ; Generator
         (x86-push ast)
         (jump-to successor (ctx-push ctx CTX_INT)))))

    ...
```

Extremely lazy compilation

- Example

```
gen-chain : sexpr x lazy-code-object → lazy-code-object
```

```
(define (gen-chain ast successor)
  (cond
    ...
    ((integer? ast)

     (make-lazy-code-object
       (lambda (ctx) ; Generator
         (x86-push ast)
         (jump-to successor (ctx-push ctx CTX_INT)))))

    ...
```

Extremely lazy compilation

```
  ((eq? (car ast) 'integer->char)

   (let* ((lazy-conv
           (make-lazy-code-object
             (lambda (ctx) ; Generator
               (x86-pop rax)
               (x86-to-char rax)
               (x86-push rax)
               (jump-to successor (ctx-push (ctx-pop ctx) CTX_CHAR)))))
          (lazy-check
           (make-lazy-code-object
             (lambda (ctx) ; Generator
               (x86-pop rax)
               (x86-cmp tag_rax TAG_INT)
               (x86-jne label-error)
               (x86-push rax)
               (jump-to lazy-conv (ctx-push (ctx-pop ctx) CTX_INT))))))
     (gen-chain
       (cadr ast)
       (make-lazy-code-object
         (lambda (ctx) ; Generator
           (let ((type (type-top ctx)))
             (cond ((eq? type CTX_INT) (jump-to lazy-conv ctx))
                   ((eq? type CTX_UNK) (jump-to lazy-check ctx))
                   (else (jump-to (gen-error-object) ctx)))))))))
  ...
))
```

Extremely lazy compilation

```
  ((eq? (car ast) 'integer->char)

   (let* ((lazy-conv
           (make-lazy-code-object
             (lambda (ctx) ; Generator
               (x86-pop rax)
               (x86-to-char rax)
               (x86-push rax)
               (jump-to successor (ctx-push (ctx-pop ctx) CTX_CHAR))))
          (lazy-check
            (make-lazy-code-object
              (lambda (ctx) ; Generator
                (x86-pop rax)
                (x86-cmp tag_rax TAG_INT)
                (x86-jne label-error)
                (x86-push rax)
                (jump-to lazy-conv (ctx-push (ctx-pop ctx) CTX_INT))))))
     (gen-chain
       (cadr ast)
       (make-lazy-code-object
         (lambda (ctx) ; Generator
           (let ((type (type-top ctx)))
             (cond ((eq? type CTX_INT) (jump-to lazy-conv ctx))
                   ((eq? type CTX_UNK) (jump-to lazy-check ctx))
                   (else (jump-to (gen-error-object) ctx)))))))))
   ...
))
```

Extremely lazy compilation

```
  ((eq? (car ast) 'integer->char)

   (let* ((lazy-conv
           (make-lazy-code-object
             (lambda (ctx) ; Generator
               (x86-pop rax)
               (x86-to-char rax)
               (x86-push rax)
               (jump-to successor (ctx-push (ctx-pop ctx) CTX_CHAR))))))
          (lazy-check
            (make-lazy-code-object
              (lambda (ctx) ; Generator
                (x86-pop rax)
                (x86-cmp tag_rax TAG_INT)
                (x86-jne label-error)
                (x86-push rax)
                (jump-to lazy-conv (ctx-push (ctx-pop ctx) CTX_INT))))))
     (gen-chain
       (cadr ast)
       (make-lazy-code-object
         (lambda (ctx) ; Generator
           (let ((type (type-top ctx)))
             (cond ((eq? type CTX_INT) (jump-to lazy-conv ctx))
                   ((eq? type CTX_UNK) (jump-to lazy-check ctx))
                   (else (jump-to (gen-error-object) ctx)))))))))
  ...
))
```

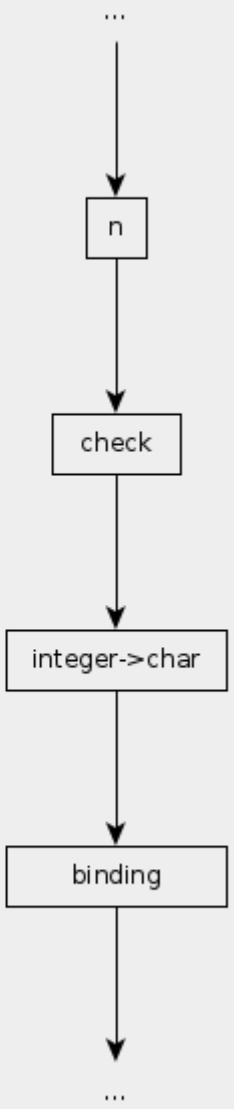
Extremely lazy compilation

```
  ((eq? (car ast) 'integer->char)

   (let* ((lazy-conv
           (make-lazy-code-object
             (lambda (ctx) ; Generator
               (x86-pop rax)
               (x86-to-char rax)
               (x86-push rax)
               (jump-to successor (ctx-push (ctx-pop ctx) CTX_CHAR)))))
          (lazy-check
           (make-lazy-code-object
             (lambda (ctx) ; Generator
               (x86-pop rax)
               (x86-cmp tag_rax TAG_INT)
               (x86-jne label-error)
               (x86-push rax)
               (jump-to lazy-conv (ctx-push (ctx-pop ctx) CTX_INT))))))
     (gen-chain
       (cadr ast)
       (make-lazy-code-object
         (lambda (ctx) ; Generator
           (let ((type (type-top ctx)))
             (cond ((eq? type CTX_INT) (jump-to lazy-conv ctx))
                   ((eq? type CTX_UNK) (jump-to lazy-check ctx))
                   (else (jump-to (gen-error-object) ctx)))))))))
  ...
))
```

Extremely lazy compilation

Example

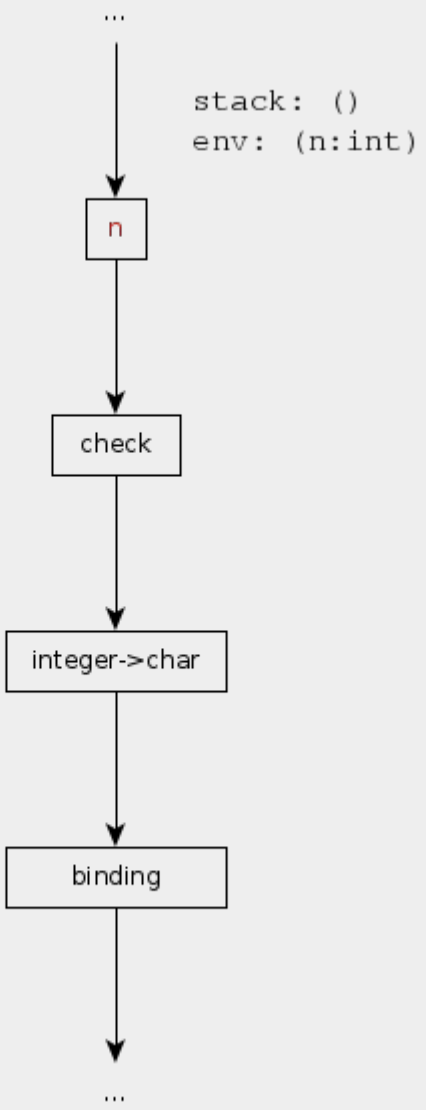
```
...
(let ((c (integer->char n)))
  ...)
```



Extremely lazy compilation

Example

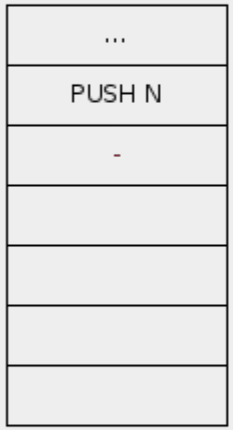
```
...
(let ((c (integer->char n)))
  ...)
```

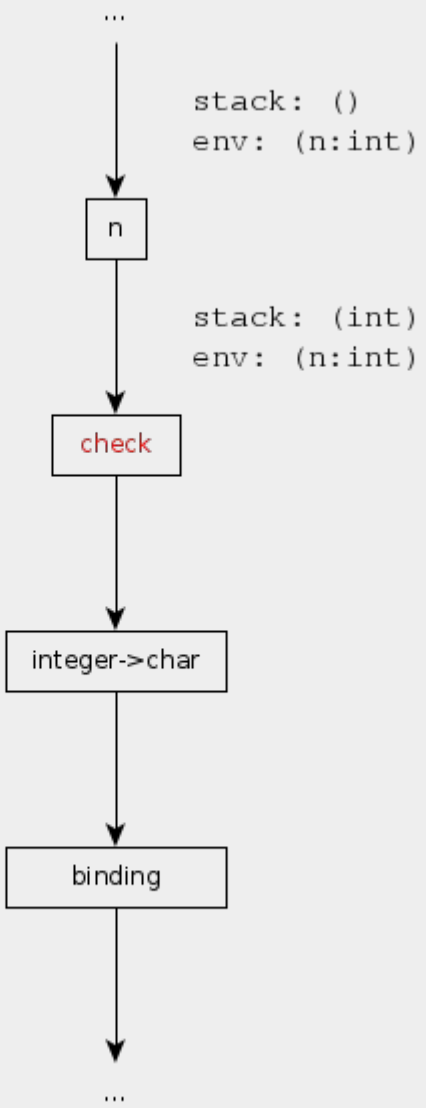


Extremely lazy compilation

Example

```
...
(let ((c (integer->char n)))
  ...)
```

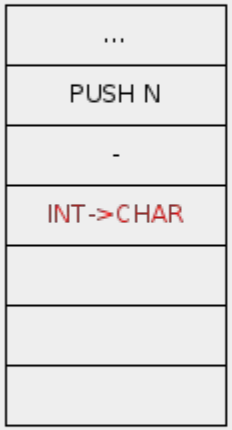


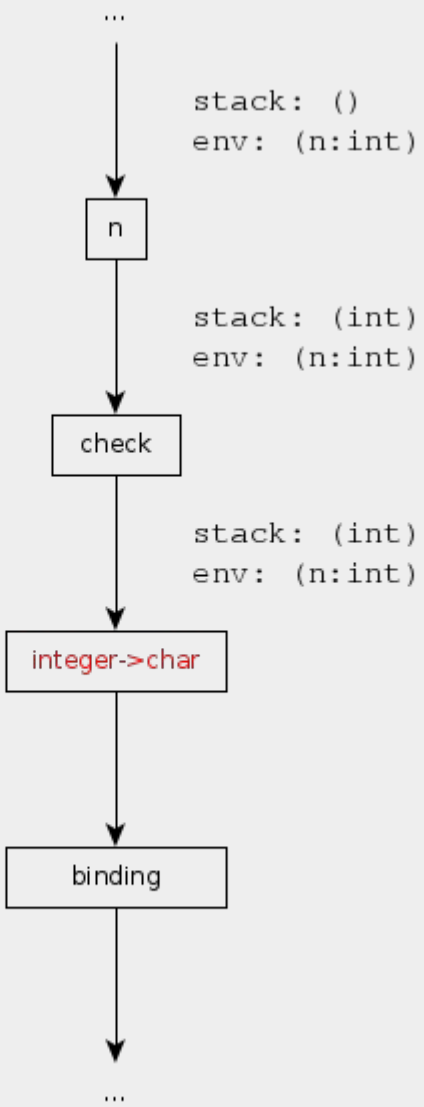


Extremely lazy compilation

Example

```
...
(let ((c (integer->char n)))
  ...)
```

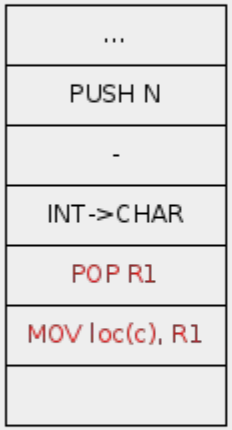


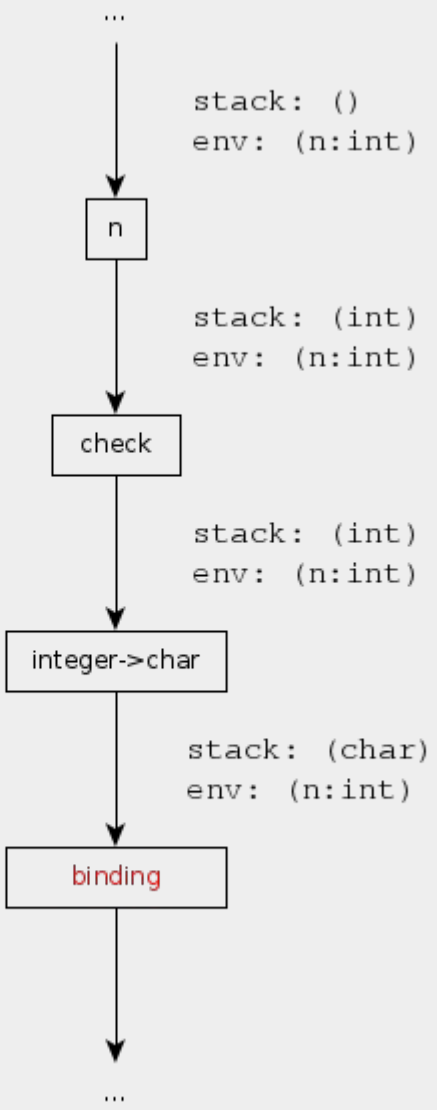


Extremely lazy compilation

Example

```
...
(let ((c (integer->char n)))
  ...)
```

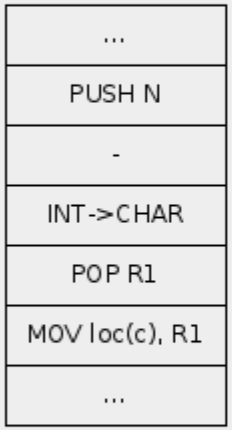


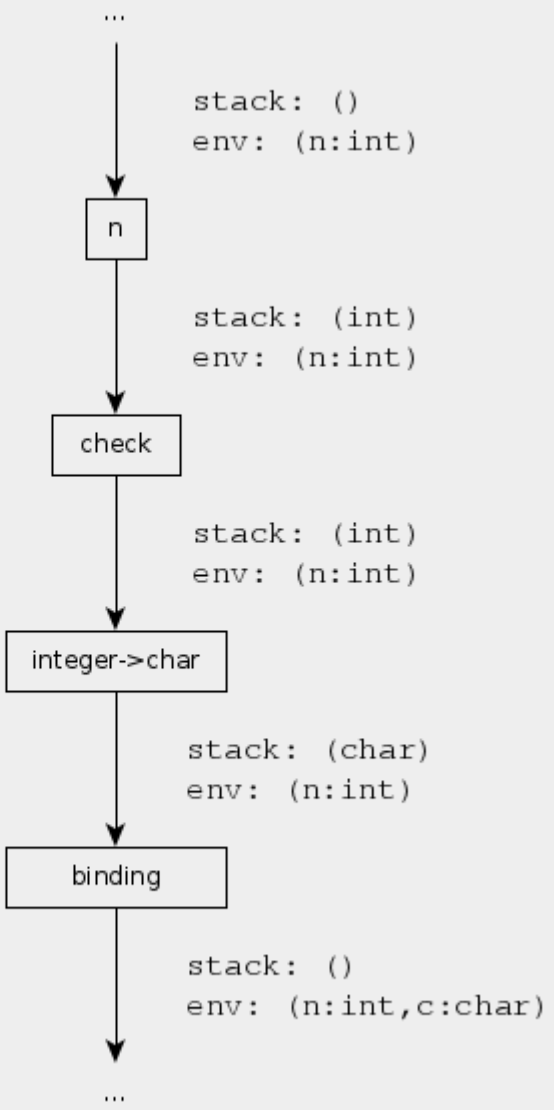


Extremely lazy compilation

Example

```
...
(let ((c (integer->char n)))
  ...)
```





- Still able to use BBV
- No more control flow instructions

** Interprocedural propagation

Interprocedural propagation

Caller → Callee

```
(define (make-adder n)
  (lambda (x) (+ n x)))

(let* ((add1 (make-adder 1))
       (x    (read))
       (y    (+ 3.14 x)))
  (* (add1 x) (add1 y)))
```

Interprocedural propagation

Caller → Callee

```
(define (make-adder n)
  (lambda (x) (+ n x)))

(let* ((add1 (make-adder 1))
       (x    (read))
       (y    (+ 3.14 x)))
  (* (add1 x) (add1 y)))
```

Interprocedural propagation

Caller → Callee

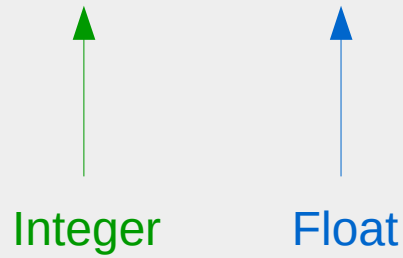
```
(define (make-adder n)
  (lambda (x) (+ n x)))

(let* ((add1 (make-adder 1))
       (x    (read))
       (y    (+ 3.14 x)))
  (* (add1 x) (add1 y)))
```

Integer

Interprocedural propagation

Caller → Callee

```
(define (make-adder n)
  (lambda (x) (+ n x)))

(let* ((add1 (make-adder 1))
       (x    (read))
       (y    (+ 3.14 x)))
  (* (add1 x) (add1 y)))
```



Interprocedural propagation

Caller → Callee

```
(define (make-adder n)
  (lambda (x) (+ n x)))

(let* ((add1 (make-adder 1))
       (x    (read))
       (y    (+ 3.14 x)))
  (* (add1 x) (add1 y)))
```

Integer

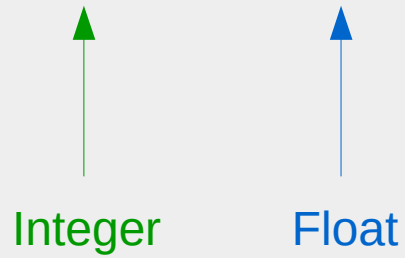
Float

Specialize entry point !

Interprocedural propagation

Caller → Callee

```
(define (make-adder n)
  (lambda (x) (+ n x)))

(let* ((add1 (make-adder 1))
       (x    (read))
       (y    (+ 3.14 x)))
  (* (add1 x) (add1 y)))
```



Specialize entry point !

→ Must keep several entry points per closure

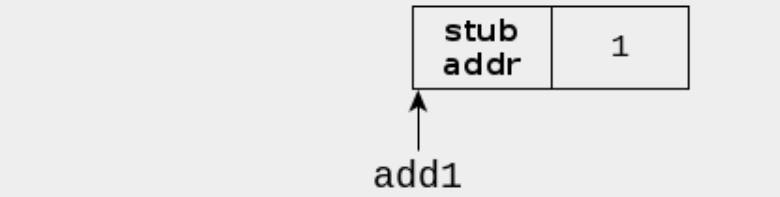
Interprocedural propagation

Caller → Callee

- Propagate discovered type information to the callee
- Specialize entry points
  - → Several entry points
  - → Extend closure representation

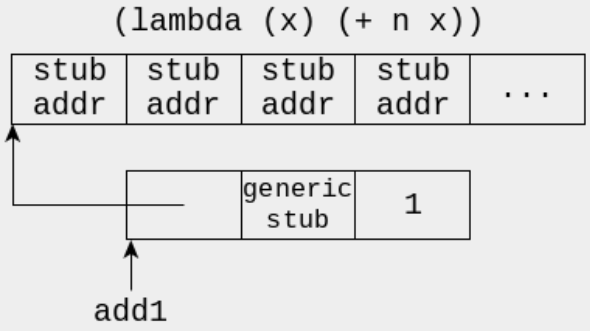
Interprocedural propagation

```
(lambda (x) (+ n x))
```

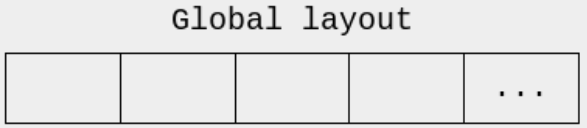


Interprocedural propagation

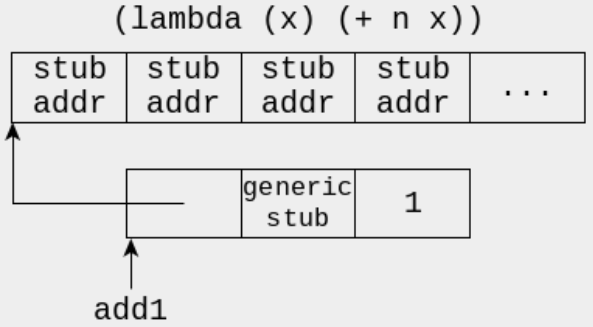
- Entry points table
  → Shared by all instances



Interprocedural propagation

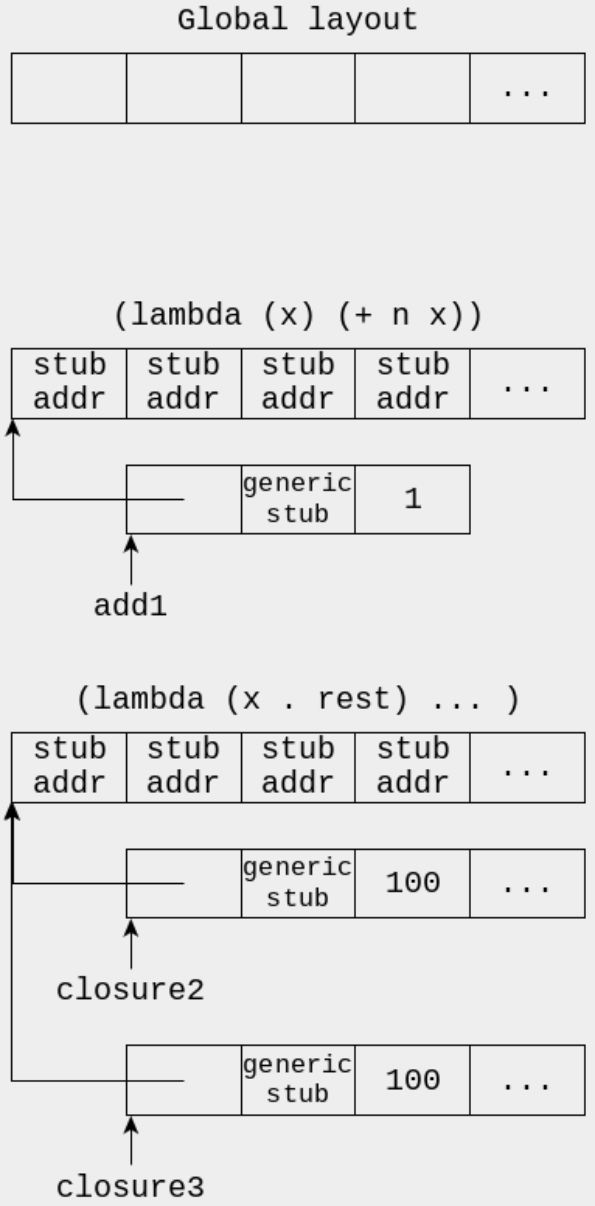


- Entry points table
  → Shared by all instances
- Global layout



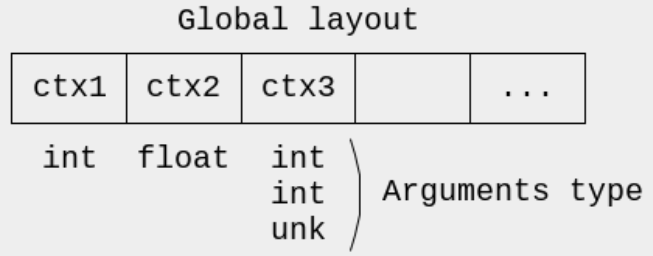
Interprocedural propagation

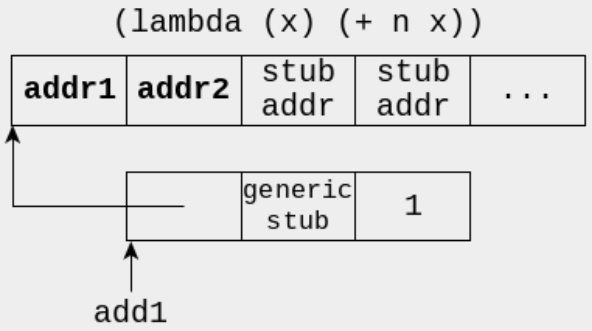
- Entry points table
  → Shared by all instances
- Global layout
  → Shared by all tables

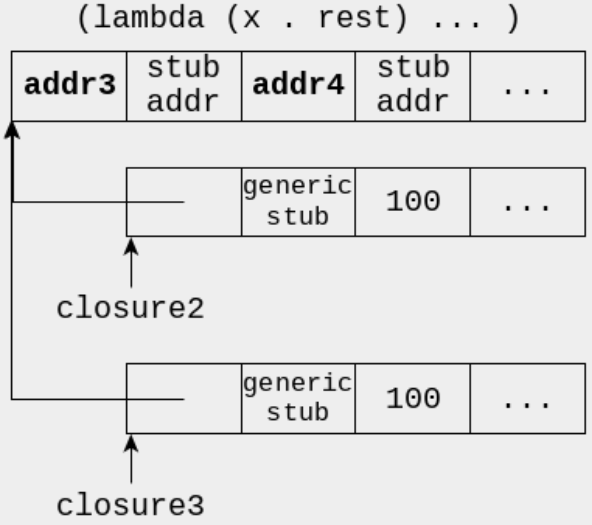


Interprocedural propagation

- Entry points table
  → Shared by all instances
- Global layout
  → Shared by all tables

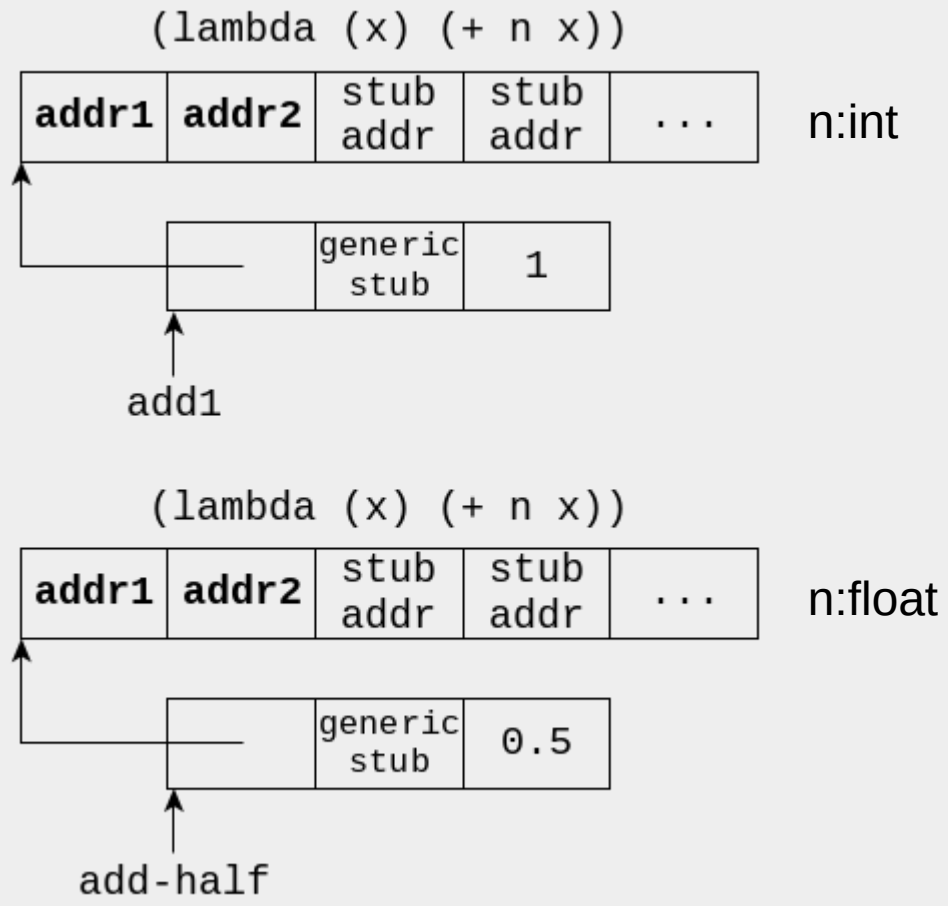






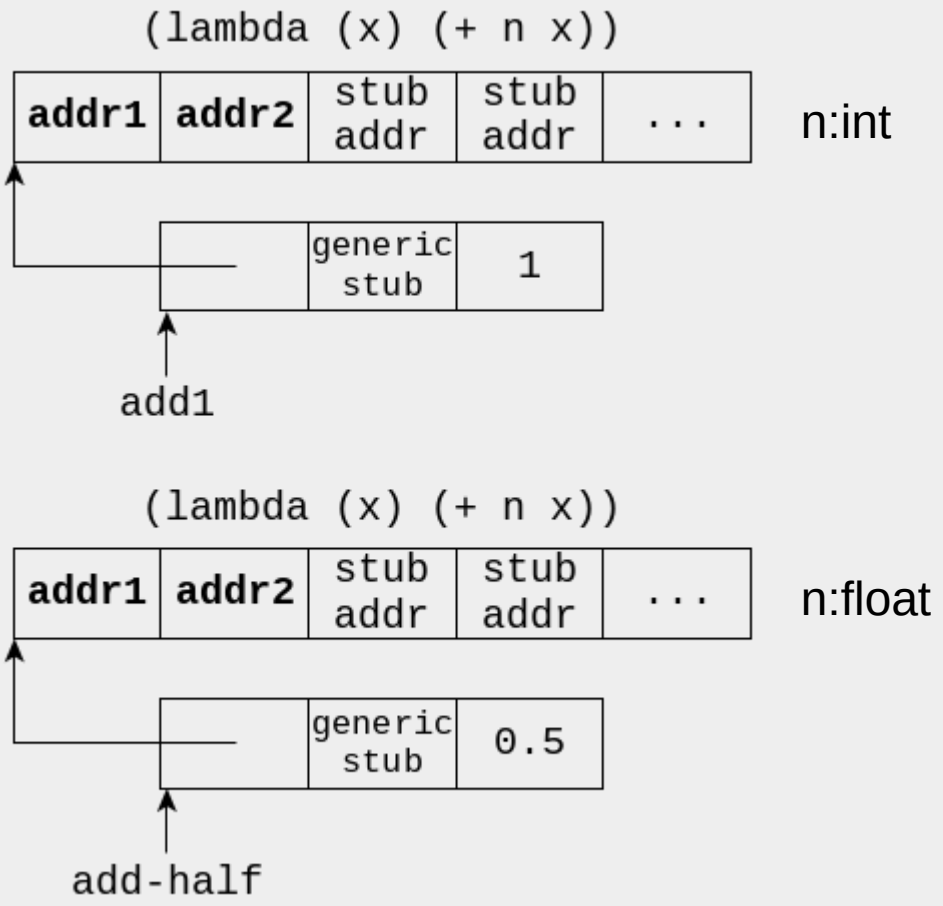
Interprocedural propagation

- Entry points table
  → Shared by all instances
- Global layout
  → Shared by all tables
- Multiple tables per lambda
  → For each combination of free variables types



Interprocedural propagation

- Entry points table
  → Shared by all instances
- Global layout
  → Shared by all tables
- Multiple tables per lambda
  → For each combination of free variables types



Memory overhead for the entry points tables ?

Interprocedural extension

- Benchmarks !

  - 110 functions in the library
  - Types: *int, float*, char, bool, procedure, pair, void, null, vector, string, symbol, port*
  - Max: 2.8 mb
  - Typically ≤ 64 kb

Benchmark	Lines of code	Number of tables	Total tables size (*kb*)
compiler	11195	1561	2847
earley	647	187	64
conform	454	208	47
graphs	598	161	43
mazefun	202	149	37
peval	629	187	31
sboyer	778	149	23
browse	187	128	16
paraffins	172	133	14
boyer	565	134	13
nqueens	30	117	12
dderiv	74	121	8
string	24	113	5
deriv	34	112	4
destruc	45	113	4
perm9	97	117	4
triangl	54	112	4
array1	25	115	3
cpstak	24	116	3
primes	26	114	3
tak	10	111	3
ack	7	111	2
divrec	15	112	2
sum	8	112	2
cat	19	112	<1
diviter	16	112	<1
fib	8	111	<1
sumloop	22	113	<1
takl	26	113	<1
wc	38	112	<1

What about return points ?

Interprocedural propagation

Caller ← Callee

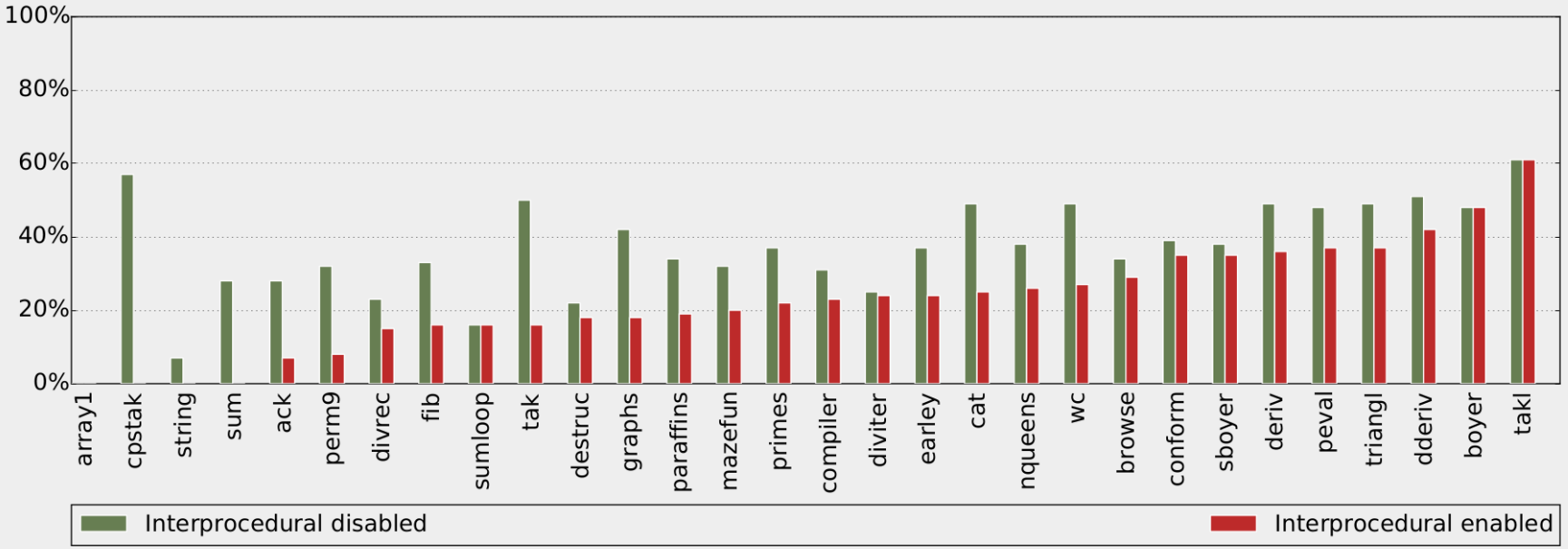
- Propagate discovered type information to the caller

- Specialize continuations
  - → “return point = entry point”
  - → Similar implementation
  - → CPS !

Evaluation

Evaluation

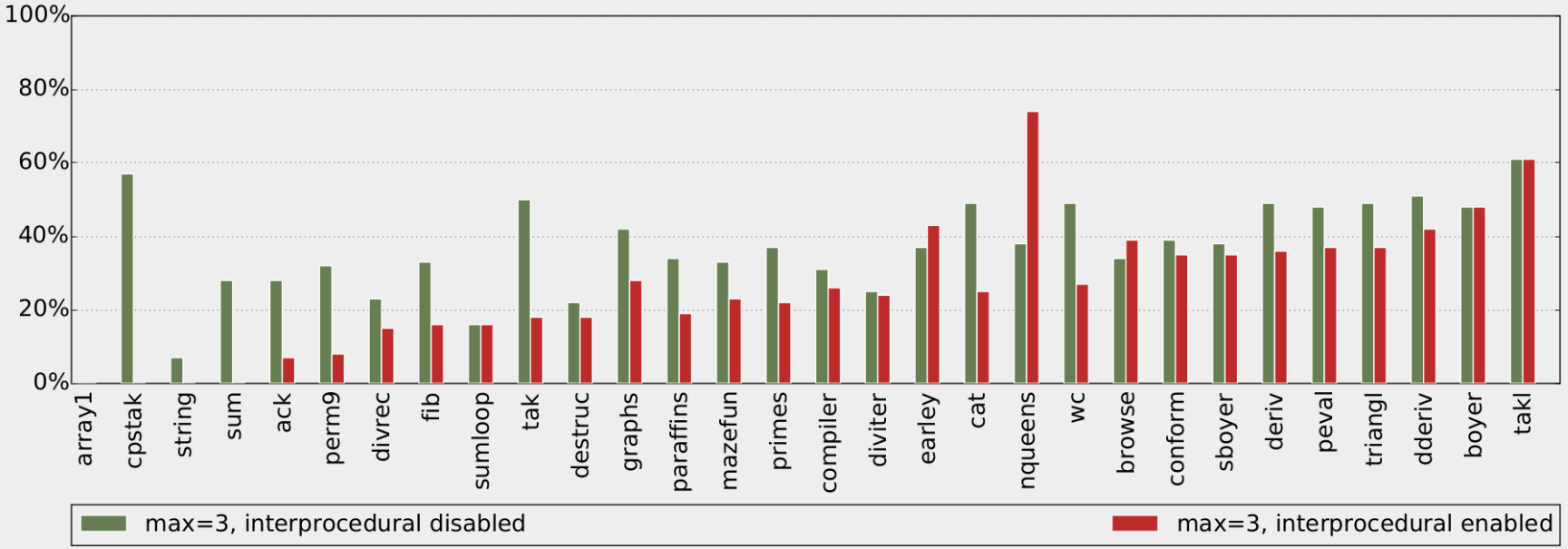
- Type checks executed (relative to generic version)



- ~64% removed without interprocedural propagation
- ~77% removed with interprocedural propagation
- Limiting the number of versions to 5 does not change the results

Evaluation

- Type checks executed (relative to generic version)



- With a version limit of 3, some versions are wasted

Evaluation

- Propagation to return points

```
(define (fibcps n k)
  (if (< n 2)
      (k n)
      (fibcps (- n 1)
              (lambda (r1)
                (fibcps (- n 2)
                        (lambda (r2)
                          (k (+ r1 r2)))))))))
(define (fib n)
  (fibcps n (lambda (r) r)))
```

Evaluation

- Propagation to return points

```
(define (fibcps n k)
  (if (< n 2)
      (k n)
      (fibcps (- n 1)
              (lambda (r1)
                (fibcps (- n 2)
                        (lambda (r2)
                          (k (+ r1 r2)))))))))
(define (fib n)
  (fibcps n (lambda (r) r)))
```

- 0 checks if n is *integer*

Evaluation

- Propagation to return points

```
(define (fibcps n k)
  (if (< n 2)
      (k n)
      (fibcps (- n 1)
              (lambda (r1)
                (fibcps (- n 2)
                        (lambda (r2)
                          (k (+ r1 r2)))))))))
(define (fib n)
  (fibcps n (lambda (r) r)))
```

- 0 checks if n is *integer*
- 1 check if n is *unknown*

Future work

PhD project

- Extend BBV
  - → Interprocedural propagation
- Study other ways to use BBV
  - → Allocation Sinking, Inlining, Register Allocation, ...
- Unify the compilation process

PhD project

- Extend BBV
  - → Interprocedural propagation ✓
- Study other ways to use BBV
  - → Allocation Sinking, Inlining, Register Allocation, ...
- Unify the compilation process

Thank you !