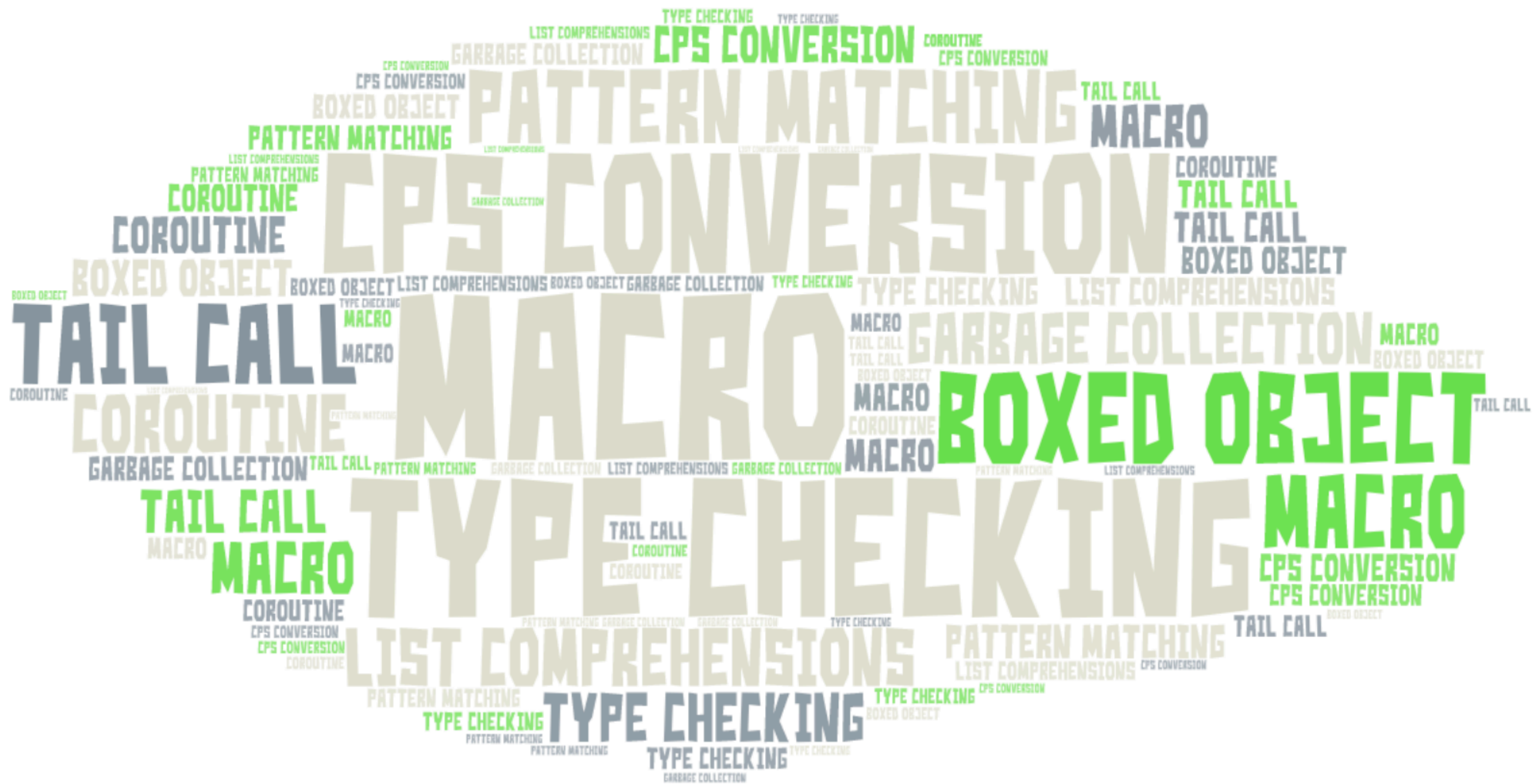
Closure conversion

powered by nanopass

Dannypsnl & 2022/07/31


TYPE CHECKING
LIST COMPREHENSIONS
GARBAGE COLLECTION
CPS CONVERSION
COROUTINE
CPS CONVERSION
PATTERN MATCHING
BOXED OBJECT
TAIL CALL
MACRO
PATTERN MATCHING
CPS CONVERSION
COROUTINE
TAIL CALL
COROUTINE
TAIL CALL
BOXED OBJECT
BOXED OBJECT
TYPE CHECKING
LIST COMPREHENSIONS
TAIL CALL
MACRO
GARBAGE COLLECTION
MACRO
BOXED OBJECT
COROUTINE
MACRO
GARBAGE COLLECTION
TYPE CHECKING
MACRO
TAIL CALL
MACRO
TAIL CALL
CPS CONVERSION
CPS CONVERSION
COROUTINE
TAIL CALL
LIST COMPREHENSIONS
PATTERN MATCHING
TYPE CHECKING
TYPE CHECKING
TYPE CHECKING
PATTERN MATCHING

Common thing?

Function as Value

```
let makeAdder = (n) => (m) => (n + m)

makeAdder(2)(3) // 5
```

C?

```c
int adder(int m) {
  return n + m;
}

typedef int (*adder_t)(int);
adder_t make_adder(int n) {
  return &adder;
}
```

Not just syntax

Pseudo C: address of n

```c
typedef int (*adder_t)(int);
adder_t make_adder(int n) {
  int adder(int m) {
    return n + m;
  }
  return &adder;
}
```

Not just syntax

Store in Global variable

```
int n1;
int adder(int m) {
  return n1 + m;
}
adder_t make_adder(int n) {
  n1 = n;
  return &adder;
}
```

Not just syntax

Store in Global variable

```
int main() {
  adder_t add1 = make_adder(1);
  add1(2); // 3
  adder_t add2 = make_adder(2);
  add2(2); // 4
}
```

Not just syntax

Store in Global variable

```
int main() {
  adder_t add1 = make_adder(1);
  adder_t add2 = make_adder(2);
  add1(2); // 4
  add2(2); // 4
}
```

Not just syntax

Store in Global variable

```
int n1;
int adder(int m) {
  return n1 + m;
}
adder_t make_adder(int n) {
  n1 = n;
  return &adder;
}
```

Closure Encoding

```
typedef struct {
  uint64_t func_ptr;
  expr_t *env;
} closure_t;
```

pseudo code: call conversion

```
closure_t make_adder(int n) {
  return make_closure(adder, n);
}
int main() {
  closure_t add1 = make_adder(1);
  closure_t add2 = make_adder(2);
  add1.func_ptr(2, add1.env);
  add2.func_ptr(2, add2.env);
}
```

Transformation

Free variables

```
let makeAdder = (n) => (m) => (n + m)
```

```
(n) => (m) => (n + m)
```

```
(m) => (n + m)
```

Transform to

```
[(m, env) => (env[0] + m), [n]]
```

```
(n) => [(m, env) => (env[0] + m), [n]]
```

Transform to

```
[(n, env) => [(m, env) => (env[0] + m), [n]], []]
```

Result

```
let makeAdder =
  [(n, env) =>
    [(m, env) => (env[0] + m)
     , [n]]
   , []]

let clos = makeAdder
let clos1 = clos[0](2, clos[1])
clos1[0](3, clos1[1]) // 5
```

Today’s target

JS to Scheme

```javascript
let makeAdder = (n) => (m) => (n + m)
makeAdder(2)(3)
```

```scheme
(begin
  (define make-adder
    (lambda (n)
      (lambda (m) (+ n m))))
  ((make-adder 2) 3))
```

JS to Scheme

```
let makeAdder = (n) => (m) => (n + m)
makeAdder(2)(3)
```

```
(begin
  (define make-adder
    (lambda (n)
      (lambda (m) (+ n m))))
  ((make-adder 2) 3))
```

JS to Scheme

```
let makeAdder = (n) => (m) => (n + m)
makeAdder(2)(3)
```

```
(begin
  (define make-adder
    (lambda (n)
      (lambda (m) (+ n m))))
  ((make-adder 2) 3))
```

```scheme
(define make-adder
  (lambda (n)
    (lambda (m) (+ n m))))
```

Shorthand

```scheme
(define (make-adder n)
  (lambda (m) (+ n m)))
```

Final

```
(begin
  (define (make-adder n)
    (lambda (m) (+ n m)))
  ((make-adder 2) 3))
```

Happy coding time

Me