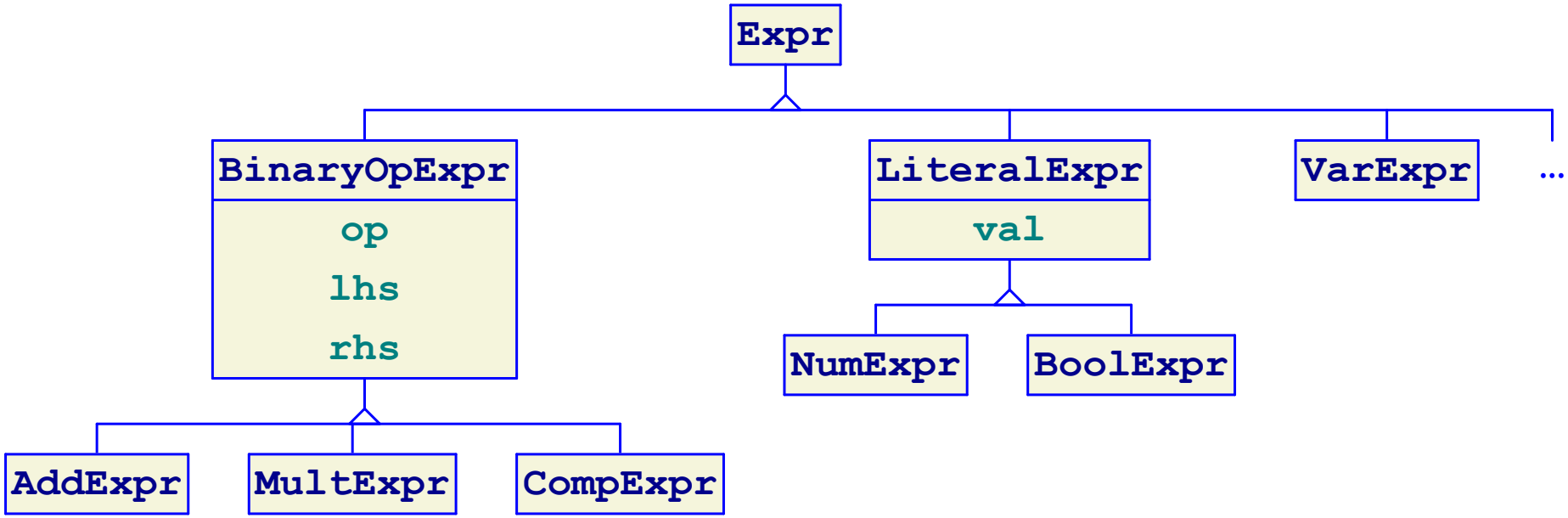
Interpreter: Better and Faster

Plan:

- Avoid allocating boolean values
- Fix a problem with long variable names
- Add `_letrec`
- Use `optimize`
- Use direct `interp` when possible

Numbers and Booleans as Literals



Numbers and Booleans as Literals

```
class LiteralExpr : public Expr {
  PTR(Val) val;

  LiteralExpr(PTR(Val) val);

  PTR(Val) interp(PTR(Env) env);
  std::string to_string();
  ....
};
```

```
class NumExpr : public LiteralExpr {
  NumExpr(int rep);
};
```

```
class BoolExpr : public LiteralExpr {
  BoolExpr(bool rep);
};
```

Numbers and Booleans as Literals

```
PTR(Val) LiteralExpr::interp(PTR(Env) env) {
  return val;
}

std::string LiteralExpr::to_string() {
  return val->to_string();
}

....
```

```
NumExpr::NumExpr(int rep)
: LiteralExpr(NEW(NumVal)(rep))
{ }
```

```
BoolExpr::BoolExpr(bool rep)
: LiteralExpr(NEW(BoolVal)(rep))
{ }
```

Numbers and Booleans as Literals

Old code:

```
PTR(Val) NumExpr::interp(PTR(Env) env) {
  return val;
}

void NumExpr::step_interp() {
  Step::mode = continue_mode;
  Step::val = NEW(NumVal)(rep);
  Step::cont = Step::cont;
}
```

Numbers and Booleans as Literals

Old code:

```
PTR(Val) NumExpr::interp(PTR(Env) env) {
  return val;
}

void NumExpr::step_interp() {
  Step::mode = continue_mode;
  Step::val = NEW(NumVal)(rep);
  Step::cont = Step::cont;
}
```

Avoid constructing a new **NumVal**

Numbers and Booleans as Literals

Old code:

```
PTR(Val) NumExpr::interp(PTR(Env) env) {
  return val;
}

void NumExpr::step_interp() {
  Step::mode = continue_mode;
  Step::val = NEW(NumVal)(rep);
  Step::cont = Step::cont;
}
```

Avoid constructing a new **NumVal**

Oops!

	8k heap	8M heap
NumExpr	0.85	0.56
LiteralExpr	0.77	0.54

Allocating Booleans

Using `LiteralExpr` means that we don't allocate for boolean literals:

```
PTR(Val) LiteralExpr::interp(PTR(Env) env) {
  return val;
}
```

```
BoolExpr::BoolExpr(bool rep)
: LiteralExpr(NEW(BoolVal)(rep))
{ }
```

Allocating Booleans

We're still allocating in a comparison:

```
PTR(Val) CompOp::perform(PTR(Val) lhs_val, PTR(Val) rhs_val) {
  return NEW(BoolVal)(lhs_val->equals(rhs_val));
}
```

We don't know whether the result will be **`_true`** or **`_false`**

... but we know that only two values are possible

Allocating Booleans

Allocate boolean values once and for all:

value.hpp

```
class BoolVal : public Val {
  ....
  static PTR(BoolVal) true_val;
  static PTR(BoolVal) false_val;
};
```

value.cpp

```
PTR(BoolVal) BoolVal::true_val = NEW(BoolVal)(true);
PTR(BoolVal) BoolVal::false_val = NEW(BoolVal)(false);
```

gc.cpp

```
void GCable::collect() {
  ....
  UPDATE(BoolVal::true_val);
  UPDATE(BoolVal::false_val);
  ....
}
```

Allocating Booleans

Use the global booleans:

value.cpp

```
BoolExpr::BoolExpr(bool rep)
: LiteralExpr(rep ? BoolVal::true_val : BoolVal::false_val)
{ }
```

op.cpp

```
PTR(Val) CompOp::perform(PTR(Val) lhs_val, PTR(Val) rhs_val) {
  return lhs_val->equals(rhs_val) ? BoolVal::true_val : BoolVal::false_val;
}
```

Booleans Performance

Seconds for `fib(fib)(28)`

	8k heap	8M heap
allocate	`0.77`	`0.54`
globals	`0.77`	`0.51`

Garbage Collection vs. `delete`

```
class NumVal : public Val {
  int rep;
  ....
};
```

Garbage collection takes care of memory for **`NumVal`**

Since only field is **`int`**, no need to call **`delete`** for **`NumVal`**

Garbage Collection vs. `delete`

```
class BinaryOpExpr : public Expr {
  PTR(BinaryOp) op;
  PTR(Expr) lhs;
  PTR(Expr) rhs;
  ....
};
```

Garbage collection takes care of memory for `BinaryOpExpr`

The `op` field is just a pointer to a singleton that doesn't need to be `delete`d

The `lhs` and `rhs` fields point to objects that garbage collection takes care of

Garbage Collection vs. `delete`

```
class FunVal : public Val {
  std::string formal_arg;
  PTR(Expr) body;
  PTR(Env) env;
  ....
};
```

Problem: the `formal_arg` is `std::string`, which may need to be `delete`d

It turns out that strings greater than 22 characters allocate additional memory...

Still Leaking

```
_let countdown = _fun (countdown)
                  _fun (abcdefghijklmnopqrstuvw)
                    _if abcdefghijklmnopqrstuvw == 0
                    _then 0
                    _else countdown(countdown)(abcdefghijklmnopqrstuvw-1)
_in countdown(countdown)(100)
```

abcdefghijklmnopqrstuvw : 23 characters

abcdefghijklmnopqrstuv : 22 character ⇒ no leak

Potentially Slow Comparison

```
PTR(Val) ExtendedEnv::lookup(std::string find_name) {
  if (name == find_name)
    return val;
  else
    return rest->lookup(find_name);
}
```

Potentially Slow Comparison

Overloaded to compare all characters

```
PTR(Val) ExtendedEnv::lookup(std::string find_name) {
  if (name == find_name)
    return val;
  else
    return rest->lookup(find_name);
}
```

From Strings to Symbols

Solution: allocate each string in the program once and for all as a ***symbol***

This program has just two symbols, **`fib`** and **`x`**:

```
_let fib = _fun (fib)
             _fun (x)
               _if x == 0
               _then 1
               _else _if x == 1
               _then 1
               _else fib(fib)(x + -1)
                     + fib(fib)(x + -2)
_in fib(fib)(28)
```

From Strings to Symbols

intern

Solution: allocate each string in the program once and for all as a ***symbol***

This program has just two symbols, **`fib`** and **`x`**:

```
_let fib = _fun (fib)
             _fun (x)
               _if x == 0
               _then 1
               _else _if x == 1
               _then 1
               _else fib(fib)(x + -1)
                     + fib(fib)(x + -2)
_in fib(fib)(28)
```

From Strings to Symbols

symbol.hpp

```
class Symbol {
private:
  std::string name;
  Symbol(std::string name);
public:
  static PTR(Symbol) intern(std::string name);
  std::string to_string();
};
```

From Strings to Symbols

not `GCable`

symbol.hpp

```
class Symbol {
private:
  std::string name;
  Symbol(std::string name);
public:
  static PTR(Symbol) intern(std::string name);
  std::string to_string();
};
```

From Strings to Symbols

symbol.hpp

```
class Symbol {
private:
  std::string name;
  Symbol(std::string name);
public:
  static PTR(Symbol) intern(std::string name);
  std::string to_string();
};
```

singleton-like instance method

From Strings to Symbols

symbol.cpp

```
Symbol::Symbol(std::string name) { // private
  this->name = name;
}

static std::map<std::string, PTR(Symbol)> interned;

PTR(Symbol) Symbol::intern(std::string name) {
  PTR(Symbol) sym;

  sym = interned[name];
  if (sym == nullptr) {
    sym = NEW(Symbol)(name);
    interned[name] = sym;
  }

  return sym;
}
```

From Strings to Symbols

symbol.cpp

```
Symbol::Symbol(std::string name) { // private
  this->name = name;
}

static std::map<std::string, PTR(Symbol)> interned;

PTR(Symbol) Symbol::intern(std::string name) {
  PTR(Symbol) sym;

  sym = interned[name];
  if (sym == nullptr) {
    sym = NEW(Symbol)(name);
    interned[name] = sym;
  }

  return sym;
}
```

Table of all **Symbol**s

From Strings to Symbols

symbol.cpp

```
Symbol::Symbol(std::string name) { // private
  this->name = name;
}

static std::map<std::string, PTR(Symbol)> interned;

PTR(Symbol) Symbol::intern(std::string name) {
  PTR(Symbol) sym;

  sym = interned[name];
  if (sym == nullptr) {
    sym = NEW(Symbol)(name);
    interned[name] = sym;
  }

  return sym;
}
```

Find or create-and-register a **Symbol**

From Strings to Symbols

Change `std::string` to `PTR(Symbol)` in all `Expr`, `Val`, `Env`, and `Cont` classes:

```
class ExtendedEnv : public Env {
  PTR(Symbol) name;
  PTR(Val) val;
  PTR(Env) rest;
  ....
};
```

```
class FunVal : public Val {
  PTR(Symbol) formal_arg;
  PTR(Expr) body;
  PTR(Env) env;
  ....
};
```

....

From Strings to Symbols

Change `std::string` to `PTR(Symbol)` in all `Expr`, `Val`, `Env`, and `Cont` classes:

....

```
class Expr {
  ....
  virtual PTR(Expr) subst(PTR(Symbol) var, PTR(Val) val) = 0;
  ....
};
```

....

From Strings to Symbols

```
PTR(Val) ExtendedEnv::lookup(PTR(Symbol) find_name) {
  if (name == find_name)
    return val;
  else
    return rest->lookup(find_name);
}
```

From Strings to Symbols

Simple pointer comparison

```
PTR(Val) ExtendedEnv::lookup(PTR(Symbol) find_name) {
  if (name == find_name)
    return val;
  else
    return rest->lookup(find_name);
}
```

From Strings to Symbols

Intern `std::string` as `PTR(Symbol)` when parsing

```
static PTR(Expr) parse_fun_form(std::istream &in) {
  ....
  std::string formal_arg = parse_alphabetic(in, "");
  ....
  PTR(Expr) body = parse_expr(in);

  return NEW(FunExpr)(Symbol::intern(formal_arg), body);
}
```

Symbols Performance

Using symbols not only fixes the leak, it saves `std::string` copying complexity

Seconds for `fib(fib)(28)`

	8k heap	8M heap
`std::string`	`0.77`	`0.51`
`Symbol`	`0.51`	`0.33`
`Symbol` with allocating ==	`0.54`	`0.34`

Allocation in Recursive Functions

```
_let fib = _fun (fib)
             _fun (x)
               _if x == 0
               _then 1
               _else _if x == 1
               _then 1
               _else fib(fib)(x + -1)
                     + fib(fib)(x + -2)
_in fib(fib)(28)
```

Allocation in Recursive Functions

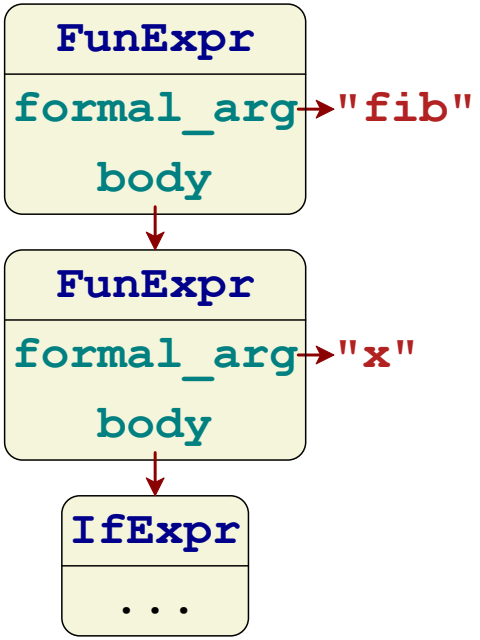
```
_let fib = _fun (fib)
             _fun (x)
               _if x == 0
               _then 1
               _else _if x == 1
               _then 1
               _else fib(fib)(x + -1)
                     + fib(fib)(x + -2)
_in fib(fib)(28)
```

Allocation in Recursive Functions

```
_let fib = _fun (fib)
             _fun (x)
               _if x == 0
               _then 1
               _else _if x == 1
               _then 1
               _else fib(fib)(x + -1)
                     + fib(fib)(x + -2)
_in fib(fib)(28)
```

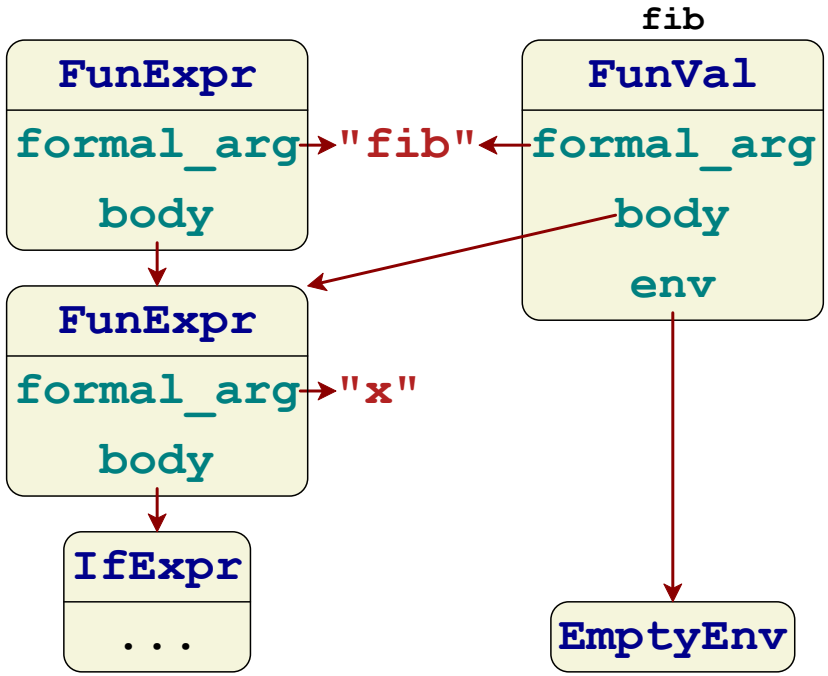
Allocates a closure every time

Allocation in Recursive Functions

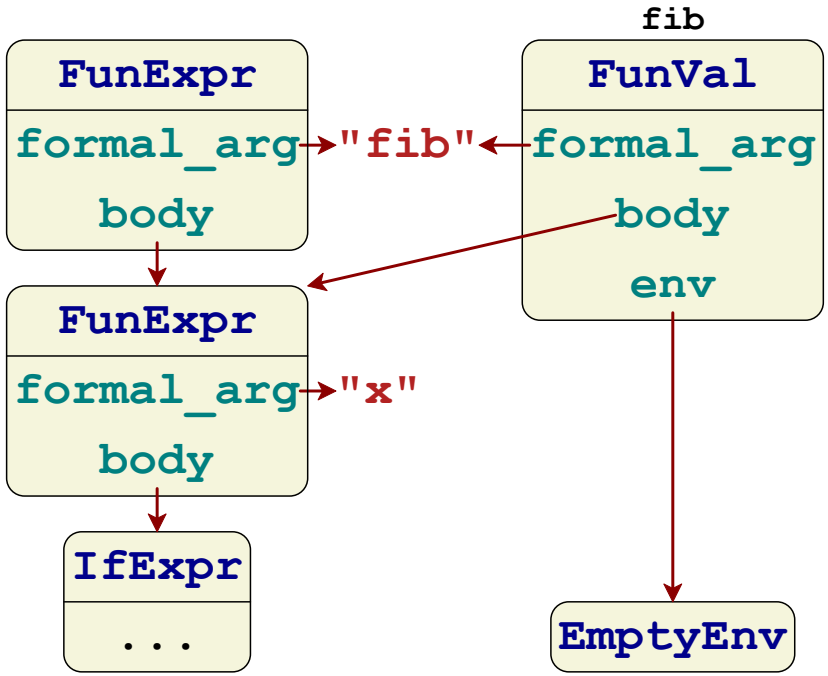



```
_fun (fib)
   _fun (x)
      _if x == 0
      _then 1
      _else _if x == 1
      _then 1
      _else fib(fib)(x + -1)
            + fib(fib)(x + -2)
```

Allocation in Recursive Functions

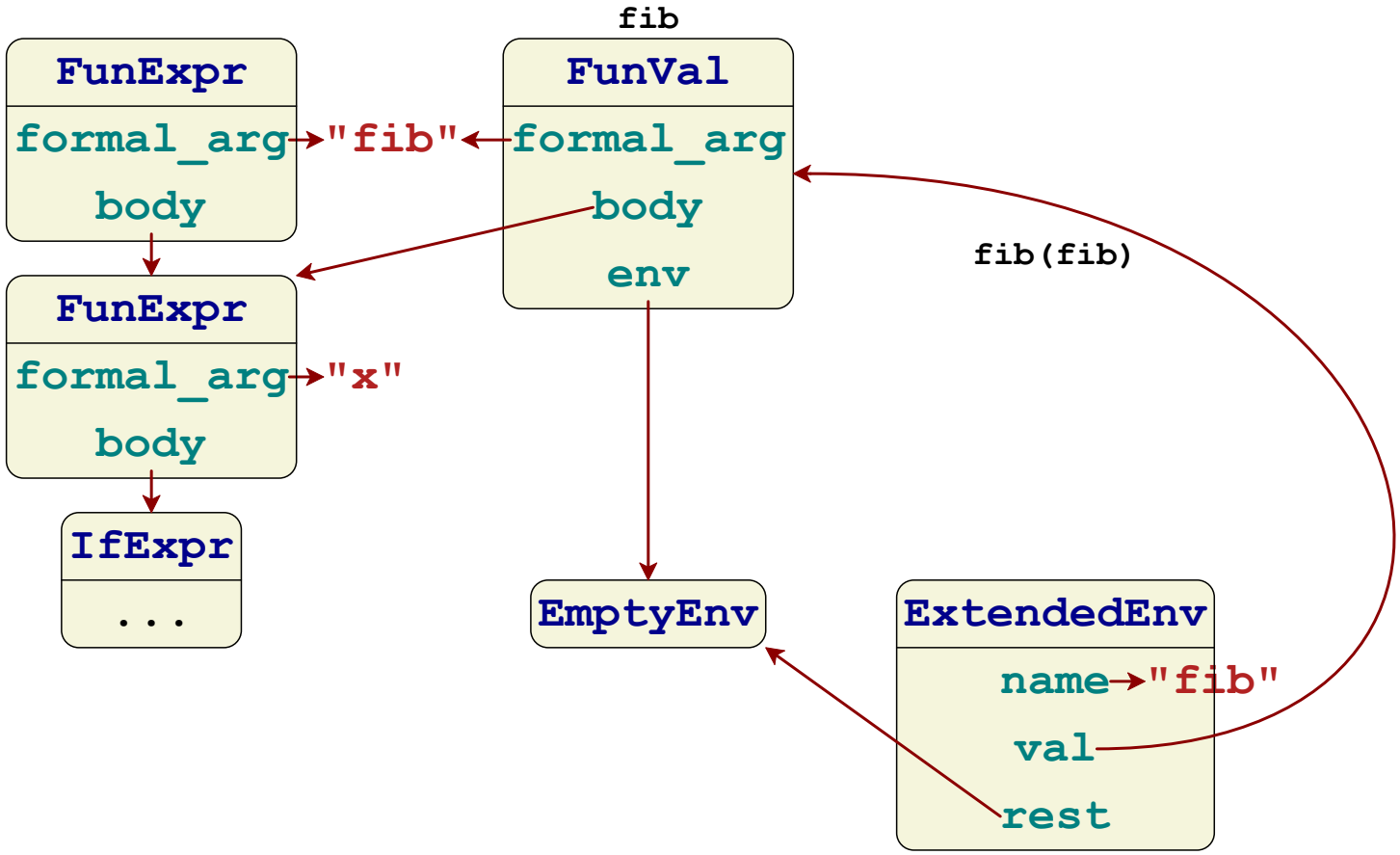


Allocation in Recursive Functions

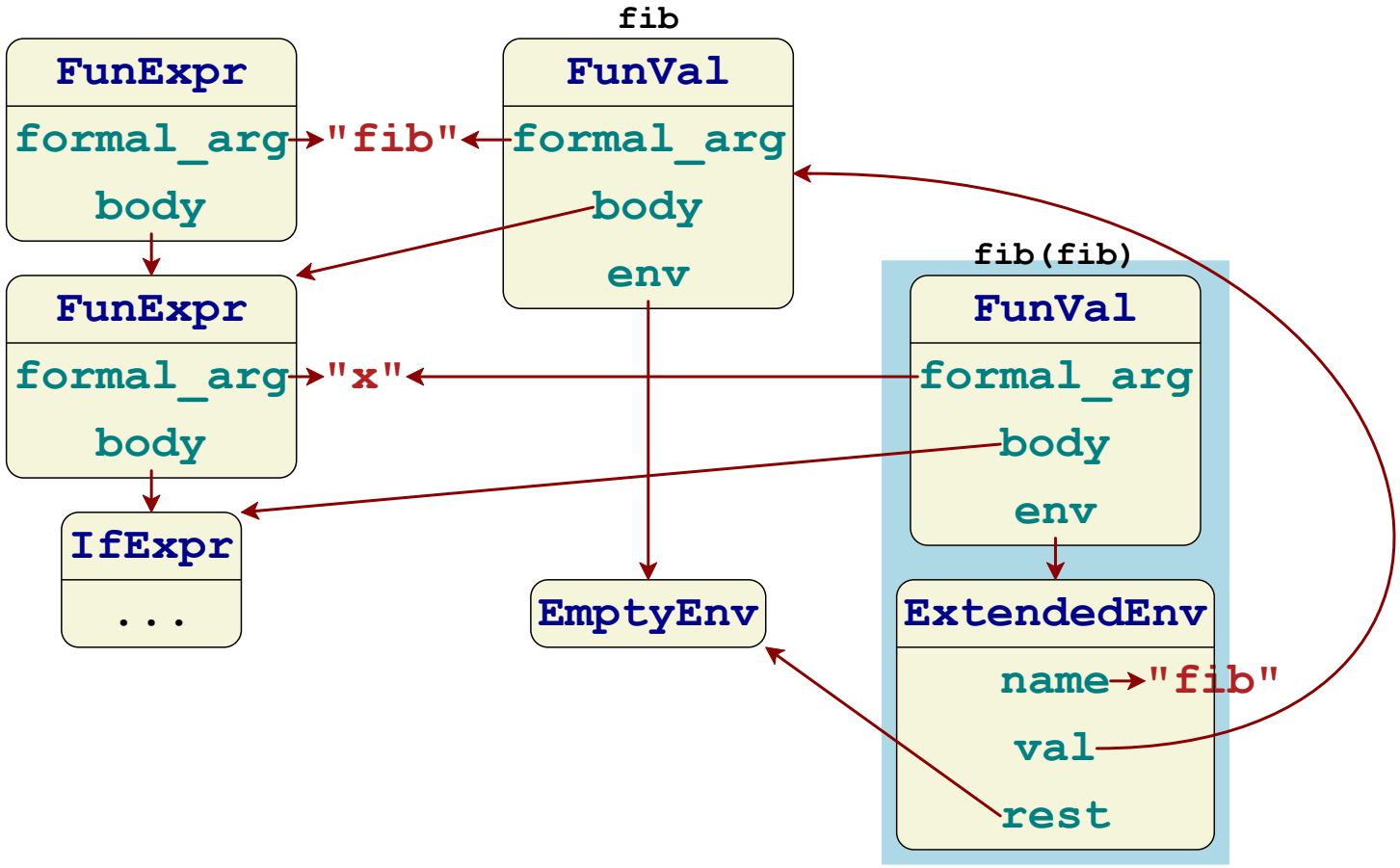



```
fib(fib)
```

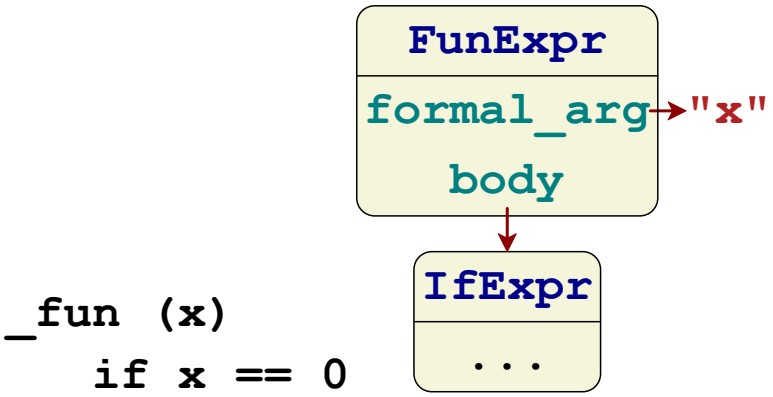
Allocation in Recursive Functions



Allocation in Recursive Functions

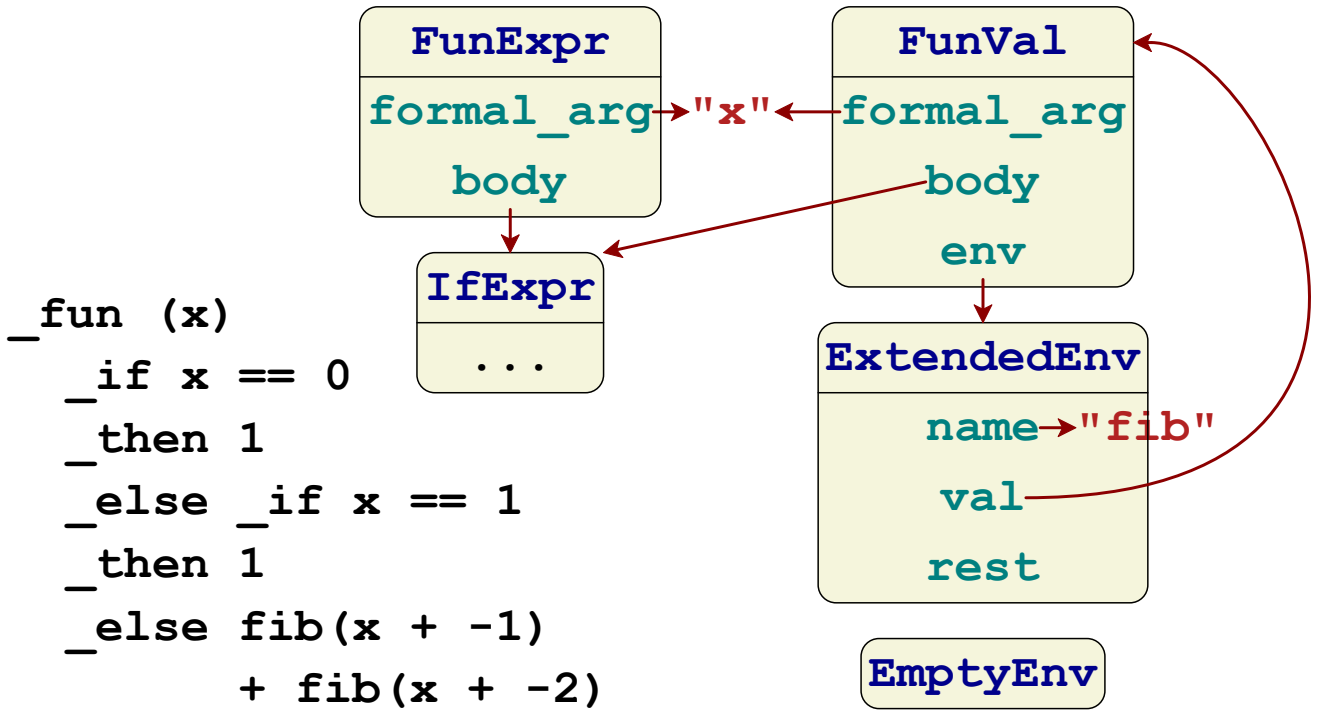


Allocation in Recursive Functions




```
_fun (x)
   _if x == 0
   _then 1
   _else _if x == 1
   _then 1
   _else fib(x + -1)
         + fib(x + -2)
```

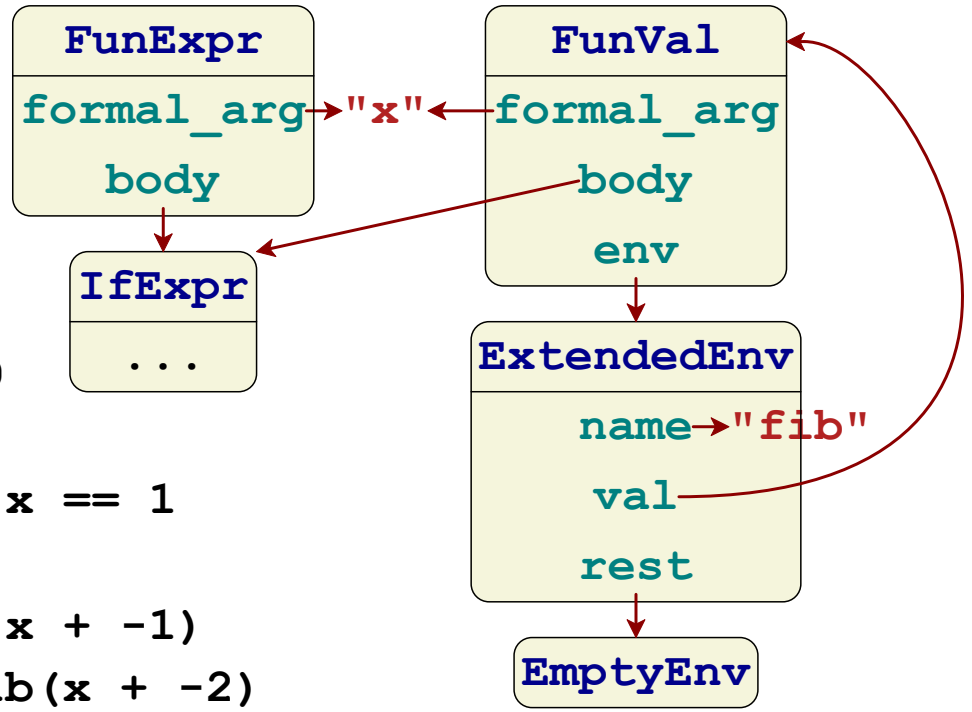
Allocation in Recursive Functions




```
_fun (x)
   _if x == 0
   _then 1
   _else _if x == 1
   _then 1
   _else fib(x + -1)
         + fib(x + -2)
```

Allocation in Recursive Functions

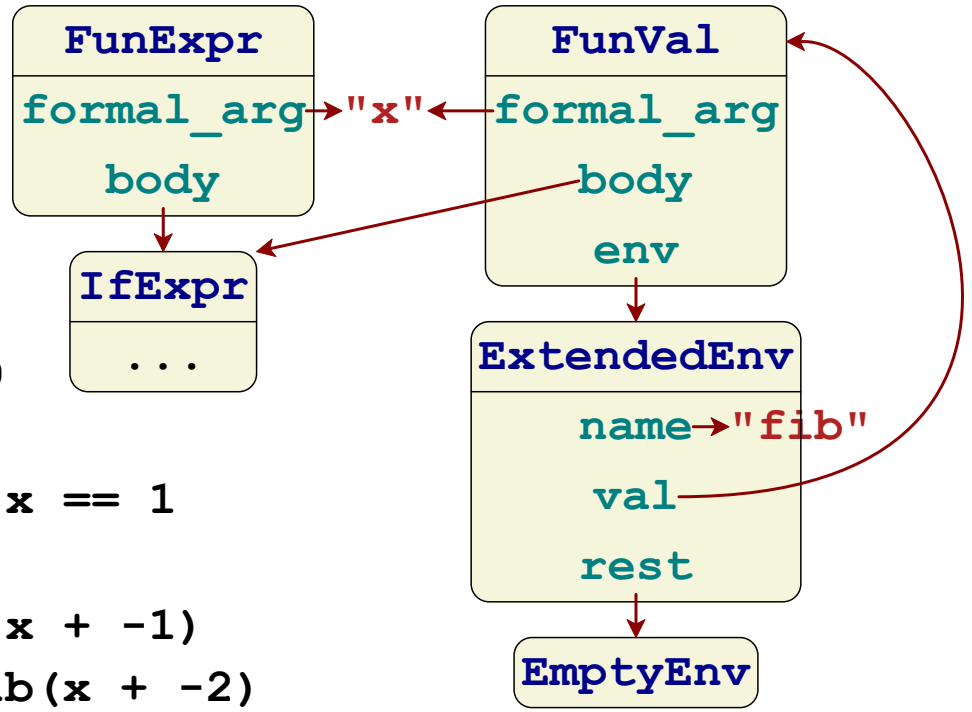
```
_letrec fib = _fun (x)
                _if x == 0
                _then 1
                _else _if x == 1
                _then 1
                _else fib(x + -1)
                      + fib(x + -2)
_in fib(28)
```



Allocation in Recursive Functions

New kind of expression, binds after = *and* after `_in`

```
_letrec fib = _fun (x)
                _if x == 0
                _then 1
                _else _if x == 1
                _then 1
                _else fib(x + -1)
                      + fib(x + -2)
_in fib(28)
```



Recursive Definitions

```
_letrec x = x + 1
_in x
```

Recursive Definitions

Prevent nonsense by requiring a **_fun** expression after **=**

```
_letrec x = x + 1
_in x
```

MSDscript with Recursive Binding

⟨expr⟩ = ⟨oparg$_0$⟩

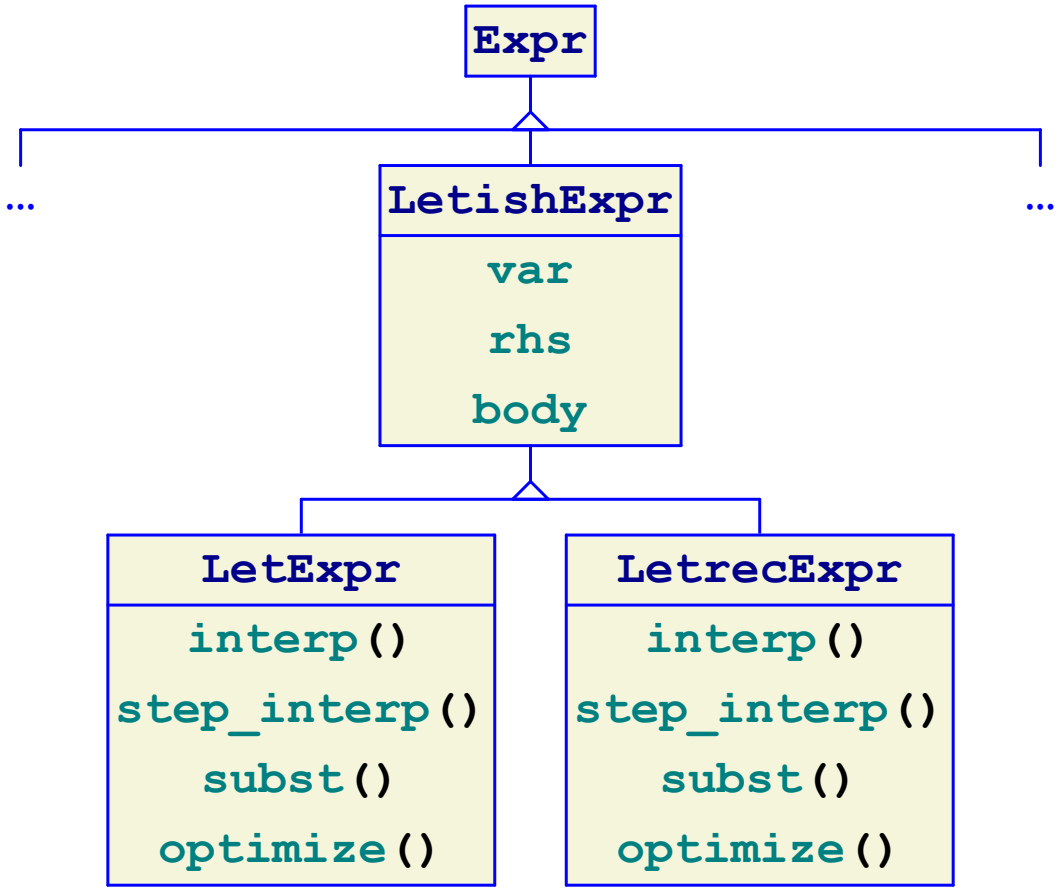
⟨oparg$_P$⟩ = ⟨oparg$_{P+1}$⟩
| ⟨oparg$_{P+1}$⟩ ⟨op$_P$⟩ ⟨oparg$_P$⟩

⟨oparg$_N$⟩ = ⟨multicand⟩
| ⟨oparg$_N$⟩ `(` ⟨expr⟩ `)`

⟨multicand⟩ = ⟨number⟩ | `(` ⟨expr⟩ `)` | ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_letrec` ⟨variable⟩ `=` ⟨fun-expr⟩ `_in` ⟨expr⟩ new
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

⟨fun-expr⟩ = `_fun` `(` ⟨variable⟩ `)` ⟨expr⟩

MSDscript with Recursive Binding



MSDscript with Recursive Binding

```
PTR(Expr) LetExpr::subst(PTR(Symbol) repl_var, PTR(Val) new_val) {
  if (var == repl_var)
    return NEW(LetExpr)(var,
                        rhs->subst(repl_var, new_val),
                        body);
  else
    return NEW(LetExpr)(var,
                        rhs->subst(repl_var, new_val),
                        body->subst(repl_var, new_val));
}
```

```
PTR(Expr) LetrecExpr::subst(PTR(Symbol) repl_var, PTR(Val) new_val) {
  if (var == repl_var)
    return THIS;
  else
    return NEW(LetrecExpr)(var,
                           rhs->subst(repl_var, new_val),
                           body->subst(repl_var, new_val));
}
```

MSDscript with Recursive Binding

```
PTR(Expr) LetExpr::subst(PTR(Symbol) repl_var, PTR(Val) new_val) {
  if (var == repl_var)
    return NEW(LetExpr)(var,
                        rhs->subst(repl_var, new_val),
                        body);
  else
    return NEW(LetExpr)(var,
                        rhs->subst(repl_var, new_val),
                        body->subst(repl_var, new_val));
}
```

```
PTR(Expr) LetrecExpr::subst(PTR(Symbol) repl_var, PTR(Val) new_val) {
  if (var == repl_var)
    return THIS;
  else
    return NEW(LetrecExpr)(var,
                           rhs->subst(repl_var, new_val),
                           body->subst(repl_var, new_val));
}
```

Block substition in `rhs`

MSDscript with Recursive Binding

```
PTR(Expr) LetrecExpr::optimize() {
  return NEW(LetrecExpr)(var, rhs->optimize(), body->optimize());
}
```

Since `rhs` is a function expression,
and we don't try to call functions in `optimize`,
just optimize subexpressions

MSDscript with Recursive Binding

```
PTR(Val) LetrecExpr::interp(PTR(Env) env) {
  PTR(ExtendedEnv) new_env = NEW(ExtendedEnv)(var, BoolExpr:false_val, env);

  PTR(Val) rhs_val = rhs->interp(new_env);
  new_env->val = rhs_val;

  return body->interp(new_env);
}
```

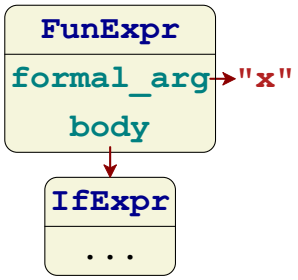
MSDscript with Recursive Binding

```
PTR(Val) LetrecExpr::interp(PTR(Env) env) {
  PTR(ExtendedEnv) new_env = NEW(ExtendedEnv)(var, BoolExpr:false_val, env);

  PTR(Val) rhs_val = rhs->interp(new_env);
  new_env->val = rhs_val;

  return body->interp(new_env);
}
```

Must be a **FunExpr**



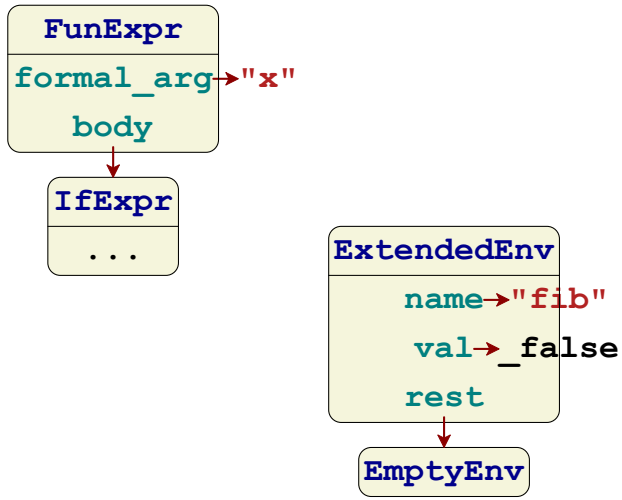
MSDscript with Recursive Binding

```
PTR(Val) LetrecExpr::interp(PTR(Env) env) {
  PTR(ExtendedEnv) new_env = NEW(ExtendedEnv)(var, BoolExpr:false_val, env);

  PTR(Val) rhs_val = rhs->interp(new_env);
  new_env->val = rhs_val;

  return body->interp(new_env);
}
```

Temporarily bind to `_false`



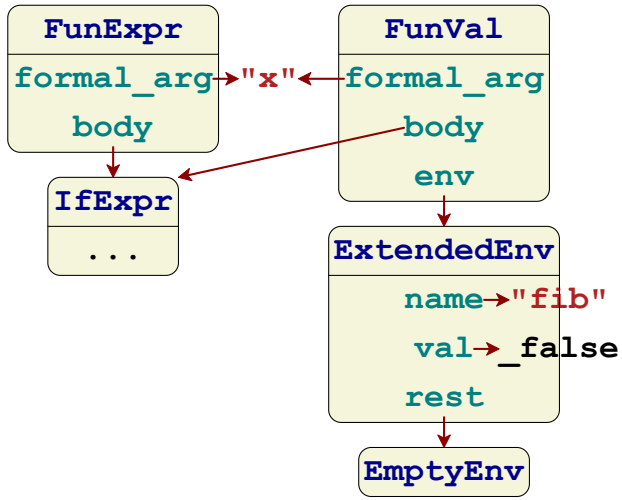
MSDscript with Recursive Binding

```
PTR(Val) LetrecExpr::interp(PTR(Env) env) {
  PTR(ExtendedEnv) new_env = NEW(ExtendedEnv)(var, BoolExpr:false_val, env);

  PTR(Val) rhs_val = rhs->interp(new_env);
  new_env->val = rhs_val;

  return body->interp(new_env);
}
```

Produces **FunVal** without inspecting **new_env**



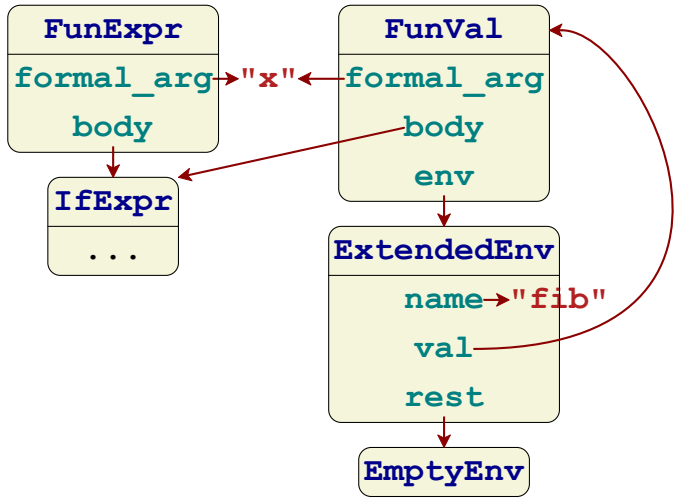
MSDscript with Recursive Binding

```
PTR(Val) LetrecExpr::interp(PTR(Env) env) {
  PTR(ExtendedEnv) new_env = NEW(ExtendedEnv)(var, BoolExpr:false_val, env);

  PTR(Val) rhs_val = rhs->interp(new_env);
  new_env->val = rhs_val;

  return body->interp(new_env);
}
```

Create cycle



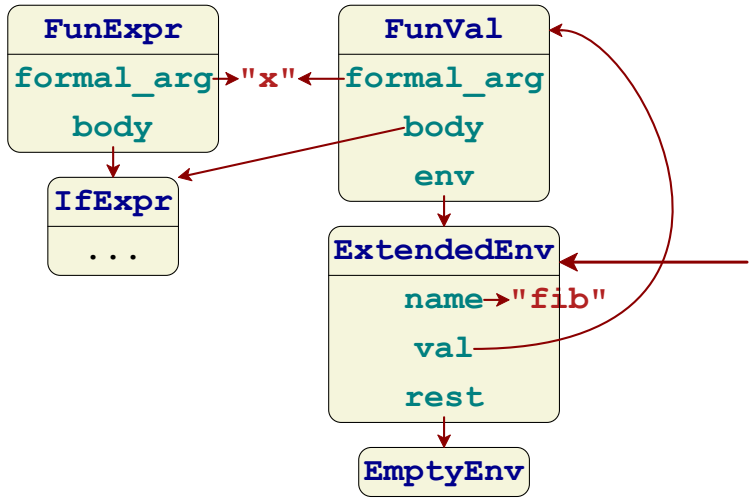
MSDscript with Recursive Binding

```
PTR(Val) LetrecExpr::interp(PTR(Env) env) {
  PTR(ExtendedEnv) new_env = NEW(ExtendedEnv)(var, BoolExpr:false_val, env);

  PTR(Val) rhs_val = rhs->interp(new_env);
  new_env->val = rhs_val;

  return body->interp(new_env);
}
```

Use cyclic **new_env** for **body**



MSDscript with Recursive Binding

```
void LetrecExpr::step_interp() {
  PTR(ExtendedEnv) new_env = NEW(ExtendedEnv)(var, BoolVal::false_val, Step::env);

  Step::mode = interp_mode;
  Step::expr = rhs;
  Step::env = new_env;
  Step::cont = NEW(LetrecBodyCont)(var, body, Step::env, Step::cont);
}
```

```
void LetrecBodyCont::step_continue() {
  Step::mode = interp_mode;
  Step::expr = body;
  CAST(ExtendedEnv)(env)->val = Step::val;
  Step::cont = rest;
}
```

MSDscript with Recursive Binding

Seconds for

	8k heap	8M heap
`_let .... fib(fib)(28)`	`0.51`	`0.33`
`_letrec .... fib(28)`	`0.37`	`0.28`

Optimization

```
_letrec fib = _fun (x)
                _if x == 2 + -2
                _then 1
                _else _if x == 1
                _then 1
                _else fib(x + -1)
                      + fib(x + -2)
_in fib(28)
```

Optimization

Let's put `optimize` to work, finally

main.cpp

```
e = e->optimize();

if (optimize_mode)
  std::cout << e->to_string() << "\n";
else if (step_mode)
  std::cout << interp_by_steps(e)->to_string() << "\n";
else
  std::cout << e->interp(Env::empty)->to_string() << "\n";
```

Direct Interpret versus Stepping

Seconds for `fib(28)`

	leak	8k heap	8M heap
`interp`	`0.14`		
`step_interp`		`0.37`	`0.28`

Direct Interpret versus Stepping

Evaluate `x + 1`:

`interp`

`step_interp`

Direct Interpret versus Stepping

Evaluate **`x + 1`**:

`interp`

dispatch **`BinaryOpExpr`**

  dispatch **`VarExpr`**

    lookup **`x`**

  dispatch **`LiteralExpr`**

    get **`1`**

  dispatch **`AddOp`**

    alloc **`NumVal`**

`step_interp`

Direct Interpret versus Stepping

Evaluate **`x + 1`**:

`interp`

dispatch **`BinaryOpExpr`**

  dispatch **`VarExpr`**

    lookup **`x`**

  dispatch **`LiteralExpr`**

    get **`1`**

  dispatch **`AddOp`**

    alloc **`NumVal`**

`step_interp`

dispatch **`BinaryOpExpr`**

  alloc **`RightThenOpCont`**

dispatch **`VarExpr`**

  lookup **`x`**

dispatch **`RightThenOpCont`**

  alloc **`OpCont`**

dispatch **`LiteralExpr`**

  get **`1`**

dispatch **`OpCont`**

  dispatch **`AddOp`**

    alloc **`NumVal`**

Direct Interpret versus Stepping

What if we used `interp` for `x + 1` in the middle of a `step_interp`?

- No function calls means doesn't recur deeply, so that's ok
- Allocates only `sizeof(NumVal)` bytes, so that's ok

But checking for simple cases in `step_interp` could slow it down...

Idea: make `optimize` change a *simple* `BinaryOpExpr` to `DirectBinaryOpExpr`

Simple means "doesn't allocate too much," so add a `bytes_allocated_on_interp` method to `Expr` for `optimize` to use

Optimizing Simple Expressions

```
class DirectBinaryOpExpr : public BinaryOpExpr {
  void step_interp();
  int bytes_allocated_on_interp();
};
```

```
void DirectBinaryOpExpr::step_interp() {
  Step::mode = continue_mode;
  Step::val = this->interp(Step::env);
}
```

Optimizing Simple Expressions

```
PTR(Expr) BinaryOpExpr::optimize() {
  PTR(Expr) new_lhs = lhs->optimize();
  PTR(Expr) new_rhs = rhs->optimize();

  PTR(Expr) new_expr;

  if (GCable::safety_margin > (new_lhs->bytes_allocated_on_interp()
                               + new_rhs->bytes_allocated_on_interp()
                               + op->bytes_allocated_on_interp()))
    new_expr = NEW(DirectBinaryOpExpr)(op, new_lhs, new_rhs);
  else
    new_expr = NEW(BinaryOpExpr)(op, new_lhs, new_rhs);
  ....
}
```

Optimizing Simple Expressions

```
int LiteralExpr::bytes_allocated_on_interp() {
  return 0;
}
```

```
int VarExpr::bytes_allocated_on_interp() {
  return 0;
}
```

```
int CallExpr::bytes_allocated_on_interp() {
  return GCable::safety_margin+1;
}
```

Optimizing Simple Expressions

```
int LiteralExpr::bytes_allocated_on_interp() {
  return 0;
}
```

```
int VarExpr::bytes_allocated_on_interp() {
  return 0;
}
```

```
int CallExpr::bytes_allocated_on_interp() {
  return GCable::safety_margin+1;
}
```

Don't know how much the called function would allocate

Optimizing Simple Expressions

```
int LiteralExpr::bytes_allocated_on_interp() {
  return 0;
}
```

```
int VarExpr::bytes_allocated_on_interp() {
  return 0;
}
```

```
int CallExpr::bytes_allocated_on_interp() {
  return GCable::safety_margin+1;
}
```

Conservatively report larger than `safety_margin`, which means *not simple*

Optimizing Simple Expressions

```
int BinaryOpExpr::bytes_allocated_on_interp() {
  return GCable::safety_margin+1;
}
```

Optimizing Simple Expressions

```
int BinaryOpExpr::bytes_allocated_on_interp() {
  return GCable::safety_margin+1;
}
```

Conservative, since not optimized

Optimizing Simple Expressions

```
int BinaryOpExpr::bytes_allocated_on_interp() {
  return GCable::safety_margin+1;
}
```

```
int DirectBinaryOpExpr::bytes_allocated_on_interp() {
  return (lhs->bytes_allocated_on_interp()
          + rhs->bytes_allocated_on_interp()
          + op->allocates_on_perform());
}
```

Optimizing Simple Expressions

```
int BinaryOpExpr::bytes_allocated_on_interp() {
  return GCable::safety_margin+1;
}
```

```
int DirectBinaryOpExpr::bytes_allocated_on_interp() {
  return (lhs->bytes_allocated_on_interp()
          + rhs->bytes_allocated_on_interp()
          + op->allocates_on_perform());
}
```

Precise calculation allows nested simple expressions

Optimizing Simple Expressions

```
class BinaryOp {
  ....
  virtual int bytes_allocated_on_perform() = 0;
};
```

```
class BinaryNumOp : public BinaryOp {
  int bytes_allocated_on_perform();
};
```

```
class AddOp : public BinaryNumOp {
  ....
};
```

Optimizing Simple Expressions

```
int BinaryNumOp::bytes_allocated_on_perform() {
  return sizeof(NumVal);
}

int BinaryBoolOp::bytes_allocated_on_perform() {
  return 0;
}
```

Optimizing Simple Expressions

```
_letrec fib = _fun (x)
                _if x == 0
                _then 1
                _else _if x == 1
                _then 1
                _else fib(x + -1)
                      + fib(x + -2)
_in fib(28)
```

Optimizing Simple Expressions

```
_letrec fib = _fun (x)
                _if x == 0
                _then 1
                _else _if x == 1
                _then 1
                _else fib(x + -1)
                      + fib(x + -2)
_in fib(28)
```

Comparison is simple

Optimizing Simple Expressions

```
_letrec fib = _fun (x)
                _if x == 0
                _then 1
                _else _if x == 1
                _then 1
                _else fib(x + -1)
                      + fib(x + -2)
_in fib(28)
```

Addition is simple

Optimizing Simple Expressions

```
_letrec fib = _fun (x)
                _if x == 0
                _then 1
                _else _if x == 1
                _then 1
                _else fib(x + -1)
                      + fib(x + -2)
_in fib(28)
```

Call is *not* simple

Optimizing Simple Expressions

Conditional is not simple, but could avoid **`IfBranchCont`** for simple tests

```
_letrec fib = _fun (x)
                _if x == 0
                _then 1
                _else _if x == 1
                _then 1
                _else fib(x + -1)
                      + fib(x + -2)
_in fib(28)
```

Optimizing Simple Expressions

```
void DirectTestIfExpr::step_interp() {
  PTR(Val) test_val = test_part->interp(Step::env);

  Step::mode = interp_mode;
  if (test_val->is_true())
    Step::expr = then_part;
  else
    Step::expr = else_part;
}
```

```
PTR(Expr) IfExpr::optimize() {
  ....
  if (GCable::safety_margin > new_test_part->bytes_allocated_on_interp())
    return NEW(DirectTestIfExpr)(new_test_part, new_then_part, new_else_part);
  else
    return NEW(IfExpr)(new_test_part, new_then_part, new_else_part);
}
```

Direct Interpret versus Stepping

Seconds for `fib(28)`

	leak	8k heap	8M heap
`interp`	`0.14`		
`step_interp`		`0.37`	`0.28`
`interp` for simple		`0.19`	`0.16`

Direct Interpret versus Stepping

Seconds for `fib(28)`

	leak	8k heap	8M heap	
`interp`	`0.14`			
`step_interp`		`0.37`	`0.28`	
`interp` for simple		`0.19`	`0.16`	
`racket -j`				`0.09`