Types

A ***type system*** allows a language to ensure the absence of certain behaviors *before* a program is run

Types

A ***type system*** allows a language to ensure the absence of certain behaviors *before* a program is run

especially errors

Types

A ***type system*** allows a language to ensure the absence of certain behaviors *before* a program is run

especially errors

such as "not a number" from +

Types

A ***type system*** allows a language to ensure the absence of certain behaviors *before* a program is run

An expression has a ***type***

A value has a ***class***

Intuitively, there's a connection between
the type of an expression and
the class of the expression's value

Type != Class

```
Expr *e;
....
e->to_string();
```

The expression `e` has type `Expr*`

When the program runs, the value of `e` can be

- an object whose class is `BinaryOpExpr`,
- an object whose class is `LitExpr`,
- ...,
- or the null pointer.

Type != Class

```
Expr *e;
....
(dynamic_cast<BinaryOpExpr*>(e))->rhs->to_string();
```

The highlighted expression has type **`BinaryOpExpr*`**

When the program runs, the value of the expression can be

- an object whose class is **`BinaryOpExpr`**,
- ...,
- or the null pointer.

Types and Values

A type is an abstraction of a value

- A type loses some information about the exact value

  An expression of type `int` has a value that is *some* integer

- A type must reflect enough information to be useful

  The type `Expr*` ensures a `to_string` method

A type checker looks like an interpreter, but produces a *type* for each expression instead of a *value*

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

`1`
has type
`Num`

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

`_false`
has type
`Bool`

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
2 + 3
```
has type
`Num`

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
2 == 3
```
has type
`Bool`

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
2 + _false
```

has no type

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
2 + _false
```
has no type

When an expression *has no type*, don't try to interpret it

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
_fun (x)
  x == 6
```

has type

```
Num -> Bool
```

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
_fun (x : Num)
  x == 6
```

has type

```
Num -> Bool
```

Need to update grammar to include type declarations

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
_fun (f : Num -> Num)
  f(1) == f(2)
```

has type

```
(Num -> Num) -> Bool
```

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
_fun (n : Num)
  _fun (m : Num)
    n == m
```

has type

```
Num -> (Num -> Bool)
```

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
_fun (n : Num)
  _fun (m : Num)
    n == m
```

has type

```
Num -> Num -> Bool
```

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
_let x = 2 + 3
_in x
```

has type

Num

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`

Types

⟨type⟩ = `Num`
| `Bool`

Type of `2 + 3` determines type for `x`

```
_let x = 2 + 3
_in x
```

has type

Num

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`
| `[function]`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

```
_letrec loop : Num -> Num
             = _fun (x : Num)
                 loop(x)
_in loop(0)
```

has type

Num

MSDscript with Types

Expressions

⟨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⟩
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

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

Values

⟨val⟩ = ⟨number⟩
| `_true` | `_false`

Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`

Need type *before* checking RHS expression

```
_letrec loop : Num -> Num
                = _fun (x : Num)
                    loop(x)
_in loop(0)
```

has type

Num

Expressions with Types

⟨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⟩ `:` ⟨type⟩ `=` ⟨fun-expr⟩ `_in` ⟨expr⟩ revised
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

⟨fun-expr⟩ = `_fun` `(` ⟨variable⟩ `:` ⟨type⟩ `)` ⟨expr⟩ revised

Expressions with Types

⟨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⟩ `:` ⟨type⟩ `=` ⟨fun-expr⟩ `_in` ⟨expr⟩ revised
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

⟨fun-expr⟩ = `_fun` `(` ⟨variable⟩ `:` ⟨type⟩ `)` ⟨expr⟩ revised

Parser can treat `:` ⟨type⟩ as optional, so untyped MSDscript still parses

Expressions with Types

⟨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⟩ `:` ⟨type⟩ `=` ⟨fun-expr⟩ `_in` ⟨expr⟩ revised
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

⟨fun-expr⟩ = `_fun` `(` ⟨variable⟩ `:` ⟨type⟩ `)` ⟨expr⟩ revised

```
class FunExpr : public Expr {
  PTR(Symbol) formal_arg;
  PTR(Type) arg_type;
  PTR(Expr) body;
  ....
};
```

Expressions with Types

⟨expr⟩ = ⟨oparg$_0$⟩

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

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

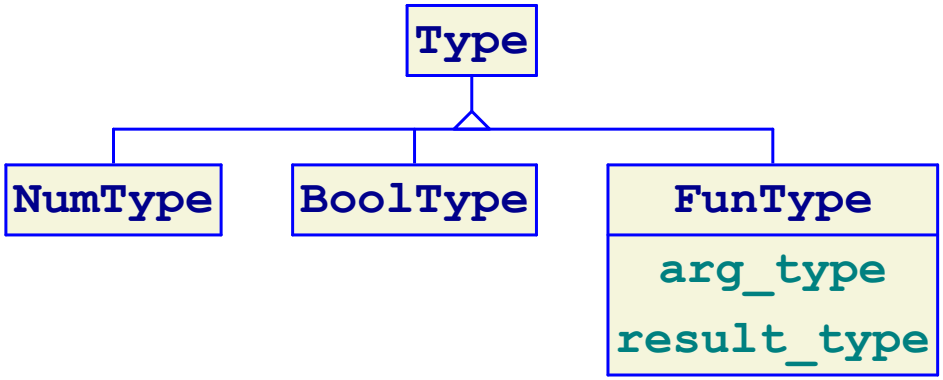
⟨multicand⟩ = ⟨number⟩ | `(` ⟨expr⟩ `)` | ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩
| `_letrec` ⟨variable⟩ `:` ⟨type⟩ `=` ⟨fun-expr⟩ `_in` ⟨expr⟩ revised
| `_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩
| ⟨fun-expr⟩

⟨fun-expr⟩ = `_fun` `(` ⟨variable⟩ `:` ⟨type⟩ `)` ⟨expr⟩ revised

```
class LetrecExpr : public LetishExpr {
  PTR(Type) rhs_type;
  ....
};
```

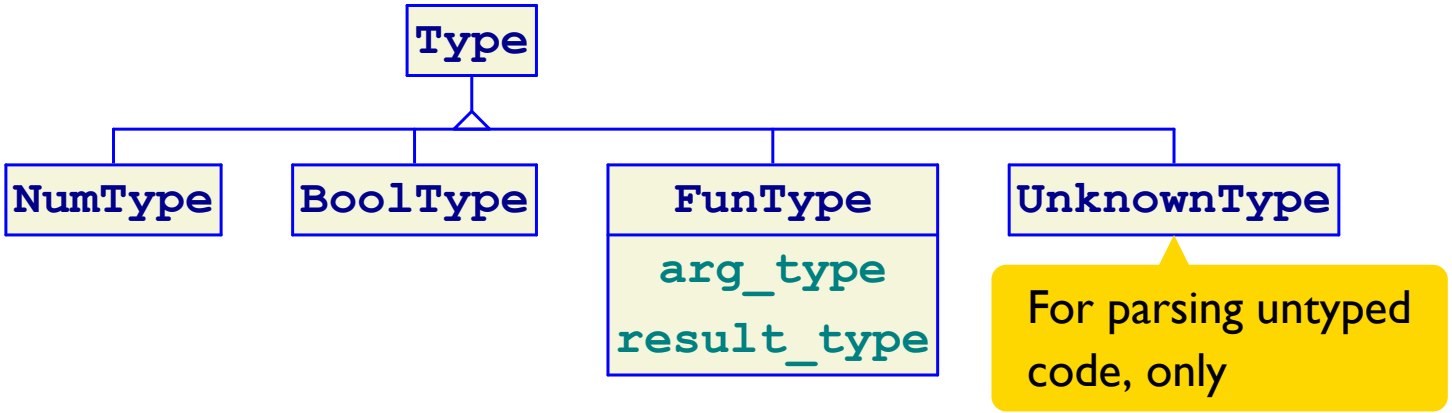
Representing Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`



Representing Types

⟨type⟩ = `Num`
| `Bool`
| ⟨type⟩ `->` ⟨type⟩
| `(` ⟨type⟩ `)`



Typechecking

Expr
....
PTR(Val) interp(...)
PTR(Type) typecheck(...)
....

The `Expr:interp()` method gets an expression's value

If something goes wrong, `interp` throws an exception

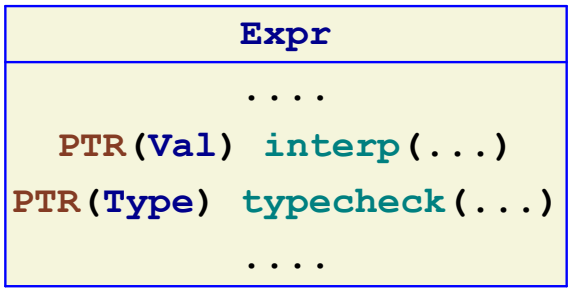
A new `Expr:typecheck()` method will get an expression's type

If checking fails, `typecheck` will throw a "no type" exception

Goal #1: no `typecheck` exception ⇒ no `interp` exception

Goal #2: `typecheck` always finishes quickly, unlike `interp`

Typechecking



The `Expr:interp()` method gets an expression's value

If something goes wrong, `interp` throws an exception

A new `Expr:typecheck()` method will get an expression's type

If checking fails, `typecheck` will throw a "no type" exception

Don't call functions

Goal #1: no `typecheck` exception ⇒ no `interp` exception

Goal #2: `typecheck` always finishes quickly, unlike `interp`

Typechecking

⟨number⟩

NumExpr
val

Every ⟨number⟩ expression has type **Num**

```
PTR(Type) NumExpr::typecheck() {
  return NEW(NumType)();
}
```

Typechecking

`_false` | `_true`

BoolExpr
val

Every boolean expression has type `Bool`

```
PTR(Type) BoolExpr::typecheck() {
  return NEW(BoolType)();
}
```

Typechecking

`_false` | `_true`

BoolExpr
val

Every boolean expression has type **`Bool`**

```
LiteralExpr::LiteralExpr(PTR(Val) val, PTR(Type) type) {
  this->val = val;
  this->type = type;
}

PTR(Type) LiteralExpr::typecheck() {
  return type;
}
```

Typechecking

⟨expr⟩ + ⟨expr⟩

AddExpr
lhs
rhs

A + will produce a number, but only if each ⟨expr⟩ is a number

So, type is **Num** if and only if the ⟨expr⟩s have type **Num**

```
PTR(Type) AddExpr::typecheck() {
  if (!lhs->typecheck()->equals(NEW(NumType)())
      || !rhs->typecheck()->equals(NEW(NumType)())
    throw runtime_error("no type");
  return NEW(NumType)();
}
```

Typechecking

⟨expr⟩ + ⟨expr⟩

AddExpr
lhs
rhs

A + will produce a number, but only if each ⟨expr⟩ is a number

```
PTR(Type) BinaryOpExpr::typecheck() {
  return op->result_type(lhs->typecheck(),
                         rhs->typecheck());
}
```

```
PTR(Type) BinaryNumOp::result_type(PTR(Type) lhs_type, PTR(Type) rhs_type) {
  if (lhs_type->equals(NEW(NumType)())
      && rhs_type->equals(NEW(NumType)()))
    return NEW(NumType)();
  throw std::runtime_error("no type");
}
```

Typechecking

`_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩

IfExpr
test_part
then_part
else_part

First ⟨expr⟩ must produce a boolean

So, the first ⟨expr⟩ must have type `Bool`

```
PTR(Type) IfExpr::typecheck() {
  if (!test_part->typecheck()->equals(NEW(BoolType)()))
    throw runtime_error("no type");
  ....
}
```

Typechecking

_if ⟨expr⟩ **_then** ⟨expr⟩ **_else** ⟨expr⟩

IfExpr
test_part
then_part
else_part

If the test ⟨expr⟩ produces **_true**, the result is the **_then** ⟨expr⟩

➡ type of whole **_if** must match **_then** ⟨expr⟩'s type

Typechecking

_if ⟨expr⟩ **_then** ⟨expr⟩ **_else** ⟨expr⟩

IfExpr
test_part
then_part
else_part

If the test ⟨expr⟩ produces **_true**, the result is the **_then** ⟨expr⟩

➡ type of whole **_if** must match **_then** ⟨expr⟩'s type

If the test ⟨expr⟩ produces **_false**, the result is the **_else** ⟨expr⟩

➡ type of whole **_if** must match **_else** ⟨expr⟩'s type

Typechecking

_if ⟨expr⟩ **_then** ⟨expr⟩ **_else** ⟨expr⟩

IfExpr
test_part
then_part
else_part

If the test ⟨expr⟩ produces **_true**, the result is the **_then** ⟨expr⟩

➡ type of whole **_if** must match **_then** ⟨expr⟩'s type

If the test ⟨expr⟩ produces **_false**, the result is the **_else** ⟨expr⟩

➡ type of whole **_if** must match **_else** ⟨expr⟩'s type

The type checker cannot interpret the test ⟨expr⟩!

Typechecking

`_if` ⟨expr⟩ **`_then`** ⟨expr⟩ **`_else`** ⟨expr⟩

`_true`

`IfExpr`
`test_part`
`then_part`
`else_part`

If the test ⟨expr⟩ produces **`_true`**, the result is the **`_then`** ⟨expr⟩

➡ type of whole **`_if`** must match **`_then`** ⟨expr⟩'s type

If the test ⟨expr⟩ produces **`_false`**, the result is the **`_else`** ⟨expr⟩

➡ type of whole **`_if`** must match **`_else`** ⟨expr⟩'s type

The type checker cannot interpret the test ⟨expr⟩!

Typechecking

`_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩

`0 == 1`

IfExpr
test_part
then_part
else_part

If the test ⟨expr⟩ produces `_true`, the result is the `_then` ⟨expr⟩

➡ type of whole `_if` must match `_then` ⟨expr⟩'s type

If the test ⟨expr⟩ produces `_false`, the result is the `_else` ⟨expr⟩

➡ type of whole `_if` must match `_else` ⟨expr⟩'s type

The type checker cannot interpret the test ⟨expr⟩!

Typechecking

`_if` ⟨expr⟩ **`_then`** ⟨expr⟩ **`_else`** ⟨expr⟩

`x == 1`

`IfExpr`
`test_part`
`then_part`
`else_part`

If the test ⟨expr⟩ produces **`_true`**, the result is the **`_then`** ⟨expr⟩

➡ type of whole **`_if`** must match **`_then`** ⟨expr⟩'s type

If the test ⟨expr⟩ produces **`_false`**, the result is the **`_else`** ⟨expr⟩

➡ type of whole **`_if`** must match **`_else`** ⟨expr⟩'s type

The type checker cannot interpret the test ⟨expr⟩!

Typechecking

`_if` ⟨expr⟩ `_then` ⟨expr⟩ `_else` ⟨expr⟩

`f(10)`

IfExpr
test_part
then_part
else_part

If the test ⟨expr⟩ produces `_true`, the result is the `_then` ⟨expr⟩

➡ type of whole `_if` must match `_then` ⟨expr⟩'s type

If the test ⟨expr⟩ produces `_false`, the result is the `_else` ⟨expr⟩

➡ type of whole `_if` must match `_else` ⟨expr⟩'s type

The type checker cannot interpret the test ⟨expr⟩!

Typechecking

_if ⟨expr⟩ **_then** ⟨expr⟩ **_else** ⟨expr⟩

IfExpr
test_part
then_part
else_part

The **_then** ⟨expr⟩ and **_else** ⟨expr⟩ must have the same type

This program is rejected even though it would *not* raise an exception:

```
_if _true _then 1 _else _true
```

Fundamentally, a type system must reject some "good" programs

Typechecking

_if ⟨expr⟩ **_then** ⟨expr⟩ **_else** ⟨expr⟩

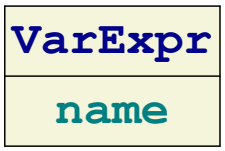
IfExpr
test_part
then_part
else_part

The **_then** ⟨expr⟩ and **_else** ⟨expr⟩ must have the same type

```
PTR(Type) IfExpr::typecheck() {
  if (!test_part->typecheck()->equals(NEW(BoolType)()))
    throw runtime_error("no type");
  PTR(type) then_type = then_part->typecheck();
  PTR(type) else_type = else_part->typecheck();
  if (!then_type->equals(else_type))
    throw runtime_error("no type");
  return then_type;
}
```

Typechecking

⟨variable⟩



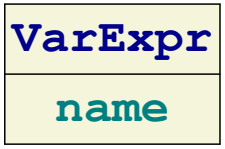
Result depends on a surrounding `_let` or `_fun` or `_letrec`

The `typecheck` method needs a *type environment*:

```
PTR(Val)  Expr::interp(PTR(Env<Val>) env);
PTR(Type) Expr::typecheck(PTR(Env<Type>) tenv);
```

Typechecking

⟨variable⟩



Result depends on a surrounding `_let` or `_fun` or `_letrec`

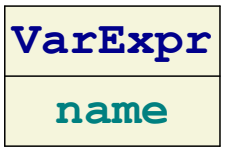
The `typecheck` method needs a *type environment*:

Change `Env` to a template

```
PTR(Val)  Expr::interp(PTR(Env<Val>) env);
PTR(Type) Expr::typecheck(PTR(Env<Type>) tenv);
```

Typechecking

⟨variable⟩



Result depends on a surrounding `_let` or `_fun` or `_letrec`

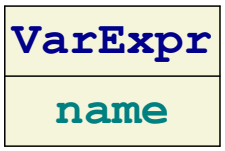
The `typecheck` method needs a *type environment*:

```
PTR(Val)  Expr::interp(PTR(Env<Val>) env);
PTR(Type) Expr::typecheck(PTR(Env<Type>) tenv);
```

```
PTR(Type) VarExpr::typecheck(PTR(Env<Type>) tenv) {
  return tenv->lookup(name);
}
```

Typechecking

⟨variable⟩



Result depends on a surrounding `_let` or `_fun` or `_letrec`

The `typecheck` method needs a *type environment*:

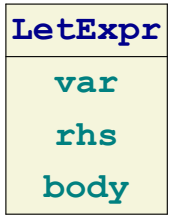
```
PTR(Val)  Expr::interp(PTR(Env<Val>) env);
PTR(Type) Expr::typecheck(PTR(Env<Type>) tenv);
```

```
PTR(Type) VarExpr::typecheck(PTR(Env<Type>) tenv) {
  return tenv->lookup(name);
}
```

Free variables rejected by checker

Typechecking

`_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩



The `_let` expression produces whatever the `_in` ⟨expr⟩ produces

Type of = ⟨expr⟩ determines type of ⟨variable⟩

Type of `_in` ⟨expr⟩ determines type of whole `_let`

```
PTR(Type) LetExpr::typecheck(PTR(Env<Type>) tenv) {
  PTR(Type) rhs_type = rhs->typecheck(tenv);
  return body->typecheck(NEW(ExtendedEnv<Type>)(var, rhs_type, tenv));
}
```

Typechecking

`_letrec` ⟨variable⟩ `:` ⟨type⟩ `=` ⟨fun-expr⟩ `_in` ⟨expr⟩

LetrecExpr
var
rhs_type
rhs
body

Like `_let`, but ⟨fun-expr⟩ must match the declared ⟨type⟩

```
PTR(Type) LetrecExpr::typecheck(PTR(Env<Type>) tenv) {
  PTR(Env<Type>) new_env = NEW(ExtendedEnv<Type>)(var, rhs_type, tenv);
  if (!rhs->typecheck(new_env)->equals(rhs_type))
    throw std::runtime_error("no type");
  return body->typecheck(new_env);
}
```

Typechecking

`_fun` `(` ⟨variable⟩ `:` ⟨type⟩ `)` ⟨expr⟩

FunExpr
formal_arg
arg_type
body

Produces a function value

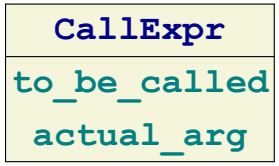
Type is a `->` with

- argument type as ⟨type⟩
- result type as type of ⟨expr⟩

```
PTR(Type) FunExpr::typecheck(PTR(Env<Type>) tenv) {
  PTR(Env<Type>) new_env = NEW(ExtendedEnv<Type>)(formal_arg, arg_type, tenv);
  PTR(Type) body_type = body->typecheck(new_env);
  return NEW(FunType)(arg_type, body_type);
}
```

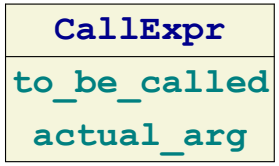
Typechecking

⟨expr⟩ **(** ⟨expr⟩ **)**



Typechecking

⟨expr⟩ **(** ⟨expr⟩ **)**



Produces whatever the function returns, assuming that

- the first ⟨expr⟩ produces a function
- the function is happy with the argument ⟨expr⟩'s value

Typechecking

(⟨type⟩ -> ⟨type⟩)

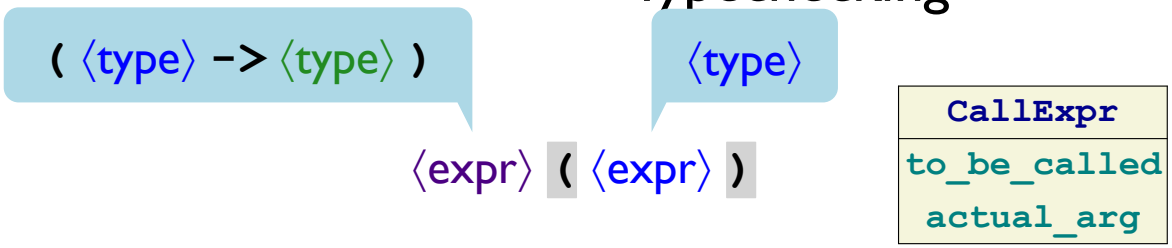
⟨expr⟩ (⟨expr⟩)

CallExpr
to_be_called
actual_arg

Produces whatever the function returns, assuming that

- the first ⟨expr⟩ produces a function
- the function is happy with the argument ⟨expr⟩'s value

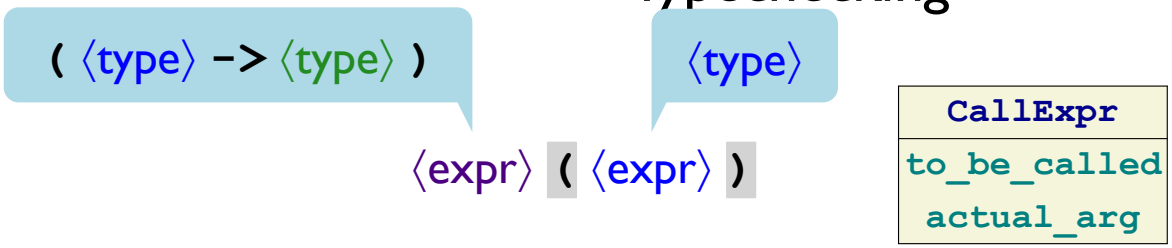
Typechecking



Produces whatever the function returns, assuming that

- the first ⟨expr⟩ produces a function
- the function is happy with the argument ⟨expr⟩'s value

Typechecking

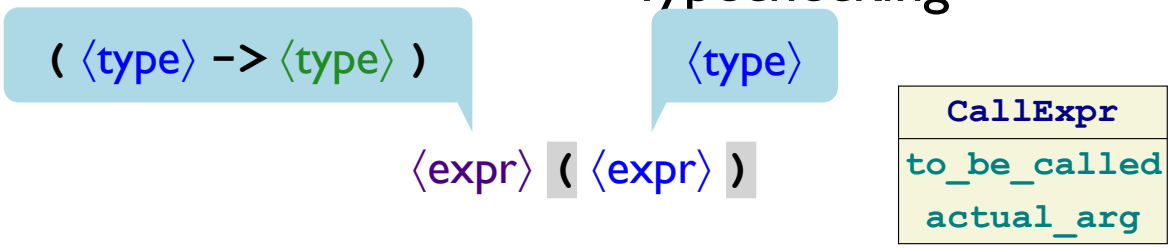


Produces whatever the function returns, assuming that

- the first ⟨expr⟩ produces a function
- the function is happy with the argument ⟨expr⟩'s value

```
PTR(Type) CallExpr::typecheck(PTR(Env<Type>) tenv) {
  PTR(Type) to_be_called_type = to_be_called->typecheck(tenv);
  PTR(Type) actual_arg_type = actual_arg->typecheck(tenv);
  return to_be_called_type->check_call(actual_arg_type);
}
```

Typechecking



Produces whatever the function returns, assuming that

- the first ⟨expr⟩ produces a function
- the function is happy with the argument ⟨expr⟩'s value

```
PTR(Type) Type::check_call(PTR(Type) arg_type) {
  throw std::runtime_error("no type");
}
```

```
PTR(Type) FunType::check_call(PTR(Type) arg_type) {
  if (this->arg_type->equals(arg_type))
    return result_type;
  throw std::runtime_error("no type");
}
```

Typecheck before Interpret

```
int main() {
  ....
  PTR(Type) type = e->typecheck(Env<Type>::empty);

  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<Val>::empty)->to_string() << "\n";
  ....
}
```

Typed Fibonacci

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