# Notes on PoPL

Apiros3

First Version : Apr 24, 2025
Last Update : Apr 24, 2025

## Contents

# 1 Definitional Interpreter

In this section we aim to explain the concept and fundamentals of a definitional interpreter under various contexts.

We will base the language to be implemented to be based off the ”Fun” programming language as specificed by the Principles of Programming Languages course at Oxford University.

## 1.1 Defining Fun

The first step in describing a fixed program is to specify the set of legal phrases (the concrete syntax), then by describing in the language that interprets the program a set of trees that capture the structure of legal phrases (the abstract syntax). For the rest of the section, we will form a Haskell datatype that captures the structure of these legal phrases.

At the simplest level, we have a function

```
parse :: String → Phrase
```

where the `String` is any line of text in the concrete syntax, then produces a corresponding tree in the abstract syntax, which we give the type `Phrase`.

Often the abstract syntax is much more simple than the concrete syntax, as the concrete syntax allows for convinient abbreviations (syntactic sugars).

Basic fun does not have type-checking, so we regard the set of valid expressions to be produced by a context free grammar, rejecting those which do not ”make sense”.

### 1.1.1 Abstract Syntax

The abstract syntax of a language can be expressed as a collection of mutually dependent datatype definitions. In Fun, there is `Expr` for expressions, `Defn` for definitions, and `Phrase` for top level phrases.

```
data Expr =
      Number Integer
    | Variable Ident
    | Apply Expr [Expr]
    | If Expr Expr Expr
    | Lambda [Ident] Expr
    | Let Defn Expr
```

Note that we can have functions with no arguemnts.

The definitions that appear after `let` also appear in the abstract syntax as

```
data Defn =
      Val Ident Expr
    | Rec Ident Expr
```

which correspond to giving variables denoted by `Ident` expressions in `Expr`.

In this way, the concrete form val $x(x_1, \ldots, x_n)$ $e$ is syntactic sugar for val $x$ $=$ lambda$(x_1, \ldots, x_n)$ $e$. We use empty “()” if the function has no inputs, and the constructor `Rec` is for a definition that starts with a lambda (but is not enforced at the datatype level).

The top-level phrase that is typed in the prompt (or included as code in fun) is either an expression which is to evaluated, or a definition to be added into the environment. So,

```
data Phrase =
      Calculate Expr
    | Define Defn
```

**Remark 1.1.1.** In the abstract syntax for fun, identifiers (Ident) are represented by strings. This limits efficiencies, and in more optimized languages have indexing into a global list of identifiers (and thus can avoid string comparisons).

### 1.1.2 Interpreter for Fun

The main component of an interpreter is a function `eval` which takes an abstract syntax tree with an environment and turns it into a value of that expression. Specifically,

```
eval :: Expr → Env → Value
```

where `Env` is the type of environments which mapps identifiers to values, with `Value` representing possible values computed by Fun programs.

At the simplest level, values are denoted by

```
data Value =
      Function ([Value] → Value)
    | IntVal Integer
    | BoolVal Bool
    | Nil
    | Cons Value Value
```

and environments are just

```
type Env = Environment Value
```

Environments is just an abstract data type which maps identifiers to values of some type $\delta$, where in Fun we take $\delta =$ `Value`.

We take the standard mapping constructors

```
type Environment δ
empty_env :: Environment δ
find ::      Environment δ → Ident → δ
define ::    Environment δ → Ident → δ → Environment δ
```

We write $find\ env\ x$ for the value to which $x$ is bound in the environment $env$, and the interpretor gives an error if $x$ is not bound to any value in $env$.

We also write `define env x v` for the environment that agrees with `env` apart from mapping $x$ to $v$, hiding any binding of $x$ from before. As notation, we write `env` $\oplus (x, v)$ for `define env x v`.

For convinience, we also define

```
make_env :: [(Ident, δ)] → Environment δ
```

such that

```
make_env [(x₁, v₁), ..., (xₙ, vₙ)] = empty_env ⊕ (x₁, v₁) ⊕···⊕ (xₙ, vₙ)
```

The function `eval` is one of the four main functions that makes the interpreter. The others are

```
apply :: Value → [Value] → Value
```

which applies a function to its list of arguemnts and produces the value returned by that function

```
abstract :: [Ident] → Expr → Env → Value
```

which forms a function value from a lambda expression

```
elab :: Defn → Env → Env
```

which elaborates a definition, producing a new environment in which the new name has been defined.

The entire process of interpretation starts with a call

```
eval exp init_env
```

where `exp` is the abstract syntax tree for an expression that has been input, and `init_env` is the initial environment.

We can define env by pattern matching on `Expr`, with

```
eval (Number n) env = IntVal n
eval (Variable x) env = find env x
eval (If e1 e2 e3) env =
    case eval e1 env of
        BoolVal True → eval e2 env
        BoolVal False → eval e3 env
        _ → error "boolean required in conditional"
```

The error message helps with dealing with expressions **if** $e_1$ **then** $e_2$ **else** $e_3$ where $e_1$ does not evaluate to a Boolean. In the `Apply` case, we need to evaluate the arguemnts first, so we do this by

```
eval (Apply f es) env =
    apply (eval f env) (map ev es)
        where ev e1 = eval e1 env
```

The inner call `apply` is simply given by

```
apply (Function f) args = f args
apply _            args = error "applying a non-function"
```

The `Lambda` is a way of creating an abstraction, which should evaluate to a function. We can define this by

```
eval (Lambda xs e1) env = abstract xs e1 env
```

where `abstract` is a function that binds the parameters `xs` to the arguments `args` then evaluates the function body `e`. That is,

```
abstract xs e env =
    Function f
    where f args = eval e (defargs env xs args)
```

Where `defargs :: Env -> [Ident] -> [Value] -> Env` is defined such that

```
defargs env0 [x1, ..., xn] [v1, ..., vn]
    = env0 ⊕ (x1, v1) ⊕···⊕ (xn, vn)
```

where `env` $\oplus$ `(x, v)` $=$ `define env x v`. Finally, we want to work with expressions of the form `let d in e`$_1$ (or `Let d e`), which can be given by extending the environment according to the definition `d`, then evaluate the expression `e`.

That is,

```
eval (Let d e1) env = eval e1 (elab d env)
```

Elaborating simply extends the definition, which we can do by

```
elab (Val x e) env = define env x (eval e env)
elab (Rec x (Lambda xs e1)) env =
    env' where env' = define env x (abstract xs e1 env')
elab (Rec x _) env =
    error "RHS of letrec must be a lambda"
```

Note the definition of `env'` uses recursion on itself, where recursion is modelled by recursion in Haskell.

Finally, we can define primitives which are definitions given in `init_env`, for instance the identifier "+" is bound to the function value `Function plus` where

```
plus [IntVal a, IntVal b] = IntVal (a + b)
```

and the Fun expression $\texttt{e}_1 + \texttt{e}_2$ has the abstract syntax

```
Apply (Variable "+") [e1, e2]
```

hence its value is obtained by looking up "+" in the environment and applying the resulting function to the values of the two arguments.

**Remark 1.1.2.** Consequently, by redefining "+", we can change our standard interpretation of the primitive operator, as there is no distinction between primitive in the environment and those which are overwritten.

## 1.2 Memory

We add assignable variables, language sequencing, and while loops.

We have `new()`, which creates assignment variables, returning its address.

```
>>> val a = new();;
--- a = <address 1>
```

Then, we can set the contents of it by assignment

```
>>> a := 3;;
--> 3
```

The value of the assignment takes the expression `x := E` where `E` is the value of the expression `E`. We can then retrieve it's contents via the ! operator :

```
>>> !a;;
--> 3
```

Language sequencing will be written as $\texttt{e}_1\texttt{;}\ \texttt{e}_2$, where a while loop will be written $\texttt{while e}_1\ \texttt{do e}_2$.

Then, we can write functions like factorial imperatively :

```
val fac(n) =
    let val k = new() in
    let val r = new() in
    k := n; r := 1;
    while !k > 0 do
```

```
          (r := !r * !k; k := !k - 1);
      !r;;
```

In the above code, `n` is a ‘constant’ whose value never changes, but `k` and `r` are mutable by the assignment construct.

We implement a memory to map vairables to contents. They support the following

```
type Memory α
type Location

contents :: Memory α → Location → α
update :: Memory α → Location → α → Memory α
fresh :: Memory α → (Location, Memory α)
```

Roughly, `contents` gives the content that is being stored at a location in memory. The `update` gives a new memory that is the same as the previous, updated to the location and $\alpha$ arguments. The function `fresh` creates and returns a fresh location such that if `(a, m') = fresh m`, then `a` is a location that is unused in `m`, and `m'` is a copy of `m` modified so that the location is now regarded as being in use. Thus if we have

```
let (a, m') = fresh m in
let (b, m'') = fresh m' in ...
```

then `a` and `b` are different locations. Note that as the implementation of memory is backed by a functional metalanguage, this imperative equivalent is indeed much slower than the functional equivalent.

In Fun, memories store items of type `Value`, written

```
type Mem = Memory Value
```

We also redefine the evaluation function :

```
eval :: Expr → Env → Mem → (Value, Mem)
eval (Number n) env mem = (IntVal n, mem)
eval (Variable x) env mem = (find env x, mem)
eval (If e1 e2 e3) env mem =
    let (b, mem') = eval e1 env mem in
    case b of
        BoolVal True → eval e2 env mem'
        BoolVal False → eval e3 env mem'
        _ → error "Boolean required in conditional"
eval (Apply f es) env mem =
    let (fv, mem') = eval f env mem in
    let (args, mem'') = evalargs es env mem' in
    apply fv args mem''
```

with helper functions

```
evalargs :: [Expr] → Env → Mem → ([Value], Mem)
evalargs [] env mem = ([], mem)
evalargs (e : es) env mem =
    let (v, mem1) = eval e env mem in
    let (vs, mem2) = evalargs es env mem1 in
```

```
      (v : vs, mem₂)
```

### 1.3 Output

## 2 Monads

### 2.1 Monad Laws

In the scope of Fun, a monad takes some $\alpha \to M\ \alpha$, equipped with an operator $\triangleright$ and a function `result` such that the following laws hold :

```
   - (xm ▷ f) ▷ g = xm ▷ (λ x → f x ▷ g)
   - (result x) ▷ f = f x
   - xm ▷ result = xm
```

These outline associativity and identity rules.

Here, $\triangleright$ is an operator which takes the evaluated value of the left, and passes the return value to its right. Thus,

$$\triangleright : M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta$$

We can rewrite this in more succinct Haskell notation, by considering the function $^* : (\alpha \to M\beta) \to M\ \alpha \to M\ \beta$ by $^*$ `f xm = xm` $\triangleright$ `f`. Writing $\texttt{f}^*$ for shorthand, we can simplify our laws to

```
   - g* · f* = (g* · f)*
   - f* · result = f
   - result* = id
```

### 2.2 Monads in Fun

In Fun, Monads allow structure in being able to pass some background information without explicitly referring to them. In the case of Memory and Output, we can pass these informations without explicitly writing their outputs by using $\triangleright$ to implicitly compute them being updated.

#### 2.2.1 Memory and Output

We first note the types for `eval` and `elab`,

```
   eval₁ :: Expr → Env → Mem → (Value, Mem)
   elab₁ :: Defn → Env → Mem → (Env, Mem)
   eval₂ :: Expr → Env → (String, Value)
   elab₂ :: Defn → Env → (String, Env)
```

Using currying and higher-order types, we can reduce these to

```
   eval :: Expr → Env → M Value
   elab :: Defn → Env → M Env
```

where

```
   type M₁ α = Mem → (α, Mem)
   type M₂ α = (String, α)
```

Now consider some examples of the evaluation functions for the two languages.

In the ccase for numeric constants, we have

```
eval₁ (Number n) env = (λ mem → (IntVal n, mem))
eval₂ (Number n) env = ("", Intval n)
```

we can wrap this in a simple function such that

```
eval₁ (Number n) env = result (IntVal n)
```

where

```
result :: α → M α
result₁ x = (λ mem → (x, mem)) :: M₁ Value
result₂ x = ("", x)             :: M₂ Value
```

The `result` function essentially does what is considered doing nothing, when mapping to inside the monad. In the case of FunMem, this is mapping to the same memory, while in the case of FunOut, this is producing the empty string.

Similarly,

```
eval (Variable x) env = result (find env x)
```

Consider now a more complex example given by the conditional, where we have (omitting error messages)

```
eval₁ (If e₁ e₂ e₃) env mem =
    let (b, mem') = eval₁ e₁ env mem in
    case b of
        BoolVal True → eval₁ e₂ env mem'
        BoolVal False → eval₁ e₃ env mem'

eval₂ (If e₁ e₂ e₃) env =
    let (out₁, b) = eval₂ e₁ env in
    let (out₂, r) =
        case b of
            BoolVal True → eval₂ e₂ env
            BoolVal False → eval₂ e₃ env in
    (out₁ ++ out₂, r)
```

We can collapse this into a single operator,

```
eval (If e₁ e₂ e₃) env =
    eval₁ e₁ env ▷ (λ b →
        case b of
            BoolVal True → eval e₂ env
            BoolVal False → eval e₃ env
    )
```

where $\triangleright$ combines in sequence of calculations, passing the result of the first operand to the second.

For the first case, we note that the memory after the first evaluation is being passed, such that

```
xm ▷₁ f =
    let (x, mem') = xm mem in f x mem'
```

For the second case, we simply add the string produced through the output, which we can do by

```
xm ▷₂ f =
    let (out₁, x) = xm in
    let (out₂, y) = f x in
    (out₁ ++ out₂, y)
```

As another instance, consider

```
eval₁ (Let d e₁) env mem =
    let (env', mem') = elab₁ d env mem in
    eval₁ e₁ env' mem'

eval₂ (Let d e₁) env =
    let (env', out₁) = elab₂ d env in
    let (x, out₂) = eval₂ e₁ env' in
    (x, out₁ ++ out₂)
```

The evaluation can be viewed as a function of the form `Env` $\to$ $M$ `Value`, which passes the new environment into the evaluation. Thus, we can write the evaluation for let as,

```
eval (Let d e) env =
    elab d env ▷ (λ env' → eval e₁ env')
```

### 2.3 Monad of Memory in Fun

The monad of memeory can be written simply as

```
type M α = Mem → (α, Mem)

result :: α → M α
result x mem = (x, mem)

▷ :: M α → (α → M β) → M β
(xm ▷ f) mem =
    let (x, mem₁) = xm mem in f x mem₁
```

#### 2.3.1 Principal Functions

Additionally, a choice of monad usually comes with a few operations to implement specific language features. For a memory, we have `get` that retrieves the value stored in a location, `put` that modifies the contents of a location, and `new` that allocates a fresh location.

Explicitly,

```
get :: Location → M Value
get a mem = (contents mem a, mem)

put :: Location → Value → M ()
put a v mem = ((), update mem a v)

new :: M Location
new mem = let (a, mem') = fresh mem in (a, mem')
```

Alternatively, `new` = `fresh`. These operations are placed so that the details of the computational model (eg exception + memory) can be implmented by only changing the inner implementation of the above functions.

The term "Semantic Domains" refers to the types that are used in the interpreter.

The semantic domains for assignment variables can be written as

```
data Value =
    // Original Alternatives
    | Addr Location
    | Function ([Value] → M Value)
```

where the existence of the output type of the function being `M Value` represents the fact the input body can interact with the memory. We also fix the decision that `Value`s are what names are bound to in the environment, and the same domain of vlaues that can be stored in memory. That is,

```
type Env = Environment Value
type Mem = Memory Value
```

We also redefine the standard functions `eval`, `abstract`, `apply`, and `elab`.

That is,

```
eval :: Expr → Env → M Value
eval (Number n) env = result (IntVal n)
eval (Variable x) env = result (find env x)
eval (Apply f es) env =
    eval f env ▷ (λ fv →
        evalargs es env ▷ (λ args →
            apply fv args
        )
    )
eval (lambda xs e₁) env =
    result (abstract xs e₁ env)
eval (If e₁ e₂ e₃) env =
    eval e₁ env ▷ (λ b →
        case b of
            BoolVal True → eval e₂ env
            BoolVal False → eval e₃ env
            _ → error "Boolean required in conditional"
    )
eval (Let d e₁) env =
    elab d env ▷ (λ env' → eval e₁ env')
```

The above is enough to define a language that is purely functional, but we can add imperative features like sequencing and while loops by

```
eval (Sequence e₁ e₂) env =
    eval e₁ env ▷ (λ v → eval e₂ env)
eval (While e₁ e₂) env = u
    where
        u = eval e₁ env ▷ (λ v₁ →
            case v₁ of
```

```
                    BoolVal True → eval e₂ env ▷ (λ v₂ u)
                    BoolVal False → result Nil
                    _ → error "Boolean required in while loop"
            )
```

where we also have standard helper functions:

```
evalargs :: [Expr] → Env → M [Value]
evalargs [] env = result []
evalargs (e : es) env =
    eval e env ▷ (λ v → evalargs es env ▷ (λ vs → result (v :: vs)))
```

Alternatively,

```
mapm :: (α → M β) → [α] → M [β]
mapm f [] = result []
mapm f (e : es) = f e ▷ (λ v → mapm f es ▷ (λ vs → result (v :: vs)))

evalargs xs env = mapm (λ e → eval e env) xs
```

The clause for `While` defines the meaning of loops as recursion.

Languages also have constructs that are unique to them, shared only with closely related ones. In Fun with memory, the only such construct is assignment, which uses the `put` operation.

```
eval (Assign e₁ e₂) env =
    eval e₁ env ▷ (λ v₁ →
        case v₁ of
            Addr a →
                eval e₂ env ▷ (λ v₂ → put a v₂ ▷ (λ () → result v₂))
            _ → error "assigning to a non-variable"
    )
```

Note that in the assignment $e_1 := e_2$, $e_1$ must be an address, updated with the value of $e_2$, and the same value is yielded as the value of the assignment itself.

Moving to `apply` and `abstract`,

```
abstract :: [Ident] → Expr → Env → Value
abstract xs e env =
    Function (λ args → eval e (defargs env xs args))

apply :: Value → [Value] → M Value
apply (Function f) args = f args
apply _ args = error "applying a non-function"
```

The type of `abstract` does not change, as forming a function value does not require interactions with the memory. The type of `apply` does change to reflect the different type of the function that is wrapped in `Function`.

The `elab` processes declaration, which should have similar definition in most interpreters. Elaborating may involve an interaction with the memory, as in `val x = e`, evaluation of the right side may need to use the memory. Naturally,

```
elab :: Defn → Env → M Env
elab (Val x e) env =
    eval e env ▷ (λ v → result (define env x v))
elab (Rec x (Lambda xs e1)) env =
    result env' where env' = define env x (abstract xs e1 env')
elab (Rec x _) env =
    error "RHS of letrec must be a lambda"
```

### 2.3.2 Primitives

Standard primitives are shared with the purely functional language, but we also need to insert a call to result, reflecting the fact primitives do not need to interact with the memory.

The primitives specific to the language with assignment variables are "`!x`" which fetches the contents of the the memory cell named by `x`, and `new`(), which allocates a fresh, uninitialised cell.

We thus have

```
primitive "!" (λ [Addr a] → get a)
primitive "new" (λ [] → new ▷ (λ a → result (Addr a)))
```

The syntax allows the expression !`x` to be written without parenthesis. Note this is a Haskell feature that allows lambda on a specific constructor without casing.

The initial environment is given by

```
init_env :: Env
init_env =
    make_env [
        // Usual constants + primitives
        // primitives from above
    ]
    where
        constant x v = (x, v)
        primitive x f = (x, Function (primwrap x f))
        pureprim x f = primitive x (result · f)
```

### 2.3.3 Main Program

We give the global state and obey as follows

```
type GloState = (Env, Mem)
obey :: Phrase → GloState → (String, GloState)
obey (Calculate exp) (env, mem) =
    let (v, mem') = eval exp env mem in
        (print_value v, (env, mem'))
obey (Define def) (env, mem) =
    let x  = def_lhs def in
    let (env', mem') = elab def env mem in
    (print_defn env' x, (env', mem'))
```

The main code is then given by

```
module FunMonad(main) where
    // common imports
import Memory
infixl 1 ▷
```

We import Memory in order to implement assignable variables. We add in the memory monad, principle functions (including catch-all for eval), primitives, initial environment, instance declarations (of Eq and Show).

The main program based on `obey` completes the program :

```
main = dialog funParser obey (init_env, init_mem)
```

## 2.4 Monadic Equivalence

With pure fun, for any expression `e`, we expect the program

```
let val x = e in x
```

to be equivalent to `e`.

To prove this, first note the meaning of the constructs :

```
eval (Let d e₁) env = elab d env ▷ (λ env' → eval e₁ env')
eval (Variable x) env = result (find env x)
elab (Val x e) env = eval e env ▷ (λ v → result (define env x v))
```

Writing `LHS` to represent the code above, we have

```
eval LHS env = (vm ▷ f) ▷ g
    where
        vm = eval e env
        f v = result (define env x v)
        g env' = result (find env' x)
```

Noting this is just an expansion of `eval (Let (Val x e) (Variable x)) env`.

Applying the associative law, we have

```
(vm ▷ f) ▷ g = vm ▷ (λ v → f v ▷ g)
```

Now, given `v` and `env`,

```
f v ▷ g = result env' ▷ g = g env' = result (find env' x) = result v
```

on the assumption that `find (define env x v) x = v`. Now, applying the right identity,

```
eval LHS env = eval e env ▷ (λ v → result v) = eval e env
```

## 2.5 Exceptions

We define a notion of "`orelse`", which runs the first argument, returns it if successive, and calls the second if it fails. For instance, this is useful when one wants to treat -1 as the error term.

Consider

```
val index(x, xs) =
    let rec search(ys) =
        if ys = nil then fail()
```

```
          else if head(ys) = x then 0
          else search(tail(ys)) + 1 in
      search(xs) orelse -1;;
```

Then, `index(x,xs)` returns -1 if the value does not appear in the list.

We can define a monad that captures failure as follows

```
data M α = Ok α | Fail
```

We make this into a monad by defining the standard functions associated with it

```
result :: α → M α
result x = Ok x

(▷) :: M α → (α → M β) → M β
(Ok x) ▷ f = f x
Fail ▷ f = Fail
```

with associated operations to implement the `fail()` primitive and the `orelse` construct as follows

```
failure :: M α
failure = Fail

orelse :: M α → M α → M α
orelse (Ok x) ym = Ok m
orelse Fail ym = ym
```

The semantic domain `Value` contains the same kinds of values as in pure Fun, as there are no values introduced, just the possibility of a failing evaluation. The type used to represent functions changes, to incorporate the monad to reflect the fact the function body may fail. So,

```
data Value =
    // Standard values
    | Function ([Value] → M Value)
```

The `eval` function then comes with a clause for the `orelse` construct, with the abstract syntax

```
data Expr = ...
    | OrElse Expr Expr
```

Then, the evaluation for this is simply

```
eval (OrElse e₁ e₂) env =
    orelse (eval e₁ env) (eval e₂ env)
```

The primitive fail is simple :

```
primitive "fail" (λ [] → failure)
```

Finally, at the top-level, we have

```
obey :: Phrase → Env → (String, Env)
obey (Calculate exp) env =
    case (eval exp env) of
        Ok v → (print_value v, env)
```

```
        Fail → ("*failed*", env)
obey (Define def) env =
    let x = def_lhs def in
    case elab def env of
        Ok env' → (print_defn env' x, env')
        Fail → ("*failed*", env)
```

Putting the parser together in the standard way.

The `orelse` definition is dependent on the lazy evaluation of Haskell, as we expect the second arguemnt to not be evaluated unless the first ends in failure. For instance, the epxression

```
3 orelse (let rec loop(n) = loop(n+1) in loop(0))
```

should evaluate to 3 and not infinite recursion. The solution to this is to use **continuations** such that the type `M` $\alpha$ becomes

```
type M α = (α → Answer) → (() → Answer) → Answer
```

for some type Answer, with the idea that `xm ks kf` calls the success continuation `ks` to signal success and failure continuation `kf` if it fails.

Alternatively, we can avoid dependence on Haskell's evaluation by making `M` $\alpha$ into a function type :

```
type M α = () → Maybe α
```

with the standard `Maybe` $\alpha$ = `Just` $\alpha$ | `Nothing`. Then,

```
result :: α → M α
result x = (λ () → Just x)

(▷) :: M α → (α → M β) → M β
xm ▷ f =
    (λ () →
        case xm() of
            Just x → f x ()
            Nothing → Nothing
    )

failure :: M α
failure = (λ () → Nothing)

orelse :: M α → M α → M α
orelse xm ym =
    (λ () →
        case xm() of
            Ok x → Ok x
            Nothing → ym ()
    )
```

This forces arguments to be functions that are called only if their results are needed, allowing expressions to yield an answer without evaluation.

Whether the metalanguage is lazy or not, we must have

```
failure ▷ f = failure
```

thus, `orelse` must be added as an expression, not a primitive, as in the evaluation for `Apply` expressions, we evaluate the arguments, which means if any fail, the entire call fails. Thus, we have no such primitive.

# 3 Simple Domain Theory

We can define an inductive set by simply taking the least set that satisfies certain rule-based properties (constructors). Then, principle of induction says that if this is a monotonic function $F$, given any set $S$ and $F\ S \subseteq S$, $S$ contains at least the least fixed point.

Some things to note: - Tarski's Fixed Point Theorem (for chain cpo) - factorial function as example - continuity

Construction of cpos

- adding $\perp$
- $[X \to Y]$
- $X \times Y$

# 4 Other

## 4.1 On elab with abstract

- some sort of closure, lasyness

- how to think of let, where, how far things are captured

- Env keeps track of variables and which values they map to. This doesn't change, unless some 'let' overrides the previous environment. Mem keeps track of a mapping from addresses to values, which are mutable. Thus, environments map variables to addresses, which map to different values instructed by Mem. Mem only changes when assignments happen.