Notes on Basic Programming Language Concepts

Apiros3

First Version : Jan 29, 2025
Last Update : Apr 12, 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 $x(x_1, \dots, x_n)$ e is syntactic sugar for val x = lambda$(x_1, \dots, x_n)$ 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 δ, 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` ⊕ `(x, v)` = `define env x v`. Finally, we want to work with expressions of the form `let d in e1` (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 $\mathtt{e}_1 + \mathtt{e}_2$ 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 Defunctionalization (d17n)

There are a few problems to cover with the interpreter, as many parts still rely on native features of Haskell. These include

- The Fun interpreter uses higher order functions and type of values include functions from values to values, which make no sense as a mathematical set. Specifically, we have

  ```
  data Value = ... | Function ([Value] → Value) | ...
  ```

  but there is no injection from $[X] \to X$ into X.

- The interpreter uses recursion to deal with the recursive syntax

- The implementation of recursive function definitions involves a recursive value definition in Haskell (as opposed to a recursive function)

- The evaluation strategy (including laziness) is not clear.

Defunctionalization is a whole program transformation that turns higher-roder functional programs into first order ones. This can be split up into steps, although it is a single transformation.

Start with an original program

```
prog x ys = (map aug ys, map sqr ys)
    where
        aug y = x + y
        sqr y = y * y
```

The first step is to **decode the dataflow**, isolating expressions that create or use or pass the value of higher order arguemnts or results, together with a trivial function apply that unwraps them when necessary.

For instance we can do

```
data Func = Func (Integer → Integer)

prog x ys = (map (Func aug) ys, map (Func sqr) ys)
    where
        aug y = x + y
        sqr y = y * y

map :: Func → [Integer] → [Integer]
map f [] = []
map f (x : xs) = apply f x : map f x

apply :: Func → Integer → Integer
apply (Func f) x = f x
```

We now remove lambdas, identifying free variables, creating constructors for each lambda expression (which will be treated by the apply function) so that we have

```
data Func = Sqr | Aug Integer
```

Then, we can rewrite prog as

```
prog x ys =
    (map (Aug x) ys, map Sqr ys)
```

We then adjust apply as an interpreter for the tiny language the Func datatype has become, associating each constructor to the expression it represents. In our case, we have

```
apply :: Func → Integer → Integer
apply (Aug x) y = x + y
apply Sqr y = y * y
```

We can apply d17n to the higher order interpreter for Fun to eliminate the use of functions in the Value type. There are two places in which functions are created, the value of a lambda expression is a function value `abstract xs e env` and each primitive is bound to a different function value specific to a primitive. We introduce two new constructors to Value, each depending on a type that lists all possible primtiives:

```
data Value =
    ...
    | Closure [Ident] Expr Env
    | Primitive Prim

data Prim =
    Plus | Minus ...
    deriving Show

type Env = Environment Value
```

Then `abstract` becomes a simple a simple call

```
abstract :: [Ident] → Expr → Env → Value
abstract xs e env = Closure xs e env
```

The original definition for `abstract` is placed as part of `apply`, used when the value being applied is one of the `Closure` objects. If it is called on a `Primitive`, we depend on a function `primapply` to be defined.

```
apply :: Value → [Value] → Value
apply (Closure xs e env) args =
    eval e (defargs env xs args)
apply (Primitive p) args =
    primapply p args
apply _ _ =
    error "applying to a non-function"
```

Before d17n, primitive names like '+' was bound in the environment to a value `Function plus` where `plus` was a specific function of type `[Value] -> Value`. Instead, we bind '+' to a value `Primitive Plus` where `Plus` is a primitive that is to be dealt with specifically through the `primapply` function.

We add in the initial environment:

```
init_env :: Env
init_env =
    make_env[
        primitive "+" Plus, ...
    ]
    where
        constant x v = (x, v)
        primitive x p = (x, Primitive p)
```

and primapply:

```
primapply :: Prim → [Value] → Value
primapply Plus [IntVal a, IntVal b] = IntVal (a + b)
...
primapply x args =
    error ("bad arguments to primitive " ++ show x ++ ": " ++ showlist
 args)
```

Remark 1.2.1.

- We have saved the environment as part of the Closure, as we care about the environment when it is being abstracted.
- d17n can be applied to any whole program to eliminate higher order functions
- The cyclic structure for recursion becomes clear, as we are effectively making a loop in a pointer-linked data structure:

  ```
  elab (Rec x (Lambda xs e)) env =
      env' where env' = define env x (Closure xs e env')
  ```

Adjusting instances for Show and Eq in Haskell, we solve the first and third problems mentioned at the start of the subsection. The third is solved as we elaborate based on Closure passing, where before we relied on wrapping it in a Function who was defined over Haskell.

Effectively, instead of passing a higher order function, we pass a token that represents the higher order function, and evaluate based on that information, which allows one to avoid making the main function itself to be a higher function.

1.3 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 $\mathtt{e}_1$`;` $\mathtt{e}_2$, where a while loop will be written `while` $\mathtt{e}_1$ `do` $\mathtt{e}_2$.

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 α 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)
```

Any declaration creates a new memory state. In particular, `elab` should take a memory state as an arguement and return another as part of its result, thus defining `let` should chain these results. Specifically,

```
eval (Let d e1) env mem =
    let (env', mem1) = elab d env mem in
        eval e1 env' mem1
```

Then, elaboration looks like

```
elab :: Defn → Env → Mem → (Env, Mem)
elab (Val x e) env mem =
    let (v, mem1)  eval e env mem in
        (define env x v, mem1)
elab (Rec x (Lambda xs e1)) env mem =
    (env', mem) where env' = define env x (abstract xs e1 env')
elab (Rec x _) env mem =
    error "RHS of letrec must be lambda"
```

Abstract does not need to touch memory, as it builds a function object, which if it requires the use of memory, will do so when it is called.

We adjust the Value type to accomodate primitives `new` and '!' as

```
data Value =
    ...
    | Addr Location
    | Function ([Value] → Mem → (Value, Mem))

eval (Assign e₁ e₂) env mem =
    let (v₁, mem') = eval e₁ env mem in
    case v₁ of
        Addr a →
            let (v₂, mem'') = eval e₂ env mem' in
            (v₂, update mem'' a v₂)
        _ → error "assigning to a non-variable"
```

The assignment returns the value of the right hand side, with the effect that the location denoted by the left is updated to contain the same value.

Sequencing can be written like

```
eval (Sequence e₁ e₂) env mem =
    let (v₁, mem') = eval e₁ env mem in eval e₂ env mem'
```

The while loop is written on the knowledge that

```
while e₁ do e₂
```

looks like

```
if e₁ then (e₂; while e₁ do e₂) else nil
```

with

```
eval (While e₁ e₂) env mem = f mem
where
    f mem =
        let (b, mem') = eval e₁ env mem in
        case b of
            BoolVal True → let (v, mem'') = eval e₂ env mem' in f mem''
            BoolVal False → (Nil, mem')
            _ → error "boolean required in while loop"
```

Where we have chosen the program to return `Nil` when the while loop terminates.

We add primitives

```
primitive "new" (λ [] mem →
    let (a, mem') = fresh mem in (Addr a, mem')
)
primitive "!" (λ [Addr a] mem → (contents mem a, mem))
```

The original primitives are adjusted such that memory passes straight through them. We adjust the environment:

```
init_env :: Env
init_env =
    make_env ...
    where
        constant x v = (x, v)
        primitive x f = (x, Function (primwrap x f))
        pureprim x f =
            (x, Function (primwrap x (λ args mem → (f args, mem))))
```

The top level for Fun with memory is

```
obey :: Phrase → GloState → (String, GloState)
```

where `GloState` is the type of global states that are passed from one evaluation to next. With memory, this is

```
type GloState = (Env, Mem)
```

Such that the memory with environment is passed between phrases. Then,

```
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'))
```

With

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

1.4 Output

We can add primitives for output such that evaluating `print(v)` would both yield the value and print it on the terminal. For instance:

```
>>> print(4) + 5;;
4
--> 9
```

Then, we would have

```
eval :: Expr → Env → (String, Value)
```

In the previous case we'd pass the memory to the next evaluation, but in this case a sequence of evaluations would result in a concatenation of the string outputs.

We have a binding for print as a primitive of the function:

```
print :: [Value] → (String, Value)
print [v] = (show v ++ "λn", v)
```

With

```
pureprin x f = (x, Function (primwrap x (λ args → ("", f args))))
```

2 Monads

2.1 Monad Laws

In the scope of Fun, a monad takes some $\alpha \to M\ \alpha$, equipped with an operator ▷ 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, ▷ 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▷f`. Writing `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 ▷ 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

```
eval_1 (Number n) env = result (IntVal n)
```

where

```
result :: α → M α
result_1 x = (λ mem → (x, mem)) :: M_1 Value
result_2 x = ("", x)             :: M_2 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)

```
eval_1 (If e_1 e_2 e_3) env mem =
    let (b, mem') = eval_1 e_1 env mem in
    case b of
        BoolVal True → eval_1 e_2 env mem'
        BoolVal False → eval_1 e_3 env mem'

eval_2 (If e_1 e_2 e_3) env =
    let (out_1, b) = eval_2 e_1 env in
    let (out_2, r) =
        case b of
            BoolVal True → eval_2 e_2 env
            BoolVal False → eval_2 e_3 env in
    (out_1 ++ out_2, r)
```

We can collapse this into a single operator,

```
eval (If e_1 e_2 e_3) env =
    eval_1 e_1 env ▷ (λ b →
        case b of
            BoolVal True → eval e_2 env
            BoolVal False → eval e_3 env
    )
```

where ▷ 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 (out_1, x) = xm in
    let (out_2, y) = f x in
    (out_1 ++ out_2, 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 α` 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 α` into a function type :

```
type M α = () → Maybe α
```

with the standard `Maybe` $\alpha =$ `Just` $\alpha \mid$ `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 Domain Theory

This section aims to cover concepts in domain theory, which is a study on a special kind of posets. There is a great connection between the concepts in domain theory to areas like denotational semantics (the motivation of this field comes initially from Dana Scott's search for denotational semantics of the lambda calculus). We can then equip notions like approximations and convergence by defining a suitable topology in which it models our intuition. Hence, this section has close ties with concepts including category theory, functional programming, and topology. Notes here are mainly based off of the following:

- Barendregt, Syntax and Semantics, Chapter 1
- Abramsky and Jung, Domain Theory
- Winskel, The Formal Semantics of Programming Languages, Chapter 8

3.1 Partial Orders, Suprema, and Continuity

Definition 3.1.1. *A set P with a binary relation $\sqsubseteq$ is a partially ordered set (or poset) if the operation is reflexive, transitive, and antisymmetric.*

Definition 3.1.2. *Given a partially ordered set D, we define a **directed set** to be a nonempty $S \subseteq D$ such that every pair of elements of S has an upper bound in D. That is, for any $x, y \in D$, there exists a $z \in D$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$.*

For instance, a chain, which is a non-empty subset which are totally ordered are directed sets. The chain of natural numbers with their natural orders is indeed a chain, and subsets of posets which are isomorphic to these are called ω-chains.

Definition 3.1.3. *Let D be a partially ordered set (poset) with order $\sqsubseteq$. We define the **least upper bound** (lub) or **supremum** of a subset $S \subseteq D$ denoted $\bigsqcup S$ if it satisfies:*

- *$\bigsqcup S$ is greater than or equal to every element in S; $\forall x \in S,\ x \sqsubseteq \bigsqcup S$*
- *$\bigsqcup S$ is the smallest such element; $\forall y \in D, (\forall x \in S, x \sqsubseteq y) \implies (\bigsqcup S \sqsubseteq y)$*

*We give the dual notion (greatest lower bound) as the **infimum** with symbol $\bigsqcap$.*

Notation 3.1.4. We write $\bigsqcup^{\uparrow} A$ to denote $\bigsqcup A$ if A is directed and has a supremum.

Notation 3.1.5. Given a partially ordered set $(D, \sqsubseteq)$ and a set $X \subseteq D$, we write

- $\uparrow X = \{d \in D \mid \exists x \in X, x \sqsubseteq d\}$
- $\downarrow X = \{d \in D \mid \exists x \in X, d \sqsubseteq x\}$

Where it is clear, we may write $\uparrow x$ to represent $\uparrow \{x\}$ and similarly for $\downarrow x$.

Definition 3.1.6. *We call a set to be an **upper set** if $\uparrow X = X$. If $\downarrow X = X$, then we say that X is a **lower set**. Furthermore, we call an element $x \in D$ to be maximal if there are no elements above it. That is, $\uparrow x \cap D = \{x\}$. An element $x \in D$ is similarly minimal if $\downarrow x \cap D = \{x\}$.*

Proposition 3.1.7. *Let D be a poset such that the following suprema and infima exist. Then,*

1. *$A \subseteq B$ implies $\bigsqcup A \sqsubseteq \bigsqcup B$ and $\bigsqcap B \sqsubseteq \bigsqcap A$*

2. $\bigsqcup A = \bigsqcup(\downarrow A)$ *and* $\bigsqcap A = \bigsqcap(\uparrow A)$

3. $A = \bigcup_{i\in I} A_i$ *implies* $\bigsqcup A = \bigsqcup_{i\in I}(\bigsqcup A_i)$ *and similarly for the infimum*

Proof. The first two cases are obvious. For the third case, note that $\bigsqcup A$ is each above $\bigsqcup A_i$ so by taking the least upper bound, it follows that $\bigsqcup_{i\in I}(\bigsqcup A_i) \sqsubseteq \bigsqcup A$. Conversely, each $a \in A$ is in A_i so is below $\bigsqcup A_i$, which is below $\bigsqcup_{i\in I} \bigsqcup A_i$. Thus, this is an upper bound for A, and $\bigsqcup A$ is the least, giving $\bigsqcup A \sqsubseteq \bigsqcup_{i\in I}(\bigsqcup A_i)$. ■

This means that we are allowed to take supremums of parts of a set and take the whole later, so long as the A_i's cover the entire set.

Definition 3.1.8. *We say that a directed lower set is an* ***ideal****. We say it is principal if it is of the form* $\downarrow x$*. The dual notion is filtered set and principal filter.*

Remark 3.1.9. As $\sqsubseteq$ is reflexive, we have $X \subseteq \uparrow X$. Therefore, to prove X is an upper set, it suffices to show that $\uparrow X \subseteq X$.

Definition 3.1.10. *We say that a partial ordered set* $(D, \sqsubseteq)$ *is an* ***ω-complete partial order*** *(ω-cpo) if every countable ascending chain* $(d_0 \sqsubseteq d_1 \sqsubseteq \dots)$ *has a least upper bound, written* $\bigsqcup\limits_{n\geq 0} d_n$*. Additionally, we say* $(D, \sqsubseteq)$ *is a cpo* ***with bottom*** *or is* ***pointed****, if it has a least element* $\bot \in D$ *(over* $\sqsubseteq$*).*

Definition 3.1.11. *We say that poset D is an* ***directed-complete partial order*** (dcpo) *if every directed subset* $X \subseteq D$ *has a supremum. It is also referred to as the* ***up-complete poset****. We write* $\bigsqcup X$ *for the suprema.*

In general, it is difficult to make a non-dcpo into a dcpo. For instance, the natural numbers by the usual order does not form a dcpo (as it has no supremum). On the other hand, every finite poset is a dcpo.

Definition 3.1.12. *A poset D is a* ***pointed-directed-complte partial order (pointed dcpo)*** *or* ***ccpo*** *is a dcpo with a least element* $\bot \in D$*. Alternatively, it is a poset which has a supremum for every directed or empty subset.*

When we refer to a "cpo" in a proof, we usually mean dcpo, and when $\bot$ is mentioned, a ccpo.

Definition 3.1.13. *A poset D is a* ***$\bigsqcup$-semilattice*** *if the supremum for each pair of elements exists.*

Remark 3.1.14. We dually give notions for infimum as the greatest lower bound, using the $\bigsqcap$ notation. Similar notions apply for definitions like $\bigsqcap$-semilattice.

Definition 3.1.15. *A* ***complete lattice*** *is a poset D where every* $X \subseteq D$ *has a supremum and infimum (where infimum is defined similarly).*

Remark 3.1.16. We can clearly see that a ccpo is a dcpo is a ω-cpo.

It is also important to distinguish between complete partial orders (including dcpo and ccpo) and complete lattices. Notably, we don't force every subset to have a suprema. A complete lattice is a cpo with bottom, as $\bot = \bigsqcup \emptyset$, but not vice versa.

Proposition 3.1.17. *A poset D is a dcpo if and only if each chain in D has a supremum.*

Proof. Uses Axiom of Choice. Proof is out of scope of notes but can be found in [2].

Definition 3.1.18. *Given cpos D, D', we say that a function $f : D \to D'$ is* ***monotonic*** *if*

$$\forall d, d' \in D, d \sqsubseteq d' \implies f(d) \sqsubseteq f(d')$$

Remark 3.1.19. The set $[P \overset{m}{\to} Q]$ of all monotone functions betwene posets ordered pointwise give rise to another poset, the **monotone function space** between P and Q.

Proposition 3.1.20. *Let A be a non-empty subset of a $\bigsqcup$-semilattice for which $\bigsqcup A$ exists. Then,*

$$\bigsqcup\nolimits^{\uparrow}\{\bigsqcup M \mid M \subseteq A \textit{ finite and non-empty}\}$$

Proof. Note that by Proposition 3.1.7, as the set is a cover for A, the suprema are equal. We now need to show that the internal set is directed as $(\bigsqcup A) \sqcup (\bigsqcup B) = \bigsqcup(A \cup B)$ and $A \cup B$ is finite if A and B are both finite. ■

Definition 3.1.21. *Let D be an cpo and $f : D \to D$. We say that $d \in D$ is a* ***fixed point*** *of f if $f(d) = d$. We say it is a* ***prefixed point*** *if $f(d) \sqsubseteq d$*

Proposition 3.1.22. *If D is a complete lattice then every monotone $f : D \to D$ has a fixpoint. The least is*

$$\bigsqcap\{x \in D \mid f(x) \sqsubseteq x\}$$

The largest with

$$\bigsqcup\{x \in D \mid x \sqsubseteq f(x)\}$$

Proof. For the least fixed point, let $X = \{x \in D \mid f(x) \sqsubseteq x\}$. Now take $d = \bigsqcap X$. Now, for each $x \in X$, we have $d \sqsubseteq x$ and $f(d) \sqsubseteq f(x) \sqsubseteq x$. By taking the infimum, we get $f(d) \sqsubseteq \bigsqcap f(X) \sqsubseteq \bigsqcap A = d$. As f is monotonic, $d \sqsubseteq f(d)$, so $d = f(d)$.

For the greatest fixed point, let $x = \bigsqcup\{x \in D \mid x \sqsubseteq f(x)\}$. If $f(x) \not\sqsubseteq x$, $f(x)$ is strictly greater than x, but $f(x) \sqsubseteq f(f(x))$ so $f(x) \in \{x \in D \mid x \sqsubseteq f(x)\}$. ■

Proposition 3.1.23. *A pointed poset D is a dcpo if and only if every monotone map on D has a fixpoint.*

Follows from the proof structure used in Proposition 3.1.17 and 3.1.22. ■

From here onwards, unless explicitly stated otherwise, we shorthand $D = (D, \sqsubseteq), D' = (D', \sqsubseteq)$, which will range over cpos and interpreted in the Scott topology.

Proposition 3.1.24. *Given D, D', define $D \times D'$ to be the cartesian product partially ordered by*

$$\langle x, x' \rangle \sqsubseteq \langle y, y' \rangle \quad \text{iff} \quad x \sqsubseteq y \text{ and } x' \sqsubseteq' y'$$

Then, $D \times D'$ is a cpo with for any directed $X \subseteq D \times D'$,

$$\bigsqcup\nolimits^{\uparrow} X = \langle \bigsqcup\nolimits^{\uparrow} X_0, \bigsqcup\nolimits^{\uparrow} X_1 \rangle$$

where

$$X_0 = \{x \in D : \exists x' \in D', \langle x, x' \rangle \in X\}$$
$$X_1 = \{x' \in D' : \exists x \in D, \langle x, x' \rangle \in X\}$$

Proof. First note that if $X \subseteq D \times D'$ is directed, so are X_0 and X_1. It can then be seen that these both must have defined suprema, thus $\bigsqcup^{\uparrow} X$ is defined. To show this is actually the suprema follows trivially from construction. ■

Definition 3.1.25. *A* ***monotone net*** *in a poset D is a monotone function f from a directed set I into D. The set I is called the* ***index set*** *of the net.*

Let $f : I \to D$ be a monotone net. If we are given a monotone function $f' : J \to I$ where J is directed and for all $i \in I$ there is $j \in J$ with $i \sqsubseteq f'(j)$, we say that $f \circ f'$ to be a subnet of f.

We say that a monotone net $f : I \to D$ has a supremum in D if the set $\{f(i) \mid i \in I\}$ has a supremum in D.

$$\begin{tikzcd}
i_1 \arrow[r] \arrow[rr, bend left] \arrow[d, "f"] & i_2 \arrow[r] \arrow[d, "f"] & i_3 \arrow[d, "f"] \\
x_1 \arrow[r, dashed] \arrow[rr, dashed, bend right] & x_2 \arrow[r, dashed] & x_3
\end{tikzcd}$$

Of course, every directed set as an embedding can be seen as a monotone net. On the other hand, the image of a monotone net is a directed set in D.

Lemma 3.1.26. *Let D be a poset and $f : I \to D$ be a monotone net. Then f has a subnet $f \circ f' : J \to D$ whose index J is a lattice in which every principal ideal is finite.*

Proof. Let J be the set of finite subsets of I. Note that J is a lattice in which every principal ideal is finite (bounded by the empty set). Define a map $f' : J \to I$ by induction by the cardinality of elements of J, by

$$\begin{aligned} f'(\emptyset) &= \text{any element of } I \\ f'(A) &= \text{any upper bound of } A \cup \{f'(B) \mid B \subsetneq A\} \end{aligned}$$

Then, f' is monotone by construction. It is a subnet as $A \sqsubseteq f'(A)$. ■

Proposition 3.1.27. *Given a directed I and $f : I \times I \to D$ to be a monotone net, if any of the following suprema exists then then the following equality holds:*

$$\bigsqcup\nolimits^{\uparrow}_{i,j \in I} f(i,j) = \bigsqcup\nolimits^{\uparrow}_{i \in I}(\bigsqcup\nolimits^{\uparrow}_{j \in J} f(i,j)) = \bigsqcup\nolimits^{\uparrow}_{j \in J}(\bigsqcup\nolimits^{\uparrow}_{i \in I} f(i,j)) = \bigsqcup\nolimits^{\uparrow}_{i \in I} f(i,i)$$

Proof. TODO!!!!

Remark 3.1.28. General directed sets may not have enough structure to allow for swapping of $\bigsqcup^{\uparrow}$, as it might not be directed after swaps. Proposition 3.1.27 is making the claim that we are allowed to make these swaps under above conditions, and Proposition 3.1.7 says that in the case we are allowed to swap, they are equal. Thus, when we are sure the suprema exists, it is safe to swap $\bigsqcup^{\uparrow}$.

Definition 3.1.29. *Let D be a partially ordered set over $\sqsubseteq$. We say that a function $f : D \to D'$ is* ***continuous*** *(or Scott continuous) if it preserves least upper bounds for directed sets. That is, for every directed set S,*

$$f(\bigsqcup\nolimits^{\uparrow} S) = \bigsqcup\nolimits^{\uparrow} f(S)$$

A function between pointed dcpos which preserves the bottom element is called ***strict****.*

Proposition 3.1.30. *Scott continuous functions are monotonic.*

Proof. Let $f : D \to D'$ over cpos D, D'. Suppose we take $d, d' \in D$ such that $d \sqsubseteq d'$. Then, by continuity,

$$f(d') = f(d \sqcup d') = f(d) \sqcup f(d')$$

It therefore follows from the definition of directed suprema that $f(d) \sqsubseteq f(d')$. ■

3.2 Family of Continuous Functions

Definition 3.2.1. *Fix cpos D and D'. Define*

$$[D \to D'] := \{f : D \to D' \mid f \text{ continuous}\}$$

We equip this with a partial order:

$$f \sqsubseteq g \iff \forall x \in D, f(x) \sqsubseteq' g(x)$$

Note that this gives a well-defined poset. We further write

$$[D \overset{\perp!}{\to} D']$$

to define the set of all continuous strict functions.

Lemma 3.2.2. *Take $\{f_i\}_{i\in I} \subseteq [D \to D']$ be a directed family of maps. Define*

$$f(x) = \bigsqcup_{i\in I}^{\uparrow} f_i(x)$$

Then, f is well-defined and continuous.

Proof. Since $\{f_i\}_{i\in I}$ is directed, $\{f_i(x)\}_{i\in I}$ is directed for any x, meaning f exists. Furthermore, given any directed $X \subseteq D$,

$$f(\bigsqcup^{\uparrow} X) = \bigsqcup_{i\in I}^{\uparrow}\bigsqcup_{x\in X}^{\uparrow} f_i(x) = \bigsqcup_{x\in X}^{\uparrow}\bigsqcup_{i\in I}^{\uparrow} f_i(x) = \bigsqcup^{\uparrow} f(X)$$

where the second equality is well defined as $f_i(x)$ is directed. ■

That is, $[D \to D']$ is a dcpo if D and D' are dcpos.

3.2.1 Into Lambda Calculus

Notation 3.2.3. Given a function f, we write $\lambda x.f\ x$ to refer to the function who maps $x \mapsto f\ x$.

Proposition 3.2.4. *$[D \to D']$ is a ccpo with the supremum of a directed $F \subseteq [D \to D']$ given by*

$$(\bigsqcup^{\uparrow} F)(x) = \bigsqcup^{\uparrow}\{f(x) \mid f \in F\}$$

Proof. First note that $\lambda x.\perp'$ is the bottom element of $[D \to D']$. By Lemma 3.2.2, the map $\lambda x.\bigsqcup^{\uparrow}\{f(x) : f \in F\}$ is continuous. This is clearly the supremum of F, and so the proof follows. ■

That is, we are allowed to replace the order of supremum and evaluation of a function.

Lemma 3.2.5. *Let $f : D \times D' \to D''$. Then, f is continuous if and only if it is continuous in its arguments separately. Specifically, both $\lambda x.f(x, x'_0)$ and $\lambda x'.f(x_0, x')$ are continuous for any x_0, x'_0.*

Proof. (⇒) Let $g = \lambda x.f(x, x'_0)$. Then, for any directed $X \subseteq D$, we have

$$\begin{aligned} g(\bigsqcup^{\uparrow} X) &= f(\bigsqcup^{\uparrow} X, x'_0) \\ &= f(\bigsqcup^{\uparrow} \{(x, x'_0) \mid x \in X\}) \\ &= \bigsqcup^{\uparrow} \{f(x, x'_0) \mid x \in X\} \\ &= \bigsqcup^{\uparrow} g(X) \end{aligned}$$

Thus g is continuous, and we can do the same proof for $\lambda x'.f(x_0, x')$.

(⇐) Let $X \subseteq D \times D'$ be directed. Then,

$$\begin{aligned} f(\bigsqcup^{\uparrow} X) &= f(\bigsqcup^{\uparrow} X_0, \bigsqcup^{\uparrow} X_1) \\ &= \bigsqcup_{x \in X_0}^{\uparrow} f(x, \bigsqcup^{\uparrow} X_1) \\ &= \bigsqcup_{x \in X_0}^{\uparrow} \bigsqcup_{x' \in X_1}^{\uparrow} f(x, x') \\ &= \bigsqcup_{\langle x, x' \rangle \in X}^{\uparrow} f(x, x') \qquad \text{as } X \text{ is directed} \\ &= \bigsqcup^{\uparrow} f(X) \end{aligned}$$

Therefore f is continuous. ■

Proposition 3.2.6 (Continuity of Application)**.** *Define the application,*

$$\text{Ap} : [D \to D'] \times D \to D'$$

by $f \mapsto x \mapsto f(x)$. Then, Ap *is continuous with respect to the Scott topology on $[D \to D'] \times D$.*

Proof. Note first that $\lambda x.f(x) = f$ is continuous. Let $h = \lambda f.f(x)$. Taking any directed $F \subseteq [D \to D']$,

$$\begin{aligned} h(\bigsqcup^{\uparrow} F) &= (\bigsqcup^{\uparrow} F)(x) \\ &= \bigsqcup^{\uparrow} \{f(x) \mid f \in F\} \qquad \text{by Proposition 3.2.4} \\ &= \bigsqcup^{\uparrow} \{h(f) \mid f \in F\} \\ &= \bigsqcup^{\uparrow} h(F) \end{aligned}$$

Therefore, h is continuous. We finish by applying Lemma 3.2.5. ■

Proposition 3.2.7 (Continuity of Abstraction)**.** *Let $f \in [D \times D' \to D'']$. Define $\hat{f}(x) = \lambda y.f(x, y)$. Then,*

- *$\hat{f}$ is continuous. That is, $\hat{f} \in [D \to [D' \to D'']]$*
- *$\lambda f.\hat{f} : [D \times D' \to D''] \to [D \to [D' \to D'']]$ is continuous.*

Proof. (i) Let $X \subseteq D$ be directed. Then,

$$\begin{aligned}
\hat{f}(\bigsqcup^{\uparrow} X) &= \lambda y.f(\bigsqcup^{\uparrow} X, y) \\
&= \lambda y.\bigsqcup_{x\in X}^{\uparrow} f(x,y) \\
&= \bigsqcup_{x\in X}^{\uparrow} (\lambda y.f(x,y)) \qquad \text{by proposition 3.2.4} \\
&= \bigsqcup^{\uparrow} \hat{f}(x,y)
\end{aligned}$$

(ii) Let $L = \lambda f.\hat{f}$. Then for $F \subseteq [D \times D' \to D'']$ directed,

$$\begin{aligned}
L(\bigsqcup^{\uparrow} F) &= \lambda x.\lambda y.(\bigsqcup^{\uparrow} F)(x,y) \\
&= \lambda x.\lambda y.\bigsqcup_{f\in F}^{\uparrow} f(x,y) \\
&= \bigsqcup_{f\in F}^{\uparrow} \lambda x.\lambda y.f(x,y) \\
&= \bigsqcup^{\uparrow} L(F)
\end{aligned}$$

■

That is, functions are continuous after currying, and the function itself that curries another function is also continuous.

Definition 3.2.8. *We define* **DCPO** *to be the category of cpo's with continuous maps.*

Theorem 3.2.9. DCPO *is a cartesian closed category.*

Proof. TODO!!!

3.2.2 Least fixed point in $[D \to D]$

Theorem 3.2.10 (Kleene's Fixed Point Theorem for pointed cpos)**.** *Let D be a pointed cpo, and $f : D \to D$ be a continuous function. Define*

$$\mathrm{lfp}(f) = \bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^n(\bot)$$

Then $\mathrm{lfp}(f)$ *is the least fixed point of f.*

Proof. We first show that $\mathrm{lfp}(f)$ is a fixed point of f. Noting that f is continuous, we have

$$\begin{aligned}
f(\mathrm{lfp}(f)) &= f(\bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^n(\bot)) \\
&= \bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^{n+1}(\bot) \\
&= \bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^{n+1}(\bot) \sqcup \{\bot\} \\
&= \bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^n(\bot) \\
&= \mathrm{lfp}(f)
\end{aligned}$$

This shows $\mathrm{lfp}(f)$ is a fixed point.

Let d be any prefixed point. Noting that $\bot \sqsubseteq d$, as scott-continuous functions are monotone, we have $f(\bot) \sqsubseteq f(d)$. As d is a prefixed point, $f(\bot) \sqsubseteq d$, and inductively $f^n(\bot) \sqsubseteq d$. This gives

$$\mathrm{lfp}(f) = \bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^n(\bot) \sqsubseteq d$$

As all fixed points are prefixed points, this shows $\mathrm{lfp}(f)$ is the least fixed point of f. ■

Lemma 3.2.11. *The function* $\mathrm{lfp} : [D \to D] \to D$ *is continuous.*

Proof. By Lemma 3.2.2, it is sufficient to show that every $\mathrm{it}_n : [D \to D] \to D$ defined by $f \mapsto f^n(\bot)$ is continuous. We proceed by induction on n. Noting that the base case is trivial, we proceed by taking a directed family F of continuous functions on D:

$$\begin{aligned}
\mathrm{it}_{n+1}(\bigsqcup\nolimits^{\uparrow} F) &= (\bigsqcup\nolimits^{\uparrow} F)\mathrm{it}_n(\bigsqcup\nolimits^{\uparrow} F) \\
&= (\bigsqcup\nolimits^{\uparrow} F)(\bigsqcup\nolimits^{\uparrow}_{f\in F}\mathrm{it}_n(f)) && \text{by inductive hypothesis} \\
&= \bigsqcup\nolimits^{\uparrow}_{g\in F} g(\bigsqcup\nolimits^{\uparrow}_{f\in F}\mathrm{it}_n(f)) && \text{by Lemma 3.2.2} \\
&= \bigsqcup\nolimits^{\uparrow}_{g\in F}\bigsqcup\nolimits^{\uparrow}_{f\in F} g(\mathrm{it}_n(f)) && \text{by continuity on } g \\
&= \bigsqcup\nolimits^{\uparrow}_{f\in F}(\mathrm{it}_{n+1}(f))
\end{aligned}$$

Note in the final line that we are allowed to swap the order of suprema by Proposition 3.1.27, as the map $(-)(\mathrm{it}_n(-))$ is a monotone net. ■

We can now define fixpoint induction. First, call a dcpo **admissible** if it contains $\bot$ and is closed under suprema of ω-chains. We have a nice property about admissible predicates.

Lemma 3.2.12. *Let D be a dcpo. Let $P \subseteq D$ is an admissible predicate and $f : [D \to D]$. If x satisfies P implies that $f(x)$ satisfies P, then* $\mathrm{lfp}(f)$ *satisfies P.*

Proof. Follows easily from utilizing the fact that P is admissible. ■

Lemma 3.2.13. *Let D and D' be ccpos, and let*

$$\begin{tikzcd}
D \arrow[r, "h"] \arrow[d, "f"'] & D' \arrow[d, "g"] \\
D \arrow[r, "h"'] & D'
\end{tikzcd}$$

be a commutative diagram of continuous functions where h is strict. Then, $\mathrm{lfp}(g) = h(\mathrm{lfp}(f))$.

Proof. By the continuity of h, we have

$$\begin{aligned}
h(\mathrm{lfp}(f)) &= h(\bigsqcup\nolimits^{\uparrow}_{n\in\mathbb{N}} f^n(\bot)) \\
&= \bigsqcup\nolimits^{\uparrow}_{n\in\mathbb{N}} h \circ f^n(\bot) && \text{by continuity of } h \\
&= \bigsqcup\nolimits^{\uparrow}_{n\in\mathbb{N}} g^n \circ h(\bot) && \text{by commutativity of diagram} \\
&= \mathrm{lfp}(g) && \text{by strictness}
\end{aligned}$$

■

3.3 Approximations

We want to somehow move this notion of dcpos and move them into the space of computation. We start first by giving a notion of approximation.

Definition 3.3.1. *We say that x **approximates** y if for all directed $X \subseteq D$, $y \sqsubseteq \bigsqcup^{\uparrow} X$ implies $x \sqsubseteq x_0$ for some $x_0 \in X$. We write $x \ll y$. We may read this as x is way-below y.*

This notion of approximation can be thought of as an 'order of definite refinement', or 'order of approximation'.

3.3.1 Basis, Compactness, and Algebraicity

We can connect this notion back into topology.

Definition 3.3.2. *We say that $x \in D$ is **compact** if for every directed $X \subseteq D$, we have*

$$x \sqsubseteq \bigsqcup^{\uparrow} X \qquad \implies \qquad x \sqsubseteq x_0 \quad \textit{for some } x_0 \in X$$

Alternatively, x is compact if $x \ll x$.

Notice that the order of approximation is not necessarily reflexive unless compact. We may refer to compact as being 'finite' or 'isolated'. The intuition behind these phrases will be seen later.

Notation 3.3.3. We will use the following notation for $x, y \in D$ and $A \subseteq D$.

$$\begin{aligned} \twoheaddownarrow x &= \{y \in D \mid y \ll x\} \\ \twoheaduparrow x &= \{y \in D \mid x \ll y\} \\ \twoheaduparrow A &= \bigcup_{a \in A} \twoheaduparrow a \\ \mathrm{K}(D) &= \{x \in D \mid x \text{ compact}\} \end{aligned}$$

Proposition 3.3.4. *Let D be a dcpo. Then for any $x, x', y, y' \in D$, we have*

- *$x \ll y$ implies $x \sqsubseteq y$*
- *$x' \sqsubseteq x \ll y \sqsubseteq y'$ implies $x' \ll y'$*

Proof. Note the first case is obvious, as the set $\{y\}$ is directed. The second also follows, as given any directed $X \subseteq D$, $y \sqsubseteq y' \sqsubseteq \bigsqcup^{\uparrow} X$ and $x' \sqsubseteq x \sqsubseteq x_0$ for some $x_0 \in X$. ■

Definition 3.3.5. *We say that a subset B of a dcpo D is a **basis** for D if every element x of D, the set $B_x = \{\twoheaddownarrow x \cap B\}$ contains a directed subset with supremum x. We call elements of B_x **approximants to** x **relative to** B.*

This means that the elements in the basis which are way-below an element $x \in D$ are directed and has supremum x.

For instance, we can think of this notion in $\mathbb{R}$ (with a top element added) as some 'dense from below' viewing the reals as Dedekind cuts. Then, $\mathbb{Q}, \mathbb{R} \backslash \mathbb{Q}$, dyadic numbers (rationals with denominator powers of 2) are all basis for $\mathbb{R}$.

Proposition 3.3.6. *Let D be a dcpo with basis B. Then,*

1. *For every $x \in D$, B_x is directed, and $x = \bigsqcup^{\uparrow} B_x$*

2. $\mathrm{K}(D) \subseteq B$

3. *Every superset of B is also a basis for D*

Proof. (1) Equality follows from definition given directedness. We wish to show directedness. By definition, B_x contains a directed A with $\bigsqcup^{\uparrow} A = x$. Now take any $y, y' \in B_x$. These must be approximating x. As $x \sqsubseteq \bigsqcup^{\uparrow} A$, there exists $a, a' \in A$ such that $y \sqsubseteq a$ and $y' \sqsubseteq a'$. As A is directed, this has an upper bound in A, which is a subset of B_x. Thus B_x is directed.

(2) Taking $x \in \mathrm{K}(D)$, using the fact that $x = \bigsqcup^{\uparrow} B_x$, as x is compact, there exists an $x_0 \in B_x$ such that $x \sqsubseteq x_0$. Thus, $x = x_0$ and so $x \in B$.

(3) Follows from definition. ■

Definition 3.3.7. *A dcpo is called **continuous** if it has a basis. We say that it is ω-continuous if there exists a countable basis.*

Proposition 3.3.8. *If x is a non-compact element of a basis B for a continuous D, then $B \backslash \{x\}$ is still a basis.*

Proof. TODO!!!!(Hint, interpolation property)

Corollary 3.3.9. *The largest basis for D is D itself. On the contrary, B is the smallest basis of D if and only if $B = \mathrm{K}(D)$.*

Proof. The first statement and the if condition for the second statement follow from Proposition 3.3.6. The only if condition follows from Proposition 3.3.8, as we can remove non-compact elements and still be a basis. ■

Remark 3.3.10. Note that this does not imply that if D is continuous then $\mathrm{K}(D)$ is a basis. However, we are allowed to remove any finite number of non-compact elements from a basis. The next part covers when $\mathrm{K}(D)$ is indeed a basis.

Definition 3.3.11. *A dcpo is called **algebraic** or is an **algebraic domain** if it has a basis of compact elements. We say that it is ω-algebraic if $\mathrm{K}(D)$ is a countable basis.*

Remark 3.3.12. We use the notion of domain to refer to a space in which we can talk about some notion of convergence and approximation.

Proposition 3.3.13. *We can combine our notion with Proposition 3.3.6 to redefine the notion of continuity and algebraicity. That is,*

1. *A dcpo D is continuous if and only if for all $x \in D$, $x = \bigsqcup^{\uparrow} \mathord{\downarrow\!\!\!\downarrow} x$.*

2. *A dcpo D is algebraic if and only if for all $x \in D$, $x = \bigsqcup^{\uparrow} \mathrm{K}(D)_x$*

Proof. (1) ($\Rightarrow$) Suppose that D has a basis. Then D is a basis for D. Specifically, for any $x \in D$, $D_x = \mathord{\downarrow\!\!\!\downarrow} x \cap D = \mathord{\downarrow\!\!\!\downarrow} x$. By Proposition 3.3.6, $x = \bigsqcup^{\uparrow} D_x = \bigsqcup^{\uparrow} \mathord{\downarrow\!\!\!\downarrow} x$.

($\Leftarrow$) Suppose that $x = \bigsqcup^{\uparrow} \mathord{\downarrow\!\!\!\downarrow} x = \bigsqcup^{\uparrow} D_x$ for any x. Then, clearly D is a basis as each D_x is a directed subset with supremum x.

(2) ($\Rightarrow$) Suppose that D has a basis of compact elements. By Proposition 3.3.6, any basis is larger than $\mathrm{K}(D)$, so $\mathrm{K}(D)$ is a basis for D. Now, by the same proposition, it follows that for any x, $x = \bigsqcup^{\uparrow} \mathrm{K}(D)_x$.

($\Leftarrow$) Suppose that for any x, $x = \bigsqcup^{\uparrow} \mathrm{K}(D)_x$. Then, as $\{x\} \subseteq \mathrm{K}(D)_x$ is trivially directed with supremum x. ■

3.3.2 Closure Systems and Algebraicity

There is a reason why we use the word "algebraic" when $\mathrm{K}(D)$ is a basis for D. We cover this here.

Definition 3.3.14. *Let X be a set and $\mathcal{L}$ be a family of subsets of X. We say that*

- *$\mathcal{L}$ is **closure system** if it closed under the formation of intersections. That is, if $A_i \in \mathcal{L}$ for $i \in I$, $\bigcap_{i\in I} A_i \in \mathcal{L}$. Note that as I can be empty, $X \in \mathcal{L}$.*
- *We call the members of $\mathcal{L}$ **hulls** or **closed sets**.*
- *Given an arbitrary $A \subseteq X$, the least superset of A contained in $\mathcal{L}$, or explicitly, $\bigcap\{B \in \mathcal{L} \mid A \subseteq B\}$ is called the **hull** or **closure** of A.*

Proposition 3.3.15. *Every closure system is a complete lattice with respect to inclusion.*

Proof. Infima is given trivially by intersection, while the supremum is given by the closure of the union. ■

Definition 3.3.16. *A closure system $\mathcal{L}$ is called **inductive** if it is closed under directed union (directed sets are closed under union).*

Proposition 3.3.17. *Every inductive closure system $\mathcal{L}$ is an algebraic lattice. The compact elements are precisely the finitely generated closed sets (closure of finite sets).*

Proof. First, if A is the closure of a finite set M and $(B_i)_{i\in I}$ is a directed family of closed sets such that $\bigsqcup^{\uparrow}_{i\in I} B_i = \bigcup_{i\in I} B_i \supseteq A$, then M is contained in some B_i as M is finite and $(B_i)_{i\in I}$ is directed. Thus closures of finite sets are compact in $\mathcal{L}$.

On the other hand, every closed set is the directed union of finitely generated closed sets. That is, given a closed set A, the set $\{B \subseteq A \mid B \text{ is finitely generated}\}$ is a directed set whose union is equal to A. As finitely generated closed sets are compact, it is way-below anything greater than x. Thus, the set of finitely generated elements form a basis for $\mathcal{L}$.

By Proposition 3.3.6, as every basis contains the set of compact elements, it follows that compact elements are precicely the finitely generated closed sets, which forms a basis. ■

Example 3.3.18. We denote some examples of algebraic domains.

- Given a group, there are two canonical inductive closure systems associated to it, the lattice of subgroups and the lattice of normal subgroups.
- Any set with discrete order is an algebraic domain. In semantics, we may add a bottom to obtain a flat domain. In both cases, a basis will contain all elements.
- Every finite poset.

Example 3.3.19. We further illustrate some examples of continuous domains.

- Every algebraic approximation is characterized by $x \ll y$ if and only if there exists a compact element between x and y. This follows from the fact that every element is a supremum of a set of compact elements.
- The unit interval is a continuous lattice.

Example 3.3.20. On the contrary, we can construct non-continuous dcpos. We illustrate this with the following example:

$$\begin{tikzcd}
 & & \top & & \\
 & a_1 \arrow[ur, dashed] & & b_1 \arrow[ul, dashed] & \\
a_0 \arrow[ur] & & & & b_0 \arrow[ul]
\end{tikzcd}$$

This dcpo has no order of approximation. Pairs (a_i, b_j) or (b_i, a_j) are not related to one another, and two points $a_i \sqsubseteq a_j$ (for a_j possibly $\top$) is not approximating as $(b_n)_{n\in\mathbb{N}}$ is a directed set with supremum above a_j but has no element above a_i.

On the other hand, if we have a pointed poset, then it has non-empty approximation as the bottom element approximates every other element.

Basis elements have many nice properties, as it not only gives approximations for elements but also for the order relation.

Proposition 3.3.21. *Let D be a continuous domain with basis B and let $x, y \in D$. Then the following are equivalent.*

1. $x \sqsubseteq y$
2. $B_x \subseteq B_y$
3. $B_x \subseteq {\downarrow} y$

Proof. (1) $\implies$ (2) Follows immediately from the fact that anything approximating x is also approximating y.

(2) $\implies$ (3) Follows from the fact that $B_y \subseteq {\downarrow} y$.

(3) $\implies$ (1) By taking the suprema on both sides, we get $x \sqsubseteq y$. ■

Remark 3.3.22. By the above proposition, if D is continuous, given $x \not\sqsubseteq y$, there exists $b \in B_x$ with $b \not\sqsubseteq y$ (by using (1) and (3)).

Furthermore, we can also see that for a continuous domain, the information about how the elements are related are already contained in the basis. The fact that $\bigsqcup^{\uparrow}_{n\in\mathbb{N}} a_n = \bigsqcup^{\uparrow}_{n\in\mathbb{N}} b_n = \top$ is precisely what is not visible in a 'basis', if it were to exist. Thus, by separating $\top$ to be distinct for (a_i) and (b_i), it gives us an algebraic domain.

Proposition 3.3.23. *Let D and D' be continuous domains with bases B and B'. Then, $f : D \to D'$ is continuous if and only if for each $x \in D$ and $y \in B'_{f(x)}$, there exists $z \in B_x$ with $f(\uparrow z) \subseteq \uparrow y$.*

Proof. ($\Rightarrow$) Suppose f is continuous. Fixing an x and y, we have

$$f(x) = f(\bigsqcup^{\uparrow} B_x) = \bigsqcup^{\uparrow}_{d\in B_x} f(d)$$

and as y approximates $f(x)$, there exists a $z \in B_x$ such that $y \sqsubseteq f(z)$. By monotonicity, $f(\uparrow z) \subseteq \uparrow y$.

($\Leftarrow$) We first show that f is monotonic. Suppose $x \sqsubseteq x'$ but $f(x) \not\sqsubseteq f(x')$. By Proposition 3.3.21, there exists a $y \in B'_{f(x)}$ with $y \not\sqsubseteq f(x')$. By assumption, there exists a $z \in B_x$ such that $f(\uparrow z) \subseteq \uparrow y$. As $x \in \uparrow z$, so is x', but now $f(x') \in \uparrow y$, which is a contradiction.

Now, let X be a directed subset of D with $x = \bigsqcup^{\uparrow} X$. By monotonicity,

$$\bigsqcup^{\uparrow} f(X) \sqsubseteq f(\bigsqcup^{\uparrow} X) = f(x)$$

If $f(x) \not\sqsubseteq \bigsqcup^{\uparrow} f(X)$, we can again find a $y \in B'_{f(x)}$ with $y \not\sqsubseteq \bigsqcup^{\uparrow} f(X)$. By assumption, there exists a $z \in B_x$ such that $f(\uparrow z) \subseteq \uparrow y$. As z approximates x, some $x' \in X$ is above z, giving

$$y \sqsubseteq f(z) \sqsubseteq f(x') \sqsubseteq \bigsqcup^{\uparrow} f(X)$$

contradicting our choice of y. ■

Proposition 3.3.24. *let D be a pointed ω-continuous domain with basis B. If $f : [D \to D]$, there exist an ω-chain $b_0 \sqsubseteq b_1 \sqsubseteq \dots$ of basis elements such that the following hold:*

1. $b_0 = \bot$
2. $\forall n \in \mathbb{N}$, $b_{n+1} \sqsubseteq f(b_n)$
3. $\bigsqcup^{\uparrow}_{n\in\mathbb{N}} b_n = \mathrm{lfp}(f)$

Proof. Out of scope, but proof may be found in [1]. The point is that as continuous functions don't preserve compactness nor order of approximation, $f^n(\bot)$ need not consist of basis elements. Nonetheless, we can construct a chain of elements which are prefixpoints of the previous item such that the suprema is the fixpoint. ■

Proposition 3.3.25. *Let D be algebraic and $f : D \to D$. Then f is continuous if and only if for any x, $f(x) = \bigsqcup^{\uparrow}\{f(e) \mid e \sqsubseteq x, e$ compact$\}$.*

Proof. ($\Rightarrow$) Let f be continuous. Then,

$$\begin{aligned} f(x) &= f(\bigsqcup^{\uparrow}\{e \sqsubseteq x \mid e \text{ compact}\}) \\ &= \bigsqcup^{\uparrow}\{f(e) \mid e \sqsubseteq x, e \text{ compact}\} \end{aligned}$$

($\Leftarrow$) First note that f is monotinic. That is, given $x \sqsubset y$,

$$\{e \sqsubseteq x \mid e \text{ compact}\} \subseteq \{e \sqsubseteq y \mid e \text{ compact}\}$$

Then,

$$\begin{aligned} f(x) &= \bigsqcup^{\uparrow}\{f(e) \mid e \sqsubseteq x, e \text{ compact}\} \\ &\sqsubseteq \bigsqcup^{\uparrow}\{f(e) \mid e \sqsubseteq y, e \text{ compact}\} = f(y) \end{aligned}$$

Suppose now that $X \subseteq D$ is directed. Then,

$$\begin{aligned} f(\bigsqcup^{\uparrow} X) &= \bigsqcup^{\uparrow}\{f(e) \mid e \sqsubseteq \bigsqcup^{\uparrow} X, e \text{ compact}\} \\ &\sqsubseteq \bigsqcup^{\uparrow}\{f(x) \mid x \in X\} \quad \text{by compactness} \\ &\sqsubseteq f(\bigsqcup^{\uparrow} X) \quad \text{by monotonicity} \end{aligned}$$

Henceforth $f(\bigsqcup^{\uparrow} X) = \bigsqcup^{\uparrow} f(X)$. ■

Proposition 3.3.26. *Let D be algebraic. Define $O_e = \{x \in D \mid e \sqsubseteq x\} = \uparrow \{e\}$. Then,*

$$\{O_e \mid e \textit{ compact}\}$$

is a basis for the topology on D.

Proof. We first show that O_e is open when e is compact. Note that O_e is an upper set. Taking any directed $X \subseteq D$ such that $\bigsqcup^{\uparrow} X \in O_e$, as $e \sqsubseteq \bigsqcup^{\uparrow} X$, by compactness, we have $e \sqsubseteq x_0$ for some $x_0 \in X$. Specifically, the intersection of X and O_e is nontrivial, hence O_e is Scott open.

Now, take any $x \in O$ where O is open. As $x = \bigsqcup^{\uparrow}\{e \sqsubseteq x \mid e \text{ compact}\}$ is directed suprema,

$$\exists e \sqsubseteq x \qquad e \in O \qquad e \text{ compact}$$

using the fact O is open. Thus, $x \in O_e \in O$ (noting that O_e is the smallest open set that contains e). Therefore, each open O is the union of opens in the basis set. ■

3.4 Topological Interpretation of Domains

Definition 3.4.1. *Let $(D, \sqsubseteq)$ be a partially ordered set. A subset $O \subseteq D$ is called **Scott-open** if is an upper set that is inaccessible by directed suprema. That is, all directed sets S with a supremum in O have a non-empty intersection with O.*

Proposition 3.4.2. *The Scott-open subsets of a partially ordered set $(D, \sqsubseteq)$ form a topology on D, which is called the **Scott topology** and written as (D, τ).*

Proof. Start by noting that $\emptyset$ and D are both trivially Scott open.

Consider a family of Scott open sets $\mathcal{U} = \{U_i\}_{i \in I}$. Take any $x \in \uparrow (\bigcup \mathcal{U})$. Then, there exists an $i \in I$ such that $y \in U_i$ and $y \sqsubseteq x$. Now,

$$x \in \uparrow U_i = U_i \subseteq \bigcup \mathcal{U}$$

Henceforth $\uparrow (\bigcup \mathcal{U}) \subseteq \bigcup \mathcal{U}$ and equality follows. Now, let $X \subseteq D$ be a directed subset such that $\bigsqcup^{\uparrow} X \in \bigcup \mathcal{U}$. Then, there exists a $i \in I$ such that $\bigsqcup^{\uparrow} X \in U_i$. Then as U_i is Scott open, $X \cap U_i \neq \emptyset$. Using

$$X \cap U_i \subseteq \bigcup_{i \in I}(X \cap U_i) = X \cap \bigcup_{i \in I} U_i = X \cap (\bigcup \mathcal{U})$$

We see that $X \cap (\bigcup \mathcal{U}) \neq \emptyset$. Thus $\bigcup \mathcal{U}$ is Scott open.

Take any $U, V \in \tau$ and let $x \in \uparrow (U \cap V)$. Then, there exists a $y \in U \cap V$ with $y \sqsubseteq x$. Now, as U and V are both upper sets, we have

$$x \in (\uparrow U) \cap (\uparrow V) = U \cap V$$

This gives $\uparrow (U \cap V) \subseteq U \cap V$. Now, let $X \subseteq D$ be a directed subset such that $\bigsqcup^{\uparrow} X \in U \cap V$. As U and V are Scott open, there exists a $u \in X \cap U$ and $v \in X \cap V$. As X is directed, there exists a $x \in X$ such that $u \sqsubseteq x$ and $v \sqsubseteq x$. Now,

$$x \in X \cap ((\uparrow U) \cap (\uparrow V)) = X \cap (U \cap V)$$

Hence, $D \cap (U \cap V) \neq \emptyset$. Thus $U \cap V$ is Scott open. ■

Proposition 3.4.3. *A subset of a poset D is closed in the Scott topology if and only if it is a lower set and is closed under the suprema of directed subsets.*

Proof. ($\Rightarrow$) Let U be Scott open. That is, it it is an upper set and every directed set with suprema in U has non-empty intersection with U. Now, take any $y \in D \backslash U$ and let $x \sqsubseteq y$. If $x \in U$, as U is open, $y \in U$, a contradiction. Thus, $D \backslash U$ is a lower set. Take any directed subset $X \subseteq D \backslash U$ with a suprema. Then, if the suprema lies in U, X has nonempty intersection with U, a contradiction.

($\Leftarrow$) Let U be a lower set and closed under the suprema of directed subsets. Then, takinga any $x \in D \backslash U$ with $x \sqsubseteq y$, if $y \in U$, we contradict with U being a lower set. Now, take any directed subset X with suprema. If this set is contained in U, it's suprema is contained in U. Otherwise, X has suprema in $D \backslash U$ and has non-empty intersection with $D \backslash U$. ■

Proposition 3.4.4. *Let $(D, \sqsubseteq)$ be a partially ordered set. For any $d \in D$, $D \backslash (\downarrow d)$ is Scott open. We write U_d for this set.*

Proof. Let $x \in \uparrow (D \backslash \downarrow d)$. Then there exists a $y \in D \backslash \downarrow d$ such that $y \sqsubseteq x$. Suppose for a contradiction that $x \in \downarrow d$. That is, $x \sqsubseteq d$. By transitivity of $\sqsubseteq$, $y \sqsubseteq p$. This contradicts $y \in D \backslash \downarrow d$. Therefore $x \in D \backslash \downarrow d$. Thus, $D \backslash \downarrow d$ is an upper set.

Now consider a directed set $X \subseteq D$ such that $\bigsqcup^{\uparrow} X \in D \backslash \downarrow d$. Then, $\bigsqcup^{\uparrow} X \not\sqsubseteq d$. Suppose for a contradiction that $X \cap (D \backslash \downarrow d) = \emptyset$. This gives $X \subseteq \downarrow d$. This means that d is an upper bound for X, giving $\bigsqcup^{\uparrow} X \sqsubseteq d$, which is a contradiction. Thus, $X \cap (D \backslash \downarrow d) \neq \emptyset$. This shows $D \backslash \downarrow d$ is inaccessible by directed suprema, meaning it is Scott open. ■

Corollary 3.4.5. *D is a T_0 space which is not necessarily T_1.*

Proof. Take $x, y \in D$ with $x \neq y$. Suppose now that without loss of generality, $x \not\sqsubseteq y$. Then, $x \in U_y, y \neq U_y$ and U_y is open. Thus D is T_0. On the other hand, if $x \sqsubseteq y$, then every neighborhood of x contains y as it must be an upper set. Thus, D need not be T_1. ■

3.4.1 On continuity

Scott continuity has an interpretation under continuity in the topological sense, which we will discuss in this subsection.

Lemma 3.4.6. *If f is continuous under the Scott topology, it is monotonic.*

Proof. Let $f : D \to D'$ be continuous under the Scott topology. Take $x, x' \in D$ such that $x \sqsubseteq x'$. Supose for a contradiction that $f(x) \not\sqsubseteq f(x')$. Then, $f(x) \in D' \backslash \downarrow f(x')$. Noting this set is Scott open, we have $x \in f^{-1}(D' \backslash \downarrow f(x'))$ which is also Scott open by continuity. As this set is upper closed, it follows that $x' \in f^{-1}(D' \backslash \downarrow f(x'))$. Now,

$$\begin{aligned} x' \in f^{-1}(D' \backslash \downarrow f(x')) &\implies f(x) \in D' \backslash \downarrow f(x') \\ &\implies f(x') \sqsubseteq f(x') \end{aligned}$$

which is a contradiction. Hence, it follows that $f(x) \sqsubseteq f(x')$. ■

Theorem 3.4.7. *A function between partially ordered sets $(D, \sqsubseteq)$ is Scott continuous if and only if it is continuous with respect to the Scott topology.*

Proof. ($\Rightarrow$) Suppose that $f : D \to D'$ is Scott continuous. Take any Scott open set U in E. We want to show that $f^{-1}(U)$ is Scott open. Specifically, we wish to show that U is (i) an upper set and (ii) all directed sets with a supremum in $f^{-1}(U)$ has a non-empty intersection with $f^{-1}(U)$.

- For (i), take any $x \in f^{-1}(U)$ such that $f(x) \in U$. Given $x \sqsubseteq x'$, by monotonicity of Scott continuous functions we have $f(x) \sqsubseteq f(x')$. As U is an upper set, we have $f(x') \in U$. It follows that $x' \in f^{-1}(U)$.

- For (ii), Take $X \subseteq D$ be any directed set such that $\bigsqcup^{\uparrow} X \in f^{-1}(U)$. That is, $f(\bigsqcup^{\uparrow} X) \in U$. By Scott continuity, $\bigsqcup^{\uparrow} f(X) \in U$. As U is Scott open, we have $f(X) \cap U \neq \emptyset$. Equivalently, there is a $x \in X$ such that $f(x) \in U$. Thus, $x \in f^{-1}(U)$, therefore it is inaccessible by a directed suprema.

($\Leftarrow$) Suppose that f is continuous in the Scott topology, such that for any Scott open set $U \in D'$, $f^{-1}(U)$ is Scott open in D. We wish to show that f is Scott continuous, such that for any directed set $X \subseteq D$ with a supremum,

$$f(\bigsqcup^{\uparrow} X) = \bigsqcup^{\uparrow} f(X)$$

We prove this by showing that $f(\bigsqcup^{\uparrow} X)$ is (i) an upper bound and (ii) the least upper bound with respect to $f(X)$.

- For (i), note that as f is monotone from Lemma 3.4.6, given any $x \in X$, we have $x \sqsubseteq \bigsqcup^{\uparrow} X$, meaning $f(x) \sqsubseteq f(\bigsqcup^{\uparrow} X)$. This shows $\bigsqcup^{\uparrow} f(X) \sqsubseteq f(\bigsqcup^{\uparrow} X)$.

- For (ii), Suppose for a contradiction that $f(\bigsqcup^{\uparrow} X) \not\sqsubseteq \bigsqcup^{\uparrow} f(X)$. Then, $f(\bigsqcup^{\uparrow} X) \in D' \setminus \downarrow \bigsqcup^{\uparrow} f(X)$. It follows that $\bigsqcup^{\uparrow} X \in f^{-1}(D' \setminus \downarrow \bigsqcup^{\uparrow} f(X))$. As this is Scott open, $X \cap f^{-1}(D' \setminus \downarrow \bigsqcup^{\uparrow} f(X)) \neq \emptyset$. Taking x to be an element in this, we have $f(x) \in f(X)$ and $f(x) \in D' \setminus \downarrow \bigsqcup^{\uparrow} f(X)$. The latter transforms into $f(x) \not\sqsubseteq \bigsqcup^{\uparrow} f(X)$, contradicting with $f(x) \in f(X)$. We therefore have $f(\bigsqcup^{\uparrow} X) \sqsubseteq \bigsqcup^{\uparrow} f(X)$.

■

3.4.2 Projective Limits

Definition 3.4.8. *Let $D_0, D_1, \ldots$ be a countable sequence of cpo's and let $f_i \in [D_{i+1} \to D_i]$. We now define*

- *The sequence (D_i, f_i) is called the* ***projective*** *or* ***inverse system*** *of cpos.*

- *The* ***projective*** *or* ***inverse limit*** *of the system (D_i, f_i) with notation $\varprojlim(D_i, f_i)$ is the poset $(D_\infty, \sqsubseteq_\infty)$ with*

$$D_\infty = \{\langle x_0, x_1, \ldots \rangle \mid \forall i, x_i \in D_i \wedge f_i(x_{i+1}) = x_i\}$$

where,

$$\langle \vec{x} \rangle \sqsubseteq_\infty \langle \vec{y} \rangle \quad \text{iff} \quad \forall i, x_i \sqsubseteq y_i$$

Note that as usual, we interpret an infinite sequence $\langle x_0, x_1, \ldots \rangle$ with a map $x : \mathbb{N} \to \cup_i D_i$ such that $x(i) = x_i \in D_i$ for all i. Where it is clear, we may write $\varprojlim(D_i)$ for $\varprojlim(D_i, f_i)$.

Proposition 3.4.9. *Let (D_i, f_i) be a projective system. Then $\varprojlim(D_i, f_i)$ is a cpo with*

$$\bigsqcup^{\uparrow} X = \lambda i. \bigsqcup^{\uparrow} \{x(i) \mid x \in X\}$$

for directed $X \subseteq \varprojlim(D_i)$.

Proof. If X is directed, then $\{x(i) \mid x \in X\}$ is directed for each i. Let

$$y_i = \bigsqcup\nolimits^{\uparrow} \{x(i) \mid x \in X\}$$

Then by the conitnuity of f_i, we have

$$\begin{aligned} f_i(y_{i+1}) &= \bigsqcup\nolimits^{\uparrow} f_i\{x(i+1) \mid x \in X\} \\ &= \bigsqcup\nolimits^{\uparrow} \{x(i) \mid x \in X\} \\ &= y_i \end{aligned}$$

Therefore $\langle y_0, y_1, \dots \rangle \in \varprojlim(D_i)$. This is clearly the supremum of X. ■

Definition 3.4.10. *Let D be a cpo and let $X \subseteq D$. Then,*

- *$f \in [D \to D]$ is a* ***retraction*** *map of D onto X if $X = \mathrm{im}(f)$ and $f = f \circ f$.*
- *X is a* ***retract*** *of D if there is a retraction map f of D onto X.*

Alternatively, we can think of a retraction map as a continuous function f for which the following diagram commutes:

$$\begin{tikzcd} D \arrow[r, two heads, "f"] \arrow[dr, two heads, "f"'] & X \arrow[d, two heads, "f"] \\ & X \end{tikzcd}$$

Proposition 3.4.11. *Let D be a cpo with retract X. Then X is a cpo whose directed subsets have the same supremum as in D whose topology is the subspace topology.*

Proof. Let $f : D \to X$ be a retraction map. Let $Y \subseteq X$ be directed. Then Y is also directed as a subset of D and $\bigsqcup^{\uparrow} Y = y$ exists in D. Now,

$$\begin{aligned} f(y) &= f(\bigsqcup\nolimits^{\uparrow} Y) \\ &= \bigsqcup\nolimits^{\uparrow} f(Y) \\ &= \bigsqcup\nolimits^{\uparrow} Y \qquad \text{as } Y \subseteq X \\ &= y \end{aligned}$$

Therefore $y \in X$ and is also clearly the supremum of Y in X. ■

3.5 Function Approximations

Definition 3.5.1. *Given two sets A and B, an* ***empty function*** *is a partial function $A \rightharpoonup B$ that has no defined maps from elements of A. Alternatively, if we interpret this as a function from A to $\mathsf{Option}\ B$ that takes $a \mapsto \mathsf{Nothing}$. We often write $\bot$ to represent this function.*

Proposition 3.5.2. *The set of partial functions from $\mathbb{N}$ to $\mathbb{N}$ with the standard $\sqsubseteq$ on partial functions is a ccpo.*

Proof. TODO!!

3.5.1 The Factorial Function

Example 3.5.3.

Consider the factorial function which might be recursively defined as

```
int factorial(nat n) {
    if (n == 0)
        then 1;
    else
        n * factorial(n-1);
}
```

under a programming context. We will write f to represent this function.

To give meaning to this f, we model this through an approximation as a partial function $\mathbb{N} \rightharpoonup \mathbb{N}$ starting with the empty function. We introduce a function $F : (\mathbb{N} \rightharpoonup \mathbb{N}) \to (\mathbb{N} \rightharpoonup \mathbb{N})$ defined by the map

$$f \mapsto \{0 \mapsto 1\} \ \oplus \ \{n \mapsto n * f(n-1) \mid n \in \mathbb{N}\backslash\{0\}\}$$

where $\oplus$ represents an overloading of function maps. Then we define $F^0(\bot) = \bot$ and $F^{n+1}(\bot) = F(F^n(\bot))$. This process builds a sequence of $\mathbb{N} \rightharpoonup \mathbb{N}$. Note that this sequence satisfies $F^n \sqsubseteq F^{n+1}$. As $\mathbb{N} \rightharpoonup \mathbb{N}$ is an ω-cpo, by Kleene's Fixed Point Theorem, setting

$$f := \bigsqcup_{n\in\mathbb{N}}^{\uparrow} F^n(\bot)$$

this define a suitable interpretation of the recursive function defined above. Notice that we take the least fixed point on F, as we are talking about functions that terminate.

4 Sources

References

[1] S. Abramsky. A generalized Kahn principle for abstract asynchronous networks. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, Mathematical Foundations of Programming Semantics, volume 442 of Lecture Notes in Computer Science, pages 1–21. Springer Verlag, 1990.

[2] G. Markowsky. Chain-complete p.o. sets and directed sets with applications. Algebra Universalis, 6:53–68, 1976.