Fixed points

- Denotational semantics for IMP rely on taking fixed point to define `while`
- Fixed points occur in most language definitions: needed to deal with loops
  - control flow loops: while
  - data loops: recursive functions, recursive data structures, recursive types
- Only know how to find least fixed pts for continuous functions `f`
- Need easy way to ensure continuity

Meta-language

- Idea: define restricted language for expressing mathematical functions
- All functions expressible in this language are continuous
- Looks like a programming language (ML)
  - not executed: just mathematical notation
  - can talk about non-termination!
  - “evaluation” is lazy (vs. eager in ML)

“Types” for Meta-language

- Meta-language contains domain declarations indicating the set of values meta-variables can take on, e.g.
  - `λf ∈ Σ → Σ. λσ ∈ Σ. if ¬[b] σ then σ else f([c])`
- Domains will function as types for meta-language
  - but with precisely defined meaning, ordering relation, etc.
- `T_1 * T_2` is not necessarily modeled by `T_1 × T_2`
- Meta-language consists of domains and associated operations

Lifting

- If `D` is a domain (for now: cpo), can “lift” by adding new bottom element to form pointed cpo `D_⊥`
- cpo defined by underlying set plus complete ordering relation `⊆`
- Elements of `D_⊥` are `[d]_⊥` where `d ∈ D`
- Ordering relation:
  - `<[d_1] ⊆ [d'_1]` if `d_1 ⊆ d'_1`
- Complete?
**Discrete cpos**

- Various discrete cpos: booleans (T), natural numbers (ω), integers (Z), ...
- Corresponding functions over discrete cpos exist: \( + : Z \rightarrow Z, \wedge : T \rightarrow T \)
- Often want to lift discrete cpos to take fixed points; helpful to extend fns to pointed cpos
  - If \( f : D \rightarrow E \), then \( f \in D_1 \rightarrow E_1, f' \in D_2 \rightarrow E \) are \( f_1 = \lambda d \in D_1. \text{if } d=1 \text{ then } \perp \text{ else } f(d) \)
  - \( f' = \lambda d \in D_2. \text{if } d=1 \text{ then } \perp \text{ else } f(d) \)  (if \( E \) pointed)
- \( 2 + 2 = 4, \ 3 + 1 = 1, \ \perp \wedge \perp \), \( \perp \) true = true
- If \( f \) continuous, are \( f \perp \) ?

**Unit**

- Simplest cpo: empty set (\( \emptyset \))
- Next simplest: unit domain (U) 
  - single element: \( u \)
  - ordering relation: reflexive \( \cdot \)
  - complete: only directed set is \( \{u\} \)
- Used to represent computations that terminate but do not produce a value, argument for functions that need no argument
- Also building block for other domains

**Let**

- Useful syntax: given \( d_1, D_1 \)
  
  \[
  \lambda x \in D_1. e \equiv (\lambda x \in D_1. e)^d
  \]

- Expresses evaluation of \( e \) that is strict in \( d \)
- Example: \( \text{\l[while]} \)
  
  \[
  \text{fix } f : \Sigma_1 \rightarrow \Sigma_1. \text{let } \sigma = \sigma' \text{ in if } \sigma[b] = \sigma \text{ then } \sigma \text{ else } f(\sigma[c])
  \]

**Products**

- If \( D_i, D_j \) are domains, then \( D_1 \times D_2 \) is a product domain
- Underlying set: pairs \( \langle d_1, d_2 \rangle \) where \( d_i \in D_i \)
- Ordering: \( \langle d_1, d_2 \rangle \leq \langle d'_1, d'_2 \rangle \) iff \( d_i \leq d_i', \forall i \leq m \)
  - Extends to \( n \)-tuples
  - Operations:
    - tupling: \( \langle d_1, \ldots, d_m \rangle \)
    - projection: \( \pi_i (\langle d_1, \ldots, d_m \rangle) = d_i \)

**CPO?**

- Is product domain a cpo if \( D_i, D_j \) are?
- Any chain \( \langle d_0, d'_0 \rangle \leq \langle d_1, d'_1 \rangle \leq \ldots \) must have LUB in \( D_1 \times D_2 \)
- Definition of \( \leq : d_0 \leq d_1 \leq \ldots \) is chain in \( D_1 \), \( d'_0 \leq d'_1 \leq \ldots \) is chain in \( D_2 \)
- If \( d_m \in D_1, d'_m \in D_2 \) are respective LUBs, \( \langle d_m, d'_m \rangle \in D_1 \times D_2 \) is LUB of chain of pairs
- Operations continuous?
  - \( \bigcup \mathcal{X}_n = \bigcup \mathcal{X}_n' = \bigsqcup d_m \)
  - \( \bigcup (\mathcal{X}_n, \ldots, \mathcal{X}_m) = \bigsqcup (\mathcal{X}_n, \ldots, \mathcal{X}_m) \)

**Sums**

- Sometimes want to allow values of one kind or another: \( D_1 \uplus D_2 \)
- Elements of domain are elements of \( D_k \) or \( D_m \) tagged with origin: \( \text{in}(d_k) \)
  - Form of \( \text{in} \) is irrelevant (could be \( \lambda d. (i, d) \))
  - Preserves ordering of individual domains: \( \text{in}(d_k) \leq \text{in}(d_m) \) iff \( i=j, d_i \leq d_j \)
  - Injection function \( \text{in} \) is continuous
  - Extends naturally to multi-domain sum
- CPO, but not pointed

---

**Diagram**

- Hasse diagram
- Elements of domain are \( \pi_1 \) and \( \pi_2 \)
- Operation: \( \pi_i \) (\( \langle d_1, \ldots, d_m \rangle \) = \( d_i \))

---

**Note**

- Sometimes want to lift discrete cpos to take fixed points; helpful to extend functions to pointed cpos
- If \( f : D \rightarrow E \), then \( f \in D_1 \rightarrow E_1, f' \in D_2 \rightarrow E \) are \( f_1 = \lambda d \in D_1. \text{if } d=1 \text{ then } \perp \text{ else } f(d) \)
- \( f' = \lambda d \in D_2. \text{if } d=1 \text{ then } \perp \text{ else } f(d) \)  (if \( E \) pointed)
- \( 2 + 2 = 4, \ 3 + 1 = 1, \ \perp \wedge \perp \), \( \perp \) true = true
- If \( f \) continuous, are \( f \perp \) ?
Sums, cont’d
- Why tag? Distinguishes identical domains
  - T = U + U, true = in₁(u), false = in₂(u)
- Sums unpacked with case construction:
  \( \text{case } e \text{ of } x₁ \rightarrow e₁ | x₂ \rightarrow e₂ = \text{ case } e \text{ of } D₁(x₁).e₁ | D₂(x₂).e₂ \)
- Given \( e = \text{in}_1(d₁) \), has value \( f(d₁) \in E \) where \( f \in D₁ \rightarrow E = (λx.∈D₁.e) \)
- Continuous function of \( e \) if all \( f \) continuous:
  \[ \text{case } e \text{ of } ... = \text{ case } \bigcup_{n} e \text{ of } ... \]
- Also continuous function of each \( f_i \)
  \[ \text{case } e \text{ of } f_i \in \bigcup_{n} f = \text{ case } e \text{ of } \bigcup_{n} f \in \bigcup_{n} f(d_i) \]

Continuous functions
- Given cpos \( D, E \), define \( D \rightarrow E \) as domain of continuous functions mapping \( D \) to \( E \) (subset of \( E^D \))
- Pointwise ordering: \( f \sqsubseteq g \iff f(d) \sqsubseteq g(d) \)
- Complete?
  \[ \bigcup_{n} f(n) = \lambda d \in D . \bigcup_{n} f(n)(d) \text{ continuous?} \]
  \[ (λd.\bigcup_{n} f(n)(d))((\bigcup_{m} d_m) = \bigcup_{m} (λd.\bigcup_{n} f(n)(d))(d_m) \]

Proof of Continuity
\[ (λd \in D . \bigcup_{n} f(n)(d)) \bigcup_{m} d_m = \bigcup_{m} (λd \in D . \bigcup_{n} f(n)(d) \bigcup_{m} d_m) \]
\[ = \bigcup_{m} \bigcup_{n} f(n)(d_m) \]
\[ = \bigcup_{m} \bigcup_{n} f(n)(d_m) \]
\[ = \bigcup_{m} \bigcup_{n} f(n)(d_m) \]
\[ = \bigcup_{m} \bigcup_{n} (λd \in D . \bigcup_{n} f(n)(d)) \bigcup_{m} d_m) \]

Lemma
\[ \bigcup_{n} \bigcup_{m} f(n)(d_m) = \bigcup_{n} f(n)(d_n) = \bigcup_{m} \bigcup_{n} f(n)(d_m) \]
Let \( n \leq m' \Rightarrow e_{nm} \subseteq e_{n'm'} \)
\[ e_{nm} \subseteq e_{n'm'} \text{ for } n' = \text{max}(m, n), \text{ so } \bigcup_{n,m} e_{nm} \subseteq \bigcup_{n} e_{n'm} \]
\[ e_{nm} \subseteq e_{n'm} \text{ so } \bigcup_{n} e_{nm} \subseteq \bigcup_{n} \bigcup_{m} e_{nm} \subseteq \bigcup_{n,m} e_{nm} \]

Operations on functions
- \( \text{apply } \in (D \rightarrow E) \times D \rightarrow E = \lambda p.\pi.p\pi.p \)
- \( \text{curry } \in ((D \rightarrow E) \rightarrow F) \rightarrow ((D \rightarrow E) \rightarrow F) \)
  \[ = λf.\pi.e.\lambda d.\lambda e.\pi.f(\langle d, e \rangle) \]
- \( \text{compose } \circ \in (D \rightarrow E) \times (E \rightarrow F) \rightarrow (D \rightarrow F) \)
  \[ = \lambda (f, g).\lambda d.\pi.\pi.f(\pi.g(d)) \]
- \( \text{fix } \in (D \rightarrow D) \rightarrow D \) (\( D \) pointed)
  \[ = \lambda g.\pi.e.\lambda d.\pi.g^\pi(\perp) \]
  \[ = \bigcup_{n} \lambda g.\pi.e.\lambda d.\pi.g^\pi(\perp) \] (LUB of \( \text{cont.functions} \))

Meta-Language
- Have defined various constructs that we can use to define continuous functions
- Constructs are a syntax for a meta-language in which only continuous functions can be defined
- How do we know when expression \( \lambda x.e \) is continuous?
- Idea: use structural induction on form of \( e \) so every syntactically valid \( e \) can be abstracted over any variable to produce continuous function
- Problem: structural induction \( \rightarrow \) need to consider open terms \( e \)
Continuity in variables

- Idea: consider a meta-language expression \( e \) to be implicitly function of its free variables
- \( e \) is continuous in variable \( x \) if \( \lambda x. e \) is continuous for arbitrary values of other (non-\( x \)) free variables in \( e \)
- \( e \) is continuous in variables not free in \( e \)
- structural induction: for each syntactic form, show that term is continuous in variables assuming sub-terms are