Interpreting types

- Types can define names; need type environment $\chi : \text{Type} \to \text{Domain}$ to define inductively
- $T \downarrow \chi$ gives domain corresponding to $T$
  - $\text{unit} \downarrow \chi = U$
  - $\text{Int} \downarrow \chi = \mathbb{Z}$
  - $\tau \downarrow \chi = \chi(X)$
  - $\tau_1 \cdot \tau_2 \downarrow \chi = \mathcal{A}[\tau_1] \times \mathcal{A}[\tau_2]$
  - $\tau_1 \to \tau_2 \downarrow \chi = \mathcal{A}[\tau_1] \to \mathcal{A}[\tau_2]$
  - $\mu X. \tau \downarrow \chi = D. \mathcal{A}[\tau \downarrow \chi[X \mapsto D]]$

Recursive types in compilation

- Two implementation options:
  1. Represent types syntactically
  2. Construct fixed points as cyclical graphs (can avoid replication: hash)

Structural equivalence

- Given $T = \mu X. \tau$, $T \equiv \tau(T/X)$
  - $\mu X. X+X+U \equiv \mu X. X+(X+X)+U$
- Language with explicit fold/unfold: expression has unique type
- Typical language w/ structural equivalence: how to decide $\tau_1 \equiv \tau_2$?
  - Idea: two types equivalent if infinite unfoldings are identical
  - Why structural equivalence is rare...

Example

$\mu S. \text{Int} \to S \equiv \mu t. \text{Int} \to (\text{Int} \to t)$?

- Idea: Infinite unfoldings identical if all (finite or infinite) paths in one are possible in the other
  - Don’t need to actually walk down infinite paths
  - Check unfolding under assumption $s = t$?

Simple types

$\begin{align*}
\tau_1 &\equiv \tau_3 \\
\tau_2 &\equiv \tau_4 \\
\tau_1 \cdot \tau_2 &\equiv \tau_3 \cdot \tau_4 \\
\tau_1 \to \tau_2 &\equiv \tau_3 \to \tau_4 \\
\mu X. \tau &\equiv \tau' \\
\tau_1 + \tau_2 &\equiv \tau_3 + \tau_4
\end{align*}$
Solution: add a context

- Algorithm: depth-first tandem walk of types
- Context E records type expressions assumed to be equivalent
- Rule: \( \mu X. \tau \equiv \mu Y. \tau' \) if 
  \( E \vdash \tau \equiv \tau' \)
  - assuming \( \mu X. \tau \equiv \tau \),
  - unfoldings are equiv: \( \tau(\mu X.t:X) \equiv \tau(\mu Y.\tau'/Y) \)

\[
\begin{align*}
E, \mu X. \tau & \equiv \mu Y. \tau' \vdash (\mu X. t:X) \equiv (\mu Y. \tau'/Y) \\
E & \vdash \tau \equiv \tau'
\end{align*}
\]

Recursive domain constructor

- \( \mu D. \mathcal{D}(D) \)
  - Functor \( \mathcal{D} \) maps one domain into another domain
  - \( D = \mu X. \mathcal{D}(X) \) produces a domain related to \( \mathcal{D}(D) \) by continuous functions up and down that are inverses of one another

\[
\begin{align*}
D & \equiv \mathcal{D}(D) \\
& \overset{up}{\Rightarrow} D_1 \equiv d_1 \sqsubseteq d_2 \ldots \in D \Rightarrow up(\sqcup d_i) = \sqcup up(d_i) \\
& \overset{down}{\Rightarrow} down(\sqcup e_i) = \sqcup down(e_i)
\end{align*}
\]

Denotational Models

- We have left up and down implicit – can fold into notion of domain injection (ala ML):

\[
\begin{align*}
\text{Result} \equiv (\text{Value} + \text{Error}) \bot \\
\in \text{Value} = \lambda \nu. \text{up} \text{Value} = \lambda_\nu (\text{in}_\text{Value}(\nu))
\end{align*}
\]

For what functors (maps from domains to domains) \( \mathcal{D} \) can we take a fixed point?

- functors built out of sum, product, lifting, lifted function space, discrete CPOs
- Can we define fixed point constructor as \( \mu D. \mathcal{D}(D) = \sqcup \mathcal{D}(0) \)
  - for appropriate \( \mathcal{D} \) if we define \( \sqcup \) correctly
  - won’t always work: \( \mathcal{D}(\Lambda) = \Lambda \rightarrow \Lambda \)

\[
\begin{align*}
\mathcal{D}(0) & \equiv \top \\
\mathcal{D}(\top) & \equiv \top + \top \equiv \top \\
\mathcal{D}(\bot) & \equiv \bot
\end{align*}
\]

Questions

- Maps one domain into another
  - elements
  - ordering relations
- To have fixed point
  - must be monotonic
  - must preserve fixed points within domains

\[
\begin{align*}
D & \equiv \mathcal{D}(D) \\
& \overset{up}{\Rightarrow} D_1 \equiv d_1 \sqsubseteq d_2 \ldots \in D \Rightarrow up(\sqcup d_i) = \sqcup up(d_i) \\
& \overset{down}{\Rightarrow} down(\sqcup e_i) = \sqcup down(e_i)
\end{align*}
\]
Solving equations

- Previous recipe: construct a functor \( F \) whose fixed point is solution; find least fixed point
- \( N = F(F(F(...F(∅)))) = \text{fix } F(∅) \)
- For some functions, inductive definition suffices:
  \[
  N = \text{unit} + N \quad \text{in}_1(\text{unit}) \in N \quad \text{in}_x(x) \in N
  \]

\[
  FN' = \{ \text{in}(u), \text{in}(\text{in}(u)), \text{in}(\text{in}(\text{in}(u))) \}
  \]

\[
  \text{fix } F(∅) = \{ \text{in}_1(\text{unit}), \text{in}_2(\text{in}_1(\text{unit})), \text{in}_2(\text{in}_2(\text{in}_1(\text{unit}))) \}
  \]

Isomorphic to natural numbers… are we done?

Problem: completeness

- Consider \( N \equiv \text{unit} + N \)
- Inductive definition gives
  \[
  \text{in}_1(u), \text{in}_2(\text{in}_1(u)), \text{in}_3(\text{in}_2(\text{in}_1(u))), \ldots (0, 1, 2, \ldots)
  \]

\[
  \text{CPO? (Note } 0 \perp 1 \perp 2 \perp \ldots\text{)}
  \]

Lazy language:

\[
  \infty = \text{rec } n: (\text{µ } N. \text{unit} + N) . \text{inr}(n)
  \]

Problem: Cardinality

- What about domain corresponding to type \( \text{µ } T. \text{ T } \rightarrow \text{ bool } \)?
  - set of continuous functions from infinite cpo \( D \) to truth value \( \text{T} \)
    - isomorphic to powerset \( \text{℘}(D) \)
  - Cantor’s diagonal argument: no isomorphism between \( D \) and \( \text{℘}(D) \) (Winskel, Ch.1)

- No solution to \( D \equiv D \rightarrow T \perp ? \)
- Can find solution for some domains
- One important class: bc-domains / Scott domains

Example: Integer Lists

\[
  L \equiv \text{Z }^*(\text{unit } + L)
  \]

- Solution 1: all finite lists of integers
  - anything buildable using finite \# of up’s (inductively defined) – countable set
  - adequate for a CBV language

- Solution 2: all finite or infinite lists of integers
  - CBN language: \( \text{rec } x (\text{l}, (2, x)) \)
    - anything that looks like a list: can apply a finite number of down’s (co-inductive defn) – uncountable set, only a infinitesimal fraction computable
    - Only infinite lists that are limits of finite lists are constructable – countable set!

“Finite” vs. “Infinite” elements

- Problem: how to control the “infinite” elements
- Know how to generate all the finite elements using rule induction as previously, need to add all the “infinite” elements
  - without increasing cardinality
- “Finite” elements are the compact elements
  - \( x \) is compact if for every chain \( M \) where \( x \leq M \), there exists a \( y \) in \( M \) such that \( x \leq y \)
- Idea: set of compact elements \((0, 1, 2, \ldots)\) defines a basis
  - form which the non-compact elements (e.g. infinity) can be extrapolated.
- Basis for domain \( D \) is \( \text{K}(D) \): contains finite approximations to the non-compact elements of \( D \)

Algebraic domains

- bc-domain \( D \) must be algebraic: every element must be LUB of the compact elements \( \sqsubseteq \) it.
  - directed set: all pairs of elements \( a, b \) have least upper bound \( a \sqcup b \) in the set
  - for all \( x \in D \) the set \( M = \{ a \in \text{K}(D) \mid a \sqsubseteq x \} \) is directed, \( x = \sqcup M \)
  - structure of non-compact elements determined completely by compact elements – “no surprises at infinity”
bc-domains

- Another problem: given algebraic domains D, E, domain of continuous functions D→E may not be algebraic! (Example: Gunter Ch. 5)
- Can fix by requiring domains to be bounded complete: if two elements in D have an upper bound, they have a least upper bound
- bc-domain: bounded-complete, algebraic CPO
  - Restricts compact and non-compact elements of D so continuous functions D→E can form a bc-domain of the same cardinality as D
  - Information systems (Winskel, Ch. 12) are a way to generate functors that always map bc-domains to bc-domains properties, allowing fixed points over bc-domains. (Also defines CPO over domains)