CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 31
Recursive Domains
8 Nov 00

Interpreting types

- Types can define names; need type environment χ : Type $\to$ Domain to define inductively

$\mathcal{T}[\![\tau]\!]\chi$ gives domain corresponding to τ

$\mathcal{T}[\![\text{unit}]\!]\chi = \text{U}$

$\mathcal{T}[\![\text{int}]\!]\chi = \text{Z}$ Compiler algorithm!

$\mathcal{T}[\![X]\!]\chi = \chi(X)$

$\mathcal{T}[\![\tau_1 * \tau_2]\!]\chi = \mathcal{T}[\![\tau_1]\!]\chi \times \mathcal{T}[\![\tau_2]\!]$

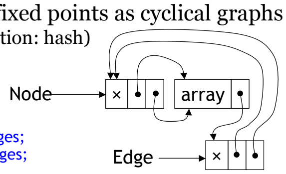
$\mathcal{T}[\![\tau_1 \to \tau_2]\!]\chi = \mathcal{T}[\![\tau_1]\!]\chi \to \mathcal{T}[\![\tau_2]\!]_\perp$

$\mathcal{T}[\![\mu X.\tau]\!]\chi = \mu D.\, \mathcal{T}[\![\tau]\!]\chi[X \mapsto D]$

Recursive types in compilation

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

μ = loop




```
class Node {
    Edge[] outgoing_edges;
    Edge[] incoming_edges;
}
class Edge {
    Node from;
    Node to;
}
```

Node = μN. array[N*N] * array[N*N]
Edge = μE.(array[E] * array[E]) * (array[E] * array[E])

Structural equivalence

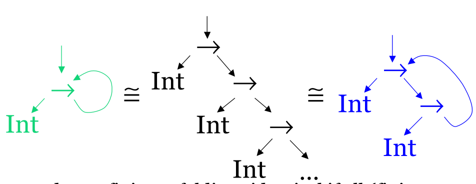
- Given T = μX. τ, T $\cong \tau\{\text{T}/\text{X}\}$

μX. X$*$X+U $\cong$ μX. (μX. X$*$X+U)$*$X+U
$\cong$ μX. X$*$(μX. X$*$X+U)+U

- Language with explicit fold/unfold: expression has unique type
- Typical language w/ structural equivalence: how to decide $\tau_1 \cong \tau_2$?
- Idea: two types equivalent if *infinite* unfoldings are identical
  – Why structural equivalence is rare...

Example

$\mu s.\text{Int}\to s \cong \mu t.\text{Int}\to(\text{Int}\to t)$?



- 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 $s = t$?
    Doesn't work

Simple types

$$\frac{\tau_1 \cong \tau_3 \quad \tau_2 \cong \tau_4}{\tau_1 * \tau_2 \cong \tau_3 * \tau_4} \qquad \frac{\tau_1 \cong \tau_3 \quad \tau_2 \cong \tau_4}{\tau_1 \to \tau_2 \cong \tau_3 \to \tau_4}$$

$$\frac{\tau_1 \cong \tau_3 \quad \tau_2 \cong \tau_4}{\tau_1 + \tau_2 \cong \tau_3 + \tau_4} \qquad \frac{?}{\mu X.\tau \cong \tau'}$$

$$\frac{?}{\mu X.\tau \cong \mu Y.\tau'}$$

Solution: add a context

- Algorithm: depth-first tandem walk of types
- Context E records type expressions assumed to be equivalent
- Rule: $\mu X.\tau \cong \mu Y.\tau'$ if
  - assuming $\mu X.\tau \cong \mu Y.\tau'$,
  - unfoldings are equiv: $\tau\{\mu X.\tau/X\} \cong \tau\{\mu Y.\tau'/Y\}$

$$\frac{\tau \cong \tau' \in E}{E \vdash \tau \cong \tau'}$$

$$\frac{E,\ \mu X.\tau \cong \mu Y.\tau' \vdash \tau\{\mu X.\tau/X\} \cong \tau\{\mu Y.\tau'/Y\}}{E \vdash \mu X.\tau \cong \mu Y.\tau'}$$

$$\frac{E,\ \mu X.\tau \cong \tau' \vdash \tau\{\mu X.\tau/X\} \cong \tau'}{E \vdash \mu X.\tau \cong \tau'} \qquad \frac{\begin{array}{c}E \vdash \tau_1 \cong \tau_3 \\ E \vdash \tau_2 \cong \tau_4\end{array}}{E \vdash \tau_1 \oplus \tau_2 \cong \tau_3 \oplus \tau_4}$$

Example

$\mu s.\ (s \to s) \to s \cong \mu t.\ t \to (t \to t)$?

Let $S = \mu s.\ (s \to s) \to s$, $T = \mu t.\ t \to (t \to t)$

Proof: (simple to implement with graph representation of types: E = set of pairs of pointers)

$$\dfrac{\dfrac{\dfrac{\{\ldots\} \vdash S = T \qquad \dfrac{\dfrac{\{\ldots\} \vdash S \to S = T \qquad \{\ldots\} \vdash S = T}{\{S{=}T,\ S \to S{=}T,\ S = T \to T\} \vdash (S \to S) \to S = T \to T}}{\{\ldots\} \vdash S = T \to T}}{\{S{=}T,\ S \to S{=}T\} \vdash S \to S = T \to (T \to T)} \qquad \dfrac{\ldots}{\{S{=}T\} \vdash S = (T \to T)}}{\{S{=}T\} \vdash (S \to S) \to S = T \to (T \to T)}}{\varnothing \vdash S = T}$$

Recursive domain constructor

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

$$D \underset{down}{\overset{up}{\rightleftarrows}} \mathcal{F}(D)$$

$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \ldots \in D \Rightarrow up(\bigsqcup d_i) = \bigsqcup up(d_i)$

$e_0 \sqsubseteq e_1 \sqsubseteq e_2 \ldots \in \mathcal{F}(D) \Rightarrow down(\bigsqcup e_i) = \bigsqcup down(e_i)$

Denotational Models

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

$\text{Result} \cong (\text{Value} + \text{Error})_\perp$

$in_{\text{Result} \leftarrow \text{Value}} = \lambda v.up_{\text{Result} \leftarrow \text{Value}} \lfloor in_1(v) \rfloor$

$\rho \in \text{Var} \to \text{Value}$

$\mathcal{C}[\![x]\!]\rho = in_{\text{Result} \leftarrow \text{Value}}(\rho x)$

…

Questions

- For what functors (maps from domains to domains) $\mathcal{F}$ 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
  $\mu D\,.\,\mathcal{F}(D) = \bigsqcup \mathcal{F}^n(\mathbf{0})$
  where $\mathbf{0}$ is the empty domain?
  - for appropriate $\mathcal{F}$ if we define $\bigsqcup$ correctly
  - won't always work: $\mathcal{F}(\Lambda) = \Lambda \to \Lambda$

$\mathcal{F}(\mathbf{0}) \cong U \qquad \mathcal{F}^2(\mathbf{0}) \cong U \to U \cong U \qquad \mathcal{F}^n(\mathbf{0}) \cong U$

Functor properties

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

$$D \underset{down}{\overset{up}{\rightleftarrows}} \mathcal{F}(D)$$

Solving equations

- Previous recipe: construct a functor F whose fixed point is solution ; find least fixed point
- $N = F(F(F(F(\ldots F(\varnothing))))) = \text{fix } F(\varnothing)$
- For some functions, inductive definition suffices:

$N \cong \text{unit} + N$ $\qquad \dfrac{}{in_1(\text{unit}) \in N} \qquad \dfrac{x \in N}{in_2(x) \in N}$

$F(N') = \{ in_1(u) \} \cup \{in_2(x) \mid x \in N\}$
fix $F(\varnothing) = \{in_1(u), in_2(in_1(u)), in_2(in_2(in_1(u)))\ldots\}$

- Isomorphic to natural numbers… are we done?

Problem: completeness

- Consider $N \cong \mathbb{U} + N_\perp$
- Inductive definition gives

$in_1(u), in_2(in_1(u)), in_2(in_2(in_1(u))), \ldots$ (0, 1, 2,…)
$in_2(\perp), in_2(in_2(\perp)), in_2(in_2(in_2(\perp))), \ldots$ ($0_\perp, 1_\perp, 2_\perp, \ldots$)

CPO? (Note $0_\perp \sqsubseteq 1_\perp \sqsubseteq 2_\perp \sqsubseteq \ldots$)
Lazy language:
∞ = rec n: (μN.unit+N) . inr(n)

Problem: Cardinality

- What about domain corresponding to type μT. T→bool ?
  - set of continuous functions from infinite cpo D to truth value $\mathbb{T}$ is isomorphic to powerset $\wp$(D)
  - Cantor's diagonal argument: no isomorphism between D and $\wp$(D) (Winskel, Ch.1)
- No solution to $D \cong D \to \mathbb{T}_\perp$?
- Can find solution for some domains
- One important class: bc-domains / Scott domains

Example: Integer Lists

$L \cong \mathbb{Z}^*(\mathbb{U} + L)$

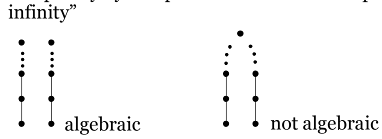
- Solution 1: all finite lists of integers
  - anything buildable using finite # of up's (inductively defined) – countable set
  - adequate for a CBV language
- Solution 2: all finite or infinite lists of integers
  - CBN language: (rec x $\langle 1, \langle 2, x\rangle\rangle$)
  - anything that looks 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
  - infinite elements of interest are limits of chains of finite values
  - without increasing cardinality
- "Finite" elements are the *compact* elements
  - x is compact if for every chain M where x $\sqsubseteq \bigsqcup$M, there exists a y in M such that x $\sqsubseteq$ y
- Idea: set of compact elements (0,1,2,…) defines a *basis* from which the non-compact elements (e.g. infinity) can be extrapolated.
- *Basis* for domain D is 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 = { $a \in$ K(D) | $a \sqsubseteq x$ } is directed, $x = \bigsqcup$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 $D \to E_\perp$ 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 $\trianglelefteq$ over domains)