CS 611: Lecture 31

Recursive types and domains

November 10, 1999

Cornell University Computer Science Department
Andrew Myers

Data structures?

- Last time: finally got a type-safe universal language
- Inconvenient: no data structures
- Binary tree in C:

```
struct Tree {
  bool leaf;
  union {
    struct { Tree *left, Tree *right; } children;
    int value;
  } u;
}
```

- How to express in our type notation? A try:
  tree = tree ∗ tree + int
- Equation!

Other examples:

- How to assign a type to (λ x (x x))?
- Can write terms that don't get stuck:
  (λ x (x x)) (λ x (x x)) diverges
  (λ x (x x)) (λ y #f) ⇓ #f
- Need solution to $\tau = \tau \rightarrow \text{bool}$

Outline

- Fixed point type constructor
- Type unfolding & equivalence
- Recursive types as domain equations
- bc-domains

Fixed point type constructor

- Want to solve equations of form X = τ where X is type variable mentioned in τ
- Type constructor $\mu X.\tau$ produces this solution
  - analogue of rec *x e* for types
  - ≈ ML datatype declaration
- Can define useful types:

$$tree = \mu T.\ T * T + \text{int}$$

$$nat = \mu N.\ \text{unit} + N$$

0 = inl(#u), 1 = inr(inl(#u)), 2 = inr(inr(inl(#u)))…

succ = λn: *nat* . inr(n)

lambda calculus: $\mu D.D \rightarrow D$

Type equivalence

- Problem: types no longer have one unique syntactic form
- $\mu X.\tau$ is solution to $X = \tau$: can substitute $\mu X.\tau$ for X wherever it appears in τ
- *Unfolding* of type $\mu X.\tau$ is *equivalent* type $\tau[\mu X.\tau / X]$
  - strong notion of equivalence: type expressions are fully substitutable for each other
  - weaker notion: types are isomorphic & bijection exists between the type and its unfolding.
    $\mu X.\tau \cong \tau[\mu X.\tau / X]$

Abstract and concrete views

down
rep
*
$\mu X.\tau \cong \tau[\mu X.\tau / X]$
up
abs
&

• down operator allows access to internals of recursive type; up operator packages concrete value as abstract value
• up/down are bijection: types are isomorphic

up and down in real languages

- Modula-3: recursive type and unfolding are substitutable: up/down supplied automatically as needed
- ML: datatype constructor is up, match operation provides implicit down for each arm of the sum
- Java: class constructor is up and down is supplied automatically
- CLU, C require explicit use of operators to shift between views (up/down), (*/&)
- Simplification: in all these languages up and down are coupled with other features (ML: sums, CLU & Java: classes/modules, C: references)

Typing rules

$$\frac{\Gamma \vdash e : \tau[\mu X.\tau / X]}{\Gamma \vdash \text{up } e : \mu X.\tau}$$

$$\frac{\Gamma \vdash e : \mu X.\tau}{\Gamma \vdash \text{down } e : \tau[\mu X.\tau / X]}$$

Example

- Goal: show (λ x (x x)) can be typed as μT. T→bool
- Add declarations, up and down operators: up (λ x : μT. T→bool ((down x) x))

$$\dfrac{\dfrac{\dfrac{\dfrac{\{x : \mu T.\ T\to bool\} \vdash x : (\mu T.\ T\to bool)}{\{x : \mu T.\ T\to bool\} \vdash \text{down } x : (\mu T.\ T\to bool)\to bool} \quad \dfrac{\{x : \mu T.\ T\to bool\} \vdash x : (\mu T.\ T\to bool)}{\{x : \mu T.\ T\to bool\} \vdash ((\text{down } x)\ x) : bool}}{\vdash [\ \lambda x : \mu T.\ T\to bool\ ((\text{down } x)\ x)\ ] : (\mu T.\ T\to bool)\to bool}}{\vdash [\ \text{up } (\lambda x : \mu T.\ T\to bool\ ((\text{down } x)\ x))\ ] : \mu T.\ T\to bool}$$

Y operator

- Can use recursive types to write Y_τ operator as ordinary lambda expressions
- Desugar rec x: τ . e as $Y_\tau(\lambda x . e)$!

Constructing a model

- Our semantics models τ as domain $\mathcal{T}[\![\tau]\!]$
- How do we model the new type constructor?
  $\mathcal{T}[\![\mu X.\tau]\!] = ?$
- Since $\mu X.\tau \cong \tau[\mu X.\tau / X]$, we expect isomorphism to hold in domains as well:
  $\mathcal{T}[\![\mu X.\tau]\!] \cong \mathcal{T}[\![\tau[\mu X.\tau / X]\ ]\!]$
- Example: natural numbers
  $N = \mathcal{T}[\![\mu N.\ \text{unit} + N]\!] \cong \mathcal{T}[\![\text{unit} + (\mu N.\ \text{unit} + N)\ ]\!]$
  $N \cong \text{unit} + N$
- Model for this types are solutions to domain equations we have been using all along

Solving equations

- Our recipe: construct a functor F whose fixed point is solution ; find least fixed point
- $N = F(F(F(F(F(...F(\emptyset)))))) = \text{fix } F(\emptyset)$
- Can build functors based on inference rules:

$N \cong \text{unit} + N$

$$\frac{}{in_1(\text{unit}) \in N} \qquad \frac{x \in N}{in_2(x) \in N}$$

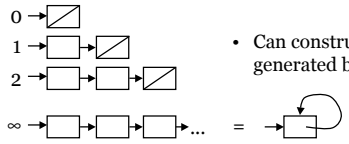
$F(N) = \{ in_1(unit) \} \cup \{in_2(x) \mid x \in N\}$

$\text{fix } F(\emptyset) = \{in_1(unit), in_2(in_1(unit)), in_2(in_2(in_1(unit)))...\}$

- Isomorphic to natural numbers... are we done?

Problem: fixed point operator

- Existing functor only generates the finite natural numbers
- Language has a fixed point operator; can take fixed point over naturals:
- rec n: nat . succ(n) = rec n:(μN. unit + N) . up(inr(n))



- Can construct values not generated by rules!

Problem: Cardinality

- What about domain corresponding to type μT. T→bool ?
- Problem: set of continuous functions from infinite cpo D to truth value T has in general cardinality $2^{|D|}$; no isomorphism can be arranged between D and 2^D since $|D| < 2^{|D|}$
- No solution to $D \cong D \to T$?
- Can find solution for some domains
- One important class: bc-domains / Scott domains

"Finite" vs. "Infinite" elements

- Intuition: know how to generate all the finite elements using rule induction as previously
- Need to add all the "infinite" elements that might be the result of taking a fixed point over finite values.
- "Finite" elements are the *compact* elements
  - x is compact if for every directed subset M where x M, there exists a y in M such that x y
  - directed set generalizes a chain: every subset has a LUB.
- Idea: set of compact elements (0,1,2,...) defines a *basis* from which the non-compact elements (∞) can be extrapolated.
- Basis for domain D is K(D); K(D) contains finite approximations to the non-compact elements of D

Additional properties

- bc-domain D must be *algebraic*: every element must be the least upper bound of the compact elements less than it.
  - for all x ∈ D the set M = { a∈ K(D) | a x } is directed and x = M
- Finally, must be *bounded complete*: every subset with an upper bound in D has a *least* upper bound
- These properties restrict both the compact and non-compact elements of D so cardinality problems do not arise; continuous functions from D→E can form a bc-domain of the same cardinality as D
- *Information systems* (Ch. 12) are a way to generate functors that preserve the bc-domain properties, allowing fixed points over bc-domains. See also Gunter, Ch. 5.