CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 30
Recursive Types
5 Nov 00

tF example

- Last time: finally got a type-safe universal language (tF)

$e ::= x \mid b \mid \text{fn } x{:}\tau\,.\,e \mid e_1\, e_2 \mid e_1 + e_2 \mid \langle e_1, e_2\rangle \mid$
$\quad \text{first } e \mid \text{rest } e \mid \text{inl } e \mid \text{inr } e \mid \text{case } e_0\, e_1\, e_2 \mid$
$\quad \text{rec } y{:}\tau.\ \text{fn } x.e \mid \text{let } x{=}e_1 \text{ in } e_2$
$\tau ::= B \mid \tau_1 \to \tau_2 \mid \tau_1 {*} \tau_2 \mid \tau_1 + \tau_2$

```
let factorial = (rec fact: int→int.
   fn n. if (n<2) 1 (fact(n-1)*n)) in          : int
factorial(5)
```


Type equivalence

- Many languages: multiple ways to write a type
- Write $\vdash \tau_1 \cong \tau_2$ when type expressions τ_i are equivalent types:

$$\frac{\vdash \tau_1 \cong \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2}$$

Example: Extend tF with $(\tau_1{*}\tau_2)\to\tau_3 \cong \tau_1\to(\tau_2\to\tau_3)$

- curried fcns applicable to pairs, vice versa
- based on (compiler-inserted) bijection between types: *curry* and *uncurry*

$\mathcal{C}[\![e : (\tau_1{*}\tau_2)\to\tau_3]\!] = \mathcal{C}[\![\lambda x{:}\tau_1\,.\,\lambda y{:}\tau_2\,.\,e\langle x,y\rangle : \tau_1\to(\tau_2\to\tau_3)]\!]$
$\mathcal{C}[\![e : \tau_1\to(\tau_2\to\tau_3)]\!] = \mathcal{C}[\![\lambda p{:}\tau_1{*}\tau_2\,.\,e(\pi_1 p)(\pi_2 p) : (\tau_1{*}\tau_2)\to\tau_3]\!]$

Name vs. structural equivalence

- When are two types equivalent?
  - **tF :** identical syntactic form (*structural equivalence*)
  - **C, Java, Pascal:** *name equivalence* (name is part of type identity)
  - C: struct foo { int a, b; } ≇ struct bar { int a, b; }
    - typedef struct foo Foo; Foo ≅ struct foo
- Binding a type to a name also brands it with unique identity
  - **Modula-3:** structural equivalence, explicit branding:
    - TYPE intpair = RECORD x,y: int END
    - TYPE foo= BRANDED intpair, bar = BRANDED intpair;
    - foo ≇ bar ≇ intpair ≅ RECORD x,y: int END

Data structures?

- tF is 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:
  - $\tau = \text{bool}{*}(\tau * \tau + \text{int})$
- 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 $\tau = \tau\to\text{bool}$

Fixed point type constructor

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

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

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

$$0 \triangleq \text{inl}(\#u),\ 1 \triangleq \text{inr}(\text{inl}(\#u)),\ 2 \triangleq \text{inr}(\text{inr}(\text{inl}(\#u)))\ldots$$

$$\text{successor} \triangleq \lambda n{:}\ nat\ .\ \text{inr}(n)$$

Closed vs. Open recursion

- Many modern languages allow types to refer to one another arbitarily (even to other source file!)
- *Open recursion*: type expression not closed
- Requires fixed point over all types in scope

```
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])
```


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

$$\mu N.(\text{unit} + N) = \mu N.(\text{unit} + (\mu N.\ \text{unit} + N))$$

$$nat = (\text{unit} + nat) = (\text{unit} + (\text{unit} + nat))\ \ldots$$

- *Unfolding* of type $\mu X.\tau$ is *equivalent* type $\tau\{\mu X.\tau / X\}$
- $\mu X.\tau \cong \tau\{\mu X.\tau / X\}$
  - implicit notion of equivalence: type expressions are fully substitutable for each other
  - weaker notion: types are isomorphic, expressions must be explicitly mapped between types

Abstract and concrete views

$$\mu X.\tau \underset{\text{fold / abs}}{\overset{\text{unfold / rep}}{\rightleftarrows}} \tau\{\mu X.\tau / X\}$$

- unfold operator allows access to internals of value of recursive type; fold operator packages concrete value as abstract value
- Winskel: abs/rep
- fold/unfold are bijection: types are isomorphic

Typing, evaluation rules

$$\frac{\Gamma \vdash e : \tau\{\mu X.\tau / X\}}{\Gamma \vdash \text{fold}_{\mu X.\tau}\ e : \mu X.\tau}$$

$$\frac{\Gamma \vdash e : \mu X.\tau}{\Gamma \vdash \text{unfold}\ e : \tau\{\mu X.\tau / X\}}$$

$$\text{unfold}\ (\text{fold}_{\mu X.\tau}\ e) \rightarrow e$$

$$C ::= \ldots \mid \text{fold}\ C \mid \text{unfold}\ C$$

Example

- Goal: show (λ x (x x)) can be typed as $\mu T.\ T \rightarrow \text{bool}$
- Add declarations, explicit fold and unfold: $\text{fold}_{\mu T.\ T\rightarrow\text{bool}}(\lambda\ x{:}\ (\mu T.\ T\rightarrow\text{bool})\ .\ ((\text{unfold}\ x)\ x))$

$$\frac{\dfrac{\dfrac{\dfrac{\{x : \mu T.\ T\rightarrow\text{bool}\} \vdash x : (\mu T.\ T\rightarrow\text{bool})}{\{x : \mu T.\ T\rightarrow\text{bool}\} \vdash \text{unfold}\ x : (\mu T.\ T\rightarrow\text{bool})\rightarrow\text{bool} \quad \ldots}}{\{x : \mu T.\ T\rightarrow\text{bool}\} \vdash ((\text{unfold}\ x)\ x) : \text{bool}}}{\vdash [\ \lambda\ x{:}\ (\mu T.\ T\rightarrow\text{bool})\ .\ ((\text{unfold}\ x)\ x)\ ] : (\mu T.\ T\rightarrow\text{bool})\rightarrow\text{bool}}}{\vdash [\ \text{fold}_{\mu T.\ T\rightarrow\text{bool}}\ (\lambda\ x{:}\ (\mu T.\ T\rightarrow\text{bool})\ .\ ((\text{unfold}\ x)\ x))\ ] : \mu T.\ T\rightarrow\text{bool}}$$

Capturing the untyped λ

- All lambda calculus expressions can be used in any context
  - Evaluation never gets stuck
- Type of lambda calculus terms:
  $\Lambda = \mu X.X \rightarrow X \;\cong\; (\mu X.X \rightarrow X) \rightarrow (\mu X.X \rightarrow X)$
- Translating λ to $\lambda^{\rightarrow\mu}$: (note $\Lambda \cong \Lambda \rightarrow \Lambda$)
  $\mathcal{D}[\![x]\!] = x$
  $\mathcal{D}[\![\lambda\, x\, e]\!] = (\text{fold}\ (\lambda x{:}\Lambda\ .\ \mathcal{D}[\![e]\!]))$
  $\mathcal{D}[\![e_1\, e_2]\!] = (\text{unfold}\ \mathcal{D}[\![e_1]\!])\ \mathcal{D}[\![e_2]\!]$

fold and unfold in real languages

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

Y operator

- Can use recursive types to write Y_τ operator as ordinary expression
- Desugar rec $f{:}\tau.\ e_r$ as $Y_\tau\,(\lambda f{:}\tau\ .\ e_r)$!

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
  $N = \mathcal{T}[\![\mu\text{N. unit + N}]\!] \cong \mathcal{T}[\![\text{unit + } (\mu\text{N. unit + N})]\!]$
  $N \cong \text{unit} + N$
- Modeling these types requires solutions to domain equations we have been using all along

Recursive domain constructor

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

up
$D \cong \mathcal{F}(D)$
down

$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \ldots \in D \;\Rightarrow\; up(\sqcup d_i) = \sqcup up(d_i)$
$e_0 \sqsubseteq e_1 \sqsubseteq e_2 \ldots \in \mathcal{F}(D) \;\Rightarrow$
$down(\sqcup e_i) = \sqcup down(e_i)$

Interpreting types

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

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

$\mathcal{T}[\![\text{unit}]\!]\chi = \mathbb{U}$
$\mathcal{T}[\![\text{int}]\!]\chi = \mathbb{Z}$
$\mathcal{T}[\![X]\!]\chi = \chi(X)$
$\mathcal{T}[\![\tau_1 * \tau_2]\!]\chi = \mathcal{T}[\![\tau_1]\!]\chi \times \mathcal{T}[\![\tau_2]\!]\chi$
$\mathcal{T}[\![\tau_1 \rightarrow \tau_2]\!]\chi = \mathcal{T}[\![\tau_1]\!]\chi \rightarrow \mathcal{T}[\![\tau_2]\!]\chi_\perp$
$\mathcal{T}[\![\mu X.\tau]\!]\chi = \mu D.\ \mathcal{T}[\![\tau]\!]\chi[X \mapsto D]$