CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

Recap

- Type rules.
- Simply typed lambda calculus.
- Type soundness proof.

Recursive types

- A data type for values that may contain other values of the same type.
- Also called inductive data types.
- Compared to simple types that are finite, recursive types are not.

```
interface I {
   void s1(boolean a);
   int m1(J a);
}

interface J {
  boolean m2(I b);
}
```

- Infinite graph.

Recursive types

- Can be viewed as directed graphs.
- Useful for defining dynamic data structures such as Lists, Trees.
- Size can grow in response to runtime requirements (user input); compare that to static arrays.

Equality and subtyping

- In Java two types are considered equal iff they have the same name. Tricky example?
- Same with subtyping.
- Contrast the name based subtyping to structural subtyping.
- Why is structural subtyping interesting?

Grammar for recursive types

- We will extend the grammar of our simple types.
- $$t ::= t_1 \to t_2 | \text{Int} | \alpha | \mu\alpha.(t_1 \to t_2)$$

  where
  - α is a variable that ranges over types.
  - $\mu\alpha.t$ - is a recursive type that allows unfolding.

    $$\mu\alpha.t = t[\alpha := (\mu\alpha.t)]$$
  - Example: Say $u = \mu\alpha.(\alpha \to \text{Int})$. Now unfold
    - Once: $u = u \to \text{Int}$
    - Twice: $u = (u \to \text{Int}) \to \text{Int}$
    - ...
    - Infinitely: Infinite tree - the type of u.
  - A type derived from this grammar will have finite number of *distinct* subtrees - *regular* trees.
  - Any regular tree can be written as a finite expression using μs.

Type derivation example

- Type of the lambda term $\lambda x.xx$.
- Use a type $u = \mu\alpha.(\alpha \to \text{Int})$.
- $$\dfrac{\dfrac{\phi[x:u] \vdash x : u \to \text{Int} \qquad \phi[x:u] \vdash x : u}{\phi[x:u] \vdash xx : \text{Int}}}{\phi \vdash \lambda x : u.xx : u \to \text{Int}}$$

Type derivation, example II

- $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- Y-combinator is also called fixed point combinator or paradoxical combinator.
- When applied to any function g, it produces a fixed point of g.
- That is $Y(E) = E(Y(E))$
- $$\begin{array}{lll} Y(E) & =_\beta & (\lambda x.E(xx))(\lambda x.E(xx)) \\ & =_\beta & E((\lambda x.E(xx))(\lambda x.E(xx))) \\ & =_\beta & E(Y(E)) \end{array}$$

Useless assignment: For the factorial function $F = \lambda f.\lambda n.if\ (zero?\ n)\ 1\ (mult\ n\ (f\ pred\ n))$, show that $(Y\ F)\ n$ computes factorial n.
Use the definition of factorial function:
Fact n = if (zero? n) 1 (mult n (Fact (pred n))) Ueless assignment II: Write the Y combinator in Scheme.

Type derivation of Y-combinator

- Y combinator cannot be typed with simple types.
- Use a type $u = \mu\alpha.(\alpha \to \text{Int})$.
- $$\frac{\phi[f : Int \to \text{Int}] \vdash (\lambda x.f(xx))(\lambda x.f(xx)) : \text{Int}}{\phi \vdash \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (\text{Int} \to \text{Int}) \to \text{Int}}$$
- If we can get the type of $\lambda x.f(xx)$ to be type u then using $u = u \to \text{Int}$ like above, we can get the premise.
- Goal $\phi[f : \text{Int} \to \text{Int}] \vdash \lambda x.f(xx) : u$
- $$\cfrac{\cfrac{\phi[f : \text{Int} \to int][x : u] \vdash f : \text{Int} \to \text{Int} \qquad \phi[x : u] \vdash xx : \text{Int}}{\phi[f : \text{Int} \to \text{Int}][x : u] \vdash f(xx) : \text{Int}}}{\phi[f : \text{Int} \to \text{Int}] \vdash \lambda x : u.f(xx) : u}$$
- Not all terms can be typed with recursive types either: $\lambda x.x(\text{succ } x)$
- Type soundness theorem can be proved for recursive types as well.

Equality of types

- Isorecursive types: $\mu\alpha.t$ and $t[\alpha/\mu\alpha.t]$ are distinct (disjoint) types.
- Equirecursive types: Type type expessions are same if their infinite trees match.
  - Direct comparison is not enough.
  - Convert a given type into a canonical (normal/standard) form and then compare.

Representation of types - as functions

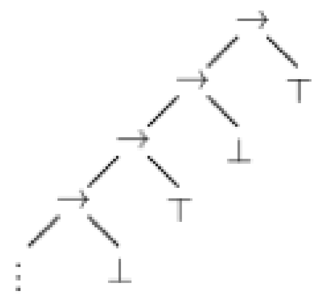
- Denote an alphabet Σ that contains all the labels and paths of the type tree.
- We can represent such a tree by a function that maps *paths* to *labels* — called a *term*.
- Say we denote *left* by 0 and *right* by 1, for the types discussed before: path $\in \{0,1\}^*$.
- And the labels are from the set $\Sigma = \{\text{Int}, \to\}$.
- A term t over Σ is a partial function

$$t : \{0,1\}^* \to \Sigma$$

- The domain $D(t)$ must satisfy:
  - $D(t)$ is non-empty and is prefix-closed.
  - if $t(\alpha) = \to$ then $\alpha 0$, $\alpha 1 \in D(t)$.

Types as functions (contd)

- Example.



- The term is given by:

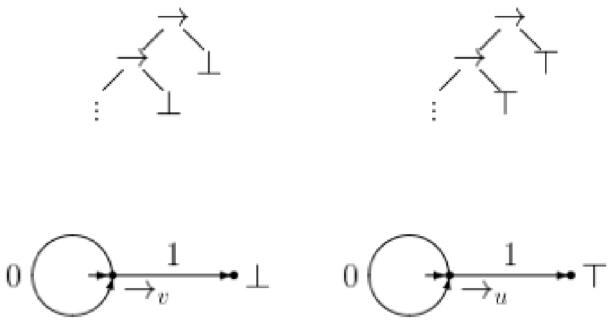
$$\begin{array}{lcl} t(0^n) & = & \to \\ t(0^{2n}1) & = & \top \\ t(0^{2n+1}1) & = & \bot \end{array}$$

- A term over Σ is a partial function: $t : w^* \to \Sigma$
- Define a new partial function $t \downarrow \alpha$:
  - $t \downarrow \alpha(\beta) = t(\alpha\beta)$.
- A term t is *finite* if its domain $D(t)$ is a finite set – finite types
- If $t \downarrow \alpha$ has non empty domain $\Rightarrow$ it is a term and is called the subterm of t at position α.
- t is *regular* if it has only finitely many distinct subterms. That is, $\{t \downarrow \alpha | \alpha \in w^*\}$ is a finite set.
- A term t is regular $\equiv$ it represents a recursive type.

Types as automata

If t is a term then following are equivalent:

- t is regular.
- t is representable by a term automata
- t is describable by a type expression involving μ.



Subtyping

- We want to denote that some types are more informative than other.
- We say $t_1 \leq t_2$ to indicate that every value described by t_1 is also described by t_2.
- That is, if you have a function that needs a value of type t_2, you can give safely pass a value of type t_1.
- t_1 is a subtype of t_2 or t_2 is a super type of t_1.
- Example: C++ and Java.
- $$\text{subsumption}\,\frac{A \vdash e : t \qquad t \leq t'}{A \vdash e : t'}$$

Rules for subtyping

- $$(\mathit{reflexive}) \;\; t \leq t$$
- $$\text{transitive}\,\frac{t_1 \leq t_2 \qquad t_2 \leq t_3}{t_1 \leq t_3}$$
- $$\text{Arrow}\,\frac{t_1 \leq s_1 \qquad s_2 \leq t_2}{s_1 \rightarrow s_2 \leq t_1 \rightarrow t_2}$$
    - The subtype relation is reversed (contravariant) for the argument types.
    - The subtype relation in the result types - covariant.

Special types

- $$(\mathit{Top}) t \leq \top$$
- $\top$ = Java `Object` class.
- $\perp$ = Subtype of all the classes - undefined type.
    - (lambda (x) (zero? x) 4 (error # mesg))
- $t =$ Int $|\perp|\top|t \rightarrow t|v|\mu v.(t \rightarrow t)$

Subtyping algorithm for recursive types

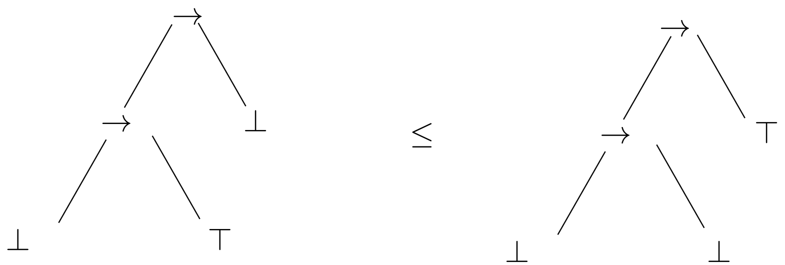
- Roberto M Amadio. and Luca Cardelli. Subtyping recursive types. In ACM Symposium on Principles of Programming Languages, 1990. - self reading.
- Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive sub-typing. In ACM Symposium on Principles of Programming Languages, 1993.

Parity

- The partiy of $\alpha \in \{0,1\}^*$ is even - if α has even number of zeros.
- The partiy of $\alpha \in \{0,1\}^*$ is odd - if α has odd number of zeros.
- Denote parity of α by $\pi\alpha$ = 0 if even, 1 if odd.
- We will definte two orders.
  - co-variant: $\bot \leq_0 \top$
  - contra-variant: $\top \leq_1 \bot$

Type ordering

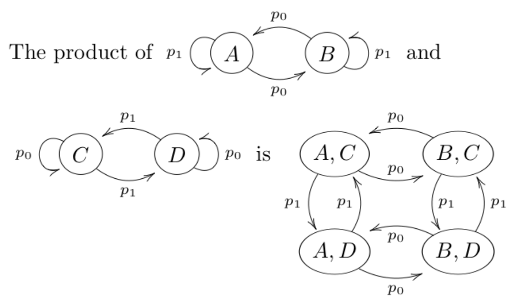
- For two types s, and t, we define $s \leq t$, iff $s(\alpha) \leq_{\pi\alpha} t(\alpha)$ for all $\alpha \in D(s) \cap D(t)$.



- A counter example to $s \leq t$: $\exists$ a path $\alpha \in D(s) \cap D(t)$, where $s(\alpha) \not\leq_{\pi\alpha} t(\alpha)$
  - Two trees are ordered if no common path detects a counter example.
- For finite types, we can compare all the paths (*cost?*) in the tree. For recursive types?

Recap product autoamta

- A prduct automata represents interaction between two finite state machines.



If we start from A, C and after the word w we are in the state A,D we know that w contains an even number of p_0s and odd number of p_1s

Slide from Thierry Coquand @ University of Gothenburg

Modified product automata

- Given two term automata M and N, we will construct a product automata – (non-deterministic?)
  $A = (Q^A, \Sigma, q_0^A, \delta^A, F^A)$
  where
  - $Q^A = Q^M \times Q^N \times \{0,1\}$
  - $\Sigma = \{0,1\}$
  - $q_0^A = (q_0^M, q_0^N, 0)$ – start state of A.
  - $\delta^A : Q^A \times \Sigma \to Q^A$.
    For $b, i \in \Sigma$, $p \in Q^M$, and $q \in Q^N$,
    we have $\delta^A((p,q,b),i) = (\delta^M(p,i), \delta^N(q,i), b \oplus \pi i)$
    ($\oplus$ = xor)
  - Final states
    - Recall: $s \not\leq t$ iff
      $\{\alpha \in D(s) \cap D(t) | s(\alpha) \not\leq_{\pi\alpha} t(\alpha)\}$
    - Goal: create an automata, where final states are denoted by states that will lead to $\not\leq$.

    $F^A = \{(p,q,b) | l^M(p) \not\leq_b l^N(q)\}$ – l gives the label of that node.

Decision procedure for subtyping

Input: Two types s, t.
Output: If $s \leq t$.

1. Construct the term automata for s and t.
2. Construct the product automaton $s \times t$. Size = ?
3. Decide, using depth first search, if the product automaton accepts the nonempty set.
   - Does there exist a path from the start state to some final state?
4. If yes, then $s \not\leq t$. Else $s \leq t$.

Compute the time complexity - $O(n^2)$

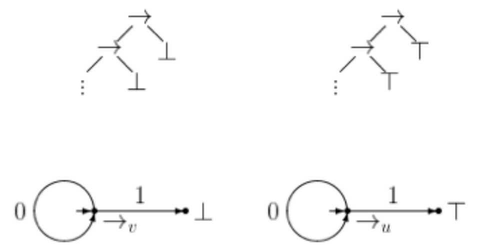
Example 0

- $(\bot \to \top)$ and $(\top \to \bot) \not\leq$
- $((\bot \to \top) \to (\bot)$ and $((\top \to \bot) \to (\bot) \leq$

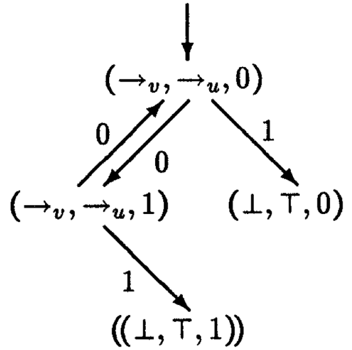
Example 1

- $\mu v.(v \to \bot)$ and $\mu u.(u \to \top)$

Term automata



Product automata

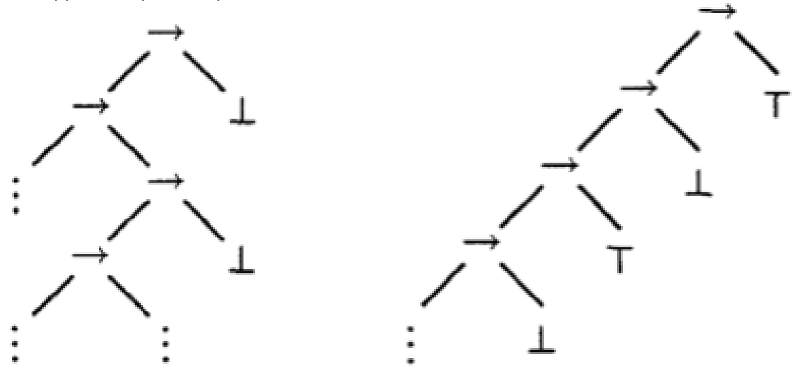


- **Unreachable states**
  $((\to_V, \top, 1))$, $(\to_V, \top, 0)$, $(\bot, \to u, 1)$, $((\bot, \to u, 0))$,
- $\mu v.(v \to \bot) \not\leq \mu u.(u \to \top)$
  Note: *Some of the unreachable states are ((final))*

Example 2

- $\mu u.((u \to u) \to \perp)$ and $\mu v.((v \to \perp) \to \top)$
- **Term automata**

$\mu u.((u \to u) \to \perp)$ $\mu v.((v \to \perp) \to \top)$



- Product automata - derive. Ans: $\leq$.

Type inference

- Goal: Given a program with some types.
- Infer "consistent" types of all the expressions in the program.

First order unification

- Goal: To do type inference
- Given: A set of variables and literals and their possible types.
  - Remember: type = constraint.
- Target: Does the given set of constraints have a solution? And if so, what is the most general solution?
- Unification can be done in linear time: M. S. Paterson and M. N. Wegman, Linear Unification, Journal of Computer and System Sciences, 16:158167, 1978.
- We will instead present a simpler to understand, complex to run algorithm.

Definitions

- We will stick to simple type experssions generated from the grammar:

$$t ::= t \to t | \text{Int} | \alpha$$

where α ranges over type variables.
- Type substitution, example:

$$((\text{Int} \to \alpha) \to \beta)[\alpha := \text{Int}, \beta := (\text{Int} \to \text{Int})] = (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int})$$

$$((\text{Int} \to \alpha) \to \gamma)[\alpha := \text{Int}, \beta := (\text{Int} \to \alpha)] = (\text{Int} \to \text{Int}) \to \gamma$$

- We say given a set of type equations, we say a substituion σ is an *unifier or solution* if for each of the equation of the form $s = t$, $s\sigma = t\sigma$.
- Substituions can be composed:

$$t(\sigma\ o\ \theta) = (t\sigma)\theta$$

- A substitution σ is called a most general solution of an equation set provided that for any other solution θ, there exists a substituon τ such that $\theta = \sigma\ o\ \tau$

Unification algorithm

Input: G: set of type equations (derived from a given program).
Output: Unification σ

1. failure = `false`; σ = $\{\}$.
2. while $G \neq \phi$ and $\neg$ failure do
   1. Choose and remove an equation e from G. Say $e\sigma$ is $(s = t)$.
   2. If s and t are variables, or s and t are both Int then continue.
   3. If $s = s_1 \to s_2$ and $t = t_1 \to t_2$, then $G = G \cup \{s_1 = t_1, s_2 = t_2\}$.
   4. If $(s = \text{Int}$ and t is an arrow type) or vice versa then failure = `true`.
   5. If s is a variable that does not occur in t, then $\sigma = \sigma \ o \ [s := t]$.
   6. If t is a variable that does not occur in s, then $\sigma = \sigma \ o \ [t := s]$.
   7. If $s \neq t$ and either s is a variable that occurs in t or vice versa then failure = `true`.
3. end-while.
4. if (failure = true) then output “Does not type check”. Else o/p σ.

Q: Composability helps?
Q: Cost?

Examples

$$\alpha = \beta \to \text{Int}$$
$$\beta = \text{Int} \to \text{Int}$$

$$\alpha = \text{Int} \to \beta$$
$$\beta = \alpha \to \text{Int}$$

Recap

- Structural subtyping
- Unification algorithm