March 4, 2015

Type inference

1969 1978

Hindley - Milner type inference algorith

A simple language:

Value Language An expression e is either a

variable	x
an application	$e_1\ e_2$
an abstraction	$\lambda x.\ e$
a let	$\text{let } x = e_1 \text{ in } e_2$

Type language A type is either a

monotype τ which is either

- a type variable α
- a type constructor application $D^K\ \tau_1, \tau_2 \ldots \tau_K$
  $\tau_1 \to \tau_2$

or a polytype

- a monotype
- a quantified type $\forall \alpha_1, \alpha_2 \ldots \alpha_n.\ \tau$

$\alpha \to \alpha$

Intuition about monotypes:

- int
- string
- int → int
- α

type variables in monotypes are not like haskell type variables.

Intuition about polytypes:

$\forall \alpha . \alpha \to \alpha$

$\forall \alpha : (\text{Set } \alpha) \to \text{int}$

$[] :: \forall \alpha . [\alpha]$

but we dont allow

$\forall \alpha . \alpha \not\to \forall \alpha . \alpha$

Specialization / Generalization of types

$\forall \alpha . \alpha \to \alpha \sqsubseteq int \to int$

general — specialized

Partial order on types

To specialize a type:

replace some type variables consistently with a monotype and drop the variables from the quantified list.

Specialization Rule

if $\to \tau' = [\alpha_i := \tau_i]\tau$, β_i not free in $\forall \alpha_1 \ldots \alpha_n . \tau$

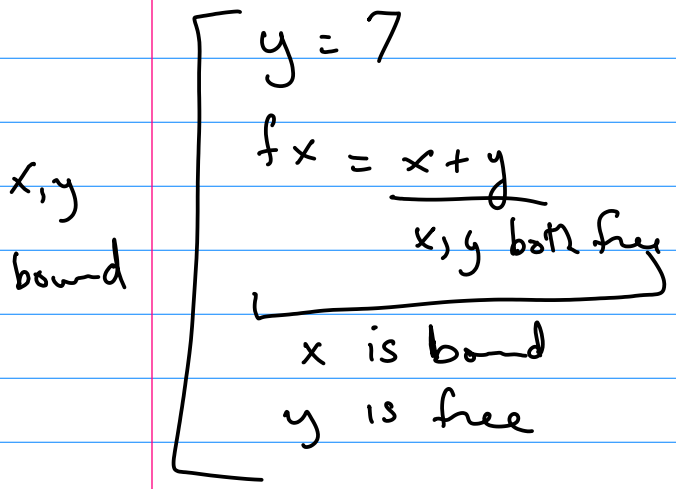
Then $\to \quad \forall \alpha_1 \ldots \alpha_n . \tau \sqsubseteq \forall \beta_1 \ldots \beta_m . \tau'$

a free variable is one

not bound by a $\forall$ quantifier (type language)

not bound by λ or let (value language)

free or bound depends on context



$\lambda x . x$ id function

int → int

(bool → int) → (bool → int)

$\forall x, y . (x \to y) \to (x \to y)$

$\forall \alpha . \alpha \to \alpha$

Goal for type inference: find the most general type of every value expression in the program.

More about inference rules: propositional logic.

$$\frac{a \wedge b}{a} \qquad \frac{a \wedge b}{b \wedge a} \qquad \frac{a}{a \vee b} \qquad \frac{a}{b \vee a} \qquad \frac{a}{\neg\neg a} \qquad \frac{a \vee b, \neg a}{b}$$

? $\dfrac{a \vee b}{b \vee a}$

$a \vee b, \neg a \vdash b$

$a \vee b, \neg a \vdash b \vee a$

Syntax for deduction system for typing involves

Contexts: $\Gamma ::= \varepsilon$ (empty)

$\mid \Gamma, x : \alpha$ (x: value variable, α: type variable) $\quad \cong \{x : \alpha\} \cup \Gamma$

Typing: $\Gamma \vdash e : \sigma$ (e: value exp, σ: polytype)

↑ e can have type σ

Predicates: $\sigma \sqsubseteq \sigma'$

$\alpha \notin free(\Gamma)$

$x : \alpha \in \Gamma$

$free(\Gamma) = \bigcup_{x:\sigma \in \Gamma} free(\sigma)$???

Judgement = Typing

Premise = Judgement | Predicate

Conclusion = Judgement

Rule $\quad \dfrac{\text{Premise} \ldots}{\text{Conclusion}}$ [Name]

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ [\text{Var}]$$

Access

$$\frac{\Gamma \vdash e_0 : \tau \to \tau', \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\, e_1 : \tau'}\ [\text{App}]$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\ [\text{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \sigma, \quad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash (\text{let } x = e_0 \text{ in } e_1) : \tau}\ [\text{Let}]$$

Specialization and generalization rules

$$\frac{\Gamma \vdash e : \sigma', \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \text{ [Inst]}$$

Instantiation? Specialization

$$\frac{\Gamma \vdash e : \sigma, \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall \alpha . \sigma} \text{ [Gen]}$$

Generalization

3/9/15 Union - Find

elements

$n_0, n_1, \ldots n_k \ldots$

build sets of elements s.t. each element is always in exactly one set.

when we start each element is in its own set.

operations: Union $(S1, S2)$

Find (n_i) - returns the identity of the set containing n_i

Simple and obvious: list of lists, each representing a subset.
$O(n^2)$

Data structure: Parent Tree.

Associate a parent pointer w/ each element.

```
for n in elements:
    n.parent = n
```

```
Find(n):
    p = n.parent
    while p ≠ n:
        n = p
        p = n.parent
    return p
```

Union(S_1, S_2):

S_1.parent = S_2

to improve runny time always make the shorter tree be the child of the taller.

- $O(n \log n)$

One more trick: after doing a Find, change the parent pointers of all the nodes on the path to the root to point to the root.

Fact: with this change to Find the running time of the alg. is $O(n(\alpha\, n))$

$(\alpha\, n)$ is the "Inverse Ackermann function"

$(\alpha\ n)$ is the smallest i s.t. $(a\ i) > n$.

where

$$a\ i = a'\ i\ i$$

$$a'\ 0\ n = n+1$$

$$a'\ m\ 0 = a'\ (m-1)\ 1$$

$$a'\ m\ n = a'\ (m-1)\ (a'\ m\ (n-1))$$

Ackermann Function first known of example of a total recursive function that is not primitive recursive.

$$a\ 0 = a'\ 0\ 0 = 1$$

$$a\ 1 = a'\ 1\ 1 = a'\ 0\ (a'\ 1\ 0)$$

$$= a'\ 0\ (a'\ 0\ 1)$$

$$= (a'\ 0\ 1) + 1 = 3$$

$$a\ 2 = 7$$

$$a\ 3 = 61$$

$$a\ 4 = 2^{2^{65536}} - 3$$

Unification : widely used - Theorem proving, type inference, logic programming. Link survey paper for today's lecture.

given two terms made up of

constants - a, b, c ~ specific monotypes
variables - x, y, z - monotype variables
functions - f, g, h - type constructors

function application $(f\ t_1\ t_2 \ldots t_k)$ terms do not contain unapplied functions.

$x \quad a \quad (f\ x)\ (g\ (h\ x\ a))$

A substitution is a function from variables to terms: named w/ small Greek letters

$$x \leftarrow (f\ a) \qquad \{x \leftarrow (f\ a),\ y \leftarrow b\} = \sigma$$
$$y \leftarrow b$$

intuition: the variable is replaced by the term. when the substitution is applied to a term.

$$(\sigma\ '(g\ x\ y)) = g\ (f\ a)\ b$$

now note, because substitutions are functions we can compose them

$\sigma \cdot \theta$: a substitution that first does replacements according to θ and then applies σ to the result.

$\sigma \cdot \theta$ in closed form: do the replacements of σ in θ and then keep the add'l substitutions of σ.

Note as usual composition is associative but not commutative

What is unification?

two terms s, t are unifiable if there is a substitution σ s.t. $(\sigma\ s) = (\sigma\ t)$

if so σ is called a unifier for s and t

$s \equiv f\ x$

$t \equiv f\ (g\ y)$

$\theta = \{ y \leftarrow a,\ x \leftarrow (g\ a) \}$

$(\theta s) \equiv f\ (g\ a)$

$(\theta t) \equiv f\ (g\ a)$

We don't want just any unifier, we want the "most general unifier":

$\hat{\sigma}$

a unifier for s, t such that for any other unifier θ there exists another unifier τ s.t.

$$\tau\sigma = \theta$$

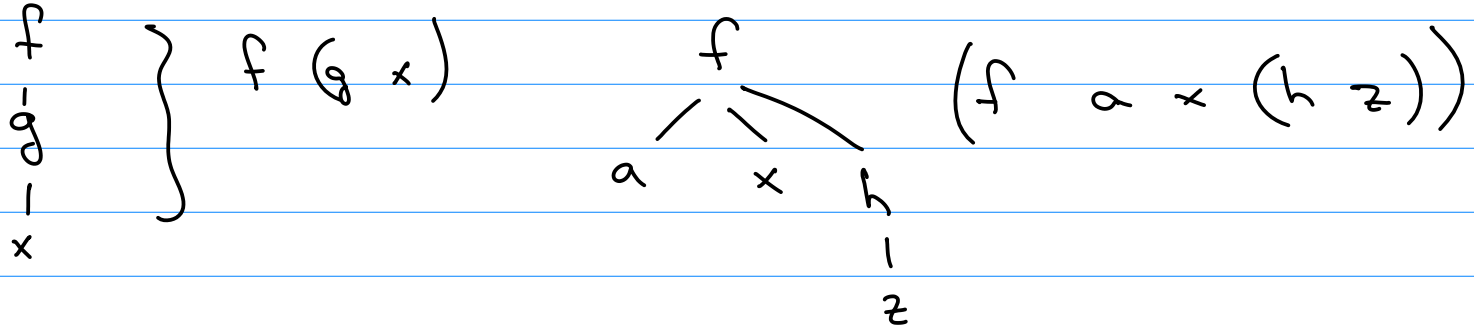
claim $\sigma = \{x \leftarrow g\ y\}$

$\theta = \{y \leftarrow a, x \leftarrow g\ a\}$

Then $\theta = \underbrace{\{y \leftarrow a\}}_{\tau} \cdot \sigma$

$(\sigma\ s) = (f\ (g\ y))$ $(\sigma\ t) = (f\ (g\ y))$ σ is a unifier.

1. Representation of terms for unification alg.



Unify t_1 t_2 =

workList ← [(t_1, t_2)]

for each node z in t_1 and t_2:

z.parent = z ← parent as in union-find.

while workList ≠ []:

(x, y) ← pop workList

u ← Find x

v ← Find y

if u ≠ v:

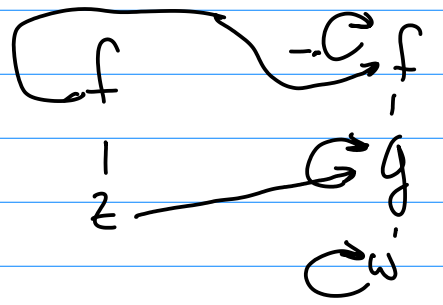
if neither u nor v is a variable and (root symbols don't match or arity is different): return Nothing

else:
$w = \text{Union}(u, v)$ ← implement union so the non-variable is the root.

if neither u, v is a variable:
append zip (u.children, v.children) onto the work list.

our substitution is created by finding each the variables in the resulting forest of union-find trees.

$s = (f\ z) \quad t = (f\ (g\ w))$



$\{z \leftarrow (g\ w)\}$ — the m.g.u.