

Type-Based Analysis

Lecture 19

Comments on Abstract Interpretation

- Why is abstract interpretation either forwards or backwards?
- Answer 1
 - Polynomial to compute in one direction
 - Exponential to compute in the other direction
- Answer 2
 - Abstract functions often implemented as functions
 - Impossible to invert---they're code!

Outline

- A language
 - Lambda calculus
- Types
 - Type checking
 - Type inference
- Applications to program analysis
 - Representation analysis
 - Tagging optimization
 - Alias analysis

The Typed Lambda Calculus

- Lambda calculus
 - But types are assigned to bound variables.
 - Pascal, or C
- Add integers, addition, if-then-else
- Note: Not every expression generated by this grammar is a properly typed term.

$$e = x \mid \lambda x : \tau. e \mid e \ e \mid i \mid e + e \mid \text{if } e \ e \ e$$

Types

- Function types
- Integers
- Type variables
 - Stand for definite, but unknown, types

$$\tau = \alpha \mid \tau \rightarrow \tau \mid \text{int}$$

Function Types

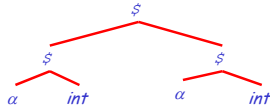
- Intuitively, a type $\tau_1 \rightarrow \tau_2$ stands for the set of functions that map arguments of type τ_1 to results of type τ_2 .
- Placeholder for any other structured datatype
 - Lists
 - Trees
 - Arrays



Types are Trees

- Types are terms
- Any term can be represented by a tree
 - The parse tree of the term
 - Tree representation is important in algorithms

$(\alpha \ \$ \ \text{int}) \ \$ \ \alpha \ \$ \ \text{int}$



Profs. Aiken & Dill CS 357 Lecture 19

7

Examples

- We write $e:t$ for the statement "e has type t."

$\lambda x:\alpha.x:\alpha \rightarrow \alpha$

$\lambda x:\alpha.\lambda y:\beta.x:\alpha \rightarrow \beta \rightarrow \alpha$

$\lambda f:\alpha \rightarrow \beta.\lambda g:\beta \rightarrow \gamma.\lambda x:\alpha.g(f\ x):(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$

$\lambda f:\alpha \rightarrow \beta \rightarrow \gamma.\lambda g:\alpha \rightarrow \beta.\lambda x:\alpha.(f\ x)\ (g\ x):(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

Profs. Aiken & Dill CS 357 Lecture 19

8

Untypable Terms

- Some terms have no valid typing.
 - $\lambda x.x\ x$
 - $\lambda x.\lambda y.(x\ y) + x$
- Focus on first example
 - Types are finite
 - Becomes typable if we allow recursive types
 - Recursive types are possibly infinite, regular trees

Profs. Aiken & Dill CS 357 Lecture 19

9

Type Environments

- To determine whether the types in an expression are correct we perform *type checking*.
- But we need types for free variables, too!
- A *type environment* is a function from variables to types. The syntax of environments is:

$$A = \emptyset \mid A, x : \tau$$

- The meaning is:

$$(A, x : \tau)(y) = \begin{cases} \tau & \text{if } x = y \\ A(y) & \text{if } x \neq y \end{cases}$$

Profs. Aiken & Dill CS 357 Lecture 19

10

Type Checking Rules

- Type checking is done by structural induction.
 - One inference rule for each form
 - Assumptions contain types of free variables
 - A term is *well-typed* if $\triangleright \vdash e : \tau$

$$\frac{A(x) = \tau \quad A \vdash x : \tau}{A \vdash x : \tau} \quad \frac{A, x : \tau \vdash e : \tau'}{A \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \quad \frac{A \vdash e_1 : \tau \quad A \vdash e_2 : \tau'}{A \vdash e_1\ e_2 : \tau'}$$

$$\frac{A \vdash e_1 : \text{int} \quad A \vdash e_2 : \text{int}}{A \vdash i : \text{int}} \quad \frac{A \vdash e_1 : \text{int} \quad A \vdash e_2 : \text{int}}{A \vdash e_1 + e_2 : \text{int}} \quad \frac{A \vdash e_1 : \text{int} \quad A \vdash e_2 : \tau \quad A \vdash e_3 : \tau}{A \vdash \text{if } e_1\ e_2\ e_3 : \tau}$$

Profs. Aiken & Dill CS 357 Lecture 19

11

Example

$$\frac{x : \alpha, y : \beta \vdash x : \alpha}{x : \alpha \vdash \lambda y : \beta. x : \beta \rightarrow \alpha} \\ \frac{}{\emptyset \vdash \lambda x : \alpha. \lambda y : \beta. x : \alpha \rightarrow \beta \rightarrow \alpha}$$

Profs. Aiken & Dill CS 357 Lecture 19

12



Type Checking Algorithm

- There is a simple algorithm for type checking
- Observe that there is only one possible "shape" of the type derivation
 - only one inference rule applies to each form.

$$\frac{\frac{? \vdash x : ?}{? \vdash \lambda y : \beta. x : ?}}{\emptyset \vdash \lambda x : \alpha. \lambda y : \beta. x : ?}$$

Algorithm (Cont.)

- Walk the proof tree from the root to the leaves, generating the correct environments.
- Assumptions are simply gathered from lambda abstractions.

$$\frac{\frac{x : \alpha, y : \beta \vdash x : ?}{x : \alpha \vdash \lambda y : \beta. x : ?}}{\emptyset \vdash \lambda x : \alpha. \lambda y : \beta. x : ?}$$

Algorithm (Cont.)

- In a walk from the leaves to the root, calculate the type of each expression.
- The types are completely determined by the type environment and the types of subexpressions.

$$\frac{\frac{x : \alpha, y : \beta \vdash x : \alpha}{x : \alpha \vdash \lambda y : \beta. x : \beta \rightarrow \alpha}}{\emptyset \vdash \lambda x : \alpha. \lambda y : \beta. x : \alpha \rightarrow \beta \rightarrow \alpha}$$

A Bigger Example

$$\frac{\frac{\frac{x : \alpha \rightarrow \alpha, y : \beta \vdash x : \alpha \rightarrow \alpha}{x : \alpha \rightarrow \alpha \vdash \lambda y : \beta. x : \beta \rightarrow \alpha \rightarrow \alpha} \quad \frac{z : \alpha \vdash z : \alpha}{\emptyset \vdash \lambda z : \alpha. z : \alpha \rightarrow \alpha}}{\emptyset \vdash \lambda x : \alpha \rightarrow \alpha. \lambda y : \beta. x : (\alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \alpha \rightarrow \alpha} \quad \frac{\emptyset \vdash \lambda z : \alpha. z : \alpha \rightarrow \alpha}{\emptyset \vdash (\lambda x : \alpha \rightarrow \alpha. \lambda y : \beta. x) \lambda z : \alpha. z : (\alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \alpha \rightarrow \alpha}}$$

What Do Types Mean?

- **Thm.** If $A \vdash e : \tau$ and $e \xrightarrow{\beta} d$, then $A \vdash d : \tau$
 - Evaluation preserves types.
- This is the basis of a claim that there can be no runtime type errors
 - functions applied to data of the wrong type
 - Adding to a function
 - Using an integer as a function

Type Inference

- The *type erasure* of e is e with all type information removed (i.e., the untyped term).
- Is an untyped term the erasure of some simply typed term? And what are the types?
- This is a *type inference* problem. We must infer, rather than check, the types.



Outline

- We will develop the inference algorithm in the following steps:
 - recast the type rules in an equivalent form
 - show typing in the new rules reduces to a constraint satisfaction problem
 - show the constraint problem is solvable via term unification.
- We will use this outline again.

The Problems

$$\frac{A(x) = \tau \quad A, x : \tau \vdash e : \tau'}{A \vdash x : \tau} \quad \frac{A \vdash e_1 : \tau \rightarrow \tau' \quad A \vdash e_2 : \tau}{A \vdash e_1 e_2 : \tau'}$$

$$\frac{A \vdash e_1 : \text{int} \quad A \vdash e_2 : \text{int}}{A \vdash i : \text{int}} \quad \frac{A \vdash e_1 : \text{int} \quad A \vdash e_2 : \tau \quad A \vdash e_3 : \tau}{A \vdash \text{if } e_1 e_2 e_3 : \tau}$$

- There are three problems in developing an algorithm
 - How do we construct the right type assumptions?
 - How do we ensure types match in applications?
 - How do we ensure types match in if-then-else?

New Rules

$$\frac{A(x) = \alpha_x \quad A, x : \alpha_x \vdash e : \tau \quad \tau_1 = \tau_2 \rightarrow \beta}{A \vdash x : \alpha_x \quad A \vdash \lambda x. e : \alpha_x \rightarrow \tau} \quad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash e_1 e_2 : \beta}$$

$$\frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \tau_1 = \tau_2 = \text{int}}{A \vdash i : \text{int}} \quad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \tau_1 = \text{int} \quad \tau_2 = \tau_3}{A \vdash \text{if } e_1 e_2 e_3 : \tau_2}$$

- Sidestep the problems by introducing explicit unknowns and constraints

New Rules

$$\frac{A(x) = \alpha_x \quad A, x : \alpha_x \vdash e : \tau \quad \tau_1 = \tau_2 \rightarrow \beta}{A \vdash x : \alpha_x \quad A \vdash \lambda x. e : \alpha_x \rightarrow \tau} \quad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash e_1 e_2 : \beta}$$

$$\frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \tau_1 = \tau_2 = \text{int}}{A \vdash i : \text{int}} \quad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \tau_1 = \text{int} \quad \tau_2 = \tau_3}{A \vdash \text{if } e_1 e_2 e_3 : \tau_2}$$

- Type assumption for variable x is a fresh variable α_x

New Rules

$$\frac{A(x) = \alpha_x \quad A, x : \alpha_x \vdash e : \tau \quad \tau_1 = \tau_2 \rightarrow \beta}{A \vdash x : \alpha_x \quad A \vdash \lambda x. e : \alpha_x \rightarrow \tau} \quad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash e_1 e_2 : \beta}$$

$$\frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \tau_1 = \tau_2 = \text{int}}{A \vdash i : \text{int}} \quad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \tau_1 = \text{int} \quad \tau_2 = \tau_3}{A \vdash \text{if } e_1 e_2 e_3 : \tau_2}$$

- Equality conditions represented as side constraints

New Rules

$$\frac{A(x) = \alpha_x \quad A, x : \alpha_x \vdash e : \tau \quad \tau_1 = \tau_2 \rightarrow \beta}{A \vdash x : \alpha_x \quad A \vdash \lambda x. e : \alpha_x \rightarrow \tau} \quad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash e_1 e_2 : \beta}$$

$$\frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \tau_1 = \tau_2 = \text{int}}{A \vdash i : \text{int}} \quad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \tau_1 = \text{int} \quad \tau_2 = \tau_3}{A \vdash \text{if } e_1 e_2 e_3 : \tau_2}$$

- Hypotheses are all arbitrary
 - Can always complete a derivation, pending constraint resolution



Notes

- The introduction of unknowns and constraints works only because the shape of the proof is already known.
 - This tells us where to put the constraints and unknowns.
- The revised rules are trivial to implement, except for handling the constraints.

Solutions of Constraints

- The new rules generate a system of type equations.
- Intuitively, a solution of these equations gives a derivation.
- A solution is a substitution $\text{Vars} \ \$ \ \text{Types}$ such that the equations are satisfied.

Example

$$\begin{aligned}\alpha &= \beta \rightarrow \gamma \\ \alpha &= \gamma \rightarrow \beta \\ \beta &= \text{int}\end{aligned}$$

- A solution is

$$\alpha = \text{int} \rightarrow \text{int}, \beta = \text{int}, \gamma = \text{int}$$

Solving Type Equations

- Term equations are a unification problem.
 - Solvable in near-linear time using a union-find based algorithm.
- No solutions $\alpha = \mathbb{T}[\alpha]$ are permitted
 - The *occurs check*.
 - The check is omitted if we allow infinite types.

Unification

- Close constraints under four rules.
- If no inconsistency or occurs check violation found, system has a solution.
 - $\text{int} = x \ \$ \ y$

$$\begin{aligned}\mathcal{S} \cup \{\alpha = \alpha\} &\Rightarrow \mathcal{S} \\ \mathcal{S} \cup \{\alpha = \tau\} &\Rightarrow \mathcal{S}[\tau/\alpha] \cup \{\alpha \cong \tau\} \\ \mathcal{S} \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\Rightarrow \mathcal{S} \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\ \mathcal{S} \cup \{\text{int} = \text{int}\} &\Rightarrow \mathcal{S}\end{aligned}$$

Syntax

- We distinguish *solved* equations $\alpha \cong \tau$
- Each rule manipulates only unsolved equations.

$$\begin{aligned}\mathcal{S} \cup \{\alpha = \alpha\} &\Rightarrow \mathcal{S} \\ \mathcal{S} \cup \{\alpha = \tau\} &\Rightarrow \mathcal{S}[\tau/\alpha] \cup \{\alpha \cong \tau\} \\ \mathcal{S} \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\Rightarrow \mathcal{S} \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\ \mathcal{S} \cup \{\text{int} = \text{int}\} &\Rightarrow \mathcal{S}\end{aligned}$$



Rules 1 and 4

- Rules 1 and 4 eliminate trivial constraints.
- Rule 1 is applied in preference to rule 2
 - the only such possible conflict

$$\begin{aligned} \mathcal{S} \cup \{\alpha = a\} &\Rightarrow \mathcal{S} \\ \mathcal{S} \cup \{\alpha = \tau\} &\Rightarrow \mathcal{S}[\tau/\alpha] \cup \{\alpha \equiv \tau\} \\ \mathcal{S} \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\Rightarrow \mathcal{S} \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\ \mathcal{S} \cup \{\text{int} = \text{int}\} &\Rightarrow \mathcal{S} \end{aligned}$$

Rule 2

- Rule 2 eliminates a variable from all equations but one (which is marked as solved).
 - Note the variable is eliminated from all unsolved as well as solved equations

$$\begin{aligned} \mathcal{S} \cup \{\alpha = a\} &\Rightarrow \mathcal{S} \\ \mathcal{S} \cup \{\alpha = \tau\} &\Rightarrow \mathcal{S}[\tau/\alpha] \cup \{\alpha \equiv \tau\} \\ \mathcal{S} \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\Rightarrow \mathcal{S} \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\ \mathcal{S} \cup \{\text{int} = \text{int}\} &\Rightarrow \mathcal{S} \end{aligned}$$

Rule 3

- Rule 3 applies structural equality to non-trivial terms.
- Note rule 4 is a degenerate case of rule 3 for a type constructor of arity zero.

$$\begin{aligned} \mathcal{S} \cup \{\alpha = a\} &\Rightarrow \mathcal{S} \\ \mathcal{S} \cup \{\alpha = \tau\} &\Rightarrow \mathcal{S}[\tau/\alpha] \cup \{\alpha \equiv \tau\} \\ \mathcal{S} \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\Rightarrow \mathcal{S} \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\ \mathcal{S} \cup \{\text{int} = \text{int}\} &\Rightarrow \mathcal{S} \end{aligned}$$

Correctness

- Each rule preserves the set of solutions.
 - Rules 1 and 4 eliminate trivial constraints.
 - Rule 2 substitutes equals for equals.
 - Rule 3 is the definition of equality on function types.

$$\begin{aligned} \mathcal{S} \cup \{\alpha = a\} &\Rightarrow \mathcal{S} \\ \mathcal{S} \cup \{\alpha = \tau\} &\Rightarrow \mathcal{S}[\tau/\alpha] \cup \{\alpha \equiv \tau\} \\ \mathcal{S} \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\Rightarrow \mathcal{S} \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\ \mathcal{S} \cup \{\text{int} = \text{int}\} &\Rightarrow \mathcal{S} \end{aligned}$$

Termination

- Rules 1 and 4 reduce the number of equations.
- Rule 2 reduces the number of variables in unsolved equations.
- Rule 3 decreases the height of terms.

$$\begin{aligned} \mathcal{S} \cup \{\alpha = a\} &\Rightarrow \mathcal{S} \\ \mathcal{S} \cup \{\alpha = \tau\} &\Rightarrow \mathcal{S}[\tau/\alpha] \cup \{\alpha \equiv \tau\} \\ \mathcal{S} \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\Rightarrow \mathcal{S} \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\ \mathcal{S} \cup \{\text{int} = \text{int}\} &\Rightarrow \mathcal{S} \end{aligned}$$

Termination (Cont.)

- Rules 1, 3, and 4 always terminate
 - because terms must eventually be reduced to height 0.
- Eventually rule 2 is applied, reducing the number of variables.

$$\begin{aligned} \mathcal{S} \cup \{\alpha = a\} &\Rightarrow \mathcal{S} \\ \mathcal{S} \cup \{\alpha = \tau\} &\Rightarrow \mathcal{S}[\tau/\alpha] \cup \{\alpha \equiv \tau\} \\ \mathcal{S} \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\Rightarrow \mathcal{S} \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\ \mathcal{S} \cup \{\text{int} = \text{int}\} &\Rightarrow \mathcal{S} \end{aligned}$$



A Nitpick

- We really need one more operation.
- $\tau = \alpha$ should be flipped to $\alpha = \tau$ if τ is not a variable.
 - Needed to ensure rule 2 applies whenever possible.
 - We just assume equations are maintained in this "normal form".

Solutions

- The final system is a solution.
 - There is one equation $\alpha \cong \tau$ for each variable.
 - This is a substitution with all the solutions of the original system
- Must also perform occurs check to guarantee there are no recursive constraints.

Example

rewrites

$$\begin{array}{l} \alpha = \beta \rightarrow \gamma, \alpha = \gamma \rightarrow \beta, \beta = \text{int} \\ \hline \alpha = \text{int} \rightarrow \gamma, \alpha = \gamma \rightarrow \text{int}, \beta \cong \text{int} \\ \hline \gamma \rightarrow \text{int} = \text{int} \rightarrow \gamma, \alpha \cong \gamma \rightarrow \text{int}, \beta \cong \text{int} \\ \hline \gamma = \text{int}, \text{int} = \gamma, \alpha \cong \gamma \rightarrow \text{int}, \beta \cong \text{int} \\ \hline \text{int} = \text{int}, \gamma \cong \text{int}, \alpha \cong \text{int} \rightarrow \text{int}, \beta \cong \text{int} \\ \hline \gamma \cong \text{int}, \alpha \cong \text{int} \rightarrow \text{int}, \beta \cong \text{int} \end{array}$$

An Example of Failure

$$\begin{array}{l} \alpha = \beta \rightarrow \gamma, \alpha = \gamma \rightarrow (\beta \rightarrow \beta), \beta = \text{int} \\ \hline \alpha = \text{int} \rightarrow \gamma, \alpha = \gamma \rightarrow (\text{int} \rightarrow \text{int}), \beta \cong \text{int} \\ \hline \gamma \rightarrow (\text{int} \rightarrow \text{int}) = \text{int} \rightarrow \gamma, \alpha \cong \gamma \rightarrow \text{int}, \beta \cong \text{int} \\ \hline \gamma = \text{int}, \text{int} \rightarrow \text{int} = \gamma, \alpha \cong \gamma \rightarrow \text{int}, \beta \cong \text{int} \\ \hline \text{nt} \rightarrow \text{int} = \text{int}, \gamma \cong \text{int} \rightarrow \text{int}, \alpha \cong \text{int} \rightarrow \text{int}, \beta \cong \text{int} \end{array}$$

Notes

- The algorithm produces the *most general unifier* of the equations.
 - All solutions are preserved.
- Less general solutions are all substitution instances of the most general solution.

An Efficient Algorithm

- The algorithm we have sketched is polynomial, but not very efficient.
- The repeated substitutions on types is slow.
- Idea: Maintain equivalence classes of types directly.



Union/Find

- Consider sets in which one element is the designated *representative*.
 - If int or $\$$ is in a set, then it is the representative
 - o.w. the representative is arbitrary.
- Two operations
 - $\text{Union}(s,t)$ union two sets together
 - $\text{Find}(s)$ return the representative of set s
- Equal types will be put in the same set.

Algorithm

```

fun unify(m,n) =
  let s = find(m), t = find(n) in
    if s = t then
      true
    elseif s = s1 → s2 and t = t1 → t2 then
      { union(s,t); unify(s1,t1) && unify(s2,t2) }
    elseif s or t is a variable then
      { union(s,t); true }
    else false -- one is → and one is int
  end
    
```

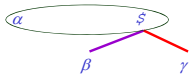
Rules 1 and 4

Rule 3

Rule 2

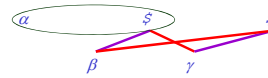
Example

$\alpha = \beta \$ \gamma \quad \alpha = \gamma \$ \beta \quad \beta = \text{int}$



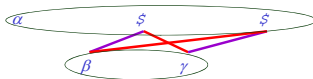
Example

$\alpha = \beta \$ \gamma \quad \alpha = \gamma \$ \beta \quad \beta = \text{int}$



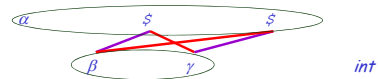
Example

$\alpha = \beta \$ \gamma \quad \alpha = \gamma \$ \beta \quad \beta = \text{int}$



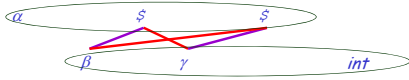
Example

$\alpha = \beta \$ \gamma \quad \alpha = \gamma \$ \beta \quad \beta = \text{int}$



Example

$\alpha = \beta \$ \gamma$ $\alpha = \gamma \$ \beta$ $\beta = \text{int}$



Notes

- Any sequence of *union* and *find* operations can be made to run in nearly linear time (amortized).
- The constants are very small, giving excellent performance in practice.

Applications

Representation Analysis

- Which values in a program must have the same representation?
 - Not all values of a type need be represented identically
 - Shows abstraction boundaries
- Which values must have the same representation?
 - Those that are used "together"

The Idea

- Old type language

$\tau = \alpha \mid \tau \rightarrow \tau \mid \text{int}$

- New type language

- Every type is a pair: old type x variable

$\tau = [\alpha, \beta] \mid [\tau \rightarrow \tau, \beta] \mid [\text{int}, \beta]$

Type Inference Rules

$$\frac{A(x) = [\alpha_x, \beta_x] \quad A, x : [\alpha_x, \beta_x] \vdash e : \tau \quad \tau_1 = \tau_2 \rightarrow [\alpha, \beta]}{A \vdash x : [\alpha_x, \beta_x] \quad A \vdash \lambda x. e : [[\alpha_x, \beta_x] \rightarrow \tau, \gamma] \quad A \vdash e_1 e_2 : [\alpha, \beta]}$$

$$\frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \tau_1 = \tau_2 = [\text{int}, \beta] \quad \tau_1 = [\text{int}, \beta] \quad \tau_2 = \tau_3}{A \vdash i : [\text{int}, \beta] \quad A \vdash e_1 + e_2 : [\text{int}, \beta] \quad A \vdash \text{if } e_1 e_2 e_3 : \tau_2}$$



Example

- A lambda term:

$\lambda x.\lambda y.\lambda z.\lambda w.\text{if } (x + y) (z + 1) w$

- Equivalence classes:

$\lambda x.\lambda y.\lambda z.\lambda w.\text{if } (x + y) (z + 1) w$

Uses

- "Re-engineering"
 - Make some values more abstract
- Find bugs
 - Every equivalence class with a *malloc* should have a *free*
- Implemented for C in a tool Lackwit
 - O'Callahan & Jackson

Dynamic Tag Optimization

- Untyped languages need runtime tags
 - To do runtime type checking
 - E.g., Lisp, Scheme
- Consider an untyped version of our language:
 $e = x \mid \lambda x.e \mid e \ e \mid i \mid e + e \mid \text{if } e \ e \ e$
- Every value carries a tag
 - For us, just 1 bit "function" or "integer"

Term Completion

- View lambda terms as "incomplete"
 - Still need the tagging/tag checking operations
 - $T!$ Tags a value as having type T
 - Every operation that constructs a T must invoke $T!$
 - $T?$ Checks if a value has the tag for type T
 - Every operation that expects to use a T must invoke $T?$
- Example
 - $\lambda f.\lambda x. f (x + 1)$
 - $\text{fun! } \lambda f. (\text{fun! } \lambda x. (\text{fun? } f) (\text{int! } ((\text{int? } x) + (\text{int? } (\text{int! } 1))))$

Tagging Optimization

Optimization problem: remove pairs of tag/untag operations without changing program semantics

$\text{fun! } \lambda f. (\text{fun! } \lambda x. (\text{fun? } f) (\text{int! } ((\text{int? } x) + (\text{int? } (\text{int! } 1))))$

- $\text{fun! } \lambda f. (\text{fun! } \lambda x. (\text{fun? } f) (\text{int! } ((\text{int? } x) + 1)))$

Coercions

- The tagging/untagging operations are *coercions*
 - Functions that change the type
 - But change it to what?
 - Introduce type *dynamic* τ to indicate tagged values
- New types:

$\tau = \alpha \mid \tau \rightarrow \tau \mid \text{int} \mid \tau \mid \mu\beta.\tau$



Coercion Signatures

- With type dynamic, we can give signatures to the coercions:

```
int!: int $ A
int?: A $ int
func!: (A $ A) $ A
func?: A $ (A $ A)
noop: τ $ #
```

- Problem: Decide whether to insert proper coercions or `noop`.

Type Ordering and Constraints

- Types are related by tagging operations:

```
int ≤ A
A $ A # A
τ ≤ τ
```

- Now the choice of a proper coercion or `noop` can be captured by a constraint:

$int \leq \tau$

- Says τ is either `A` or `int`

Type Inference Rules

$$\frac{A(x) = \alpha_x \quad A \vdash x : \alpha_x}{A \vdash x : \alpha_x} \quad \frac{A, x : \alpha_x \vdash e : \tau_1 \quad \alpha_x \rightarrow \tau_1 \leq \tau_2}{A \vdash \lambda x. e : \tau_2} \quad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad \tau_2 \rightarrow \beta \leq \tau_1}{A \vdash e_1 e_2 : \beta}$$

$$\frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \text{int} \leq \tau \quad \text{int} \leq \tau_1 \quad \text{int} \leq \tau_2 \quad \text{int} \leq \tau_3}{A \vdash i : \tau} \quad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_3 \quad \text{int} \leq \tau_1 \quad \tau_2 = \tau_3}{A \vdash \text{if } e_1 e_2 e_3 : \tau_2}$$

Constraint Resolution Rules

$$\begin{aligned} S \cup \{\alpha = \alpha\} &\Rightarrow S \\ S \cup \{\alpha = \tau\} &\Rightarrow S[\tau/\alpha] \cup \{\alpha \equiv \beta\} \\ S \cup \{\alpha_1 \rightarrow \alpha_2 \leq \gamma, \text{int} \leq \gamma\} &\Rightarrow S \cup \{\alpha_1 = \alpha_2 = \gamma = \tau\} \\ S \cup \{\alpha_1 \rightarrow \alpha_2 \leq \gamma, \beta_1 \rightarrow \beta_2 \leq \gamma\} &\Rightarrow S \cup \{\alpha_1 \rightarrow \alpha_2 \leq \gamma, \alpha_1 = \beta_1, \alpha_2 = \beta_2\} \end{aligned}$$

- Note: Arguments of $\$$ and rhs of inequality constraints are always variables

Complexity

- Inequality constraints are generated only by inference rules
 - No new ones are ever added by resolution
- All constraint resolution is of equality constraints
 - Runs at the speed of unification
- Solution of the constraints shows where to insert coercions

Alias Analysis

- In languages with side effects, want to know which locations may have *aliases*
 - More than one "name"
 - More than one pointer to them
- E.g.,


```
Y = &Z
X = Y
*X = 3 /* changes the value of *Y */
```



The Types

- Deal just with pointers and atomic data

$$\tau = \alpha \mid \text{ref}(\tau) \mid \perp$$

A Type Rule

- Consider a C assignment $x = y$
- Intuition: x points to whatever y points to

$$\frac{\begin{array}{l} A \vdash x:\text{ref}(\alpha) \\ A \vdash y:\text{ref}(\beta) \\ \beta = \alpha \end{array}}{A \vdash x=y \text{ :.}}$$

A Problem

- X and Y are always references
 - They're variables
- But what their contents may be atomic:
 - $A = 4$
 - $X = A$
 - $Y = A$
- Now x and y are inferred to always "point" to the same thing
 - But it is obvious there are no pointers here

Type Ordering

- Define an ordering on types:

$$\tau_1 \leq \tau_2 \quad / \quad (\tau_1 = \text{B} \ \& \ \tau_1 = \tau_2)$$

- Change the inference rule:

$$\frac{\begin{array}{l} A \vdash x:\text{ref}(\alpha) \\ A \vdash y:\text{ref}(\beta) \\ \beta \leq \alpha \end{array}}{A \vdash x=y \text{ :.}}$$

Example Inference Rules

$$\frac{\begin{array}{l} A \vdash x:\text{ref}(\alpha) \\ A \vdash y:\text{ref}(\beta) \\ \beta \leq \alpha \end{array}}{A \vdash x=y \text{ :.}}$$

$$\frac{\begin{array}{l} A \vdash x:\text{ref}(\alpha) \\ \alpha = \tau \end{array}}{A \vdash x=\&y \text{ :.}}$$

$$\frac{\begin{array}{l} A \vdash x:\text{ref}(\alpha) \\ A \vdash y:\text{ref}(\text{ref}(\beta)) \\ \beta \leq \alpha \end{array}}{A \vdash x=*y \text{ :.}}$$

$$\frac{\begin{array}{l} A \vdash x:\text{ref}(\text{ref}(\alpha)) \\ A \vdash y:\text{ref}(\beta) \\ \beta \leq \alpha \end{array}}{A \vdash *x=y \text{ :.}}$$

Constraint Resolution Rules

$$\begin{array}{ll} S \cup \{\alpha = \alpha\} & \Rightarrow S \\ S \cup \{\alpha = \tau\} & \Rightarrow S[\tau/\alpha] \cup \{\alpha \equiv \tau\} \\ S \cup \{\text{ref}(\tau_1) = \text{ref}(\tau_2)\} & \Rightarrow S \cup \{\tau_1 = \tau_2\} \\ S \cup \{\alpha \leq \tau_1, \alpha = \text{ref}(\tau_2)\} & \Rightarrow S \cup \{\alpha = \tau_1, \alpha = \text{ref}(\tau_2)\} \end{array}$$



Implementation

- No new inequality constraints are generated by resolution
- Keep a list of pending equality constraints for each variable α
 - These constraints "fire" when α is unified with a *ref*
 - More generally, unified with a constructor

Context Sensitivity: Polymorphic Types

- Add a new class of types called *type schemes*:

$$\sigma = \forall \alpha. \sigma \mid \tau$$

- Example: A polymorphic identity function

$$\forall \alpha. \alpha \rightarrow \alpha$$

- Note: All quantifiers are at top level.

A Useful Lemma

$$A \vdash e : \tau \Rightarrow A[\tau'/\alpha] \vdash e : \tau[\tau'/\alpha]$$

- A variable in a typing proof can be instantiated to something more specific and the proof still works.
- Proof: Replace α by τ' in derivation for e . Show by cases the derivation is still correct.

The Key Idea

$$\frac{A \vdash e : \tau}{\alpha \text{ not free in } A} \quad A \vdash e : \forall \alpha. \tau$$

- This is called *generalization*.

Instantiation

- Polymorphic assumptions can be used as usual.
- But we still need to turn a polymorphic type into a monomorphic type for the other type rules to work.

$$\frac{A \vdash e : \forall \alpha. \sigma}{A \vdash e : \sigma[\tau'/\alpha]}$$

Where is Type Inference Strong?

- Handles data structures smoothly
- Works in infinite domains
 - Set of types is unlimited
- No forwards/backwards distinction
- Type polymorphism good fit for context sensitivity
 - Lexically based
 - Less sensitive to program edits than call strings



Where is Type Inference Weak?

- No flow sensitivity
 - Equality-based analysis only gets equivalence classes
 - "backflow" problem
- Context-sensitive analyses don't always scale
 - Type polymorphism can lead to exponential blowup in constraints