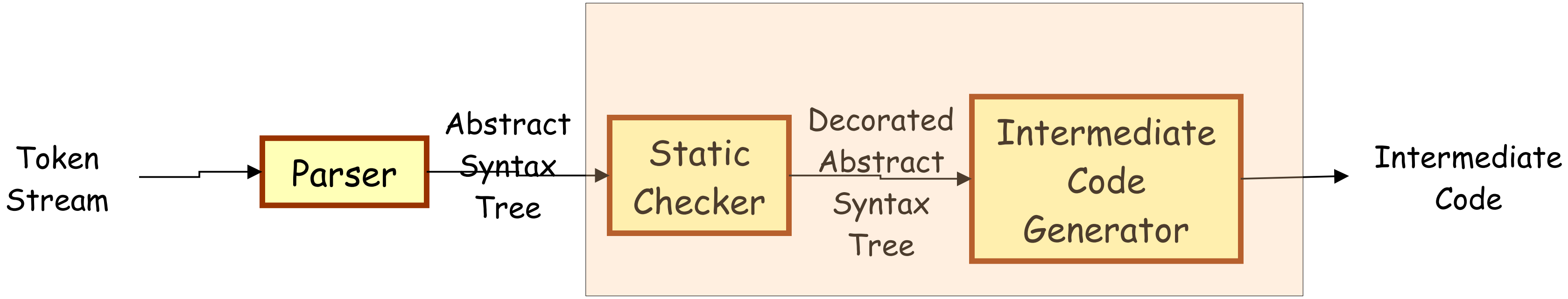
Compiler Design

Type Checking

Winter 2010

Static Checking



- Static (Semantic) Checks
  - Type checks: operator applied to incompatible operands?
  - Flow of control checks: break (outside while?)
  - Uniqueness checks: labels in case statements
  - Name related checks: same name?

Type Checking

- Problem: Verify that a type of a construct matches that expected by its context.
- Examples:
    - mod requires integer operands (PASCAL)
    - * (dereferencing) – applied to a pointer
    - a[i] – indexing applied to an array
    - f(a1, a2, …, an) – function applied to correct arguments.
- Information gathered by a type checker:
    - Needed during code generation.

Type Systems

- A collection of rules for assigning type expressions to the various parts of a program.
- Based on: Syntactic constructs, notion of a type.
- Example: If both operators of "+", "-", "*" are of type integer then so is the result.
- Type Checker: An implementation of a type system.
  - Syntax Directed.
- Sound Type System: eliminates the need for checking type errors during run time.

Type Expressions

- Implicit Assumptions:
  - Each program has a type
    - Expressions
    - Statements
  - Types have a structure

Basic Types

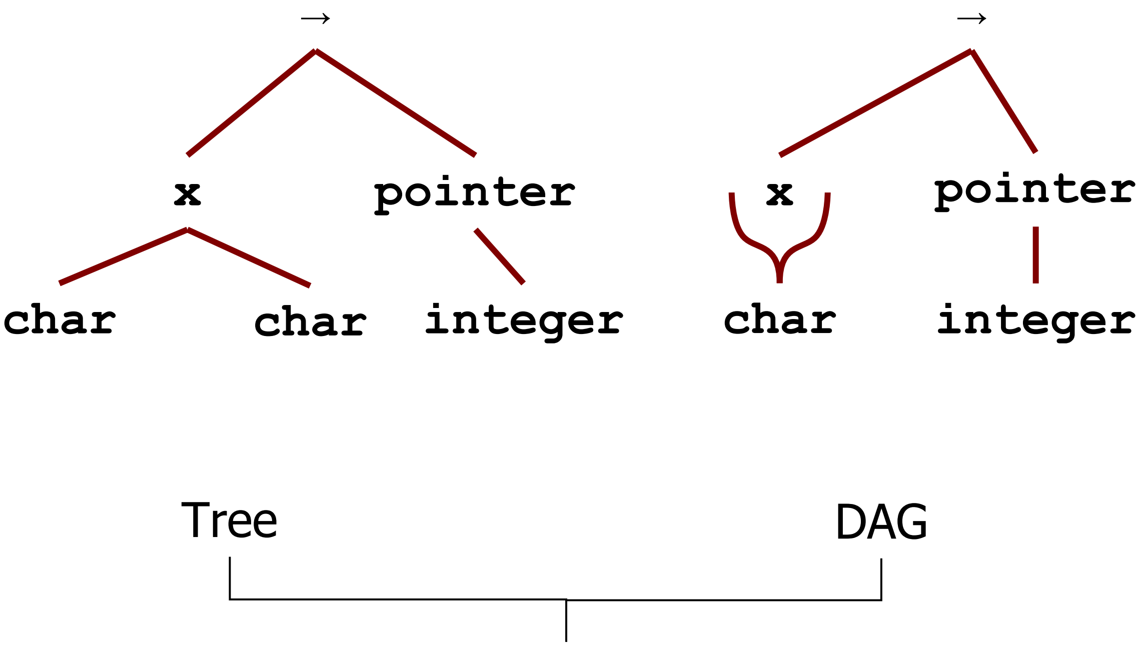
Boolean
Real
Enumerations
Void
Variables

Character
integer
Sub-ranges
Error
Names

Type Constructors

Arrays (strings)
Records
Sets
Pointers
Functions

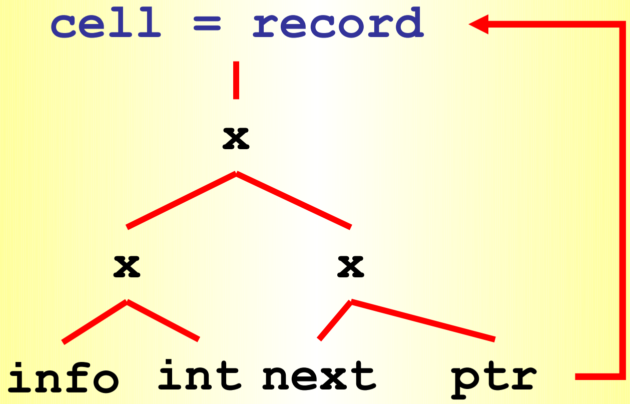
Representation of Type Expressions



Tree

DAG

(char x char)→ pointer (integer)



```
struct cell {
        int info;
        struct cell * next;
};
```

Type Expressions Grammar

Type → int | float | char | …
| void
| error
| name
| variable

Basic Types

| array(size, Type)
| record((name, Type)*)
| pointer(Type)
| tuple((Type)*)
| fcn(Type, Type) (Type → Type)

Structured Types

A Simple Typed Language

Program → Declaration; Statement

Declaration → Declaration; Declaration
| id: Type

Statement → Statement; Statement
| id := Expression
| if Expression then Statement
| while Expression do Statement

Expression → literal | num | id
| Expression mod Expression
| E[E] | E ↑ | E (E)

Type Checking Expressions

E → int_const { E.type = int }

E → float_const { E.type = float }

E → id { E.type = sym_lookup(id.entry, type) }

$E \rightarrow E_1 + E_2$ {E.type = if E_1.type $\notin$ {int, float} |

E_2.type $\notin$ {int, float})

then error

else if E_1.type == E_2.type == int

then int

else float }

Type Checking Expressions

$E \rightarrow E_1\ [E_2]$ {$E.type$ = if $E_1.type = array(S, T) \wedge E_2.type = int$ then T else error}

$E \rightarrow {*}E_1$ {$E.type$ = if $E_1.type = pointer(T)$ then T else error}

$E \rightarrow \&E_1$ {$E.type = pointer(E_1.type)$}

$E \rightarrow E_1(E_2)$ {$E.type$ = if $(E_1.type = fcn(S, T) \wedge E_2.type = S$, then T else error}

$E \rightarrow (E_1, E_2)$ {$E.type = tuple(E_1.type, E_2.type)$}

Type Checking Statements

$S \rightarrow$ id := E {S.type := if id.type = E.type then void else error}

$S \rightarrow$ if E then S_1 {S.type := if E.type = boolean then S1.type else error}

$S \rightarrow$ while E do S_1 {S.type := if E.type = boolean then S_1.type}

$S \rightarrow S_1; S_2$ {S.type := if S_1.type = void ∧ S_2.type = void then void else error}

Equivalence of Type Expressions

Problem: When in E_1.type = E_2.type?

- We need a precise definition for type equivalence
- Interaction between type equivalence and type representation

Example:

```
type vector = array [1..10] of real
type weight = array [1..10] of real
var x, y: vector; z: weight
```

Name Equivalence: When they have the same name.

- x, y have the same type; z has a different type.

Structural Equivalence: When they have the same structure.

- x, y, z have the same type.

Structural Equivalence

- Definition: by Induction
  - Same basic type (basis)
  - Same constructor applied to SE Type (induction step)
  - Same DAG Representation
- In Practice: modifications are needed
  - Do not include array bounds – when they are passed as parameters
  - Other applied representations (More compact)
- Can be applied to: Tree/ DAG
  - Does not check for cycles
  - Later improve it.

Algorithm Testing Structural Equivalence

```
function sequiv(s, t): boolean
{   if (s ∧ t are of the same basic type) return true;
     if (s = array($s_1$, $s_2$) ∧ t = array($t_1$, $t_2$))
          return sequiv($s_1$, $t_1$) ∧ sequiv($s_2$, $t_2$);
     if (s = tuple($s_1$, $s_2$) ∧ t = tuple($t_1$, $t_2$))
          return sequiv($s_1$, $t_1$) ∧ sequiv($s_2$, $t_2$);
     if (s = fcn($s_1$, $s_2$) ∧ t = fcn($t_1$, $t_2$))
          return sequiv($s_1$, $t_1$) ∧ sequiv($s_2$, $t_2$);
     if (s = pointer($s_1$) ∧ t = pointer($t_1$))
          return sequiv($s_1$, $t_1$);
}
```

Recursive Types

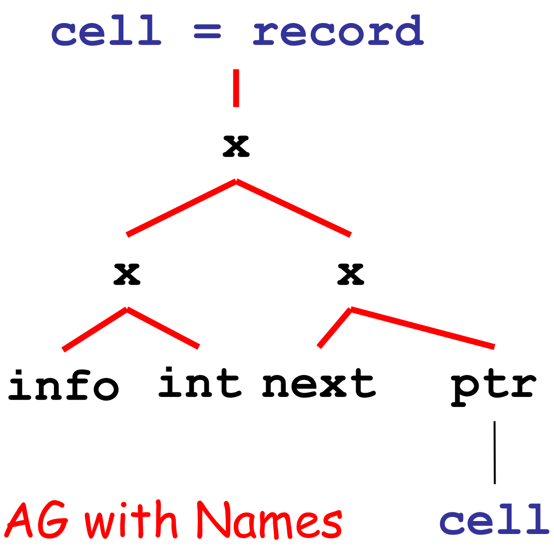
Where: Linked Lists, Trees, etc.

How: records containing pointers to similar records

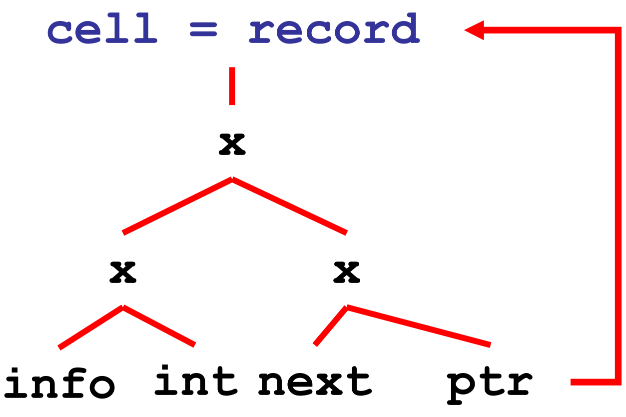
Example: type link = ↑ cell;

cell = record info: int; next = link end

Representation:


cell = record
x
x
x
info
int
next
ptr
cell

DAG with Names


cell = record
x
x
x
info
int
next
ptr

Substituting names out (cycles)

Recursive Types in C

- C Policy: avoid cycles in type graphs by:
  - Using structural equivalence for all types
  - Except for records → name equivalence
- Example:
  - `struct cell {int info; struct cell * next;}`
- Name use: name cell becomes part of the type of the record.
  - Use the acyclic representation
  - Names declared before use – except for pointers to records.
  - Cycles – potential due to pointers in records
  - Testing for structural equivalence stops when a record constructor is reached ~ same named record type?

Overloading Functions & Operators

- Overloaded Symbol: one that has different meanings depending on its context
- Example: Addition operator +
- Resolving (operator identification): overloading is resolved when a unique meaning is determined.
- Context: it is not always possible to resolve overloading by looking only the arguments of a function
    - Set of possible types
    - Context (inherited attribute) necessary

Overloading Example

```
function "*" (i, j: integer) return complex;
function "*" (x, y: complex) return complex;
* Has the following types:
   fcn(tuple(integer, integer), integer)
   fcn(tuple(integer, integer), complex)
   fcn(tuple(complex, complex), complex)
int i, j;
k = i * j;
```

Narrowing Types

$E' \rightarrow E$ {E'.types = E. types

E.unique = if E'.types = {t} then t else error}

$E \rightarrow$ id {E.types = lookup(id.entry)}

$E \rightarrow E_1(E_2)$ {E.types = {s' | $\exists$ s $\in$ E_2.types and

s$\rightarrow$s' $\in$ E_1.types}

t = E.unique

S = {s | s $\in$ E_2.types and S$\rightarrow$t $\in$ E_1.types}

E_2.unique = if S ={s} then S else error

E_1.unique = if S = {s} then S$\rightarrow$t else error

Polymorphic Functions

- **Defn**: a piece of code (functions, operators) that can be executed with arguments of different types.
- **Examples**: Built in Operator indexing arrays, pointer manipulation
- **Why use them**: facilitate manipulation of data structures regardless of types.
- **Example ML**:

```
fun length(lptr) = if null (lptr) then 0
                               else length(+l(lptr)) + 1
```

A Language for Polymorphic Functions

P → D ; E

D → D ; D | id : Q

Q → ∀ α. Q | T

T → fcn (T, T) | tuple (T, T)

| unary (T) | (T)

| basic

| α

E → E (E) | E, E | id

Type Variables

- Why: variables representing type expressions allow us to talk about unknown types.
  - Use Greek alphabets α, β, γ …
- Application: check consistent usage of identifiers in a language that does not require identifiers to be declared before usage.
  - A type variable represents the type of an undeclared identifier.
- Type Inference Problem: Determine the type of a language constant from the way it is used.
  - We have to deal with expressions containing variables.

Examples of Type Inference

```
Type link ↑ cell;
Procedure mlist (lptr: link; procedure p);
{ while lptr <> null
    { p(lptr); lptr := lptr↑ .next}}
```

Hence: p: link → void

```
Function deref (p)
{ return p ↑; }
```

P: β, β = pointer(α)

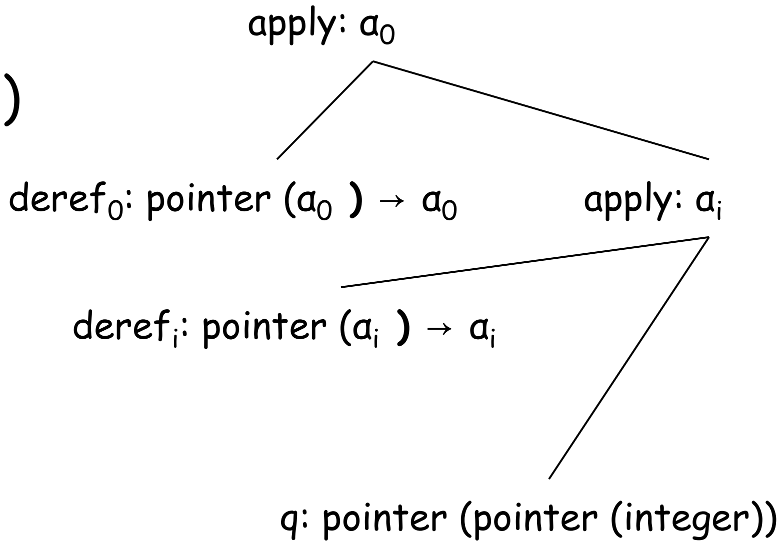
Hence deref: $\forall$ α. pointer(α) → α

Program in Polymorphic Language

deref: $\forall\ \alpha.\ \text{pointer}(\alpha) \rightarrow \alpha$

q: pointer (pointer (integer))

deref (deref((q))



apply: α_0

deref$_0$: pointer (α_0) → α_0

apply: α_i

deref$_i$: pointer (α_i) → α_i

q: pointer (pointer (integer))

Notation:

→ fcn

x tuple

Subsripts i and o distinguish between the inner and outer occurrences of deref, respectively.

Type Checking Polymorphic Functions

- Distinct occurrences of a p.f. in the same expression need not have arguments of the same type.
  - deref (deref (q))
  - Replace α with fresh variable and remove $\forall$ (α_i, α_o)

- The notion of type equivalence changes in the presence of variables.
  - Use unification: check if s and t can be made structurally equivalent by replacing type vars by the type expression.

- We need a mechanism for recording the effect of unifying two expressions.
  - A type variable may occur in several type expressions.

Substitutions and Unification

- Substitution S: a mapping from type variables to type expressions.

```
Function aplly (t: type Expr, S: Substitution): type Expr
  { if (t is a basic type) return t;
    if (t is a variable) return S(t);   -- check if $t \notin S$
    if (t is $t_1 \to t_2$)  return  (apply ($t_1$) → apply ($t_2$));     }
```

- Instance: $S(t)$ is an instance of t written $S(t) < t$.
    - Examples: pointer (integer) < pointer (α) , int → real ≠ $\alpha \to \alpha$
- Unify: $t_1 \approx t_2$ if $\exists$ S. $S(t_1) = S(t_2)$
- Most General Unifier S: A substitution S:
    - $S(t_1) = S(t_2)$
    - $\forall S'. S'(t_1) = S'(t_2) \Rightarrow \forall t. S'(t) < S(t)$.

Polymorphic Type checking Translation Scheme

$E \rightarrow E_1 (E_2)$	{ p := mkleaf(newtypevar); unify (E_1.type, mknode('→', E_2.type, p); E.type = p}
$E \rightarrow E_1, E_2$	{E.type := mknode('x', E_1.type, E_2.type); }
$E \rightarrow id$	{ E.type := fresh (id.type) }

fresh (t): replaces bound variables in t by fresh variables. Returns pointer to a node representing result type.

fresh($\forall$ α.pointer(α) → α) = pointer(α_1) → α_1.

unify (m, n): unifies expressions represented by m and n.

- Side-effect: keep track of substitution
- Fail-to-unify: abort type checking.

PType Checking Example

Given: derefo (derefi (q))

q = pointer (pointer (int))

Bottom Up: fresh (∀α. Pointer(α) → α)