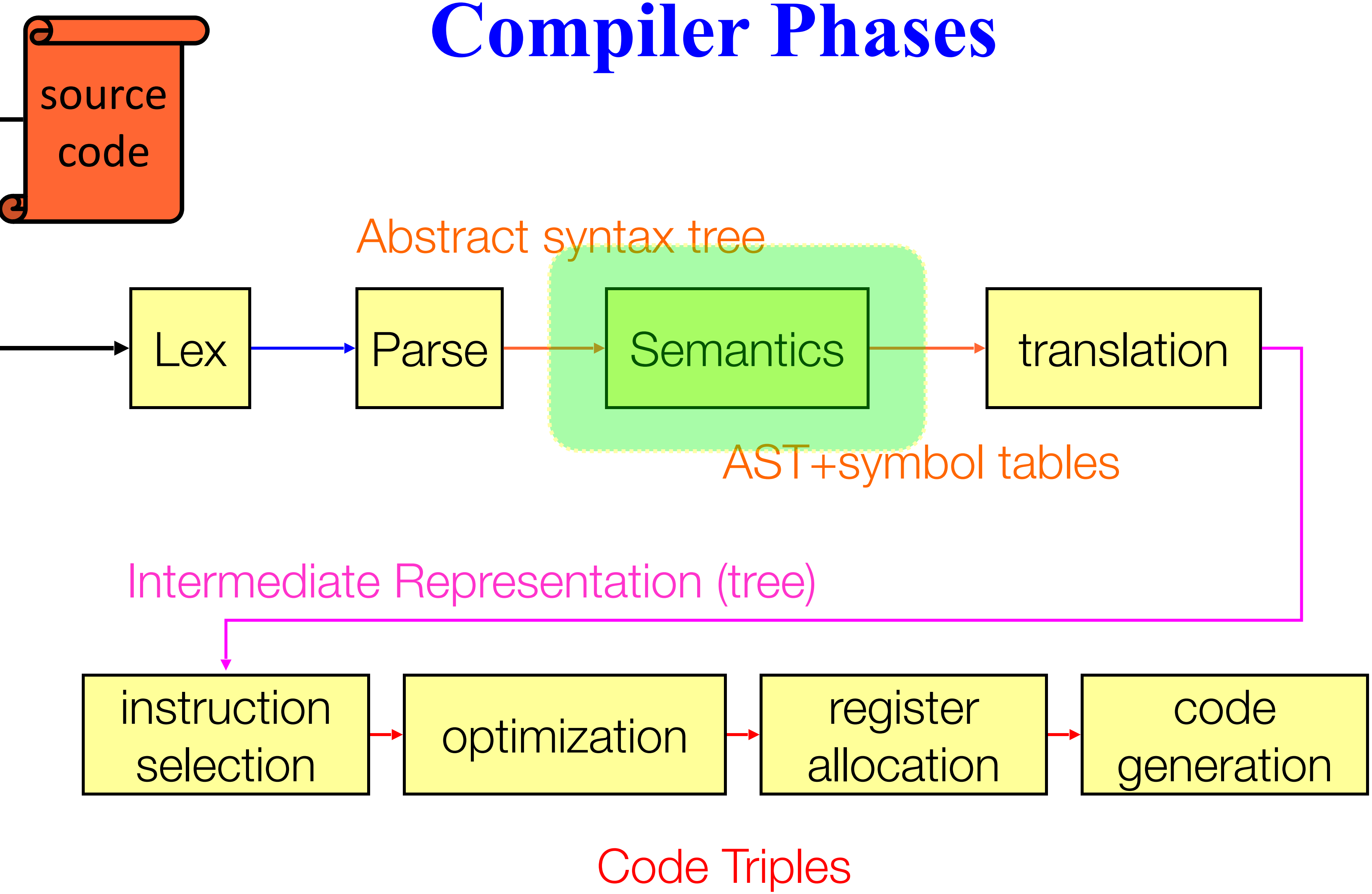
# Type checking

**15-411/15-611 Compiler Design**

Ben L. Titzer and Seth Copen Goldstein

Feb 13, 2025

# Compiler Phases



# Today

- Types & Type Systems
- Type Expressions
- Type Equivalence
- Type Checking

# Types

- A **type** is a set of values and a set of operations that can be performed on those values.
  - E.g, **`int`** in c0 is in $[-2^{31}, 2^{31})$
  - **`bool`** in C0 is in { **`false`**, **`true`** }
  - **`int`**s allow arithmetic operators + - * /
  - **`bool`**s allow logical operators && ||

# Types & Type systems

- A **type** is a set of values and a set of operations that can be performed on those values.
- **A Type system** is a set of rules which assign types to expressions, variables, storage locations,, and thus the entire program
  - What operations are valid for which types
  - Concise formalization of the checking rules
  - Specified as rules on the structure of expressions, …
  - Language specific

# Static vs Dynamic Types

- Static type: type assigned to an *expression* or *storage location* at compile time
- Dynamic type: type of a *value* at runtime
- Statically-typed language: every expression and storage location must have a type at compile time
- Dynamically-typed language: values carry dynamic type information used at runtime
- Untyped language: no typechecking, e.g., assembly

# Why Static Typing?

- Allows error detection by compiler
- Compiler can reason more effectively
  - don’t have to check for unsupported operations
  - values have most efficient representations
  - More optimizations
- Documentation!
- But:
  – requires at least some *type declarations*
  – type decls often can be inferred (ML, C+11)

# Dynamic checks

- Array index out of bounds
- null and casts Java
  - (maybe) null pointers in C
- Load-time type checking in Java
- Property access in JavaScript
- Sometimes can be eliminated statically
- Managed runtimes optimize dynamic checks through dynamic analysis

# Sound Type System

- If an expression is assigned type *t*, and it evaluates to a value *v*, then *v* is in the set of values defined by *t*
- IOW, dynamic type of value (at runtime) will always be within the static type of the expression (derived at compiled time)

- SML, OCAML, Scheme and Ada have sound type systems
- Most implementations of C and C++ do not

# Strongly Typed Language

- When no application of an operator to arguments can lead to a run-time type error, language is **strongly typed**
- strongly typed != statically typed

# Strongly Typed Language

- C++ claimed to be "strongly typed", but
  - Union types allow creating a value of one type and using it at another
  - Type coercions may cause unexpected (undesirable) effects
  - No array bounds check (in fact, no runtime checks at all)
  - Uninitialized values cause havoc
- SML, OCAML "strongly typed" but still must do dynamic array bounds checks, runtime type case analysis, and other checks

# Limitations

- Can still have runtime errors:
  - division by zero
  - exceptions
- Static type analysis has to be conservative, thus some “correct” programs will be rejected.

# Example: c0 type system

- Language type systems have *primitive types* (also: *basic types, atomic types*)
- C0: int, bool, char, string
- Also have *type constructors* that operate on types to produce other types
- C0: for any type *T, T* [ ], T* is a type.
- Extra types: void denotes absence of value

# Type Expressions

- *Type expressions* are used in declarations and type casts to define or refer to a type
  - *Primitive types*, such as **`int`** and **`bool`**
  - *Type constructors*, such as pointer-to, array-of, records and classes, templates, and functions
  - *Type names*, such as typedefs in C and named types in Pascal, refer to type expressions

# Type expressions: aliases

- Some languages allow type aliases (e.g., type definitions)
  - C: typedef int int_array[ ];
  - Modula-3: type int_array = array of int;
- int_array is type expression denoting same type as int [ ] -- not a type constructor

# Type Expressions: Arrays

- Different languages have various kinds of array types
- w/o bounds: array(T)
  - C, Java: T[ ], Modula-3: array of *T*
- size: array(T, L) (may be indexed 0..L-1)
  - C: T[*L*], Modula-3: array[*L*] of *T*
- upper & lower bounds: array(T,L,U)
  - Pascal, Modula-3: indexed L..U
- Multi-dimensional arrays (FORTRAN)

# Records/Structures

- More complex type constructor
- Has form $\{id_1: T_1, id_2: T_2, \ldots\}$ for some ids and types $T_i$
- Supports access operations on each field, with corresponding type
- C: struct { int a; float b; } corresponds to type {a: **int**, b: **float**}

# Function Types

- Some languages have first-class function types (C, ML, Modula-3, Pascal, not Java[1])
- Function value can be invoked with some argument expressions with types $T_i$, returns return type $T_r$.
- Type: $T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r$
- C: int f(float x, float y)
  - f: **float** × **float** → **int**
- Function types useful for describing methods, as in Java, even though not values, but need extensions for exceptions.

[1] Java 8 added lambda expressions and function interfaces

# Type Equivalence

- Name equivalence: Each distinct type name is a distinct type.
- Structural Equivalence: two types are identical if they have the same structure

# Name Equivalence

- Each type name is a distinct type, even when the type expressions the names refer to are the same
- Types are identical only if names match
- Used by Pascal (inconsistently)

```
type link = ^node;
 var next : link;
     last : link;
        p : ^node;
     q, r : ^node;
```

Using name equivalence:

$\mathbf{p} \neq \mathbf{next}$

$\mathbf{p} \neq \mathbf{last}$

$\mathbf{p} = \mathbf{q} = \mathbf{r}$

$\mathbf{next} = \mathbf{last}$

# Structural Equivalence

- Two types are the same if they are structurally identical
- Used in C0, C, Modula 3

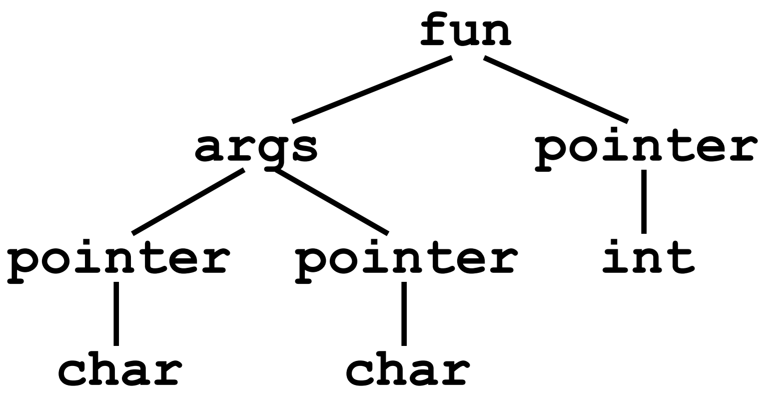
```
typedef node* link;
link next;
link last;
node* p;
node* q;
```

Using structural equivalence:

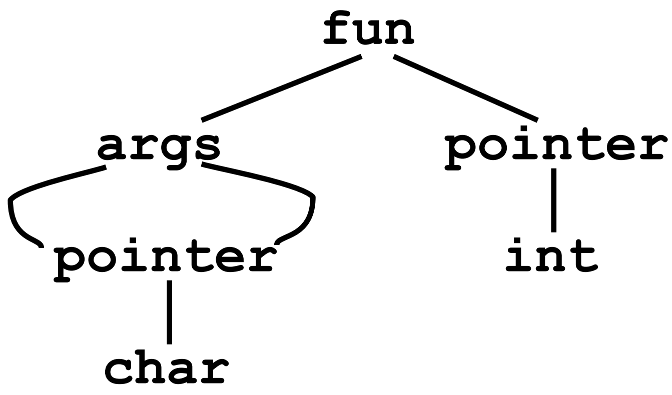
```
p = q = next = last
```

# Representing Types

```
int *f(char*,char*)
```



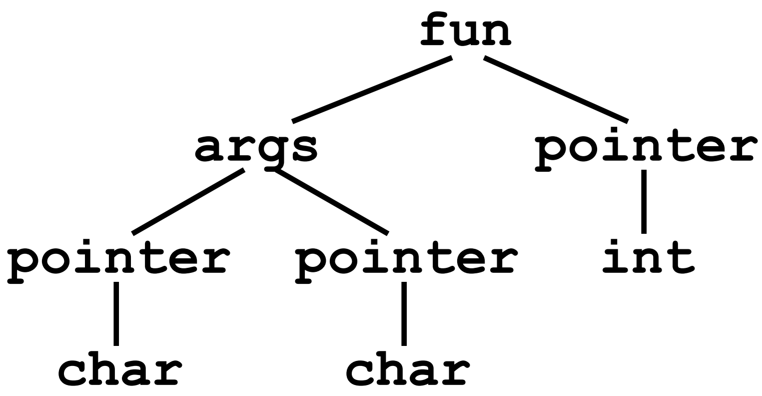

Tree forms



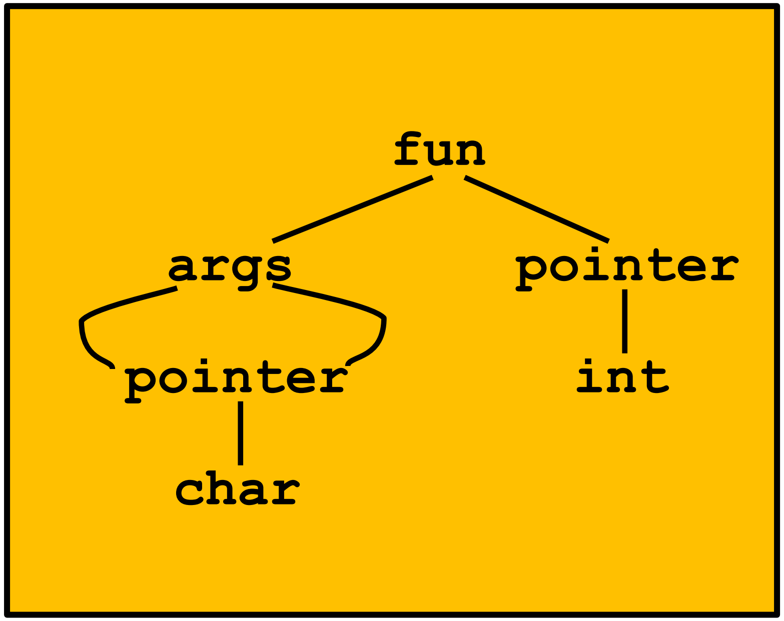

Directed Graph

# Representing Types

`int *f(char*,char*)`



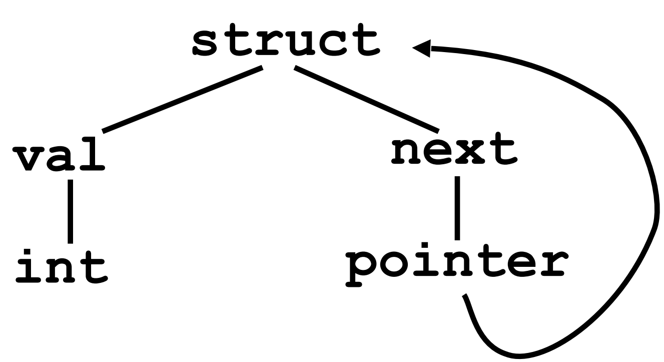

Tree forms




Directed Graph

# Cyclic Graph Representations

```
struct Node
{
  int val;
  struct Node *next;
};
```

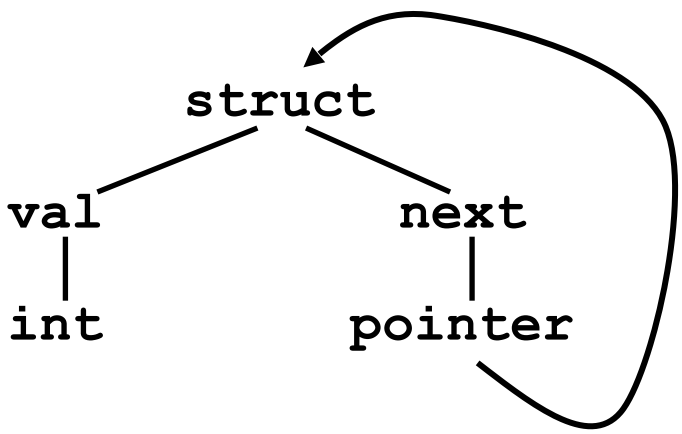



Cyclic graph

# Structural Equivalence (cont'd)

- Two structurally equivalent type expressions have the same pointer address when constructing graphs by sharing nodes

```
struct Node
{
  int val;
  struct Node *next;
};
```

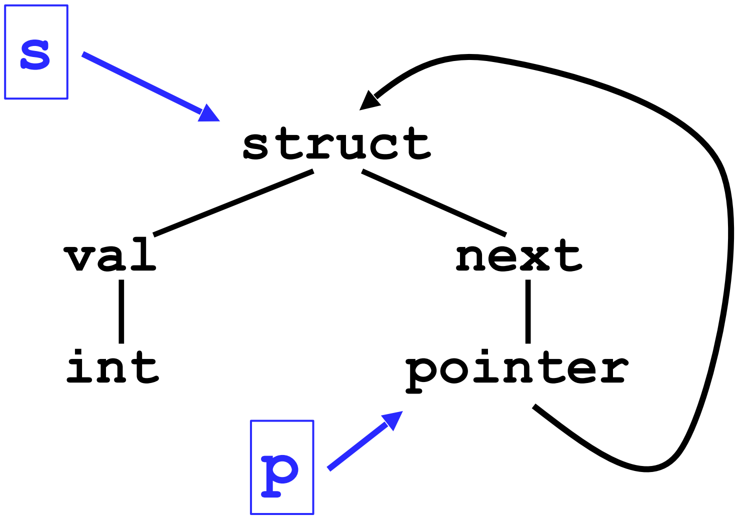


# Structural Equivalence (cont'd)

- Two structurally equivalent type expressions have the same pointer address when constructing graphs by sharing nodes

```
struct Node
{
  int val;
  struct Node *next;
};

struct Node s, *p;
```



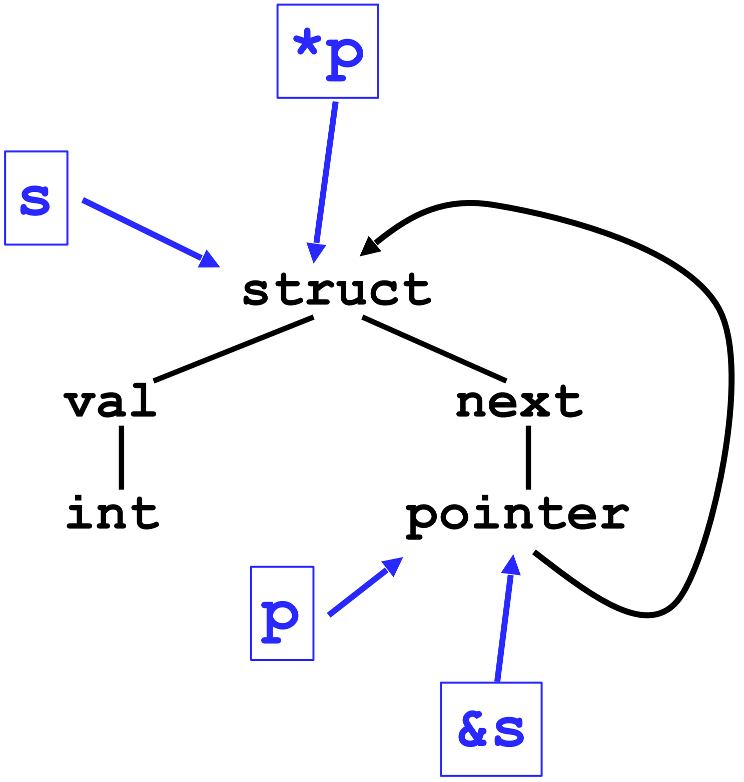
# Structural Equivalence (cont'd)

- Two structurally equivalent type expressions have the same pointer address when constructing graphs by sharing nodes

```
struct Node
{
  int val;
  struct Node *next;
};

struct Node s, *p;

… p = &s; // OK
… *p = s; // OK
```



# Constructing Type Graphs

- Construct over AST (or during parse)

| | | | |
|---|---|---|---|
| type | → | **`int`** | $$ = getIntType(); |
| | \| | **`bool`** | $$ = getBoolType(); |
| | \| | **`*`** type | $$ = makePtrType($2); |
| | \| | type **`[`** num **`]`** | $$ = makeArrayType($1, $3); |
| typedef | → | `typedef` type id | install($3,$2); |

- Invariant:

  Same structural type is same pointer.

# Type Checking

- When is op(arg1,…,argn) allowed?
- Type checking ensures that operations are applied to the right number of arguments of the right types
  Right type may mean:
  - same type as was specified, or
  - may mean that there is a predefined implicit coercion that will be applied
- Used to resolve overloaded operations

# Type Checking

- Statically-typed languages do *most* type checking statically
- Dynamically-typed languages (eg LISP, Prolog, JavaScript) do only dynamic type checking
- Gradually-typed languages do a mix of both

# Dynamic Type Checking

- Variables and storage locations don’t have types
  - Same variable may contain values of different types at different times
- Values carry type information
- Type checks are performed at runtime before executing an operation on values

# Dynamic Type Checking

- May introduce extra overhead at runtime
- Space overhead
  - values must carry type information
  - less efficient representation, such as a box on the heap
- Time overhead:
  - dynamic checks such as checking for string or int
- Errors aren't detected until invalid operation is executed => latent bugs
- Can make code harder to understand
- Some claim it is easier to prototype code

# Static Type Checking

- Performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time

# Static Type Checking

- Catches many programming errors at earliest point
- Can’t check types that depend on dynamically computed values
  - E.g. array bounds
- Can eliminate need to store type information on *most* values

# Static Type Checking

- Typical language restrictions
  - All variables initialized when created
  - Variable only used as one type
    - Union types allow for work-arounds, but effectively introduce dynamic type checks
- For memory safety
  - Can't convert pointers to ints
  - No manual free() => garbage collection

# Memory Safety

- Program doesn’t read/write “unauthorized” memory
  - Execution stack, return addresses
  - Heap, data structures
  - Executable code
- Requires a form of strong type safety
- Usually enforced with a combination of static and dynamic checks
- Allows a program to co-inhabit an address space with other programs
- All modern languages strive for memory safety

# Type Inference

- *Type inference*: A program analysis to assign a type to an expression from the program context of the expression
  - Fully static type inference first introduced by Robin Milner in ML
  - Haskell, OCAML, SML all use powerful type inference
    - Records complicate type inference
  - Java, C#, Rust, and others have local type inference

# Format of Type Judgments

- A *type judgement* has the form

$$\Gamma \vdash \text{exp} : \tau$$

- $\Gamma$ is a typing environment
  - Supplies the types of variables and functions
  - $\Gamma$ is a set of the form $\{ x : \sigma , . . . \}$
  - For any $x$ at most one $\sigma$ such that $(x : \sigma \in \Gamma)$
- exp is a program expression
- $\tau$ is a type to be assigned to exp
- $\vdash$ pronounced "turnstile", or "entails" (or "satisfies" or, informally, "shows")

# Axioms - Constants

$$\frac{}{\Gamma \vdash n : \text{int}}$$ (assuming *n* is an integer constant)

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

- These rules are true in any typing environment
- $\Gamma$, *n* are meta-variables

# Axioms – Variables

Notation: Let $\Gamma(x) = \tau$ if $x : \tau \in \Gamma$

Variable axiom:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

# Simple Rules - Arithmetic

Primitive operators ( $\oplus \in \{ +, *, \&\&, \ldots \}$ ):

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau}$$

$\tau$ is a type variable, i.e., it can take any type but all instances of $\tau$ must be the same.

# Simple Rules – Relational Ops

Relations ( $\sim\ \in \{ <\ ,\ >\ ,\ ==,\ <=,\ >= \}$):

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \sim e_2 : \text{bool}}$$

Do we know what $\tau$ is here?

# Example: $\{x{:}int\} \vdash x + 2 == 3 : bool$

What do we need to show first?

$$\frac{}{\{x{:}int\} \vdash x + 2 == 3 : bool}$$

# Example: $\{x{:}int\} \vdash x + 2 == 3 : bool$

What to do on left side?

$$\frac{\{x : int\} \vdash x + 2 : int \qquad \{x{:}int\} \vdash 3 : int}{\{x{:}int\} \vdash x + 2 == 3 : bool}$$

# Example: $\{x:int\} \vdash x + 2 == 3 :bool$

Almost Done

$$\frac{\dfrac{\{x:int\} \vdash x:int \qquad \{x:int\} \vdash 2:int}{\{x : int\} \vdash x + 2 : int} \qquad \{x:int\} \vdash 3 :int}{\{x:int\} \vdash x + 2 == 3 : bool}$$

# Example: $\{x{:}int\} \vdash x + 2 == 3 : bool$

Complete Proof (type derivation)

$$
\dfrac{
  \dfrac{
    \dfrac{\Gamma(x) = \text{int}}{\{x{:}\text{int}\} \vdash x{:}\text{int}}
    \qquad
    \dfrac{}{\{x{:}\text{int}\} \vdash 2{:}\text{int}}
  }{\{x : \text{int}\} \vdash x + 2 : \text{int}}
  \qquad
  \dfrac{}{\{x{:}\text{int}\} \vdash 3 : \text{int}}
}{\{x{:}\text{int}\} \vdash x + 2 == 3 : \text{bool}}
$$

# Simple Rules - Booleans

Connectives

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \,\&\&\, e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \,||\, e_2 : \text{bool}}$$

# Function Application

- Application rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}$$

- If you have a function expression $e_1$ of type $\tau_1 \to \tau_2$ applied to an argument $e_2$ of type $\tau_1$, the resulting expression $e_1(e_2)$ has type $\tau_2$

# What about statements?

- Don't normally care about the type.
- But, they result in a function returning a value with a type.
- If a function returns type $\tau$, then we say s is well typed if,

$$\Gamma \vdash s:[\tau]$$

read as: "s is well typed if it is consistent with the function returning type $\tau$"

# Language

- Our language:

```
e := n | x | e1+e2 | e1 && e2
s := x←e
  |  if(e,s1,s2)
  |  while(e,s)
  |  return(e)
  |  seq(s1,s2)
  |  decl(x,τ,s)
  |  nop
```

# What about statements?

$$\frac{\Gamma(x) = \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash \mathsf{assign}(x, e) : [\tau]} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \mathsf{if}(e, s_1, s_2) : [\tau]}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash s : [\tau]}{\Gamma \vdash \mathsf{while(e, s)} : [\tau]} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{return}(e) : [\tau]}$$

$$\frac{}{\Gamma \vdash \mathsf{nop} : [\tau]} \qquad \frac{\Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \mathsf{seq}(s_1, s_2) : [\tau]}$$

$$\frac{\Gamma, x{:}\tau' \vdash s : [\tau]}{\Gamma \vdash \mathsf{decl}(x, \tau', s) : [\tau]}$$

# Effect on Γ

$$\frac{\Gamma(x) = \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash \mathsf{assign}(x, e) : [\tau]} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \mathsf{if}(e, s_1, s_2) : [\tau]}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash s : [\tau]}{\Gamma \vdash \mathsf{while}(\mathsf{e}, \mathsf{s}) : [\tau]} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{return}(e) : [\tau]}$$

$$\frac{}{\Gamma \vdash \mathsf{nop} : [\tau]} \qquad \frac{\Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \mathsf{seq}(s_1, s_2) : [\tau]}$$

$$\frac{\Gamma, x{:}\tau' \vdash s : [\tau]}{\Gamma \vdash \mathsf{decl}(x, \tau', s) : [\tau]}$$

# Shadowing?

$$\frac{\Gamma(x) = \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash \mathsf{assign}(x, e) : [\tau]} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \mathsf{if}(e, s_1, s_2) : [\tau]}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash s : [\tau]}{\Gamma \vdash \mathsf{while(e, s)} : [\tau]} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{return}(e) : [\tau]}$$

$$\frac{}{\Gamma \vdash \mathsf{nop} : [\tau]} \qquad \frac{\Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \mathsf{seq}(s_1, s_2) : [\tau]}$$

$$\frac{\Gamma, x{:}\tau' \vdash s : [\tau]}{\Gamma \vdash \mathsf{decl}(x, \tau', s) : [\tau]} \quad x \notin \mathrm{dom}(\Gamma)$$

# Or, as in L2 handout

$$\frac{x : \tau' \notin \Gamma \text{ for any } \tau' \quad \Gamma, x : \tau \vdash s \; valid}{\Gamma \vdash \mathsf{declare}(x, \tau, s) \; valid}$$

# Function Rule

- Rules describe types, but also how the environment $\Gamma$ may change

$$\frac{\Gamma, \{f : \tau_1 \rightarrow \tau_2 ,\ x : \tau_1 \} \vdash s\ [\tau_2]}{\Gamma \vdash \tau_2\ f(\tau_1\ x)\ s}$$

# Implementing rules

- Start from goal judgments for each function

$$\Gamma \vdash \tau\, id\ (\ \ldots,\ \tau_i\ a_{i,}\ \ldots\ )\ \{\ s\ \}$$

- Work backward applying inference rules to sub-trees of abstract syntax trees
- Exactly the same kind of recursive traversal as lecture 7

# Other Issues

- What to do with types after type checking?
  - decorate AST?
  - Typed IR?
  - Typed triples?
- What to do on errors?
  - uninitialized variable?
  - undeclared variable?
  - wrong return type?
  - wrong operator type?