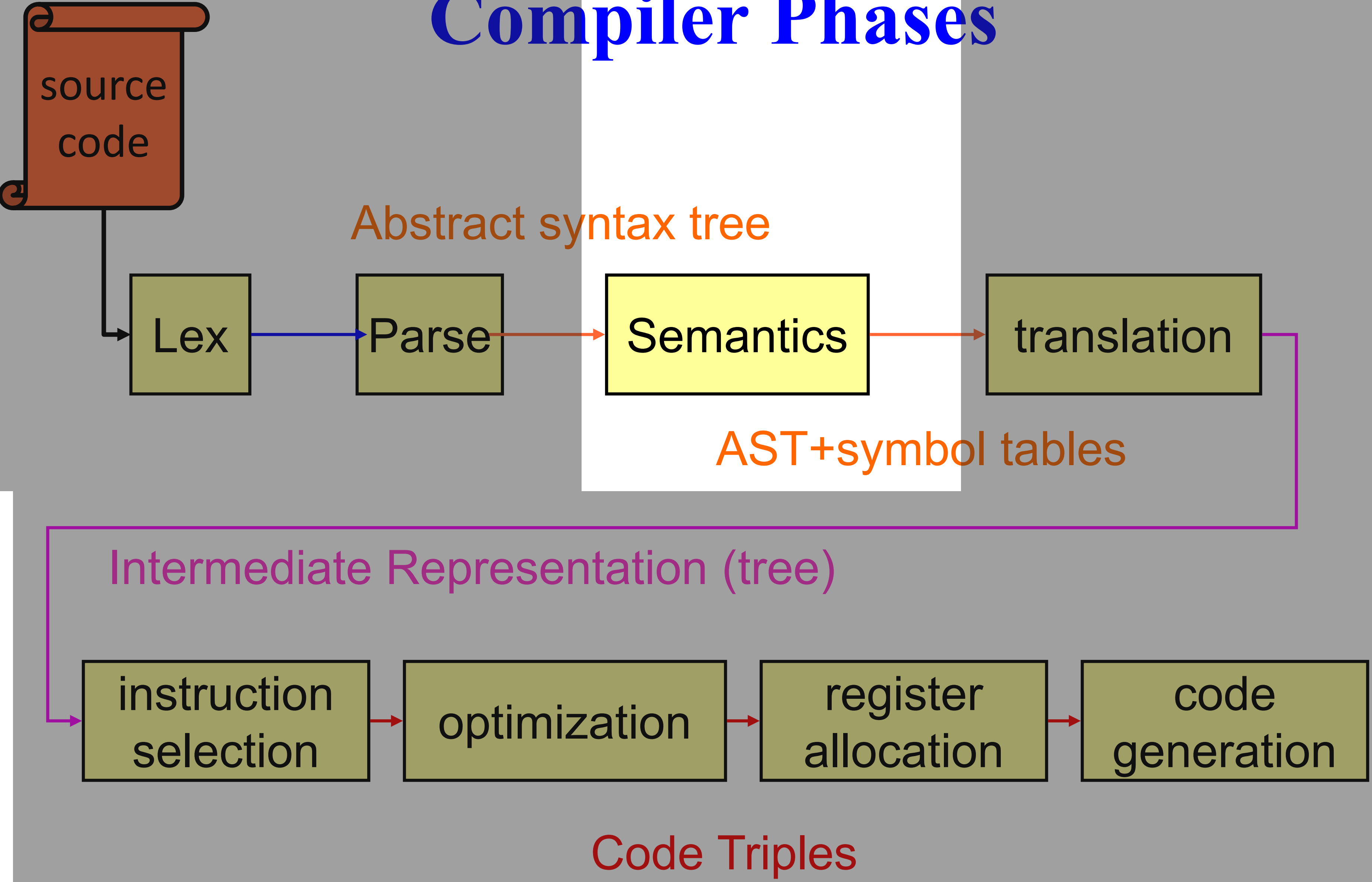
Typechecking

15-411/15-611 Compiler Design

Seth Copen Goldstein

October 6, 2020

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`** }

Types & Typesystems

- A **type** is a set of values and a set of operations that can be performed on those values.
- **A Typesystem** is a set of rules which assign types to expressions, statements, 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 at compile time
- Dynamic type: type assigned to a storage location at run time
- Statically typed language: static type assigned to every expression at compile time
- Dynamically typed language: type of an expression determined at run time
- Untyped language: no typechecking, e.g., assembly

Why Static Typing?

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

Dynamic checks

- Array index out of bounds
- null in Java, null pointers in C
- Inter-module type checking in Java
- Sometimes can be eliminated through static analysis (but usually harder than type checking)

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 expression (at runtime) is 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)
- 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, …} 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**}
- Class types (e.g. Java) extension of record types

Functions

- Some languages have first-class function types (C, ML, Modula-3, Pascal, not Java)
- 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.

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:

$\mathtt{p} \neq \mathtt{next}$

$\mathtt{p} \neq \mathtt{last}$

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

$\mathtt{next} = \mathtt{last}$

Structural Equivalence

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

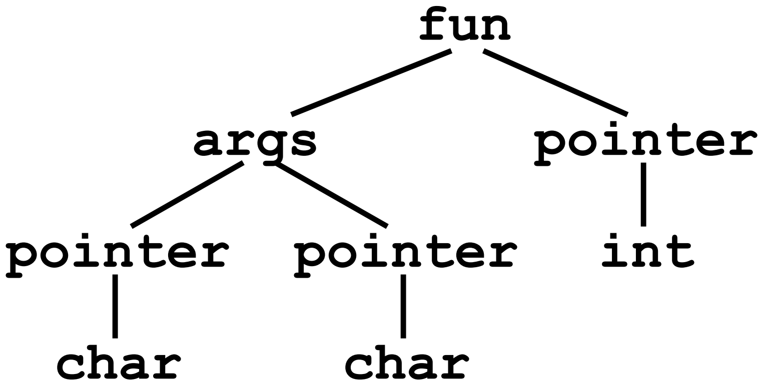
```
typedef node* link;
link next;
link last;
node* p;
node* q;
```

Using structural equivalence:

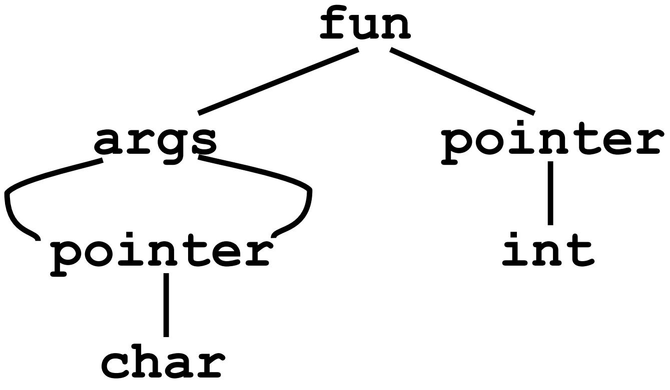
```
p = q = next = last
```

Representing Types

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



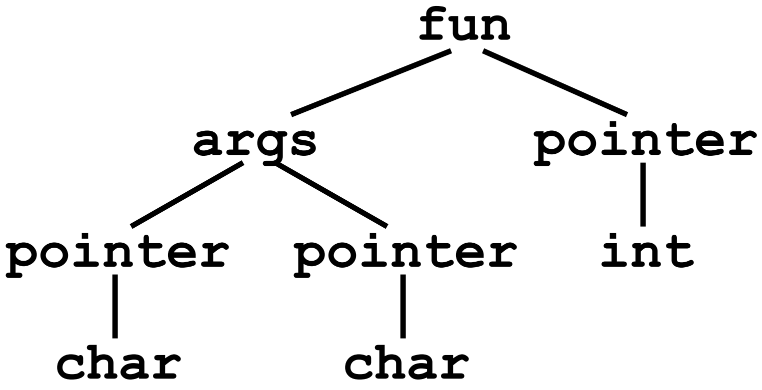
Tree forms



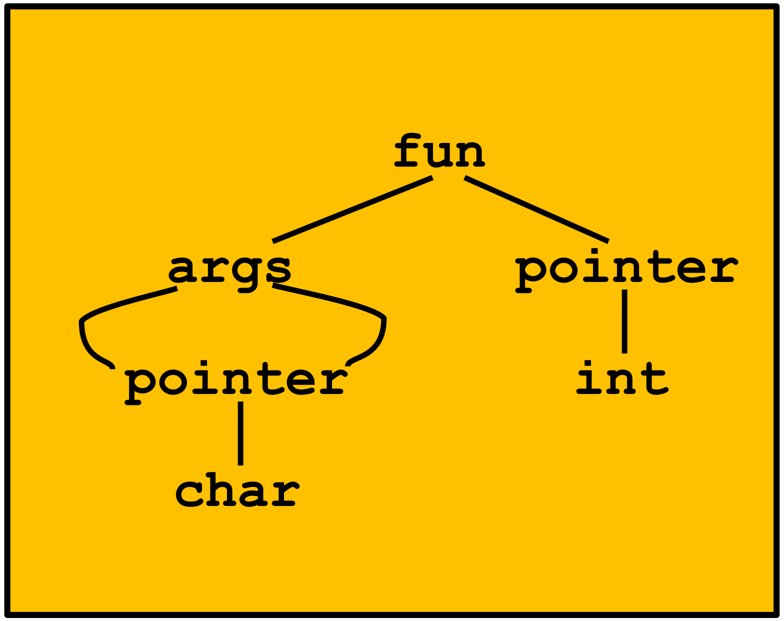
DAGs

Representing Types

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


fun
args
pointer
pointer
pointer
int
char
char

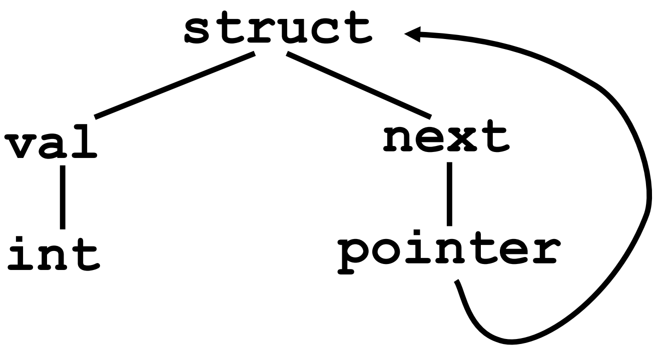
Tree forms


fun
args
pointer
pointer
int
char

DAGs

Cyclic Graph Representations

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

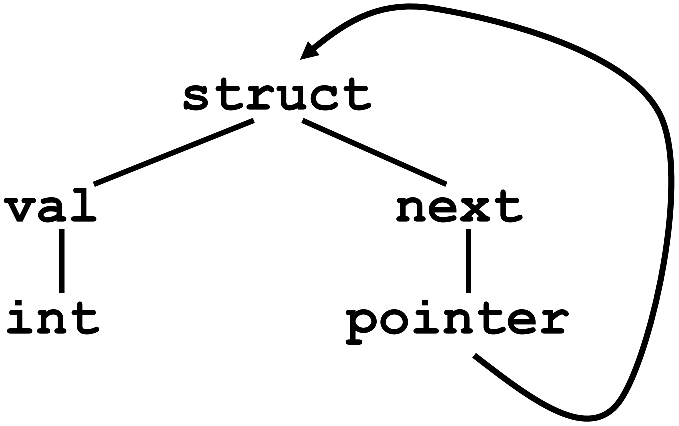


Cyclic graph

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;
};
```

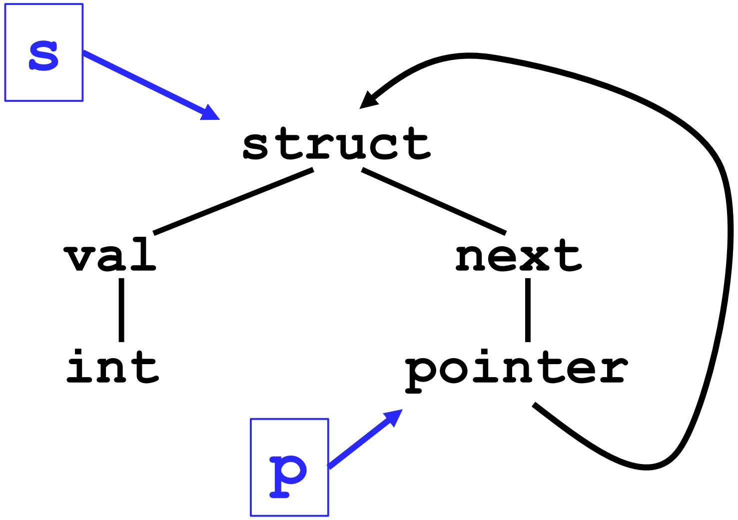


Structural Equivalence (cont'd)

- 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;
```

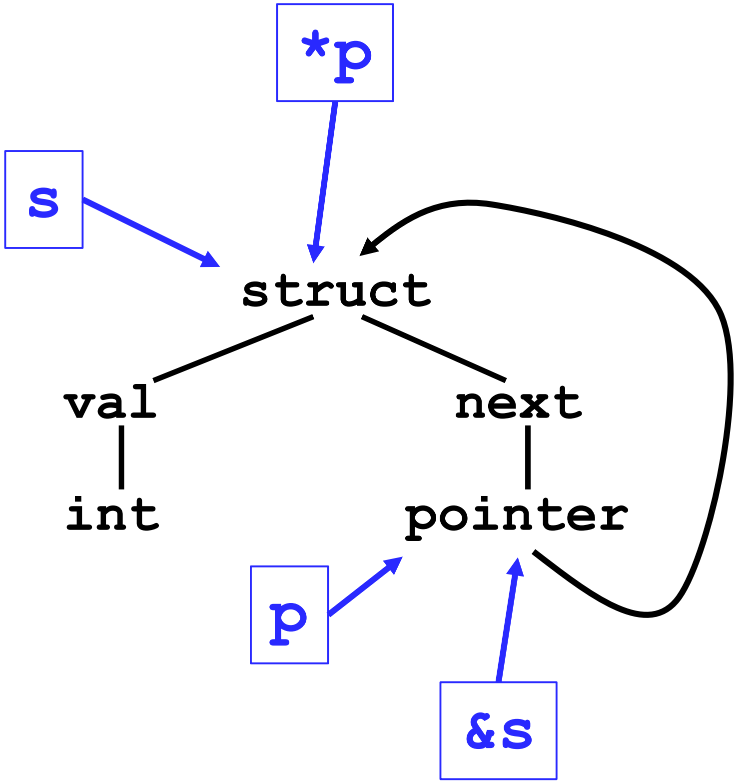


Structural Equivalence (cont'd)

- 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

- Type checking may be done statically at compile time or dynamically at run time
- Dynamically typed languages (eg LISP, Prolog, javascript) do only dynamic type checking
- Statically typed languages can do most type checking statically

Dynamic Type Checking

- Performed at run-time before each operation is applied
- Types of variables and operations left unspecified until run-time
  - Same variable may be used at different times with different types

Dynamic Type Checking

- Data object must contain type information
- Errors aren’t detected until violating application is executed
- May introduce extra overhead at runtime.
- Can make code hard to read
- Supposedly, 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

- Can eliminate need to store type information in data object if no dynamic type checking is needed
- Catches many programming errors at earliest point
- Can’t check types that depend on dynamically computed values
  - Eg: array bounds

Static Type Checking

- Typically places restrictions on languages
  - Garbage collection
  - References instead of pointers
  - All variables initialized when created
  - Variable only used as one type
    - Union types allow for work-arounds, but effectively introduce dynamic type checks

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 Miller in ML
  - Haskle, OCAML, SML all use type inference
    - Records are a problem for type inference

Format of Type Judgments

- A *type judgement* has the form

$$\Gamma \mid\!- \text{exp} : \tau$$

- Γ is a typing environment
  - Supplies the types of variables and functions
  - Γ is a set of the form $\{ x : \sigma , \ldots \}$
  - For any x at most one σ such that $(x : \sigma \in \Gamma)$
- exp is a program expression
- τ is a type to be assigned to exp
- |- pronounced “turnstyle”, or “entails” (or “satisfies” or, informally, “shows”)

Axioms - Constants

$$\frac{}{\Gamma \;|\!-\; n : \mathrm{int}}$$ (assuming n is an integer constant)

$$\frac{}{\Gamma \;|\!-\; \mathrm{true} : \mathrm{bool}} \qquad \frac{}{\Gamma \;|\!-\; \mathrm{false} : \mathrm{bool}}$$

- These rules are true with any typing environment
- Γ, 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 ($\oplus \in \{ +, *, \&\&, \ldots \}$):

$$\frac{\Gamma \mid\!- \; e_1 : \tau \qquad \Gamma \mid\!- \; e_2 : \tau}{\Gamma \mid\!- \; e_1 \oplus e_2 : \tau}$$

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

Simple Rules – Relational Ops

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

$$\frac{\Gamma \mid\!- e_1 : \tau \qquad \Gamma \mid\!- e_2 : \tau}{\Gamma \mid\!- e_1 \sim e_2 : \mathrm{bool}}$$

Do we know what τ is here?

Example: {x:int} |- x + 2 = 3 :bool

What do we need to show first?

$$\frac{}{\{x:int\} \mid\!- \; x + 2 = 3 : bool}$$

Example: {x:int} |- x + 2 = 3 :bool

What to do on left side?

$$\frac{\{x : \text{int}\} \;|\!-\; x + 2 : \text{int} \qquad\qquad \{x{:}\text{int}\} \;|\!-\; 3 : \text{int}}{\{x{:}\text{int}\} \;|\!-\; x + 2 = 3 : \text{bool}}$$

Example: {x:int} |- x + 2 = 3 :bool

Almost Done

$$\frac{\dfrac{\{x:\text{int}\} \text{ |- } x:\text{int} \qquad \{x:\text{int}\} \text{ |- } 2:\text{int}}{\{x : \text{int}\} \text{ |- } x + 2 : \text{int}} \qquad \{x:\text{int}\} \text{ |- } 3 :\text{int}}{\{x:\text{int}\} \text{ |- } x + 2 = 3 : \text{bool}}$$

Example: {x:int} |- x + 2 = 3 :bool

Complete Proof (type derivation)

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

Simple Rules - Booleans

Connectives

$$\frac{\Gamma \mid\!- \; e_1 : \text{bool} \qquad \Gamma \mid\!- \; e_2 : \text{bool}}{\Gamma \mid\!- \; e_1 \;\&\&\; e_2 : \text{bool}}$$

$$\frac{\Gamma \mid\!- \; e_1 : \text{bool} \qquad \Gamma \mid\!- \; e_2 : \text{bool}}{\Gamma \mid\!- \; e_1 \;||\; e_2 : \text{bool}}$$

Function Application

- Application rule:

$$\frac{\Gamma \text{ |- } e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \text{ |- } e_2 : \tau_1}{\Gamma \text{ |- } e_1(e_2) : \tau_2}$$

- If you have a function expression e_1 of type $\tau_1 \rightarrow \tau_2$ applied to an argument e_2 of type τ_1, the resulting expression $e_1(e_2)$ has type τ_2

What about statements?

- Statements don’t have types.
- But, they result in a function returning a value with a type.
- If a function returns type τ, then we say s is well typed if,

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

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

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

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

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

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

Shadowing?

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

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

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

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

Or, as in L2 handout

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

Function Rule

- Rules describe types, but also how the environment Γ may change

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

Implementing rules

- Start from goal judgments for each function

$$\Gamma \mid\!- (\mathit{id} (\ldots, a_i : T_{i,} \ldots) : T = E)$$

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

Other Issues

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