Static Type Checking

Type checking means looking at a parsed program to make sure that:
- every piece of the program is well-typed;
- all identifiers are used in a type-consistent way.

Input:
- program AST or parse tree (or perhaps done during parsing)

Outputs:
- “OK” or typing error message(s)
- Perhaps type-annotated AST, perhaps per-identifier type information, etc.

Will see how to specify type-checking rules using
- Attribute grammars
- Inference rules

Environment and Symbol Tables

To type-check code that uses variables, function names, or other identifiers, must maintain an environment mapping identifiers to types.
- Environment is just a dictionary with key=identifier and value=type.
- Many possible representations: hash table, linked list or tree structure, etc.
- Current environment grows and shrinks as identifiers enter and leave scope.

It is often convenient to treat the current environment as a parameter of the type-checking code, so it can vary as we do a recursive traversal.

A more traditional approach is to treat the environment as a global whose contents are mutated as we traverse the program. In this case, the environment is usually called a symbol table.

Syntax-Directed Type Checking

Initially, we will show how to specify type-checking using attribute grammars over parse trees.
- Calculate a synthesized type attribute for each expression.
- Check that expressions are used correctly within other expressions and statements.
- Maintain environment information in an attribute or a global symbol table.
**Motivating Inherited Attributes**

Sometimes it’s convenient to make a node’s attributes dependent on siblings or ancestors in tree.

Useful for expressing dependence on context, e.g., relating identifier uses to declarations. (This is especially important because CF grammar cannot capture such dependencies.)

Example: Simple C-like Variable Declarations

\[ D \rightarrow T \ L \]
\[ T \rightarrow \text{int} \mid \text{real} \]
\[ L \rightarrow L_1, \text{id} \mid \text{id} \]

Parse tree for `real a, b, c`:

```
D
□
□
T
real
@
@
L
□□
L
□□
L
a
,@@b
,
@@c
```

**Attribute Evaluation**

Dependency arrows for a dependency graph; we must evaluate attributes in topological order of dependency graph.

If attributes are defined on parse tree, may want to evaluate attributes while (or instead of) building the tree. This is sometimes possible:

- Saw how to evaluate **S-attributed** grammar, in which all attributes are synthesized, during bottom-up parsing; this method doesn’t work for inherited attributes.
- Top-down parser can easily evaluate **L-attributed** grammars, in which attributes don’t depend on their right ancestors. (Bottom-up parsers can sometimes handle these too, though with difficulty.) Example follows.
- For more complicated attribute grammars, might have to build some or all of tree before evaluating attributes.

**Inherited Attribute Grammar**

\[ D \rightarrow T \ L \]
\[ T \rightarrow \text{int} \]
\[ T \rightarrow \text{real} \]
\[ L \rightarrow L_1, \text{id} \]
\[ L \rightarrow \text{id} \]

Here `addsymb` adds `id` and its type to symbol table, and `L.type` is an inherited attribute.

A parse tree showing **dependency** relations among attributes:

```
D
□
□
T
real
@
@
L
□□
L
□□
L
a
,@@b
,
@@c
```

**Attribute Evaluation during Recursive Descent**

Each non-terminal function now takes inherited attribute values as arguments and return (record of) synthesized attribute value(s) as result.

Example revisited (with left-recursion removed):

```java
class Ty {};
static Ty intTy = new Ty(); static Ty realTy = new Ty();

void D() { Ty ty = T(); L(ty); }
Ty T() {
  if (tok == INT) {
    tok = lex(); return intTy;
  } else if (tok == REAL) {
    tok = lex(); return realTy;
  } else error();
}
void L(Ty ty) {
  if (tok == ID) {
    addsymb(lexeme,ty); tok = lex();
  } else error();
  if (tok == ',') {
    tok = lex(); L(ty); }
```
AVOIDING INHERITED ATTRIBUTES

When using bottom-up parser (e.g., with yacc or CUP), it is desirable to avoid inherited attributes.

There are several approaches:

- Move the activity requiring the attribute to a higher node in the tree, by substituting a synthesized attribute for the inherited one, e.g.:

  \[
  D \rightarrow T \ id \quad \text{for each } \ id \text{ in } L.\text{list} \\
  \quad \text{addsym}(\ id.\text{name}, T.\text{type})
  \]

  \[
  T \rightarrow \text{int} \quad T.\text{type} := \text{integer}
  \]

  \[
  T \rightarrow \text{real} \quad T.\text{type} := \text{real}
  \]

  \[
  L \rightarrow L_1, \ id \quad L.\text{list} := \text{append-list}(\ id, L_1.\text{list})
  \]

  \[
  L \rightarrow \ id \quad L.\text{list} := \text{singleton-list}(\ id)
  \]

AVOIDING INHERITED ATTRIBUTES (2)

- Can sometimes rewrite grammar, e.g.:

  \[
  D \rightarrow T \ id \quad \{ D.\text{type} := T.\text{type}; \\
  \quad \text{addsym}(\ id.\text{name}, T.\text{type}) \}
  \]

  \[
  D \rightarrow D_1, \ id \quad \{ D.\text{type} := D_1.\text{type}; \\
  \quad \text{addsym}(\ id.\text{name}, D.\text{type}) \}
  \]

  \[
  T \rightarrow \text{int} \quad T.\text{type} := \text{integer}
  \]

  \[
  T \rightarrow \text{real} \quad T.\text{type} := \text{real}
  \]

ATTRIBUTES ON AST’S

Attribute grammar method extends to abstract grammars (not intended for parsing), e.g., AST grammars.

- Same concept, but attribute evaluation always occurs after whole tree is built.

- Can use recursive descent as an attribute evaluation technique (regardless of how parsing was performed).

- Typical applications: typechecking, code generation, interpretation.

Why attribute grammars?

- Compact, convenient formalism.

- Local rules describe entire computation.

- Separate traversal from computation.

- (Purely functional rules can be evaluated in any order.)

CHECKING OF E LANGUAGE (HOMEWORK 1)

Can view checking process as evaluation of following attribute grammar, where

- \( \text{exp.ok} \) and \( \text{exp.oks} \) are synthesized boolean attributes indicating whether expression has checked successfully; and

- \( \text{exp.env} \) and \( \text{exp.envs} \) are inherited environment attributes (with operators empty, extend, and lookup) containing entries for all in-scope variables.
Checking Types for a Richer Language

Consider a simple language of declarations, statements, and expressions.

\[ P \rightarrow D ; S \quad \{ \text{S.env} = D.env; \} \]

Actions for declarations synthesize environment attributes:

\[ D \rightarrow \epsilon \quad \{ \text{D.env} = \text{empty} \} \]
\[ D \rightarrow \text{id} : T_1 ; D_1 \quad \{ \text{D.env} = \text{extend}(D_1.env, \text{binding(id, T_1.type))} \} \]
\[ T \rightarrow \text{bool} \quad \{ \text{T.type} := \text{boolean} \} \]
\[ T \rightarrow \text{int} \quad \{ \text{T.type} := \text{integer} \} \]
\[ T \rightarrow \text{array} \ T_1 \quad \{ \text{T.type} := \text{array}(T_1.type) \} \]
\[ T \rightarrow \text{pair} T_1 T_2 \quad \{ \text{T.type} := T_1.type \times T_2.type \} \]

Expressions

Actions for expressions check for compatible operands and synthesize attribute type:

\[ E \rightarrow \text{num} \quad \{ \text{E.type} := \text{integer} \} \]
\[ E \rightarrow \text{id} \quad \{ \text{E.type} := \text{lookup(E.env,id)} \} \]
\[ E \rightarrow (E_1,E_2) \quad \{ \text{E.env} = E_1.env; E_2.env = E_2.env; E.type = E_1.type \times E_2.type \} \]
\[ E \rightarrow E_1 \text{ div} E_2 \quad \{ \text{E.env} = E_2.env; \text{E.type} := \text{integer} \}
  \text{if not (E_1.type = integer and E_2.type = integer) then}
  \text{issue type error;}
  \text{T.type} := T_1; \]
\[ E \rightarrow E_1 \text{ or} E_2 \quad \{ \text{E.env} = E_2.env; \text{E.type} := \text{boolean} \}
  \text{if not (E_1.type = boolean and E_2.type = boolean) then}
  \text{issue type error;}
  \text{E.type} := \text{boolean} \}
\[ E \rightarrow E_1 [ E_2 ] \quad \{ \text{E.env} = E_2.env; \text{E.type} := \text{array(T)} \}
  \text{if (E_1.type = array(T) and E_2.type = integer) then}
  \text{E.type} := T; \]
\[ E \rightarrow E_1 . fst \quad \{ \text{E.env} = E.env; \}
  \text{if E_1 = T_1 \times T_2 then}
  \text{T.type} := T_1; \]
\[ E \rightarrow E_1 < E_2 \quad \{ \text{E.env} = E_2.env; \text{E.type} := \text{array(T)} \}
  \text{if not (E_1.type = integer and E_2.type = integer) then}
  \text{issue type error;}
  \text{E.type} := \text{boolean} \}
\[ E \rightarrow E_1 = E_2 \quad \{ \text{E.env} = E_2.env; \text{E.type} := \text{boolean} \}
  \text{if not ((E_1.type = boolean or E_1.type = integer) and E_1.type = E_2.type) then}
  \text{issue type error;}
  \text{E.type} := \text{boolean} \}

Issuing error might or might not stop the checking process. If it doesn’t, try to choose a synthesized type value that prevents a cascade of messages from a single mistake.
**Checking Statements**

In most languages, statements don’t have a type, so no point in synthesizing an attribute. Actions just check component types:

\[ S \rightarrow \text{id} := E_1 \quad \{ E_1.env = S.env; \]
\[ \quad \text{if } E_1.\text{type} \neq \text{lookup}(S.env,\text{id}) \text{ then} \]
\[ \quad \text{issue type error} \} \]
\[ (\text{Must also check that } \text{id} \text{ is an l-value that can be assigned into.}) \]

\[ S \rightarrow \text{if } E_1 \text{ then } S_1 \quad \{ E_1.env = S.env; S_1.env = S.env; \]
\[ \quad \text{if } E_1.\text{type} \neq \text{boolean} \text{ then} \]
\[ \quad \text{issue type error} \} \]

\[ S \rightarrow S_1 ; S_2 \quad \{ S_1.env = S.env; S_2.env = S.env; \} \]

**Type Conversions**

Implicit conversions (or “coercions”) occur as a result of applying semantic rules of the language, e.g., perhaps evaluating \( r + i \), where \( r \) is a real and \( i \) is an integer, causes implicit conversion of the fetched value of \( i \) to a real before the addition. This complicates type-checking:

\[ E \rightarrow E_1 + E_2 \quad \{ E_1.env = E.env; E_2.env = E.env; \]
\[ \quad \text{case } \text{\( (E_1.\text{type},E_2.\text{type}) \)} \text{ of} \]
\[ \quad \text{ (integer,integer): } E.\text{type} := \text{integer} \]
\[ \quad \text{ (integer,real): } \]
\[ \quad \text{ (real,integer): } \]
\[ \quad \text{ (real,real): } E.\text{type} := \text{real} \]
\[ \quad \text{otherwise: issue type error} \} \]

The relationship between integer and real is a special case of subtyping (more later).

**Typing Judgments**

A more compact way to specify typing rules is by using inference rules or judgments in the style of mathematical logic.

Each judgment for expressions has the form

\[ TE \vdash e : t \]

Intuitively this says that expression \( e \) has type \( t \), under the assumption that the type of each variable used in \( e \) is given by the type environment \( TE \).

We write \( TE(x) \) for the result of looking up \( x \) in \( TE \), and \( TE + \{ x \rightarrow t \} \) for the type environment obtained from \( TE \) by extending it with a new binding from \( x \) to \( t \).

The key point is that an expression is well-typed if-and-only-if we can derive a typing judgment for it.
**Sample Typing Judgments**

Here are some of the rules from the attribute-grammar formalism transformed into judgments.

\[
\begin{align*}
\text{TE} & \vdash \text{num} : \text{integer} \quad \text{(Num)} \\
\text{id} & \in \text{dom(TE)} \\
\text{TE} & \vdash \text{id} : \text{TE(id)} \quad \text{(Var)} \\
\text{TE} & \vdash e_1 : t_1 \quad \text{TE} \vdash e_2 : t_2 \\
\text{TE} & \vdash (e_1, e_2) : t_1 \times t_2 \quad \text{(Pair)} \\
\text{TE} & \vdash e_1 : \text{integer} \quad \text{TE} \vdash e_2 : \text{integer} \\
\text{TE} & \vdash e_1 \text{ div } e_2 : \text{integer} \quad \text{(Div)} \\
\text{TE} & \vdash e_1 : \text{boolean} \quad \text{TE} \vdash e_2 : \text{boolean} \\
\text{TE} & \vdash e_1 < e_2 : \text{boolean} \quad \text{(LT)} \\
\text{TE} & \vdash e_1 : \text{boolean} \quad \text{TE} \vdash e_2 : \text{boolean} \\
\text{TE} & \vdash e_1 = e_2 : \text{boolean} \quad \text{(EQI)} \\
\text{TE} & \vdash e_1 : \text{boolean} \quad \text{TE} \vdash e_2 : \text{boolean} \\
\text{TE} & \vdash e_1 = e_2 : \text{boolean} \quad \text{(EQB)}
\end{align*}
\]

**Statement Rules**

Judgments for statements omit the result environment, and simply assert that the statement is well-typed.

\[
\begin{align*}
\text{TE} & \vdash e : t \quad \text{TE(id)} = t \\
\text{TE} & \vdash \text{id} := e \quad \text{(Assign)} \\
\text{TE} & \vdash e : \text{boolean} \quad \text{TE} \vdash s \quad \text{(If)} \\
\text{TE} & \vdash \text{if } e \text{ then } s \\
\text{TE} & \vdash s_1 \quad \text{TE} \vdash s_2 \\
\text{TE} & \vdash s_1 ; s_2 \quad \text{(Sequence)} \\
\vdash \text{texp : t} \quad \text{TE} + \{ \text{id } \mapsto t \} \vdash s \\
\text{TE} & \vdash \text{var \ id \ : \ texp ; s} \quad \text{(Decl)}
\end{align*}
\]

**Type Expression Rules**

Judgments for type expressions just translate the external syntax for types into the internal representation:

\[
\begin{align*}
\vdash \text{bool : boolean} \quad \text{(Bool)} \\
\vdash \text{int : integer} \quad \text{(Int)} \\
\vdash \text{array of \ texp : array(t)} \quad \text{(Array)} \\
\vdash \text{pair \ texp_1 \ texp_2 : t_1 \times t_2} \quad \text{(Pair)}
\end{align*}
\]
When do two identifiers have the “same” type, or “compatible” types?

E.g., if a has type $t_1$, b has type $t_2$ and f has type $t_2 \rightarrow t_3$, how must $t_1$ and $t_2$ be related for these to make sense?

\[
a := b
\]

\[
f(a)
\]

To maintain type safety we must insist at a minimum that $t_1$ and $t_2$ are structurally equivalent.

Structural equivalence is defined inductively:

- Primitive types are equivalent iff they are exactly the same type.
- Cartesian product types are equivalent if their corresponding component types are equivalent. (Record field names are typically ignored.)
- Disjoint union types are equivalent if their corresponding component types are equivalent.
- Mapping types (arrays and functions) are the same if their domain and range types are the same.

Another way to say this: two types are equal if they have the same set of values.

Recursive types are a challenge. Are these two types structurally equivalent?

\[
type t1 = \{ a:int, b: POINTER TO t1 \};
\]

\[
type t2 = \{ a:int, b: POINTER TO t2 \};
\]

Intuitively yes, but it’s (a little) tricky for a type-checking algorithm to determine this!

Question of equivalence is more interesting if language has type names, which arise for two main reasons:

- As a convenient shorthand to avoid giving the full type each time. E.g.,

\[
function f(x:int * bool * real) : int * bool * real = ...
\]

\[
type t = int * bool * real
\]

\[
function f(x:t) : t = ...
\]

- As a way of improving program correctness by subdividing values into types according to their meaning within the program.

\[
type polar = \{ r:real, a:real \};
\]

\[
type rect = \{ x:real, y:real \};
\]

\[
function polar_add(x:polar,y:polar) : polar ...
\]

\[
function rect_add(x:rect,y:rect) : rect ...
\]

\[
var a:polar; c:rect;
\]

\[
a := (150.0,30.0) (* ok *)
\]

\[
polar_add(a,a) (* ok *)
\]

\[
c := a (* type error *)
\]

\[
rect_add(a,c) (* type error *)
\]

For this to be useful, some structurally equivalent types must be treated as inequivalent.

Simplistic idea: Two types are equivalent iff they have the same name.

Supports polar/rect distinction.

But pure name equivalence is very restrictive, e.g.:

\[
type ftemp = real
\]

\[
type ctemp = real
\]

\[
var x:ftemp, y:ftemp, z: ctemp;
\]

\[
x := y; (* ok *)
\]

\[
x := 10.0; (* probably ok *)
\]

\[
x := z; (* type error *)
\]

\[
x := 1.8 * z + 32.0; (* probably type error *)
\]

Different types now seem too distinct; can’t even convert from one form of real to another.
NAME EQUIVALENCE (CONTINUED)

Also: what about unnamed type expressions?

type t = int * int
procedure f(x: int * int) = ...
procedure g(x: t) = ...
var a: t = (3,4)
g(a); (* ok *)
f(a); (* ok or not ?? *)

Because of these problems with pure name equivalence, most languages use mixed solutions.

ML TYPE EQUIVALENCE

ML uses structural equivalence, except that each datatype declaration creates a new type unlike all others.

datatype polar = POLAR of real * real
datatype rect = RECT of real * real
val a = POLAR(1.0,2.0) and b = RECT(1.0,2.0)
if (a = b) ... (* type error *)

Note that the mandatory use of constructors makes it possible to uniquely identify the types of literals.

Note that a datatype need not declare a record:

datatype fahrenheit = F of real
datatype celsius = C of real
val a = F 150.0 and b = C 150.0
if (a = b) ... (* type error *)
fun convert(F x) = C(1.8 * x + 32.0) (* ok *)

For type abbreviation, ML offers the type declaration, which simply gives a new name for an existing type.

type centigrade = celsius
fun g(x: centigrade) = if x = b ... (* ok *)

C TYPE EQUIVALENCE

C uses structural equivalence for array and function types, but name equivalence for struct, union, and enum types. For example:

    char a[100];
    void f(char b[]);
    f(a); /* ok */

    struct polar{float x; float y;};
    struct rect{float x; float y;};
    struct polar a;
    struct rect b;
    a = b; /* type error */

A type defined by a typedef declaration is actually just an abbreviation for an existing type.

Note that this policy makes it easy to check equivalence of recursive types, which can only be built using structs.

    struct fred {int x; struct fred *y;} a;
    struct bill {int x; struct fred *y;} b;
    a = b; /* type error */

JAVA TYPE EQUIVALENCE

Java uses nearly strict name equivalence, where names are either:

• One of eight built-in primitive types (int, float, boolean, etc.), or
• Declared classes or interfaces (reference types).

The only non-trivial type expressions that can appear in a source program are array types, which are compared structurally, using name equivalence for the ultimate element type. Java has no mechanism for type abbreviations.

Java types form a subtyping hierarchy:

• If class A extends class B, then A is a subtype of B.
• If class A implements interface I, then A is a subtype of I.
• If numeric type t can be coerced to numeric type u without loss of precision, then t is a subtype of u.

If T1 is a subtype of T2, then a value of type T1 can be used wherever a value of T2 is expected.