Type Checking
Outline

• General properties of type systems

• Types in programming languages

• Notation for type rules
  - Logical rules of inference

• Common type rules
Static Checking

• Refers to the compile-time checking of programs in order to ensure that the semantic conditions of the language are being followed

Examples of static checks include:
  - Type checks
  - Flow-of-control checks
  - Uniqueness checks
  - Name-related checks
Static Checking (Cont.)

Flow-of-control checks: statements that cause flow of control to leave a construct must have some place where control can be transferred; e.g., break statements in C

Uniqueness checks: a language may dictate that in some contexts, an entity can be defined exactly once; e.g., identifier declarations, labels, values in case expressions

Name-related checks: Sometimes the same name must appear two or more times; e.g., in Ada a loop or block can have a name that must then appear both at the beginning and at the end
Types and Type Checking

• A *type* is a set of values together with a set of operations that can be performed on them.

• The purpose of *type checking* is to verify that operations performed on a value are in fact permissible.

• The type of an identifier is typically available from declarations, but we may have to keep track of the type of intermediate expressions.
Type Expressions and Type Constructors

A language usually provides a set of *base types* that it supports together with ways to construct other types using *type constructors*

Through *type expressions* we are able to represent types that are defined in a program
Type Expressions

- A base type is a type expression
- A type name (e.g., a record name) is a type expression
- A type constructor applied to type expressions is a type expression. E.g.,
  - **arrays**: If T is a type expression and I is a range of integers, then `array(I,T)` is a type expression
  - **records**: If T1, ..., Tn are type expressions and f1, ..., fn are field names, then `record((f1,T1),..., (fn, Tn))` is a type expression
  - **pointers**: If T is a type expression, then `pointer(T)` is a type expression
  - **functions**: If T1, ..., Tn, and T are type expressions, then so is `(T1,...,Tn) \rightarrow T`
Notions of Type Equivalence

Name equivalence: In many languages, e.g. Pascal, types can be given names. Name equivalence views each distinct name as a distinct type. So, two type expressions are name equivalent if and only if they are identical.

Structural equivalence: Two expressions are structurally equivalent if and only if they have the same structure; i.e., if they are formed by applying the same constructor to structurally equivalent type expressions.
Example of Type Equivalence

In the Pascal fragment

```pascal
type nextptr = ^node;
prevptr = ^node;
var  p : nextptr;
q : prevptr;
```

\( p \) is not name equivalent to \( q \),
but \( p \) and \( q \) are structurally equivalent.
Static Type Systems & Their Expressiveness

• A static type system enables a compiler to detect many common programming errors
• The cost is that some correct programs are disallowed
  - Some argue for dynamic type checking instead
  - Others argue for more expressive static type checking
  - But more expressive type systems are also more complex
Compile-time Representation of Types

• Need to represent type expressions in a way that is both easy to construct and easy to check

Approach 1: Type Graphs
- Basic types can have predefined “internal values”, e.g., small integer values
- Named types can be represented using a pointer into a hash table
- Composite type expressions: the node for $f(T_1, \ldots, T_n)$ contains a value representing the type constructor $f$, and pointers to the nodes for the expressions $T_1, \ldots, T_n$
Example:

```pascal
var x, y : array[1..42] of integer;
```
## Approach 2: Type Encodings

Basic types use a predefined encoding of the low-order bits.

<table>
<thead>
<tr>
<th>BASIC TYPE</th>
<th>ENCODING</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>0000</td>
</tr>
<tr>
<td>char</td>
<td>0001</td>
</tr>
<tr>
<td>integer</td>
<td>0010</td>
</tr>
</tbody>
</table>

The encoding of a type expression $\text{op}(T)$ is obtained by concatenating the bits encoding $\text{op}$ to the left of the encoding of $T$. E.g.:

<table>
<thead>
<tr>
<th>TYPE EXPRESSION</th>
<th>ENCODING</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>00 00 00 0001</td>
</tr>
<tr>
<td>array(char)</td>
<td>00 00 01 0001</td>
</tr>
<tr>
<td>ptr(array(char))</td>
<td>00 10 01 0001</td>
</tr>
<tr>
<td>ptr(ptr(array(char)))</td>
<td>10 10 01 0001</td>
</tr>
</tbody>
</table>
Compile-Time Representation of Types: Notes

• Type encodings are simple and efficient.
• On the other hand, named types and type constructors that take more than one type expression as argument are hard to represent as encodings. Also, recursive types cannot be represented directly.

• Recursive types (e.g., lists, trees) are not a problem for type graphs: the graph simply contains a cycle.
Types in an Example Programming Language

• Let’s assume that types are:
  - integers & floats (base types)
  - arrays of a base type
  - booleans (used in conditional expressions)

• The user declares types for all identifiers

• The compiler infers types for expressions
  - Infers a type for every expression
Type Checking and Type Inference

*Type Checking* is the process of verifying fully typed programs.

*Type Inference* is the process of filling in missing type information.

- The two are different, but are often used interchangeably
Rules of Inference

• We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions (for the lexer)
  - Context-free grammars (for the parser)

• The appropriate formalism for type checking is logical rules of inference
Why Rules of Inference?

• Inference rules have the form
  \[ \text{If Hypothesis is true, then Conclusion is true} \]

• Type checking computes via reasoning
  \[ \text{If } E_1 \text{ and } E_2 \text{ have certain types, then } E_3 \text{ has a certain type} \]

• Rules of inference are a compact notation for "If-Then" statements
From English to an Inference Rule

• The notation is easy to read (with practice)

• Start with a simplified system and gradually add features

• Building blocks:
  - Symbol $\land$ is “and”
  - Symbol $\Rightarrow$ is “if-then”
  - $x:T$ is “$x$ has type $T$”
From English to an Inference Rule (2)

If \( e_1 \) has type \( \text{int} \) and \( e_2 \) has type \( \text{int} \),
then \( e_1 + e_2 \) has type \( \text{int} \)

\[(e_1 \text{ has type } \text{int} \land e_2 \text{ has type } \text{int}) \Rightarrow (e_1 + e_2 \text{ has type } \text{int})\]

\[(e_1 : \text{int} \land e_2 : \text{int}) \Rightarrow e_1 + e_2 : \text{int}\]
From English to an Inference Rule (3)

The statement
\[(e_1: \text{int} \land e_2: \text{int}) \Rightarrow e_1 + e_2: \text{int}\]
is a special case of
\[\text{Hypothesis}_1 \land \ldots \land \text{Hypothesis}_n \Rightarrow \text{Conclusion}\]

This is an inference rule
**Notation for Inference Rules**

- By tradition inference rules are written
  
  \[ \vdash \text{Hypothesis}_1 \quad \vdash \text{Hypothesis}_n \]
  \[ \vdash \text{Conclusion} \]

- Type rules have hypotheses and conclusions of the form:
  
  \[ \vdash e : T \]

- \( \vdash \) means “it is provable that . . .”
Two Rules

\[
\begin{align*}
\text{i is an integer} & \quad \text{[Int]} \\
\therefore i : \text{int} & \\
\hline
\text{\[ e_1 : \text{int} \quad e_2 : \text{int} \] [Add]} & \\
\therefore e_1 + e_2 : \text{int} &
\end{align*}
\]
Two Rules (Cont.)

- These rules give templates describing how to type integers and + expressions

- By filling in the templates, we can produce complete typings for expressions
Example: $1 + 2$

\[
\frac{1 \text{ is an integer}}{\vdash 1 : \text{int}} \quad \frac{2 \text{ is an integer}}{\vdash 2 : \text{int}} \quad \vdash 1 + 2 : \text{int}
\]
Soundness

• A type system is *sound* if
  - Whenever $\vdash e : T$
  - Then $e$ evaluates to a value of type $T$

• We only want sound rules
  - But some sound rules are better than others
  - Consider the rule:
    
    $i$ is an integer
    
    $\vdash i : \text{number}$

  - This rule loses some information
Type Checking Proofs

• Type checking proves facts $e : T$
  - Proof is on the structure of the AST
  - Proof has the shape of the AST
  - One type rule is used for each kind of AST node

• In the type rule used for a node $e$:
  - Hypotheses are the proofs of types of $e$’s subexpressions
  - Conclusion is the type of $e$

• Types are computed in a bottom-up pass over the AST
Rules for Constants

\[ \vdash \text{true : bool} \quad [\text{Bool}] \quad \vdash \text{false : bool} \quad [\text{Bool}] \]

\[ f \text{ is a floating point number} \quad \vdash f : \text{float} \quad [\text{Float}] \]
Two More Rules

\[
\frac{\vdash e : \text{bool}}{\vdash \neg e : \text{bool}} \quad \text{[Not]}
\]

\[
\frac{\vdash e_1 : \text{bool} \quad \vdash e_2 : \text{T}}{\vdash \text{while } e_1 \text{ do } e_2 : \text{T}} \quad \text{[While]}
\]
A Problem

• What is the type of a variable reference?

\[
\frac{x \text{ is an identifier}}{\vdash x : ? \quad [\text{Var}]}
\]

• See the problem?
• The local, structural rule does not carry enough information to give \( x \) a type
A Solution

• Put more information in the rules!

• A type environment gives types for free variables
  - A type environment is a function from Identifiers to Types
  - A variable is free in an expression if it is not defined within the expression
Type Environments

Let $E$ be a function from Identifiers to Types.

The sentence $E \vdash e : T$ is read:

Under the assumption that variables have the types given by $E$, it is provable that the expression $e$ has the type $T$. 
Modified Rules

The type environment is added to the earlier rules:

\[
\frac{i \text{ is an integer}}{E \vdash i : \text{int}} \quad \text{[Int]}
\]

\[
\frac{E \vdash e_1 : \text{int} \quad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}} \quad \text{[Add]}
\]
New Rules

And we can now write a rule for variables:

\[
\frac{E(x) = T}{E \vdash x : T} \quad [\text{Var}]
\]
Type Checking of Expressions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \to id$</td>
<td>{ if (declared(id.name)) then E.type := lookup(id.name).type else E.type := error(); }</td>
</tr>
<tr>
<td>$E \to int$</td>
<td>{ E.type := integer; }</td>
</tr>
<tr>
<td>$E \to E1 + E2$</td>
<td>{ if (E1.type == integer AND E2.type == integer) then E.type := integer; else E.type := error(); }</td>
</tr>
</tbody>
</table>
May have automatic *type coercion*, e.g.

<table>
<thead>
<tr>
<th>E1.type</th>
<th>E2.type</th>
<th>E.type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>integer</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>float</td>
<td>integer</td>
<td>float</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>float</td>
</tr>
</tbody>
</table>
Type Checking of Statements: Assignment

Semantic Rules:

\[ S \rightarrow \text{Lval} := \text{Rval} \quad \{ \text{check\_types}(\text{Lval}\.\text{type},\text{Rval}\.\text{type}) \} \]

Note that in general Lval can be a variable or it may be a more complicated expression, e.g., a dereferenced pointer, an array element, a record field, etc.

Type checking involves ensuring that:
- Lval is a type that can be assigned to, e.g., it is not a function or a procedure
- the types of Lval and Rval are “compatible”, i.e., that the language rules provide for coercion of the type of Rval to the type of Lval
Type Checking of Statements: Loops, Conditionals

Semantic Rules:

Loop → while E do S   {check_types(E.type, bool)}

Cond → if E then S1 else S2
        {check_types(E.type, bool)}