IT 302 Compiler Design

TyPe ChEcKiNg, TyPe SyStEm, TypE wAr,

Type System

- A type is a set of values and operations on those values
- A language's type system specifies which operations are valid for a type
- The aim of type checking is to ensure that operations are used on the variable/expressions of the correct types.
- Type errors arise when operations are performed on values that do not support that operation.

Why Do We Need Type Systems? Consider the assembly language fragment

addi $r1, $r2, $r3

What are the types of $r1, $r2, $r3?

- If $r2 = 5 (integer) and $r3 = 10 (integer) → fine.
- But if $r2 holds a **float** or a **memory address**, the result of addi won't make sense.

The same operation (`addi`) might give **wrong or meaningless results** if used with data of the wrong type.

Contd.,

Concept	Assembly Language	High-Level Language
Type info	Not present	Declared and checked
Error detection	Only at runtime (maybe)	Detected at compile time
Purpose	Performance, flexibility	Safety, correctness

- In **assembly**, the machine doesn’t know or care about types —it only sees bits
- But in **high-level languages**, the **type system** ensures operations make sense.
- We need type systems because they **prevent invalid operations** and **ensure data is used correctly**.
- Without them, as in assembly, the computer has no idea what kind of data it’s manipulating.

Types and Operations

- Certain operations are legal for values of each type
  - It doesn't make sense to add a function pointer and an integer in C
    - A **function pointer** points to the memory address of a function.
    - An **integer** is just a numeric value.
    - Adding them together doesn't have a logical meaning (what does "add 3 to a function address" even mean?
  - It does make sense to add two integers
  - But both have the same assembly language implementation!

addi $r1, $r2, $r3

This instruction doesn't know or care **what** $r2 and $r3 represent — it just adds bits.

Type Systems

- *A language's type system specifies which operations are valid for which types*
- The goal of type checking is to ensure that operations are used with the correct types
- Type systems provide a brief validation of the semantic checking rules

What Can Types do For Us?

- Can detect certain kinds of errors
  - Memory errors:
    - Reading from an invalid pointer, etc.
  - Violation of abstraction boundaries.
- Allow for a more efficient compilation of programs

Type Checking Overview

- Three kinds of languages:

- Statically typed: All or almost all checking of types is done as part of compilation (C, ML, Java)
- Dynamically typed: Almost all checking of types is done as part of program execution (python, Scheme, Prolog)
- Untyped: No type checking - the system **doesn’t care** about data types. (machine code)

Types in an Example Programming Language

- Let's assume that types are:
  - integers & floats (base types)
  - arrays of a base types
  - booleans (used in conditional expressions)
- The user declares types for all identifiers
- The compiler infers (determines) types for expressions
  - Infers a type for every expression

```
int a;
float b;
bool flag;
```

```
a = b + 5;
```

Here:

- `b` is a float
- `5` is an integer

The compiler infers the result as **float** automatically

Type Checking and Type Inference

- Type Checking is the process of verifying fully typed programs
  - int x = 10;
  - float y = "text"; // ❌ Type mismatch error

- Type Inference is the process of filling in missing type information
  - If a variable or expression doesn't have an explicit type, the compiler figures it out from usage.

```
x = 10        # Compiler infers x as int
y = x + 2.5   # Infers y as float
```

- *The two are different, but are often used interchangeably*

  - **Type checking**: verifies types → "Are the types correct?"
  - **Type inference**: fills in missing types → "What should the types be?"
  - Both are part of the **semantic analysis** phase.

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
  - **Rules of inference** are logical rules that describe **how to deduce conclusions from known facts.**

Why Rules of Inference?

- Inference rules have the form like logical "**If–Then**" statements:

  If Hypothesis is true, then Conclusion is true

- Type checking computes via reasoning

  If E1and E2 have certain types (integer),

  then E3 has a certain type (integer)

- Instead of writing many "if-then" statements in English,

- we use compact inference rules — a formal mathematical way to describe them.

From English to an Inference Rule

- **The Goal:**
- To **translate English reasoning** ("if X and Y are true, then Z is true") into a **formal rule of inference** that the compiler can understand.
- The notation is easy to read (with practice)

- Start with a simplified system and gradually add features

- Building blocks
  - Symbol $\wedge$ is "and" (A $\wedge$ B → both A and B are true)
  - Symbol $\Rightarrow$is "if-then" (A $\Rightarrow$ B → If A is true, then B is true)
  - x:T is "x has type T" (a : int → a is an integer variable)

From English to an Inference Rule (2)

If e_1 has type int and e_2 has type int,
then $e_1 + e_2$ has type int

$(e_1$ has type int $\wedge$ e_2 has type int$)$ $\Rightarrow$
$e_1 + e_2$ has type int

$(e_1\text{: int} \wedge e_2\text{: int}) \Rightarrow e_1 + e_2\text{: int}$

From English to an Inference Rule (3)

The statement

$$(e_1\!: \text{int} \wedge e_2\!: \text{int}) \Rightarrow e_1 + e_2\!: \text{int}$$

is a special case of

$$\text{Hypothesis}_1 \wedge \ldots \wedge \text{Hypothesis}_n \Rightarrow \text{Conclusion}$$

This is an inference rule

Notation for Inference Rules

- By tradition inference rules are written

$$\frac{\vdash \text{Hypothesis}_1 \quad \dots \quad \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$

- Type rules have hypotheses and conclusions of the form:

$$\vdash e : T$$

- $\vdash$ means "it is provable that . . ."

The symbol $\vdash$ is called the **"turnstile"** (pronounced *turn-style*).

Two Rules

- Meaning:
  If i is an integer constant (like 3, 7, 42),
  then we can conclude that i has type int.
- It's a base rule — no condition is needed.

$$\frac{i \text{ is an integer}}{\vdash i : \text{int}} \;[\text{Int}]$$

$$\frac{\vdash e_1 : \text{int} \qquad \vdash e_2 : \text{int}}{\vdash e_1 + e_2 : \text{int}} \;[\text{Add}]$$

- If both expressions e_1 and e_2 have type **int**,
  then the result of $e_1 + e_2$ is also **int**.
- This ensures that addition is only valid for integers.

Example: 1 + 2

$$\dfrac{\dfrac{1 \text{ is an integer}}{\vdash 1 : \text{int}} \qquad \dfrac{2 \text{ is an integer}}{\vdash 2 : \text{int}}}{\vdash 1 + 2 : \text{int}}$$

- **Meaning:**
  - The compiler first confirms that both operands are integers.
  - Then it concludes that the **whole expression `1 + 2`** is of type **`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:

$$\frac{i \text{ is an integer}}{\vdash i : \text{number}}$$

- A type system is sound if it guarantees that:
- Whenever a program is type-correct, it will not produce type errors when executed.

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

- Constants are the simplest elements in a program.
- The compiler must assign them a fixed type immediately — no computation or dependency is needed.

$$\frac{}{\vdash \text{false} : \text{bool}} \ [\text{Bool}]$$

The constant **`false`** is of type **`bool`** **(boolean)**.

$$\frac{f \text{ is a floating point number}}{\vdash f : \text{float}} \ [\text{Float}]$$

If `f` is a floating-point literal (e.g., `3.14`, `0.5`), then `f` has type **`float`**.

Two More Rules

$$\frac{\vdash e : bool}{\vdash not\ e : bool} \text{[Not]}$$

- If an expression e has type boolean, then not e (its logical negation) also has type boolean.
- Ex: e = true → not e → false (still boolean).

$$\frac{\begin{array}{c}\vdash e_1 : bool \\ \vdash e_2 : T\end{array}}{\vdash while\ e_1\ do\ e_2 : T} \text{[While]}$$

- e_1 is the **loop condition**, which must be a **boolean**.
- e_2 is the **body** of the loop, which can be of any type T.
- *The overall while expression has type **T**.*

A Problem

- What is the type of a variable reference?

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

- The local, structural rule does not carry enough information to give x a type

- If x is a variable (identifier),
  the rule doesn't tell us **what its type is** — only that it exists

- This simple local rule cannot determine types by itself.
- The compiler needs **extra information** — typically from a **symbol table**, which stores variable types.

A Solution

- Put more information in the rules!
- A Type Environment (usually denoted by E or Γ) is a mapping from variable identifiers to their types.

$E: Identifiers \rightarrow \text{Types}$

Ex: $E = \{x: int, y: float, flag: bool\}$

- This means:
  - variable x has type int
  - variable y has type float
  - variable flag has type bool

Contd.,

- Let E be the function from identifiers to types

$$E \vdash e : T$$

Read as:"Under the assumption that all variables have the types given by E, the expression e has type T."

- This statement represents a **type judgment** — it's what the compiler tries to prove during type checking.
- Example: If $E = \{x : int, y : int\}$,
  then $E \vdash x + y : int$ means "x + y is an integer expression under E."

Modified Rules

- The type environment is added to the earlier rules:

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

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

Earlier rules like `[Int]` and `[Add]` are now extended to include `E`

New Rules

And we can write new rules:

$$\frac{E(x) = T}{E \vdash x : T} \text{[Var]}$$

- If the environment `E` says that variable `x` has type `T`,
- Then we can conclude that expression `x` is of type **`T`**.

Type Checking of Expressions

Production	Semantic Rules
$E \rightarrow id$	{ if (declared(id.name)) then E.type := lookup(id.name).type else E.type := error(); }
$E \rightarrow int$	{ E.type := integer; }
$E \rightarrow E1 + E2$	{ if (E1.type == integer AND E2.type == integer) then E.type := integer; else E.type := error(); }

- Checks whether the variable `id` exists in the **symbol table**.
- If it does, fetch its type.
- If not declared, signal an **error**

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**)}

Rule for while Loops:

- E is the **loop condition**.
- The condition must have **a Boolean type** (true or false).
- The loop body S is type-checked independently

Rule for if Statements:

- E is the conditional expression.
- Must be a boolean.
- Both branches (S1, S2) are type-checked separately.

Type Checking of Statements: Assignment

Semantic Rules:

$S \rightarrow Lval := Rval$ {check_types(Lval.type,Rval.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

The compiler allows assignment when:

- Types are **exactly the same**, or
- There exists a **legal coercion** (automatic conversion).