Type Systems

Shawn Rasheed
S.Rasheed@massey.ac.nz

June 22, 2022

"Industry is ready and waiting for more graduates educated in the principles of programming languages."

— B. Zorn and T. Ball (Microsoft), Comm. of the ACM

Context

- Abstract machines
- Foundations (expressiveness and computability)
- Programming languages (syntax, semantics and pragmatics)
- Bindings: names and environments
- Memory management
- Control
- Data
  - **Type systems**
- Paradigms
  - Imperative (procedural, object-oriented...)
  - Declarative (logic, functional...)

What are types?

- A way to avoid paradoxes in naive set theory (Russell 1903)
- A concept in programming languages to classify constructs in programs
- Facilitates thinking and communicating about programs
- First used in Fortran for efficient numeric calculations (1950s)

Uses for types

- Predicting or preventing execution errors
  - Check program consistency with programmer intent
  - Identifies nonsensical programs
  - Prevents expressing or executing invalid program
- Why?
  - Safety
- An abstraction for structuring large programs
- Types are documentation
- Efficiency

Predicting errors

Problem

- Context-sensitive constraints

```
// e.g. In Java, declare variable
// before use
num = 42;
int num = 42;
```

- Limits of context-free grammars

Solutions

- Testing
- Runtime checks
- Compile-time methods (e.g. formal methods)

Tradeoffs

Testing	Runtime	Compile-time
Some inputs	Some inputs	All inputs
Precise	Precise	Overapproximates
	Overhead	No runtime overhead
	Permissive	Conservative

Examples

```
// rejected in static type checking
// because analysis overapproximates
if <complex test> then 5
else <type error>
```

```
# runtime check in Python
# fails on one branch
def f(param):
  if param == "yes":
     return "a" + 1 # type error
  else:
     return 1
```

Safety

- Meaning of safety
- A program is type safe if it cannot violate the language's type abstractions (e.g.: invoke operations on wrong types)
- Trapped errors at compile time / runtime in safe language (e.g. JavaScript, Haskell, Java)
- Untrapped errors in unsafe languages (e.g. out of bounds array access in C)

Example

Out-of bounds write

Weakness exploited in security attacks

```
int f[2];
f[10000000] = 42; // Error can crash program
```

Types

The most widely used lightweight formal method for predicting program errors

Types in programs

- Data and behaviour have types (i.e. values and functions)
- `X = 4` implies that `X` is a numeric type
- `int n = 4;` declares that `n` is an integer type
- `a + b`. Depends on types of a and b
- Meaning of an operation depends on input types, i.e. context. E.g. a+b (string concatenation or addition?)

Example types and values

- Void type. Absence of a type and has only one value
- Null: does not hold a value
- Product and sum types

Kinds of type systems

- Statically typed
- Dynamically typed
- Gradual typing

Different interpretations

- Denotational: set of values
- Structural: builtin primitive types, composites from simpler types
- Abstraction-based: interface providing set of operations

Elements of type systems

Defines types and their use for a programming languages

- Objects in a program (e.g. variable, record fields, functions) have types
- Rules
  - Type equivalence
  - Type compatibility
  - Type checking/inference

Polymorphism

- Monomorphism
- Polymorphism: same code works for different types
- Types of polymorphism
    - Parametric polymorphism (Type parameters. e.g. Java generics)
    - Subtype polymorphism (Extending supertype. e.g. Java)

Type equivalence

- Used for type checking
- Two approaches:
  - Structural: compare structure recursively. For records, name and types of fields.
  - Name equivalence. Types with different names are different (e.g. Java)

Type compatibility

- Is combining two values valid?
- What is combining?
  - Assignment (both sides)
  - Operators (operands with operator and each other.)
  - Functions: arguments and formal parameters
- Compatibility for different types
  - Assign subtype to supertype
  - Collection of same type compatible even if length differs

Type conversion

- Explicit conversion (casting)
- Three cases
  - Structurally equivalent, no code generation
  - Types have different set of values but same representation in memory: may need check that value is of target type
  - Different low level representation: need code for conversion
- Coercion is implicit (runtime)
- Coercion can be lossy

Formally defining type systems

- Implemented in compilers/runtimes
- Can be formally described.
- Other formalisms in compilers
  - regular expressions
  - context-free grammars
- Typing rules

Inference rules

- Logical rules of inference
- If we know the premises, we know the conclusion
- Type checking is reasoning. e.g. if we know the types of $e1$ and $e2$, then we know the type of $e3$
- Other notation:
  $e : t$ is read as e has type t
  $\Gamma \vdash e : t$ read as "environment Γ shows or proves e has type t"

$$\frac{P1 \quad P2 \quad ...}{C}$$

Properties of type systems

- Soundness: if $e : t$, then the expression e evaluates to the type t
- Precision: rules can be imprecise, but still sound
- Progress: can take another step in evaluating (unless expression is a value)
- Preservation: Evaluation to the next step does not change type
- Type safe if progress and preservation can be established

Type checking

- Determine if an expression is ill-typed or well-typed
- Ensures program obeys type compatibility rules.
- Checking proves facts, $e : t$
- Bottom-up pass over AST (abstract syntax tree)
- Premises are proofs of types of subexpressions

Type inference

- Types without annotations
- Infer the types of expressions
- Parses program (AST)
- Assign type variables to nodes
- Generate constraints
- Unification to find solution