Static Type Checking
what is static type checking

• Static type checking is the process of ensuring that a program is **well-typed**

• Central idea: types constrain the behavior of a program
  • A *well-typed* program is one whose run-time behavior stays within that set of constraints

• Example: an expression that is well-typed and has type `int` will, at runtime, produce a value that is an `int`
syntactic type checking

- Static type checking is based on the **syntax** of the expressions being typed as well as the **context** in which the type checking happens.

- Is \( x = y + z \) well-typed?

- Depends on context: are \( x, y, \) and \( z \) all the same type, and types that can have arithmetic operations performed on them?
syntactic type checking

• Being **well-typed** is an inductive property
• Basic idea: assign a type to every expression
  • If you can assign a type to an expression, it is well-typed
• Type check expressions and statements by breaking them down into smaller components
  • Find the types of smaller expressions
  • Combine types of smaller expressions to assign a type to the larger expression

Is: \( x = (a + b) + c \) well typed?  

Is: \( x \) well typed?  

Is: \( a + b \) well typed?  

Is: \( c \) well typed?  

Is: \( a \) well typed?  

Is: \( b \) well typed?
• Being **well-typed** is an inductive property
• Basic idea: assign a type to every expression
  • If you can assign a type to an expression, it is well-typed
• Type check expressions and statements by breaking them down into smaller components
  • Find the types of smaller expressions
  • Combine types of smaller expressions to assign a type to the larger expression
type rules

- For each syntactic form in your language
  - Expressions
  - Statements
- Describe the rules under which the form is well typed based on the types of its sub-components
- Structure captured by AST!
walk the AST

- For each syntactic structure that can have a type, add a type field to the AST
- Perform a post-order walk of the AST to assign types to each node in the AST

- Base cases:
  - Variables: get types from symbol table
  - Literals: get types from node type

Is: $x = (a + b) + c$
Well typed? int

Is: $x$
Well typed? int

Is: $a + b$
Well typed? int

Is: $c$
Well typed? int

Is: $a$
Well typed? int

Is: $b$
Well typed? int
walk the AST

- For each syntactic structure that can have a type, add a type field to the AST
- Perform a post-order walk of the AST to assign types to each node in the AST
- Inductive cases:
  - Expressions: compare types of component sub-expressions
  - Assignment: compare LHS and RHS

Is:
\[ x = (a + b) + c \]
well typed?

Is:
\[ x \]
well typed?

Is:
\[ a + b \]
well typed?

Is:
\[ c \]
well typed?

Is:
\[ a \]
well typed?

Is:
\[ b \]
well typed?

\[ \text{int} \]
walk the AST

• For each syntactic structure that can have a type, add a type field to the AST
• Perform a post-order walk of the AST to assign types to each node in the AST

• Inductive cases:
  • Expressions: compare types of component sub-expressions
  • Assignment: compare LHS and RHS
  • Conditionals: compare LHS and RHS
walk the AST

- For each syntactic structure that can have a type, add a type field to the AST
- Perform a post-order walk of the AST to assign types to each node in the AST

Inductive cases:
- Function calls: compare types of sub-expressions to argument types in symbol table
- Note: type assigned to function call should be the \textit{return} type of the function!
For each syntactic structure that can have a type, add a type field to the AST
Perform a post-order walk of the AST to assign types to each node in the AST

Inductive cases:
- Function calls: compare types of sub-expressions to argument types in symbol table
- Return statements: compare return expression type to return type of function in symbol table
walk the AST

• For each syntactic structure that can have a type, add a type field to the AST
• Perform a post-order walk of the AST to assign types to each node in the AST
• If any node cannot be typed, return an error!

Is: return (x + y) well typed?

Is: foo well typed?

Is: x -> int well typed?

Is: x + y well typed?

Is: y well typed?

Is: int well typed?