Constraint Analysis

(a.k.a. Contextual Analysis or Analysis of Static Semantics)

1

Specification of a Programming Language

- Syntax (form)
  - basic language symbols (or tokens)
  - allowed structure of symbols to form programs
  - specified by a context-free grammar
- Contextual Constraints
  - program rules and restrictions that can not be specified in a context-free grammar
  - consist primarily of type and scope rules
- Semantics (meaning)
  - behavior of program when it is run on a machine
  - usually specified informally

2

Syntax Analysis versus Constraint Analysis

- Syntax analysis verifies that a program conforms to the formal syntax of the language as defined by a context-free grammar.
- Syntax analysis is performed by the parser.
- Constraint analysis verifies that a program conforms to the additional language rules and requirements.
  (usually expressed informally)
- Constraint analysis is performed partly by the parser using helper classes `IdTable`, `LoopContext`, and `SubprogramContext`, and partly by the abstract syntax trees in methods named `checkConstraints()`.

3

Categories of Constraints

Constraints fall into three general categories

- Scope rules: Rules associated with declarations and applied occurrences of identifiers.
- Type rules: Rules associated with the types of expressions and their use in certain contexts.
- Miscellaneous rules: Language constraints that do not fall into either of the above categories. (Some of these rules represent internal errors within the compiler that might have occurred during parsing.)

4

Scope Rules in CPRL

The scope rules for CPRL are fairly simple:

- Every user-defined identifier (constant, variable, type name, subprogram name, etc.) must be declared. When we encounter an applied occurrence of an identifier, we must be able to discover its declaration and associate the declaration with the identifier.
- All identifiers appearing in declarations must be unique within their scope. In other words, the same identifier must not be used in two different declarations within the same scope.

CPRL has a what is known as a ***flat block structure***.
Declarations are either global in scope or local to a subprogram.

5

Scope Analysis
(a.k.a. Identification)

- Scope analysis is the process of verifying the scope rules.
- For CPRL, scope analysis is implemented within the parser using class `IdTable`.
- Class `IdTable` is capable of handling nested scopes of two levels as required by CPRL, but it could easily be extended to handle arbitrary nesting of scopes.

6

Scope Analysis Using Class `IdTable`

- When an identifier is declared, the parser will attempt to add a reference to its declaration to `IdTable` within the current scope.
  - throws an exception if a declaration with the same name (same token text) has been previously added in the current scope
- When an applied occurrence of an identifier is encountered (e.g., in a statement), the parser will
  - check that the identifier has been declared
  - store a reference to the identifier's declaration as part of the AST where the identifier is used

7

Static Typing

- CPRL is a statically-typed language.
  - every variable and expression has a type
  - type compatibility is a static property (i.e., it can be determined by the compiler)
- Type rules for CPRL define how and where certain types can be used.
- We define type rules in any context where a variable or expression can appear.

8

Examples of Types Rule in CPRL

- For an assignment statement, the type of the variable on the left side of the assignment symbol must be the same as the type of the expression on the right side.
  Note that some languages do not require equality here, only that the types be assignment compatible. For example, in C it is perfectly acceptable to assign a character to an integer variable.
- For a negation expression, the operand must have type Integer, and the result of a negation expression is type Integer.

9

Constraint Analysis

- Constraint analysis is the process of verifying that all constraint rules have been satisfied.
- For CPRL, most type and miscellaneous rules are verified using the method `checkConstraints()` in the AST classes.
- Even AST classes that do not have associated constraints will implement the method `checkConstraints()` if they contain references to objects of other AST classes. They simply call `checkConstraints()` on those other AST objects.

10

Example: Constraint Checking for Class `Program`

```
override fun checkConstraints()
  {
    declPart.checkConstraints()
    stmtPart.checkConstraints()
  }
```


11

Example: Constraint Checking for Class `DeclarativePart`

```
override fun checkConstraints()
  {
    declPart.checkConstraints()
    stmtPart.checkConstraints()
  }
```


12

Example: Constraint Checking for Class `StatementPart`

```
override fun checkConstraints()
  {
    for (stmt in statements)
        stmt.checkConstraints()
  }
```


13

Constraint Rules for CPRL/0

- Adding Expression and Multiplying Expression
  - Type Rule: Both operands must have type Integer.
  - Miscellaneous Rule: The result has type Integer.
- Assignment Statement
  - Type Rule: The variable (on the left side of the assignment) and the expression (on the right side) must have the same type.
- Exit Statement
  - Type Rule: If a "when" expression exists, it must have type Boolean.
  - Miscellaneous Rule: The exit statement must be nested within a loop statement.*

*Handled by the parser using `LoopContext`.

14

Constraint Rules for CPRL/0 (continued)

- If Statement
  - Type Rule: The expression must have type Boolean.
  - Type Rule: The expression for any "elsif" clauses must have type Boolean.
- Read Statement
  - Type Rule: The variable must have either type Integer or type Char.
- Logical Expression
  - Type Rule: Both operands must have type Boolean.
  - Miscellaneous Rule: The result has type Boolean.

15

Constraint Rules for CPRL/0 (continued)

- Loop Statement
  - Type Rule: If a "while" expression exists, it must have type Boolean.
- Negation Expression
  - Type Rule: The operand must have type Integer.
  - Miscellaneous Rule: The result has type Integer.
- Not Expression
  - Type Rule: The operand must have type Boolean.
  - Miscellaneous Rule: The result has type Boolean.

16

Constraint Rules for CPRL/0 (continued)

- Relational Expression
  - Type Rule: Both operands must have the same type.
  - Type Rule: Only scalar types (Integer, Char, or Boolean) are allowed for operands. (For example, in CPRL, you can't have a relational expression where both operands are arrays or string literals.)
  - Miscellaneous Rule: The result has type Boolean.
- Variable Declaration and Single Variable Declaration
  - Type Rule: The type should be Integer, Boolean, Char, or a user-defined array type.

17

Constraint Rules for CPRL/0 (continued)

- Constant Declaration and Constant Value
  - Miscellaneous Rule: If the literal value has type Integer, then it must be able to be converted to an integer value on the CPRL virtual machine. (In other words, check that Integer.parseInt() will not fail.) If the check fails for a constant declaration, then set the literal's value to a valid value for Integer in order to prevent additional error messages every time that the constant declaration is used.
- Write Statement
  - Miscellaneous Rule: For a "write" statement (but not "writeln"), there should be at least one expression.

18

Constraint Rules for Subprograms

- Return Statement
  - Type Rule: If the statement returns a value for a function, then the type of expression being returned must be the same as the function return type.
  - Miscellaneous Rule: If the return statement returns a value, then the return statement must be nested within a function declaration.
  - Miscellaneous Rule: If the return statement is nested within a function, then it must return a value.
  - Miscellaneous Rule: The return statement must be nested within a subprogram.*

*Handled by the parser using `SubprogramContext`.

19

Constraint Rules for Subprograms (continued)

- Function Declaration
  - Miscellaneous Rule: There should be no `var` parameters.
  - Miscellaneous Rule: There should be at least one return statement.
- Subprogram Call (for both procedures and functions)
  - Type Rule: The number of actual parameters should be the same as the number of formal parameters, and each corresponding pair of parameter types should match.
- Procedure Call
  - Miscellaneous Rule: If the formal parameter is a `var` parameter, then the actual parameter must be a named value (not a arbitrary expression).

20

Constraint Rules for Arrays

- Array Type Declaration
  - Type Rule: The constant value specifying the number of items in the array must have type Integer, and the associated value must be a positive number.
- Variable (and therefore also for Named Value)
  - Each index expression must have type Integer.
  - Index expressions are permitted only for variables with an array type.

21

Example: Constraint Checking for Class `AssignmentStmt`

```
override fun checkConstraints()
  {
    try
      {
        expr.checkConstraints()
        var.checkConstraints()

        if (!matchTypes(variable.type, expr.type)
          {
            val errorMsg = "Type mismatch ..."
            throw error(assignPosition, errorMsg)
          }
      }
    catch (e : ConstraintException)
      {
        ErrorHandler.reportError(e)
      }
  }
```


22

Example: Constraint Checking for Class `NegationExpr`

```
override fun checkConstraints()
  {
    try
      {
        operand.checkConstraints()

        // unary +/- can only be applied to an integer expression
        if (operand.type != Type.Integer)
          {
            String errorMsg = "Expression following ..."
            throw error(operand.position, errorMsg)
          }
      }
    catch (e : ConstraintException)
      {
        ErrorHandler.reportError(e)
      }
  }
```


23