Bab 11

Type Checking dan Semantic Analysis

11.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep semantic analysis dan perannya dalam kompilator
2. Menjelaskan sistem tipe dan aturan type checking
3. Mengimplementasikan type checker untuk ekspresi aritmatika
4. Memahami type inference dan type compatibility
5. Mengimplementasikan semantic error detection dan reporting
6. Membedakan static type checking dan dynamic type checking

11.2 Pendahuluan

Dalam proyek compiler subset C, type checking memverifikasi bahwa operasi dilakukan pada tipe yang kompatibel (`int`, `float`) dan bahwa setiap identifier merujuk ke deklarasi yang valid. Input: AST proyek (Bab 9, Bagian **??**) dan symbol table proyek (Bab 10). Type checker dipanggil setelah scope resolution, sebelum IR generation (Bab 12).

Setelah fase syntax analysis menghasilkan Abstract Syntax Tree (AST), kompilator perlu memastikan bahwa program tidak hanya valid secara sintaksis, tetapi juga secara semantik. Semantic analysis adalah fase yang memverifikasi bahwa program memenuhi aturan semantik bahasa pemrograman.

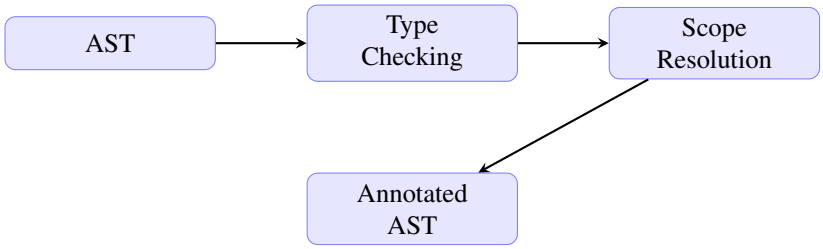
Menurut sumber dari Nguyen Thanh Vu:

> "Type checking: operator operands must be type-compatible. Return types match declared types. Implicit/explicit conversions. Semantic analysis ensures that the parse tree makes sense under language rules."[1]

Semantic analysis bertanggung jawab untuk memeriksa berbagai aspek semantik program:

- **Type Checking**: Memastikan operasi dilakukan pada tipe yang kompatibel
- **Scope Resolution**: Memastikan setiap identifier merujuk ke deklarasi yang valid
- **Name Resolution**: Menyelesaikan referensi variabel, fungsi, dan tipe
- **Contextual Checks**: Memeriksa aturan spesifik bahasa (misalnya: break hanya dalam loop, return type match, dll.)

Gambar 11.1 menunjukkan proses semantic analysis.



Gambar 11.1: Proses semantic analysis

11.2.1 Input dan Output Semantic Analysis

Semantic analyzer bekerja dengan:

Input:

- Abstract Syntax Tree (AST) dari syntax analyzer
- Symbol table yang sudah dibangun (dari bab sebelumnya)
- Type information dari deklarasi

Output:

- Annotated AST dengan informasi tipe pada setiap node
- Symbol table yang dilengkapi dengan informasi tipe
- Daftar semantic errors (jika ada)
- Type-checked program yang siap untuk code generation

11.3 Sistem Tipe (Type System)

Sistem tipe adalah kumpulan aturan yang menentukan bagaimana tipe data ditetapkan pada konstruksi program dan operasi apa yang "legal" untuk setiap tipe.

11.3.1 Jenis-jenis Type System

Static vs Dynamic Typing

- **Static Typing**: Pengecekan tipe dilakukan pada waktu kompilasi. Bahasa seperti C, C++, Java, dan Rust menggunakan static typing. Keuntungannya adalah deteksi error lebih awal dan performa runtime yang lebih baik.

- **Dynamic Typing**: Pengecekan tipe dilakukan pada waktu eksekusi. Bahasa seperti Python, JavaScript, dan Ruby menggunakan dynamic typing. Keuntungannya adalah fleksibilitas lebih tinggi, tetapi error baru terdeteksi saat runtime.

Nominal vs Structural Typing

- **Nominal Typing**: Kompatibilitas tipe ditentukan berdasarkan nama tipe yang dideklarasikan. Dua tipe dengan struktur yang sama tetapi nama berbeda dianggap tidak kompatibel. Contoh: Java, C++.

- **Structural Typing**: Kompatibilitas tipe ditentukan berdasarkan struktur tipe (field, method). Jika struktur cocok, tipe dianggap kompatibel meskipun nama berbeda. Contoh: TypeScript, OCaml.

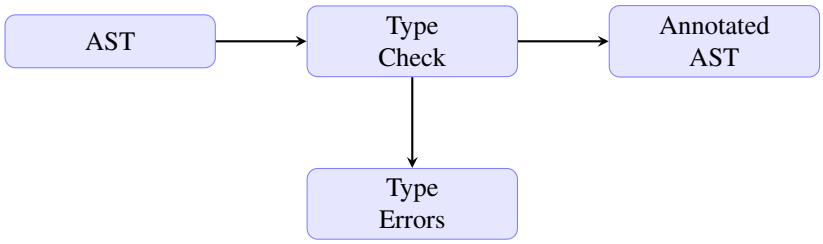
11.3.2 Type Hierarchy

Dalam bahasa berorientasi objek, tipe-tipe membentuk hierarki melalui inheritance. Misalnya:

```
Object
|-- Number
|   |-- Integer
|   `-- Float
|-- String
`-- Boolean
```

Hierarki ini memungkinkan subtyping, di mana tipe turunan dapat digunakan di tempat tipe induk (substitution principle).

Gambar 11.2 menunjukkan proses type checking.



Gambar 11.2: Proses type checking

11.4 Type Checking

Type checking adalah proses memastikan bahwa program mematuhi aturan tipe bahasa pemrograman. Menurut GeeksforGeeks:

> "The process of ensuring that a program adheres to the language's type rules. Checking for things like 'you can't add an integer to a string', that function calls match the declared parameter types, etc."[1]

11.4.1 Aturan Type Checking Dasar

Type Checking untuk Ekspresi Aritmatika

Untuk ekspresi aritmatika, aturan dasar meliputi:

1. **Literals**: Setiap literal memiliki tipe intrinsik
   - Integer literal (42) → `int`
   - Float literal (3.14) → `float`
   - String literal ("hello") → `string`
2. **Operasi Aritmatika**: Operan harus kompatibel
   - `int + int` → `int`
   - `float + float` → `float`
   - `int + float` → `float` (dengan implicit conversion)
3. **Assignment**: Tipe ekspresi harus kompatibel dengan tipe variabel
   - `int x = 42;` ✓ (valid)
   - `int x = 3.14;` × (type mismatch, atau perlu explicit cast)

[1] https://www.geeksforgeeks.org/type-checking-in-compiler-design/

Type Checking untuk Function Calls

Untuk pemanggilan fungsi, type checker memverifikasi:

1. Jumlah argumen sesuai dengan jumlah parameter
2. Tipe setiap argumen kompatibel dengan tipe parameter yang sesuai
3. Return type dari fungsi sesuai dengan konteks penggunaan

Contoh:

```
int add(int a, int b) { return a + b; }

// Valid call
int result = add(5, 10);  // OK

// Invalid calls
add(5);                   // ERROR: Wrong number of arguments
int x = add(5.0, 10.0);   // ERROR: Type mismatch (float vs int)
```

11.4.2 Implementasi Type Checker Sederhana

Berikut adalah contoh struktur data untuk type checker dalam C++:

Listing 11.1: Struktur Data untuk Type System

```
enum class TypeKind {
    INT,
    FLOAT,
    STRING,
    BOOL,
    VOID,
    ARRAY,
    FUNCTION
};

struct Type {
    TypeKind kind;
    // Untuk array: element type
    // Untuk function: parameter types dan return type
    std::vector<Type> subtypes;
};

class TypeChecker {
private:
    SymbolTable* symbolTable;

public:
    TypeChecker(SymbolTable* st) : symbolTable(st) {}

    // Type check sebuah ekspresi
    Type checkExpression(ASTNode* expr);

```

```
    // Type check sebuah statement
    void checkStatement(ASTNode* stmt);

    // Type check sebuah program
    void checkProgram(ASTNode* program);

    // Cek kompatibilitas tipe
    bool isCompatible(Type t1, Type t2);

    // Lakukan implicit conversion jika perlu
    Type promoteType(Type t1, Type t2);
};
```

Type Checking untuk Binary Operations

Berikut adalah contoh implementasi type checking untuk operasi biner:

Listing 11.2: Type Checking untuk Binary Operations

```
Type TypeChecker::checkBinaryOp(ASTNode* node) {
    ASTBinaryOp* binOp = static_cast<ASTBinaryOp*>(node);

    Type leftType = checkExpression(binOp->left);
    Type rightType = checkExpression(binOp->right);

    switch (binOp->op) {
        case OP_PLUS:
        case OP_MINUS:
        case OP_MULTIPLY:
        case OP_DIVIDE:
            // Operasi aritmatika: int atau float
            if (leftType.kind == TypeKind::INT &&
                rightType.kind == TypeKind::INT) {
                return Type{TypeKind::INT};
            }
            if (leftType.kind == TypeKind::FLOAT ||
                rightType.kind == TypeKind::FLOAT) {
                return promoteType(leftType, rightType);
            }
            reportError("Arithmetic operation on incompatible types");
            break;

        case OP_EQUAL:
        case OP_NOT_EQUAL:
        case OP_LESS:
        case OP_GREATER:
            // Operasi perbandingan: hasilnya boolean
            if (isCompatible(leftType, rightType)) {
                return Type{TypeKind::BOOL};
            }
            reportError("Comparison on incompatible types");
            break;

        default:
```

```
            reportError("Unknown binary operator");
        }

        return Type{TypeKind::VOID}; // Error type
    }
```

11.5 Type Inference

Type inference adalah kemampuan kompilator untuk secara otomatis menentukan tipe ekspresi tanpa memerlukan anotasi tipe eksplisit dari programmer.

11.5.1 Type Inference untuk Literal

Untuk literal, tipe dapat langsung diinfer:

- `42` → `int`
- `3.14` → `float`
- `"hello"` → `string`
- `true` → `bool`

11.5.2 Type Inference untuk Operasi

Untuk operasi, tipe hasil diinfer berdasarkan tipe operan:

- `int + int` → `int`
- `int + float` → `float` (promotion)
- `int == int` → `bool`
- `int < float` → `bool`

11.5.3 Type Inference untuk Variabel

Dalam beberapa bahasa (seperti C++ dengan `auto`, atau Rust), tipe variabel dapat diinfer dari initializer:

```
auto x = 42;        // x inferred as int
auto y = 3.14;      // y inferred as float
auto z = x + y;     // z inferred as float
```

11.5.4 Implementasi Type Inference Sederhana

Listing 11.3: Type Inference untuk Ekspresi

```
Type TypeChecker::inferType(ASTNode* expr) {
    switch (expr->nodeType) {
        case NODE_INT_LITERAL:
            return Type{TypeKind::INT};

        case NODE_FLOAT_LITERAL:
            return Type{TypeKind::FLOAT};

        case NODE_STRING_LITERAL:
            return Type{TypeKind::STRING};

        case NODE_VARIABLE: {
            ASTVariable* var = static_cast<ASTVariable*>(expr);
            Symbol* symbol = symbolTable->lookup(var->name);
            if (symbol) {
                return symbol->type;
            }
            reportError("Undeclared variable: " + var->name);
            return Type{TypeKind::VOID};
        }

        case NODE_BINARY_OP:
            return checkBinaryOp(expr);

        case NODE_FUNCTION_CALL: {
            ASTFunctionCall* call = static_cast<ASTFunctionCall*>(expr);
            Symbol* func = symbolTable->lookup(call->name);
            if (func && func->type.kind == TypeKind::FUNCTION) {
                return func->type.subtypes.back(); // Return type
            }
            reportError("Undeclared function: " + call->name);
            return Type{TypeKind::VOID};
        }

        default:
            reportError("Cannot infer type for node");
            return Type{TypeKind::VOID};
    }
}
```

11.6 Type Compatibility

Type compatibility menentukan apakah satu tipe dapat digunakan di tempat tipe lain. Menurut sumber dari TypeScript documentation:

> "Two types are compatible if one can be used in place of another without type errors. This often arises when assigning a value to a variable, passing arguments

to a function, etc."[2]

11.6.1 Aturan Type Compatibility

Exact Match

Dua tipe yang identik selalu kompatibel:

```
int x = 42;          // int = int (OK)
float y = 3.14;      // float = float (OK)
```

Implicit Conversion (Type Promotion)

Beberapa bahasa mengizinkan implicit conversion antara tipe yang "dekat":

```
int x = 42;
float y = x;         // int -> float (promotion) OK

float a = 3.14;
int b = a;           // float -> int (downgrade) ERROR atau perlu cast
```

Aturan umum untuk promotion:

- `int` → `float` (biasanya diizinkan)
- `int` → `long` (diizinkan)
- `float` → `double` (diizinkan)
- `float` → `int` (biasanya memerlukan explicit cast)

Subtyping

Dalam bahasa berorientasi objek, tipe turunan kompatibel dengan tipe induk:

```
class Animal { }
class Dog extends Animal { }

Animal a = new Dog();  // Dog is subtype of Animal (OK)
```

11.6.2 Implementasi Type Compatibility Check

Listing 11.4: Implementasi Type Compatibility

```
bool TypeChecker::isCompatible(Type t1, Type t2) {
    // Exact match
    if (t1.kind == t2.kind) {
        // Untuk tipe kompleks, perlu cek lebih detail
        if (t1.kind == TypeKind::ARRAY) {
            return isCompatible(t1.subtypes[0], t2.subtypes[0]);
```

[2] https://www.typescriptlang.org/docs/handbook/type-compatibility.html

```
        }
        if (t1.kind == TypeKind::FUNCTION) {
            // Cek parameter types dan return type
            if (t1.subtypes.size() != t2.subtypes.size()) {
                return false;
            }
            for (size_t i = 0; i < t1.subtypes.size() - 1; i++) {
                if (!isCompatible(t1.subtypes[i], t2.subtypes[i])) {
                    return false;
                }
            }
            return isCompatible(
                t1.subtypes.back(),
                t2.subtypes.back()
            );
        }
        return true;
    }

    // Implicit conversion rules
    if (t1.kind == TypeKind::INT && t2.kind == TypeKind::FLOAT) {
        return true;  // int dapat di-promote ke float
    }

    // Subtyping (jika ada)
    // ... implementasi subtyping check

    return false;
}

Type TypeChecker::promoteType(Type t1, Type t2) {
    // Jika salah satu float, hasilnya float
    if (t1.kind == TypeKind::FLOAT || t2.kind == TypeKind::FLOAT) {
        return Type{TypeKind::FLOAT};
    }
    // Default: int
    return Type{TypeKind::INT};
}
```

11.7 Semantic Error Detection dan Reporting

Semantic analyzer harus mendeteksi berbagai jenis error dan memberikan pesan error yang jelas dan informatif.

11.7.1 Jenis-jenis Semantic Error

Undeclared Variable

Error terjadi ketika variabel digunakan sebelum dideklarasi:

```
x = 42;  // Error: 'x' is not declared
int x;
```

Type Mismatch

Error terjadi ketika tipe tidak kompatibel:

```
int x = "hello";  // Error: cannot assign string to int
int y = 3.14;     // Error: cannot assign float to int (tanpa cast)
```

Undefined Function

Error terjadi ketika fungsi dipanggil tetapi tidak didefinisikan:

```
int result = add(5, 10);  // Error: function 'add' is not defined
```

Wrong Number of Arguments

Error terjadi ketika jumlah argumen tidak sesuai:

```
int add(int a, int b) { return a + b; }
int x = add(5);  // Error: expected 2 arguments, got 1
```

Return Type Mismatch

Error terjadi ketika return type tidak sesuai dengan deklarasi:

```
int getValue() {
    return "hello";  // Error: function should return int
}
```

11.7.2 Error Reporting yang Informatif

Pesan error yang baik harus:

- Menunjukkan lokasi error (baris, kolom)
- Menjelaskan jenis error dengan jelas
- Memberikan konteks yang relevan
- Menyarankan solusi jika memungkinkan

Contoh implementasi error reporting:

Listing 11.5: Error Reporting System

```
class ErrorReporter {
private:
    std::vector<Error> errors;

public:
```

```
    void reportError(const std::string& message,
                     int line, int column) {
        errors.push_back(Error{message, line, column});
        std::cerr << "Error at line " << line
                  << ", column " << column
                  << ": " << message << std::endl;
    }

    void reportTypeError(const std::string& expected,
                         const std::string& got,
                         int line, int column) {
        std::string msg = "Type mismatch: expected " + expected +
                          ", got " + got;
        reportError(msg, line, column);
    }

    bool hasErrors() const {
        return !errors.empty();
    }

    const std::vector<Error>& getErrors() const {
        return errors;
    }
};
```

11.8 Integrasi dengan Symbol Table

Type checking bekerja erat dengan symbol table yang telah dibangun pada fase sebelumnya. Symbol table menyediakan informasi tentang:

- Tipe setiap variabel yang dideklarasikan
- Tipe parameter dan return type setiap fungsi
- Scope di mana setiap identifier dideklarasikan

Contoh integrasi:

Listing 11.6: Type Checking dengan Symbol Table

```
Type TypeChecker::checkVariable(ASTVariable* var) {
    Symbol* symbol = symbolTable->lookup(var->name);

    if (!symbol) {
        errorReporter->reportError(
            "Undeclared variable: " + var->name,
            var->line, var->column
        );
        return Type{TypeKind::VOID};
    }

    if (symbol->kind != SymbolKind::VARIABLE) {
```

```
        errorReporter->reportError(
            var->name + " is not a variable",
            var->line, var->column
        );
        return Type{TypeKind::VOID};
    }

    return symbol->type;
}

void TypeChecker::checkAssignment(ASTAssignment* assign) {
    Type varType = checkVariable(assign->variable);
    Type exprType = checkExpression(assign->expression);

    if (!isCompatible(varType, exprType)) {
        errorReporter->reportTypeError(
            typeToString(varType),
            typeToString(exprType),
            assign->line, assign->column
        );
    }
}
```

11.9 Type Checking untuk Kontrol Flow

Type checker juga perlu memverifikasi kontrol flow statements:

11.9.1 If Statement

Listing 11.7: Type Checking untuk If Statement

```
void TypeChecker::checkIfStatement(ASTIf* ifStmt) {
    Type condType = checkExpression(ifStmt->condition);

    if (condType.kind != TypeKind::BOOL) {
        errorReporter->reportError(
            "If condition must be boolean",
            ifStmt->line, ifStmt->column
        );
    }

    checkStatement(ifStmt->thenBranch);
    if (ifStmt->elseBranch) {
        checkStatement(ifStmt->elseBranch);
    }
}
```

11.9.2 While Loop

Listing 11.8: Type Checking untuk While Loop

```
void TypeChecker::checkWhileLoop(ASTWhile* whileLoop) {
    Type condType = checkExpression(whileLoop->condition);

    if (condType.kind != TypeKind::BOOL) {
        errorReporter->reportError(
            "While condition must be boolean",
            whileLoop->line, whileLoop->column
        );
    }

    checkStatement(whileLoop->body);
}
```

11.9.3 Return Statement

Listing 11.9: Type Checking untuk Return Statement

```
void TypeChecker::checkReturn(ASTReturn* ret, Type expectedReturnType) {
    if (ret->expression) {
        Type exprType = checkExpression(ret->expression);
        if (!isCompatible(expectedReturnType, exprType)) {
            errorReporter->reportTypeError(
                typeToString(expectedReturnType),
                typeToString(exprType),
                ret->line, ret->column
            );
        }
    } else {
        if (expectedReturnType.kind != TypeKind::VOID) {
            errorReporter->reportError(
                "Function must return a value",
                ret->line, ret->column
            );
        }
    }
}
```

11.10 Annotated AST

Setelah type checking, setiap node AST di-annotate dengan informasi tipe. Ini memudahkan fase selanjutnya (code generation) untuk mengetahui tipe setiap ekspresi.

Listing 11.10: Annotated AST Node

```
class ASTNode {
public:
    NodeType nodeType;
    int line, column;
    Type type;  // Annotated type (setelah type checking)

```

```
    virtual ~ASTNode() = default;
};

// Contoh penggunaan
Type TypeChecker::checkExpression(ASTNode* expr) {
    Type t = inferType(expr);
    expr->type = t;  // Annotate node dengan type
    return t;
}
```

11.11 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Semantic analysis memverifikasi bahwa program memenuhi aturan semantik bahasa
2. Type checking memastikan operasi dilakukan pada tipe yang kompatibel
3. Type inference memungkinkan kompilator menentukan tipe secara otomatis
4. Type compatibility menentukan apakah satu tipe dapat digunakan di tempat tipe lain
5. Semantic error detection dan reporting memberikan feedback yang jelas kepada programmer
6. Type checking terintegrasi dengan symbol table untuk mengakses informasi deklarasi
7. Annotated AST menyimpan informasi tipe untuk digunakan pada fase selanjutnya

Pemahaman tentang type checking dan semantic analysis ini penting karena memastikan bahwa program yang dikompilasi tidak hanya valid secara sintaksis, tetapi juga benar secara semantik sebelum masuk ke fase code generation. Type checking proyek subset C memakai AST proyek (Bab 9) dan symbol table (Bab 10); annotated AST menjadi input untuk IR generation (Bab 12) dan code generation (Bab 14).

11.12 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang type checking dan semantic analysis, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [2] - Bab 6: Type Checking
- **Engineering a Compiler**: Cooper & Torczon (2011) [3] - Bab 4: Context-Sensitive Analysis

- **Nguyen Thanh Vu - Compiler Class Notes**: Semantic Analysis [1]
- **GeeksforGeeks**: Type Checking in Compiler Design [3]
- **Wikipedia - Type Inference**: [4]
- **Wikipedia - Type System**: [5]
- **TypeScript Handbook - Type Compatibility**: [6]

[3] https://www.geeksforgeeks.org/type-checking-in-compiler-design/
[4] https://en.wikipedia.org/wiki/Type_inference
[5] https://en.wikipedia.org/wiki/Type_system
[6] https://www.typescriptlang.org/docs/handbook/type-compatibility.html

Daftar Pustaka

[1] Nguyen Thanh Vu. *Compiler Class Notes: Semantic Analysis*. Class notes. 2024. URL: https://nguyenthanhvuh.github.io/class-compilers/notes/sem.html.

[2] Alfred V. Aho **andothers**. *Compilers: Principles, Techniques, and Tools*. 2nd. Dragon Book. Pearson Education, 2006.

[3] Keith D. Cooper **and** Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2011.