Static Type Checker Implementation Report

X++ Interpreter

1 Overview

The X++ static type checker is implemented using a syntax-directed approach where each AST node contains its own type checking logic through the `typecheck` method defined in `ASTNode.java`:

```
public interface ASTNode {
    IValue eval(Environment<IValue> e) throws
        InterpreterError;
    ASTType typecheck(TypeEnvironment gamma,
                      TypeDefEnvironment typeDefs) throws
                          TypeError;
}
```

2 Architecture

2.1 Type Representation

The type system is built around the `ASTType` interface (`ASTType.java`). For example, function types are represented in `ASTTArrow.java`:

```
public class ASTTArrow implements ASTType {
    final ASTType dom;
    final ASTType codom;

    public ASTType getDomain() { return this.dom; }
    public ASTType getCodomain() { return this.codom; }
}
```

2.2 Type Environments

The `TypeEnvironment.java` implements a scoped symbol table for variable types:

```
public final void assoc(String id, ASTType type) throws
    TypeError {
    if (this.bindings.containsKey(id))
        throw new TypeError("Variable␣" + id +
                              "␣already␣defined␣in␣this␣scope");
    this.bindings.put(id, type);
}
```

3 Key Implementation Examples

3.1 Binary Operations

Consider the implementation of addition in `ASTPlus.java`, which handles both integer addition and string concatenation:

```
public ASTType typecheck(TypeEnvironment gamma,
                        TypeDefEnvironment typeDefs) throws
                            TypeError {
    final ASTType leftType = this.lhs.typecheck(gamma,
        typeDefs);
    final ASTType rightType = this.rhs.typecheck(gamma,
        typeDefs);

    // Integer addition
    if (leftType instanceof ASTTInt && rightType instanceof
        ASTTInt)
        return new ASTTInt();

    // String concatenation
    if (leftType instanceof ASTTString || rightType
        instanceof ASTTString)
        return new ASTTString();

    throw new TypeError("+␣operator␣requires␣int␣or␣string␣
        operands,␣got␣"
                      + leftType.toStr() + "␣and␣" +
                        rightType.toStr());
}
```

3.2 Function Application with Subtyping

The `ASTApp.java` implementation shows how subtyping is integrated:

```
public ASTType typecheck(TypeEnvironment gamma,
                        TypeDefEnvironment typeDefs) throws
                            TypeError {
    final ASTType funType = this.function.typecheck(gamma,
        typeDefs);
    final ASTType argType = this.argument.typecheck(gamma,
        typeDefs);

    if (!(funType instanceof ASTTArrow))
```

```
        throw new TypeError("Function␣application␣requires␣a
            ␣function␣type");

    final ASTTArrow arrowType = (ASTTArrow) funType;

    if (!Subtyping.isSubtype(argType, arrowType.getDomain(),
         typeDefs))
        throw new TypeError("Argument␣type␣" + argType.toStr
            () +
            "␣is␣not␣compatible␣with␣parameter␣type␣" +
            arrowType.getDomain().toStr());

    return arrowType.getCodomain();
}
```

3.3 Struct Subtyping

The width subtyping for structs is implemented in `Subtyping.java`:

```
private static boolean isStructSubtype(ASTTStruct subStruct,
                                        ASTTStruct superStruct
                                            ,
                                        TypeDefEnvironment
                                            typeDefs) {
    final Map<String, ASTType> subFields = subStruct.
        getFields();
    final Map<String, ASTType> superFields = superStruct.
        getFields();

    for (Map.Entry<String, ASTType> superField : superFields
        .entrySet()) {
        final String fieldName = superField.getKey();
        if (!subFields.containsKey(fieldName))
            return false; // Missing required field

        final ASTType subFieldType = subFields.get(fieldName
            );
        if (!isSubtype(subFieldType, superFieldType,
            typeDefs))
            return false; // Field type mismatch
    }
    return true;
}
```

3.4 Union Type Constructors

The `ASTTypeDef.java` shows how union constructors are added during type checking:

```
if (type instanceof ASTTUnion unionType) {
    for (Map.Entry<String, ASTType> variant :
         unionType.getVariants().entrySet()) {
        final String variantName = variant.getKey();
        final ASTType variantType = variant.getValue();

        final ASTType constructorType =
            new ASTTArrow(variantType, new ASTTId(typeName))
                ;
        newGamma.assoc(variantName, constructorType);
    }
}
```

4 Integration with Main Interpreter

The type checker runs before evaluation in `L0int.java`:

```
ASTType type = TypeChecker.typecheck(exp);
IValue v = exp.eval(new Environment<IValue>());
```

This ensures type safety before runtime execution, catching errors early in the compilation pipeline.