Static Type Checker Implementation for LX++

Guilherme Leitão ist199951

1 Architecture

1.1 Core Components

The type checker consists of four main components:

1. **TypeChecker**: Entry point that initiates type checking
2. **TypeEnvironment**: Manages variable-to-type bindings (Γ)
3. **TypeDefEnvironment**: Manages type definitions (Φ)
4. **Subtyping**: Implements subtyping rules and type equality

1.2 Type Representation

Types are represented as AST nodes implementing the `ASTType` interface:

```
public interface ASTType {
    String toStr();
}

// Basic types
class ASTTInt implements ASTType { ... }
class ASTTBool implements ASTType { ... }
class ASTTString implements ASTType { ... }
class ASTTUnit implements ASTType { ... }

// Compound types
class ASTTRef implements ASTType { ... }
class ASTTList implements ASTType { ... }
class ASTTStruct implements ASTType { ... } // labeled product types
class ASTTUnion implements ASTType { ... }  // labeled sum types
```

Listing 1: Basic Type Hierarchy

2 Type Checking Algorithm

2.1 AST Node Type Checking

Each AST node implements a `typecheck` method following the typing rules:

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

Listing 2: Type Checking Interface

2.2 Example: Dereferencing Operations

The type checker ensures operands have compatible types:

```
public ASTType typecheck(TypeEnvironment gamma, TypeDefEnvironment
   typeDefs) throws TypeError {
        final ASTType exprType = this.expr.typecheck(gamma, typeDefs);

        if (!(exprType instanceof ASTTRef) && exprType != null)
            throw new TypeError("Cannot␣dereference␣non-reference␣type␣" +
                exprType.toStr());

        return ((ASTTRef) exprType).getType();
}
```

Listing 3: Dereferencing Type Checking

3 Subtyping Implementation

3.1 Subtyping Rules

The subtyping system implements standard rules including:

- **Reflexivity**: $A <: A$
- **Transitivity**: $A <: B \wedge B <: C \implies A <: C$
- **Function countervariance**: $C <: A \wedge B <: D \implies A \to B <: C \to D$
- **Labeled products width subtyping**: structs with more fields are subtypes
- **Reference invariance**: $A <:> B \implies \mathrm{ref}(A) <: \mathrm{ref}(B)$

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

    for (Map.Entry<String, ASTType> superField : superFields.entrySet()) {
        String fieldName = superField.getKey();
        ASTType superFieldType = superField.getValue();

        if (!subFields.containsKey(fieldName))
```

```
            return false; // Missing required field

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

Listing 4: Struct Subtyping Implementation

3.2 Recursive Type Resolution

The type checker handles recursive types through careful resolution:

```
private static ASTType resolveType(ASTType type,
                                  TypeDefEnvironment typeDefs) {
    if (type instanceof ASTTId typeId) {
        String id = typeId.id;

        if (resolvingTypes.contains(id)) // Prevent infinite recursion
            return type;

        resolvingTypes.add(id);
        try {
            return typeDefs.find(id);
        } finally {
            resolvingTypes.remove(id);
        }
    }
    return type;
}
```

Listing 5: Recursive Type Resolution

4 Separate Match Constructs

LX++ implements two distinct pattern matching constructs for clarity:

1. **ASTMatch**: For list pattern matching (nil and cons cases)
2. **ASTCaseMatch**: For union type pattern matching

This separation provides better type safety and clearer semantics:

```
public ASTType typecheck(TypeEnvironment gamma,
                        TypeDefEnvironment typeDefs) throws TypeError {
    ASTType exprType = this.expr.typecheck(gamma, typeDefs);

    if (!(exprType instanceof ASTTList))
        throw new TypeError("List␣match␣requires␣a␣list␣type");

    ASTTList listType = (ASTTList) exprType;
    ASTType elementType = listType.getElementType();

    // Type check nil case
    ASTType nilCaseType = this.nilCase.typecheck(gamma, typeDefs);

    // Type check cons case with extended environment
    TypeEnvironment consEnv = gamma.beginScope();
    consEnv.assoc(this.headVar, elementType);
    consEnv.assoc(this.tailVar, listType);
    ASTType consCaseType = this.consCase.typecheck(consEnv, typeDefs);

    // Ensure compatible branch types
    if (!Subtyping.isSubtype(nilCaseType, consCaseType, typeDefs) &&
        !Subtyping.isSubtype(consCaseType, nilCaseType, typeDefs))
        throw new TypeError("Match␣cases␣must␣have␣compatible␣types");

    return /* more general type */;
}
```

Listing 6: List Match Type Checking

5 Type Definitions and Environments

5.1 Type Definition Handling

Type definitions are processed before the main program body:

```
public ASTType typecheck(TypeEnvironment gamma,
                        TypeDefEnvironment typeDefEnv) throws TypeError {
    TypeDefEnvironment newTypeDefEnv = typeDefEnv.beginScope();
    TypeEnvironment newGamma = gamma.beginScope();

    // Register all type definitions
    for (Map.Entry<String, ASTType> entry : this.typeDefs.entrySet())
        newTypeDefEnv.assoc(entry.getKey(), entry.getValue());

    // Add union constructors to type environment
    for (Map.Entry<String, ASTType> entry : this.typeDefs.entrySet()) {
        if (entry.getValue() instanceof ASTTUnion unionType) {
            for (Map.Entry<String, ASTType> variant :
                 unionType.getVariants().entrySet()) {
                ASTType constructorType = new ASTTArrow(
                    variant.getValue(),
                    new ASTTId(entry.getKey())
                );
                newGamma.assoc(variant.getKey(), constructorType);
            }
        }
    }

    return this.body.typecheck(newGamma, newTypeDefEnv);
}
```

Listing 7: Type Definition Processing