Static Type Checker Implementation for LX++

Guilherme Leitão ist199951

1 Overview

The LX++ interpreter implements a static type checker that ensures type safety before program execution. The type system supports functional programming with mutable state, labeled types (products and sums), and recursive type definitions. The implementation follows the big-step typing rules studied in the course. As a side note, labeled product and sum types are terms I've personally interchanged with structs and unions, respectively.

2 Architecture

2.1 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

2.2 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 { ... } // Which are the labeled
    product types
class ASTTUnion implements ASTType { ... }  // which are the labeled sum
    types
```

Listing 1: Basic Type Hierarchy

3 Type Checking Algorithm

3.1 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

3.2 Example: Binary Operations

The type checker ensures operands have compatible types:

```
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");
}
```

Listing 3: Plus Operation Type Checking

3.3 Dual Function Type System

LX++ implements a dual function type system to support both single-parameter and multi-parameter functions elegantly:

- **ASTTArrow**: Traditional curried function type $(A \to B)$
- **ASTTFunction**: Multi-parameter function type $(A_1, ..., A_n) \to B$

This allows natural syntax for multi-parameter functions while maintaining compatibility with curried functions:

```
public ASTType typecheck(TypeEnvironment gamma,
                         TypeDefEnvironment typeDefs) throws TypeError {
    if (this.expectedType != null) {
        // Type inference from context
        List<ASTType> inferredParamTypes;
        ASTType expectedReturnType;
        switch (this.expectedType) {
            case ASTTFunction funcType -> {
                inferredParamTypes = funcType.getParamTypes();
                expectedReturnType = funcType.getReturnType();
            }
            case ASTTArrow arrowType -> {
                inferredParamTypes = Arrays.asList(arrowType.getDomain());
                expectedReturnType = arrowType.getCodomain();
            }
        }
        // ... validate and type check body
    }
}
```

Listing 4: Function Type Checking with Type Inference

4 Subtyping Implementation

4.1 Subtyping Rules

The subtyping system implements standard rules including:

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

```
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 5: Struct Subtyping Implementation

4.2 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;

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

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

Listing 6: Recursive Type Resolution

5 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 7: List Match Type Checking

6 Type Definitions and Environments

6.1 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 8: Type Definition Processing

7 Notable Features

7.1 String Concatenation with Type Coercion

The + operator supports automatic type conversion to strings:

```
"The␣answer␣is:␣" + 42  // Results in "The answer is: 42"
```

7.2 Reference Type Invariance

Reference types require exact type matching to ensure memory safety:

```
ref(A) <: ref(B)  iff  A <:> B  (A and B are equivalent)
```

8 Conclusion

The LX++ type checker implements a sophisticated type system that balances expressiveness with safety. The dual function type system, separate match constructs, and careful handling of subtyping relationships provide a robust foundation for static type checking in a functional-imperative language. The implementation closely follows the formal semantics while providing practical features like type inference and automatic string conversion.