Static Type Checker Implementation for LX++

Guilherme Leitão ist199951

1 Product and Union Types Implementation

1.1 Product Types (Labeled Records)

Product types are implemented through `ASTTStruct`, representing labeled records with named fields:

```
public class ASTTStruct implements ASTType {
    private final Map<String, ASTType> fields;

    ...
}
```

Listing 1: Product Type Representation

Type checking for struct literals validates each field and constructs the appropriate type:

```
final Map<String, ASTType> fieldTypes = new HashMap<>();

for (Map.Entry<String, ASTNode> entry : this.fields.entrySet())
    fieldTypes.put(entry.getKey(),
        entry.getValue().typecheck(gamma, typeDefs));

return new ASTTStruct(fieldTypes);
```

Listing 2: Struct Literal Type Checking (ASTStructLiteral.java)

Field access type checking ensures the field exists and returns its type:

```
ASTType structType = this.struct.typecheck(gamma, typeDefs);

if (structType instanceof ASTTId aSTTId) // Resolve type aliases
    structType = typeDefs.find(aSTTId.id);

if (!(structType instanceof ASTTStruct))
    throw new TypeError("Field␣access␣requires␣a␣struct␣type");

...

final ASTType fieldType = structTypeAST.getFields().get(this.fieldName);
if (fieldType == null)
    throw new TypeError("Field␣" + fieldName + "␣not␣found");
```

Listing 3: Field Access Type Checking (ASTFieldAccess.java)

1.2 Union Types (Labeled Sums)

Union types are implemented through `ASTTUnion`, representing tagged variants:

```
public class ASTTUnion implements ASTType {
    private final Map<String, ASTType> variants;

    ...
}
```

Listing 4: Union Type Representation

Union constructors are added to the type environment as functions:

```
if (type instanceof ASTTUnion unionType) {
    for (Map.Entry<String, ASTType> variant :
         unionType.getVariants().entrySet()) {
        ...

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

Listing 5: Union Constructor Type Assignment (ASTTypeDef.java)

2 Subtyping Integration

2.1 Core Subtyping Algorithm

Subtyping is implemented in `Subtyping.java` with the main entry point:

```
public static boolean isSubtype(ASTType subType, ASTType superType,
                                 TypeDefEnvironment typeDefs) throws
                                    TypeError {
    if (subType instanceof ASTTFunction) // Handle curried functions
        subType = ((ASTTFunction)subType).toCurriedType();

    if (typeEquals(subType, superType, typeDefs)) // Reflexivity: A <: A
        return true;

    // Resolve type aliases
    ASTType resolvedSub = resolveType(subType, typeDefs);
    ASTType resolvedSuper = resolveType(superType, typeDefs);

    // Apply specific subtyping rules...
}
```

Listing 6: Main Subtyping Function

2.2 Width Subtyping for Products

Structs support width subtyping - a struct with more fields is a subtype of another:

```
private static boolean isStructSubtype(...) {
    // Check all required fields exist with compatible types
    for (Map.Entry<String, ASTType> superField :
         superFields.entrySet()) {
        ...
        if (!subFields.containsKey(fieldName))
                return false; // Missing mandatory field
        ...
        if (!isSubtype(subFieldType, superFieldType, typeDefs))
                return false; // Field type mismatch
        ...
    }
    return true; // Extra fields are allowed
}
```

Listing 7: Struct Width Subtyping (Subtyping.java)

2.3 Integration in Type Checking

Subtyping is used throughout type checking, particularly in:

- Function application (`ASTApp.java`): argument subtyping
- Variable assignment (`ASTAssign.java`): right-hand side subtyping
- Let bindings with explicit types (`ASTLet.java`)
- If expressions and match branches: finding common supertypes

Example from function application:

```
if (!Subtyping.isSubtype(argType, firstParamType, typeDefs))
    throw new TypeError(...);
```

Listing 8: Subtyping in Function Application (ASTApp.java)

3 Recursive Types Implementation

3.1 Type Representation and Resolution

Recursive types are represented using type identifiers (`ASTTId`) that reference type definitions stored in `TypeDefEnvironment`:

```
// Resolution with cycle detection
private final static Set<String> resolvingTypes = new HashSet<>();

private static ASTType resolveType(...) {
    if (type instanceof ASTTId typeId) {
        final String id = typeId.id;

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

```
        resolvingTypes.add(id);

        try {
            final ASTType resolved = typeDefs.find(id);
            return resolved;
        } finally {
            resolvingTypes.remove(id);
        }
    }
    return type;
}
```

Listing 9: Type Identifier and Resolution (Subtyping.java)

3.2 Type Checking with Recursive Types

The type checker handles recursive types by resolving type identifiers when needed:

```
public ASTType typecheck(...) {
    ASTType structType = this.struct.typecheck(gamma, typeDefs);

    // Resolve type identifier to actual type definition
    if (structType instanceof ASTTId aSTTId)
        structType = typeDefs.find(aSTTId.id);

    // Now work with resolved type
    if (!(structType instanceof ASTTStruct))
        throw new TypeError(...);

    ...
}
```

Listing 10: Recursive Type Handling Example (ASTFieldAccess.java)

3.3 Recursive Type Definitions

Type definitions support mutual recursion through forward references:

```
// Example from test32.l0:
type Btree = union { #Nil:(), #Node:NodeT };
type NodeT = struct { #left:Btree, #val:int, #right:Btree };
```

Listing 11: Recursive Type Example

The type definition environment handles these by:

1. Registering all type names first
2. Allowing forward references during type checking
3. Using cycle detection during resolution to prevent infinite loops