Salvador Carvalhinho
103667

TÉCNICO LISBOA

Project Report

Phase 2 - Programming Languages

1. Implementation Overview

Succinct explanation of the overall Phase 2 code implementation

The base used to start this 2nd Phase of the *Programming Language* Project was the code from the 1st Phase, available here: https://github.com/salvadorcarvalhinho/LPProject-1

Although the first impressions upon reading the *Project Statement* gave the idea that we should start by defining the new nodes and making sure our Parser worked accordingly, I decided to start by adding support for the given types and using the **ASTNode.*typeCheck(Environment*<ASTType> *env)***.

This gave a good basis to work upon. After successfully compiling a version of the code with the types and the .***typeCheck(Environment*<ASTType> *env)*** support for all nodes, I then started implementing the newer nodes and adapting the ***Parser.jj***. This proved to be a difficult part because not only I had to make sure the new nodes were working correctly, but also had to ensure that the parser worked correctly and didn't affect any of the previous work.

In the end, only the **recursive types** remained, and after many discussions with my colleagues and coding attempts I finally managed to add support to them and finish the coding part of this project.

Finally, only the SasyLF file remained and supported by the Lab Classes Files I managed to complete it fairly easily.

2. *Product* Type

Explanation of Product Type implementation in the code

Like the other types, Product (or Struct) was implemented using 3 distinct files:

- **ASTStruct**: represents the node itself and has the ***.eval(Environment<IValue> env)*** and ***.typeCheck(Environment<ASTType> env)*** methods, useful in the evaluation and type checking of the code, accordingly.
- **ASTTStruct**: represents the ***Product Type*** and has the ***.isSubtypeOf(ASTType other, Environment<ASTType> e)*** and ***.simplify(Environment<ASTType> env, Set<String> simplified)*** used to simplify/flatten the type and to check for sub-typing (more on this later)
- **VStruct**: represent the value Product/Structure and is used at runtime to store each value in its corresponding field

I also used **HashMap<String, ASTNode>** in **ASTStruct** (to help in the evaluation, that created the *VStruct*, and on the type checking phases), **TypeBindList** in **ASTTStruct** (to use when checking the associations between each field, its type and the type of the value it stores) and **HashMap<String, IValue>** in **VStruct** (to store the values each field holds in runtime).

3. *Union* Type

Explanation of UnionType implementation in the code

This type had a very similar implementation to *Product Type*, although some differences in some methods and attributes, everything else is the same.

Like the other types, Union was implemented using 3 distinct files:

- **ASTUnion**: represents the node itself and has the ***.eval(Environment<IValue> env)*** and ***.typeCheck(Environment<ASTType> env)*** methods, useful in the evaluation and type checking of the code, accordingly.
- **ASTTUnion**: represents the ***Union Type*** and has the ***.isSubtypeOf(ASTType other, Environment<ASTType> e)*** and ***.simplify(Environment<ASTType> env, Set<String> simplified)*** used to simplify/flatten the type and to check for sub-typing (more on this later)
- **VUnion**: represent the value Union itself and is used at runtime to store each value in its corresponding field

I also used **String + Name** in **ASTUnion** (to help in the evaluation, that creates the *VUnion*, and on the type checking phases), **TypeBindList** in **ASTTUnion** (to use when checking the associations between each field, its type and the type of the value it stores) and **String + Name** in **VUnion** (to store the values each field holds in runtime).

4. Sub-Typing Support

Explanation of the Sub-Typing implementation in the code

In order to add Sub-Typing support to this project, and after many failed attempts involving lists, I decided to add a function to every type that would receive a type and output a boolean (True if the type we are evaluating is a subtype of the one received, false if not). The function is **ASTType.*isSubtypeOf(ASTType other, Environment<ASTType> env)*** and is present in all types, facilitating its call and subtyping evaluation.

5. Recursive Types

Explanation of Recursive Types implementation in the code

Implementing recursiveness in this compiler proved to be very difficult. First of all because of one function that we already mentioned: ***.simplify(Environment<ASTType> env)***. This function initially was created only with the objective of simplifying and allowing for “custom” types to be defined. For example, let’s say we want to create a type called ***testInt*** which is supposed to be equivalent to a normal ***int*** (type testInt = int;). Now, imagine we are defining a variable which is supposed to receive something of type *testInt and we “feed”* it the integer 2 (let var:testInt = 2;), if we do not use this function, we would report a mismatch, because effectively these are different things.

Despite working well and doing precisely what it was made for, this function ended up being a big challenge when creating recursive types, because if it was possible to “simplify” it would do it, no restrictions. This would create a chain reaction that would crash our program due to the infinite “simplifying” possibilities in recursive types.

To tackle this problem, I came up with the idea of somehow keeping track of what we had already simplified or not, by storing the simplified id’s in a set and passing them everytime. I had to change the function's signature that passed to .***simplify(Environment<ASTType> env, Set<String> simplified)*.**

In the end, the only problems emerged when ASTId’s were being misused and not properly “simplified” in nodes like ***ASTApp***, ***ASTFun*** and obviously in ***ASTLetType***. Changing these functions and adapting some things here and there was sufficient to implement recursive types in the compiler.

6. New Tests Created

Some informations about the new tests created

To further enhance the testing capabilities of the test suite, I decided that instead of small, localized tests, I would create fewer, but larger tests. I made this decision because research shows that most faults not detected during testing are the ones more complex and that incorporate many of the program capabilities. Therefore, I ended up creating 4 big, extensive tests that made me feel more confident and secure in my implementation.

7. Conclusion

Resume of the work developed and perspective on where it could be improved

Developing this new compiler really was a challenging, but an interesting one. The final compiler supports many interesting behaviours that already make it very complete, but further expanding it, for example, with the additions of classes and implementing inheritance would make it even more fantastic, in my opinion. The support of the presented nodes was implemented and the types gave it an interesting touch to differentiate it from basic ones.

Overall, I really enjoyed developing this compiler!