Implementing Class Theory for Mathematical Type Checking

Blair M. Durkee
Clemson University
bdurkee@clemson.edu

Abstract—The RESOLVE verifying compiler utilizes a mathematical specification language to define and reason about mathematical models of software components. These models must be backed by theories, and therefore the compiler must implement a rich, expressible mathematical language. One key component of this language is the type system. The RESOLVE mathematical type system has been through several iterations, but none have properly implemented the Class Theory that provides the language with its sound basis. This paper describes the first steps of an implementation that will incorporate a fully developed Class Theory.

Index Terms—software verification, mathematical theory

I. INTRODUCTION

COMPONENT-BASED software in the RESOLVE language is based on both implementation code and mathematical modeling. Programmatic data structures are given mathematical representations by the “is modeled by” keyword in a concept file. These mathematical constructs must be backed by a sound and comprehensive mathematical theory that allows the verification system to reason about the models and the specifications that use them.

A major part of this mathematical framework is a type theory that allows the RESOLVE verifying compiler to quickly check whether certain mathematical statements pass a certain level of “type safety,” much in the same way that programmatic code is type checked. RESOLVE mathematical specifications are based on a version of Class Theory that allows the type system to achieve broad expressibility with relative simplicity and the ability to statically perform sound type judgments. A unique challenge of using a comprehensive Class Theory is the task of translating the pure mathematical concepts into concrete programming structures and algorithms. The implementation must maintain identical logic and consistency and provide mathematical proofs to demonstrate its soundness.

In this paper, we will examine a basic structure for implementing a mathematical type system based on RESOLVE Class Theory in the Java-based RESOLVE verifying compiler. This structure consists of a Java class hierarchy and an auxiliary enhancement. It seeks to approximate the fundamental concepts of the Class Theory while hard-coding only the minimal structures needed for the type system.

II. CLASS THEORY

Before discussion of the implementation, let’s take a brief look at the RESOLVE Class Theory. The Class Theory forms the basis of the mathematical universe in which all RESOLVE mathematical types reside. Figure 1 shows a high-level diagram describing this universe.

Because every type must also have a type, we first must address the issue of where to contain the universe. The top-level container in this universe is HyperSet, which is a meta construct outside the scope of RESOLVE types. It contains a number of HyperSets that form the basis of our usable types. MathEntity contains all of the constructs which are usable in RESOLVE. Cls is a slightly more restrictive HyperSet which contains only the containers (i.e., the MathElements minus the Atoms). Cls is used to create a number of RESOLVE types such as SSet and SStr. SSet contains all sets, and Atom contains the remaining elements, which do not contain any other elements.

III. IMPLEMENTATION

The Java implementation of the RESOLVE Class Theory involves a hierarchy of Java classes that allows for the type system to keep track of the classification of types and leverages the ability of inheritance to mimic subtype relations. Figure 2 shows the class hierarchy this implementation will use.
On the top level is MTType, which is shorthand for the entire mathematical universe (all math entities and meta math entities). Because an MTType can be literally anything, it contains the least information of all the type classes. MTAtom is the type of anything known with certainty to not be container (and therefore cannot be used as a type). It is worth noting that an MTType also cannot be used as a type because of the possibility that it may also be an MTAtom.

MTHyperSet and its descendants represent all of the classes that are known to contain elements, with the exception of MTEmptySet (this is a special case). MTClass is the type of all mathematical classes, and MTSet defines the type of all mathematical sets. Because MTSet inherits from MTClass, this ensures that all sets are recognized as classes as well. At the bottom of the figure is the MTEmptySet class, which is implemented using the Singleton Pattern. This ensures that empty set exists uniquely in the type system.

Finally, the rightmost descendent of MTTtype is MTHyperSet.Application. Typically, the types of function applications are equivalent to the function domain. Because the domain of an arbitrary function may contain atoms, it is not known that all function applications will yield a usable container type. It is therefore not included in the MTHyperSet hierarchy of known containers.

The MTTtype class contains a method called createElement(). This method is used whenever a type is used on the right-hand-side of a colon to define a new symbol. By default, it will simply throw a TypeUnknownToContain-ElementsException, but the method is overridden in the MTHyperSet class. The MTHyperSet method will instead return a new MTTtype object from its createElement() method. Recall that MTType contains the least amount of information, which is appropriate given that the elements of a given HyperSet may be anything—a class, a set, or even an atom.

In RESOLVE, types are first-class citizens, and we want to be able to define new types within the RESOLVE language. To accommodate this, we allow specific instances of MTHyperSet to be assigned an MTFactory object. The MTFactory, as its name implies, uses the Factory Method Pattern to create new types of the specified class. Its code is shown in Figure 3:

```java
public class MTFactory{
    public Class<?> extends MTType> myFactoryType;

    public MTFactory(Class<?> extends MTType> factoryType) {
        myFactoryType = factoryType;
    }

    public MTType createType(String canonicalName) {
        MTType type = null;
        try {
            type = myFactoryType.newInstance();
            type.setCanonicalName(canonicalName);
        } catch (Exception e) {
            // something went very wrong
        }
        return type;
    }
}
```

With this ability, we can assign a MTClass factory to the MTHyperSet object named Cls. This is an axiomatic property that states Cls contains only classes. Similarly, we can assign an MTSet factory to the MTClass object named SSet.

IV. CONCLUSION

Altogether this implementation structure should provide a clean foundation on which to build a sound mathematical type system. It clearly defines which types are classes, sets, or something else, a key distinction in the RESOLVE mathematical language. The type checker that utilizes this structure will be far simpler than previous iterations of the compiler while still maintaining a very expressive mathematical language.

REFERENCES


ACKNOWLEDGEMENTS

This research is funded in part by NSF grants CCF-1161916 and DUE-1022941. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.