Verified Subtyping with Traits and Mixins

Asankhaya Sharma
Department of Computer Science
National Univeristy of Singapore
asankhs@comp.nus.edu.sg

Traits allow decomposing programs into smaller parts and mixins are a form of composition that resemble multiple inheritance. Unfortunately, in the presence of traits, programming languages like Scala give up on subtyping relation between objects. In this paper, we present a method to check subtyping between objects based on entailment in separation logic. We implement our method as a domain specific language in Scala and apply it on the Scala standard library. We have verified that 67% of mixins used in the Scala standard library do indeed conform to subtyping between the traits that are used to build them.

1 Introduction

Traits [8] have been recognized as a mechanism to support fine grained reuse in programming. Several programming languages (Scala, Fortress, Ruby, etc.) support the use of traits in some form or other. Traits and mixins provide support for code reuse and composition that goes beyond classes and inheritance in object oriented programs. In addition, object oriented (OO) programs themselves are notoriously hard to verify in a modular fashion. Recently [4, 11, 6] separation logic based approach has yielded success in verification of object oriented programs. This includes support for verifying inheritance and behavior subtyping, in conformance with OO paradigm. In this paper, we extend the work done on verification of OO programs in separation logic to verify subtyping with traits and mixins.

Below we consider an example that illustrates the problem of subtyping with traits and mixins. The *ICell* trait captures an object with an integer value that can be accessed with *get* and *set* methods. The *BICell* trait provides a basic implementation for *ICell*, while the *Double* and *Inc* traits extend the *ICell* trait by doubling and incrementing the integer value respectively.

```
trait ICell {
 def get() : Int
 def set(x : Int)}
```

```
trait BICell extends ICell {
 private var x : Int = 0
 def get()
    { x }
 def set(x : Int)
    { this.x = x }
}
```

```
trait Double extends ICell {
 abstract override def set(x : Int)
     { super.set(2 * x)}
 }
trait Inc extends ICell {
 abstract override def set(x : Int)
     {super.set(x + 1)}
 }
```

These traits are used in the following class mixins. The integer value field of the objects of *OddICell* mixin is always odd, while the value is even for objects of *EvenICell* mixin.

```
class OddICell extends BICell with Inc with Double
class EvenICell extends BICell with Double with Inc
```

Submitted to:
FSFMA 2014

In the presence of traits, the type system of Scala is not strong enough to distinguish between accepted uses of the traits. This can be illustrated by the following example.

$$\begin{array}{l}\text{def } m\ (c:\ \mathit{BICell} \text{ with } \mathit{Inc} \text{ with } \mathit{Double}):\ \mathit{Int} = \{c.\mathit{get}\}\\ \text{val } \mathit{oic} = \text{new } \mathit{OddICell}\\ \text{val } \mathit{eic} = \text{new } \mathit{EvenICell}\\ m(\mathit{oic})\ \ //\ \mathit{Valid}\\ m(\mathit{eic})\ \ //\ \mathit{Valid}\end{array}$$

The method m can be called with an object of both mixins $EvenIntCell$ and $OddIntCell$, even though the expected object type is a subtype of $OddIntCell$ and not $EvenIntCell$. Thus, the type system in Scala cannot distinguish between the two calls made to method m as it does not check for subtyping between the objects. The key contributions of this paper are two fold, first we present a domain specific language (DSL) which enables separation logic based reasoning in Scala, and second we present a method for checking subtyping in the presence of traits and mixins in Scala using our DSL. The DSL lays the foundations for separation logic based OO verification of traits and mixins in Scala. In section 2, we present an approach based on entailment in separation logic to verify subtyping. In section 3, we present a domain specific language which is embedded in Scala and can support verified subtyping with traits and mixins. We apply our technique to the mixin class hierarchies in the Scala standard library and verify subtyping in 67% of the traits as shown in section 4. Our complete development including the source code of the domain specific language and all examples are available on-line at the following URL.

`http://loris-7.ddns.comp.nus.edu.sg/~project/SLEEKDSL/`

2 Verified Subtyping

We consider a core language based on [4] for formalizing our approach. As shown in figure 1, to simplify the presentation we focus only on type information for traits and mixins while ignoring all other features in our core language. We also assume that all classes are part of mixin compositions and only traits are used to create mixins. Since, existing approaches [4] can handle class based single inheritance, we focus only on mixin compositions in this paper. The rest of the constructs in the core language are related to predicates (Φ) in separation logic. Each trait (and mixin) C can be represented by a corresponding predicate $C\langle v^*\rangle$.

$$\begin{array}{lll}\mathit{mixin} & ::= & \mathit{class}\ C\ [\mathit{extends}\ C_1]\ [\mathit{with}\ C_2]^*\\ \mathit{pred} & ::= & C\langle v^*\rangle \equiv \Phi\ \ [\mathit{inv}\ \ \pi]\\ \Phi & ::= & \bigvee\ (\exists w^* \cdot \kappa \wedge \pi)^*\\ \kappa & ::= & \texttt{emp} \mid C\langle v^*\rangle \mid \kappa_1 * \kappa_2\\ \pi & ::= & \alpha \mid \pi_1 \wedge \pi_2 \quad \alpha ::= \beta \mid \neg\beta\\ \beta & ::= & v_1 = v_2 \mid v = \texttt{null} \mid a \leq 0 \mid a = 0\\ a & ::= & k \mid k \times v \mid a_1 + a_2\end{array}$$

Figure 1: Core Language for Traits and Mixins

Predicates based on separation logic are sufficient to specify mixins because of class linearization in Scala [10]. After class linearization a mixin class composition (unlike multiple inheritance) has a single linear hierarchy. In the case of our running example, the mixins give rise to the following linearizations:

$$OddICell \leftarrow Double \leftarrow Inc \leftarrow BICell$$
$$OddICell\langle this\rangle \equiv BICell\langle this, v\rangle * Inc\langle v, v_1\rangle * Double\langle v_1, \texttt{null}\rangle$$
$$EvenICell \leftarrow Inc \leftarrow Double \leftarrow BICell$$
$$EvenICell\langle this\rangle \equiv BICell\langle this, v\rangle * Double\langle v, v_1\rangle * Inc\langle v_1, \texttt{null}\rangle$$

A mixin class composition can be treated as a single inheritance hierarchy based on the linearization and thus, subtyping between the mixins can be decided by checking the entailment based on separation logic predicates. Class linearization in Scala corresponds nicely with linked list predicates used in separation logic. In case of our running example, the call to method *m* is valid with *oic* object but not the *eic* object as the following entailments show.

$$OddICell\langle oic\rangle \vdash BICell\langle c, v\rangle * Inc\langle v, v_1\rangle * Double\langle v_1, \texttt{null}\rangle \quad Valid$$
$$EvenICell\langle eic\rangle \vdash BICell\langle c, v\rangle * Inc\langle v, v_1\rangle * Double\langle v_1, \texttt{null}\rangle \quad Invalid$$

We now show how the problem of checking subtyping between objects belonging to two different mixins is reduced to an entailment between the corresponding predicates in separation logic. This entailment can be checked with the help of existing solvers for separation logic (like SLEEK [3]). The entailment rule for checking subtyping with traits and mixins is given in figure 2. An object of mixin *D* is a subtype of mixin *C* when the entailment between their corresponding predicates in separation logic is valid.

$$\frac{\begin{array}{c}[\textbf{ENT-Subtype-Check}]\\ class\ C\ [extends\ C_1]\ [with\ C_2]^*\\ class\ D\ [extends\ D_1]\ [with\ D_2]^*\\ C_1\langle this, v_1\rangle[*C_2\langle v_1, v_2\rangle]^* \vdash D_1\langle this, u_1\rangle[*D_2\langle u_1, u_2\rangle]^*\end{array}}{C :> D}$$

Figure 2: Checking Subtyping with Entailment

Entailment checking in separation logic can be used to decide subtyping with traits and mixins. But in order to integrate subtyping support inside Scala we face some engineering challenges. In particular, it is too restrictive and infeasible to do this kind of checking for all the mixins. This requires support for selective subtyping as all mixins will not satisfy the subtype relation. In order to provide the programmer the choice of checking subtyping usage in their methods we have implemented an embedded domain specific language (DSL) in Scala. This DSL uses the SLEEK entailment checker for checking the validity of entailments in separation logic. In the next section we describe the SLEEK DSL and how it is integrated in Scala. We note that our DSL is not limited to checking subtyping but enables generic separation logic based reasoning in Scala. We have also used it to statically verify executable contracts in Scala. We focus only on verified subtyping in this paper as it lays the foundation for OO verification of traits and mixins and helps illustrate a concrete use case for the DSL.

3 Implementation with SLEEK DSL

The implementation of the verified subtyping in Scala with SLEEK has 3 main components:

- a Scala library that supports all SLEEK interactions
- a domain specific language (DSL) implemented in Scala that models the SLEEK input language. With this DSL we get for free embedded type checking in Scala.

– a helper library designed for the Scala interpreter. The library runs SLEEK in interactive mode in the background to provide seamless integration with Scala.

In short, the SLEEK library provides basic functionality for constructing Scala objects representing separation logic formulas. The entailment checking method is in fact the actual front-end for SLEEK. It takes two Scala objects representing separation logic formulas, translates them to the SLEEK input language and invokes SLEEK. The result and the possible residue is captured and parsed using the Scala parser combinator library to extract the Scala representation. To facilitate a better syntax for writing formulas and support for richer interplay with the Scala types we present a domain specific language, SLEEK DSL implemented on top of the Scala library. We will outline the SLEEK DSL by presenting how an entailment checking can be encoded in our DSL.

3.1 SLEEK DSL

As an example consider the following entailment check between two separation logic formulas defined using SLEEK DSL.

$$\text{val } r = \quad \text{x::}\texttt{node}\langle \_, \text{null}\rangle \vdash \text{x::}\texttt{ll}\langle \text{m}\rangle \;\&\&\; \text{m}===1$$

It encodes an entailment between two formulas, one describing a single heap node, an instance of a data structure called *node*. The second formula describes a state in which x is the root pointer of for a data structure described by the *ll* predicate. This predicate abstracts a linked list of size m.

SLEEK DSL relies on the functions defined in the SLEEK Library to create new easy to use operators that provide a more user friendly syntax. A special operator, the double colon (::) is used to describe the points-to relation commonly used for heap nodes. It also provides the usual arithmetic (e.g. $+,-$) and boolean (e.g. $\&\&, ||, ===, !==, \vdash$) operators to help in constructing the separation logic formula. The notation used in the DSL is similar to the one used for SLEEK in [3]. The use of a DSL allows easy intermixing of SLEEK formulas with other Scala types. We use implicit conversions between types (e.g. from *scala.Int* to *formula*[*IntSort*]) to make it even easier to use these formulas in Scala programs.

Furthermore, our library provides a definition for the *isValid* method in the formula class. In order to check the validity of the above entailment it is sufficient to call *r.isValid* which feeds the entailment to SLEEK and converts the result back into a *scala.Boolean* for use as a conditional. Implicit methods provide an easy mechanism to convert from one type of object to the desired type. This enables the support for a SLEEK like syntax within Scala. Formulas allow for a variety of types for the parameters used (such as *x* and *m*). In the Scala library these types are grouped under the following type hierarchy.

```
sealed trait Top
trait BoolSort extends Top
trait IntSort extends Top
trait BagSort extends Top
trait ShapeSort extends Top
trait Bottom extends BoolSort
      with IntSort with BagSort with ShapeSort
```

This trait allows the embedding of the types used in the separation logic formula as Scala types. By defining the various operators using these types, soft type checking for SLEEK formulas is automatically ensured by the underlying Scala type system. The benefit of using a DSL is that it provides a simpler syntax and familiar look and feel for the user of the library. The formula represented by the DSL is also much more concise. The SLEEK DSL allows programmers to verify entailments written in separation logic. In addition, programmers can use the DSL to encode subtyping check as an entailment check in separation logic as described in section 2.

3.2 SLEEK Interactive Mode

The Scala runtime provides a good interpreter for rapid prototyping which can be used from the command line. Similarly, SLEEK also has an interactive mode in which it accepts commands and gives the results back to command line. In order to make SLEEK's interactive mode available to the Scala interpreter, we provide a helper library that hides the extra intricacies incurred by using SLEEK interactively. The benefit of using the interactive mode is that the user defined predicates and data types will not be defined again with each call to *isValid* method. This makes the interactive mode of SLEEK DSL faster when compared to calling the same function from the basic SLEEK library.

Our implementation for verified subtyping integrates into Scala as an API (SLEEK library), as a language (SLEEK DSL) and as an interpreter (SLEEK Interactive mode). This provides programmers the ability to use our procedure in different ways as desired.

4 Experiments

We have used SLEEK DSL to verify subtyping of mixin compositions from the Scala standard library. To the best of our knowledge this it the first such study of subtyping in Scala. The following table presents the results. The first column is the name of the class hierarchy. The second column lists the total number of mixins in the hierarchy, while the third column gives the number of mixins for which we can verify that the subtyping relation holds. The last column gives the percentage of mixins with subtyping.

Class Hierarchy	*Total Num of Mixins*	*Mixins with Subtyping*	*Percentage*
Exceptions	11	11	100
Maths	5	4	80
Parser Combinator	6	6	100
Collections	27	12	44
Total	49	33	67

As an example of mixin hierarchy whose subtyping relations are verified consider the following which represents the maths library in Scala. The only mixin which breaks the subtyping relation is PartialOrdering. Rest of the mixins can be verified to respect the expected subtyping. Thus we have verified that subtyping holds for 4 out of 5 mixins that are part of math class hierarchy.

Equiv is SUPERTYPE of PartialOrdering
PartialOrdering is NOT SUPERTYPE of Ordering
Ordering is SUPERTYPE of Numeric
Numeric is SUPERTYPE of Integral
Numeric is SUPERTYPE of Fractional

5 Related Work

The work that comes closest to our method for checking subtyping is the work of Bierman et.al [2], they provide a mechanism to use SMT solvers for deciding subtyping in a first order functional language. On the other hand, we use SLEEK an entailment checker for separation logic to decide subtyping between traits and mixins. SMT solvers have also been used [1] for verifying typing constraints. Similar to our implementation of SLEEK DSL, the $Scala^{Z3}$ proposal of Köksal et. al [9] integrates the Z3 SMT solver into Scala. Although the integration is similar, the two solvers have different focuses: Z3 is a general SMT solver, while SLEEK is a prover for separation logic.

Another line of work is on specification and verification of traits and mixins. Damiani et. al explore trait verification in [5]. They observe the need for multiple specifications and introduce the concept of proof outline. They support a trait based language with limited composition - symmetric sum of traits and trait alteration. Our work does not directly address the issue of trait verification but checking subtyping is essential part of OO verification using separation logic [4]. We believe that dynamic specifications of [4] along with verified subtyping can be used to verify traits and mixins. Behavior subtyping is a stronger notion of subtyping between objects. The approach of lazy behavioral subtyping [7] can support incremental verification of classes in presence of multiple inheritance. However, this is overly restrictive for mixin compositions in Scala and our method provides a more flexible support for subtyping in Scala.

6 Conclusions

In this paper, we presented a method to enable verified subtyping in Scala. Our method is based on a reduction to entailment checking in separation logic. We implemented a domain specific language (SLEEK DSL) in Scala to enable programmers to check subtyping in their programs. Using SLEEK DSL we carried out a study of the Scala standard library and verified that 67% of the mixins were composed of traits that are in a subtyping relation.

References

[1] Michael Backes, Cătălin Hriţcu & Thorsten Tarrach (2011): *Automatically verifying typing constraints for a data processing language*. In: *Certified Programs and Proofs*, Springer, pp. 296–313.

[2] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu & David Langworthy (2010): *Semantic Subtyping with an SMT Solver*. ICFP '10, pp. 105–116.

[3] Wei-Ngan Chin, Cristina David & Cristian Gherghina (2011): *A HIP and SLEEK verification system*. In: *SPLASH*, pp. 9–10.

[4] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen & Shengchao Qin (2008): *Enhancing modular OO verification with separation logic*. In: *POPL*, pp. 87–99.

[5] Ferruccio Damiani, Johan Dovland, Einar Broch Johnsen & Ina Schaefer (2011): *Verifying traits: a proof system for fine-grained reuse*. In: *FTfJP*, pp. 8:1–8:6.

[6] Dino Distefano & Matthew J. Parkinson (2008): *jStar: towards practical verification for java*. In: *OOPSLA*, pp. 213–226. Available at `http://doi.acm.org/10.1145/1449764.1449782`.

[7] Johan Dovland, Einar Broch Johnsen, Olaf Owe & Martin Steffen (2011): *Incremental reasoning with lazy behavioral subtyping for multiple inheritance*. *Sci. Comput. Program.*, doi:10.1016/j.scico.2010.09.006. Available at `http://dx.doi.org/10.1016/j.scico.2010.09.006`.

[8] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts & Andrew P. Black (2006): *Traits: A mechanism for fine-grained reuse*. *ACM Trans. Program. Lang. Syst.* 28(2), pp. 331–388. Available at `http://doi.acm.org/10.1145/1119479.1119483`.

[9] Ali Sinan Köksal, Viktor Kuncak & Philippe Suter (2011): *Scala to the power of Z3: integrating SMT and programming*. In: *CADE*, pp. 400–406.

[10] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon & Matthias Zenger (2006): *An Overview of the Scala Programming Language*. Technical Report, EPFL.

[11] Matthew J. Parkinson & Gavin M. Bierman (2008): *Separation logic, abstraction and inheritance*. In: *POPL*, pp. 75–86. Available at `http://doi.acm.org/10.1145/1328438.1328451`.