Verifying Scala's Vals and Lazy Vals

Bachelor Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

By: Simon Fritsche
simonfri@student.ethz.ch

Supervised by: Malte Schwerhoff
Prof. Dr. Peter Müller

Date: August 27, 2014

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Contents

1 Introduction

Software correctness is a property that is much desired yet hard to achieve. Especially in environments where small errors in a program can lead to big damage, a prove of the correctness of a piece of software would be ideal. There are a lot of examples where such a prove could have saved a lot of money or even human lives. Two of the most popular are the Ariane 5 bug [5] and the Patriot accident [12]. This shows that correctness of software is not only a problem for theoretical computer science but a problem that affects daily life.

1.1 The Viper Project

The Viper Project (Verification Infrastructure for Permissoin-Based Reasoning) is a project of the Chair of Programming Methodology at ETH Zurich. The goal of this project is to provide a verification infrastructure around its intermediate language Silver. Silver is based on permission logic such as separation logics[18] and implicit dynamic frames[19]. The Viper project also contains two back-end verifiers for Silver named Silicon and Carbon. Silicon is a verifier based on symbolic execution and Carbon is based on verification condition generation. Both Silicon and Carbon use Microsoft's theorem prover Z3 [8]. The goal of this infrastructure is to reduce the effort needed to build permission-based verifiers by allowing developers to encode higher level constructs into Silver and therefore work at a higher level of abstraction. For more information about Viper one can have a look at the Viper tech report [10].

1.2 Scala

As already hinted in the title, this thesis aims at verifying Scala features. But why Scala and not any other programming language? Scala is a new up and coming programming language that has many interesting features. These features are very often challenging to verify and therefore work well as research topics. Lazy initialization is one example of such a challenging feature and its verification will be addressed in this thesis.

1.2.1 Scala2Silver

Scala2Silver (formerly Scala2SIL, it has been renamed) is the project of a former Master's thesis written by Bernhard Brodowsky [7] and the basis of our work. Scala2Silver uses the Viper infrastructure to implement a verifier for Scala features. These Scala features are enriched with contracts by the user and then encoded in Silver code. Brodowsky's work laid out the foundations for a verifier, but there are still many things to be done.

Figure 1 shows the Scala verification toolchain with Scala2Silver on top of the Viper toolchain.

1.3 Goals of this thesis

This thesis brings Scala2Silver one step further by improving the translation of Scala's lazy features.
Verification for Scala's lazy vals is currently existing but limited. Our approach to overcome these limitations is to first look at large software projects and evaluate which limitations are acceptable in practice and which are too strict. Based on these insights we will improve the translation of lazy vals. Furthermore, the testbench of Scala2Silver will benefit from the gained knowledge and some test cases based on real world programs will be added.
The encoding of Scala's constant variables, also called vals, will be unified with the encoding of lazy vals, in order to make reasoning about them easier and make them benefit from the same improvements and features. A further achievement of this thesis is the translation of another

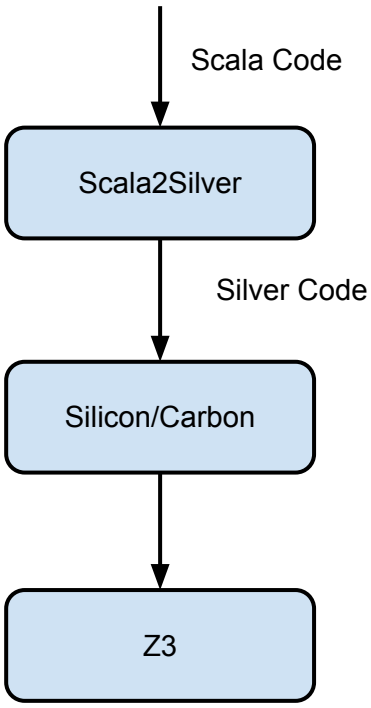

Figure 1: Scala verification toolchain

one of Scala's lazy features, namely singleton objects. A translation for them is currently not existing and this thesis will introduce it.

2 Silver

Silver (or Sil in [7], it has been renamed) is a simple intermediate verification language developed at ETH. The language was influenced by Boogie[4] and Chalice [11], both verification infrastructures.
Silver provides basic programming features such as methods or loops. Silver also provides controlled access to the heap by having a permission based verification approach.
The following subsections give an overview over Silver features that are required for our work on translating Scala's *vals* and *lazy vals*. For further reading on Silver confer the Viper tech report [10].

2.1 Fractional Permissions

Permissions in Silver are used to control access to heap locations such as fields and *abstract predicates*. Permissions to a heap location are represented by rational numbers. In a Silver program they are denoted by *access predicates*:

```
acc(c.f, v)
```

Here we have access to field `f` of object `c` with permission `v`. The permission value `v` determines the kind of access we have. This is a real value and has to be between 1 and 0, where 1 denotes write access and 0 denotes no access. Read access is everything in between. Note that value `v` can be left out denoting write permission.
Permissions can be transferred to other program contexts. One example for this are method calls. Since a method body will most likely use heap operations it also needs to have permissions for the required access locations. These permissions cannot be created out of nowhere but need to be obtained from the caller. This is done by specifying the required permission amount in the precondition. On calling a method, permissions specified in the precondition get transferred to the callee. After the execution of the body, the caller receives the permission amount specified in the postcondition. If the callee demands a higher amount of permission than the amount the caller currently has, it will lead to an error.

The two basic statements for transferring permissons are the *inhale* and *exhale* statements. Everything that transfers permissions can be expressed with inhale and exhale.
Inhale The inhale statement is an analogue to the assume statement of first order verification languages. `inhale A` will assume the information provided by an expression in `A` and will add any amount of permission specified in `A` to the program state.
Exhale The exhale statement is the counterpart of the inhale statement and is comparable to the assert statement of first order verification languages. `exhale A` will assert every expression in `A` and remove any amount of permission in `A` from the program state.
With this information one can see that a method call with precondition `pre` and postcondition `post` is equivalent to this:

```
exhale pre
inhale post
```

One special kind of permission are wildcard permissions. Chalice has them as well, however in Chalice they are denoted as rd^* and not as wildcards. They were introduced because it is cumbersome to carry around arbitrarily small fractions of permissions and to help with writing specifications that only use read permissions. Wildcard permissions look like this in a Silver Program:

```
acc(c.f, wildcard)
```

Wildcards denote some amount of permission being strictly smaller than the amount currently held and always larger than zero. The exact value is unknown. Each occurrence of a wildcard

permission is interpreted as a fresh permission amount. This means that once wildcard permissions to an access location are given away, one can never have the same amount of permission again and therefore will lose write access forever, which renders the access location immutable.

For further reading about fractional permissions, consult the Fractional Permissions Paper of Boyland [6]

2.2 Types

Silver has a very simple type system that supports only 4 different primitive types: `Int, Bool, Ref, Perm`. `Int` denotes an unbounded integer, `Bool` represents a boolean value. `Ref` represents every reference type in Silver and is used to refer to objects. `Perm` is a type denoting permission amounts. Silver also has collection types such as sets and sequences, see [10] for further information.

2.3 Objects and Fields

In Silver objects are created by using the `new` keyword. Since Silver has no differently typed objects (just the `Ref` type), there is a different way of creating object than instantiating classes. Field names are global and not bound to one object. In fact, the only thing that indicates that an object has a field is permission to the field of the object, like in `acc(x.f)` we know that field f belongs to object x. Such permissions can be created with Silver's `new` keyword. It takes each field as a parameter and returns full permissions that describe the field object relation. Fields have to be declared before use by using the `field` keyword. This is best seen by an example:

```
field a: Int
field b: Bool

...

//client
var newObject: Ref
newObject = new(a,b)
```

In the first two lines we see the declaration of two fields `a` and `b` with their respective type. Then a new object is created with both fields `a` and `b` which is then assigned to variable `newObject`. After this object creation write permissions to all fields, in this case `acc(newObject.a, write)` and `acc(newObject.b, write)`, are added to the program state.

2.4 Methods

Methods in Silver are similar to functions in Java, C# or other common programming languages. They can take parameters and return some result. Since Silver is a language designed for verification purposes, one can add pre- and postconditions to it and the verifier will check if the postcondition follows from the precondition and the method body. A method in Silver looks as follows:

```
method methodName(parameter:  Type) returns (vresult:  Type)
   requires P
   ensures Q
{
   Body
}
```

Pre- and postconditions have to be side-effect free and self-framing which means intuitively, that the specification needs to be self-contained and e.g. no `acc(x.f, p > 0)` can be in a contract without an `acc(x, p > 0)` and `x != null`. Further explanations to the self-framing concept can be found here [7, S. 12]

2.5 Abstract Predicates

Abstract Predicates [17] in Silver are constructs that take arguments and have assertions as their bodies. These assertions can contain permissions. The assertions of an abstract predicate have to be self-framing like pre- and postconditions have to be. Predicates can be used for information hiding and are needed to provide permissions for recursive data structures such as linked lists.

```
predicate pred(param: Ref)
{
   acc(param.f, 1/2) &&
   param.f == 3
}
```

This predicate encapsulates the information of read access to field `f` of parameter object `param` and states that the field has value 3. Note that `param != null` is checked implicitly and therefore does not have to be added to the predicate.
As stated in chapter 2.1 one can have access permission to a predicate. These permissions can also be transferred like permissions to heap locations, for example by calling a method that includes such an access predicate in its pre- or postcondition:

```
method foo(arg: Ref) returns (vresult: Int)
   requires acc(pred(arg))
   ensures acc(pred(arg))
{
   // Use pred in some way.
}
```

There are two statements and one expression for the usage of predicates:
Fold: The `Fold` keyword is used to create an instance of a predicate. This means that after folding the predicate one will have access to an instance of the folded predicate and the permissions that are now encapsulated in the predicate are removed from the program state (exhaled).
Unfold: The `Unfold` statement is the counterpart of the fold statement. When unfolding a predicate one will receive the information it encapsulates i.e. inhale the body of the predicate.
Unfolding: The `unfolding` expression takes two parameters, a predicate and an expression. The syntax looks like this: `unfolding acc(pred(arg)) in E` where `E` is an arbitrary expression. Semantically this statement unfolds the predicate, evaluates the expression and folds it again. However `unfolding` is not primarily used to save lines of code. In contrast to `Fold` and `Unfold`, which are statements, `unfolding` is an expression. This means that `unfolding` can be used in pure contexts.

One could write a method body for `foo` like this:

```
method foo(arg: Ref) returns (vresult: Int)
   requires acc(pred(arg))
   ensures acc(pred(arg))
   ensures vresult == 3
{
   unfold acc(pred(arg))
   vresult = arg.f
```

```
   fold acc(pred(arg))
}
```

or

```
method foo(arg: Ref) returns (vresult: Int)
   requires acc(pred(arg))
   ensures acc(pred(arg))
   ensures vresult == 3
{
   vresult = unfolding acc(pred(arg)) in arg.f
}
```

It is not required to always specify full permission (value 1) in the access predicate and in the fold/unfold statements. One can have an arbitrary permission amount to a predicate. When folding or unfolding a predicate, the permission value of the predicate gets multiplied with the permission value of the heap location in the predicate. This means that if we have `acc(pred(arg), 1/2)` and we unfold it, we get `acc(arg.f, 1/2 * 1/2)` which is the same as `acc(arg.f, 1/4)`.

3 Scala Features

This section gives an overview over the Scala features whose verification will be addressed in this thesis. Note that everything that is related to contracts and permissions is specified for verification purposes only and does not have any effect at runtime. For further reading about Scala one should consider taking a look into "Programming in Scala"[16]

3.1 Vals

Vals and vars are the two kinds of fields defined in Scala. Vars behave like mutable variables known from C like programming languages. Vals are immutable and similar to Java's final variables. However in contrast to Java's finals, both have to be initialized right away. Here is an example initialization of a val and a var:

```
var x = 3 // Mutable
val y = 3 // Immutable
```

The code to the right of the assignment symbol (=) is called the initializer or the right-hand side of a field and can be an arbitrary block of Scala code.
Vars and vals can also be used to declare local variables but this thesis is only concerned with their application as fields.
An interesting feature for immutable variables, which is not yet implemented, are field invariants. A field invariant is a property that always holds for a field after it is initialized. In the example above, a possible field invariant would be that `y` equals 3. Such invariants are helpful for the verification of immutable data structures in general because every time the structure is accessed, one can assume the field invariant. For example, every time `y` is used we can assume that `y` equals 3 without this actually being written into contracts.

3.2 Lazy Vals

Lazy vals are a special kind of vals. They get initialized on first use. This means that the complete initializer of a lazy val gets executed on first use of the according lazy val. Therefore, lazy vals are often used for speeding up program startups and to improve performance in general by delaying or avoiding expensive computations.
However, the drawback of lazy vals is that it is quite challenging to find a Silver encoding for them. This is due to the fact that in a permission based language, permissions are needed in order to evaluate expressions and lazyness makes it hard to tell when the initializer is executed, hence permissions would need to be available wherever the initialization could happen. We will demonstrate their initialization on first use property with a small example:

```
class Foo {
   val x = { println("val"); 1}
   lazy val y = { println("lazyVal"); 2}
}
```

Both `x` and `y` get initialized to the result of a block of code. These code blocks behave like a normal block of Scala code and the expression at the end (in this case an integer) is the return value of the code block. Now if we execute the following code:

```
val newObject = new Foo
println("break")
newObject.y
```

then the observed console output is:

```
val
break
lazyVal
```

Here we can see by looking at the order of the output strings that `x` is initialized when a new `Foo` object is instantiated. Then `break` is printed, which indicates that the constructor of `Foo` has finished. Line 3 in the code example above reads the lazy val and finally `y` gets initialized and string `lazyVal` is printed.

3.3 Singleton Objects

Singleton objects are a special Scala construct that is similar to Java's static classes. A singleton object cannot be instantiated, unlike normal classes. Instead, everyone uses the same given instance. Singleton objects are globally accessible. Therefore, they are often used as global containers of methods and data.
This thesis deals with singleton objects because one of their properties is that they get initialized lazily. More precisely, the whole object gets initialized on first use, where first use means on assigning the object to a variable or accessing one of its members. Therefore, singleton objects come with similar challenges as lazy vals.
Here is some example code that should help with understanding singleton objects and their laziness property.

A simple singleton object:

```scala
object Foo {
   println("constructor")

   val x = {println("init_x"); 5}
   var y = {println("init_y"); 4}
}
```

A program that uses this Singleton Object:

```scala
def main() = {
   println("start")
   val y = Foo
   println("end")
}
```

The output of the program:

```
start
constructor
init_x
init_y
end
```

Here we see that on first use of the singleton object, the constructor is called **and** each member is initialized.

3.4 Contracts and Permissions in Scala

Since we are writing code that is intended to be verified with an automated verifier, one has to be able to add contracts to Scala code. With contracts we mean pre- and postconditions for methods, loop invariants, class invariants etc. Since Scala does not support contracts natively, Rokas Matulis, a former master's student at ETH, developed a code contract library called

CC4S (Code Contracts for Scala)[13] inspired by Microsoft's Code Contracts for .NET[14]. These contracts result in no-ops and hence have no effect at runtime. However, they can be used to reason about the program, for example by performing static verification (as we will do), or by performing runtime verification (as in [13]). The most important features from CC4S for this thesis are pre- and postconditions for methods.
Here is an example of how contracts are written in Scala using CC4S:

```
def method(p1: Type1, ... ) = {
   requires( P )
   ensures( Q )

   Body
}
```

As in Silver, `requires` denotes the precondition, in this case assertion `P` and `ensures` denotes the postcondition, in this case assertion `Q`.
When writing permissions one can use access predicates as in Silver, e.g. `acc(c.f, 0.5)`. Note that decimal representation is used for permission values but this is merely a cosmetic difference. Scala2Silver translates decimals into fractions for the Silver code. Additionally, there is a `read(*)` assertion denoting an abstract read permission amount if `*` is a var and wildcard permissions if `*` is a val or lazy val.
This means for vars that each occurrence of `read(x.f)` in a precondition postcondition pair denotes the same (unknown) amount, whereas each occurrence of `acc(x.f, wildcard)` denotes a potentially different (unknown) permission amount. This has the consequence that a caller calling a method with `read(x.f)` in its pre- and postcondition knows that it holds the same permission amount to `x.f` after the call as it held before the call, whereas calling a method with `acc(x.f, wildcard)` in both pre- and postcondition does not allow the caller to assume that it received the same permission amount to `x.f` that it gave to the callee, which effectively renders `x.f` immutable. For more information about abstract read permissions see [9]). There is also a `write(*)` assertion that denotes write permission to `*`, i.e. it is syntactic sugar for `acc(*, 1)`

```
class Node {
   ensures(write(value))
   ensures(write(next))

   var value: Int = 0
   var next: Node = null

   def copyNode(n: Node) {
      requires(read(n.value) && read(n.next))
      requires(write(value) && write(next))
      ensures(read(n.value) && read(n.next))
      ensures(write(value) && write(next))
      ensures(value == n.value)
      ensures(next == n.next)

      value = n.value
      next = n.next
   }
}
```

Here one can see pre- and postconditions in action. The contracts at the beginning of the class body are the pre- and postconditions of `Node`'s constructor.

4 Existing translation of Vals and Lazy Vals

This section explains the already existing translation of vals and lazy vals. This knowledge is needed because our approach is based on what has been done before and improves this translation.

4.1 Encoding of Vals

Vals are encoded as Silver fields. As already mentioned in 3.1, our thesis only deals with val fields, hence the rest of the content will only be about val fields. There are two key properties of Scala's vals that need to be encoded in Silver. The first one is that they are initialized at the same place where they are declared. The second property is that they are read-only.
The initialization is encoded as an assignment in the constructor of the object that declares the val. Note that the constructor is called after the object creation and therefore requires write permission to all fields that are initialized. After the initialization the field will be rendered immutable by returning wildcard permissions to the caller of the constructor.

```
class Foo {
   ensures(read(v) && v == 2)

   val v = 2
}
```

gets encoded as

```
method Foo_constructor(this: Ref)
   requires this != null
   requires acc(this.v)
   ensures acc(this.v, wildcard) && this.v == 2
{
   this.v := 2
}
```

4.1.1 Desired properties for Vals

Although the translation of vals is a rather simple task, there are some approaches that would reduce the annotation overhead required for using vals. Currently one has to put all the information (permissions and assumptions on vals) into contracts.
Consider a field invariant for immutable data structures. This field invariant could be assumed wherever the val is read, providing all the relevant information about the val. This is not yet implemented and is a problem that will be addressed in this thesis.
Another desired feature would be that one can read vals without having to specify read access explicitly. In the Silver translation this could be done by inhaling wildcard permissions because no matter how often one inhales wildcard permissions on an access location, the permissions will never add up to write permission and the val is guaranteed to remain immutable. The problem that currently prevents this feature from being implemented is that one could access a val before it is actually initialized in the constructor and therefore use a field invariant that has not yet been established.

```
class Bar {
   m()

   val v = 5
```

```
   def m() = {
      assert v > 0
   }
}
```

If method `m` assumed the field invariant of `v` (`v == 5`) and the assertion would hold. However, at runtime `m` is executed before `v` is initialized and therefore before the field invariant is established. This means that in method `m`, `v` has default value 0 and the assertion does not hold. Therefore the verification is unsound. In the rest of this thesis i will refer to this problem as the "Initialization Problem".

4.2 Translation of Lazy Vals

4.2.1 Basic Idea

As hinted in chapter 3.2, the implementation of lazy vals is quite challenging. The main problem is that one does not know when the side effects of the initializer will be executed and when the actual assignment takes place. In [7] this problem is addressed and since our improvements build up on this approach we provide a detailed description of it.
The main idea of lazy vals is that it does not matter when they are evaluated, so one can save or delay the computation time of the initializer by declaring them lazy. Therefore, the right hand side has to stay constant after some point in time, which means that it evaluates to the same value no matter where it is used first. Furthermore, side-effects like field assignments are dangerous to have in the initializer of a lazy val because if one does not know when the lazy val is initialized one does not know when these side-effects are executed, which makes reasoning about the correctness of the code very hard. Therefore, the existing work forbids side effects for lazy val initializers. Note that object creation, which is usually considered a side effect, is not forbidden because it is not observable from the outside that an object creation takes place. Only assignments to access locations outside of the scope of the initializer can be noticed.

4.2.2 Forcing the initializer to remain constant

In the section above we stated that the right-hand side of a lazy val should remain constant but this is not enough. Verification needs a guarantee and we need to enforce this.
By discarding a fraction of the access permission to each heap location the initializer depends on, we achieve that the initializer will always evaluate to the same value in the future, because all locations the initializer depends on are now immutable. This preserves read access and turns write access into read access. This is a drawback, especially if one wants to involve vars in the computation of the right hand side, but as described above it is considered bad practice and this limitation has to exist in order to enable verification of lazy vals in our solution. Here is an example that illustrates how the permissions are lost.

```
var foo: Int
foo = 1 // Fine, we got write access to vars
lazy val x = foo // exhale acc(foo, wildcard) happens
foo = 2 // Invalid, no write access to foo anymore.
```

We cannot write to `foo` anymore after the declaration of `x`, because otherwise the right-hand side of `x` would not remain constant. Since we ensure this by exhaling some permissions, we no longer have write permissions after the initialization of `x`.

4.2.3 Delaying Evaluation

With the approach described up to now, the Silver translation is only possible if the lazy val gets initialized immediately after declaration, which is obviously not the general case. Therefore, a possibility to delay the evaluation of the initializer until the point in time where the initializer remains constant is provided. This is done with the `commit(*)` method.
Each lazy val has to be committed at some point in time by calling the commit method with the according lazy val as parameter. After this call the evaluation of the right-hand side remains constant as described above. To track if a lazy val is not committed twice and not read before it is committed, an `uncommitted(*)` assertion is introduced. This assertion has to hold before `commit(*)` is called, at which point it is invalidated, and then has to be invalid every time the lazy val is read. It also takes the lazy val as parameter. The `uncommitted(*)` assertion can be put into contracts and has to be carried around until the commit happens. This is our current solution to deal with the Initialization Problem described in 4.1.1
Note that the check `!uncommitted(*)` when reading a lazy val is done automatically and therefore does not have to be added in the Scala code by the user. After calling the commit method a lazy val can be treated like a normal val. An example showing a delayed commit:

```
class Foo {
   ensures(uncommitted(y))
   ensures(write(x) && x == 5)

   var x = 5
   lazy val y = x
}

class Main {

   def main() {
      val foo = new Foo
      foo.x = 7
      commit(y)
   }
}
```

Assume that method `main` in class `Main` is the starting point of this example. Although `x` has value 5 at the point where `y` is declared, `y` will have value 7 after the commit because the commit happens after `x` has been changed to 7. Note that `x` becomes immutable after committing `y` because wildcard permissions to `x` are exhaled during the commit. It follows that `y` is ensured to be initialized to the from now on immutable value of `x`.

4.2.4 Complex Initializers

The commit statement is encoded in Silver as a method call. A detailed translation of this commit is shown in the next subsection. The challenge of writing the commit method is that in order to discard permissions to all locations accessed by an initializer, one needs to know which these are.
For simple initializers like an integer or a field it is not a big problem to compute these access locations, but for complex initializers, i.e. if they consist of an arbitrary block of code, including method calls, this computation can become a very difficult task. Therefore, the decision was made to restrict the form of the initializer.
Our restrictions are: At most one method call. Furthermore, this method call (if there is one) has to be the outermost expression. Some examples that should give an intuition of what is not allowed (These examples are taken directly from [7] to be able to compare them later on

with our new translation):

```
def f(x:Int) = x

def m() = ...
def n() = ...

// Several method calls are not allowed .
lazy val b = m() + n()

// Method call has to be the outermost expression .
lazy val b = f(m())
lazy val b = m() + 1
```

4.2.5 Translation into Silver

Since our work builds on the translation into Silver we will also go into the details of [7]'s Silver translation. First of all, for each lazy val two fields will be generated. One field holds the value of the lazy val and the other field is a flag that indicates whether or not the lazy val has already been committed. Furthermore, each lazy val has a corresponding commit method. The encoding of this method is best explained with a translation example:

```
class Bar {
   val v = 7
   lazy val f = v
   commit(f)
}
```

The translated Silver code looks like this (parts that are unimportant for the lazy val translation are left out):

```
field f: Int // value of Bar.f
field unc_f: Bool // true iff Bar.f not yet committed

method commit_f(this: Ref)
   requires(acc(this.v, wildcard))
   requires(acc(this.unc_f, write))
   requires(acc(this.f, write))
   requires(this.unc_f)
   ensures(acc(this.v, wildcard))
   ensures(acc(this.unc_f, wildcard))
   ensures(acc(this.f, wildcard))
   ensures(!this.unc_f)
   ensures(this.f == v) // field invariant
{
   this.f := v
   this.unc_f := false
}
```

The top two lines are the fields that hold the value of `f` and the uncommitted field of `f`. In the real translation the naming scheme is different in order to avoid ambiguities between lazy vals.
In the precondition of `commit_f` we have to make sure that we get read access to all resources that are required for the evaluation of `f`'s initializer. In this case it is only field `v`. Then we must have write access to both fields of the lazy val because both will be changed by the

commit method. If the initializer contained a method call, the precondition of this call would also be in this list of requires statements.
In the postcondition we return read access to all required fields but we only return read access instead of write access to both fields of the lazy val since they are not allowed to be changed anymore. In the body of the commit method the initialization of the lazy val is performed, and in the last statement the committed field of the lazy val is set.

5 Empirical Study

One part of this thesis was an empirical study. The goal of this study was to understand the use cases of lazy vals better and thereby learn how severe the limitations are in practice. In order to get a good statistical base we looked at the sources of 5 different programs written in Scala that serve very different purposes. The following section will give a short description of each of them. LOC means lines of code and LV the number of lazy vals the project contributed to the study.
Note that LV does not equal the amount of lazy vals in the project because we decided to ignore lazy vals in configuration files that get assigned to primitives after some time.

5.1 Projects

5.1.1 Akka

Akka is an open source library that simplifies the implementation and design of concurrent and distributed programs. The key features of Akka are actors for solving concurrency problems. Akka's actors talk to each other by asynchronous message passing. Akka also provides a hierarchy for their actors which makes the implementation of supervising actors and error handling much easier.
The Akka version used in our study is: Akka 2.2.3
LOC: 389813 LV: 181 [1]

5.1.2 Lift

Lift is an open source web application framework that is designed for programming web applications in Scala. Lift is designed to support the actor concept for high performing and responsive web applications.
The Lift version we worked with is Lift 2.6
LOC: 154186 LV: 42 [2]

5.1.3 Sbt

Sbt is an open source build tool for projects written in Java and Scala. Among other things, Sbt provides dependency management via Ivy (Maven) repositories as well. Sbt is the most used Scala build tool every project mentioned in the empirical study and also the project of this thesis, Scala2Silver, use Sbt.
Our analyzed version is: Sbt 0.13
LOC: 82706 LV: 56 [3]

5.1.4 Silicon

Silicon is a symbolic-execution based verifier for Silver. Silicon is part of the Viper project.
LOC: 58066 LV: 33 [10]

5.1.5 Silver

Silver is an intermediate language and like Silicon part of the Viper project. Silver was presented in chapter 2
LOC: 78092 LV: 51 [10]

5.2 Categories

Since an empirical study about the use of lazy vals could be a dedicated thesis itself, we decided to focus on the amount of lazy vals looked at and therefore did not analyze the lazy vals in detail. This led to a relatively large sample of lazy vals and a categorization of lazy vals into the following 7 categories. The categories are ordered from the most general to the most specific. Each lazy val is put into the most specific category.

5.2.1 Code Block

Lazy vals can be initialized by arbitrary code blocks, which results in the code block being executed on first use of the lazy val.

5.2.2 Method Call/No Side Effect

This category is for lazy vals where the initializer is a special method call. The method call is special in terms of having no side effect (except the creation of new objects, which we said was permitted in chapter 4.2.1).

5.2.3 Method Call/Side Effect

This category collects lazy vals where the initializer is a method call that has side effects.

5.2.4 Immutable Collection

A lazy val belongs into this category if its right hand side is an immutable collection, or there is a computation on the right-hand side that involves an immutable collection. Lazy vals are often used in this way because computations on immutable collections are expensive and one wants to ensure that these costs only occur when the results are actually used.

5.2.5 Field

Everything that gets assigned to a field is put into the Field category. This field can belong to the same class or can also be in another class.

```
class Foo {
   val f = // some initializer

   lazy val x = f
}
```

5.2.6 Object

The Object consists of initializers that are new object creations and primitive types like strings, integers and functions (since in Scala functions are treated as objects). Some examples:

```
lazy val obj1 = "Error"
lazy val obj2 = new Project()
lazy val obj3 = 5
```

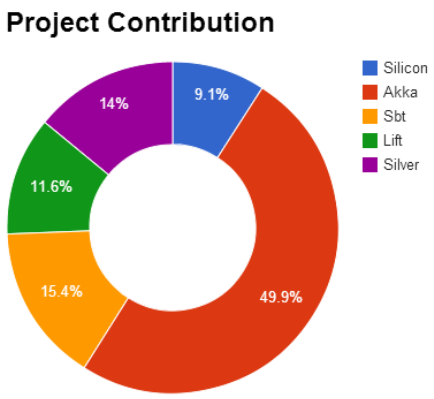

Figure 2: Contribution of each project to the empirical study

5.2.7 Override

Category Override contains each lazy val that overrides another Lazy Val from a super class. The motivation for this category was that we wanted to find out how important it is to support this feature.

5.3 Statistics

I collected the gathered information in a spread sheet which allowed us to create two diagrams. Diagram 2 concerns the distribution of the lazy vals over the projects. Since the projects are different in size and purpose, they also gave a different amount of examples. Diagram 3 shows more interesting information, namely the size of the categories and therefore the categorization of the use cases. In total we looked at 350 different lazy vals.

5.4 Conclusion

If we look at figure 3, we can see that our current restriction that limits the use of method calls with side effects does not cripple the programmer, as it only accounts for less than one percent of our samples. However, method calls that do not have side effects are quite common. Another thing that stands out is that lazy vals are most of the time assigned to initializers that consist of a simple call to `new`. This is the case because very often lazy vals are used in configuration files for error messages or file paths and are therefore initialized to strings or file objects etc. This case of lazy vals can currently be handled.
To cover most of the use cases one needs to support arbitrary forms of method calls that have no side-effects because they make up a substantial amount of the use cases we found.
We also mentioned in the introduction that we will take examples from this empirical study and enrich the Scala2Silver test suite with them. They are described in section 9 at the end of this thesis.

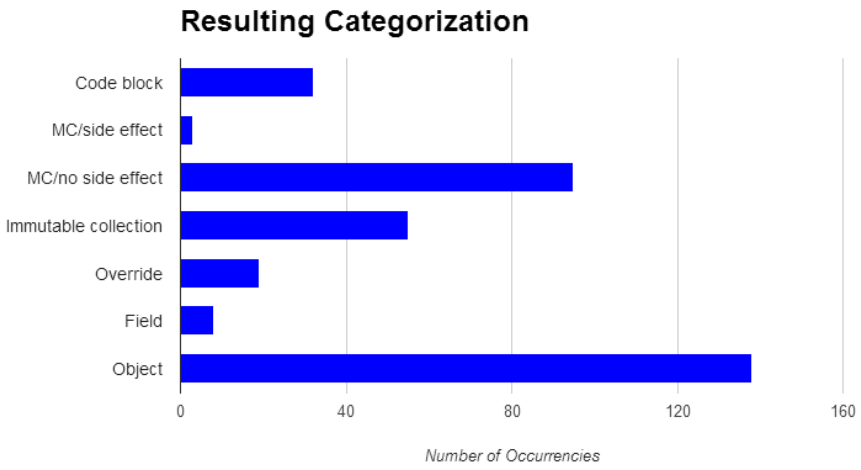

Figure 3: Categorization of lazy vals

6 Translation of Vals and Lazy Vals

One main goal of this thesis was to use the insight gained during the empirical study to determine which restrictions imposed on the specifications of vals and lazy vals described in section 4 should be lifted because they exclude use-cases that are relevant in practice. This section describes both the theoretical side as well as the implementational changes that were made.

6.1 Specifying Lazy Vals

In the conclusion of the empirical study we assessed that by supporting arbitrary (side-effect free) code blocks one can cover most of the use cases we found. Therefore, we want the initializers of a lazy val to be such a code block.
However, the problem that caused us to forbid such complex initializers in the first place still remains: How can we generate contracts for a code block? We decided to go for the simple solution first: let the user do it. Generating them automatically remains as future work. We already have a contract library that enables us to put contracts into the code block of a method call, hence there is already a framework that allows the user to specify contracts for our code block initializer.
Note that the restrictions for the precondition of the code block and for the precondition generated for the commit method as described in chapter 4, remain the same: A small permission amount to every access location in the precondition is exhaled to make sure that the initializer stays constant and is side-effect free.
A new feature that we want to introduce for lazy vals that was already mentioned in chapter 3.1 are field invariants. A field invariant `I` can be added to a code block initializer by adding the new contract `fieldInvariant(I)` to the code block. Note that the field invariant is always assumed along with read access to a lazy val. To make sure that we do not arrive at an invalid program state by assuming permissions that would add up to a value greater than 1, field invariants are scaled with wildcard permissions. This means that putting write permissions or any fixed permission amount into the field invariant is legit, but one will never get the same amount of permission out of it again.
The decision of adding a field invariant is also supported by our empirical study because most lazy vals are initialized with an immutable object, about which we can preserve assumptions by putting them into the field invariant.
An example of such a code block initializer with contracts:

```
lazy val x = {
   requires(P)
   fieldInvariant(I)

   Body
}
```

`P` and `I` are arbitrary assertions. `Body` contains arbitrary (side-effect free) code that is used to initialize the lazy val. Note that object creation is not considered a side effect in this context and is therefore permitted in `Body`.

6.1.1 Referring to Lazy Vals in Initializer Contracts

Since we want to describe the properties of lazy vals in the field invariant we need to have something that enables us to refer to lazy vals themselves. For this purpose one can use the `result` keyword known from postconditions of methods.

```
lazy val x = {
```

```
   fieldInvariant(result == 4)
   4
}
```

6.1.2 Using Lazy Vals

The usage of lazy vals remains the same as before.
A lazy val has to be committed at some point in the program by calling `commit(*)` and cannot be used before the commit. Calling the commit method is equivalent to exhaling the precondition and inhaling read access to the lazy val. Before the commit an `uncommitted(*)` assertion needs to be carried around in contracts. This `uncommitted(*)` assertion gets added to the program state after the declaration of the lazy val and needs to be present in the program state when committing the lazy val. After the commit, read access is interpreted as the field invariant along with read access to the lazy val itself. This read access then has to be present wherever the lazy val is used by putting it into contracts.
We want to show an example that demonstrates that the newly added features are a powerful improvement and show how the features are used.

```
class Cell(val f: Int) {
   ensures(read(f))
}

val y = 3

lazy val x = {
   requires(read(y) && y == 3)
   fieldInvariant(read(result.f) && result.f == 10)

   new Cell(calcMethod(y))
}
commit(x)
main()

def calcMethod(param: Int) = {
   ensures(result == param + 7)
   param + 7
}

def main() {
   requires(read(x))
   assert x.f == 10
}
```

This example is not possible to translate with the existing translation because lazy val `x` contains a method call which is not the outermost expression. With our new translation, however, this works fine.
The other new thing in this example is the field invariant. In the translation described in chapter 4 one had to carry around `read(x) && read(x.f) && x.f == 10`, whereas now it is only `read(x)` which is interpreted as `read(x) && fieldInvariant(x)`.
To show that this new translation is actually an improvement we can see that each forbidden example we gave in chapter 4 is now working:

```
def f(x:Int) = x
```

```
def m() = ...

def n() = ...

// Several method calls are now allowed.
lazy val b = {
   m() + n()
}

// Method call does not have to be the outermost expression.
lazy val b = {
   f(m())
}
lazy val b = {
   m() + 1
}
```

6.1.3 The Postcondition

Sometimes it is required to return permission to the context in which the commit method is called. For example, if the lazy val gets initialized with a new object and one wants to transfer permissions to the fields of the object back to the caller of the commit method. This can be done by including them in the field invariant. However, as explained in 6.1 one cannot get write access out of the field invariant. Therefore one has to add write access to the postcondition of the code block. With the introduction of the postcondition, calling the commit method is equivalent to exhaling the precondition and inhaling the postcondition and inhaling read access to the lazy val.
An example for the use of postconditions:

```
class Cell() {
   ensures(write(f) && f == 2)
   var f: Int = 2
}

lazy val x = {
   ensures(write(result.f) && result.f == 2)
   fieldInvariant()

   new Cell()
}
commit(x)
main()

def main() {
   requires(read(x) && write(x.f) && x.f == 3)
   x.f = x.f + 1
   assert x.f == 3
}
```

6.2 Specification of Vals

The translation of lazy vals described in this chapter implements a feature that we want to have for vals too, namely field invariants. Therefore, we want to use the same approach we use

for lazy vals and adapt it for the translation of vals.
This means that we want to have a code block initializer for vals that works the same way as the initializer for lazy vals. A very important difference between vals and lazy vals is that the code block initializer of vals may contain side-effects and the preconditions can contain any form of permission, even write permissions. This is because at runtime a val is initialized immediately at declaration and we do not have the problem of not knowing when the initialization takes place. Note that it is still not possible to get write permission out of the field invariant because of the same reasons as there were for lazy vals.
The commit of a val happens automatically after the declaration of the val and therefore does not have to be specified by the user.
We can also remove the uncommitted flag and everything that has to do with it because the val is committed right at declaration and will therefore never be uncommitted. We want to present the same example we used for lazy vals in order to see what changes if one goes from a lazy val to a val.
The Scala code:

```
class Cell(val f: Int) {
   ensures(read(f))
}

val y = 3

~~lazy~~ val x = {
   requires(read(y) && y == 3)
   fieldInvariant(read(result.f) && result.f == 10)

   new Cell(calcMethod(y))
}
~~commit(x)~~

def calcMethod(param: Int) = {
   ensures(result == param + 7)
   param + 7
}

def main() {
   requires(read(x))
   assert x.f == 10
}
```

6.3 Implementation of Lazy Vals

The implementation for lazy vals needs to be changed in order to make the new features work. The affected parts are the commit method, the encoding of the field invariant and the use of lazy vals in the code.

6.3.1 Implementation of the commit method

We want to show the implementation of the commit method by the example given before in chapter 6.1.2.
The Scala code (parts that are irrelevant for the commit method translation are left out):

```
lazy val x = {
   requires(read(y) && y == 3)
```

```
   fieldInvariant(read(result.f) && result.f == 10)

   new Cell(calcMethod(y))
}
```

This gets translated into the following Silver code:

```
field x: Ref
field unc_x: Bool

method commit_x(this: Ref)
   requires acc(this.x, write)
   requires acc(this.unc_x, write)
   requires this.unc_x
   requires acc(this.y, wildcard) && this.y == 3
   ensures acc(inv_x(this), wildcard)
{
   this.x := // translation of new Cell(calcMethod(y))
   this.unc_x := false
   fold acc(inv_x(this), wildcard)
}
```

The Silver code for `new Cell(calcMethod(y))` is not stated here because it is not part of our thesis and it would just distract from the real problem.
In the first two lines of the Silver translation, we see the two fields that were already generated in the existing translation. The precondition of the commit method contains write access to both generated fields and the translated precondition of the initializer, which is in this case read access to val `y` and the assumption that `y == 3`. In the body, field `x` which represents the lazy val is assigned and the uncommitted flag `unc_x` is set to false.
The field invariant gets translated into an abstract predicate. This will be explained in the next subsection. For now it is only important to know that the abstract predicate `inv_x` represents the field invariant. An instance of this field invariant predicate is created at the end of the body and given to the caller of the commit method by putting it into the ensures clause.
The rest of the postcondition contains the translated postcondition of the lazy val initializer, which is empty in this case.

6.3.2 Implementation of the Field Invariant

As already mentioned in the previous section the field invariant is translated into an abstract predicate. The main idea is that this predicate can contain read access to the lazy val and the field invariant itself. The predicate can then be unfolded before using the lazy val.
For an example of the field invariant we want to reuse the example from above:

```
lazy val x = {
   requires(read(y) && y == 3)
   fieldInvariant(read(result.f) && result.f == 10)

   new Cell(calcMethod(y))
}
```

The Silver field invariant predicate looks like this:

```
predicate inv_x(this: Ref) {
   acc(this.x, wildcard) &&
   acc(this.x.f, wildcard) &&
   this.x.f == 10
```

```
}
```

The first line in the predicate ensures that we have read access to the lazy val. This information will be there even when the field invariant clause is empty. The other two lines represent the content we wrote into the field invariant.

6.3.3 Implementation of Lazy Val access

Since we want the information of the field invariant present wherever the lazy val is read, we have to provide the information inside the predicate every time the lazy val is accessed. This is done by translating read access to a lazy val in Scala into wildcard access to the according invariant predicate in the Silver translation. Consequently, wherever the lazy val is accessed, an instance of the field invariant is present.
In the Silver translation, accessing a lazy val is then surrounded with an unfolding expression that takes the according invariant predicate as parameter. This looks as follows:

```
def main() {
   requires(read(x))
   assert x.f == 10
}
```

This gets translated into the following Silver code:

```
method main(this: Ref)
   requires acc(inv_x(this), wildcard)
{
   assert (unfolding acc(inv_x(this), wildcard) in this.x.f) == 10
}
```

6.4 Implementation of Vals

We want to present the same example we used for lazy vals in order to see what changes if one goes from a lazy val to a val.
The Scala code (irrelevant parts are omitted):

```
lazy val x = {
   requires(read(y) && y == 3)
   fieldInvariant(read(result.f) && result.f == 10)

   new Cell(calcMethod(y))
}
commit(x)

def main() {
   requires(read(x))
   assert x.f == 10
}
```

The Silver translation:

```
field x: Ref
field unc_x: Bool

method commit_x(this: Ref)
   requires acc(this.x, write)
```

```
   requires acc(this.unc_x, write)
   requires unc_x
   requires acc(this.y, wildcard) && this.y == 3
   ensures acc(inv_x(this), wildcard)
{
   this.x := /* new Cell(calcMethod(y)) */
   this.unc_x := false
   fold acc(inv_x(this), wildcard)
}

predicate inv_x(this: x) {
   acc(this.x, wildcard) &&
   acc(this.x.f, wildcard) &&
   this.x.f == 10
}

method main(this: Ref)
   requires acc(inv_x(this), wildcard)
{
   assert (unfolding acc(inv_x(this), wildcard) in this.x.f) == 10
}
```

In this example we see that everything stays the same as in the lazy val translation except that anything concerning the uncommitted flag is gone.

6.5 Formal translation model

The translations shown so far were intended to give an intuition about how the translation works, but for an implementation of the translation one needs a formal model. For this we define a translation function that describes the translation of vals and lazy vals on a general level. This function was already used in [7]. Also the translation of other features like Scala classes, methods, functions, loops etc. is described[7] there.
For our purpose we change the function a bit:

$$translate : \mathcal{A} \rightarrow \mathcal{B}$$

Here $\mathcal{A}$ represents the set of valid Scala ASTs and $\mathcal{B}$ the set of all valid Silver ASTs. For starters a rather simple example, that illustrates how a Scala addition is recursively translated by translating each operand.

$$translate(a + b) = translate(a) + translate(b)$$

The following functions define the translation of Scala lazy vals into their Silver encoding. The implementation in Scala2Silver follows the same rules, although some details that do not belong into our thesis, like the rules for the type system defined in [7], or the real naming conventions to avoid name clashes, are left out here.
We want to start with the function that defines the members of the Silver code, i.e. the two fields, the commit method and the field invariant.
However we need to define a helper function first: `isLazyVal(ident)` is equal to 1 if `ident` is the identifier of a lazy val and 0 otherwise (then it is a val).

```
translate( "lazy val/val" ident ":" type "=" "{"
precs
posts
fInvs
```

```
stmts
exp
"}")

=
let b = if isLazyVal(ident) then 1 else 0

"field" ident ":" translate(type)

if b then "field unc_" ident ": Bool" else ""

"method commit_" ident "(this: Ref) {"
   "requires acc(this." ident ", write)"
   if b then "requires acc(this.unc_" ident ", write)" else ""
   if b then "requires this.unc_" ident else ""
   translate(precs)
   "ensures acc( inv_" ident "(this), wildcard)"
   translate(posts)
"{"
   translate(stmts)

   "this." ident ":=" translate(exp)
   if b then "this.unc_" ident ":= false" else ""
   "fold acc(inv_" ident "(this), wildcard)"
"}"

"predicate inv_" ident "(this: Ref) {"
   "acc(this. ident ", wildcard) &&" translate(fInvs)
"}"
```

The next function defines how read access to vals and lazy vals is translated into contracts: Function `isVal(ident)` is analogously defined for vals, as function `isLazyVal(ident)` is for lazy vals.

```
translate( "read(" rcvr "." ident ")")
=
if isLazyVal(ident) or isVal(ident) then
   "acc(inv_" ident "(" rcvr "), wildcard)"
else
   Existing translation as defined in [7]
```

The last function defines how the access of lazy vals gets translated.

```
translate(rcvr "." ident)
=
if isLazyVal(ident) OR isVal(ident) then
   "unfolding(inv_" ident "(" rcv "), wildcard) in rcvr "." ident
else
   Existing translation as defined in [7]
```

7 Singleton Object Translation

The translation of singleton objects has its own challenges. Our encoding has to model that a singleton object is initialized lazily i.e. on first use and that there can be only one given instance of a singleton object. Moreover, singleton objects have to be globally accessible.

7.1 Specification of Singleton Objects

The specification for singleton objects is relatively similar to the one of Scala classes. For the singleton object itself one has to specify each member inside the body and add contracts for the constructor.

```
object Foo {
   ensures(read(x))
   ensures(uncommitted(y))
   ensures(write(z))

   val x = // initializer of x
   lazy val y = // initializer of y
   var z = // initializer of z
}
```

7.2 Use of Singleton Objects

Since the singleton object instance is created on first use, we want to use the same approach as for lazy vals. Therefore, one needs to specify a commit point by calling `commit(*)` with the singleton object as parameter. At this point the initialization of the singleton object is modeled, which forces every access location that is involved in the precondition of the constructor to remain constant, like it was for lazy vals in the precondition of the initializer.
An `uncommitted(*)` assertion for every singleton object is added to the program state at the beginning of the program. The `uncommitted(*)` assertion needs to be carried around in contracts for every singleton object until it is committed.
After the commit, read access to the singleton object will be added to the program state, along with write permissions and assumptions for every var in the singleton object. In addition to that, `uncommitted(*)` assertions for every lazy val that has not yet been committed in the constructor of the singleton object will be added to the program state. Read access and assumptions to vals and already committed lazy vals will be contained in the field invariant of the singleton object.

```
// we have uncommitted(Foo)
commit(Foo)
// we have read(Foo), read(Foo.x), uncommitted(Foo.y) and write(Foo.z)
```

7.3 Translation of Singleton Objects into Lazy Vals

We want to say in advance that this translation is purely conceptual and Scala2Silver will do the translation directly from the singleton object into Silver code by using the insights gained from this section.
The main idea behind our translation approach is that we want to reuse the lazy initialization model we introduced for lazy vals. This is done by translating the singleton object into a class and a lazy val. The class defines the layout of the singleton object. The lazy val is assigned to an instance of the created class and represents the single instance of the singleton

object. However, this means that since singleton objects use the same approach to deal with lazy initialization as lazy vals, the same restrictions that hold for lazy val initializers hold for singleton object constructors. Consequently, a small amount of permission to every access location in the precondition gets discarded, to ensure that no side-effects occur and that the expressions of the member initializers stay constant. Note that the members of the singleton objects may be written or rewritten in the constructor, because these are not side-effects that are visible from outside of the constructor.
We want to pick up the example of section 7.1 again. The translation looks as follows:

```
object Foo {
   ensures(read(x))
   ensures(uncommitted(y))
   ensures(write(z))

   val x = // initializer of x
   lazy val y = // initializer of y
   var z = // initializer of z
}
```

this singleton object gets translated into the following class and lazy val

```
class Foo_class {
   ensures(read(x))
   ensures(uncommitted(y))
   ensures(write(z))

   val x = // initializer of x
   lazy val y = // initializer of y
   var z = // initializer of z
}

lazy val Foo {
   ensures(uncommitted(y))
   ensures(write(z))
   fieldInvariant(read(x))

   new Foo_class()
}
```

With this example the transfer of permissions described in section 7.2 makes much more sense. The `uncommitted(*)` flag is present from the beginning of the program because a singleton object is globally accessible and therefore already present at the program start. Committing the singleton object makes sense because knowing that it uses the lazy val approach.
Permissions to the fields of a singleton object are partially transferred to the caller of the commit method and partially to the field invariant. As stated in the section above we can put access to vals and already committed lazy vals into the field invariant, but this is not possible for write access, which means that permissions to vars and uncommitted assertions of lazy vals need to be put into the postcondition of the initializer.

7.4 Concrete Example

This section demonstrates the translation described above with a substantial example. Note that after the commit of `Foo`, `Bar.f` becomes immutable. If this would not be the case and one could change the value of `Bar.f` after `commit(Foo)`, the translation would be unsound.

```
object Foo {
   ensures(read(x))
   ensures(uncommitted(y))
   ensures(write(z) && z == 2)

   val x = {
      fieldInvariant(result == 5)
      5
   }

   var z = 2

   lazy val y = {
         requires(read(z) && z == 4)
      fieldInvariant(result == 8)
      z * 2
   }
}

def main() {
   commit(Foo)
   assert(Foo.x == 5)
   Foo.z = 4
   commit(Foo.y)
   assert(Foo.y == 8)
}
```

The translated version looks as follows:

```
class Foo_class {
   ensures(read(x))
   ensures(uncommitted(y))
   ensures(write(z) && z == 2)

   val x = {
      fieldInvariant(result == 5)
          5
   }

   var z = 2

   lazy val y = {
         requires(read(z) && z == 4)
      fieldInvariant(result == 8)
      z * 2
   }
}

lazy val Foo = {
   ensures(uncommitted(result.y))
   ensures(write(result.z) && result.z == 2)
   fieldInvariant(read(result.x))

   new Foo_class()
```

```
}

def main() {
   commit(Foo)
   assert(Foo.x == 5)
   Foo.z = 4
   commit(Foo.y)
   assert(Foo.y == 8)
}
```

Since the real goal of this thesis is to translate singleton objects into Silver code we want to give a Silver translation of the Scala code above.
However, there is still one challenge remaining that is not yet solved: How do we represent global accessibility? The solution to this problem in Scala is putting the lazy vals that represent singleton objects into a package object [15]. In Silver we create a helper object in the beginning that contains all fields that correspond to singleton objects. This object is passed to every Silver method as parameter.
The following code example shows the Silver translation of the Scala code above. Pieces of the code that are not part of our thesis are omitted.

```
field x: Int
field Foo: Ref
field unc_Foo: Bool
field y: Int
field unc_y: Bool
field z: Int

predicate inv_y(this: Ref) {
   acc(this.y, wildcard) && (this.y == 8)
}

predicate inv_Foo(this: Ref) {
   acc(this.Foo, wildcard) && (this.Foo != null) && acc(inv_x(this.Foo),
      wildcard)
}

predicate inv_x(this: Ref) {
   acc(this.x, wildcard) && (this.x == 5)
}

method Foo_class_constructor(rd: Perm, this: Ref)
   requires this != null
   requires acc(this.x, write)
   requires acc(this.z, write)
   requires acc(this.unc_y, write) && this.unc_y && acc(this.y, write)
   ensures acc(inv_x(this), wildcard)
   ensures acc(this.unc_y, write) && this.unc_y && acc(this.y, write)
   ensures acc(this.z, write) && (this.z == 2)
{
   commit_x(this)
   this.z := 2
}

method commit_x(this: Ref)
```

```
   requires this != null
   requires acc(this.x, write)
   ensures acc(inv_x(this), wildcard)
{
   this.x := 5
   fold acc(inv_x(this), wildcard)
}

method commit_y(this: Ref)
   requires this != null
   requires acc(this.unc_y, write)
   requires this.unc_y
   requires acc(this.y, write)
   requires acc(this.z, wildcard) && (this.z == 4)
   ensures acc(inv_y(this), wildcard)
{
   this.y := this.z * 2
   this.unc_y := false
   fold acc(inv_y(this), wildcard)
}

method commit_Foo(this: Ref)
   requires this != null
   requires acc(this.unc_Foo, write)
   requires this.unc_Foo
   requires acc(this.Foo, write)
   ensures acc(inv_Foo(this), wildcard)
   ensures acc((unfolding acc(inv_Foo(this), wildcard) in this.Foo).unc_y,
       write)
   ensures (unfolding acc(inv_Foo(this), wildcard) in this.Foo).unc_y
   ensures acc((unfolding acc(inv_Foo(this), wildcard) in this.Foo).y,
       write)
   ensures acc((unfolding acc(inv_Foo(this), wildcard) in this.Foo).z,
       write)
   ensures (unfolding acc(inv_Foo(this), wildcard) in this.Foo.z) == 2
{
   this.Foo := // translation of new Foo_class()
   this.unc_Foo := false
   fold acc(inv_Foo(this), wildcard)
}

method main()
{
   var SOs: Ref // Global object for singleton objects will be passed to
       every method that needs them
   SOs := new(Foo, unc_Foo)
   SOs.unc_Foo := true
   commit_Foo(SOs)
   assert (unfolding acc(inv_Foo(SOs), wildcard) in (unfolding acc(inv_x(
       SOs.Foo), wildcard) in

       SOs.Foo.x)) == 5
   (unfolding acc(inv_Foo(SOs), wildcard) in SOs.Foo).z := 4
```

```
   commit_y((unfolding acc(inv_Foo(SOs), wildcard) in SOs.Foo))
   assert (unfolding acc(inv_Foo(SOs), wildcard) in (unfolding acc(inv_y(
       SOs.Foo), wildcard) in  SOs.Foo.y)) == 8
}
```

The first couple of lines of this code example declare all required fields.
The fields are followed by the field invariant predicates for val `x`, lazy val `y` and the lazy val that represents singleton object `Foo`. They are translated as described in chapter 6.3. Remember that read access is represented in Silver code as wildcard access to an instance of the field invariant predicate. Therefore, we have `acc(inv_x(this.Foo), wildcard)` instead of `acc(this.Foo.x, wildcard)` inside predicate `inv_Foo`.
Below the predicates come the commit methods of `x`, `y` and `Foo`. They are also translated as described in chapter 6.3. There should be no surprises here.
The last function in the example above is the `main` function that uses singleton object `foo`. `main` first creates a new container for singleton object foo by creating a new object that contains `Foo` and `unc_Foo`. Then `unc_Foo` is initialized to `true` in order to initialize the uncommitted state. The following code is directly translated from the Scala code as described in chapter 6.3. The only difference here is that unfolding of depth two is required to gain all information required for the assertions.

7.5 Formal translation

As already seen in the lazy vals chapter, we want to give a formal translation function for the translation of singleton objects from Scala to Scala code. In this case, we cannot use the function we defined before because the image of the function is now Scala code instead of Silver code. Therefore, we use the adapted function `translateScala` which is applied to every singleton object before the `translate` function does its work.

$$translateScala : \mathcal{A} \rightarrow \mathcal{A}$$

$\mathcal{A}$ is defined as in 6.5. The translation function for a singleton object looks as follows:
`PRE` is the precondition of the constructor of the singleton object, `POST` is the postcondition, \ denotes the set subtraction and `WRITE` denotes the set of write accesses to access locations in form of write access to vals and uncommitted assertions in form of uncommitted lazy vals.

```
translateScala( "object" ident "{"
   mems
"}")

=

"class" ident "_class {"
   mems
"}"

"lazy val" ident "= {"
   reqires(PRE)
   ensures(WRITE)
   fieldInvariant(POST \ WRITE)

   "new " ident "_class()"
"}"
```

8 Conclusion

8.1 Current Limitations

Although this thesis has pushed the limitations of lazy vals further, there are still limitations remaining. One limitation that remains for lazy val translation is that the initializers of lazy vals may not contain side-effects, except object creation. It is stated in chapter 4 that this limitation exists to ensure that modeling the initialization of the lazy val at the point where it is committed has the same effects as modeling the initialization on the real first use.
The other limitation for the programmer is that the result of evaluating the right-hand side of the lazy val may not change anymore after the lazy val has been committed. The reason for this is the same as for the limitation above. If one allows the value of the lazy val to change after the commit point the verification gets unsound.

8.2 Conclusions

This thesis presents a basis for future design decisions in the verification of lazy vals in form of an empirical study on lazy vals described in chapter 5. Further, meaningful test-cases based on use-cases found during the empirical study were added to the Scala2Silver test suite and can be seen in chapter 9.
Then a solution for overcoming the limitations about the right-hand side as described in 4.2.4 is presented in chapter 6 and this solution was implemented for the Scala2Silver project.
Furthermore, this thesis takes the approach for lazy vals and applies it to vals. Therefore, the encoding of vals and lazy vals is unified and both vals and lazy vals benefit from the same features.
As the last point, the theoretical work on the translation of Singleton objects is finished and presented in chapter 7.

8.3 Future Work

There are several points of future work left after our thesis finishes:

- Implementation of the singleton object translation described in chapter 7. This thesis only described the theory, an implementation in Scala2Silver is still missing.
- Implementation of local vals and lazy vals in Scala2Silver. This is mainly an implementational task as the theory for locals is essentially the same as for fields.
- Solving the initialization problem shown in chapter 4.1.1.
- Generate contracts for the code block initializers of vals and lazy vals automatically.

9 Test Cases

One part of our thesis was to develop use case oriented test cases and we came up with three cases that cover the most important functionality. Note that each use case had to be adapted a bit because Silver does not support built-in Scala features. Therefore, every built-in type is represented with a simple class or a primitive.
The first testcase belongs to the category field.

```
/* Example taken from Lift; DB.scala, Line 1023

class SuperConnection(val connection: Connection, val releaseFunc: () =>
    Unit, val schemaName: Box[String]) {
   def this(c: Connection, rf: () => Unit) = this (c, rf, Empty)

   lazy val brokenLimit_? = driverType.brokenLimit_?

   def createTablePostpend: String = driverType.createTablePostpend

   def supportsForeignKeys_? : Boolean = driverType.supportsForeignKeys_?

   lazy val driverType: DriverType = DriverType.calcDriver(connection)

   lazy val metaData = connection.getMetaData
}

The interesting part of this use case is the assignment of a class field
    passed as parameter to a lazy val seen in the assignment of lazy val
    metaData.
We want to be able to do this too, therefore I prepared a similar testcase
    below ( we introduced the var because constructor params are not yet
    fully working)
*/
import viper.contracts.Contracts._

class SuperConnection(con: Connection) {
   ensures(read(metaData))
   ensures(uncommitted(metaData))

   var connection: Connection = con

   lazy val metaData = {
      requires(read(connection) && read(connection.getMetaData) &&
          connection.getMetaData == 7)
      fieldInvariant(result[Int] == 7)
      connection.getMetaData
   }
   commit(metaData)

   def test() {
      val connection = new Connection()
      val lazyclass = new SuperConnection(connection)
      assert(lazyclass.metaData == 7)
   }
}
```

```
class Connection {
   ensures(read(getMetaData))

   val getMetaData = {
      fieldInvariant(result[Int] == 7)
      7
   }
}
```

The second example involves method chaining and is part of the category MethodCall/NoSideEffect

```
/* Example taken from Sbt; Escapes.scala, Line 81

lazy val genWithoutTerminator = genRawString.map( _.filter { c => !
    isEscapeTerminator(c) } )

Basically a lot of methods called on a string object. We will model this by
     applying some modifying functions onto an immutable object. We will for
     now ignore the closure in the end because closures are out of scope
*/
import viper.contracts.Contracts._

class CallChaining {
   ensures(read(lval))

   lazy val lval = {
      fieldInvariant(read(result[Modifiable].value) && result[Modifiable].
          value == 7)
      new Modifiable().m1().m2().m3(3)
   }
   commit(lval)

   def test() {
      val lazyclass = new CallChaining()
      assert(lazyclass.lval.value == 7)
   }
}

class Modifiable {
   ensures(write(value))
   ensures(value == 0)

   var value = 0

   def m1() = {
      requires(read(value))
      ensures(read(value))
      ensures(read(result[Modifiable].value))
      ensures(result[Modifiable].value == 8)

      val temp = new Modifiable()
      temp.value = 8
```

```
      temp
   }

   def m2() = {
      requires(read(value))
      ensures(read(value))
      ensures(read(result[Modifiable].value))
      ensures(result[Modifiable].value == value / 2)

      val temp = new Modifiable()
      temp.value = value / 2
      temp
   }

   def m3(num: Int) = {
      requires(read(value))
      ensures(read(value))
      ensures(read(result[Modifiable].value))
      ensures(result[Modifiable].value == value + num)

      val temp = new Modifiable()
      temp.value = value + num
      temp
   }
}
```

The third example is about an arbitrary code block.

```
/* Example taken from Akka; ActorSelection.scala, Line 128

override lazy val hashCode: Int = {
   import MurmurHash._
   var h = startHash(anchor.##)
   h = extendHash(h, path.##, startMagicA, startMagicB)
   finalizeHash(h)
}

Need to steamline this example a bit.
Essentialy the use a Code Block initializer with three method calls inside
    the block is important for us here.
*/

import viper.contracts.Contracts._
import viper.contracts.annotations._

class TestClass {
   ensures(read(hash))

   val anchor = {
      fieldInvariant(result[Int] == 5)
      5
   }

   val hashExtender = {
```

```
      fieldInvariant(result[Int] == 7)
      7
   }

   lazy val hash = {
      requires(read(anchor) && read(hashExtender))
      fieldInvariant(result[Int] == 27)
      var h = startHash(anchor)
      h = extendHash(h , hashExtender)
      finalizeHash(h)
   }
   commit(hash)

   def startHash(param: Int) = {
      ensures(result[Int] == param*2)
      param*2
   }

   def extendHash(param1: Int, param2: Int) = {
      ensures(result[Int] == param1 + param2)
      param1 + param2
   }

   def finalizeHash(param: Int) = {
      ensures(result[Int] == param + 10)
      param + 10
   }

   def test() {
      val testInstance = new TestClass()
      assert(testInstance.hash == 27)
   }
}
```

References

[1] Akka. Homepage: http://akka.io.

[2] Lift. Homepage: http://liftweb.net/.

[3] Sbt. Homepage: http://www.scala-sbt.org/.

[4] M. Barnett, B. yuh Evan Chang, R. Deline, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science, pages 364–387. Springer, 2006.

[5] M. Ben-Ari. The bug that destroyed a rocket. In Journal of Computer Science Education, pages 15–16, 1999. Online: http://www.cs.jhu.edu/~jorgev/cs106/bug.pdf.

[6] J. Boyland. Checking interference with fractional permissions. In Proceedings of the 10th international conference on Static analysis, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] B. Brodowsky. Translating scala to sil. Master's thesis, ETH Zürich, June 2013.

[8] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver, volume 4963/2008 of Lecture Notes in Computer Science, chapter 24, pages 337–340. Springer Berlin, Berlin, Heidelberg, April 2008.

[9] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Fractional permissions without the fractions. In Formal Techniques for Java-like Programs (FTfJP), 2011.

[10] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. 2014.

[11] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with chalice. In Foundations of Security Analysis and Design V, volume 5705 of LNCS, pages 195–222. Springer, 2009.

[12] A. Lum. Patriot missile software problem. Online Article: http://sydney.edu.au/engineering/it/~alum/patriot_bug.html.

[13] R. Matulis. Extensible code contracts for scala. Master's thesis, ETH Zürich, September 2012.

[14] Microsoft. Code contracts for .net. Homepage: http://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970.

[15] M. Odersky and L. Spoon. Package objects. Scala Documentation: http://www.scala-lang.org/docu/files/packageobjects/packageobjects.html.

[16] M. Odersky, L. Spoon, and B. Venners. Programming in Scala. Artima Inc., 2007.

[17] M. Parkinson and G. Bierman. Separation logic and abstraction. In In POPL, pages 247–258, 2005.

[18] J. Reynolds. Separation logic: A logic for shared mutable data structures. page 55–74. IEEE Computer Society, 2002.

[19] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, pages 148–172, Berlin, Heidelberg, 2009. Springer-Verlag.