ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Closure verification in an automated fractional permission setting

Bachelor Thesis

F. Meier

Oktober 7, 2014

Supervisor: Dr. Yannis Kassios

Department of Software Engineering, ETH Zürich

Abstract

Closures are values which contain executable code - code which is generally not known statically. This dynamic nature of closures makes it hard to verify them statically without a strong specification language and a strong program verifier. A program verifier is a tool that allows developers to prove statically that their code satisfies its specifications in every environment.
In this Thesis I discuss a methodology on how to add closures to Chalice and how the Viper verifier has to be adapted to verify closures. The developed concept supports closures as first-class-citizen values and allows users to add specifications to closures. It allows specifications of state that is hidden from the client of a closure and an explanation is found why the system does not suffer from the well known circularity issues regarding self-referencing method specifications. I also develop a concept for the use of higher order functions in Chalice.
The Viper project was enriched with the features for closures. There are limitations in the implementation regarding old expressions (for specification proofs and higher order functions) and while dealing with hidden state. Examples for all the features of closures could be verified.

Contents

Chapter 1

Introduction

Closures are values containing executable code. Many of today's mainstream programming languages support closures, but few of them feature verification of these closures. This is mainly because of their dynamic nature and the difficulty to specify them using first order logic.

Goal of this project is to find a sound methodology to add closures to Chalice [1] and verify them with the already existing Viper verifier called Silicon. The developed concept has to find solutions for several features and well known issues of closures:

Closures as values Closures must first-class-citizen values, for example they have to be supported as arguments or as return values of methods.

Specification for Closures Method contracts may refer to the specification of other methods, for example a closure factory must describe the behaviour of the delegate it creates. In particular statements like `pre(c,·)` and `post(c,·)` for a closure `c` must be supported.

State Capturing Closures can hide state which makes it difficult to verify them and auxiliary access methods are needed to deal with hidden state.

Circularity Many closure verification methodologies have the *Recursion through the store* phenomenon, when a closure has statically unknown references to itself in its specification. This may lead to inconsistencies, which need to be avoided in the developed concept.

Higher Order Functions Higher order functions are functions or methods that work on delegates without knowing their exact specifications. They are heavily used in design patterns involving delegates such as Visitor or Chain of Responsibility.

1.1 The Viper Project

The Viper Project is one of the projects of ETH's department of Software Engineering concerned with software verification. It started development in 2010 and was then known under the name Semper. In 2014 the Semper project was renamed to Viper and made open-source [3]. The project has grown quite large, but only four components are relevant for this thesis, as shown by Figure 1.1.

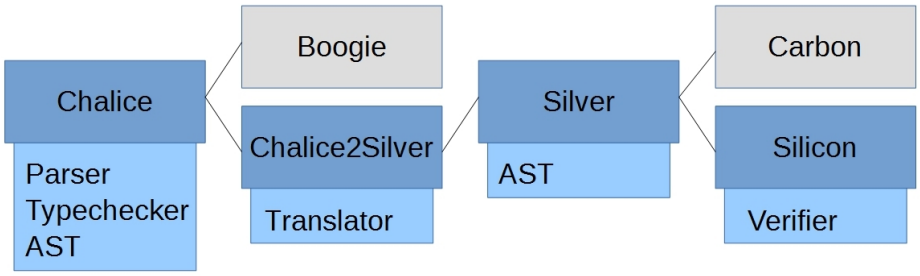

Figure 1.1: The components from Chalice to Silicon

Chalice is the front-end research language developed by Microsoft. It features Parsing, has its own AST and Type-checking.

Boogie is the original verifier infrastructure for Chalice developed by Microsoft. It is not relevant for this thesis.

Chalice2Silver is the tool that translates the Chalice AST into Silver AST.

Silver is the intermediate language of the Viper infrastructure. Front-end languages like Chalice are translated into Silver before they are verified.

Silicon and Carbon are the back-end verifiers of the viper infrastructure. Carbon is based on verification condition generation and Silicon on Symbolic Execution. This thesis uses Silicon for the verification of closures.

Summary of the Thesis

Section 2 provides an introduction to the programming language Chalice. In section 3 I present the methodology how closures can be added to Chalice and how various issues are resolved. In section 4 the rest of the used Viper infrastructure is presented and section 5 gives insight into the back-end of the Viper infrastructure by showing how to verify the added closures. Section 6 shows the final result of the implementation and gives some exam-

ples on how it can be used. Further features are presented in section 7 and section 8 concludes the thesis.

Chapter 2

Chalice

This chapter is an introduction to the programming language Chalice. In section 2.2 the basic language concepts will be explained. If you're already familiar with object oriented languages you're advised to sikp this section. The verification process will be shown in section 2.3, by giving an introduction into symbolic verification in section 2.3.1 and then introducing the concept of fractional permissions in section 2.3.2.

2.1 Introduction to Chalice

Chalice is a research language developed by Microsoft. One of the core concepts of Chalice is the use of fractional permissions - a concept that simplifies verifying concurrent programs. More details about fractional permissions can be found in section 2.3.
Chalice was originally verified by Microsoft's Boogie platform. Recently ETH developed their own verification infrastructure around Chalice with the verifier Silicon.

2.2 Concepts of Chalice

Chalice is an object oriented language without inheritance. It is statically typed and can therefore be type-checked. Chalices top-level definitions are classes and channels. Channels were used to pass information between different threads but they are no longer supported in the Viper infrastructure. Classes are still supported and they can have members of type

- Field,
- Method,
- Function,

- Invariant and
- Predicate.

The following subsections introduce each type and give some examples.

Note *In this chapter we build up a running example that is then used in all the following chapters.*

2.2.1 Members

Fields

Fields are just as one knows them from other languages. They have a type and a name. The two relevant basis types are `int` and `bool`. Fields can also have user defined classes as types. Let's start the example with a class `Cell` that can hold an integer value.

```
class Cell {
    var value: int
}
```

Methods

Methods can have multiple arguments and multiple return values. A method `foo` can be added to the class `Cell` with the following syntax:

```
method foo(i: int, c: Cell) returns (j: int, d: Cell)
{
    j := i + c.value
    d := c
}
```

Note *that return values are assigned like normal variable assignments while arguments can not be assigned to.*

Functions

Functions are pure methods that only have one return value. The body of a function consists of an expression. Class `Cell` can for example be extended with function `getLinear`:

```
function getLinear(a: int, b: int): int
{
    a * this.value + b
}
```

Invariants and Predicates

Invariants and predicates are explained in section 2.3.

2.2.2 Instructions

Chalice supports the following instruction set:

Object creation

There are no special constructor methods in Chalice. To create a new object of type `Cell` one can write

```
var myCell := new Cell
```

This creates a new object of type `Cell` and stores a reference to it in the variable `myCell`. All fields will be initialized with the the default values of their respective type.

Method Call

Methods can be called with the call syntax:

```
var d: Cell
call myJ, d := myCell.foo(7, myCell)
```

The return values of `foo` will end up in the variables `myJ` and `d`. The call statement also declares undeclared variables (for example `myJ`).

Fork and Join

Concurrent execution of methods is achieved with `fork` and `join` instructions. The `fork` statement returns a token which can be used in a later `join` instruction to join that particular concurrent invocation:

```
fork t := myCell.foo(7, myCell)

// do other work

join myJ, d := t
```

The type of `t` is `token<Cell.foo>` but this token can not be given as argument to another method.

Function Call

Functions can be invoked more freely, for example in expressions:

```
myJ := d.getLinear(3, 8 * myCell.getLinear(1, 2))
```

Contol Flow

There are also two control-flow modification statements: `if` and `while`:

```
if (myJ == 8)
{
    while (myJ < 100)
    {
        myJ := myJ + 1
    }
}
else
{
    myJ := 100
}
```

2.3 Verifying a Chalice program

This section describes the process how a Chalice program is verified by Silicon, a symbolic execution based verifier. The next section gives a brief introduction to symbolic verification and section 2.3.2 establishes the concept of fractional permissions and how Silicon handles them.

2.3.1 Symbolic Verification

Symbolic execution is a technique where all variables get assigned symbolic values and then the program is verified by stepping through the instructions one by one. During this the verifier maintains a state that contains the current symbolic values and constraints of these variables. This state can be checked and modified by the very central operations assume and assert. Assume adds conditions to the state while assert checks whether the state satisfies a condition. This can be illustrated by an example:

```
if (a == 3) {
    // body
}
```

The body of this `if` statement can be verified with the additional assumption that `a == 3`.

```
b := 6 / a
```

To prove that this assignment is valid, the verifier has to assert that `a != 0`. Symbolic verifiers go through each method of a program and verify them. A way of achieving scalability for large programs is to add contracts in order to increase modularity of the program. These contracts are added in form of method pre- and postconditions and class invariants. With these each

method can be verified against its contracts - it has to satisfy its own specification. This also helps to reduce the complexity of verifying a method call - the caller can simply assume that the called method satisfies its specifications.
In Chalice method specifications are added with the following syntax.

```
method foo(i: int, c: Cell) returns (j: int, d: Cell)
    requires i >= 10 && c != null && c.value > 0
    ensures j > i && c == d
{
    ...
}
```

Note *Pre- and post conditions can be split up over multiple lines by repeating the requires or ensures keyword.*

Postconditions can also contain `old` expressions. This is useful to specify the change of a field. For example a method that increases the value of a cell can specify it by having postcondition `c.value == old(c.value) + 1`.

Note *From the perspective of the verifier a method call is nothing more than asserting its precondition then assuming its postcondition. A method itself can be verified by assuming the pre condition then verifying the body and in the end trying to assert the post condition.*

In Chalice functions do only have preconditions - postconditions are implicitly inferred from the body of the function.

```
function getLinear(a: int, b: int): int
    requires a > 0
{
    a * this.value + b
}
```

Note *Since loops are hard to verify with symbolic execution the user can help by specifying a loop invariant.*

```
    while (myJ < 100)
        invariant myJ <= 100
    {
        myJ := myJ + 1
    }
```

The invariant has to hold at the start of the while and after each iteration - this includes the last one.

2.3.2 Access permissions

Fraction Permissions are a way to help verifying concurrent programs. The hardest part of verifying concurrent programs is that at any point in time an-

other thread could modify a field you currently work on. Chalice prevents that by introducing access permissions - the permission to read or write a field.
These permissions are kept in the state in a separate heap. Whenever there is a field-access the verifier checks whether the appropriate permissions are present. Permissions of the state can be added and removed by the operations inhale and exhale. They work very similar to assume and assert, they even include the behaviour of them because permissions are usually added simultaneously to other conditions, for example in pre- and postconditions.

inhale is the operation that adds the permissions it is given to the state. If it receives a normal expression it will *assume* the expression.

exhale removes permissions from the state. If normal expressions are exhaled it will *assert* them.

Note *An expression containing no access permissions is called pure. For pure expressions* `inhale` *is equivalent to assume,* `exhale` *to assert.*

Let's illustrate this behaviour with an example:
The following method `bar` cannot be verified because it's missing the access permissions to `c.value`:

```
method bar(c: Cell) returns (i: int)
    requires c != null
    ensures i == old(c.value) && c.value == 0
{
    i := c.value
    c.value := 0
}
```

To attain read and write access for `c.value` the method `bar` must require the access permission:

```
method bar(c: Cell) returns (i: int)
    requires c != null && acc(c.value)
    ensures i == old(c.value) && c.value == 0
{
    i := c.value
    c.value := 0
}
```

Access permissions can also be split. This is useful to allow concurrent access to the same resources. Split access is always read only - to write a field 100% of the access is required. This results in a thread safe environment because there can never be more than 100% of access to a field.
For example 50% read access is denoted by `acc(c.value, 50)`. Any percentage can be given - there is also the possibility to give infinitely small amounts or arbitrary amounts of access to another method.

Note *The current version of `bar` would consume the access to `c.value` because it does not return the access. This means, after it is called no-one can access `c.value` any more. To fix this `bar` should have a postcondition `ensures acc(c.value)` to give access back to the caller.*

Note *Accessing fields in pre- or postconditions needs at least read permission. An expression that covers all its field-accesses with permissions is called self-framing. Only self-framing pre- and postconditions are allowed in Chalice.*

Note *Accessing fields of the `this` object needs permission too, like any other access.*

Note *Functions do not have post conditions but can still require access permissions. This access is not consumed, instead functions always return all access permissions which they require.*

Bundling Permissions

Specifying permissions can lead to a huge overhead very fast - often the same fields are required for each operation on an object. This is where predicates come into play - they bundle access permissions together.

```
predicate valid { acc(this.value) && acc(this.value2) }
```

Predicates are members of classes and they usually hold access rights to the classes' fields. Predicates can or must be folded and unfolded. Folding transforms the individual permissions into the predicate and unfolding reverses this. These operations only succeed if the necessary permissions are present.

```
fold this.valid
unfold this.valid
```

To access fields which are enclosed by a predicate, the predicate has to be unfolded first. Predicates can also be temporarily unfolded in expressions. This is useful to unfold a predicate in a function which can only have an expression as body:

```
unfolding this.valid in this.value
```

Note *Predicates can also hold other expressions than accesses. This is especially useful for recursive data structures where a `valid` predicate of an outer element can contain the access to the inner element and simultaneously the `valid` predicate of the inner element.*

Predicates and functions allow classes to hide their inner workings. For example we can add the following members to the `Cell` class to hide that it contains a `value` field:

```
class Cell {
    var value: int

    function getValue(): int
        requires valid
    { unfolding valid in value }

    predicate valid { acc(value) }

    method set(i: int)
        requires valid
        ensures valid && getValue() == i
    {
        unfold valid
        value := i
        fold valid
    }

    method inc()
        requires valid
        ensures valid && getValue() == old(getValue()) + 1
    {
        unfold valid
        value := value + 1
        fold valid
    }
}
```

A client of the `Cell` can now verify the following method without knowing that `Cell` contains an integer:

```
method test(c: Cell)
    requires c.valid && c.getValue() == 0
    ensures c.valid && c.getValue() == 10
{
    c.set(4)
    c.set(c.getValue() * 2)
    c.inc()
    c.inc()
}
```

`Cell` could also internally maintain a history of values and use only the newest entry and the client would not notice the difference.

Chapter 3

Adding Closures to Chalice

In this chapter I present a concept of how closures can be added to Chalice and then be verified with the existing Viper verifier. Section 3.1 gives a short introduction of the type of closures I add to Chalice. Then in section 3.2 and 3.3 the syntax of the closures is explained and in section 3.4 I show how the added closures can be specified in order to verify them. Several issues of the concept are then discussed in section 3.5 (Hidden State), section 3.6 (Circularity) and section 3.7 (Higher Order Functions).

3.1 Overview of the Design

Closures are variables that can hold executable code which can change dynamically. Here is an example that shows a closure in action:

```
method closures() {
    var a := 0

    var c := {
        a := a + 1
        print a
    }
    c()
    c()

    c := {
        a := a - 1
        print a
    }
    c()
}
```

Note *A closure named* `c` *is created and invoked twice. Then the executable code of* `c` *is changed and* `c` *is invoked once more. The final output of this program is* `1 2 1`.

This example shows an anonymous closure. It's anonymous because the code `a:= a + 1; print a` is not part of any named method.
Another approach to closures is to only allow pointers to named methods. This approach is well known from mainstream languages like C#. There closures are the first-class-citizen objects called delegates that point to methods of other objects. This is not a weaker concept than allowing anonymous closures, because every anonymous closure can be transformed into a delegate with the same behaviour.
The following sections describe in detail how delegates were added to Chalice.

3.2 Delegate Type definition

The first modification of Chalice is the introduction of a new top-level declaration: the delegate type definition. Delegate types are what make delegates type-safe by specifying the signature of the method a delegate can point to (the so-called delegate target method).

```
delegate Creator(int) returns (Cell)
```

Note *There can be as many arguments and return types as you like, but the signature has to match exactly, even the order of the arguments.*

This means delegate objects of type Creator could point to the following two methods, but not to the third:

```
method createCell(i: int) returns (c: Cell)
    ensures c.valid && c.getValue() == i
{
    c := new Cell
    c.value := i
    fold c.valid
}
method createIncreasedCell(i: int) returns (c: Cell)
    ensures c.valid && c.getValue() == i + 1
{
    c := new Cell
    c.value := i + 1
    fold c.valid
}
method createDiffCell(i: int, j: int) returns (c: Cell)
    ensures c.valid && c.getValue() == i - j
{
    c := new Cell
    c.value := i - j
    fold c.valid
}
```

3.3 Delegate Object

Instances of a delegate type can appear as fields of other classes, as arguments to methods and as local variables. The delegate object points to a method of the so called delegate target object or delegate target for short. Both the delegate target and the target method are immutable.

3.3.1 Delegate creation

Delegate objects are constructed very similarly to normal objects. Instead of the `new` keyword the `newdel` keyword is used.

```
var cre := newdel Creator(this.createCell)
```

Reminder *During this operation the delegate target method and delegate target are assigned and after this they are immutable.*

3.3.2 Delegate call

Similar to a Chalice method call, delegates can also be called with the `call` keyword.

```
call myCell := cre(7)
```

Note *Delegates can be called multiple times.*

3.3.3 Delegate fork and join

Delegates can be forked and joined to allow concurrent execution of delegates. Similar to method forking the delegate fork produces a token of type `token<Creator>`. This token can be used to join the delegate execution with the current execution.

```
fork t := cre(7)

// do other work

join myCell := t
```

Note *Similar to a method fork, a delegate can be forked multiple times before it's joined again (if you can provide the necessary permissions for each fork).*

3.4 Specifying delegates

These were already all of the added instructions, but now we need some mechanics on how to verify delegates. Let us look at an example of the issue first:

```
method client(cre: Creator) {
    call myCell := cre(7)

    assert myCell.getValue() == 8
}
```

In this example the `client` has no information which method is executed when it calls `cre`. It could be createCell or createIncreasedCell or something completely different. We need to be able to specify what `cre` does and then adding that specification to the precondition of `client`. This specification should then reside in the state of the symbolic execution until it is needed when the delegate is called.
There are two different ways to achieve this. Both will be explained using the `creatorFactory` example below.

```
method creatorFactory() returns (cre: Creator)
    ensures ?
{
    if (random()) {
        cre := newdel Creator(this.createCell)
    } else {
        cre := newdel Creator(this.createIncreasedCell)
    }
}
```

Exact Specification The First possibility is to specify the exact target method of `cre`. This allows us to write the postcondition of `creatorFactory` as something similar to:

```
(cre points to createCell) or
(cre points to createIncreasedCell)
```

Stronger Preconditions and Weaker Postconditions The other approach is to allow specifications to be more ambiguous. This means not always the exact specification is given but rather something different. For the system to be sound the specified constraints have to be stronger than the actual precondition (in case of a precondition) or otherwise weaker than the actual postcondition. This design allows us to specify the postcondition of `creatorFactory` as

```
cre(value) returning (myCell) has postcondition
    myCell.valid and value <= myCell.getValue() and
    myCell.getValue() <= value + 1
```

Note *This postcondition is weaker than both, the postcondition of `createCell` and of `createIncreasedCell`.*

Since the second practice is more flexible we decided to go with it. The approach also does not suffer from a well known phenomenon called *Recursion*

through the store. What it exactly is and why this design is resilient against it will be discussed in section 3.6. But the added flexibility comes at a cost that we have to do difficult proves during program verification.

To add specifications we used the entailment operator and representative values for pre- and postconditions. The entailment operator is written as `|=` and its formal semantic is

$$\texttt{A |= B} \quad \Longleftrightarrow \quad \forall \textit{states} : \texttt{A} \implies \texttt{B}$$

Note *This will be defined later more formally.*

The syntax of the place-holders for pre- and postconditions is

```
pre(cre, value)
post(cre, value, myCell)
```

Note *The place-holders specify which delegate they belong to with the first argument. The then have a list of formal names for the arguments of* `cre`*. The* `post` *expression has additionally a list of formal names for return values.*

This structure can be used to specify delegates in the following manner: Assume `cre` is a delegate pointing to `createCell`, but the verifier does not know about this. The only hints for the verifier are the following entailments in the state:

```
i > 5 |= pre(cre, i)
post(cre,i,result) |= result.valid && result.getValue()>=i
```

Those both are valid entailments to have in the state, we can check this quickly since we know the real pre- and postconditions:

$$\forall \textit{states} : \texttt{i > 5} \implies \texttt{true}$$

The real precondition of `createCell` is `true`, thus this trivially holds. For the postcondition we need to check:

$$\forall \textit{states} : \texttt{result.valid \&\& result.getValue() == i}$$

$$\Longrightarrow$$

$$\texttt{result.valid \&\& result.getValue() >= i}$$

This holds as well because in these proofs access permissions and predicates can be treated like boolean functions.

Note *The effect of* $\forall states$ *is that it quantifies over the variables* `i` *and* `result`*.*

Note *The place-holders* `pre` *and* `post` *can only occur inside of entailment expressions. An entailment has either a* `post` *on its left side or a* `pre` *on the right side. These entailments are named post-entailment and pre-entailment respectively. The entailment body is defined as the part of the entailment which is not a pre or a post expression. The scope of the body are the formal arguments from the pre or post statement.*

Since we have these predicates in the state the verifier can still verify a delegate call to `cre`, even though he has no information over the real precondition.

```
call myCell := cre(7)
```

The verifier does not know the real precondition of `cre`. But it has an entailment in the state which specifies a stronger precondition. We can use that out of two reasons:

- The entailment quantifies over all states, so too in this state the implication holds.
- If the stronger precondition holds, then the real pre condition must hold too, so it is sound to exhale the stronger one instead.

We can do this by plugging in the actual argument `7` into the formal argument `i` of the pre expression and then exhaling the entailments body which is `7 > 5`. Since this holds we can successfully call the delegate.
With a similar reasoning we conclude that the body of the post-entailment can be inhaled because it is a weaker condition weaker that the real postcondition. In this example we can inhale `myCell.valid && myCell.getValue() >= 7`.
For this operation to be sound we can only be allowed to store stronger preconditions and weaker postconditions in the state.

Note *To increase readability, the entailment sign can also be turned around, such that both place-holders stand on the left side.*

```
pre(c) =| A
post(c) |= B
```

3.4.1 Creating an Entailment

Entailments are created implicitly when a delegate object is created. The verifier adds automatically one pre-entailment and one post-entailment to the state. These entailments hold the real contracts of the at that point known target method.
After the instruction `var cre := newdel Creator(this.createCell)` the two entailments can be found in the state:

```
pre(cre, i) =| true
post(cre,i,myCell) |= myCell.valid && myCell.getValue()==i
```

These entailments do hold because both sides of the entailment are exactly the same if you replace the place-holders with what they stand for. Let us define a formal semantic for entailments such that we can reason more clearly over them.

Definition 3.1 *A* mathematical predicate *is a boolean expression over some Variables. The meta variables* `A` *and* `B` *are used for predicates.*

Note *There are no access permission included in this definition. All the proofs that follow work very similarly under the consideration of access permissions.*

Definition 3.2 *A* state φ *is a mapping from Variables to Values. Mathematical predicates can be* evaluated *in a state* φ *written as* `A(φ)`*. The result of this is the boolean value of the predicate where each variable was substituted with the value it had in the mapping* φ*. We say* `A` holds in state φ *if* `A(φ)` *evaluates to true.*

Note *This definition defines not a symbolic state, but the real state during the execution of a program.*

Definition 3.3 *The entailment operator is formally defined as:*

$$\texttt{A |= B} \iff \forall \varphi : State.\ \ \texttt{A}(\varphi) \implies \texttt{B}(\varphi)$$

Definition 3.4 `pre(c)` *is the mathematical predicate defined by the precondition of the target method of* `c`*. Since the target method of a delegate and the precondition of a method are both immutable,* `pre(c)` *never changes.*

Definition 3.5 `post(c)` *is the mathematical predicate defined by the postcondition of the target method of* `c`*. Since the target method of a delegate and the postcondition of a method are both immutable,* `post(c)` *never changes.*

Theorem 3.6 *Assume* `A` *is the predicate of the precondition of the target method of* `c` *and* `B` *is the respective postcondition. Then the entailments* `pre(c) =| A` *and* `post(c) |= B` *assumed after a delegate creation hold.*

Proof We prove that `pre(c) =| A` holds by using the entailment definition and get

$$\forall \varphi : State.\ \ \texttt{A}(\varphi) \implies \texttt{pre(c)}(\varphi)$$

Note that both sides of the implication are the same mathematical predicate. Since `pre(c)` does not change after c is created (target of a delegate is immutable), this implication holds in all states.
`post(c) |= B` can be proven in the same way. □

3.4.2 Using an Entailment

Entailments are used when the symbolic execution reaches a delegate call statement. At that point it would like to exhale the precondition of the target method and then inhale its postcondition. As marked before we can also exhale stronger preconditions and inhale weaker postconditions. The next two sections cover these operations.

Note *These two operations are handled separately since the fork and join operations split the call operation in the exact same way.*

Precondition

Theorem 3.7 states that it sound to verify a delegate fork instruction of delegate `cre` when we can exhale a condition `A` from a state containing the entailment `pre(cre) =| A`.

Theorem 3.7 *Assume a state φ. If we want to prove that `pre(cre)` holds in φ and we can assume the entailment `pre(cre) =| A`, then it is sufficient to prove that `A` holds in φ.*

Proof Let φ be a state and assume `pre(cre) =| A`. Using the entailment definition and eliminating the $\forall$ we get that `A(`φ`)` $\implies$ `pre(cre)(`φ`)`. Now it is simple to show that if we can prove `A` holds in φ, then `pre(cre)` will hold in φ too. □

Note *If we can prove `A` in φ, then `pre(cre)` holds in φ, otherwise we do not know whether `pre(cre)` holds.*

Note *In the actual implementation, the user should make sure that there is one fitting pre-entailment in the state. If there is none, a delegate fork can not be verified. If there are multiple pre-entailments, the verifier chooses one that works arbitrarily.*

Postcondition

A similar proof can be made for post-entailments. Here we prove that it is sound to add a condition `B` to the state after a delegate join instruction of delegate `cre` when the state contains the entailment `post(cre) |= B`.

Theorem 3.8 *Assume a state φ. It is sound to assume the predicate `B` in a state φ when `post(cre)` holds in φ and we additionally know `post(cre) |= B`.*

Proof Let φ be a state and assume we have `post(cre) |= B` and `post(cre)(`φ`)`. Using the definition of entailments we get `post(cre)(`φ`)` $\implies$ `B(`φ`)` which results in `B(`φ`)`. □

Note *When we consider access permissions we have to make sure that the permissions are removed from the state only once.*

Note *The user is responsible to make sure that there is one fitting post-entailment in the state. If there are multiple, one is chosen arbitrarily. If there is none, nothing will be inhaled.*

Additional Notes

Note *Both proofs can be adjusted to hold with the consideration of access permissions, predicates and old expressions.*
The entailments remain in the state after these operations (an entailment holds in every state, it also holds in a state where it was used once or more).

A post-entailment can also contain old expressions in its body. In that case the entailment operator not only quantifies over the current state, but also over all old states. For old expressions to work in post-entailment bodies the "old state" is stored when the precondition is checked. This is done by adding the whole state to the state during the fork and then old expressions can be evaluated in this stored state while joining.

3.4.3 Proving an entailment

Let us again have a look at `creatorFactory` at the start of section 3.4. At the end of this method the verifier has to prove that the entailment

```
post(cre, i, myCell) |= myCell.valid &&
    value <= myCell.getValue() &&
    myCell.getValue() <= value + 1
```

holds after the method is executed. This is difficult because the verifier does not know the real postcondition `post(cre, i, myCell)` dynamically. Luckily the delegate creations each put one of the entailments below is in the state:

```
post(cre, i, myCell) |=
    myCell.valid && myCell.getValue() == value
post(cre, i, myCell) |=
    myCell.valid && myCell.getValue() == value + 1
```

Both of these can be used to prove the entailment above. Using theorem 3.10 we see that it is sufficient to prove

```
myCell.valid && myCell.getValue() == value

|=

myCell.valid &&

value <= myCell.getValue() && myCell.getValue() <= value + 1
```

This principle allows the verifier to automatically prove entailments. Below are the formal theorems used for it.

Theorem 3.9 *If we want to prove* `pre(cre) =| A` *and we can assume an entailment* `pre(cre) =| B`*, then it is sufficient to prove* `A |= B`*.*

Theorem 3.10 *If we want to prove* `post(cre) |= A` *and we can assume an entailment* `post(cre) |= B`*, then it is sufficient to prove* `B |= A`*.*

Proof (Proof of theorem 3.9) We have `pre(cre) =| B` and assume `A |= B` holds. Then we have

$$\forall \varphi : state.\quad \texttt{B}(\varphi) \implies \texttt{pre(cre)}(\varphi) \land \forall \pi : state.\quad \texttt{A}(\pi) \implies \texttt{B}(\pi)$$

$$\iff$$

$$\forall \varphi : state.\quad \big(\texttt{B}(\varphi) \implies \texttt{pre(cre)}(\varphi) \land \texttt{A}(\varphi) \implies \texttt{B}(\varphi)\big)$$

$$\implies$$

$$\forall \varphi : state.\quad \texttt{A}(\varphi) \implies \texttt{pre(cre)}(\varphi)$$

$$\iff$$

$$\texttt{A |= pre(cre)}$$

□

The proof of theorem 3.10 is very similar and let as an exercise for the reader.

Proof with Old Expressions

Post-entailment bodies can also contain old expressions. These should in theory be provable in the same way, but it's still worthy to note the differences. Let us have a look at the method `increaseCell`.

```
method increaseCell(c: Cell)
  requires c.valid
  ensures c.valid && c.getValue() == old(c.getValue()) + 1
{
    c.inc()
}
```

This method has a postcondition with an old expression. Assume we have an entailment for that postcondition in the state and we want to prove the entailment

```
post(inc,c) |= c.valid && c.getValue() > old(c.getValue())
```

Using theorem 3.10 again we see that it is sufficient to prove

```
c.valid && c.getValue() == old(c.getValue()) + 1

                    |=

c.valid && c.getValue() > old(c.getValue())
```

The difference is that values inside old expressions are not equal to the same values outside (looking at `c.getValue()`). The proof itself works in the same way as without old expressions.

3.5 Hidden State with Delegates

State capturing occurs when an anonymous method accesses variables which are not defined in the anonymous method, but in the method surrounding it. These variables are called captured and they may be out of scope when the delegate is executed.
The current system does not exactly allow state capturing, but some state can be hidden. Let us illustrate this with the example of a `counter` delegate that counts the number of times it was called.

```
class CounterObject {
    var value :int;

    method count() returns (result: int)
        requires valid
        ensures valid
        ensures result == old(getCount())
        ensures getCount() == old(getCount()) + 1
    {
        unfold valid;
        result := value;
        value := value + 1;
        fold valid;
    }

    function getCount() : int
        requires valid
    {
        unfolding valid in value
    }

    predicate valid {
        acc(value);
    }
}
```

The following method `createCounter` creates a delegate pointing to the `count` method of a `counterObject c`.

```
delegate Counter() returns (int)

class Client {
    method createCounter() returns (res: Counter)
        ensures pre(res) =| c.valid
        ensures post(res, i) |= c.valid &&
                i == old(c.getCount()) &&
                c.getCount() == old(c.getCount()) + 1
        ensures c.valid && c.getCount() == 0
    {
        var c := new CounterObject
```

```
        c.value := 0;
        fold c.valid
        res := newdel Counter(c.count)
    }
    method client() {
        call c := createCounter()
        call i := c()
        call j := c()

        assert i == 0 && j == 1
    }
}
```

Note that the `createCounter` method does not specify any information about `c` which causes the postcondition of `createCounter` to be invalid. The reason for this is that it refers to `c`, an object that is unknown at the callers side. Non the less we want a specification that allows a `client` to verify calls to `res`.

Ghost Functions and Predicates

The basic issue is that the count method has a specification about the `this` object, which is out of scope in the `client` method. We introduce ghost functions and predicates as a way to allow specifications to access functions and predicates of the delegate target object. A ghost function is a function of the delegate object which points to a function of the delegate target object and a ghost predicate is the same for predicates.

Note *This is, `res.getCount()` points to `(target object of res).getCount()`.*

To make this approach type-safe the delegate type definition has to be enriched with these new members.

```
delegate Counter() returns (int) {
    function getCount(): int
    predicate valid
}
```

When we now create a Counter delegate to `c.count`, we have to make sure that `c` also provides the function `getCount()` and the predicate `valid`. This is checked at delegate creation, and when the delegate target object does not provide the full interface an error is thrown during type-checking.

Note *A delegate type definition can have any number of ghost functions and predicates.*

The method specification of `createCounter` can now be rewritten to the following:

```
ensures pre(res) =| res.valid
ensures post(res, i) |= res.valid &&
            i == old(res.getCount()) &&
            res.getCount() == old(res.getCount()) + 1
ensures res.valid && res.getCount() == 0
```

Here we also see how ghost members are evaluated. The predicate `res.valid` can be exhaled since it points to `c.valid` which is available at the end of `createCounter`. Similarly `res.getCount()` points to `c.getCount()` which is `0` in that state. With this specification the method `client` can be verified.

Note *Ghost Functions can only be used as part of specifications and ghost predicates can not be folded or unfolded.*

Note *This concept allows the notion of* possible target types of a delegate type. *It is the set of all classes that implement the full interface defined by a delegate type.*

3.6 Circularity Issue

The circularity issue, or more generally known as the phenomenon *Recursion through the store* arises when a specification refers to itself and causes an inconsistency. An example would be a method with a precondition that says `pre(c) = false`. If we now set `c` to be that method we get a recursive inconsistent definition.
In our system this issue does not emerge, thanks to the one-sided relation between pre- and postconditions and their specifications. This is, we can only ever say that one condition is implied by another condition, never that they are equal.

3.7 Higher order functions

Higher order functions are functions or methods which work with delegates without knowing their exact specification. They are necessary for most design patterns involving delegates, for example *Visitor* or *Chain of Command*. The current system has to be adapted to make higher order functions possible because we can only specify exact pre- and postconditions of delegates, and higher order functions should work with unknown delegates.
Here is a basic example, the method `Conditional` which takes a boolean and a delegate and executes the delegate if the boolean is `true`. This method does not care about the exact requirements or effects of the delegate, it only needs that if the first argument is `true` then the delegate should be executable. To make this specifiable in the current system, we introduce a new mechanism.

```
delegate Command()

method Conditional(b: bool, c: Command)
    requires b ==> pred(c)
    ensures b ==> postd(c)
{
    if (b) {
        call c()
    }
}
```

`pre` and `post` expressions standing on their own, not inside an entailments allow us to specify higher order methods. A client can now use this method with a variety of different delegates and still rely on the exact behaviour for each call:

Note *Due to a limitation of the parser we use the keywords `pred` and `postd` for free `pre` and `post` expressions.*

```
class Client {
    var value: int
    method set3()
        requires acc(value)
        ensures acc(value) && value == 3
    {
        value := 3
    }
    method set5()
        requires acc(value)
        ensures acc(value) && value == 5
    {
        value := 5
    }
    method client()
        requires acc(value)
    {
        value := 0
        var c1 := newdel Command(this.set3)
        var c2 := newdel Command(this.set5)

        call Conditional(true, c1)
        assert value == 3

        call Conditional(false, c2)
        assert value == 3

        call Conditional(true, c2)
        assert value == 5
    }
}
```

At the `call Conditional` statement we do the normal pre-exhalement. After the call we can inhale the postcondition which is in this case `post(c1)`, the inhalement of this is the same as discussed in section 3.4.2. The only new operations become apparent during the verification of method `Conditional`. Assuming `b` is true we need to inhale `pre(c)`. This operation is not standard, but it is intuitively clear that we have to add this `pre(c)` expression to the state. This works fine when we want to exhale `pre(c)` at the delegate call instruction: `pre(c)` is already in the state and we only need to remove it to allow the call.
When we return from the call we are inhaling `post(c)`. Since there is no post-entailment for `c` the `post(c)` was originally just thrown away. We can of course do better by adding it to the state, because at the end of the method we'll need it to satisfy the postcondition.

All in all this is not a too graving modification of the system. The appendix 9.1 contains an example where the delegate works with input and output. Then, in appendix 9.2 we see the first limitation of this system - composition of delegates can not be done. In section 5.2.6 we see the other limitation of this approach - `old` expressions will not work with higher order functions.
This mechanism was implemented only one day before the end of the project and is therefore not fully tested. Basic examples were verifiable and the feature seems to be sound.

Chapter 4

Viper Verification Infrastructure

Verifying the added delegates of Chalice is hard. To understand what needs to be done the verifier must be understood first. This Chapter serves as introduction to the back-end of the Viper project. Section 4.1 describes the Silver language and section 4.2 looks into Silicon.

As mentioned in the introductions, the viper verification system works in four steps:

- Chalice code is parsed into a Chalice AST
- This AST is then type-checked
- After that it is translated into a Silver AST
- Then this Silver AST is verified using Silicon

The first two steps should already be clear at that point. This and the next chapter discuss the last two steps.

4.1 Introduction to Silver

Silver is an intermediate language designed by ETH for the Viper project. It supports fractional permissions and has a simpler, more dedicated syntax than Chalice. There are also translations from other front-end languages like Scala into Silver.
The type-checking is done in Chalice, thus Silver programs do not have any type information. This makes classes obsolete and thence all classes are merged into one big class named `ref`. This class contains all members of all original classes and also some generated members which are not present in the original Chalice program.
The Silver `ref` class can have members of the following types:

- Fields
- Methods with pre- and postconditions
- Functions with preconditions
- Predicates (including invariants)

4.1.1 Translation

Members are translated into equivalent members for the `ref` object. To avoid naming conflicts they are renamed to a combination of their class and the original name. For example the `value` field of `Cell` is translated into a field `Cell_value` of the only class `ref`. Silver does not provide a `this` field, so methods, functions and predicates get an additional argument `this`.
Chalice's object creations are translated into `new ref` statements. This instruction adds permissions to all fields to the state, but not all fields are used by the Chalice program. Method calls are translated into simple `exhale` and `inhale` statements. For example a call to a method with precondition `pre` and postcondition `post` is translated into

```
exhale pre
inhale post
```

All other instructions have their equivalent in Silver.

4.2 Introduction to Silicon

Silicon is a verifier based on symbolic execution. A short introduction on symbolic execution can be found in section 2.3.1. This introduction is enough to understand this thesis, the next section further describes the parts of Silicon for those who are interested.

4.2.1 The parts of Silicon

Silicon is designed in a cake-pattern which makes it very flexible in regards of switching out components. It is written almost purely functional and for most components a continuation passing style is used. This makes Silicon a hard verifier to understand, non the less I learned a lot about Silicon's inner workings. Here are the components of Silicon, even if they are not relevant in this theses, they were relevant for the implementation of the system.

Verifier The Verifier is the main component. It takes a program and passes each method to the executor.

Executor The Executor steps through a method and does predefined actions depending on the statement that is processed.

Producer Inhalations of expressions are handled by the Consumer. It changes the state according to the given expression.

Consumer Exhalations of expressions are handled by the Producer. It asserts conditions and can remove access permissions depending on the given expression.

Evaluator The Evaluator takes an expression and evaluates it to a symbolic term.

Decider The Decider is the proofer of Silicon, it can check if a term evaluates to true or false. It uses Microsoft's extremely strong theorem proofer called Z3.

Reminder *As mentioned before Silicon verifies modularly, each method by itself. This is possible because methods can be linked by their contracts.*

Silicon divides the symbolic state into these three (and more) parts:

Store The store is where the mapping from variable names to symbolic terms is defined.

Path-condition Known relations between terms are stored in the path-condition.

Heap The heap stores the access permissions, predicates and values of fields (as terms).

As you might have guessed the entailments will later be stored in the heap, in the same way as access permissions are stored. Elements of the heap are called chunks and we introduce two new sorts of chunks: `DelegateChunks` and `StateChunks`. `DelegateChunks` are used to store entailments, `pre` and `post` expressions and `StateChunks` can store a whole state.

Chapter 5

Modifications to the Viper Back-end

The goal of the thesis was to leave the back-end of Viper (Silver and Silicon) as it is and try to encode all special behaviour of delegates into the translation from Chalice to Silver. This was not possible; three new AST nodes had to be added to Silver and Silicon had to be modified to handle these nodes.

This chapter discusses the necessary modifications to the Viper back-end in order to verify the delegates added in section 3. Section 5.1 describes the translation of the added Chalice AST nodes into Silver and section 5.2 shows how Silicon has to be modified to handle delegates.

5.1 Translation to Silver

This section specifies a way to translate each added AST node into Silver statements such that Silicon can verify them. To do this three new Silver AST nodes had to be introduced to be able to encode all the behaviour of delegates in Silver. The new nodes are `Entail`, `Pre` and `Post`.

Reminder *The original translation from Chalice to Silver removes all type information from the program. Most Chalice Statements are encoded into a series of instructions in Silver. For example consider a call to a method with precondition* `pre` *and postcondition* `post`*. The translation of the call results in*

```
exhale pre
inhale post
```

Note *Delegate type definitions are not translated into Silver code - they are only needed for type-checking which is handled by Chalice.*

Note *The translation will be explained using the example presented in section 3 - a delegate targeting the method* `createCell`*. Of course the implemented translation can also handle more general programs.*

5.1.1 Additions to the Silver AST

The new AST nodes in Silver are the three expressions `Entails`, `Pre` and `Post`. These three nodes had to be added to support the new functionality - and they are also sufficient for all the features of the design.
Consequently the translation of Chalice entailments, pre and post expressions is quite easy - they are translated one to one into the corresponding Silver expressions. Silicon needs to be modified to understand `inhale` and `exhale` operations for these three nodes, but this will be covered in section 5.2.

Note *As discussed in chapter 3 the symbolic state can contain entailments. These entailments are identified with the delegate object they're associated with as identifiers.*

Note *In Silver the `pre` and `post` expressions can take an optional additional last argument. This argument is unused when translating `pre` or `post` expressions from Chalice to Silver, it is only used for `pre` and `post` expressions generated by the translation (as seen and explained in detail in section 5.2.4).*

5.1.2 Translating a Delegate Creation

To understand the translation of a delegate creation to Silver, let us review how it looks like in Chalice. Using the example from section 3, a delegate of type Creator can be constructed using the code

```
c := newdel Creator(o.createCell)
```

Reminder *As discussed in section 3.3.1 and section 3.4.1 after a delegate creation we have*

- *pre- and post-entailments in the state*
- *`c` is an new object that represents a delegate*

Keeping this remark in the head the translation of a creation can be written as the pseudo-code below (Silver does not have a written syntax, only an abstract syntax tree).

```
c := new ref
c.delegateTarget := o
c.delegateTargetType := 2
inhale pre(c, value) =| value >= 0
inhale post(c, value, myCell) |= myCell.value == value
```

The translation into Silver adds two new fields `delegateTarget` and `delegateTargetType` to the `ref` class. They are only added if delegates are used in the program. `delegateTarget` stores the target object of the delegate. It is later used to access the "hidden state" also known as the `this`

object of the invoked method. The type of the target object is stored in `delegateTargetType`. We need that type later in the translation of expressions that access ghost functions and predicates. The type is stored as an integer, in detail it is the index of the type - with respect to all possible target types of `Counter`.

Since inhaling is the operation of adding specification to the state it is only logical to `inhale` entailments in order to add them to the state. This process is described in detail in section 5.2.1. The real conditions are taken from the specification of the target method. All references to the `this` object in these conditions are replaced with references to `c.delegateTarget`, which is coincidently the same logical object.

Note *To avoid conflicts the names of all formal arguments and return values of the entailments are changed to new unique identifiers.*

5.1.3 Delegate Call, Fork and Join

Delegate calls, forks and joins do not exist in Silver as AST nodes. They are translated into the appropriate inhales and exhales. In order to reduce the overhead of translating all three operations into Silver, delegate calls are first transformed into a fork followed by an immediate join instruction. The resulting fork and join operate on a unique token for this delegate call.

Fork

A Chalice delegate fork expression looks like this:

```
fork t := c(7)
```

Reminder *The `fork` statement produces a token `t` which can be used to `join` the method later. Section 3.4 describes the process that should happen during a delegate fork.*

- *Search the state for a pre-entailment of `c` and try to exhale its body.*
- *Store the current state for later ("old state" in the join operation).*

Searching an entailment is not a standard procedure of Silicon. This is why the translation introduces the new operation of exhaling a `pre` expression. With that the translation of the fork statement results in

```
t = new ref
t.delegate := c
t.Creator_argument0 := 7
exhale pre(c, 7, t)
```

The token stores a lot of information which will be used later in the `join` operation.

- The delegate object will be stored in a field `delegate` which is added to the `ref` class.
- All arguments of this invocation will be stored in fields added to the `ref` class. This is done once per delegate type.

The process of the exhaling a `pre` expression will be explained in detail in section 5.2.2. In general it finds a pre-entailment in the state with the identifier `c` and then exhales its body where it substituted the formal arguments with the actual arguments. You may have noticed the unexpected last argument `t` of the `pre` expression - it will be used to store the "old state" as explained at the end of this section.

Note *Since the token can not be given into another context we do not have to manage access permissions of these added fields.*

Join

Assume `t` is a token which was created by a previously forked delegate of type `Creator`. Then a `join` statement for this token has the following code:

```
join myCell := t
```

Reminder *According to the specification of section 3.4 a `join` statement has the effects*

- *Search the state for a post-entailment for the delegate `c`, then inhale its body.*
- *Use the state stored in the `fork` expression to evaluate `old` expressions.*

This searching is another new operation for Silicon. It is specified as an inhale of a post expression. This results in a simple translation:

```
inhale post(t.delegate, t.Creator_argument0, result0, t)
myCell := result0
```

All the fields previously stored in the token are now used in the `post` expression. The process of inhaling a `post` expression will be explained in section 5.2.3. On a high level it finds a post-entailment in the state and inhales its body where it substitutes the formal arguments and return values with the actual arguments from the token and some fresh unique local variables like `result0`. After that the actual result variable `myCell`, which could already have constraints in the state, is assigned with `result0`, which does only have the constraints from this `join`.

The next section explains why the post expression needs the token as the last argument.

Old Expressions in Post-entailments

Post-entailments can have `old` expressions in their body. When inhaling the postcondition they need to be evaluated in the state of the corresponding `fork` statement when the token is created. This is done by storing the whole state in the state. To find this state later we use the token as identifier. This is why the `pre` and `post` expression take the token as the last argument - the state is stored and retrieved during these operations.

5.1.4 Ghost Functions and Predicates

Reminder *In section 3.5 ghost functions and ghost predicates are described. They allow references to the state that is hidden by the delegate. When evaluated they should return the value of the function or predicate of the delegate's target object.*

Reminder *A normal function access `myCell.getValue()` (assuming `myCell` is of type `Cell`) is translated into `Cell_getValue(myCell)` in Silver. Predicates are translated similarly.*

As noted, ghost function applications and ghost predicate accesses should point to the target objects' functions and predicates with the same name. The difficulty arising is that a ghost function can point to functions with the same name of many different classes. In Silver these functions will all have different names. Luckily we have stored the type in the delegate object during its creation. This allows us to select the correctly typed function dynamically. A ghost function application `c.getValue()` can be translated into

```
c.delegateTargetType == 0? Cell_getValue(c.delegateTarget)
    : (c.delegateTargetType == 1 ? ... : ...)
```

We have to enumerate each possible target class for the delegate type here. In the end the verifier picks the correctly typed expression during verification. A similar translation is done for ghost predicate accesses.

This implementation has issues in states where `c.delegateTargetType` is not known. Below is such an example. It uses the `Counter` delegate type as introduced in section 3.5.

```
method foo(d: Counter)
    requires d.valid && d.getCount() == 0
    requires pre(d) =| d.valid
    requires post(d, res) |= d.valid &&
        res == d.getCount() &&
        d.getCount() == old(d.getCount()) + 1
    ensures d.valid && d.getCount() == 1
{
    call d()
}
```

There are many ghost function and predicate accesses in this example, but `d.delegateTargetType` is unknown. Assuming there are at least two possible target classes - there will be multiple possible targets for the ghosts `valid` and `getCount`.
The way Silicon works in this case is that it will try all cases once and it will only succeed if all of the cases succeed. This introduces two issues:

- The method gets verified multiple times for each possible target type once. This can slow the verification process significantly.
- Functions can require access predicates. If there exists a function `getCount` which requires other access predicates than `valid` the verification of the whole method will fail.

Both these issues can be solved by renaming the functions of all classes which are not intended to be targeted with a `Counter` delegate.

Issues with Delegate Object fields

The example in the previous section has a second, more serious issue - we do not get access to the fields `delegateTarget` and `delegateTargetType` and also we do not know whether `delegateTarget` is null or not. This issue only appears during ghost function or predicate applications since that is the only time these two fields are accessed. There is no easy way to allow the user to specify access to these fields because they do not appear in Chalice code.

The implemented solution was to add the following expression to each pre-, postcondition and entailment entailment that required access to one of the fields:

```
acc(c.delegateTarget, _) && c.delegateTarget != null &&
    acc(c.delegateTargetType, _)
```

Reminder *These expressions give wild-card access to the fields. Wild-card means an arbitrary positive amount of access.*

This access must also be provided for each method or delegate call (and fork) that requires it, since the user does not have the possibility to ensure the availability of these access permissions. This solution does not cover all cases - access to these fields can also be hidden inside of predicates which would result in missing access permissions.
The more general solution would have been to exclude these fields from the whole access mechanic, but this was not possible on the side of Silicon.

Even after that addition there was still a bug in the implementation: Assume we create a delegate to the `inc` method of `Cell` and therefore we know the target type of the resulting delegate object. If we now want to prove the following postcondition

```
c.delegateTargetType == 0 ? Cell_valid(c.delegateTarget) :
    OtherClass_valid(c.delegateTarget)
```

the verifier forgets about the field delegateTargetType again, since it is not mentioned in the postcondition. The verifier can then not verify `OtherClass_valid(c.delegateTarget)` since we only have `Cell_valid(c.delegateTarget)` in the state. This limitation is severe and many examples do not work with it.

5.2 Modifications of Silicon

In the previous section we saw that the translation of delegates is only possible if Silicon is modified to support the used operations. This section specifies the details of these operations. Table 5.1 has an overview what operations Silicon needs to support.

Table 5.1: New Silicon Operations

	Entailment	`pre`	`post`
`inhale`	required	only for 3.7	required
`exhale`	required	required	only for 3.7

The operations marked with "only for 3.7" are only needed for higher order functions. They were implemented at the end of the project and they were not tested very much.

5.2.1 Inhaling an Entailment

Reminder *Inhaling an entailment is used during delegate object creation (section 5.1.2) to simply add an entailment to the state.*

Entailments are stored in the state in a similar way access permissions are stored. An entailment inhalation adds the entailment to the state. The symbolic value of the delegate object of the entailment is used to identify the entailment later.

Note *The entailment is stored unevaluated as an expression.*

5.2.2 Exhaling a pre Expression

Reminder *A pre expression is exhaled during a delegate fork operation of Chalice. The exact translation can be found in section 5.1.3. According to section 3.4 axhaling a* `pre` *expression should cause Silicon to find a pre-entailment for the delegate*

and exhale its body instead. The formal arguments of the entailment must be replaced with the actual arguments that are contained in the `pre` expression we want to `exhale`.

Let us assume we want to exhale the pre expression `pre(c, 7, t)` out of the state. Below is pseudo-code that resembles what Silicon has to do in this situation.

```
exhale(pre(c, 7, t)) {

    store the chunk (t, current state) in the state

    for each entailment e in the state
        if (e is a pre-entailment for c) {
            e' := substitute the formal arguments in the
                  body of e with the actual arguments
                  (here 7)

            exhale e'
                failure: reverse this exhale and continue
                         with next entailment
                success: return success and continue with
                         the modified state
        }

    no entailment was successful: return failure
}
```

This procedure finds a fitting pre-entailment in the state and exhales its body. It returns either when the first entailment can be exhaled or fails when no entailment can be exhaled (this results in a verification error). It also stores the current state with the token as identifier as discussed in section 5.1.3.

Note *After the substitution of the formal arguments there are no free variables in `e'`, because the scope of an entailment is only its formal arguments and return values.*

Note *This operation can modify the state, for example when access permissions are consumed.*

Note *This pseudo-code looks very different form the solution implemented in Silicon since Silicon is programmed in an almost purely functional matter.*

5.2.3 Inhaling a post Expression

Reminder *A `post` expression is inhaled during a delegate join operation. Section 5.1.3 describes the exact translation and from section 3.4 we can learn what has to happen. Inhaling a post expression Silicon should find a post-entailment for*

the delegate and inhale its body instead. The formal arguments and return variables must be substituted with the actual arguments and return values given in the `post` expression.

Here is an example how Silicon handles the inhalation of `post(c, 7, result0, t)` into the state.

```
inhale(post(c, 7, result0, t)) {

    OS := retrieve the old state with identifier t

    for each entailment e in the state
        if (e is a post-entailment for c) {
            e' := substitute the formal arguments in the
                  body of e with the actual arguments
                  (here 7 and result0)

            inhale e' and use old state OS

            return success and continue with the modified
                   state
        }

    no entailment was successful: return success and
        continue with the unmodified state
}
```

This procedure finds a fitting post-entailment in the state and inhales its body. If there are `old` expressions in the body they are evaluated in a state which was stored during the `exhale` of the `pre` expression. This procedure returns without modifying the state if no fitting post-entailment was found.

Note *Again `e'` does not contain any free variables after the substitution.*

Note *If there are no `old` expressions in the body of the entailment, the old state is not necessary.*

5.2.4 Proving an Entailment

Entailments are exhaled when they need to be proven, for example at the end of a method that ensures some entailment.

Reminder *How an entailment can be proven is described in detail in section 3.4.3.*

The verifier can only do this proof when there is already an entailment for the same delegate in the state. In that case, using theorem 3.9 and 3.10, it is possible to reduce the to prove statement into an entailment `A |= B` of two known expressions `A` and `B`. The verifier can prove such entailments using the following steps:

- Create a new, empty state φ
- Inhale A into φ with fresh symbolic values assigned to all formal arguments.
- Try to exhale B from φ with the same symbolic values assigned to the corresponding arguments.

The code below shows how Silicon handles the proof. There are two procedures - one for pre-entailment proofs and one for post-entailment proofs.

Reminder *The difference between a pre-entailment and a post-entailment is that a new pre-entailment can only be stronger and an new post-entailment can only be weaker.*

```
exhale(newEntail = pre(c, x) =| x > 10) {
    for each entailment e in the state
        if (e is a pre-entailment for c) {
            for example e = pre(c, a) =| a > 0

            prove newEntail is stronger than e
                success: return success
                failure: continue with next entailment
        }

    no entailment was successful: return failure
}
exhale(newEntail = post(c, x, y) |= y >= x + 1) {
    for each entailment e in the state
        if (e is a post-entailment for c) {
            for example e = post(c, a, b) |= b == a + 1

            prove e is stronger than newEntail
                success: return success
                failure: continue with next entailment
        }

    no entailment was successful: return failure
}
prove_stronger(strong, weak) {
    substitute the formal arguments and return values of
        strong with the formal arguments of weak

    create empty state s0

    inhale strong's body into s0

    exhale weak's body from s0
        return the result of this operation
}
```

The first two procedures both first find a fitting entailment and then use the `prove_stronger` procedure to prove the right relationship between the original and the new entailment. If this succeeds for one entailments of the state the operation succeeds and fails otherwise.

Examples

Here are the proofs for the examples we used in this section:

```
inhale a > 10
exhale a > 0
```

This is of course possible and thus the entailment `pre(c, x) =| x > 10` can be exhaled. For the post-entailment we have the following proof:

```
inhale y == x + 1
exhale y >= x + 1
```

This holds too and the entailment `post(c, x, y) |= y >= x + 1` can be exhaled.
This procedure works when there are no `old` expressions in post-entailments. The next section will discuss what happens if there are old expressions.

Note *The actual implementation uses the usual state instead of a new empty state. This can be done because all formal arguments and return values have unique names, such that no variable has associations with any other that is already in the state.*

5.2.5 Proving an Entailment with Old Expressions

In Silicon, the proof with old expressions is especially difficult, because we have only one state at our disposal. But this should be sufficient, since we do not want to evaluate an expression, but only prove if one of the conditions is stronger than the other.

Let us look at an example. Assume we have the post-entailment below in the state.

```
post(c, myCell, i) |=
    myCell.valid && i == old(myCell.getValue())
```

We should be able to infer the weaker post condition from the original post-entailment:

```
post(c, myCell, i) |=
    myCell.valid && i > old(myCell.getValue() - 3)
```

Generally this should be done the same way as before - by inhaling the body of the original entailment into an empty state and then trying to exhale the

body of the weaker entailment. But since old expressions can not be inhaled they have to be transformed into something else first.

The next subsection describes the obvious attempt to handle this situation and then shows that it can fail. The subsection after that describes a better solution that does not have the previous weaknesses but is much harder to implement. The last subsection then describes the implemented solution, which is not complete but still sound because there was not enough time to implement the correct solution.

First attempt

This is a concept we came up early and we were convinced it was strong enough to allow a correct proof. The concept was to rename all variables in `old` expressions and then replace the `old` expressions with their content. For example:

```
old(myCell.getValue())
```

was replaced with

```
old_myCell.getValue()
```

To allow this to work fresh symbolic values were assigned all these new variables and permissions to fields of these variables were added to the state. Unfortunately there were always some edge cases where this did not work. Finally we discovered that when the expression `x == old(x)` was in the body of an entailment, then the proof was unsound. Here an example:

```
post(c, x) |= old(x) == x && acc(x.f) && old(x.f) == 0
```

The `old` expressions of the body of this entailment were replaced and in the end something like the expression below was inhaled.

```
old_x == x && acc(x.f) && old_x.f == 0 && acc(old_x.f, _)
```

Since `x` and `old_x` are equal there is actually more than 100% access to `x.f` inhaled, which is unsound.

In retrospective the renaming variables was also not quite the logical step, since the variables are actually the non-changing objects. Variable inside `old` expressions are always either formal arguments of the entailment or the delegate object. Return values can not be in `old` expressions. But the arguments of a method actually never change in chalice, neither does the delegate object. Thus it did not make sense to rename the variables.

A better Way

As noted in the previous subsection the renaming of the variables was not a good solution because the variables do not change between the old and

current state. The better solution is to rename all fields, functions and predicates. For example:

```
old(myCell.getValue())
```

is replaced with

```
myCell.old_getValue()
```

The appendix 9.3 contains an example why this is not enough to be complete: functions and predicates have to be unpacked and recursively renamed. It can go so far as to double the program size.

This solution was not implemented due to time constraints. I think this solution would be sound, and it would definitely be stronger than the currently implemented solution which is discussed in the next section.

Implemented solution

The implemented solution is minimal but definitively sound. It replaces whole `old` expressions with variables, and only if they are syntactically identical they are replaced with the same variable. This is a very restrictive solution, but most examples can be adjusted to work despite this restriction.

5.2.6 The remaining Operations

The remaining two operations of table 5.1 are implemented to allow higher order functions. The implementation was not fully tested, but the concept behind these operations can be found in section 3.7.

Inhaling pre expressions

When `pre` expressions are inhaled they should just be added to the state. These `pre` expressions do not have formal arguments but rather actual arguments for which the `pre` should hold. This is why those pre expressions are added to the state with all arguments evaluated to their symbolic values.

The exhale operation of `pre` expressions has to be modified too - otherwise the `pre` expressions in the state could never be used. The modification is simple: at the start of the operation the state is scanned for `pre` expressions. When one matches with the delegate and all the other arguments (as symbolic values) it can be used instead of trying to use an entailment.

Note *This manual addition of a `pre` expression has one downside: there is no token which we can use to store the "old state". This is why `old` expressions are not supported in this design of higher order functions.*

Exhaling post expressions

To enable the exhaling of `post` expressions they first need to have a way to get into the state. For this the inhale of `post` expressions is modified in the following way: If there is no suiting entailment, instead of doing nothing, the `post` expression is added to the state. This is too done with all the arguments and return values evaluated to their symbolic expressions.

Now when `post` expressions are exhaled, they can be found in the state and if all arguments and return values match the exhale is successful.

Note *`old` expressions are not supported in this design of higher order functions, because there is no "old state" to evaluate them in.*

Chapter 6

The Implementation and Examples

The implementation of the project was done in a very rushy manner towards the end of the project, especially he features regarding hidden state and higher order functions did not have the deserved time to ripe and are therefore not fully tested. As mentioned before, the entailment proof with old expressions is handled poorly by the implementation. This allowed in the end only basic examples, including all the examples mentioned in this thesis, to be verified with the implemented system. The basic examples are listed in section 6.1.

This chapter contains some further examples with delegates and shows how they could be verified or why the current system is not strong enough to verify them. In section 6.2 an example of the Command pattern is given. The next section then shows an extension of it. Both examples were not fully verifiable with the final implementation.

6.1 Basic Examples from this Thesis

6.1.1 Basic operations

This section contains the full Cell example. It was fully verifiable.

```
class Cell {
    var value: int

    function getValue(): int
        requires valid
    { unfolding valid in value }

    predicate valid { acc(value) }

    method Set(i: int)
        requires valid
```

```
        ensures valid && getValue() == i
    {
        unfold valid
        value := i
        fold valid
    }

    method inc()
      requires valid
      ensures valid && getValue() == old(getValue()) + 1
    {
        unfold valid
        value := value + 1
        fold valid
    }
}

delegate Creator(int) returns (Cell)

class Client {
    method client() {
        var cre := newdel Creator(createCell)
        call myCell := cre(7)
        assert myCell.getValue() == 7
        fork t := cre(8)
        join myCell := t
        assert myCell.getValue() == 8
    }
    method createCell(i: int) returns (c: Cell)
        ensures c != null && c.valid && c.getValue() == i
    {
        c := new Cell
        c.value := i
        fold c.valid
    }
    method createIncreasedCell(i: int) returns (c: Cell)
        ensures c != null && c.valid &&
            c.getValue() == i + 1
    {
        c := new Cell
        c.value := i + 1
        fold c.valid
    }
    method creatorFactory(r: bool) returns (cre: Creator)
        ensures pre(cre, i) =| true
        ensures post(cre, i, myCell) |= myCell != null &&
            myCell.valid &&
            i <= myCell.getValue() &&
            myCell.getValue() <= i + 1
```

```
    {
        if (r) {
            cre := newdel Creator(createCell)
        } else {
            cre := newdel Creator(createIncreasedCell)
        }
    }
}
```

6.1.2 Hidden State

The counter example serves as proof of concept for the solution for hidden state. The following program was fully verifiable.

```
class CounterObject {
    var c :int;

    method count() returns (res: int)
        requires valid
        ensures valid
        ensures res == old(getCount())
        ensures getCount() == old(getCount()) + 1
    {
        unfold valid;
        res := c;
        c := c + 1;
        fold valid;
    }

    function getCount() : int
        requires valid
    {
        unfolding valid in c
    }

    predicate valid { acc(c) }
}

delegate Counter() returns (int) {
    function getCount(): int
    predicate valid
}

class Client {
    method createCounter() returns (res: Counter)
        ensures pre(res) =| res.valid
        ensures post(res, i) |= res.valid &&
                i == old(res.getCount()) &&
                res.getCount() == old(res.getCount()) + 1
```

```
        ensures res.valid && res.getCount() == 0
    {
        var c := new CounterObject
        c.c := 0;
        fold c.valid
        res := newdel Counter(c.count)
    }
    method client() {
        call c := createCounter()
        call i := c()
        call j := c()

        assert i == 0 && j == 1
    }
}
```

6.1.3 Higher order functions

This example was fully verifiable and it is a proof of concept for higher order functions. Appendix 9.1 shows another fully verifiable example for higher order functions with return values.

```
delegate Command()
class Client {
    var value: int

    method Conditional(b: bool, c: Command)
        requires b ==> pred(c)
        ensures b ==> postd(c)
    {
        if (b) {
            call c()
        }
    }
    method set3()
        requires acc(value)
        ensures acc(value) && value == 3
    {
        value := 3
    }
    method set5()
        requires acc(value)
        ensures acc(value) && value == 5
    {
        value := 5
    }

    method client()
        requires acc(value)
```

```
    {
        value := 0
        var c1 := newdel Command(set3)
        var c2 := newdel Command(set5)

        call Conditional(true, c1)
        assert value == 3

        call Conditional(false, c2)
        assert value == 3

        call Conditional(true, c2)
        assert value == 5
    }
}
```

6.1.4 Entailment proofs

The test file `entailment.chalice` contains interesting proofs with entailments that were verifiable.

6.2 Command Pattern

This example is about modifying a Document which is equivalent to a Cell. The client creates delegates to methods which can work on the Document. The example shows the basic operations in action.

```
delegate Command(Document) {
    predicate valid

    function state() : int
}
delegate Command2(Document)

class Document {
    var val: int

    method Set(i: int)
        requires valid
        ensures valid && value() == i
    {
        unfold valid
        val := i
        fold valid
    }

    predicate valid { acc(val) }
```

```
    function value(): int
        requires valid
    {
        unfolding valid in val
    }
}

class AddObject {
    var summand : int;

    method Add(d:Document)
        requires valid && d != null && d.valid
        ensures valid && d.valid &&
            old(d.value()) + state() == d.value()
    {
        call d.Set(d.value() + state())
    }

    predicate valid { acc(summand) }

    function state() : int
        requires valid
    {
        unfolding valid in summand
    }

}

class Program {
    method InverseMethod(d:Document)
      requires d != null && d.valid && d.value() != 0
      ensures d.valid && d.value() == 100 / old(d.value())
    {
        call d.Set(100 / d.value())
    }

    method createAdd(value:int) returns (c:Command)
        ensures c != null && c.valid && c.state() == value
        ensures pre(c, d) =| c.valid && d!=null && d.valid
        ensures post(c, d) |= c.valid && d.valid &&
            old(d.value()) + c.state() == d.value()
    {
        var o := new AddObject;
        o.summand := value;
        fold o.valid
        c := newdel Command(o.Add)
    }

    method client() {
```

```
        var inverse : Command2;
        var add : Command;

        var d := new Document;
        fold d.valid
        call d.Set(100)

        inverse := newdel Command2(this.InverseMethod);

        call inverse(d);

        call add := this.createAdd(1);
        call add(d);
        assert d.value() == 2;

        call add := this.createAdd(10);
        call add(d);
        assert d.value() == 12;
    }
}
```

This program is almost fully verifiable with the current implementation. The only part where it fails due to a strange bug of which I could not find the reason is the last postcondition of method `createAdd`. While such entailment proofs with old expressions and ghost functions and predicates usually work, this particular one did not. The entailment proof looks like this:

```
stronger: acc(c.delegate$target$, wildcard) && (c.delegate
   $target$ != null) && acc(c.delegate$targettype$,
   wildcard) && (true && (acc(ArbCommandObjectvalid$(c.
   delegate$target$), write) && acc(Documentvalid$(d$_2),
   write) && (Documentvalue$(d$_2) == r$_1) && Post(
   ArbCommandObjectff$(c.delegate$target$), old$$$$0, r
   $_1)))

weaker: acc(c.delegate$target$, wildcard) && (c.delegate$
   target$ != null) && acc(c.delegate$targettype$,
   wildcard) && (acc(ArbCommandObjectvalid$(c.delegate$
   target$), write) && acc(Documentvalid$(d$_2), write) &&
    (Documentvalue$(d$_2) == r$_1) && Post(
   ArbCommandObjectff$(c.delegate$target$), old$$$$0, r
   $_1))
```

We can easily verify this proof, since the only difference is a `true` expression which is added as a conjunct. The exception is thrown from the inner workings of Silicon which I do not understand.
When I in-line the method `createAdd` twice and remove the actual method, the verifier can verify the program.

6.3 Arbitrary Command Pattern

In this example the client can construct his command by himself by giving a function that specifies how the document's value should be modified.

```
delegate Command(Document) returns (int) {
    predicate valid

    function ff() : Func
}
delegate Func(int) returns (int)


class Document {
    var val: int

    method Set(i: int)
        requires valid
        ensures valid && value() == i
    {
        unfold valid
        val := i
        fold valid
    }

    predicate valid { acc(val) }

    function value(): int
        requires valid
    {
        unfolding valid in val
    }
}

class ArbCommandObject {
    var f : Func;

    method ArbCommand(d:Document) returns (r: int)
        requires valid && d != null && d.valid &&
            pred(ff(), d.value())
        ensures valid && d.valid && d.value() == r &&
            postd(ff(), old(d.value()), r)
    {
        call r := f(d.value())
        call d.Set(r)
    }

    predicate valid { acc(f) }

    function ff(): Func
```

```
        requires valid
    { unfolding valid in f }
}


class Program {

    method createArbCommand(f:Func) returns (c:Command)
        ensures c.valid
        ensures pre(c, d) =| c.valid && d != null &&
            d.valid && pred(c.ff(), d.value())
        ensures post(c, d, r) |= c.valid && d.valid &&
            d.value() == r &&
            postd(c.ff(), old(d.value()), r)
    {
        var o := new ArbCommandObject;
        o.f := f;
        fold o.valid
        c := newdel Command(o.ArbCommand);
    }

    method Arb(i: int) returns (j: int) {
        j := (i + 3) * 2
    }
    method client() {
        var d := new Document;
        fold d.valid
        call d.Set(10)

        var fnc := newdel Func(Arb)

        call ac := createArbCommand(fnc)

        call res := ac(d)

        assert d.value() == 26
    }
}
```

Here the exact same error appeared for method `createArbCommand`. Verifying every method by hand yields that this program is correct.

Chapter 7

Further Features

There are several features that are not yet supported in the current design but the addition of these features would contribute to a full tool-set for closures in Chalice.

7.1 Stronger Entailment Proofs

The current system only supports very weak proofs of entailments with old expressions. A stronger entailment proof would improve the system a lot and it is definitively possible as discussed in section 5.2.5.

7.2 Functional Delegates

Functional Delegates are delegate objects that point to functions instead of methods. They are useful because they can appear in expressions, especially also in pre- and postconditions. But this added functionality also makes it hard to verify them.

Functional delegates were implemented in Chalice but they are not supported further than that - the translation into Silver will throw errors.

An example of the current implementation in Chalice:

```
functional delegate Func(int, int): int {
    predicate valid
    function getValue(): int
}

class Client {
    method Test() returns (f: Func)
        ensures f.valid && f(3, 4) == 7
        ensures post(f, a, b) |= a * f.getValue() + b
    {
```

```
        var c := new Cell
        c.value := 1
        fold c.valid

        f := newdel Func(c.getLinear)
    }
}
```

This example shows a functional delegate pointing to the function `getLinear` of class `Cell`. For this to work the delegate type has to be declared as `functional`.

7.3 Higher order Functions

The support for higher order functions is very limited in the current system. As shown in appendix 9.2 they do not support delegates executed sequentially and there is no obvious simple solution to this issue. Also the implementation of the concept (see section 5.2.6) has the further limitation that it does not work with `old` expressions. Further work to extend the support for higher order functions can definitively be done.

7.4 Anonymous Delegates

Anonymous delegates are delegates that do not point to a specific method but still have executable code associated with them. This code is said to be contained in an anonymous method, which is why the delegate is called anonymous.
I believe it is possible to add anonymous delegates to the system without modifying the back-end of the Viper infrastructure. This can be done by creating an anonymous class each anonymous delegate definition during the translation into Silver. This is only speculation and further investigations are required regarding the soundness of such a system. A detailed example can be found in Appendix 9.4.

Chapter 8

Conclusions

This thesis has shown a feasible approach to introduce closures as first-class-citizen values to Chalice. The concepts developed in section 3 provide a good basis for delegates in Chalice where they work well with fractional permissions. The methodology provides solutions for several well known issues of closures, including dealing with hidden state (Section 3.5), the circularity problem (Section 3.6) and higher order functions (Section 3.7).

The biggest issue with this thesis is the lack of real-world examples that could be verified. This is partially due to that the concept is not strong enough for all examples or that the implementation did not achieve all the features of the concept. Another major point was that I simply did not have the needed time to do these tests due to bad time management.

Section 7 has some interesting examples how the system could be extended to support functional delegates, delegates that point to functions instead of methods, and anonymous closures, delegates that do not point to any method and can capture local variables.

The implementation is documented by many comments in the source code wherever I modified the original code of the viper project. It can be accessed form the `Bitbucket` repositories `yChaliceClosures`, `chalice2silClosures`, `silClosures` and `siliconClosures`. All examples of this thesis can be found in the directory

```
chalice2silclosures/src/test/resources/chaliceSuite/
   closures
```

At this point I want to thank everyone who helped me, this is Ioannis T. Kassios who supervised and supported me even in times where I did not make much progress, Malte Schwerhoff who gave me a detailed introduction into Silicon and answered many questions about it and Prof. Müller who allowed this thesis.

Chapter 9

Appendix

9.1 Higher Order Functions

This example shows how higher order methods involving arguments and return values work in chalice. `condF` is a higher order method that takes a boolean and executes the `Func` delegate when this boolean is true. The comments in the code describe the state at various times. This example was verifiable with the implemented verifier.

```
delegate Func(int) returns (int)
class HigherOrderMethodTest {
    var value: int

    method condF(b: bool, f: Func, i: int) returns (j:int)
        requires b ==> pred(f, i)
        ensures b ==> postd(f, i, j)
    {
        // assuming b == true we have here pre(f, i) in
        // the state. Note that both f and i are stored
        // in evaluated form.

        if (b) {
            // this call succeeds because it can get
            // pre(f, i) from the state and all arguments
            // match with the stored arguments.
            call j := f(i)
            // here post(f, i, j) is added to the state
            // (because no fitting entailment is found)
            // - again the arguments are evaluated.
        }
        // we can exhale post(f, i, j) here because it's
        // in the state with the exact same arguments.
    }
```

```
    method id(i: int) returns (j: int)
        requires acc(value, 50)
            // just to make it more interesting
        ensures j == i
    { j := i }

    method test()
        requires acc(value)
    {
        var f := newdel Func(id)

        call a := f(3)
        // here we have half access to value left

        // this consumes access permissions even though it
        // says nothing of this in its specification.
        call b := condF(true, f, 4)

        // this does not consume access permissions
        call c := condF(false, f, 5)

        assert a == 3 && b == 4 // holds
    }
}
```

Reminder *`pre` and `post` expressions embedded in entailments can only take formal names as arguments (except the delegate object). Free `pre` and `post` expressions can also take expressions.*

Free `pre` and `post` expressions can also take expressions instead of formal arguments. Free `post` expressions can still only take local variables as return values (or even only return variables of the method).

9.2 Limits of Higher Order Functions

This example shows the limitations of the implemented solution for higher order functions. The `composition` method takes two `Func` delegates and composes them to a `FuncComp` delegate.

```
delegate Func(int) returns (int)
delegate FuncComp(int) returns (int, int) {
    function f0(): Func
    function f1(): Func
    predicate valid
}

class CompositionClass {
    var f0: Func
```

```
    var f1: Func

    method composition(i: int) returns (t: int, j: int)
        requires pred(f0, i)
        requires post(f0, i, t) |= pred(f1, t)
        ensures postd(f1, t, j)
    {
        call t := f0(i)
        call j := f1(t)
    }

    predicate valid { acc(f0), acc(f1) }
}

class client {
    method createComposition(f0: Func, f1: Func) returns (
   res: FuncComp)
        ensures pre(res, i) |= ( pred(res.f0(), i) &&
            post(res.f0(), i, t) |= pred(res.f1(), t) )
        ensures post(res, i, t, j) |= postd(res.f1(), t, j
   )
    {
        var compObj := new CompositionClass
        compObj.f0 := f0
        compObj.f1 := f1
        fold compObj.valid
        res := newdel FuncComp(compObj.composition)
    }

    method func0(i: int) returns (j: int)
        ensures j == 2 * i
    {
        j := 2 * i
    }

    method func1(i: int) returns (j: int)
        ensures j == i + 5
    {
        j := i + 5
    }

    predicate valid { true }

    method client() {
        fold valid
        var f0 := newdel Func(this.func0)
        fold valid
        var f1 := newdel Func(this.func1)
```

```
        call f := createComposition(f0, f1)

        call t, j := f(10)

        assert j == 25
    }
}
```

This construction with the double output for `FuncComp` is necessary to serve the client with the right `t` for the postcondition `post(f1, t, j)`.

The expression `post(f0, i, t) |= pre(f1, t)` is quite interesting. It works perfectly fine if we define the `post` expression to be bound by the entailment and the `pre` expression to be free. The method `composition` will try to inhale `pre(f1)` after the first delegate call. This will succeed in the same way the `pre(f0)` expression was inhaled at the start of it. In the client the proof will try to exhale the `pre(f1)` out of the empty state filled with the `post(f0)`. It will find the right pre-entailment for `f0` to exhale which works if the postcondition of `f0` really implies the precondition of `f1`.

This example can still not be verified because of a subtle detail: The value of `t` returned by `composition` is and can not be specified by the postcondition. We would need to also return `post(f0, i, t)` to specify `t` which is not possible because we need it to get `pre(f1, t)`. This is a serious constraint on the system.

9.3 Renaming in Old Expressions

This example shows how throughout the renaming of fields, functions and predicates must be to make the system complete.

```
inhale post(c, myCell, i) |= myCell.valid &&
    i == old(myCell.getValue())

exhale post(c, myCell, i) |= myCell.valid &&
    unfolding myCell.valid in i >= old(myCell.value)
```

It does not suffice to rename `myCell.getValue()` to `myCell.old_getValue()` - the body of `old_getValue` has to be changed to `myCell.old_value` too. Otherwise the proofer could not associate `old(myCell.getValue())` with `unfolding myCell.valid in old(myCell.value)`. When there are further references to other functions in `getValue` this renaming can cause the program size to double.

9.4 State capturing

This example shows a concept of how anonymous methods could be implemented in chalice and how they can be transformed into standard Chalice code. The assumption is that an anonymous method also provides one (or more) ghost functions to access its captured state and also a ghost predicate that specifies the footprint of the closure. This assumption is based on the paper [2].

```
method foo() {
    var a := new Cell
    fold a.valid
    call a.setValue(1)

    var closure := delegate()
        requires valid
        ensures valid &&
            a.getValue() == old(a.getValue()) + 1
    {
        call a.inc()
    } with ghosts {
        function getState(): int
            { a.getValue() }
        predicate valid { acc(a) && a.valid }
    }

    closure()
    closure()

    assert a.getValue() == 3
}
```

This closure can be transformed automatically into the following normal Chalice code:

```
delegate AnonDel() {
    function getState(): int
    predicate valid
}

class AnonType {
    var a: Cell
    function getState(): int
        requires valid
    { a.getValue() }

    predicate valid { acc(a) && a.valid }

    method anonMeth()
        requires valid
```

```
        ensures valid &&
            a.getValue() == old(a.getValue()) + 1
    {
        unfold valid
        call a.inc()
        fold valid
    }
}

method foo() {
    var a := new Cell
    fold a.valid
    call a.setValue(1)

    var anonObj := new AnonType
    anonObj.a := a
    fold anonObj.valid
    var closure := newdel AnonDel(anonObj.anonMeth)

    closure()
    closure()

    assert a.getValue() == 3
}
```

In this example we do not deal with the possibility that `a` might be reassigned to another `Cell` inside the anonymous closure. The transformation can be adjusted to work even under such circumstances by boxing `a` and replacing every occurrence of `a` with the boxed expression. This only needs to be done with local variables since the `this` object and other arguments are immutable.

Bibliography

[1] K. Rustan M. Leino, Peter Müller, Jan Smans *Verification of Concurrent Programs with Chalice*. ETH Zürich, Microsoft Research Redmond, KU Leuven, 2009.

[2] Ioannis T. Kassios, Peter Müller *Modular Specification and Verification of Delegation with SMT Solvers*. ETH Zurich, 2013.

[3] Uri Juhasz, Ioannis T. Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, Alexander J. Summers *Viper: A Verification Infrastructure for Permission-Based Reasoning*. ETH Zurich, 2014.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Closure verification in an automated fractional permission setting

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Meier	Fabian

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
7. October 2014	F. Meier

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.