Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis

Verification of Finite Blocking in Chalice

Robert Meier

Supervisor:
Prof. Dr. Peter Müller

Chair of Programming Methodology
Department of Computer Science
ETH Zürich

September 2, 2015

Abstract

In this report we describe the implementation of obligations in Chalice. This verification technique [BM15] is used to prove finite blocking, that is, every operation that potentially blocks a thread is eventually going to terminate. A previous project [Kla14] introduced several Silver extensions to enable the encoding of obligations in Chalice. In light of recent changes in the theoretical background of obligations, we adapted this encoding and were able to successfully reduce the number of necessary Silver extensions while simultaneously making the analysis more accurate and user friendly.

Contents

1 Introduction **4**
1.1 Related Work 5

2 Background **6**
2.1 Viper 6
2.1.1 Silver 6
2.1.2 Silicon 13
2.2 Chalice 13
2.2.1 Channels 13
2.2.2 Forking and Joining 14

3 Obligations **16**
3.1 Lifetime 17
3.2 Types of Obligations 17
3.3 Leak Checks 18
3.3.1 Method Calls 18
3.3.2 Loops 18
3.3.3 Rule of Consequence 19
3.4 Unbounded Obligations 20
3.5 Creation of Obligations 21
3.6 Consumption of Obligations 23

4 Silver Extensions **25**
4.1 For All References 25
4.2 Behaviour of Perm Expression 27
4.3 Removed Extensions 28

5 Encoding of Obligations in Silver **29**
5.1 Encoding Obligations 29
5.2 Transfer of Obligations 30
5.2.1 Inhale 31
5.2.2 Exhale 32
5.2.3 Termination Obligation 39
5.3 Leak Check 40
5.4 Lifetime 40
5.4.1 Initialising the Lifetime 41
5.5 Methods 44
5.6 Method Call 46
5.7 Fork 47
5.8 Loops 47
5.9 Send and Receive 49
5.10 Acquire and Release 50

6 Evaluation **51**
6.1 Deviation From Original Scheme 51
6.2 Comparison to Earlier Version 51
6.3 Error Messages 56

6.4 Cancellation 58
6.5 Existing Test Suite 61

7 Conclusion **62**
7.1 Future Work 62

1 Introduction

Finite blocking is a property denoting that no thread is ever blocked indefinitely. To show this in a system with fair scheduling, one has to prove that every operation that can potentially block a thread is eventually going to terminate. In this report we present a verification technique [BM15] that uses so-called obligations to achieve this goal. An obligation forces a thread to perform some action that will unblock another thread. An intuitive example is an obligation to release a lock. A thread obtains this obligation as soon as it performs a locking operation and it can get rid of it by releasing the lock again. If a thread fails to satisfy an obligation, the verification will fail as there exists the possibility that some thread might not make progress on account of not being unblocked.

While there exist solutions that are able to prove deadlock freedom for *terminating* programs ([WTE05]), these analyses are often not equipped to cope with real-world applications that contain potentially non-terminating threads. One can think of a background process that is never shut down and keeps exclusive access to some resource. If another thread tries to access this shared resource, it cannot succeed since the resource is never released. This scenario is not comprised of a deadlocking situation, which could easily be statically detected by existing tools, but nevertheless shows a thread that is unable to make progress. The proposed verification technique provides a way to modularly prove the absence of such errors.

Relying on modular verification has the advantage that every method can be analysed in isolation without relying on the implementation of the context it is used in. On one hand this simplifies the maintenance as there is no need to re-verify the complete system after some parts have been adapted. On the other hand, such techniques are especially appealing in systems where the interacting components are not be known or their implementation is unavailable.

In this report we present an encoding of obligations in the Viper verification infrastructure [JKM+14]. We will show changes made to the Chalice language that allow obligations to be incorporated in this front-end language. To fully support the concept, we introduce extensions to the Silver language and Silicon verifier. To evaluate our project we will show the verification of larger examples and compare our encoding to previous work [Kla14] that encoded obligation under differing theoretical assumptions.

Outline. The remainder of this report is structured as follows. Assuming a minimal background in software verification techniques, we will summarise relevant background information for this project in Section 2 and introduce obligations in Section 3. To make the implementation of obligations possible, we extended the Silver language as described in Section 4. The final encoding of obligations is then described in Section 5 and evaluated in Section 6. We summarise our project in Section 7 and discuss possible next steps.

1.1 Related Work

A previous master's thesis [Kla14] thesis describes an implementation of obligations in Chalice. To ultimately encode obligations in Silver, it was necessary to enhance the language in several ways. It is important to note, that the theory behind obligations changed substantially over time which made it apparent that not all extensions were needed any more. This current thesis re-visits the same topic in light of this new theoretical background.

The Chalice language and verification is described in great detail in [LMS09] and [LMS10]. An important part is the handling of the so-called waitlevels. This concept can be used to statically detect deadlocking programs which are ignored in our current project.

Static deadlock detection is, for example, described in [WTE05] where the analysis of Java libraries is presented. Compared to this analysis we do not rely on terminating threads however.

2 Background

In this section we present relevant background information for this project. First we will describe the Viper verification infrastructure [JKM+14] where we will focus mostly on the Silver language. At the end of this section, we will introduce the Silicon verifier and present Chalice. We will then use this language throughout this report to express input programs that are translated to Silver and verified by Silicon.

2.1 Viper

Figure 1 shows an overview of the Viper verification infrastructure [JKM+14]. At the heart lies Silver, an intermediate language designed specifically for program verification. In Section 2.1.1 we will introduce Silver and its most important concepts to provide a basic understanding of how it can be used to verify programs.

The two back-ends, Silicon and Carbon, can be used to verify programs written in Silver. Both of them rely ultimately on the Z3 solver [DMB08], however they use different verification techniques: Silicon uses symbolic execution, while Carbon is based on verification condition generation. In this report we will focus solely on Silicon and introduce the basic concepts in Section 2.1.2.

Programs can either be written directly in Silver or one can use different front-ends to translate Java, Scala or Chalice code. In this report we will focus on the Chalice2Silver tool that compiles Chalice programs to Silver. Relevant information about Chalice can be found in Section 2.2 and in Section 5 we will further describe the way we translate Chalice to Silver.

2.1.1 Silver

Silver is a permission based verification language. It is intended as an intermediate language and was designed to be as simple as possible while still providing enough functionality to encode numerous source level programming constructs.

Throughout this section we will use the term **source program** to denote a program that is translated *to* Silver by a front-end. It is important to keep in mind that Silver is intended as an intermediate language and we will often switch our point of view when we describe its basic concepts.

Types. The built-in primitive types are `Int, Bool, Perm` and `Ref`. The `Perm` type denotes permission amounts and we will describe it in more detail in the next paragraph. `Ref` types denote references in general. In particular there is no information about the type of object the reference points to.

Figure 2 shows the translation of the two classes `A` and `B`. Silver has no notion of class, instead the `field` identifier denotes a memory location that can be

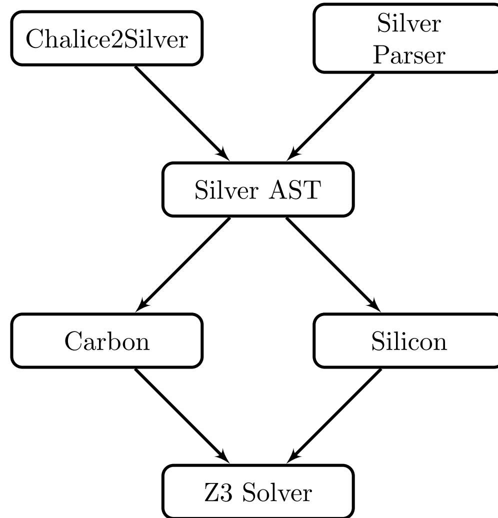

Figure 1: (Simplified) Viper Architecture Overview [JKM+14]

```
class A {
  var x : int;
  var y : int;
}

class B {
  var f : bool;
  var g : bool;
}
```

(a) Two Chalice Classes

```
//class A:
field Ax : Int

field Ay : Int

//class B:
field Bf : Bool

field Bg : Bool
```

(b) Silver Translation

Figure 2: Translation of Chalice Fields

```
field x : Int
field y : Int

method f(r : Ref)
  requires r != null && acc(r.x, write) && acc(r.y, write/2)
{
    r.x := r.y
}

method g(r: Ref)
  requires r != null && acc(r.x, write) && acc(r.y, write/2)
  ensures acc(r.x, write) && acc(r.y, write/2)
{
    r.x := r.y
}

method caller(r : Ref)
  requires r != null && acc(r.x, write) && acc(r.y, write/2)
{
    g(r)
    f(r)

    //f(r) ; f(r) //error!
}
```

Figure 3: Access Permissions

accessed by using a reference. Or in other words, the definition

```
field Ax :  Int
```

announces that there exists a field `Ax` of type `Int`, but there is no knowledge about the class the field belongs to. In this example, every Silver reference `r` could be used, as far as Silver is concerned, to access every field: `r.f`, `r.g`, `r.x`, `r.y`. It is up to the front-end to enforce type-safety and correctly encode different types.

Access Permissions. Every access to a memory location needs a positive amount of permissions. To be able to write to a location we need *full permission* while reading requires only a *fraction* of the full permission. The creation of an object provides us with full permission to all fields and whenever we call a method we may give some of our permission amount away. As mentioned above, variables of type `Perm` store permission amounts.

For example, assume we have p permissions to location `r.x`. If we call a method that requires q permissions to `r.f` we are left with $p - q$ permissions, because we gave q away to the method we called.

We can have at most one full permission to a location and the amount of permission cannot be negative. In turn this means that whenever we hold *any* permission to a location, we know that there cannot exist any other thread that has full permission. This is the central property that enables the reasoning about values in a concurrent setting because we know that whenever we can read a location, no other thread can alter the values concurrently.

Consider the example in Figure 3. Line 5 ensures that method `f` gets full access

to `r.x` (the keyword `write` denotes full access) and read access to `r.y`. This precondition shows the fractional nature of permissions: to get read access to `r.y` we take "half of a full permission".

Method `g` is almost the same as `h`, however there is one key difference: the postcondition of `g` makes sure that the caller has access to `r.x` and `r.y`, respectively, after the method has terminated. In other words, the method *returns* the permission to the caller while `f` consumes it.

This means that we can call method `f` at most once for any given argument, since it consumes one full permission to `r.x`. On the other hand, method `g` can be called any number of times because it returns all the permissions it gets.

Pointers to more detailed descriptions of permission based reasoning can be found in Section 1.1.

Inhale, Exhale. Just as many first-order verification languages make heavy use of `assume` and **assert** statements, Silver provides the permission-aware `inhale` and `exhale` counterparts. The statement

`inhale` A

behaves like an `assume` statement if A is pure, i.e. if A does not contain any accessibility predicates. Similarly, the `exhale` statement is essentially an `assert` statement if its argument is pure.

If the arguments of the statements contain accessibility predicates, i.e. are not pure, the permission amounts are added or subtracted from the current amount. For example the statement

`inhale acc(r.f, write)`

adds a full permission to the location `r.f`. Therefore, after this statement was executed, we have permission to access this memory location. Analogously, the exhale statement checks that we have enough permissions and then removes it them from our current amount.

Consider again the example in Figure 3. On line 20, we call method `g`. This call gets automatically translated to the following two statements:

```
exhale r != null && acc(r.x, write) && acc(r.y, write/2)
inhale acc(r.x, write) && acc(r.y, write/2)
```

This corresponds to the typical translation of a method call: the assertion (exhale) of the precondition and the assumption (inhale) of the postcondition.

When we look at line 23 of Figure 3 it becomes clearer why method `f` can only be called at most once: the exhale statement of the precondition removes one full permission for `r.x` and "half of a full permission" for `r.y`. Since the postcondition does not exits, i.e. is equal to true, we do not get any permission back. Therefore the second call cannot be verified since the exhale of the precondition cannot remove enough permissions from the current state.

```
field f : Int

method headache(r : Ref)
    requires r != null
    requires (perm(r.f) == none) ==> acc(r.f, write)
{
    //have full control over r.f
}

method caller(r : Ref)
    requires r != null
    requires acc(r.f, write)
{
    fork headache(r)

    //have full control over r.f
}
```

Figure 4: Unsoundness Due to `perm` Expression

Current Permission. If we are interested in the current permission amount for a reference we can use the `perm` statement:

$$\texttt{perm}(x.f)$$

This deceptively simple statement must be used with caution as the example in Figure 4 shows. The precondition of method `headache` gives the method full access to the field $r.f$: in the beginning, the method has no access to $r.f$ (represented by the keyword `none`) and thus satisfies the antecedent of the implication which leads to the permissions being inhaled. On the side of the caller however the situation is completely different as the antecedent will evaluate to false. Consequently the permission will not be exhaled and the caller keeps full control over $r.f$. This behaviour is *unsound* because we have effectively created permission out of thin air and two threads have full access to the same reference.

To avoid these kinds of problems, `perm` expressions were originally forbidden in the specifications of methods. However, in the context of this project we had to lift this ban and make the front-ends responsible for producing sound programs.

Predicates. The main purpose of predicates is to provide a way to specify access permissions to an unbounded number of heap locations. However, in this report we focus on another property, namely that it is possible to store more than a full write permission to a predicate. Consider the example shown in Figure 5. When executing the statement on line 11 we use the predicate to store a total of `(5*write)` permission amount in it. This means that when we exhale one full write permission, we still have `(4*write)` permission amount left.

Since the verifier can keep track of aliasing information, we basically get a way to perform bookkeeping with references. Say we are interested in an integer value p that is assigned to every reference. By using multiples of full permissions, we can use predicates to represent this integer value p. Figure 6 shows an example

```
predicate p(x : Ref) {
    true
}

method f(r : Ref, s : Ref)
    requires s != null
{

    // ...

    inhale acc(p(r), 5*write)

    exhale acc(p(r), write)
    //still have 4*full

    exhale acc(p(s), 2*write)

}
```

Figure 5: Silver Predicates

```
predicate p(x : Ref) {
    true
}

method f(r : Ref, s : Ref)
    requires s != null
{
    assume r == s

    inhale acc(p(r), 5*write)

    exhale acc(p(s), 2*write)

    assert perm(p(r)) == perm(p(s)) && perm(p(r)) == 3*write
}
```

Figure 6: Silver Predicates to Do Calculations

```
function m(a : Int, b : Int): Int {
    a > b ? b : a
}

function p(x : Int) : Bool
    requires x >= 0

method f() {

    var a : Int := 4
    var b : Int := 5

    exhale m(a,b) == m(b,a)

    inhale p(2) == true
    inhale p(3) == true
    inhale p(5) == true
    inhale p(7) == true
    inhale p(9) == false

    exhale p(2) == p(1+6)
    exhale p(11)    //Error!
}
```

Figure 7: Silver Functions

of this technique. The statement on line 10 increases the value p of reference r by 5, while the statement on line 12 decreases it again by 2. Since r and s are aliases (line 8), the assertion on line 14 can be verified.

Functions. A function definition consists of a name, parameter list, return type and optional pre- and post-conditions. Figure 7 shows how Silver functions can be used. The first three lines define the function `m` that calculates the minimum of its parameters by using the ternary `?:` operator.

The second function differs from the first one in that it is underspecified: the precondition requires that the argument is not negative, but the body is not defined. This allows us to axiomatise the function ourselves throughout the program, as shown on lines 15-19. Since the function is only partially defined, the verifier cannot prove the assertion on line 22.

Functions provide a way define recursive specifications, for our purposes however, we do not need these properties and use them simply as a means to keep track of values assigned to references. As with predicates, the verifier automatically makes use of aliasing information which greatly facilitates our bookkeeping.

Paired Assertions. If an assertion needs to behave differently depending on whether it is being inhaled, or exhaled, we can use paired assertions. This special type of assertions is of the following form

$$\varphi := [\alpha, \beta]$$

when φ appears in an inhale statement, it is replaced by α and if it is exhaled, it is replaced by β.

A natural example for this is induction:

$$[\forall n \geq 0 : P(n), P(0) \land \forall n \geq 0 : P(n) \rightarrow P(n+1)]$$

Whenever this assertion is exhaled we need to provide the proof by induction and when we inhale it we can assume that $\forall n \geq 0 : P(n)$ holds.

2.1.2 Silicon

Silicon is a verifier back-end for Silver that is based on an earlier verifier for Chalice [Sch11]. Compared to Carbon, the second verifier for the Silver language that generates a Boogie [BCD+06] encoding to verify Silver programs, this solver is based on symbolic execution. Silicon itself does not perform any theorem proving but instead passes all generated queries to the Z3 solver [DMB08].

For our project we need not worry about the exact details of the Silicon verification process. The most important thing is the representation of the heap. To represent the heap during the verification, the notion of a *heap chunk* is introduced. A heap chunk is a mapping from a field reference to a symbolic value and the amount of permission that is currently held for that field:

$$r.f \rightarrow t\#p$$

where

$$\begin{aligned} r &:= \text{object reference (symbolic)} \\ f &:= \text{filed identifier} \\ t &:= \text{field value (symbolic)} \\ p &:= \text{amount of permission (symbolic)} \end{aligned}$$

The heap is then essentially a list of these heap chunks. This representation is important to keep in mind as we present the extensions to the Silver language in Section 4.

2.2 Chalice

In this section we will briefly describe Chalice [LMS09]. This permission based language was created to study the verification of concurrent programs. We will only cover selected features that are directly relevant for our project. This means we will present concepts that introduce *blocking* operations to the language. For example, when waiting for a message on a channel, the thread gets blocked until another thread unblocks it by performing a send operation. Note that the examples shown in this section are written in Chalice *without* any extensions introduced later in this report.

2.2.1 Channels

Channels offer a way for threads to communicate with each other via message passing. Each channel defines the kind of message that can be transmitted. For

```
channel C(x:int) where x >= 5;

class A {

    method f(c : C) returns (res:int)
        requires c != null
        ensures res >= 4
    {
        receive res := c
    }

    method g(c : C)
        requires c != null
    {
        var data : int := 2
        send c(data) //error: invariant not satisfied
    }
}
```

Figure 8: Chalice: Sending and Receiving Messages

example:

$$\texttt{channel}\ C(n : \texttt{int})\ \underbrace{\texttt{where}\ n > 0}_{\text{channel invariant}};$$

Defines a channel type that can be used to send positive integers. The **channel invariant** makes it possible to reason about the message that is transmitted over the channel: on the sending side, the invariant has to be verified and on the receiving end, it can be assumed.

Consider the example shown in Figure 8. Method `f` receives a message over channel `c`. Since the channel invariant can be assumed when receiving a message, the postcondition `res >= 4` can be verified. On the sending side, in method `g`, the verification will fail since the channel invariant is not satisfied.

2.2.2 Forking and Joining

Unlike Silver, Chalice has a built in notion of executing methods asynchronously. The fork statement executes a method in another thread and returns a **token** that can be used to join the thread later. Consider the example shown in Figure 9. On line 11 we fork method `f` and join it again on line 14 by using the token `t`. As soon as we join, we inhale the postcondition of the method which allows us to verify the assertion on line 16.

Note that a token can be joined only once. This is critical since the postcondition is inhaled at the place we join. If it were possible to join a thread multiple times and the postcondition of that method contained access permissions, we would have a way of duplicating these permissions, which would be unsound.

```
class A {

    method f(x : int) returns (n:int)
        ensures n == x + 1
    {
        n := x + 1
    }

    method main() {

        fork t := f(5)

        var res : int
        join res := t

        assert res == 6
    }
}
```

Figure 9: Forking and Joining in Chalice

3 Obligations

In this section we will describe the concept of obligations as introduced by Boström and Müller [BM15]. This verification technique is used to modularly prove finite blocking for non-terminating programs. The term 'finite blocking' refers to all threads being able to make progress: whenever an operation is executed that can potentially block the execution of a thread, we have a guarantee that it is going to terminate, thereby ensuring that the thread is not blocked indefinitely. Compared to terminating programs, it does not suffice to simply show the deadlock freedom to prove finite blocking. Imagine a background process that continuously updates a log file and in doing so keeps exclusive access to this resource. A thread that tries to exclusively access this resource may never succeed as the logging thread will potentially not terminate and hence never release the resource. This is not a deadlocking situation and therefore not easily detectable by conventional analyses.

The proposed technique is modular which means that we can verify each method independently from the implementation of other methods. One key advantage of modular approaches is that the verification of a particular method does not have to be repeated if the implementation of other parts of the software is changed. Additionally the complexity of the analysis is greatly reduced since we never try to analyse the system as a whole.

The central idea behind obligations is that every operation that blocks a thread can only be executed if there is a guarantee that it will eventually terminate, i.e. that the thread is not blocked indefinitely. For example we can only receive a message over a channel if we know that there exists another thread that is going to send a message over this channel at some point. In general, every operation that causes *any* thread to block, generates a corresponding obligation to unblock the thread.

Consider the example in Figure 10, that shows a simplified Chalice program. The `locker` method first acquires the lock on its argument. Since locking an object can cause other threads to block, the statement generates a corresponding

```
1  class Lock {
2
3      method unlock(l : Lock)
4        requires mustRelease(l)
5      {
6          release l;
7      }
8
9      method locker(l : Lock)
10         //some preconditions have been omitted
11     {
12         acquire l
13         call unlock(l)
14     }
15 }
```

Figure 10: Basic Release Obligation

obligation to ensure that the lock is eventually released. This means that after we successfully locked the object, we inhale a so-called *release obligation*.

Let us now take a closer look at the precondition of method `unlock`. At the call site we exhale the precondition which causes the caller to give away its release obligation. In the callee however, we get a release obligation as we inhale the precondition. We have therefore essentially transferred the obligation from the caller to the callee. The lock is then finally released on line 6, which satisfies the obligation.

In summary we see that any thread that is blocked by the locking of the object on line 12, will eventually be unblocked.

Side note: the acquire operation on line 12 could cause the current thread to be blocked. However since the locking of an object always generates a corresponding release obligation, we know that the statement will eventually terminate.

3.1 Lifetime

To ensure that obligations are satisfied eventually, we associate a termination measure with each obligation. This measure is an expression that evaluates to a value in a well-founded set [BM15]. As soon as an obligation is passed to another context, say to another method, or between loop iterations, this measure has to strictly decrease. This rule guarantees that the measure will eventually reach $\perp$, at which point it must be satisfied.

It is important to note however that this lifetime cannot be mapped directly to a notion of "real-time". For example, the obligation:

$$\texttt{mustSend}(\texttt{c}, 1, 5)$$

denotes an obligation to send `one` message over `channel c` with the `measure 5`. This does, for example, *not* mean that the message will have been sent after the next 5 statements have completed. The measure simply ensures that the obligation gets satisfied **eventually** since it cannot be postponed indefinitely. We will cover lifetime expressions and their initialisation in more detail when we describe the encoding of the different statements in Section 5.4. The creation of obligations and the situations in which it is necessary to consider the lifetime expressions will be covered in Section 3.5.

3.2 Types of Obligations

In this subsection we describe the different kinds of obligations we are working with in the context of this work. One can certainly think of other useful types of obligations, however the focus of this work is to present techniques to handle obligations in general, rather than to give a complete set of use cases.

`mustRelease`(r, t) A release obligation forces a thread to release the lock on `r` within time t. As we saw in the example shown in Figure 10, this type of obligation is generated by every `acquire` statement and it is consumed by `release` statements.

`mayReceive`(c, n) As the name suggests, this kind of obligation does not enforce any kind of behaviour. It simply gives a thread permission, or *credit* to receive n messages over channel c. The creation and consumption of this kind of obligation is a bit more involved and will be covered later in Sections 3.5 and 3.6.

`mustSend`(c, n, t) Send obligations guarantee that (at least) n messages are sent over channel c within time t and are therefore the counterpart to receive credits. Since the creation and consumption of these obligations is connected to receive credits, it will also be covered later in Sections 3.5 and 3.6.

`mustTerminate`(t) Termination obligations force methods to terminate within time t. This means that each loop inside the method *must* terminate and each method that gets called *must* terminate too. Whenever a method gets called that "promises" to terminate, i.e. if its precondition takes a termination obligation, the obligation is *copied* rather than given away. As a direct consequence, this kind of obligation cannot be consumed, i.e. a method cannot get rid of it by passing it to another method. The only way to fulfil this obligation is by actually terminating.

3.3 Leak Checks

Consider again method `locker` in Figure 10. Had we not called method `unlock`, there would have been one release obligation left at the end of the method. This behaviour is unsound and has to be prevented because otherwise the complete concept of obligations is useless, if a method is free to ignore them. To enforce that all obligations are satisfied, we perform so-called leak checks at the end of each method to ensure that no more obligations are held. This leak check, albeit necessary, is not enough however to ensure correctness. There are several other situations that require a leak check.

3.3.1 Method Calls

If we call a method that promises to terminate by requiring a `mustTerminate` obligation, we may keep some obligations in the caller. Since the method is going to terminate, we know that there will be a possibility for the caller to satisfy them eventually. On the other hand, if the method cannot give a termination guarantee, we cannot keep *any* obligations in the caller as there might not exist a possibility to satisfy them. Therefore, whenever a non-terminating method is called, a leak check has to be performed at the call site.

Forking a method is no problem since the method will execute concurrently in another thread and the caller is not blocked.

3.3.2 Loops

The situation with loops is similar to method calls. If a loop does not promise to terminate, we have to perform a leak check before we enter it. Additionally however, we need to perform a leak check at the end of the loop body to ensure that no obligations are leaked throughout the loop body.

```
channel C(x : int);

method normal(c : C)
    requires c != null
{
    //leak check causes no problems
}

method stronger(c : C)
    requires c != null
    requires mustSend(c, 1, 1)
{
    //leak check will fail
}
```

Figure 11: Violation of the Rule of Consequence

To summarise, we note that we need a leak check before every non-terminating loop, at the end of each loop body, before every call of a non-terminating method and at the end of every method.

3.3.3 Rule of Consequence

One central rule in Hoare logic is the Rule of Consequence [Hoa69]:

$$\frac{P \rightarrow P' \wedge \{P'\}\ \ S\ \ \{Q'\} \wedge Q' \rightarrow Q}{\{P\}\ \ S\ \ \{Q\}}$$

If the execution of a program S results in a state that satisfies Q' then it is also admissible to say that the program results in a state satisfying Q if $Q' \rightarrow Q$. Similarly, if P' is the precondition of S, then if S is executed in a state P, where $P \rightarrow P'$, it will result in a state that satisfies Q'. In other words, it is always legal to strengthen the precondition and weaken the postcondition.

A fundamental problem of the leak checks is, that it violates the Rule of Consequence. Consider the example shown in Figure 11. Method `normal` passes the leak check at the end of its body without problems. The second version of the method has a stronger precondition that implies the first one. However the leak check of this method will fail as the obligation is not satisfied. We have therefore created a situation where it is no longer possible to blindly strengthen any precondition. *Side note:* This is an intrinsic property and does not depend on the specific implementation of the leak check.

The verifiers used in the Viper framework do not make use of this property and we will therefore not run into problems in the context of our verification. However it is worth noting that in general, if the concept of obligations is implemented, the verification technique might need to be adapted to account for this innateness.

```
class Data {
  var ready : bool
  var data : int
}

class Consumer {

  method waitUntilReady(d : Data)
    requires d != null && mustRelease(d,2)
    ensures mustRelease(d,1)
  {
    var wait : bool := true

    while (wait)
      invariant mustRelease(d,1)
    {
      //give another thread the chance to
      //write the result
      release d
      acquire d

      //check if data is ready
      if (d.ready) {
        wait := false;
      }
    }
  }
}
```

Figure 12: Unbounded Obligations

3.4 Unbounded Obligations

Consider the example in Figure 12 which shows method `waitUntilReady` that waits until the data field of its argument is ready for consumption. Note that neither the loop, nor the method promises to terminate. It is therefore perfectly acceptable that we wait indefinitely. A comparable situation could for example arise in a control process that regularly checks a consistency criterion and it is reasonable that such a process never terminates.

As we defined the lifetime of an obligation so far, this example would not verify: the lifetime expression of the release obligation does not decrease throughout the loop. The problem is that the expressions of *different* obligations are compared. In the the loop we release the lock and re-acquire it. This means that in the meantime, other threads are unblocked. As soon as we get hold of the lock again, we are left with another release obligation. That means that the lifetime mentioned in the invariant does not apply to this fresh obligation.

To differentiate between these two kinds of obligations, *unbounded*, or *fresh* obligations were introduced. This kind of obligations are not associated with a lifetime expression and are therefore exempted from lifetime checks.

Throughout this report we will use the terms unbounded and fresh interchangeably.

```
channel C(x:int)

class A {

    method f(c : C)
        requires c != null
    {
        var n : int
        receive n := c  //wait indefinitely

        //satisfy obligation to send
        send c(0)

        //leak check succeeds
    }
}
```

Figure 13: Receive Statements Must Not Produce Obligations

3.5 Creation of Obligations

There are essentially three ways to create obligations. The first one is by executing designated statements. We have already encountered this case in the form of the `acquire` statement in the example shown in Figure 10: whenever we lock an object, we get an obligation to unlock it eventually.

Note, that the `receive` statement, used to get a message over a channel, does *not* create an obligation to send a message. If we are in a state where we hold no `mayReceive` credit and a receive statement is executed we produce a verification error. Figure 13 shows an example of why this is necessary. If the receive statement on line 9 would produce an obligation to send, we could simply satisfy it later and the leak check would pass. It is obvious however that the receive statement would not terminate.

The second way of generating obligations is by executing methods, or to be more specific, by exhaling their precondition. While calling methods might also consume obligations as described in Section 3.6, there is also a situation that produces obligations. Consider the example shown in Figure 14. By giving method `g` credits to receive a message, we generate an unbounded obligation to send a message.

The most interesting way of generating obligations is by inhaling them. This can happen in three situations:

Inhaling Preconditions. If the precondition specifies obligations we inhale them at the beginning of the method body. All obligations that are inhaled in this way are bounded by a lifetime expression. Originally [BM15], unbounded obligations could be mentioned in the preconditions and would then get transformed to bounded ones with a lifetime of $\top$. Since the implementation cannot rely on such a maximum value, we decided to be more restrictive and only allow bounded obligations. In practice this is not a problem as the expressiveness is not limited.

If we have more than one obligation for the same object, for example:

```
channel C(x:int)

class A {

    method g(c:C)
        requires c != null
        requires mayReceive(c, 1)
    {
        // ...
    }

    method f(c:C)
        requires c != null
    {
        fork g(c)

        //satisfy obligation
        send c(1)
    }
}
```

Figure 14: Exhaling Credits

requires mustSend(c, 1, t_1)

...

requires mustSend(c, 1, t_n)

the lifetime corresponds to the minimum:

$$t_b := \min(t_1, \ldots, t_n)$$

Inhaling Loop Invariants. In this case the actions are similar to inhaling obligations from the method precondition. The lifetime of the bounded obligations is stored in a separate value that is later needed to check that the lifetime - expression has decreased at the end of the loop, to prevent the obligations from being passed from iteration to iteration indefinitely. Similar to preconditions, we disallow unbounded obligations in invariants.

Inhaling Postconditions of Called Methods. After a method m terminates we inhale its postcondition p_m. Inhaling obligations mentioned in this way is straightforward because their lifetime can be ignored as they are inhaled. This means that the lifetime we stored for our bounded obligations does not change. To understand why ignoring the lifetime is admissible, consider the following two cases:

1) The obligation we inhale from p_m was originally inhaled in the precondition of m. This means that we originally passed it to m when we called it. In that case we effectively treat the method call as a "no-op" that happened instantaneously and had no effect on the lifetime.

2) The obligation was fresh in m and then converted to a bounded obligation in the postcondition p_m. By replacing the exact value of the inhaled lifetime l_0 with

the lifetime l_1 stored for our bounded obligations, we may effectively increase l_0 if $l_1 > l_0$. However, if we think of m as being inlined at the call site, the fresh obligation is effectively converted directly to a bounded one with lifetime l_1. It is essential that it gets bounded, but the exact lifetime is secondary.

Both cases highlight the fact mentioned earlier, that the lifetime expression does not directly correspond to a notion of real-time. The key property of lifetime expressions is that bounded obligations are never converted to unbounded ones and that the lifetime always decreases when they are moved to another context, to avoid that they are passed around indefinitely.

3.6 Consumption of Obligations

There are two ways of getting rid of obligations. The first one is by satisfying them by executing particular statements. For example the `release` statement will satisfy the `mustRelease` obligation.

Each such statement can satisfy both bounded and unbounded obligations. Say we are in the following state:

$$\sigma_1 := \{\texttt{mustSend(c, 1, }\top\texttt{)}\}$$

where we have an `unbounded` obligation to `send` one message over channel c and we execute the following statement:

```
send c(msg)
```

This will satisfy the obligation and it does not matter if the obligation is bounded or not.

If we are in a state where we hold bounded and unbounded obligations at the same time, it is advantageous to satisfy the bounded obligations first since they need to be satisfied 'earlier', i.e. they cannot be passed to any other context.

Another possibility to decrease the number of obligations is to pass them to other methods. When exhaling the precondition of the methods it is necessary to look at bounded and unbounded versions at the same time. Consider the following state:

$$\sigma_2 := \{\texttt{mustSend}(c, 2, \top),\ \texttt{mustSend}(c, 3, t_1)\}$$

where we have three bounded and two unbounded send obligations for channel c. If we call method `f` with the following specification:

```
method f(c : C)
   requires mustSend(c, 4, t2)
```

we are forced to compare the lifetimes t_1 and t_2. There are two cases:

case a if $t_1 \geq t_2$

case b if $t_1 < t_2$

In case a, the lifetime check succeeds and we can transfer all our bounded obligations. In the second case we cannot pass our bounded obligations to this method, because that would essentially increase their lifetime. In either case we have not enough obligations to satisfy the precondition of `f`. The key idea here is that unbounded obligations are normal bounded obligations with an extremely high lifetime. That means that every lifetime check will automatically succeed. We can therefore substitute bounded obligations with unbounded ones if we need to (it is always acceptable to satisfy an obligation earlier than needed). Consider again the two cases:

Case a. We have already exhaled three bounded obligations and can substitute the remaining one with an unbounded obligation. The resulting state is

$$\sigma_a := \{\texttt{mustSend}(c, 1, \top)\}$$

Case b. Since we have exhaled no bounded obligation we need to substitute with our two unbounded obligations. However we are still short two obligations. Since there are no more obligations in our current state, we can generate receive credits now. The resulting state is

$$\sigma_b := \{\texttt{mustSend}(c, 3, t_1),\ \texttt{mayReceive}(c, 2\}$$

In general the approach is always the same: first we try to get exhale all bounded obligations and if there are not enough, we substitute with unbounded ones. If there still not enough obligations, we can generate credits. If credits are not allowed, we generate a verification error. In Section 5.2.2 we will describe in detail how the exhale statements are generated.

4 Silver Extensions

In this section we describe the extension to the Silver language that enables the encoding of obligations. One of the core requirements of this project was to keep the necessary changes to Silver to a minimum and to remove earlier extensions that were no longer necessary.

4.1 For All References

The main extension to Silver is the `forallrefs` expression, originally introduced by Klauser [Kla14]. The expression

$$\texttt{forallrefs}[f_1, f_2, \ldots, f_n]\ r :: e$$

makes it possible to evaluate e for all objects in the heap that have at least one of the fields $f_1, f_2, \ldots, f_n$. The expression is instantiated for every applicable heap chunk and the place-holder r is then replaced by the reference to that object.

The main purpose of this expression is to be more efficient than a quantification over the complete heap, as we only need to consider heap chunks that have specific fields.

Predicates. For the purpose of this project we enhanced the `forallrefs` expression in a way that made it possible to reason about predicates too. Consider the example shown in Figure 15. Line 13 shows an application to all fields `f`: we check that, if we have access to a field `r.f`, we have full access to it. Note that since we have no access to `y.f`, the property is not checked for reference `y` and the statement yields true.

On line 14 we see an application of the expression that references the predicate `p`. Because in general predicates can have multiple parameters of different types, we restrict the applicable arguments to predicates that take only one argument of type `Ref`. In Section 7.1 we will discuss how this restriction could be lifted.

In-Place Evaluation. Consider the example shown in Figure 16. On line 14 we first exhale one full permission to the field and then check whether it is smaller than 10000 or not. To avoid problems with access permissions, the exhaling of the permission takes effect *after* all properties have been checked.

Internally we keep a copy of the heap in the pre-state before the exhale and check all properties on this copy. All statements that modify the heap (i.e. all `acc` statements) are executed on the 'real' heap. After the exhale has completed, the copy is thrown away and changes to the heap become visible.

In the context of the `forallrefs` expression we need the expression to be executed on the partially consumed heap *instead* of the copy. Lines 16-20 show exactly this behaviour. On line 16 we first exhale one full permission and then check the property `x.f < 100` for all *remaining* references. Similarly, the property on line 19 can still be checked, since it is evaluated on the heap in the

```
predicate p(r : Ref) { true }

field f : Int

method g(x : Ref, y : Ref)
    requires x != null && acc(x.f)
    requires y != null

{
    inhale acc(p(x), 3*write)
    inhale acc(p(y), 2*write)

    assert forallrefs [f] r :: perm(r.f) == write
    assert forallrefs [p] r :: perm(p(r)) >= 2*write

    assert forallrefs [p] r :: perm(p(r)) == 2*write //Error!
}
```

Figure 15: Reasoning About Predicates With `forallrefs` Expression

```
field f : Int

method g(r : Ref, s : Ref, t : Ref, u : Ref)
    requires r != null && acc(r.f, write)
    requires s != null && acc(s.f, write)
    requires t != null && acc(t.f, write)
    requires u != null && acc(u.f, write)
{
    r.f := 9973
    s.f := 997
    t.f := 97
    u.f := 7

    exhale acc(r.f, write) && r.f < 10000

    exhale acc(s.f) && (forallrefs [f] x :: x.f < 100)
    exhale  (forallrefs [f] x1 :: x1.f < 100) &&
            acc(t.f) &&
            t.f < 100 &&
            (forallrefs [f] x2 :: x2.f < 10)
}
```

Figure 16: In-Place Evaluation of `forallrefs` Expression

pre-state and the last `forallrefs` expression on line 20 also yields true as it is executed on the partially consumed heap.

Outlook: This new behaviour makes it possible to implement some of our ideas in the first place. Consider the leak check, for example. To enforce that no obligation gets leaked at the end of the method, we iterate over all obligations using the `forallrefs` expression. This leak check needs to be done *after* the postcondition has been exhaled. The only way to achieve this, apart from manually exhaling the postcondition, is to encode the leak check as part of the postcondition itself as described in detail in Section 5.5. Because the postcondition can transfer some of the obligations back to the caller, the leak check must not check the pre-state of the exhale, but rather the current state after the remaining obligations may have been returned.

Implementation. As explained above, to evaluate the expression

$$\texttt{forallrefs}\ [F]\ r :: e$$

we need to iterate over all applicable heap chunks in the partially consumed heap h_p. As an optimisation, we only consider chunks that have a field listed in the parameter list F. The expression e is then evaluated in every state where the place-holder r has been replaced by the reference to the heap chunk. The pseudocode of this algorithm looks like this:

σ_p *//state with partially consumed heap*
var result : **bool** := **true**
$\forall c \in$ chunks
    var temp : **bool** := c.field $\in F$
    $\sigma_t := \sigma_p[r \leftarrow c.\text{reference}]$
    result := temp $\wedge$ $[\![e]\!]_{\sigma_t}$

4.2 Behaviour of Perm Expression

As mentioned before, we lifted some restrictions with respect to the usage of the `perm` expression. In particular, we allow the expression to appear in method specifications and loop invariants. This is vital for our encoding and enables, for example, the transfer of obligations between methods via the pre- and postcondition.

To further facilitate our encoding we changed the behaviour of the `perm` expression. Similar to the evaluation of the `forallrefs` expression, perm is now executed in-place on the partially consumed heap. This change was necessary since we must be able to make decisions 'in the middle' of executing exhale expressions. For example, this needs to be done when exhaling loop invariants before the entry of the loop. If a termination obligation has been exhaled, there is no need for a leak check, but this can only be detected if the `perm` expression is evaluated on the partially consumed heap.

4.3 Removed Extensions

Compared to an earlier version of the implementation of obligations [Kla14], we were able to remove all extensions that were related to the so-called 'token amounts'. Originally, there was a need to keep track of *negative* obligation amounts and the 'token fields' were introduced to handle this requirement. These were another type of field for which it was possible to hold positive and negative permission amounts. Along with this new kind of field came two other extensions, the 'token predicate' and the 'token amount term', which made it possible to perform efficient calculations with the token fields.

In newer versions of the concept of obligations it was no longer necessary to work with negative values, which rendered these extensions superfluous.

5 Encoding of Obligations in Silver

In this section we will describe how Chalice programs are translated to Silver and how the concept of obligations is encoded. We will not cover the complete translation of Chalice programs, but focus on the interesting parts related to obligations. This means that the aspects that are not mentioned in this report remain unchanged compared to earlier versions of Chalice2Silver.

5.1 Encoding Obligations

There are two types of obligations we need to encode: 'boolean obligations' that are either true or false, or 'numeric obligations' that are associated with an integer amount. For example, the obligation to send n messages over channel c in time t, is a numeric obligation because we essentially have n obligations to send 1 message over c. On the other hand, we cannot release a lock multiple times, which means that release obligations can be represented by boolean values.

We distinguish these types because depending on the kind of obligation we are dealing with, we have to use different mechanics to encode them. The underlying idea is the same in both cases: we use **access permissions** to represent the amount of obligations that are held in the current state. Throughout this project we only use non-negative multiples of full permissions, i.e. neither case makes use of fractional permissions.

Boolean Obligations. In Silver we can have at most one full access permission to a field `f`. Intuitively, our encoding uses the following mapping to represent boolean values:

$$(\texttt{perm(x.f)} = \texttt{none}) \rightarrow \texttt{false}$$

$$(\texttt{perm(x.f)} = \texttt{write}) \rightarrow \texttt{true}$$

If we have full access to the field, we interpret it as the value `true` whereas no permission means `false`.

Numeric Obligations. To represent numeric values we use Silver predicates of the following form:

```
predicate p(r : Ref) {true}
```

The amount of permission that is stored with a predicate `p` represents the amount of obligations we currently hold. For example:

$$(\texttt{perm(p(c.f))} = 3 * \texttt{write}) \rightarrow 3 \text{ obligations}$$

If we have $(i * \texttt{write})$ permission, we interpret this as a state in which we hold i obligations. If we need to add, say two obligations, to our current state, we can simply execute the following statement

```
inhale acc(p(x.f), 2*write)
```

Which increases the current amount of permission by $(2 * \texttt{write})$.

The following table summarises the types of the chalice obligations and are how they are encoded in Silver:

	Chalice Obligation	Type	Silver

Messages	`mustSend(c, n, t)`
	`mustSend(c, n)`
	`mayReceive(c, n)`
		numeric	`mustSendBounded`
			`mustSendUnbounded`
			`mayReceive`

Locks	`mustRelease(o, t)` `mustRelease(o)`	boolean	`mustReleaseBounded` `mustReleaseUnbounded`
Termination	`mustTerminate(t)`	numeric	`mustTerminate`

Aside from the termination obligation, all types are straightforward and directly dictated by the usage of the obligation. Although termination is a boolean property, we often run into situations where a binary value is insufficient to represent this kind of obligation. In Section 5.5 will describe in detail why the numeric property is needed.

Consequences. Encoding release obligations using Silver fields has subtle consequences when verifying methods. Consider the following method:

```
method f(a : A, b : A)
    requires a != null && mustRelease(a, 1)
    requires b != null && mustRelease(b, 1)
    { ... }
```

Because we inhale an obligation for each reference at the beginning of the method body, which is essentially executing:

```
inhale acc(a.mustReleaseBounded, write)
inhale acc(b.mustReleaseBounded, write)
```

we implicitly excluded the possibility of `a` and `b` being aliases. Since we *separately* inhaled full access for the field `mustReleaseBounded` for both references, the verifier can conclude that the references must be to different objects, because we can have at most one full access permission to a field.

This simplifies our encoding since there is no need to explicitly check for the possibility of aliasing because we know that a caller can never satisfy the precondition if the method is called with aliasing parameters. On the other hand, from the point of view of the method, if we assume that the parameters are aliases, the precondition is equal to false and the body will verify in any case.

5.2 Transfer of Obligations

Silver provides a wide variety of built-in concepts such as methods with pre- and postconditions, loop invariants, etc. and it is beneficial to use these mechanisms as often as possible since verifier back-ends handle them automatically. For example, a method call is automatically replaced by the exhaling (assertion) of the precondition and the inhaling (assumption) of the postcondition.

Depending on whether we inhale obligations, or exhale them, we need to perform different actions. To correctly encode this behaviour we make use of two designated macros that can produce assertions that can be inhaled/exhaled.

This idea of using parametrised macros [BM15] to produce assertions that are specially tailored for the context they are used in, is useful since it provides a way of treating all obligations in a uniform way.

We will now describe how these macros work and how they are used to encode the transfer of obligations by method specifications and loop invariants. Throughout this description we will omit small implementation details, like the encoding of specific obligations, and focus on the general techniques. We will often refer to the predicates/fields used to represent the obligations as 'fields'.

When we present the pseudo code of the macros, we use *code written in italics* to denote the parts that are executed by the front-end. The `code written in typewriter-font` denotes the Silver code that is generated by the macro.

5.2.1 Inhale

The first macro produces assertions that can be inhaled. This is relatively straightforward as we essentially just have to increase the amount of the obligation. The complete algorithm can be found in Figure 17. Parameter `a` is a multiple of a full write permission that represents the amount of obligations we inhale, `r` is reference to an object and `f` denotes the field representing the obligation that is increased. Since the macro is used by the front-end to generate these inhalable assertions, the `argument types` symbolise types of the Silver AST. Note that we omitted implementation details to distinguish between fields and predicates. The body of the macro consists of one statement on line 3, which leads to an access permission being inhaled.

Depending on the kind of obligation we need to inhale, we call the macro with different arguments. For example, if we need to inhale a bounded release obligation for object `r` and time t:

mustRelease(r, t)

we would use the macro like this:

`inhale` *inhaleMacro*(`write`, `r`, `mustReleaseBounded`)

which produces the following statement that inhales one write permission for the field `mustReleaseBounded` of reference r:

```
inhale acc(r.mustReleaseBounded, write)
```

The termination obligation uses a special argument, `this`, since it is not tied to an object, but rather the current thread. In Section 5.2.3, after the description of the exhale macro, we will shortly explain why this exceptional treatment is necessary.

Figure 18 summarises all possible calls of the inhale macro. The Identifier 'Inhale' represents the translation of expressions used in inhale expressions.

```
inhaleMacro(a : PermExp, r : Ref, f : Field) =
    acc(r.f, a)
```

Figure 17: Inhale Macro

```
w := write                    iM := inhaleMacro

Inhale(mustSend(r, a))       := iM(a, r, mustSendUnbounded)
Inhale(mustSend(r, a, t))    := iM(a, r, mustSendBounded)
Inhale(mayReceive(r, a))     := iM(a, r, mayReceive)
Inhale(mustRelease(r))       := iM(w, r, mustReleaseUnbounded)
Inhale(mustRelease(r, t))    := iM(w, r, mustReleaseBounded)
Inhale(mustTerminate(t))     := iM(w, this, mustTerminate)
```

Figure 18: Usage of the Inhale Macro, Depending on the Obligation

5.2.2 Exhale

Compared to the inhale macro, the macro to produce assertions that can be exhaled, is much more complex. Several different tasks may need to be performed, depending on the kind of obligation we exhale:

- Decrease the amount of obligations
- Perform lifetime checks
- Generate credits

There is an inherent problem with the generation of credits, since the term 'generation' implies an operation that *increases* some value. However, as this assertion is *exhaled*, all operations that are performed are negative: we can only give away permission to fields/predicates. This mismatch leads to implementation problems and it becomes necessary to split the exhale operation into multiple statements. We will therefore first describe the theoretical idea of the exhale macro and then highlight some implementation details.

Theoretical Idea. When exhaling an obligation, it is necessary to look at the bounded and unbounded versions at the same time. For example, when exhaling the obligation **mustSend(c, 1, 1)**, we check whether we have such an obligation or not. If we have no bounded obligation, we can still exhale an unbounded one, since it is simply an obligation with an arbitrarily high lifetime. If we are in a state where there is also no unbounded obligation, we generate a credit.

Similarly, if we exhale a bounded obligation, we may need to perform a lifetime check. If this check fails, i.e. the exhaled obligation has a *higher* lifetime than our bounded one, we can still try to exhale an unbounded obligation, for which the lifetime check would succeed by definition, before generating a credit.

```
exhaleStageOne(a : PermExp, t : IntExp, performLTCheck : Boolean
               r : Ref, bf : Field, df : Field, cf : Field) =

exhaleStageTwo(a : PermExp, r: Ref, df : Field, cf : Field) =

exhaleStageThree(a : PermExp, r : Ref, cf : Field) =
```

Figure 19: Interfaces of the Three Exhale Stages

Both cases suggest the same underlying structure: whenever the exhale of a bounded obligation fails, we may try to *substitute* it with its unbounded version. If this second exhale fails too, we can generate a credit (if they are allowed), or produce a verification error.

It is interesting to note that a similar idea can be applied to the exhaling of credits. Compared to obligations, there is no unbounded version, which means that if there are no credits to exhale we directly generate obligations. Aside from disregarding 'unbounded credits', the only difference is that this assertion will never fail when it is exhaled.

The fundamental idea of the exhale macro is to think of it as a combination of three designated stages where each individual stage performs one task:

- Stage One: perform lifetime checks and exhale bounded obligations
- Stage Two: exhale unbounded obligations
- Stage Three: generate credits

The first stage tries to exhale bounded obligations and if it succeeds we are done. This stage might fail due to an insufficient number of obligations, or due to the lifetime check if the exhaled lifetime is bigger than the one of our bounded obligations. In either case, there are some obligations left that need to be exhaled and they are passed on to the second stage.

The second stage tries to exhale unbounded obligations. This stage might fail due to an insufficient number of unbounded obligations. If that is the case, all remaining obligations that need to be exhaled are passed on to the third stage.

The third stage is concerned only with the generation of credits. We know at this point that there are no more obligations left that could be exhaled and all additional obligations we exhale will generate credits. As mentioned above, there is a problem with the generation of credits at this point as we are still in an exhale statement. However, we will ignore this for now and explain the solution to it below.

Depending on the kind of obligation we exhale, we skip certain stages and depending on the context, we may omit the lifetime check in stage one.

Figure 19 shows the interfaces of the three stages. Parameter `a` is straightforward and denotes the number of obligations that need to be exhaled. Since we may need to perform a lifetime check in the first stage, the integer expression `t` corresponds to the lifetime of the obligation that is exhaled. That means that

for the lifetime check to succeed, the exhaled lifetime `t` must be smaller than the one of our bounded obligations `tb`:

$$\texttt{tb} > \texttt{t}$$

The lifetime of our bounded obligations `tb` is stored by using a domain function that is axiomatised at the beginning of each method. In Section 5.4 we will describe in detail how this lifetime is calculated. For now we can assume that this value is available and the comparison can be executed.

Depending on the context we may omit the lifetime check altogether which can be controlled by the flag `performLTCheck`.

The three parameters `bf`, `df` and `cf` represent the **bounded field**, **dual field** and **credit field**, respectively. The bounded field is the name of the field that stores the bounded obligations, while its dual field keeps track of the unbounded obligations. For example, if we have

$$\texttt{bf} := \texttt{mustSendBounded}$$

its dual field would be:

$$\texttt{df} := \texttt{mustSendUnbounded}$$

The credit field is straightforward and denotes the field that stores credits.

Depending on the kind of obligation that is exhaled, some fields may not exist. For example, the credit field has no dual field. This is no problem however since the macro is never used in a way that would require these non-existing fields.

To exploit the previously mentioned underlying structure of the exhale statement, we must also be able to express the exhaling of credits in terms of these fields. The field we need to decrease first is the credits field, that means we set

$$\texttt{df} := \texttt{mayReceive}$$

Because credits are unbounded we can directly start in the second stage with the dual field and ignore the non-existing 'bounded credits'. If we exhale more credits than we currently have, we must generate obligations. That means if we set

$$\texttt{cf} := \texttt{mustSendUnounded}$$

we will automatically generate obligations if the amount of credits exhaled is too high.

Figure 20 summarises how the different obligations are exhaled, i.e. how the exhale macro is used depending on the obligation. To keep the overview readable we introduced abbreviations for the macro- and field/predicate names. The value `null` marks non-existing fields, as for example for the termination obligation, for which neither the dual predicate nor the credit predicate exist. The identifier 'Exhale' represents the translation of expressions in exhale statements.

Similar to the inhale macro, the termination obligation is tied to the `this` reference. In Section 5.2.3 we will describe the reasons for this choice.

```
es1 := exhaleStageOne          es2 := exhaleStageTwo

mr := mayReceive               mt := mustTerminate
msb := mustSendBounded         msu := mustSendUnbounded
mrb := mustReleaseBounded      msu := mustReleaseUnbounded

w := write                     pc := performLifetimeCheck

Exhale(mustSend(r, a), _)        := es2(a, r, msu, mr)
Exhale(mustSend(r, a, t), pc)    := es1(a, t, pc, r, msb, msu, mr)
Exhale(mayReceive(r, a), _)      := es2(a, r, mr, msu)
Exhale(mustRelease(r), _)        := es2(w, r, mru, null)
Exhale(mustRelease(r, t), pc)    := es1(w, t, pc, r, mrb, mru, null)
Exhale(mustTerminate(t), pc)     := es1(w, t, pc, this, mt, null, null)
```

Figure 20: Usage of the Exhale Macro

```
exhaleStageOne(a : PermExp, t : IntExp, performLTCheck : Boolean
               r : Ref, bf : Field, df : Field, cf : Field) =

    (performLTCheck ==> tb > t ) ? (

        (perm(r.bf)) ≥ a) ? (
            //enough obls. are available to exhale
            acc(r.bf, a)
        ) : (
            //exhale remaining obls. from dual field
            exhaleStageTwo((a-perm(r.bf)), r, df, cf) &&
            //set f to zero
            acc(r.bf, perm(r.bf))
        )
    ) : (
        //lifetime check has failed
        exhaleStageTwo(a, r, df, cf)
    )
```

Figure 21: Stage One of the Exhale Macro

```
exhaleStageTwo(a : PermExp, r: Ref, df : Field, cf : Field) =

    if (df != null) {
        (perm(r.df) ≥ a) ? (
            //enough obls. are available to exhale
            acc(r.df, a)
        ) : (
            //generate credits with remaining obls.
            exhaleStageThree((a-perm(r.df)), r, cf) &&
            //set df to zero
            acc(r.df, perm(r.df))
        )
    } else {
        //termination obligation wasn't exhaled
        //not necessarily an error
        true
    }
```

Figure 22: Stage Two of the Exhale Macro

```
exhaleStageThree(a : PermExp, r : Ref, cf : Field) =

    if (cf != null) {
        //generate credits: increase the value of cf
        acc(r.cf, -a)
    } else {
        (a == none)
    }
```

Figure 23: Stage Three of the Exhale Macro

Stage One. Figure 21 shows the first stage of the exhale macro. Note the usage of the ternary operator `?:` that has standard semantics. If the lifetime check fails we directly continue with the second stage on line 17. If it succeeds, or there is no need to perform one, we check if there are enough bounded obligations to exhale (line 6). If there are, we exhale them on line 8 and if not, we compute how many are still missing (`a - perm(bf)`) and propagate this amount to the second stage (line 11). After the second stage has completed we exhale all remaining bounded obligations on line 13.

Stage Two. The second stage is similar to the first one and it its pseudocode can be found in Figure 22. The *if* statement on line 3 is executed by the front-end and checks if this stage needs to be executed at all. This is the case if a termination obligation is being exhaled and the lifetime check in the first stage fails. Please refer to Section 5.2.3 for more details and why we allow this case at all.

If we have enough unbounded obligations we exhale them on line 6. If there are not enough obligations we start the third stage to generate an appropriate amount of credits (line 9). After the third stage has finished we exhale all unbounded obligations on line 11.

Stage Three. Figure 23 shows the third stage. If there are no credits allowed for this obligation ($cf =$ *null*) we assert that the amount is zero (line 7). This

means that at this point we must have gotten rid of all obligations, otherwise we have a verification error. If credits are allowed we generate them on line 5. Note that we hinted at the credit generation by exhaling a negative amount, thereby increasing the value of `cf` (this is impossible in Silver however).

Implementation Details. As mentioned several times, the exhale macro cannot be directly implemented as presented in the previous section. The sole reason for this is the statement on line 5 in Figure 23. Before we sketch the solution to this problem it is worth noting that, depending on the situation, we may not even need to deal with this problem. For example, at the end of a method, when we exhale the postcondition, we know that the generation of a fresh obligation is forbidden since it would directly lead to a violation of the leak check. In that case we can simply exhale `false` in stage three which results in a verification error. To switch between these two possibilities we introduced a boolean flag that can disable the third stage by exhaling false, or use the solution presented next.

Consider the following method:

```
method f(a : A, b : Bool, i : int)
   requires i > 0
   requires mustSend(a, 5)
   requires b ==> mustSend(a, i)
   { ... }
```

Statically we can compute φ, the *symbolic* sum of all exhaled amounts mentioned in obligations:

$$\varphi := \texttt{i+5}$$

Since all amounts must be positive we know that the overall sum will be positive. When exhaling the precondition of `f`, we give away *at most* φ send obligations. In turn this means that the exhale statement can generate at most φ send credits. This knowledge allows us to temporarily use an 'inverted encoding'. Instead of directly using the credit field in stage three, we use a 'guard predicate' that gets decreased. Before the exhale, we initialise this guard such that we have *exactly* φ full access permissions to it. After the precondition has been exhaled, we calculate by how much it has been decreased, which equals the number of obligations that have been generated. For the previous example this would look like this:

```
φ := (i + 5) * write
inhale acc(guard(a), φ)
f(a,b,i) //exhales the precondition
n := φ - perm(guard(a))
inhale acc(mayReceive(a), n)
```

Since we might generate credits and obligations in an exhale, we use two separate guards to calculate the correct results:

```
predicate creditGenerationGuard(r: Ref) { true }
predicate obligationGenerationGuard(r: Ref) { true }
```

This approach can suffer when we are dealing with aliasing parameters. If multiple parameters are aliases, the initialisation before the exhale statement

may inhale too much permission because it is executed multiple times for the same object. Consequently we must adapt the calculation after the exhale to deal with possible aliasing parameters.

We will present the idea for this adaptation for the case of *two* aliasing parameters and later sketch the general case. Assume we are given the following method:

```
method f(a : A, b : A, i : int, j : int)
   requires i > 0 && j > 0
   requires mustSend(a, i)
   requires mustSend(b, j)
   { ... }
```

The maximal number of credits that can be generated by this method is

$$\varphi = \texttt{i + j}$$

this means that whenever we call this method, we perform the following initialisation of the guard predicate:

```
φ := (i + j) * write
inhale acc(guard(a), φ)
inhale acc(guard(b), φ)
f(a,b,i,j) //exhales the precondition
```

Assume now, that `a` and `b` are aliases. In that case we will have initialised the guard twice, resulting in a state where:

```
perm(guard(a)) == perm(guard(b)) == 2 * φ
```

To account for this and all other cases of aliasing, we execute the calculation of the credits as follows:

```
if(a != b) {
   inhale acc(mayReceive(a), φ - perm(guard(a))
   inhale acc(mayReceive(b), φ - perm(guard(b))
}
if(a == b) {
   inhale acc(mayReceive(a), 2*φ - perm(guard(a))
}
```

This case distinction ensures that we inhale the correct amount of credits in every case. As an optimisation we only have to compare parameters of the same type, since obligations for different kinds of objects are unrelated. The drawbacks of this computations are obvious however. Since we have to consider *all* possible aliasing situations, we effectively have to consider all partitions of the input parameters which results in an exponential runtime of the algorithm. The sole consolation is that when we are dealing witch realistic examples, the number of parameters of the same type will typically be very small.

```
method f(a : bool, b : bool)
    requires a ==> mustTerminate(1)
    requires b ==> mustTerminate(10)
{ }

method g(a : bool, b : bool)
    requires mustTerminate(20)
{
    call f(a,b)
}
```

Figure 24: Multiple Termination Obligations

5.2.3 Termination Obligation

Since we use access permissions to encode our obligations we always need an object to which we can tie the fields/predicates. This is problematic for the termination obligation since it is a property of a thread, rather than an object. As indicated earlier, we use the `this` reference for this property since it exists for every method and is always non-null. The biggest advantage is related to its scope. Because it is an argument of the method, both the caller and the callee have access to it which enables the transfer of the obligation in the first place.

The drawback is related to methods that are called on other objects. Since the `this` reference passed to the method is the object reference, the caller needs to copy the termination obligation and tie it to that reference.

Implementation Details and Deviation From Original Scheme. Compared to the original scheme we must internally allow states in which we hold multiple termination obligations. Consider the methods shown in Figure 24. Method `g` inhales one termination obligation from its precondition and, depending on the values of `a` and `b`, might give away two termination obligations as it calls `f`.

Since we cannot hold a negative permission amount to a predicate, we must make sure that we have enough obligations to give away when we execute the call statement. To avoid any problems, we simply inhale 'enough' obligations before the call statement and then restore the original value afterwards. To determine how many obligations we need to inhale, we can syntactically check how many termination obligations could possibly be exhaled and then inhale this amount.

It might also be possible to encode the precondition in such a way that it only exhales a termination obligation if there is one available. However, the caller needs to inhale a termination obligation in some cases anyway. For example, if a non-terminating method calls a terminating one, it must give away a termination obligation. To ensure that this works we need to inhale one extra obligation. Because of this, we decided to keep the encoding of the specifications as simple as possible and push the complexity to the caller.

As mentioned in Section 5.2.2, we allow the possibility that the lifetime check

can fail when we exhale a termination obligation. This is necessary since we allow the transfer of multiple termination obligations. If the lifetime check fails, there might still be another termination obligation that gets transferred later in the exhale and hence we cannot automatically assume an error.

5.3 Leak Check

The task of the leak check is to detect if we are in a state where there are any obligations left. In the context of our encoding, we have to check if we hold any permission to a predicate that corresponds to an obligation. To achieve this, we use the `forallrefs` extension described earlier in Section 4.1.

The complete leak check λ would then look like this:

$\lambda_1 :=$ `forallrefs [mustSendBounded] r1 :: false`

$\lambda_2 :=$ `forallrefs [mustSendUnbounded] r2 :: false`

$\lambda_3 :=$ `forallrefs [mustReleaseBounded] r3 :: false`

$\lambda_4 :=$ `forallrefs [mustReleaseUnbounded] r4 :: false`

$$\lambda := \lambda_1 \ \&\& \ \lambda_2 \ \&\& \ \lambda_3 \ \&\& \ \lambda_4$$

Whenever we encounter a reference, for which we would have to evaluate the expression we know that we will fail (since we cannot assert `false`). This means that if there are any permissions left to any fields, or predicates, we will produce a verification error.

In Section 5.6 we will describe how the leak check is integrated in the encoding of method calls, in Section 5.5 we will describe how they are incorporated to ensure that no obligation gets leaked at the end of a method and finally in Section 5.8 we will discuss their usage in the context of loops.

5.4 Lifetime

As mentioned in Section 3.1, each obligation has an expression associated with it that ensures that it gets satisfied eventually. Whenever the obligation is passed to another context, for example to another method, this expression must strictly decrease. As soon as this expression reaches zero, the obligation must be satisfied, otherwise we get a verification error.

For our purposes this means that we need to perform lifetime checks whenever a method is called/forked and at the end of each loop body, when we exhale the loop invariant.

To store the lifetime, we use special functions for each kind of obligation. In our case this corresponds to the following three domain functions:

```
function lifetimeTerminates() : Int
function lifetimeMustSend(r : Ref) : Int
function lifetimeMustRelease(r : Ref) : Int
```

For example, if we have the following method specification

```
method f(c : Channel)
   requires mustTerminate(n)
   requires mustSend(c, 1, 1)
```

We would initialise the lifetime functions as follows:

```
inhale lifetimeTerminates() == n
inhale lifetimeMustSend(c) == 1
```

This lifetime is initialised once at the beginning of the method body and at the beginning of each loop body, for the obligations mentioned in the loop invariant.

Whenever we exhale an obligation the lifetime function is available and the lifetime expression of the exhaled obligation can be compared to it in the first stage of the macro, as described in Section 5.2.2.

5.4.1 Initialising the Lifetime

There are two main problems when initialising the lifetime functions: aliasing parameters and obligations that are guarded by implication. Consider the following two methods:

```
method g(a : Channel)
   requires mustSend(a, 1, 1)
   { ... }

method f(a : Channel, b : Channel)
   requires mustSend(a, 1, 5)
   requires mustSend(b, 1, 1)
   { call g(a); ... }
```

Should the call statement in `f` be allowed? Or to be more specific, is it safe to pass the obligation to send a message over `a` to method g? The answer is no. If `a` and `b` are aliases for the same object, the lifetime would not decrease as the obligation is passed to another context. If we take aliasing into account, the initialisation of the lifetime functions for method `f` look like this:

```
inhale (a == b) ==> lifetimeMustSend(a) == min(1, 5)
inhale (a != b) ==> lifetimeMustSend(a) == 5 &&
                    lifetimeMustSend(b) == 1
```

No matter what objects the parameters point to, we are always on the safe side and there is no way of circumventing the lifetime check. To show the problem with implications, consider the following example:

```
method g(a : Channel)
   requires mustSend(a, 1, 1)
   { ... }

method f(a : Channel, x : bool)
   requires mustSend(a, 1, 5)
   requires x ==> mustSend(a, 1, 1)
   { call g(a); ... }
```

Again, the call is unsafe since the lifetime functions must be initialised as follows:

```
inhale (a && !x)      ==> lifetimeMustSend(a) == 5
inhale (a && x)       ==> lifetimeMustSend(a) == min(1, 5)
```

Next we will describe the algorithm used to create the list of statements that initialise all lifetime functions.

Algorithm. We explain the idea for the initialisation of the lifetime functions for a method, but it is essentially the same when we are dealing with loops. To simplify the explanation we assume that we are given a method of the following form:

```
method f(...)
   requires b_1 → mustX(c_1, ..., t_1)
   ...
   requires b_n → mustX(c_n, ..., t_n)
```

Additionally make the following assumptions:

1. *All* receiver expressions c_i are of the same type. This simplifies the explanation of the algorithm and in practice, when there are parameters of multiple types, we have to execute the algorithm for each type of parameter separately. This is an optimisation since it is unnecessary to compare the lifetimes of obligations for different kinds of objects, since they are completely independent.

2. All obligations `mustX` are the same. The reason for this is that the lifetimes of different obligations are independent and there is no need to compare them. Again we can easily lift this restriction in practice.

3. All obligations are guarded by implications. This assumption simplifies the explanation, but is not a requirement in practice. Note that if an obligation is not guarded by an implication, we simply assume that is the consequence of an implication where the antecedent equals true.

Before we show the actual initialisation we explain the helper functions shown in Figure 25. The first three functions are straightforward. The idea of the last one, function `init2`, is to compute a 'conditional minimum' of its arguments t_1, t_2, where the value t_i only contributes to the minimum if the condition b_i is true. The value that is passed to the function `min2` depends on the value of b_i. If this condition is true, we consider the value t_i and if it is false, we pass the maximum of all inputs to the minimum function, thus ignoring t_i. If

```
// If-Then-Else
function ite(b : Bool, x : Int, y : Int) : Int {
    (b ? x : y)
}

// Calculate the minimum of 2 numbers
function min2(a: Int, b: Int): Int {
    (a > b ? b : a)
}

// Calculate the maximum of 2 numbers
function max2(a: Int, b: Int): Int {
    (a > b ? a : b)
}

function init2(b1 : Bool, t1 : Int, b2 : Bool, t2 : Int) : Int {

    min2(ite(b1, t1, max2(t1,t2)), ite(b2, t2, max2(t1,t2)))
}
```

Figure 25: Helper Functions for Lifetime Axiomatisation

all conditions b_i are false, the function will return the maximum of its input parameters.

In general we use versions of these helper functions with more parameters, but the idea is exactly the same. We then simply change the suffix to denote the arity of the function. For example,

`initN`

would denote the `init` function that computes the 'conditional minimum' of n parameters.

The axiomatisation of the lifetime functions is done at the beginning of the method body, after the precondition has been inhaled. At this point we have already inhaled the obligations mentioned in the specification. Since we use Silver fields and predicates for our bookkeeping, the verifier will have already taken care of aliasing for us. For example the execution of the following Silver statements, where `mustSendBounded` is a predicate:

```
assume a == b
inhale acc(mustSendBounded(a), write)
inhale acc(mustSendBounded(b), write)
```

will lead to a state that satisfies

```
assert perm(mustSendBounded(a)) == 2*write
```

In our context this means that we can use the `forallrefs` expression at the beginning of the method body to iterate over the current heap and check for which references we have inhaled an obligation:

`forallrefs [mustSendBounded] r ::` α

In combination with the helper functions shown in Figure 25, we can initialise the lifetime function of obligation `mustX` by executing the following statement:

```
inhale (forallrefs [mustSendBounded] r ::
 lifetimeX(r) ==
  initN(((r == c_1) && b_i), t_1, ..., ((r == c_n) && b_n), t_n))
```

Using the `forallrefs` expression we make sure that the function `lifetimeX` is initialised for all references for which we hold an obligation. The value is equal to the 'conditional minimum' of all lifetimes t_i mentioned in the precondition. The heart of this computation is the condition according to which the values are selected for the minimum. The condition:

$$((\texttt{r} == c_i)\ \&\&\ b_i)$$

is true if `r` is an alias for c_i and the guard b_i is true. Consequently we initialise the lifetime of the obligation to the minimum of all relevant lifetimes.

5.5 Methods

Compared to the original translation of methods without obligations, there are three additional tasks that need to performed when dealing with obligations. We need to inhale obligations via the precondition, perform a leak check at the end of the method and maybe perform a leak check at the call sites.

Inhaling the Precondition. The use of the precondition is twofold. At the call site we need to exhale it and at the start of the method body we need to inhale it. To translate this behaviour correctly, we use paired assertions in combination with the macros defined in Section 5.2. To translate the precondition α, we generate the following expression in Silver:

$$[\text{Inhale}(\alpha), \text{Exhale}(\alpha)]$$

At the start of the method body we inhale the correct amount of obligations and at the call site we correctly exhale the precondition, which may involve lifetime checks, etc. At the beginning of the method body we initialise the lifetime functions as described in Section 5.4.

Since it is allowed for the exhaling of the precondition to produce fresh obligations in certain situations, we enable the third stage for this exhale. This means that whenever credits or obligations are generated, they can be inhaled afterwards. We will describe this in more detail in Section 5.6 and Section 5.7 where we describe the encoding of the call and fork statements, respectively.

We deviate from the original scheme when dealing with the termination obligation. Because the precondition can contain multiple termination obligations, we could end up with more than one obligation to terminate. To avoid this situation inhale a termination obligation only if needed. To determine this, we use the perm expression and check if we already have an obligation to terminate or not.

Leak Check at the Call Site. At the end of the precondition, after it has been exhaled, we may need to perform a leak check. If the callee might not terminate, we must enforce that the caller does not retain any obligations. To achieve this, we add a leak check at the end of the precondition that checks if the termination obligation has decreased. If it has, a termination obligation has been exhaled and there is no need for a leak check. If no termination obligation has been exhaled, we know that we need to perform a leak check since the method might not terminate.

This leak check is also encoded using paired assertions to assure that it only applies to the call site:

$$[\texttt{true}, (\text{might not terminate}) \rightarrow \lambda]$$

There is a difference between calls and forks of the method. If the method is forked there is no need to perform a leak check as the forking method is not blocked. Since the leak check is encoded in the precondition of the method there must be a way to distinguish between these two situations. This simple predicate

```
predicate callLeakCheckGuard() {true}
```

is used as a flag to indicate if the method is called or forked. If we have access permission to the predicate, the method is called and if not, the method is forked. This means that in the caller we must inhale permission before calling the method and in the forker we can simply ignore this aspect.

There would also be the possibility to let the caller worry about the leak check. This solution would have several drawbacks since it is not evident how the caller could readily perform the leak check after the precondition of the called method has been exhaled.

Regardless of whether the leak check at the call site is encoded in the precondition or performed manually, the caller of the method has to perform some additional actions besides simply calling the method. Therefore it seems reasonable to keep this auxiliary work confined to the calling method such that the fork statement is not affected.

Leak Check at the End of the Method. Because it is possible to transfer obligations in the postcondition, the leak check has to be performed *after* the postcondition has been exhaled at the end of the method body. To achieve this, we encode the leak check as part of the postcondition of a method.

If we are given a postcondition β, we enhance the postcondition like this:

$$\beta' := \beta \wedge [\texttt{true}, \lambda]$$

Note the use of the paired assertion that ensures that the leak check is only evaluated when the postcondition is being exhaled. It becomes apparent why the `forallrefs` expression needs to be executed on the partially consumed heap. If this were not the case, the leak check could not be executed like this because it would always fail.

As mentioned earlier, it is never allowed for the postcondition to produce fresh obligations. To easily encode this, the third stage is disabled for this exhale as described in Section 5.2.2.

Summary. If we are given a method

$$\texttt{method f() requires } \alpha \texttt{ ensures } \beta \ \{\mathcal{S}\}$$

We enhance it as follows

```
method f()
   requires [Inhale(α), Exhale(α)]
   requires [true, !terminates → λ]
   ensures β ∧ [true, λ]
{
   //initialise lifetime functions
   //inhale termination obligation if needed
   S
}
```

and then continue with the normal Chalice2Silver translation.

5.6 Method Call

Since the verification is modular, we do not care how a method is implemented. The specification is the only relevant information that is needed to translate a method call. If we are given the call

```
call f()
```

the statement will be replaced by the exhaling of the precondition and the inhaling of the postcondition:

$$\texttt{exhale pre}_f$$
$$\texttt{inhale post}_f$$

As discussed previously, the exhaled precondition is generated by the exhale macro. Since this exhale might generate new obligations or credits, for example when exhaling an obligation in a state where we have no such obligation, we may need to inhale these afterwards. As described in detail in Section 5.2.2, we cannot handle the generation exclusively in the exhale macro. In the case of the method call, we check if credits/obligations should have been generated and then inhale them after the method call. To make this check possible we initialise the 'creation guards' before the method call and afterwards check if they have decreased. If they did, we know that credits/obligations have been generated and we can inhale them.

The second problem with exhaling the precondition is that there might be multiple termination obligations transferred. Since the current thread has a maximum of one termination obligation this might not be possible. To avoid problems in advance we simply inhale 'enough' termination obligations so the exhale will succeed in any case. This is a syntactic check of the precondition that counts the number of termination obligations that appear in the expression and then inhales the corresponding number of termination obligations. After the method call we restore the termination obligation to the original value.

If the method does not promise to terminate, the caller needs to perform a leak check at the call site. This leak check has to happen after the precondition of the called method has been exhaled. To achieve this we encode this check as part of the precondition of the method. This is described in more detail in Section 5.5. For the caller, this means that we have to initialise the `callLeackCheckGuard`, which is a simple inhale of one full access permission.

Summary. If we have the following call statement

```
call f()
```

we will enhance it as follows and then continue with the normal translation:

```
//inhale enough termination obligations
//inhale generation guards
//inhale callLeackCheckGuard
call f()
//inhale pending credits/obligations
//exhale callLeackCheckGuard
//restore termination obligation to original
```

5.7 Fork

The Silver language has no direct concept of executing a method asynchronously and it is up to the front-end to correctly simulate the behaviour of forked methods. Similar to a normal method call, the precondition needs to be checked at the fork site. This exhale operation might generate obligations/credits which means that after the fork has happened, they must be inhaled.

Since the method is executed in another thread, we cannot simply inhale the postcondition when the method is forked. This can be done only after we have joined the method at a later point. To make this possible, the fork statement returns a token.

Summary. The following statement

```
fork tok := f()
```

is replaced by the exhaling of the precondition and initialisation of a token.

```
exhale pre_f
//initialise the token tok
```

5.8 Loops

When translating loops we try to use as much of the existing Silver machinery as possible. For loops, this specifically includes the handling of loop-invariants. The following general loop statement

```
while (𝒜)
   invariant ℐ
{ 𝒮 }
```

will internally be translated to the following statements:

```
exhale 𝓘
while (𝓐)
{
   inhale 𝓘
   𝓢
   exhale 𝓘
}
inhale 𝓘
```

The encoding of a loop combines several aspects of method calls and the translation of method bodies: before we enter the loop, we check if it terminates. If it does, we are allowed to retain some obligations as there is a chance to satisfy them after the loop. On the other hand, if the loop does not promise to terminate, we must perform a leak check before entering. Inside the loop we first inhale the invariant which may include obligations. At the end of the loop body, after we exhaled the invariant, we must perform a leak check to ensure that no obligations are leaked.

We will now briefly explain how these different tasks are encoded, but we omit certain details as the techniques are essentially the same as before.

Leak Checks. Similar to the leak check performed when calling a method (Section 5.5), we need to assert that there are no obligations leaked before a loop is entered that does not promise to terminate. That means that, after the invariant has been exhaled before the loop, we may need to perform a leak check.

The second leak check needs to be performed after the loop invariant has been exhaled at the end of every loop iteration. This ensures that no obligation gets leaked throughout the loop body.

As with earlier leak checks, we encode them as part of the loop invariant. There is a problem however: the first one is optional and the second one has to be executed after every loop iteration. To distinguish between these situations we introduce a local variable that signals if we are inside the loop or outside:

```
var check :  Bool := false
while (𝓐)
   invariant 𝓘
{
   check := true
   𝓢
}
```

If the `check` variable is set, we execute the leak check and if it is not set, we execute the leak check only if no termination obligation has been exhaled. The following additions to the loop invariant will achieve exactly this:

```
[true, (!check && !terminates) → λ]
[true, check → λ]
```

Lifetime Functions. To ensure that the obligations mentioned in the invariant are not passed from iteration to iteration indefinitely, their lifetimes must decrease. To achieve this, we initialise the lifetime of each obligation in the loop. At the end of the loop body we perform the exhale operation with a lifetime check that compares to these lifetimes. This means that if a lifetime will not have decreased throughout the loop body, it cannot be exhaled and the leak check will fail.

Because we use functions to encode the lifetime, we need to use separate functions for *each* loop/method body. This is simply because we cannot assume different initialisations for a function. For example, if we were to use the same function to store the lifetime of bounded obligations for method bodies and loop bodies, we would execute both of these statements at some point:

```
inhale lifetimeMustSend(x) == t1     //start of the method
inhale lifetimeMustSend(x) == t2       //start of the loop
```

This means that we have `lifetimeMustSend` that maps one argument to two values. In particular, this is no longer a function which means the verifier is essentially inhaling `false`.

5.9 Send and Receive

Send. The encoding of the send operation is straightforward. There are three steps that need to be taken:

- Assert that the channel object is not null
- Exhale the channel invariant
- Exhale one *bounded* send obligation

Internally we use a lifetime of 0 for the exhaled obligation to make sure that the exhale operation will always succeed. Otherwise, if we hold obligations with a bounded lifetime of 1, the exhale would try to substitute with unbounded ones.

Summary Send. The following send statement

```
send c(x)
```

will be translated as follows:

```
assert c != null
exhale Exhale(Ic) //exhale channel invariant
exhale Exhale(mustSend(c, 1, 0))
```

Receive. To receive a message we need to perform the following steps:

- Assert that the channel object is not null
- Assert that we have at least one receive credit
- Exhale one receive credit
- Inhale the channel invariant

Note that we first assert that we have at least one receive credit before exhaling it. This is *indispensable* since it must not happen that the exhale statement produces an obligation afterwards (which would be unsound). Since we know that there is at least one credit, we can directly exhale it and there is no need to use the exhale macro described earlier.

Summary Receive. The following receive statement

```
x := receive c
```

will be translated as follows:

```
assert c != null
assert perm(mayReceive(c)) > none
exhale acc(mayReceive(c), write)
inhale Inhale(I_c) //inhale channel invariant
```

5.10 Acquire and Release

Compared to the original translation of the release and acquire statements we simply need to inhale or exhale an obligation, respectively. To summarise, the statement

```
acquire a
```

will be enhanced with the addition inhale statement

```
inhale acc(mustReleaseUnbounded(a), write)
```

that makes sure that the lock is released eventually. To handle the release

```
release a
```

we add the following exhale that makes sure that we get rid of the release obligation

```
exhale Exhale(mustRelease(a, 0)
```

Again we use a lifetime of 0 to ensure that the exhale will also consider bounded obligations.

6 Evaluation

In this section we will show larger examples and discuss how our solution can handle them. We will compare our solution to the existing one and discuss strengths and weaknesses of our design decisions.

6.1 Deviation From Original Scheme

In the original scheme, method preconditions and loop invariants could contain unbounded obligations. The lifetime of these obligations is assumed to be $\top$. Since our encoding uses integer measures, there is a problem with implementing this maximal value. The advantage of using the *mathematical* value $\top$ is that all lifetime checks automatically succeed since every other element in the lattice is, by definition, smaller.

One possibility would be to use a special value, say -1, to simulate this top value. This would be possible since the integer measures are always positive and hence there would be no conflicts. However, it seems a bit cumbersome since we would then have to explicitly cover this possibility in every lifetime check. In practice, this restriction does not pose any problems when expressing programs.

6.2 Comparison to Earlier Version

In this paragraph we will compare our encoding to an earlier project ([Kla14]) that implemented obligations, but originally relied on different theoretical assumptions.

Producer-Consumer. Consider the example shown in Figure 26 that shows a classical producer-consumer scenario. The channel `C` is used to send the data and the necessary credits to receive further messages. If the producer decides to terminate, it can simply send `next = false` over the channel which signals the consumer that no more data will be available and simultaneously prevents it from trying to receive messages.

In previous versions it was not possible to verify this example because of the loop invariant in the producer method. In our encoding this is no longer a problem since we have completely decoupled the encoding of credits and bounded obligations. At the end of the loop body we will have one fresh send obligation that gets converted to a bounded one when the loop invariant is exhaled and the leak check succeeds.

Well-Formedness Check. Previously it was necessary to explicitly perform well-formedness checks at the Silver level. The reason for that was that the method specification was encoded manually, meaning that a Silver method of the form:

```
channel C(x:int, next:bool) where next ==> mayReceive(this, 1);

class A {

    method consumer(c:C)
        requires c != null
        requires mayReceive(c, 1)
    {
        var x : int
        var running : bool

        receive x,running := c

        while (running)
            invariant running ==> mayReceive(c, 1)
        {
            receive x,running := c
        }
    }

    method getNextNumber(x:int)
        returns (y:int)
        requires mustTerminate(1)
    {
        y := x + 2
    }

    method producer(c:C)
        requires c != null
    {
        var next : int := 0
        var running : bool := true

        fork consumer(c)

        while (running)
            invariant running ==> mustSend(c, 1, 1)
        {
            call next := getNextNumber(next)

            if (next % 2 == 1) {
                running := false
            } else {
                running := true
            }

            send c(next, running)
        }
    }
}
```

Figure 26: Example: Producer Consumer

```
method f()
   requires α
   ensures β
   { S }
```

was transformed to:

```
method f()
   requires true
   ensures true
{
   inhale α
   S
   exhale β
}
```

If a method was called, the pre- and postconditions were explicitly exhaled or inhaled, respectively. This made it possible to verify methods of the following form

```
method f()
   requires acc(x.f) //assuming x.f is an integer field
   ensures x.f >= 5
   { S }
```

This is problematic since the postcondition is not 'self-framing', meaning that a caller cannot inhale it because it does not contain the necessary access permission to `x.f`. To avoid these problems the well-formedness checks were manually encoded in the Silver program by the front-end.

However, encoding the well-formedness check at the Silver level is difficult and imposes additional restrictions on the programmer. For example it was necessary to repeat certain properties in the postcondition. Consider the example shown in Figure 27. As a consequence of the implementation of the check, the knowledge `n >= 0` was lost in the postcondition of method `fibSeq` and needed to be repeated.

Silicon automatically performs this well-formedness check for pre- and postconditions. Since our encoding fully uses the method specification to encode the transfer of obligations, we need not worry about self-framing explicitly. This makes it possible to handle examples that were previously not verifiable.

Handling Predicates. Similar to the earlier implementation, we suffer from the same limitation when an obligation is mentioned inside an unfolding expression. Consider the example in Figure 28. The idea is to perform some work on a node in a binary tree while two worker threads recursively work on the child nodes. To enforce termination we use the level of the tree, denoted by `height`. We have to use the same trick ([Kla14]) to avoid problems with unfolding predicates by using a separate parameter that is equal to the current height of the node (line 20). Otherwise the termination obligation on line 21 would have to be inside an `unfolding` expression.

Aliasing Release Obligations As described in 5.1, our encoding implicitly excludes the possibility of aliasing release obligations. Consider the example shown in Figure 29. If `a` and `b` were aliases, the call statement on line 17 would

```
class Test {
    var f: int;

    function fib(n: int): int
        requires n >= 0
    {
        n < 2 ? n : fib(n - 1) + fib(n - 2)
    }

    method fibSeq(n: int) returns (r: int)
        requires n >= 0
        requires acc(this.f)
        ensures acc(this.f)

        //previous error: n >= 0 had to be repeated
        ensures r == fib(n)
    {
        if (n < 2) {
            r := n
        } else {
            var f1: int; var f2: int
            call f1 := fibSeq(n - 1)
            call f2 := fibSeq(n - 2)
            r := f1 + f2
        }
    }
}
```

Figure 27: Chalice2Silver Test 'workitem-10200.chalice': Well-Formedness Check Succeeds

```
class Tree {

    var left : Tree
    var right : Tree
    var height : int

    predicate valid {
        acc(left) && acc(right) && rd(height) && height >= 0 &&
        (left != null ==>    left.valid  &&
                             rd(left.height) &&
                             left.height == height -1)   &&
        (right != null ==>  right.valid &&
                             rd(right.height) &&
                             right.height == height -1)
    }

    method work(callHeight : int)
        requires valid
        requires callHeight >= 0
        requires unfolding valid in height == callHeight
        requires mustTerminate(callHeight+1)
        ensures valid
    {
        var t1 : token<Tree.work>
        var t2 : token<Tree.work>

        if (callHeight > 0) {

            unfold valid

            if (left != null) {fork t1 := left.work(callHeight-1)}
            if (right != null) {fork t2 := right.work(callHeight-1)}
            //work
            if (left != null) { join t1 }
            if (right != null) { join t2 }

            fold valid
        }
    }
}
```

Figure 28: Parallel Tree Processing [Kla14]

```
class A {

    method rel(a : A)
        requires a != null
        requires mustRelease(a, 1)
    {
        release a
    }

    method f(a: A, b : A)
        requires a != null
        requires b != null
        requires mustRelease(a, 1)
        requires mustRelease(b, 2)
    {
        release a
        call rel(b)
    }
}
```

Figure 29: Aliasing Release Obligations

```
channel C(x:int);

class A {
    method f(c:C)
        requires c != null &&  mustSend(c, 1, 1)
    { }
}
```

Figure 30: Leaking Obligation

not be allowed: the meet of the lifetimes is 1 and hence the measure would not decrease as the obligation is passed to method `rel`. However, since we know that we cannot hold a lock multiple times, we can exclude the possibility of `a` and `b` being aliases.

The original encoding did not automatically exclude this possibility which means that the verification of this example would lead to an error. It is important to note however, that this does not make the original analysis unsound, it is simply imprecise in certain situations.

6.3 Error Messages

In this paragraph we will shortly discuss the error messages that are generated when using our encoding of obligations.

Consider method `f` shown in Figure 30. The method receives a send obligation from the caller but fails to fulfil it which will be caught by the leak check. If we try to verify this method we will get the following error message:

```
class A {

    method g(a : A)
        requires a != null && mustRelease(a, 1)
    {
        release a
    }

    method f(a : A)
        requires a != null && mustRelease(a, 1)
    {
        call g(a)
    }
}
```

Figure 31: Fail of Lifetime Check

```
Postcondition of Af might not hold.  Assertion
   (forallrefs [mustSendBounded] r ::  false)
might not hold.
```

The error occurs in the verification of the postcondition where the leak is encoded (Section 5.5). For the careful reader of this report, this message might be meaningful since this assertion was presented in Section 5.3. However for a person unfamiliar with the implementation details of our encoding, this error message provides few pointers as where to begin the debugging process.

Consider the example shown in Figure 31. Method `f` tries to pass its release obligation to method `g` but this is impossible since the lifetime would not decrease during this transfer. When we try to verify this example we get the following error message:

```
The precondition of method Ag might not hold.  Assertion
   1 * write - perm(a.mustReleaseUnbounded) == none
might not hold.
```

This error message is produced by the third stage of the exhale macro (Figure 23, page 36, line 7). Since the lifetime check fails in the first stage, the second stage tries to substitute with the unbounded field (Figure 22, page 36, line 4) and fails too, since there is no unbounded obligation. From the point of view of the macro, all remaining obligations produce credits. However, since there are no 'release credits' ($cf == null$), the third stage tries to assert that there are no remaining obligations and fails.

Compared to the first example, this error message is even worse for the user since it inadvertently draws attention to unbounded obligations, even though there appear no such obligations in the example.

Both examples show one major drawback of using Silver mechanisms to simulate a high-level concept such as obligations. If an error is discovered, it is reported in terms of the Silver statement that is failing. However, for a user it might be hard to draw a connection between this 'low level' error message and the error in the higher level concept.

```
channel C(x : int)

class A {
    method g(c : C)
        requires c != null && mustSend(c, 1, 5)
        requires mustTerminate(1)
    { /* ... */ }

    method f(c : C)
        requires c != null
        requires mustSend(c, 1, 1)
    {
        call g(c)
    }
}
```

Figure 32: Unsoundness Due to Cancellation of Obligations and Credits

Of course this design has the big advantage of keeping the intermediate language as thin as possible. This greatly facilitates the design and maintenance of the complete verification infrastructure.

6.4 Cancellation

The cancellation of credits and obligations is forbidden because it is unsound as the example in Figure 32 suggests. The call statement on line 13 produces a credit since we exhale an obligation with a larger measure (5) than the one we currently have (1). If cancellation were allowed we could use this credit to remove the obligation in `f` that was inhaled with the precondition. This means we have found a way to replace our old obligation with another one that has a larger measure. Consequently we can arrange a sequence of method calls in such a way that an obligation is passed from one method to the next, without ever getting satisfied.

It is possible to avoid this unsoundness by allowing situations where the obligation that gets cancelled has a strictly larger measure than the measure of the exhaled obligation that created the credit in the first place [BM15]. Since this is rather impractical and requires large additional effort, cancellation is simply forbidden in general.

There are situations where cancellation would be desirable as shown in Figure 33 and Figure 34. The idea is the following: we have two threads, a producer and a consumer that can communicate with each other by using two channels of different types. The consumer, shown in Figure 33, continuously asks the producer for new data (line 28) and in doing so, creates a send obligation for itself. When the data is received the consumer automatically gets a fresh permission for the next message.

If the consumer decides to quit the execution, modelled with method `enough()`, it simply sends the receive credit back over the channel on line 35 and then stops its execution.

```
channel C(x : int) where mayReceive(this, 1);
channel D(c:C)
    where (c == null ? mayReceive(this, 1) : mayReceive(c, 1));

class Consumer {

    method enough() returns (b:bool)
        requires mustTerminate(1)
    {
        b := false
    }

    method consumer(input : C, output : D)
        requires input != null && output != null
        requires mustSend(output, 1, 1)
        requires mayReceive(input, 1)
    {
        var run : bool := true
        while (run)
            invariant run ==> mustSend(output, 1, 1)
            invariant run ==> mayReceive(input, 1)
        {
            //continue?
            call run := enough()

            if (run) {
                //signal producer for more data
                send output(null)

                //get data
                var d : int
                receive d := input
            } else {
                //signal end of run
                send output(input)
            }
        }
    }
}
```

Figure 33: Producer-Consumer With Cancellation, Part 1

```
channel C(x : int) where mayReceive(this, 1);
channel D(c:C)
    where (c == null ? mayReceive(this, 1) : mayReceive(c, 1));

class Producer {

    method producer(input : D, output : C)
        requires input != null && output != null
        requires mayReceive(input, 1)
        requires mustSend(output, 1, 1)
    {
        var run : bool := true
        while (run)
            invariant run ==> mustSend(output, 1, 1)
            invariant run ==> mayReceive(input, 1)
        {
            //check if we need to produce
            var answer : C
            receive answer := input

            run := (answer == null)

            if (run) {
                //produce data
                send output(151)
            } else {
                if (answer == output) {
                    //the credit cancels our obligation
                    send output(0) //cancellation would be nice
                } else {
                    //error! cannot cancel obligation
                    assume false
                }
            }
        }
    }
}
```

Figure 34: Producer-Consumer With Cancellation, Part 2

<table>
<tr><td rowspan="4">Existing Test Suite</td><td rowspan="4">353</td><td rowspan="2">Not Related</td><td rowspan="2">304</td><td>unchanged</td><td>292</td></tr>
<tr><td>changed</td><td>12</td></tr>
<tr><td rowspan="2">Previous Project</td><td rowspan="2">49</td><td>changed</td><td>38</td></tr>
<tr><td>broken</td><td>11</td></tr>
</table>

Table 1: Overview of Adaptation of Existing Test Suite

The producer, shown in Figure 34, produces more data on line 25 if needed. As soon as the consumer signals the end of its execution, the producer has received its own receive credit back over the input channel. In this situation it would be convenient (and safe) to cancel the credit with the obligation. However, since cancellation is not allowed, we need to send a dummy message on line 29.

Note that we need the if statement shown on line 27. Since the verifier has no knowledge about the reference the method receives, we need to insert some form of error handling. In this example we simply ignore it by inhaling `false` on line 32.

6.5 Existing Test Suite

To test our implementation we added 181 new methods in 31 files to the test suite. Some tests of the existing test suite had to be adapted slightly to handle the new analysis. For example we often encountered release obligations that were leaked at the end of the method. Most often these tests were added to test future implementations, for example, in combination with waitlevel violations. Since this analysis is not yet implemented, the test cases often failed due to leaks and we could simply add the corresponding annotations. Most of these unrelated tests could be left unchanged, however. Table 1 shows a summary of the changes we made.

The existing project already provided many tests that could easily be adapted for our purposes. These were mostly syntactic changes, since the previous project used a different notation to express obligations in Chalice. Additionally we had to swap error messages since the encoding is different. By far the most common changes we made had to do with the transfer of obligations. Since we encoded everything in the specification, our error messages were related to the invariants, pre- and postconditions rather than explicit in- and exhale statements that were previously used to explicitly simulate the specification.

Some tests from the previous project had to be ignored completely and are marked as "broken" in Table 1. Frequently this was because in our encoding it is not allowed to transfer release obligations to other threads, a restriction that was not in place in the original analysis. Some errors were also related to aliasing release obligations. As mentioned in 5.1, our encoding implicitly forbids the possibility of aliasing release obligations and hence does not suffer from this problem.

7 Conclusion

We have presented an encoding of obligations for the Viper verification framework. Compared to earlier solutions we were able to make use of more Silver mechanics and remove several extensions which is important to keep the intermediate language as lean as possible. The removal of extensions results in a more complex encoding, a cost that is ultimately paid by the user when dealing with the generated error messages that require a detailed understanding of the implementation. Overall we were able to make the analysis more accurate (aliasing release obligations) and lift restrictions that often forced the programmer to come up with crafty workarounds (producer-consumer).

7.1 Future Work

Deadlock Detection. The most important next step is to add deadlock detection to the analysis. Obligations by themselves are not able to completely proof finite blocking. Consider the two methods shown in Figure 35. Since method `f` terminates, we can call it inside of `g` and retain some obligations. Inside `f` we try to acquire a lock on the argument but this will of course fail and we have a deadlock.

There already exists a solution [BM15] to this problem and in combination with the so-called 'waitlevels' of Chalice this would be a viable addition to the verification infrastructure.

Arrays. Arrays are of course omnipresent in computer programming and so far we have not addressed them at all. Depending on how obligations for array elements are ultimately encoded we might run into problems when initialising the lifetime functions. As the algorithm is presented so far, it cannot handle an unspecified number of references, as they might appear in arrays, since we are generating code statically.

Behaviour of Leak Checks. As mentioned in Section 3.3, adding leak checks to the end of method bodies leads to a direct violation of the Rule of Consequence. Since this rule is of such importance and leak checks are of course an integral part when dealing with obligations, it seems reasonable to further investigate the interaction between obligations and Hoare Logic. Precisely because leak checks are indispensable for obligations it is important to know how this verification technique can be soundly combined with other techniques, or if this does not affect other analyses.

Error Handling. The implementation of obligations as presented in this report is an encoding in the truest sense of the word. We take available Silver concepts and use them differently as they were originally intended to simulate our desired behaviour. As a consequence, the generated error messages are very unintuitive and it requires a detailed understanding of the implementation to fully understand them.

Since one of our goals was to use as few Silver extensions as possible, this is at first unavoidable. However it would make the verification process more user

```
class A {

    method f(a:A)
        requires a != null
        requires mustTerminate(1)
    {
        acquire a
        release a
    }

    method g(a:A)
        requires a != null
    {
        acquire a
        call f(a)
        release a
    }
}
```

Figure 35: Deadlocking Methods

friendly if there was a way to add some sort of error message to Silver statements. For example, one could think of a construct similar to a try-catch block:

```
execute {
   𝒮
} alert {
   "Error in block 𝒮"
}
```

Where all statements in the `execute`-block are executed normally and if an error occurs in $\mathcal{S}$, the error message in the `alert`-block is printed.

Such a mechanism would be very useful, not just for this work, but also for any other encoding that uses Silver concepts to simulate some behaviour.

ForallReferences Expression. As mentioned in Section 4.1 we enhanced the `forallrefs` expression to enable predicates in the parameter list. The restriction we imposed on the predicates was that they had to take one parameter of type `Ref`. To make the `forallrefs` expression more user friendly this restriction could be lifted in the future. Say we had the following predicate:

```
predicate p(r : Ref, i : Int, b : Bool) {...}
```

One possibility to allow the iteration over such predicates could be to use one bounded variable that is essentially a place holder for a heap chunk. This bounded variable could then be used to refer to the arguments of the predicate:

```
forallrefs [p] x :: x.r != null && r.i < 5 && r.b
```

This possibility only works if we iterate over one argument at a time because otherwise the argument names and types could lead to conflicts.

Another approach might be to restrict the arguments in advance and use some sort of pattern matching to filter the applicable heap chunks:

```
forallrefs x [p(x, 7, false)] ::  x != null
```

which would only consider references `x` that appear in a predicates `p` where `i == 7` and `b == false`.

Both possibilities are just sketches and there needs to be more work to discern which solution would be more useful. *Side note:* both ideas are appealing for the Silicon verifier, since they fit quite naturally into its heap representation. An implementation for Boogie might not be as intuitive.

Acknowledgements

First and foremost I would like to thank Prof. Dr. Peter Müller for supporting me throughout this project and for many helpful discussions. Additionally I would like to mention the welcoming atmosphere of the complete Programming Methodology group which made the work on this project immensely enjoyable. In particular I would like to thank Malte Schwerhoff for his patience and many helpful explanations of the internal workings of the Silicon verifier. Special thanks go to my family and friends for their continued support throughout this period.

References

[BCD+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.

[BM15] P. Boström and P. Müller. Modular verification of finite blocking in non-terminating programs. In J. Boyland, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer, 2015.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[JKM+14] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.

[Kla14] Christian Klauser. Modular verification of finite blocking. Master's thesis, ETH Zürich, 2014.

[LMS09] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer-Verlag, 2009.

[LMS10] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In A. D. Gordon, editor, *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, 2010.

[Sch11] Malte Schwerhoff. Symbolic execution for chalice. Master's thesis, ETH Zürich, 2011.

[WTE05] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.

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):

Verification of Finite Blocking in Chalice

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Meier	Robert

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)
Zumikon, 2.9.2015	R. Meier

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