Modular Verification of Finite Blocking

Master's Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zürich
www.pm.inf.ethz.ch

By: Christian Klauser
klauserc@student.ethz.ch

Supervised by: Prof. Dr. Peter Müller
Dr. Ioannis Kassios

Date: November 30, 2014

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Chalice is a program verifier for concurrent programs that has built-in support for advanced constructs often found in multi-threaded programs, such as asynchronous method calls, channels and monitor locks. But even though it features a sophisticated deadlock prevention mechanism, it only proves partial correctness, that is, it will accept programs that block forever.

In this thesis, we present a verification technique introduced by [BM14] that aims to show that either all threads in a program terminate or run forever. In other words, to show that no thread blocks forever. We have implemented the part of it that deals with termination on the Viper verification infrastructure. Since Silver, the intermediate language used in Viper, is not expressive enough for that endeavour, we have designed and implemented a handful of extensions to Silver that have the potential to be useful beyond the application of this thesis.

Contents

1 Introduction

Multi-threaded programs typically need some form of synchronization. Mutual exclusion locks, condition queues, plain old semaphores but also waiting for a message or signal to arrive are all commonly used synchronization mechanisms. One thing they have in common is that they force a thread to wait until a certain condition is fulfilled by other threads (lock released, message sent). While waiting, a thread cannot make progress. We call such a thread *blocked*. It needs other threads to *unblock* it by releasing a lock they were holding onto or by sending on the correct channel. Trouble follows when a thread that is supposed to unlock another is itself blocked. Worst case, when two or more threads form a *blocked-by* ring, we are in a deadlock situation where none of the threads will ever make progress.

Approaches to avoid the starvation of any one thread in a *terminating* program have been around for some years [Kob06, LMS09] where we ensure that thread scheduling is fair and no the program does not contain deadlock cycles. Unfortunately, this doesn't apply to many programs found in the real world because they don't necessarily terminate. Server processes or interactive applications as a whole typically only terminate based on outside input. Sometimes also just parts of an otherwise terminating program, such as threads dedicate to logging or sending database queries, operate in a perpetual fashion. When such a non-terminating thread blocks another thread, it could potentially postpone the unblocking operation indefinitely. A logging thread, for instance, could hold a lock to the log file handle until it receives a signal to terminate. If any other thread tries to acquire that lock, that other thread might never be able to proceed, even with completely fair scheduling. Since existing tools assume that all threads terminate, they are unable to detect this scenario as an error.

What we would like is a way to verify that, although parts of a program potentially don't terminate, at least no thread remains blocked forever in a system with fair scheduling. In other words, we want each thread in a program must either terminate (without being blocked forever) or run forever. We call this property *finite blocking*, a term introduced by [BM14]. The same paper also introduces a verification technique for finite blocking where each operation that potentially blocks another thread comes with an *obligation to unblock* that thread. For instance, when a thread acquires a lock, it simultaneously obtains an obligation to release that lock within a finite amount of time. While obligations are present in a thread, that thread can no longer call methods or enter loops that don't promise to terminate. Or at least not unless it can show that the obligations will be fulfilled within the allotted time frame.

The authors of [BM14] provide an encoding of their verification technique for the Boogie verifier [BDJ+06]. In this Master's thesis we present an encoding of their technique for the Viper verification infrastructure [JKM+14]. Where necessary, we have extended Viper with constructs that make our encoding possible but are designed to be useful and sensible additions beyond just verifying finite blocking.

Outline We start in section 2 with an introduction to the Viper verification infrastructure and the research programming language Chalice that we extend to support obligations. Section 3 continues with a more detailed presentation of the verification scheme in [BM14] and its informal semantics. In section 4, we trace the design process that led us to the final set of extensions to Viper that we present in section 5. We describe our encoding of the modular verification of finite blocking using our extensions in section 10. In section 7 we evaluate our translation with a handful of programs and conclude in section 8.

2 Background

In this section we first give an overview over the Viper verification infrastructure and then quickly present Chalice, a research programming language that will serve as our "source" language.

2.1 Viper

The Viper verification infrastructure is a set of tools for static program verification using permission-based verification techniques [JKM+14]. It is centred around its own intermediate language "Silver". As a relatively high-level verification language, Silver comes with built-in ideas of a memory heap and permissions. To have a heap as a native concept means that tools operating on Silver programs have considerably more freedom in how they model their execution.

There are currently two verifier backends for Silver. One, "Carbon", is based on verification condition generation (VCG) that emits code for the Boogie program verifier [BDJ+06]. The other, "Silicon", based on symbolic execution. Both backends ultimately use the Z3 SMT solver [dMB08].

2.1.1 Silver

Silver is an intermediate language intended for program verification. It is designed to be as simple as possible while still supporting high-level concepts such as a heap and permissions typically absent from verification languages. For example, even though Silver is designed with object-oriented "source" languages in mind, it doesn't differentiate between different types of objects (that's the domain of the source language's type checker). Indeed one doesn't specify classes in Silver, just methods and fields. Similarly, methods don't have an implicit **`this`** parameter. It's up to the frontend to include it if necessary.

One thing that differentiates Silver from many other verification languages, is its permission system. In order to access a heap location in Silver, the current thread needs to have *permission to access* that location. Permission is initially acquired when the object is created but subsequently needs to be explicitly passed from method to method, from thread to thread and from loop iteration to loop iteration. Permission can also be split up and distributed across multiple threads, but a fractional permission only grants reading access to memory locations. A thread requires full permission to modify a heap location.

Let's go through the very small example in listing 1. It contains two top-level definitions: the object field `a` of type `Int` on line 1 and the method `validate` starting on the next line.

Listing 1: Simple Silver program that makes sure the field `a` is non-negative.

```
1  field a:Int
2  method validate(x:Ref) returns(valid:Bool)
3    requires x != null && acc(x.a, write)
4    ensures acc(x.a, write) && x.a >= 0
5    ensures valid ==> x.a == old(x.a)
6  {
7    if(x.a < 0){
8      valid := false;
9      x.a := 0;
10   } else {
11     valid := true;
12   }
13 }
```

The method has one input parameter `x` and one output parameter `valid`. The value `x` is of type `Ref` to represent a reference to an object on the heap. References don't have a more specific type. Thus, as far as Silver is concerned, the object pointed to by `x` may or may not have a field called `a`. Lines 3 through 5 contain the method's *specification* as pre- and postconditions. Silver is intended to be verified modularly on a per-method level, that is, each method should be verified on its own without considering other method's implementations.

Preconditions in Silver don't just make assertions about the program state at the point where a method is called, they also describe the set of memory locations that a method will need to access during it's execution. On line 3, our method specified that it **`requires acc`**`(x.a, `**`write`**`)`, that is, it requires read-write access to the field `a` on object `x`. Write access is highest level of access you can have to a memory location. It is thus often referred to as a *full* access permission.

To prevent data races, there is never more than one full access permission to a location in the entire system. This means that if a method has **`write`** access to a heap location, it can be sure that no other thread can write or even observe that location at that time. If a thread only has a fraction of the full permission, we call this *read* access. That thread can be sure that, while it holds this fraction, no thread can write to the heap location. Although not obligatory, method's typically want to return the permissions they received on method entry back to their caller at the end. This happens via the specification of **`ensures acc`**`(x.a, `**`write`**`)` on line 4.

In the example in listing 2, we have a program fragment that might have been generated by a frontend that wishes to verify a programming language with explicit memory management. It defines a method `free` that takes an object and access to the object's fields (in our case just **`acc`**`(x.f, `**`write`**`)`) and drops them. In the `main` method, we create a new object with one field, `a`. After a bit of use, we call `free` on line 9. To the verifier `x.a` is as good as non-existent when the call returns, as we have lost *all* permission to it. When we then try to call the `validate` method from before, the Silver verifier will complain will report the following error

```
The precondition of method validate might not hold. There might be
    insufficient permission to access x.a. (program.sil,11:3)
```

Listing 2: Silver example that encodes a program with exhibits a "use-after-free" problem caught by the verifier.

```
1  method free(x: Ref)
2    requires x != null && acc(x.a, write)
3  { /* free memory */ }
4
5  method main(){
6    var x : Ref
7    x := new(a)
8    x.a := -6
9    free(x)
10   var b: Bool
11   b := validate(x)
12 }
```

To understand what exactly is going on here, we need to look at the method call to `free` on line 9 in more detail. When we want to verify a method call, we first need to make sure we satisfy its precondtion. Then the actual call "happens" but since we are verifying Silver programs modularly on a method-by-method basis, the only thing we can rely on, is the method's postcondition. Thus, you'd typically get to *assume* that postcondition, once you have shown that you satisfy the precondition. This would look as follows:

```
assert x != null && acc(x.a, write) // precondition of free
                                    // call "happens" here
assume true                         // postcondition of free (it's empty)
```

But this is not quite correct, since the **`assert`** statement in Silver, as in most other languages, does not modify the program's state. Silver has a separate pair of statements for just this purpose: **`inhale`** and **`exhale`**. When used instead of **`assume`** and **`assert`**, respectively, the verifier treats accessibility assertions (**`acc(x.a, write)`**) as an instruction to transfer the corresponding amount of permission into or out of the current method. Our call to `free` thus translates to

```
exhale x != null && acc(x.a, write)
inhale true
```

So when the verifier comes across the **`exhale`** statement, it first asserts that `x` = null!, then checks that we currently have at least **`write`** permission to `x.a` before it subtracts one **`write`** from our current permission to `x.a`. That just happens to be all permission we had to that memory location so from now on it might as well no longer exists, as far as we are concerned. After all if we don't have any permission to it, it is possible that some other thread has write access to it and could thus be changing it constantly. Once we reach the call to `validate` on line 11, we no longer have enough permission to satisfy **`acc(x.f, write)`** in that method's precondition.

2.1.2 Silicon

Silicon is one of two verifier backends for Silver. We implement the verification of our extensions to Silver as part of Silicon. While we could alternatively have implemented them in Carbon, the authors of [BM14] already present an encoding for Boogie, the same program verifier that Carbon uses. It is much more interesting to implement the extensions in a verifier that is *not* also using VCG.

Silicon is an adaptation of an earlier symbolic execution based verifier for Chalice [Sch11]. The way Silicon works is very different from VCG-based verifiers. The latter essentially encode entire methods into big logical formulas that they then pass to a theorem prover (Z3 in the case of Carbon). These formulas, in particular, also contain a representation of the various states of the program's heap.

Silicon, on the other hand, maintains both the local variable store, mapping from local variable names to symbolic terms, and the symbolic heap itself. Silicon represents the heap as a set of "heap chunks" [Sch11]. A heap chunk for a field is a quadruple

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

where

r : symbolic term, represents an object reference
f : field identifier
t : symbolic term, represents value of the field
p : symbolic term, represents amount of permission

As an example, when Silicon is confronted with the following snippet of Silver:

```
1 x := new()
2 inhale acc(x.a, write)
3 exhale acc(x.a, write/2)
```

On the first line, we allocate a new object. Silicon invents a new object identifier for it, say "x_1", and stores the assignment to the local variable x in its local variable store γ:

$$\gamma = [x \mapsto x_1]$$

On the next line, we inhale an access permission. Silicon turns the accessibility assertion into a heap chunk ($x_1.f \mapsto t_1\#\mathbf{write}$) using a fresh term symbol t_1. This chunk is then merged into Silicon's heap h, which now looks like this:

$$h = [x_1.f \mapsto t_1\#\mathbf{write}]$$

When we reach line 3, Silicon can directly look up the chunk for `x_1.f` in its heap because the receiver and field match literally. Silicon then needs to check whether we currently hold enough permission to perform the exhale. This, however, it cannot do on its own, so it asks the theorem prover whether ($\mathbf{write}/2 < \mathbf{write}$) holds. If that check was successful, it replaces the chunk for $x_1.f$ with the following chunk

$$x.f \mapsto t_1\#\,(\mathbf{write} - \mathbf{write}/2)$$

In general Silicon never passes its symbolic heap representation to the theorem prover. All the prover ever sees of the heap, are the symbolic object identifiers generated by Silicon.

To give the prover enough information to reason about the assertions passed to it, Silicon maintains a list of "path conditions" that the prover gets to assume. Path conditions are *facts* that Silicon learned along the execution path: logical formulas in the precondition of the method, or postconditions of called methods for instance.

One situation that Silicon needs to handle with particular care, is when multiple otherwise distinct terms are aliases for the one object. To compute the total amount of permission for fields on that object, Silicon will have to perform a *compression of the heap*. It has no other choice than to go through all chunks for the field in question pairwise and ask the prover if their receivers are the same. If so, it replaces the pair of chunks by a new chunk with the summed up permission amount.

2.2 Chalice

Chalice [LMS09] is a research programming language created specifically to study the verification of concurrent programs. While there was a compiler for Chalice at one point, it is primarily intended for verification. Chalice was the first language to be verified via Silver [Kla12]. In a sense, Silver is a lower level, more distilled version of Chalice, rather than a higher level alternative to, say, Boogie.

2.2.1 Fork-Join

Chalice has some features that are not present in Silver. One of them is a notation for calling methods asynchronously. In the example program in listing 3 we have a `main` method that spawns two worker threads. These threads operate on shared data in the form of `x.f`. With the accessibility predicate `acc(x.f, 10)` on line 5, we specify that an invocation of the method `work` requires *ten percent* of a full permission. The method `work` can therefore only read from `x.f`, but not write to it.

The **`fork`** statements create a new thread that executes the specified method. They don't block until that thread has finished work, however. Instead, they return a token that one can later use to wait for, to **`join`** that thread. A token is a first-class value that can be stored in local variables, returned from methods and passed as arguments. It cannot, however, be joined multiple times. This is because, as in our example, joined threads return permission to whoever joins their token. If it were somehow possible to join a thread multiple times, the returned permission would be duplicated.

2.2.2 Monitor Locks

Chalice also as built-in mechanisms for modelling synchronization via monitor locks. Unlike the built-in lock statements in Java and C#, which just serve as a means to ensure mutual exclusion, monitor locks in Chalice are typically associated with a *monitor invariant*. The idea is that some time after an object is initialized a thread can give the object into the care of a central monitor. This monitor requires the thread to prove that it satisfies the monitor invariant and transfer all necessary permission to the monitor. From that point on, other threads can **`acquire`** a lock on the object. Once they have the lock, they immediately get the permissions "stored" in the monitor invariant. The lock on an object can only be released when the monitor invariant can be established.

Listing 3: Chalice example of forking and joining threads.

```
method work(x: Obj, n: int) returns(s: int)
  requires x != null && acc(x.f, 10) && n >= 0
  ensures acc(x.f, 10) && s == n*x.f
{
  var i := 0; s := 0;
  while(i != n)
    invariant acc(x.f, 10) && s == x.f*i
  {
    s := s + x.f;
    i := i + 1;
  }
}

method main(){
  var s1: int; var s2: int;
  var x: Obj := new Obj { f := 5 };
  fork t1 := work(x, 3);
  fork t2 := work(x, 2);
  // do other work
  join s2 := t2; join s1 := t1;
  assert s2 == 2*5 && s1 == 3*5;
  x.f := 0;
}
```

Listing 4: Chalice example with a monitor invariant. Deadlock avoidance omitted.

```
class Obj {
  var f: int;
  var g: int;
  invariant acc(f) && acc(g) && g == -f;

  method main() {
    var x := new Obj { f := 5 , g := -5 };
    share x;
    fork inc(x);
    fork inc(x);
  }

  method inc(x: Obj)
    requires x != null
  {
    acquire x;
    x.f := x.f + 1;
    x.g := x.g - 1;
    release x;
  }
}
```

We provide an example of how monitor invariants can be used in listing 4. There, the two fields on the `Obj` class must fulfil the condition `(g == -f)` while no thread holds the lock. We spawn two threads that *both* manipulate the memory locations `x.f, x.g`, something that wouldn't be possible using just the permission mechanism. Here, when we execute line 16, we receive the contents of the monitor invariant, which includes full access to both `x.f` and `x.g`. While we are holding the lock, it is fine to break the monitor invariant, since no other thread will be able to acquire the lock at the same time and observe us. On line 19, however, when we want to release the lock again, Chalice will ensure that the monitor invariant holds and that the access permissions are transferred back to the monitor invariant.

2.2.3 Predicates

Method pre- and postconditions can become pretty large and are often very similar for methods of the same class. Chalice offers a way to abstract away a set of conditions and permissions in a *predicate*. A thread can **`fold`** a predicate and receive an access permission to the predicate in exchange for satisfying the predicate's condition *and* giving away the permissions mentioned in the predicate's definition. The opposite operation, **`unfold`**, exchanges the permission to the predicate with the predicate's definition. In the example in listing 5 we have a predicate called `valid` that encapsulates the same condition as the monitor invariant in the previous example. Before we call the method `inc2`, we need to **`fold`** the predicate on line 19. The body of `inc2` looks very similar to the previous example.

One very important difference between predicates and monitor invariants, is that while the contents of the predicate are folded away, they still under the control of the thread that holds the predicate's permission. Once a lock is released, another thread could have immediately snatched it up and modified the object's state. With predicates, we can still reason about the state of the object by "temporarily **`unfolding`**" the predicate in pre-, postconditions and assertions.

2.2.4 Channels

To allow users to model programs with message passing, Chalice offers **`channel`**s over which one can **`send`** and **`receive`** messages. Listing 6 defines a channel type `Chan` which carries messages that consist of one object of type `Obj`. A message could consist of more than one value, just like method return values. The channel definition also features a *channel invariant*. This invariant about the message must be satisfied when you send that message and can be assumed when the message is received. Permissions mentioned in the channel invariant will be transferred away together with the message during a **`send`** operation and will be delivered during a **`receive`** operation.

Naturally, a **`receive`** operation can only succeed if some thread actually sent a message on the channel. Therefore, Chalice requires that you have *"channel credit"* when you try to receive messages. We denote this using the `credit(ch, _)` annotations on lines 14 and 20. Note how we send the object `y` across the channel twice. This is absolutely fine, since each message only requires ten percent worth of access permission to the `f` field of the message object.

Listing 5: Chalice program with predicates.

```
class Obj {
  var f: int;
  var g: int;
  predicate valid {
    acc(f) && acc(g) && g == -f
  }
  method inc2()
    requires acc(valid)
    ensures acc(valid)
      && unfolding valid in f == old(unfolding valid in f) + 1
  {
    unfold valid;
    f := f + 1;
    g := g - 1;
    fold valid;
  }
  method main(){
    var x : Obj := new Obj { f := 5, g := -5 };
    fold x.valid;
    call x.inc2();
    assert unfolding x.valid in x.f == 6;
  }
}
```

2.2.5 Deadlock Prevention

One language element omitted from listings 4 and 6 is Chalice's *deadlock prevention* mechanism [LMS10, LMS09]. The basic idea is that we establish a *"locking order"* where each object that you can wait on via **acquire** or **receive** has a *lock level*. When a thread tried to acquire a lock with lock level μ, it can only do so, if this lock level is above the lock levels of all other objects it holds a lock to. Thus, if create two objects as follows:

```
var x : Obj := new Obj;
share x;
var y : Obj := new Obj;
share y above x;
```

we can only ever lock x before y. We can of course lock y directly, but the deadlock prevention mechanism will not allow us to *then also* lock x.

```
acquire y;
acquire x; // error: target of the acquire might not be above waitlevel.
```

2.2.6 Chalice2Silver

Chalice is a program verifier in and of itself. It generates a Boogie program has the Boogie verifier verify it. At the same time, we have a tool that uses Chalice's parser to read Chalice

Listing 6: Chalice program using channels. Deadlock avoidance omitted.

```
channel Chan(x: Obj) where x != null && acc(x.f, 10);
class Obj {
  var f: int;
  method main(){
    var ch := new Chan;
    var x: Obj := new Obj { f := 3 };
    send ch(x);
    var y := new Obj { f := 5 };
    send ch(y); send ch(y);
    fork tk := counter(ch,3);
    var s: int; join s := tk;
  }
  method counter(ch: Chan, n: int) returns (s: int)
    requires ch != null && n > 0 && credit(ch, n)
  {
    s := 0;
    var i: int := 0;
    while(i != n)
      invariant ch != null && i >=0 && i <= n && credit(ch, n-i)
      invariant rd(ch.mu) && waitlevel << ch.mu;
    {
      var x: Obj;
      receive x := ch;
      s := s + x.f;
      i := i + 1;
    }
  }
}
```

Listing 7: Chalice program that demonstrates the transfer of credits

```
1  channel C(x:int);
2
3  class A {
4    method main()
5    {
6      var c := new C above waitlevel;
7      assert waitlevel << c.mu
8
9      fork msg(c);
10     fork log(c);
11   }
12
13   method msg(c:C)
14     requires c != null && credit(c,-1)
15   {
16     send c(1); // settle debt
17   }
18
19   method log(c:C)
20     requires c != null && credit(c)
21     requires rd(c.mu) && waitlevel << c.mu
22   {
23     receive x := c;
24   }
25 }
```

programs but instead translate them into Silver for verification by either Silicon or Carbon. This tool, rather predictably, is called "Chalice2Silver". As part of this thesis, we have extended that tool to provide a proof-of-concept implementation of [BM14]'s verification technique for finite blocking.

3 Modular Verification of Finite Blocking

In this section we present the scheme for verifying finite blocking in non-terminating programs that we set out to implement on the viper verification infrastructure. It was introduced by [BM14], in which the authors describe an encoding for the Boogie verifier [BDJ+06].

In certain versions of Chalice [LMS10], a thread can only receive messages on a Channel if the verifier can show that the thread has "credit" on that channel. A thread can obtain credits to a channel when it *promises* to send a message on that channel. Credits can also be transferred via method calls. This way, a thread can promise to send a message and hand that credit to other threads to use. Listing 7 shows an example of this. Were you to remove the **send** statement on line 16, the verifier would fail, informing you that a method must not leak "debt".

Listing 8: Chalice program that promises to send a messsage but never does. Partially correct and accepted by Chalice

```
method msg(c:C)
  requires c != null && credit(c,-1)
{
  var x := 5;
  while(x == 0) invariant credit(c,-1) { }
  send c(1); // never reached
}
```

The authors of [BM14] have generalized the idea of "debt" to include more constructs that could cause another thread to block. Whenever a thread gets into a state where it prevents other threads from making progress, it will get an "obligation" to *eventually* allow these other threads to proceed. When a thread acquires a monitor lock on an object, for instance, it simultaneously gets an obligation to release that lock. The verification scheme also includes termination of methods, loops and entire threads in that same model: methods can promise to their callers that they terminate, receiving an obligation to terminate for themselves.

To ensure that a thread that promises termination actually terminates and that obligations to unblock other threads are not postponed indefinitely, each obligation is associated with a lifetime expression that acts as a termination measure. Together, in the author's words, this "*[...] guarantees finite blocking for programs with a finite number of threads and fair scheduling, that is, each thread in a verified program either terminates eventually or runs forever, but no thread is blocked forever.*" [BM14]

The fact that this verification scheme *does not* assume that the program terminates is very important because most verifiers, including that version of Chalice, only prove partial correctness. If we for instance change the `msg` method to the one shown in listing 8, the original Chalice will happily accept the program even though the **`send`** statement can never be reached in any execution of the program.

Chalice still accepts the program, even though `msg` will never actually send the message. Demanding, on the other hand, that every single loop and recursive method call is annotated with a termination measure excludes many programs that regularly occur in the real world: interactive applications or servers – programs that only terminate based on external input.

3.1 Obligations

For the scope of this thesis, we are interested in three kinds of obligations: termination, lock release, sending of messages. We will explain the meaning of their common lifetime parameter `t` in section 3.4.

Send (`sends(c,n,t)`) The obligation to send `n` messages on a channel `c` within the specified lifetime `t`. Send obligations are the most liberal of the three. They accumulate (obligation to send multiple messages) and they can be negative ("credit"). You can trans-

Listing 9: Chalice method that releases a lock.

```
method main(x:Obj)
  requires x != null && releases(x, 1)
{
  release x;
}
```

fer them between threads or return them from a method for as long as their lifetime permits it.

Release (`sends(r,t)`) The obligation to release a monitor lock on an object `r` within the specified lifetime `t`. Because the locks they model are not re-entrant, you cannot acquire the lock a second time if this obligation is already present. Also, it doesn't make sense to have a "release-credit", i.e. you cannot release a lock that you don't hold.

Termination (`terminates(t)`) The obligation for a loop or recursive call chain to terminate within the specified lifetime `t`. This is the most restrictive obligation of the three: it cannot be transferred to other threads or returned from a method, nor can you have "termination-credit" or accumulate stacks of termination obligation.

3.2 Transfer of Obligations

Programmers need to explicitly annotate the methods and loops that carry obligations. Similar to how `credit` was being specified in old versions of Chalice, obligations occur mostly in the *precondition* of a method. A method that promises to release a lock (listing 9) mentions it in its precondition.

This may seem very strange at first as, intuitively, the "promise to have released a lock" sounds more like a postcondition, something that the method "ensures". A better way to look at the specification is to think of the `releases` expression as a "promise to take over the obligation to release". In that way, it works like an accessibility predicate (`acc(x.f,p)`) in that it *removes* a *thing* (obligation) from the caller's scope and adds it to the callee's scope.

From a usability perspective, specifications like `requires releases(x)` will likely confuse most newcomers. Internally, we have discussed this issue multiple times but so far failed to come up with a syntax that doesn't lose expressiveness and can reasonably be implemented. Candidates included extracting these obligation expressions from postconditions and inserting them into the precondition or to have separate "guarantees-conditions" that are then merged into the precondition. That obligation expressions can occur on the right-hand-side of implications, makes it very difficult to correctly extract obligation expressions from a postcondition and would require the user to repeat the implications when we use a separate kind of specification just for obligation expressions.

As somewhat of a special case, methods can mention obligations in their postcondition to return them to their caller. A method could, for instance, acquire a lock and return the obligation to release that lock to its caller.

The amount of obligation a method stack frame holds can change when one of the following happens

- receives obligation from caller
- gives away credit
- executes a statement that comes with obligations (e.g., acquiring a lock) or satisfies obligations (e.g., releasing a lock)
- transfers obligation to a callee/forked thread
- receives corresponding credit

3.3 Leak check

One important corner-stone of this verification scheme is to make sure that threads cannot just forget about obligations. Once a stack frame has acquired an obligation, it must get rid of it using one of the ways just mentioned. To make sure this is the case, we need to perform a *leak check* whenever a thread might stop or diverge. This means we will certainly perform leak checks just before a method returns and at the end of every loop iteration.

With method calls the situation is a bit more interesting. Technically, it would be correct to perform a leak check before each method call, it might diverge after all. If we did this, each call would require the user to transfer *all* obligations currently held to the callee, even those that the callee never fulfills and will just return unchanged. In addition to being very cumbersome, that would heavily reduce the modularity of methods.

Fortunately, methods can promise that they terminate with the `terminates` obligation and we can use that knowledge to our advantage. When we know that the callee terminates, it is fine to keep some unsatisfied obligations in the caller's stack frame while the callee executes. We have the guarantee that it will eventually return, putting the caller back in charge of the remaining obligations. With methods that do not promise to terminate, we have no choice but to perform a leak check.

The situation with *entering* loops is very similar. If a loop promises to terminate, it is fine to keep some unsatisfied obligations outside the loop. We are guaranteed to be back in control of those "hidden" obligations within a finite amount of time.

Calling a method asynchronously (**`fork`**) does *not* involve a leak check as it doesn't block and thus won't prevent us from fulfilling the remaining obligations. Joining a forked thread, however, requires special attention since that thread might not terminate. In this thesis we do not allow programs to wait for threads that haven't promised to terminate. One could differentiate between threads that *might* diverge and ones that *definitely won't* diverge. We would then mandate a leak check before joining one of the former sort.

We consider it illegal to call a method asynchronously if it potentially returns obligations to its caller as we cannot guarantee that anyone would ever join on that thread (and thus take over the returned obligations). Similarly, we do not allow obligations to be sent over channels as we cannot guarantee that they will ever be received. It is, however, fine to send credits (negative obligations) over channels or leak them.

Listing 10: Termination obligations work behave like loop variants

```
var x := 0;
while(x < 5)
    invariant terminates(5 - x)
{
    x := x + 1;
}
```

3.4 Obligation Lifetime

To just check for leaked obligations is not enough, though. As it stands, we can just hide the obligation in plain sight: keep it around but delay its fulfilment indefinitely, like in listing 8. To prevent this, obligations are associated with an expression that will serve as a termination measure while that obligation is held. We call this the obligation's *lifetime expression*. At key points in the program we check that the values of these expressions strictly decreases. To simplify things, we use natural numbers as the domain of lifetime expressions. Theoretically, any well-founded-set can be used.

Note that the way we check lifetime here is a departure from the scheme proposed in [BM14]. There, an obligation was associated with an integer *"lifetime counter"* that was decreased at the beginning of every method and loop body.

When used in loop invariants, lifetime expressions behave much like loop variants. At the end of the loop iteration, their value is compared to the value they had at the beginning of the iteration. The lifetime check fails if the value at the end of the iteration is negative ("outside of the well-founded-set") or not strictly smaller than the value at the beginning of the loop. Listing 10 shows a simple example of this.

To keep method call recursion in check, obligation expressions in preconditions are also annotated with lifetime expressions. Every time the method calls another method, we compare the value of the callee's lifetime expression to the value of the caller's lifetime expression. Consider the example in figure 1. Here the method `caller`, which carries an obligation with lifetime expression `this.b`, calls the method `callee` which guarantees to release the lock within `this.b - 1`. As `this.b >= 0 && this.b - 1 < this.b` holds, the call is legal.

```
2  method caller()
3    requires rd(this.b)
4    requires this.b > 1
5    && releases(this, this.b)
6  {
7    call callee();
8  }
```

```
10 var b : int;
11 method callee()
12   requires rd(this.b)
13   && releases(this, this.b-1)
14 {
15   release this;
16 }
```

Figure 1: Chalice+Obligations program with a call that passes the lifetime check at the call site.

In the previous example (figure 1), referred to the state of the heap, albeit a frozen state since we only had read-permission on `this.b`. Lifetime expressions can also be expressed

Listing 11: A terminating recursive method in Chalice+Obligations that expresses it's lifetime expression in terms of mutable heap state.

```
1  method fib() returns (y:int)
2    requires acc(this.n) && terminates(this.n)
3    ensures acc(this.n) && old(n) == n
4  {
5    if(n <= 2){
6      y := 1;
7    } else {
8      this.n := n-2; call y := fib();
9      this.n := n+1; call t := fib();
10     this.n := n+1; y := y + t;
11   }
12 }
```

in terms of mutable heap state. The example in listing 11 manipulates the heap in preparation for each recursive method call. For the call on line 8, for instance, we need to check that the current value of `this.n` is smaller than the value of `this.n` at when we entered the method on line 4.

You will note that the lifetime expression doesn't exactly measure *time* since we can have an arbitrary number of calls inside one method as long as the lifetime expressions of each call is smaller than the caller's lifetime expression at method entry. It is more like bound on the depth of the remaining call stack. This bound also applies to terminating threads forked off by this method. Otherwise, a method could spawn a terminating thread with the same or longer lifetime and then immediately wait for it to terminate (fork directly followed by join is essentially the same as a synchronous call).

While a method can only fulfil a termination obligation by terminating, other obligations can be fulfilled earlier. In listing 12 the method `m` promises to release the monitor lock on `this` with a maximum stack depth of 1. There is no lifetime check when we enter the loop on line 5, just a leak check. In the loop invariant, we assign a different lifetime to the release obligation (line 8). We don't need to treat a loop like a new stack frame (the number of nested loops is bounded by the length of the program source file). It is thus acceptable for the lifetime of an obligation in a loop to be larger than the lifetime of the same obligation on the method's precondition (the two are never compared). When the loop reaches the iteration with `this.b == 5`, the monitor lock is released (line 13). At the end of this iteration we have no obligations left and consequently there is no need to perform a lifetime check. From this point on, the thread can diverge without breaking its promise.

3.5 Unbounded obligations

There are some limited cases where obligations are *not* associated with a lifetime expression. In the original verification scheme [BM14], these were called *fresh* obligations and had a special quasi-infinite value assigned to their lifetime counter field. We call them *unbounded* obligations in contrast to *bounded* obligations, which are associated with a lifetime expression.

Listing 12: A Chalice+Obligations example with a loop that does not terminate but fulfills a release obligation half-way through

```
1  var b:int;
2  method m()
3    requires acc(this.b) && this.b > 17 && releases(this,1)
4  {
5    while(this.b > 2)
6      invariant acc(this.b)
7      invariant this.b > 4 ==> releases(this, this.b)
8    {
9      if(this.b > 4){
10       this.b := this.b - 1;
11       if(this.b == 4) {
12         release this;
13       }
14     }
15   }
16 }
```

Unbounded obligations originate from statements that *increase* obligations for the current thread. A lock statement (`acquire x`), for instance, adds an unbounded obligation to the current stack frame. Like their bounded brethren, they must not be leaked, but they are free to skip lifetime checks.

Consider the example in listing 13 which implements a loop that waits for the shared condition variable `f` to become non-zero. The method maintains its lock between loop iterations and especially between finding a non-zero value in the condition variable and exiting. At the same time, it releases and re-acquired the lock on every loop iteration to give other threads a chance to advance (and perhaps set the condition variable). Using just bounded obligations, this method would fail to verify because when we compare the loop-lifetime of the `releases` obligation, we have to assert that `1 < 1`. Instead, the **`acquire this`** statement adds an unbounded obligation to the loop frame, which is then allowed to skip the lifetime check at the end of the loop iteration.

Unbounded obligations must only occur in "output positions" and generally represent obligations that have been acquired *during* the execution of the current scope. They are transparently converted to bounded obligations when they are transferred to a scope that expects bounded obligations. A conversion in the other direction is highly illegal. If it weren't, we could convert back-and-forth between unbounded and bounded obligations to switch associated lifetime expressions at will.

A method may have unbounded obligations in its postcondition. Those can only be satisfied by the method with obligations that are actually unbounded. This is important to allow users to build methods around **`acquire`** (and similar statements) that behave the same in terms of obligations and their lifetimes. In the example in listing 13 we have to do just this because we might never release the lock if the condition is already non-zero on entry.

Listing 13: Chalice method that waits until the shared field `f` is non-zero.

```
invariant acc(this.f);
method wait_for()
  requires releases(this,1) && acc(this.f)
  ensures releases(this,1) && acc(this.f)
{
  while(this.f == 0)
    invariant releases(this,1) && acc(this.f)
  {
    release this;
    acquire this;
  }
}
```

4 Designing Extensions to Silver

Some parts of the verification scheme presented in the previous section are extremely difficult if not outright impossible to implement using the existing viper intermediate language "Silver". A prime example is the leak check. For each kind of obligation k, we would need to assert something like:

$$\forall r : \text{Ref} :: \textit{in-current-frame}\,(r) \;\rightarrow\; \textit{obligation-amount-of}\,(r, k) \leq 0$$

but while Silver *does* have a `forall` expression, it is not quite what we need, because there is no way to express *in-current-frame* (r). The viper verification backend Silicon hands the quantifier more or less directly to the underlying SMT solver, which has no concept of heap or frames within the heap.

Some sort of extension to Silver would be necessary to allow us to encode modular verification of finite blocking in it.

4.1 Direct Encoding

One way to achieve our goal would have been to introduce support for obligations as a "first-class" feature into Silver. We would have defined an obligation predicate that looks something like this:

$$\text{obl}\,(r, k, n, t, b_{\text{acc}}, b_{\text{credit}})$$

which represents an n obligations of kind k on object r with lifetime t. Two additional boolean parameters, b_{acc} and b_{credit}, determine whether this particular obligation is allowed to accumulate or become negative, respectively. The translation in the frontend would have been very straightforward and while we would have had to add a lot of new logic to the verifier backend, it also had full knowledge of obligations and could implement them with primitives much more powerful than what silver offers.

Drawbacks are plenty, however. Every change to the finite-blocking verification scheme would affect all layers down to all the verification backends. If we wanted to add deadlock-prevention, a feature that is relatively orthogonal to obligations, we would have to touch every tool that uses silver.

4.2 Big Toolbox

Moving down the abstraction ladder, at one point we had a design where Silver would treat all obligations the same, but offer a wide list of new expressions and statements to inspect and manipulate them as necessary.

obl (r, k, n, t)

leak-check (k) an expression or statement that performs the leak check for obligations of kind k

lifetime-check (r, k) an expression that checks the lifetime on an obligation

obligation-amount (r, k) an expression that represent the amount of k obligation the thread currently holds on object r

This design left the leak check and the lifetime check firmly in the hands of the verifier backend and while they are pretty well defined, they did not have much promise for re-use outside of verifying our obligations.

The requirements for the "obligation store" (the verifier component that keeps track of obligations) itself looked suspiciously like the ones for permissions: (1) a number that can be incremented and decremented (2) ability to merge/cancel out contributions from different sources and (3) check whether amount is at a certain level. But some semantics of access permissions conflict with the semantics of obligations: (1) "zero permissions" is the same as "not in the heap" (2) negative permission amounts are illegal (3) two objects are distinct if permissions add up to more than full permission

4.3 Universal mechanisms

One element of the original verification scheme in [BM14] that was difficult to implement was the way it handled lifetimes. There a lifetime was another counter that was decreased on entry to a loop or method body. Deviating from that original scheme to comparing lifetime expressions reduced the demands on the Silver extension considerably. As we can implement the comparison between lifetime expressions entirely in the frontend, this entire dimension falls away from the extension.

At this point a Silver extension primarily needs to provide us with ways to do obligation bookkeeping and help with the leak check.

5 Final Extensions to Silver

In this section we present our extensions to Silver with their syntax, informal semantics and a sketch of their implementation in the verifier backend "Silicon". We explain how we use these extensions to implement verification of finite blocking in section 10.

5.1 Token Fields

A token field is a variation of an ordinary field in Silver, but instead of tracking access permissions in addition to their value, they track their "amount of tokens". Unlike per-

mission amounts, token amounts have no specific meaning in Silver. They are simply an accounting mechanism offered to frontends.

The idea is to re-use as much of the existing permission bookkeeping machinery in today's Viper verification backends as possible. A "token amount" is therefore represented as a permission amount on the Silver level. We use the full permission (called **`write`** in Silver) to denote one token. If a token field has two tokens, it reads `2*`**`write`**; if it has a negative token, it reads `(-1)*`**`write`**. Token amounts aren't restricted to multiples of the **`write`** permission, though we won't be using that capability in this thesis. Verification backends need to make sure they treat token fields with a zero or negative token amount properly. Until very recently, Silicon immediately removed fields that have reached zero permissions from the heap.

A token field declaration looks very similar to an ordinary field declaration:

`token` **`field`** ⟨id⟩ `:` ⟨type⟩`;`

When a new object is created, the current thread does *not* automatically inhale one token (a full write permission) for token fields like it does for ordinary fields. It is up to the frontend to decide whether a particular token field should start out empty, with a token or perhaps even with a negative token amount.

Implementation in Silicon

Silicon already differentiates between different kinds of heap chunks. One of them is the "direct field chunk", a quadruple of the form

$$(r, f) \mapsto (p, v) \qquad \text{where } r : \text{object reference}, \; f : \text{field identifier}, \; p : \text{permission amount term}, \; v : \text{field value term}$$

To support token fields, we introduce a new kind of chunk, the "direct token field chunk", also represented by a quadruple

$$(r, f) \mapsto (a, v) \qquad \text{where } a : \text{token amount term}$$

Where possible we use an also newly introduced abstraction, the "direct field-like chunk".

We need to pay particular attention to how we retrieve tokens from Silicon's symbolic heap. In the presence of aliasing, it can happen that multiple chunks in the symbolic heap really contribute to the same "location" or in our case to the same "token register". For permissions, it is often enough to find any one of the applicable chunks. If the verifier only needs to check if *some* positive permission amount is present, then just about any chunk will do. In the case it's not enough, say when a full permission is required, Silicon catches the verification failure, compresses the heap and tries again.

When retrieving token fields, we always scan the entire symbolic heap for matching tokens. If we are not sure whether a chunk matches, we ask the prover to check whether the receiver objects match. Should we find more than one token, we trigger a heap compression and look again. Since token amounts can be negative, it is absolutely vital that we have seen every chunk that belongs to a token field before we return it.

5.2 Token Predicate

Token assertions are used to transfer "token amounts" between frames (stack frames, loop bodies, threads). The amount of tokens transferred can be specified by a term of type "permission amount".

`tok` (⟨obj⟩ . ⟨tokenfield⟩ , ⟨token-amount⟩)

On the AST-level, token predicates have the same representation as accessibility predicates (**`acc`** (⟨obj⟩ , ⟨field⟩ , ⟨perm-amount⟩)). Whether the field is an ordinary field or a token field differentiates between the two kinds of predicates. Frontends, including the textual Silver representation (parser *and* pretty printer) choose between the keywords `tok` and **`acc`** as necessary.

When inhaled and exhaled, token predicates behave mostly like their accessibility counterparts. When a verification backend encounters a token predicate in an inhale statement,

```
inhale tok(r.f,a)
```

it increases the token amount of the token field `f` on object `r` by `a`. Unlike with access permissions, `a` can also be negative, which of course causes the resulting token amount to decrease accordingly. When encountered in an exhale statement

```
exhale tok(r.f,a)
```

it decreases `r.f`'s token amount by `a`. Again, `a` could be negative, which would cause the token amount to increase instead. It is perfectly legal for token amounts to become negative.

Implementation in Silicon

We adapted the handling of field access predicates that cover token field during inhale and exhale (called "produce" and "consume" in Silicon) to ensure the correct kind of field chunk gets integrated into the heap and to skip a check for a negative token amount during exhale.

Unlike with accessibility predicates, there are no checks on the token amount when exhaling tokens, that is, you can start at zero tokens and and exhale a token, no questions asked. This also means that a statement like **`assert`** `tok(r.f, 1)` will never fail. Silicon will perform an exhale and then drop the modified state and continue verification as if the assertion hadn't been there.

5.3 Token amount term

So far we have only presented slight variations on existing Silver mechanisms. One feature that we cannot live without is the ability to denote the current amount of tokens that the thread holds for a token field. Silver already has a permission amount term (**`perm`**`(r.f)`) but it's semantics are not very well defined and in particular it's implementation in Silver is unsuitable for our purposes.

Silicon uses two copies of the heap to perform an exhale. One is a read-only copy of the heap from just before the exhale statement. It is used to evaluate terms. All changes to

Listing 14: Silver program demonstrating behaviour of `tokenamount(x.f)`

```
token field f: Int;
method main(x:Ref)
  requires x != null
{
  inhale tok(x.f,2*write);
  exhale tok(x.f,1*write)
    && tokenamount(x.f) == write
    && tok(x.f,1*write)
    && tokenamount(x.f) == none;
}
```

permissions in the heap it performs on a second copy. That way, it can handle statements like

```
exhale acc(x.f, write) && x.f == 0
```

If Silicon tried to evaluate `x.f == 0` against the same heap from which it removed all access permissions to `x.f` just before, it would come to the conclusion that the access to `x.f` was illegal. As the existing `perm(r.f)` term, like all terms, looks up permission amounts in the original unmodified copy of the heap, it cannot give us the "current" amount of tokens *during* the exhale.

While we could have co-opted the `perm` to implement our desired semantics when the field in question is a token field, we decided to create an entirely new kind of term in this case, because we wanted to preserve the semantics for `perm` even for tokens. One could for instance want to use it to compare the token amount just before the exhale with the current token amount. The proposed syntax is

```
tokenamount(r.f)
```

which is admittedly very token-specific but it's not clear whether this term even makes sense for permission amounts. With this term in place, we can successfully verify the program fragment in listing 14

Implementation in Silicon

When Silicon evaluates a term during an inhale or exhale, we make a reference to the second copy of the heap (the one that Silicon modifies when it encounters an `acc`) available to the evaluation routine via ambient data structures (the "context" parameter). As we only ever *read* from this heap during the evaluation, this works just fine.

With the `tokenamount` expression, in particular, it's very important that we report token amounts accurately, because that's how the frontend can check that the expected amount of tokens is present. As mentioned in the section about token fields, we specifically handle the case where the prover finds multiple chunks on the heap and require a heap compression before we continue.

For ordinary permissions, Silicon tries to verify the method with just the amount of permission it found in the first matching chunk. If it works, fine, if it doesn't, it compresses

the heap and tries again. The worst that can happen for permissions, is that Silicon fails to verify the first time around. In the case of tokens, if we tried the same, the method might verify the first time around but only because we haven't seen the obligations hiding in chunks other than the first.

5.4 "forall references" Assertion

We need a way to evaluate an assertion for all objects in the heap but full universal quantification over all possible heap references is not very friendly to SMT solvers, especially when our Silver verifier backends more or less maintain a list of all heap references that have appeared to the current thread. We thus introduce a new expression that takes advantage of that knowledge:

```
forallrefs[f1,f2,...,fn] r :: e
```

This expression takes a list of field identifiers (`f1` through inlinesil!fn!), an identifier for the "quantified" variable `r` and a boolean expression `e` which is instantiated for each object in the heap with (`r` being a reference to that object). The list of fields is used to limit the quantification to objects that have at least one of those fields.

If this sounds a bit vague, that's because currently the behaviour of this extension is primarily dictated by its implementation in Silicon. Particularly interesting is the behaviour of **`forallrefs`** when confronted with zero permissions/tokens to a field. Up until very recently, for example, Silicon immediately removed such chunks from its heap representation. For the moment a reasonable directive for implementing this extension would be this: "The expression *e must* be instantiated at least once for every object *r* for which the permission/token amount of at least one of the fields in the field list ist positive. The expression *e may or may not* be instantiated for other objects."

Listing 15 demonstrates how the **`forallrefs`** is used. We check that there are no negative permission amounts for the token field `g` and that all chunks for token field `f` are non-zero.

Listing 15: Sil program using **`forallrefs`**

```
token field f:Int; token field g:Int; token field h:Int;

method m() {
  var x:Ref; var y:Ref; var z:Ref;
  x := new(); y := new();  z := new()
  inhale tok(x.f, write) && tok(y.f, write) && tok(z.f, write);
  inhale tok(x.g, write)                    && tok(z.g, write);
  inhale                   tok(y.h, write) && tok(z.h, write);

  assert forallrefs[g] r :: tokenamount(r.g) >= none;
  assert forallrefs[f] r :: tokenamount(r.f) != none;
}
```

Implementation in Silicon

Silicon represents it's heap as a collection of heap chunks.

$$(x, f) \mapsto (p, v)$$

When we encounter a **`forallrefs`** expression in Silicon, we iterate over all chunks currently in the heap, select those that match our field list and instantiate the **`forallrefs`** -body using the object reference term x as the value for the free variable r. Finally, we assemble the resulting expression into one big conjunction. While this implementation may cause Silicon to generate the same condition multiple times when an object matched multiple fields on the field list *and* has a heap chunk for each of those fields, it is still an extremely convenient formulation for the prover because we are handing it a list of conditions to check, that we *know* to be sufficient, instead of asking it to figure out whether the quantified condition holds universally.

In pseudo-code, the implementation of **`forallrefs`**`[F] r :: e` looks something like this:

```
σ : evaluation state
var C := True  // the conjunction
for chunk ⟵ heap
    if ( chunk.field ∈ F)
        t := eval e with state σ[r ↦ chunk.receiver]
        C := C ∧ t
```

As an optimization, we can remember each receiver term that we have emitted a condition for and skip it, if it should occur multiple times. This will naturally only prevent duplicate conditions for identical receiver *terms* but won't help against aliased receivers.

5.5 Labelled old expressions

When we need to compare the current state to a particular previous state, the final extension – the labelled old expression – comes in handy. Using it, the frontend can mark a position in a method using a new statement – the "state label" – and then further down in the method have the verifier backend evaluate expressions as if they would occur at the marked position. Syntactically, it looks as follows:

`statelabel` ⟨label⟩ **`;old`** [⟨label⟩] (⟨expression⟩)

We have deliberately decided *not* to make labelled states a first-class value that you could store in variables or pass as arguments. We didn't want to go down the rabbit hole of tracking heaps stored in other heaps. Keeping state labels completely static also gives backends considerable freedom on how to implement labelled old expressions.

Listing 16 for an example of labelled old expressions.

Naturally, it is illegal to label more than one point in a program with the same state label or to refer to a state label before control has reached that point. State labels defined inside a loop are not visible outside the loop. The body of a loop can, however, refer to state labels

Listing 16: Silver program with labelled old expressions.

```
token field tk: Int
field f: Int
method main() {
  var m: Ref
  m := new (*)

  m.f := 15
  statelabel initial
  m.f := 3
  inhale acc(m.tk, write)

  assert m.f == 3 && old[initial](m.f) == 15
  assert tokenamount(m.tk) == write && old[initial](tokenamount(m.tk)) ==
      none

  exhale acc(m.f)
  assert old[initial](m.f) == 15
}
```

that occur before that loop. And finally, when a labelled old expression refers to a state label, this state label must be passed on all possible execution paths leading to that old expression.

Implementation in Silicon

When Silicon encounters a state label, it stores a copy of the current heap and local variable store in a map. This map is part of the verification context that is passed along all possible execution paths. That way, the state label map effectively remains a separate data structure for each execution path explored by Silicon. If there is any execution path for which a statelabel remains undefined, Silicon has found a situation where an old expression is not dominated by the corresponding state label in the control flow graph and will report this as an error.

Theoretically, we could allow the same state label to be defined in different places for mutually exclusive execution paths, like in the following example:

```
if(x > 3){
  x := 5;
  statelabel label
} else {
  x := 7
  statelabel label
}
```

Each execution path explored by Silicon would only see one of these labels. Such programs are rejected by the Silver type checker but, like many other illegal Silver programs, can be constructed in-memory and passed directly to tools without passing through the type checker.

Listing 17: Legal Silver program where the state label does not strictly dominate the old expression.

```
method main(i:Int)
  requires i > 5
{
  var x : Int := 3
  if(i > 2){
    statelabel initial
  }
  x := 19
  assert old[initial](x == 3)
}
```

We do not currently reject programs where a statelabel is on all paths to an old expressions that *can* occur at runtime but does not strictly dominate it in the control flow graph, like in listing 17. Again, it is unlikely that a frontend would ever accidentally generate such a program. We might, however, still want to disallow such programs to simplify the old-expression feature and make it easier to handle for other tools.

Storing a copy of the entire heap when we encounter a state label is not the only implementation strategy. Alternatively, a verifier backend could analyse all old expressions that use the label and evaluate the inner expressions it finds ahead of time. This almost certainly saves a significant amount of memory and also potentially reduces the number of expressions to evaluate if the same term occurs in multiple old expressions. It is, however, not clear how such an implementation would deal with labelled old expressions in quantified and **`forallrefs`** expressions.

6 Encoding Modular Verification of Finite Blocking

In this section we describe how we encode the verification of finite blocking (section 3) into Silver using the extensions presented in the previous section. The source language will be a variant of Chalice that uses obligations to ensure that no thread blocks another thread for an unbounded amount of time.

6.1 Chalice-level obligations

On the level of Chalice, obligations are a bit more complex than on the level of Silver. Some obligation assertions are associated with a *lifetime expression*. That expression will be used to ensure that the obligation is met 'in time'. Those obligations *with* a lifetime expression are the *bounded* obligations. Is the lifetime expression omitted, we have an *unbounded* obligation.

To encode the difference between unbounded and bounded obligations, we use two silver token fields per obligation kind. One for bounded obligations and one for unbounded obligations. The total of the token amounts in the two fields tells us how much obligation (or credit) we have. The token amount in the bounded field alone tells us whether we have obligations that are "on a timer".

		Chalice	Silver field
release monitor lock	unbounded	**releases**(x)	rel_u
	bounded	**releases**(x, t)	rel_b
send messages	unbounded	**sends**(x, a)	send_u
	bounded	**sends**(x, a, t)	send_b
termination*	~~unbounded~~	~~**terminates**~~	~~term_u~~
	bounded	**terminates**(t)	term_b

where

x is an object reference term

a is an obligation amount (integer), can be positive, negative or zero

t is a lifetime expression, currently an integer

*There are no unbounded termination obligations. A method/loop either promises termination in its precondition/invariant or it doesn't. Unbounded obligations are only useful if they can be transferred to a surrounding context. However, for translation schemes that are the same for all kinds of obligations, we won't specifically mention the absence of term_u every time.

Additionally, the source language recognises a dedicated assertion for channel credits:

`credit` (x, a)

Even though the `credit` assertion lacks a lifetime expression, it transfers $-a$ obligations to and from the *bounded* field. It is not just syntactic sugar, though. The user can put `credit` assertions in places where obligations are forbidden, but it cannot be used to transfer obligations between frames.

6.2 Manual Specifications

As a high level verification intermediate language, Silver has many concepts already built-in. Method's have pre- and postconditions that verifier backends automatically inhale on method entry and exhale at the end. The same for loop invariants. Unfortunately, there are some situations where we need more control for our encoding. For some of these situations, Silver offers the "paired-assertion" construct, a pair of expressions $[e_1, e_2]$ where the e_1 is used whenever the paired-assertion is inhaled and e_2 is used when the expression is exhaled. However, our encoding has situation where two exhales of the same condition in different contexts needs to be translated differently.

This leaves us no choice but to ignore Silver's specification machinery and instead implement the verification of pre-, postconditions and invariants manually. This is certainly not ideal, as the resulting Silver programs become much bigger because specifications are repeated multiple times. Figure 2 demonstrates this. At the same time it shows that Silver is flexible enough to grant frontends this additional freedom, if necessary.

We will show how we implement method and loop translations in sections 6.5 and 6.5.

6.3 Assertions

There are a number of variations how we translate obligation assertions depending on whether it occurs in an **`exhale`** or and **`inhale`** statement, whether we are at the end of

```
field f:Int;
method automatic(x:Ref)
  requires x != null
    && acc(x.f,write)
  ensures acc(x.f,write)
    && x.f == 5
{
  x.f := 0
  while(x.f != 5)
    invariant acc(x.f,write)
  {
    x.f := x.f + 1
  }
}
```

```
field f:Int;
method manual(x:Ref)
  requires true
  ensures true
{
  inhale x != null
    && acc(x.f, write)
  x.f := 0
  exhale acc(x.f,write)
  var b:Bool
  while(b)
    invariant true
  {
    inhale acc(x.f,write)
      && x.f != 5
    x.f := x.f + 1
    exhale acc(x.f,write)
  }
  inhale acc(x.f,write)
    && !(x.f != 5)
  exhale acc(x.f,write)
    && x.f == 5
}
```

Figure 2: A Silver method, once using Silver specification and once with "manually" implemented specifications.

loop iteration or not, etc. We first present the general translation scheme for obligation-specific assertion expressions and mention which parts of it apply when we refer to it later.

To make the translation scheme a bit more readable we will be using the literals 0 and 1 to refer to zero and full permission amounts (**none** and **write** in Silver). Especially the **write** literal looks very out-of-place when we are actually dealing with token amounts that have nothing to do with heap access permissions.

Fundamentally, obligation assertions have a simple job: they need to add (on inhale) or subtract (on exhale) the obligation amount in the assertion to or from the corresponding token field. If the assertion includes a lifetime expression, it applies to the *bounded* token field, if it doesn't, to the *unbounded* one.

Combined, our choices to encode credit as negative obligations *and* to use two separate token fields for bounded versus unbounded obligations have one drawback: the encoding allows for both "bounded and unbounded credits". While this undesired degree of freedom doesn't affect leak checks, it can cause lifetime checks to be ignored. Imagine you have one "bounded credit" and one "unbounded obligation". Summed up, you would have zero obligations and zero credits at that point. Now inhale a bounded obligation. Naively you would end up with zero in the bounded field and one unbounded obligation. This is wrong as when summed up, we expect to have one bounded obligation and no credits.

To deal with this situation, we introduce a translation "macro" *applyCredit* which tallies

up credits and obligations across bounded and unbounded field pairs and subsequently moves all remaining credit into the bounded field. That way both leak- and lifetime checks arrive at the correct conclusion when looking at the token fields.

We define applyCredit (r, f) for an object reference r and a field pair f as follows:

transfer $(r.f, a)$:= tok $(r.f, a)$ *if* applied in an **inhale** statement
transfer $(r.f, a)$:= tok $(r.f, (-1) * a)$ *if* applied in an **exhale** statement

applyCredit (r, f) :=
  (tokenamount $(r.f_b) < 0$ && tokenamount $(r.f_u)$!= $0) \Rightarrow$
    transfer $(r.f_u$, tokenamount $(r.f_b))$
    && transfer $(r.f_b, (-1) *$ tokenamount $(r.f_b))$
    && (tokenamount $(r.f_u) < 0 \Rightarrow$
      transfer $(r, f_b$, tokenamount $(r.f_u))$
      && transfer $(r.f_u, (-1) *$ tokenamount $(r.f_u))$
)

applyCredit (r, f) first applies all credit in the bounded field to the unbounded field, then zeroes out the bounded field. If afterwards, the unbounded field has credit, i.e. we transferred too much, we perform the same operation in the opposite direction.

6.3.1 Exhale

To exhale assertions, we use the translation scheme E presented in this section. We only mention those cases that are important for obligations. All other cases behave like in the normal Chalice2Silver translation.

$E\left([\![a \&\& b]\!]_{\text{Ch}}\right) := E\left([\![a]\!]_{\text{Ch}}\right) \&\& E\left([\![b]\!]_{\text{Ch}}\right)$
$E\left([\![a \Rightarrow b]\!]_{\text{Ch}}\right) := [\![a]\!]_{\text{Ch}} \Rightarrow E\left([\![b]\!]_{\text{Ch}}\right)$

For bounded obligations, we sometimes need to check whether they are still within their lifetime. We will explain this lifetime check in section 6.8. For now, we use the following placeholder:

r : object reference to check
f : kind of obligation to check the lifetime of
t : lifetime expression to check
E : scope to compare lifetime against
lifetimeCheck (r, f, t, E)

Not all **exhale** statements will include a lifetime check. If no lifetime check is needed, lifetimeCheck (r, f, t, E) is simply true. What E is, is determined at the point where the exhale statement is issued.

Release Obligations (`releases(r,t)` and `releases(r)`) When we fulfil releases obligations, we additionally check whether the user tried have the lock released multiple times. We should have a total of zero tokens in rel_b and rel_b when exhale bounded release obligation. We must sum up *both* token fields because by exhaling a bounded obligation, you can also satisfy an unbounded obligation. The other way around when we exhale an unbounded obligation, we *must only* check the unbounded field. If the current thread instead had a bounded obligation, we would be at negative one token in the unbounded field at that point and the check would fail as it should.

$$
\begin{aligned}
&E\left(\llbracket \textbf{releases}\,(r,t)\rrbracket_{\text{Ch}}\right) :=\\
&\quad r \neq \text{null}\\
&\quad \&\&\ \text{lifetimeCheck}\,(r, \text{rel}, t, E)\\
&\quad \&\&\ \texttt{tok}\left(\llbracket r\rrbracket_{\text{Ch}}.\text{rel}_b, 1\right)\\
&\quad \&\&\ \text{applyCredit}\left(\llbracket r\rrbracket_{\text{Ch}}, \text{rel}\right)\\
&\quad \&\&\ \left(\texttt{tokenamount}\left(\llbracket r\rrbracket_{\text{Ch}}.\text{rel}_b\right) + \texttt{tokenamount}\left(\llbracket r\rrbracket_{\text{Ch}}.\text{rel}_u\right) = 0\right)
\end{aligned}
$$

$$
\begin{aligned}
&E\left(\llbracket \textbf{releases}\,(r)\rrbracket_{\text{Ch}}\right) :=\\
&\quad \llbracket r\rrbracket_{\text{Ch}} \neq \text{null}\\
&\quad \&\&\ \texttt{tok}\left(\llbracket r\rrbracket_{\text{Ch}}.\text{rel}_u, 1\right)\\
&\quad \&\&\ 0 = \texttt{tokenamount}\left(\llbracket r\rrbracket_{\text{Ch}}.\text{rel}_u\right)
\end{aligned}
$$

Send Obligations (`sends(r,a,t)`, `sends(r,a)` and `credit(r,a)`) For unbounded `sends` obligations we make sure that the message amount is non-negative to prevent "unbounded credit" from being exhaled. If you want a send credit, you need to exhale a bounded send obligation with a negative number of messages. In our case, it is legal to send channel credits over channels, but you cannot send obligations of any kind over a channel (we have no guarantee that the message will ever be received). To solve this problem, we declare obligation assertions in channel invariants illegal *except* for `credit`. At the same time `credit` must ensure that it is only ever used to transfer credit.

$$
\begin{aligned}
&E\left(\llbracket \textbf{sends}\,(r,a,t)\rrbracket_{\text{Ch}}\right) :=\\
&\quad \llbracket r\rrbracket_{\text{Ch}} \neq \text{null}\\
&\quad \&\&\ \text{lifetimeCheck}\,(r, \text{send}, t, E)\\
&\quad \&\&\ \texttt{tok}\left(\llbracket r\rrbracket_{\text{Ch}}.\text{send}_b, \llbracket a\rrbracket_{\text{Ch}}\right)\\
&\quad \&\&\ \text{applyCredit}\left(\llbracket r\rrbracket_{\text{Ch}}, \text{send}\right)
\end{aligned}
$$

$$
\begin{aligned}
&E\left(\llbracket \textbf{credit}\,(r,a)\rrbracket_{\text{Ch}}\right) :=\\
&\quad r \neq \text{null}\\
&\quad \&\&\ \llbracket a\rrbracket_{\text{Ch}} \geq 0\\
&\quad \&\&\ \texttt{tok}\left(\llbracket r\rrbracket_{\text{Ch}}.\text{send}_b, (-1) * \llbracket a\rrbracket_{\text{Ch}}\right)\\
&\quad \&\&\ \text{applyCredit}\left(\llbracket r\rrbracket_{\text{Ch}}, \text{send}\right)
\end{aligned}
$$

$$
\begin{aligned}
&E\left(\llbracket \textbf{sends}\ (r, a) \rrbracket_{\text{Ch}}\right) := \\
&\quad \llbracket r \rrbracket_{\text{Ch}} \neq \text{null} \\
&\quad \&\&\ 0 \leq \llbracket a \rrbracket_{\text{Ch}} \\
&\quad \&\&\ \text{applyCredit}\left(\llbracket r \rrbracket_{\text{Ch}}, \text{send}\right) \\
&\quad \&\&\ \texttt{tok}\left(\llbracket r \rrbracket_{\text{Ch}}.\text{send}_u, \llbracket a \rrbracket_{\text{Ch}}\right)
\end{aligned}
$$

Termination Obligations (`terminates(t)`) As already discussed, termination obligations are always bounded and there is only one token field we need to concern ourselves about. Since every token field needs to be attached to an object, we allocate a special 'thread' object in a local variable of each method. We never pass this object to any method. It's just a named location on the heap to hold the term_b field.

$$
\begin{aligned}
&E\left(\llbracket \textbf{terminates}\ (t) \rrbracket_{\text{Ch}}\right) := \\
&\quad \&\&\ \text{lifetimeCheck}\,(\text{thread}, \text{term}, t, E) \\
&\quad \&\&\ \texttt{tok}\,(\text{thread}.\text{term}_b, 1)
\end{aligned}
$$

Even though the termination obligation is binary, unlike the release obligation, we do not check whether the termination token amount is zero after the exhale. This is required by our encoding of method calls and other constructs to detect whether the "callee" has promised to terminate, even in scenarios where the caller has zero termination obligation.

6.3.2 Inhale

Inhaling obligations works in a similar way, but we don't have lifetime checks and in the case of `releases` obligations, we can skip applyCredit $(\cdot)$ as there is no such thing as unbounded credit to balance out. Again, all cases not mentioned below work like they did in Chalice2Silver before.

$$
\begin{aligned}
&I\left(\llbracket a \&\& b \rrbracket_{\text{Ch}}\right) := I\left(\llbracket a \rrbracket_{\text{Ch}}\right) \&\& I\left(\llbracket b \rrbracket_{\text{Ch}}\right) \\
&I\left(\llbracket a \Rightarrow b \rrbracket_{\text{Ch}}\right) := \llbracket a \rrbracket_{\text{Ch}} \Rightarrow I\left(\llbracket b \rrbracket_{\text{Ch}}\right)
\end{aligned}
$$

Release Obligations

$$
\begin{aligned}
&I\left(\llbracket \textbf{releases}\ (r, t) \rrbracket_{\text{Ch}}\right) := \\
&\quad \texttt{tok}\left(\llbracket r \rrbracket_{\text{Ch}}.\text{rel}_b, 1\right)
\end{aligned}
$$

$$
\begin{aligned}
&I\left(\llbracket \textbf{releases}\ (r) \rrbracket_{\text{Ch}}\right) := \\
&\quad \texttt{tok}\left(\llbracket r \rrbracket_{\text{Ch}}.\text{rel}_u, 1\right)
\end{aligned}
$$

Send Obligations

$$
\begin{aligned}
&I\left(\llbracket \textbf{sends}\,(r, a, t)\rrbracket_{\text{Ch}}\right) := \\
&\quad \text{applyCredit}\left(\llbracket r\rrbracket_{\text{Ch}}, \text{send}\right) \\
&\quad \&\&\ \texttt{tok}\left(\llbracket r\rrbracket_{\text{Ch}}.\text{send}_b, \llbracket a\rrbracket_{\text{Ch}}\right)
\end{aligned}
$$

$$
\begin{aligned}
&I\left(\llbracket \textbf{credit}\,(r, a)\rrbracket_{\text{Ch}}\right) := \\
&\quad \llbracket a\rrbracket_{\text{Ch}} \geq 0 \\
&\quad \&\&\ \text{applyCredit}\left(\llbracket r\rrbracket_{\text{Ch}}, \text{send}\right) \\
&\quad \&\&\ \texttt{tok}\left(\llbracket r\rrbracket_{\text{Ch}}.\text{send}_b, \llbracket a\rrbracket_{\text{Ch}}\right)
\end{aligned}
$$

$$
\begin{aligned}
&I\left(\llbracket \textbf{sends}\,(r, a)\rrbracket_{\text{Ch}}\right) := \\
&\quad \texttt{tok}\left(\llbracket r\rrbracket_{\text{Ch}}.\text{send}_u, \llbracket a\rrbracket_{\text{Ch}}\right) \\
&\quad \&\&\ \text{applyCredit}\left(\llbracket r\rrbracket_{\text{Ch}}, \text{send}\right)
\end{aligned}
$$

Termination Obligations

$$
\begin{aligned}
&I\left(\llbracket \textbf{terminates}\,(t)\rrbracket_{\text{Ch}}\right) := \\
&\quad \texttt{tok}\,(\text{thread}, \text{term}_b, 1)
\end{aligned}
$$

6.4 Obligation-modifying Statements

A method can gain termination obligations only via its precondition or via loop invariants. The only way to get rid of it, is to actually terminate. For the other obligations there are dedicated statements that allow the user to fulil obligations or accept new ones.

6.4.1 Channel Statements

In Chalice we need to declare Channel types on the global level. A Channel declaration has the form

$$
\begin{aligned}
&\textbf{channel}\ C\left((a : t)^{+}\right)\ \textbf{where}\ P \qquad \text{for}\ \ C : \text{channel name} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ a : \text{argument name} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ t : \text{argument type} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ P : \text{channel invariant}
\end{aligned}
$$

On the Silver level, a channel is represented by an ordinary object. We give this object the two sends token fields. Token fields start out with zero tokens.

$$
\begin{aligned}
&\llbracket \textbf{var}\ c := \textbf{new}\ C; \rrbracket_{\text{Ch}} \rightsquigarrow \\
&\quad \llbracket v\rrbracket_{\text{Ch}} := \textbf{new}\,(\text{send}_b, \text{send}_u)
\end{aligned}
$$

Listing 18: Chalice program with channels.

```
1 channel Chan(msg:int) where msg > 0;
2 method main(){
3   var x:int; var ch : Chan := new Chan;
4   fork worker(ch);
5   receive x := ch; assert x > 0;
6 }
7 method worker(ch:Chan) requires ch != null && sends(ch,1,1) {
8   send ch(15);
9 }
```

Send When we send a message on channel c, we need to substitute the message parameters in the channel invariant P with the message arguments A found at the send-site. We call this new invariant P'. For obligation assertions inside P' we also need to make sure that they only represent credits. It must not be possible to send obligations in messages because we have no guarantee that these messages are ever received. We have ensured this by only allowing `credit`, with its additional checks in channel invariants, and not the more general `sends`.

$$
\begin{aligned}
&[\![\textbf{send}\ c\ (A)\ ;]\!]_{\text{Ch}} \rightsquigarrow \\
&\quad \textbf{assert}\ [\![c]\!]_{\text{Ch}} != \text{null}; \\
&\quad \textbf{exhale}\ \texttt{tok}\left([\![c]\!]_{\text{Ch}}.\text{send}_b, 1\right)\ \&\&\ \text{applyCredit}\left([\![c]\!]_{\text{Ch}}, \text{send}\right); \\
&\quad \textbf{exhale}\ E\left([\![P']\!]_{\text{Ch}}\right)
\end{aligned}
$$

Receive To receive a message we declare a set V of fresh local variables to represent the elements of the received message. Then we create a modified invariant P' with all message parameters substituted by the corresponding fresh local variable from V. Only after we have inhaled the channel invariant do we assign the message elements to the destination variables X mentioned in the source program.

$$
\begin{aligned}
&[\![\textbf{receive}\ X^+ := c]\!]_{\text{Ch}} \rightsquigarrow \\
&\quad \textbf{assert}\ [\![c]\!]_{\text{Ch}} != \text{null}; \\
&\quad \textbf{assert} \\
&\qquad \left(\texttt{tokenamount}\left([\![c]\!]_{\text{Ch}}.\text{send}_b\right) + \texttt{tokenamount}\left([\![c]\!]_{\text{Ch}}.\text{send}_u\right)\right) \leq (-1); \\
&\quad \textbf{exhale}\ \texttt{tok}\left([\![c]\!]_{\text{Ch}}, \text{send}_b, -1\right); \\
&\quad \textbf{inhale}\ I\ (P') \\
&\quad [\![X^+]\!]_{\text{Ch}} := V^+
\end{aligned}
$$

To illustrate this, we have included a small example program in listing 18 where a `main` method creates a channel (line 3), forks a worker thread that promises to send a message (line 7) and then waits for that message to arrive (line 5). When we apply our translation scheme, we end up with a Silver program similar to that in listing 19. Some statements have been simplified. The **`fork`**, for instance, is much more verbose, but for this example, the only relevant line is the exhale of the precondition of the `worker` method. We have also omitted applyCredit $(\cdot)$ and lifetimeCheck $(\cdot)$ because they wouldn't apply anyway (we have zero obligations across the entire heap).

Listing 19: Simplified Silver translation of listing 18

```
1  token field sends_b: Int;
2  token field sends_u: Int;
3  method main(){
4    var x:Int; var Chan_msg: Int; var ch: Ref;
5    ch := new (sends_b, sends_u);
6    // fork worker is complicated but involves the following exhale
7    exhale ch != null
8      && (ch != null && tok(ch.sends_b, write));
9    assert ch != null;
10   assert (tokenamount(ch.sends_b) + tokenamount(ch.sends_u)) <= (-1)*write;
11   exhale tok(ch.sends_b, (-1)*write)
12   inhale Chan_msg > 0; // channel invariant
13   x := Chan_msg;
14   assert x > 0;
15 }
```

Exhaling the bounded send obligation on line 8 means we get a credit to receive on the channel `ch`. With that we can pass the check for and exhale the channel credit on lines 10 and 11 in exchange for inhaling the channel invariant on lines 12 and 13.

6.4.2 Lock Statements

The user can define a *monitor invariant* P for each Chalice class. Each of these invariants is translated into a Silver predicate p_I. We could insert the monitor invariant directly into inhale and exhale statements at the lock and unlock sites. But if we use predicates, not just does Silicon perform a well-formedness check on the predicate, it also handles parameter substitution for us.

Acquire Thus, to acquire a lock, we first check that all the prerequisites are met and then give ourselves full permission to the monitor invariant predicate p_I before we unfold the predicate.

$$
\begin{aligned}
&[\![\textbf{acquire}\ x]\!]_{\text{Ch}} \rightsquigarrow \\
&\quad \textbf{assert}\ [\![x]\!]_{\text{Ch}} != \text{null}; \\
&\quad \textbf{assert}\ \left(\texttt{tokenamount}\left([\![x]\!]_{\text{Ch}}.\text{rel}_b\right) + \texttt{tokenamount}\left([\![x]\!]_{\text{Ch}}.\text{rel}_u\right)\right) == 0; \\
&\quad \textbf{inhale}\ \texttt{tok}\left([\![x]\!]_{\text{Ch}}.\text{rel}_u, 1\right); \\
&\quad \textbf{inhale}\ \texttt{acc}\left([\![x]\!]_{\text{Ch}}.p_I\right); \\
&\quad \textbf{unfold}\left([\![x]\!]_{\text{Ch}}.p_I\right)
\end{aligned}
$$

Release The release works similarly in that we first check whether the prerequisites are met and then fold the monitor invariant into the predicate that represents it.

$$
\begin{aligned}
&[\![\texttt{release}\ x]\!]_{\text{Ch}} \rightsquigarrow \\
&\quad \textbf{assert}\ [\![x]\!]_{\text{Ch}} != \text{null}; \\
&\quad \textbf{assert}\ \left(\texttt{tokenamount}\left([\![x]\!]_{\text{Ch}}.\text{rel}_b\right) + \texttt{tokenamount}\left([\![x]\!]_{\text{Ch}}.\text{rel}_u\right)\right) == 1; \\
&\quad \texttt{exhale}\ \texttt{tok}\left([\![x]\!]_{\text{Ch}}.\text{rel}_b\right) \,\&\&\, \text{applyCredit}\left([\![x]\!]_{\text{Ch}}, \text{rel}\right); \\
&\quad \texttt{fold}\left([\![x]\!]_{\text{Ch}}.p_I\right); \\
&\quad \texttt{exhale}\ \texttt{acc}\left([\![x]\!]_{\text{Ch}}.p_I\right)
\end{aligned}
$$

Implicit Sharing In the original Chalice verifier, every object would start out *unshared*, that is it could not be locked by other threads. A method would first have to **share** the object. The **share** statement would check the monitor invariant and then transfer it and all the permissions it entails to the monitor. The share statement would determine the *lock level* of the object. That mechanism would incidentally also prevent programmers from acquiring the lock before the object was shared.

Since we don't have deadlock avoidance in our translation, users could construct objects and immediately acquire locks on them, potentially getting both the access permissions from the object creation *and* the ones stored in the monitor invariant. To prevent this, we have decided to include an implicit **share** statment at the end of every object allocation. This change is not compatible with some Chalice programs. See our evaluation section (7) for details on the impact.

6.4.3 Method Call

Silver already comes with a built-in **call** statement. It automatically exhales the precondition, allocates fresh values for the callee's output parameters and inhales the postcondition. Extremely convenient, but synchronous method calls are one of the most complicated constructs to translate. Not just is this one of the points where we need to perform a lifetime check, we also may or may not need to do a leak check.

Given a synchronous call statement for method m on object r with arguments A^+ and result variables X^+

$$\textbf{call}\ X^+ := r.m\,(A^+);$$

we first store the current obligation amount for `terminates` in a new local variable called *oldTerminates*. We also allocate a series of local variables to capture the values of the call arguments A^+. This is to make sure that we can refer to those exact arguments *after* the call, when some of the argument expressions might evaluate to a different value. We call these captured arguments B^+.

Once that's done, we translate m's precondition m_{pre} *with lifetime checks enabled* (section 6.8) and exhale it. The scope E for the lifetime check is the current method, that is, the lifetime expressions in the callee's precondition are compared to the lifetime expressions of the caller's preconditions on method entry.

After that, we compare the new obligation amount for `terminates` with the value we rememberd just before. If we have lost `terminates` obligation from the exhale, the method

m has promised to terminate. This means that it's legal to keep some obligations in the current frame and consequently we can skip the *leak check* (section 6.7). This is why it is important that the `terminates` obligation assertion does not prevent us from arriving at negative obligation amounts. If a non-terminating method calls a terminating one, we temporarily want to have negative one termination obligation so that we can observe that the caller promises to terminate.

After the leak check, we need to restore our previous obligation amount for `terminates`. We unfortunately cannot assign token amounts directly, only add or subtract tokens. Instead we exhale the difference between the current amount and the previous amount.

Instead of letting the callee's precondition manipulate our own `terminates` obligation amount, we could also have rewritten it to apply these changes to a separate *thread* object. Our approach has the advantage, that the leak check for `terminates` is automatically performed correctly, that is, the leak check will cause a program to fail verification if the caller has a termination obligation and the callee doesn't "take it away" before the leak check.

At this point, the call has "happened" and we can start to handle return values and the method's postcondition. As with receiving messages, we allocate fresh variables R^+ for all output parameters and use them to substitute result variables in the postcondition m_{post}. We implement `old`-expressions by replacing it with a labelled old expressions that points to a state label placed just before the method call.

$[\![$**call** $X^+ := r.m\,(A^+)]\!]_{\text{Ch}} \rightsquigarrow$

assert $[\![r]\!]_{\text{Ch}}! = \text{null}$;
var oldTerminates : Perm := `tokenamount` (thread.term$_b$) ;
var$^+$ $B^+ := A^+$; (includes receiver r)
statelabel before-call;
assert $[\![r]\!]_{\text{Ch}}! = \text{null}$
exhale $E\,(m_{\text{pre}})$; // (with lifetime check)
if (`tokenamount` (thread.term$_b$) $\geq$ oldTerminates) {
    leakcheck
}
exhale `tokenamount` (thread.term$_b$) $-$ oldTerminates;
//call "happens" here
var$^+$ R^+;
inhale $I\,(m_{\text{post}})$; // (using *before-call* for `old`)
$X^+ := R^+$

6.4.4 Asynchronous Method Call

Unlike ordinary method calls, asynchronous method calls have no direct equivalent in Silver. They have to be implemented manually even by the original Chalice2Silver translator. Asynchronous calls, in general, look like this:

```
fork tk := r.m(a1, a2, ..., an);
// later, maybe even in another method/thread
join r1, r2, ..., rn := tk;
```

The tricky part is the **`join`** statement. It doesn't necessarily have to be in the same method that performed the **`fork`**. It could have been passed to a method or even another thread. This means that when we want to inhale the forked method's postcondition when joining, none of the original method arguments will be around. The same goes for the state used to evaluate `old` expressions in. The only thing that you are guaranteed to have, is the reference to the join-token. Therefore, the entire postcondition must be expressed in terms of that token.

The Chalice2Sil translator thus generates and assigns fields on the token for every argument and every old expression in the forked method's postcondition. For the join statement, it uses those fields in place of the actual arguments and old expressions.

Forking The obligations model imposes some additional rules on asynchronous method calls. First, it is illegal to fork a method that mentions obligations in its postcondition (the `credit` assertion is allowed). We cannot be sure that a thread will ever be joined, thus such obligations might never be fulfilled. We implement this via a syntactic check on the callee's postcondition at each fork-site.

Second, a thread can only be joined if it promises to terminate. Thus when we create the thread, we check whether it promises to terminate. If it does so, we assign a special field on the token (`joinable`) to `true`. Only if this field is set can the thread be joined later.

Third, if the caller promises to terminate, any threads it forks must promise to terminate as well. This might seem counter-intuitive at first. After all, as long as the caller doesn't hand over obligations to the diverging thread, it is still in control. If we allow terminating threads to fork non-terminating threads, something like the following can happen:

```
Thread A: terminates(1)
  channel c;
  fork B(c);
  receive t:= c;
  join t;
Thread B: sends(c, 1, 1)
  fork t := D;
  send c(t);
Thread D: terminates(2)
```

Here thread *A* uses the diverging thread *B* as a "trampoline" to spawn thread *D*. Thread *B* returns the join-token for *D* back to *A* via a channel. When *A* now waits for *D* to terminate, it has effectively managed to call a method with a longer lifetime than itself. This is why *B* needs to promise termination for the fork in *A* to be legal. Once the termination obligation is present, the lifetime check during the call makes sure that this scenario cannot occur.

To translate forking, we use a set of fields to capture arguments (B^+) and old expressions in the postcondition (O^+). We also remember the old termination obligation amount in the local variable *oldTerminates*. Instead of a leak check, we only check that the callee has not left us with our termination obligation (if we had any in the first place). We do, however, perform a lifetime check for every obligation transferred to the new thread. The scope E against which the lifetime expressions of the callee are compared, is the surrounding method, just like with synchronous method calls.

$[\![\textbf{fork}\ t := r.m\,(A^+)]\!]_{\text{Ch}} \rightsquigarrow$
  assert $[\![r]\!]_{\text{Ch}} != \text{null}$;
  var oldTerminates : Perm := `tokenamount` (thread.term$_b$) ;
  $[\![t]\!]_{\text{Ch}}$:= **new** (joinable, B^+, O^+) ;
  $t.B^+ := A^+$; (includes receiver r)
  $t.O^+ := \left(\texttt{old}\text{ expressions in } m_{\text{post}}\right)^+$;
  exhale $E\left(m_{\text{pre}}\right)$; // (with lifetime check)
  assert `tokenamount` (thread.term$_b$) ≤ 0;
  t.joinable := `tokenamount` (thread.term$_b$) $<$ oldTerminates;
  exhale `tokenamount` (thread.term$_b$) $-$ oldTerminates;

Joining To join a thread, we first need to make sure that the token is joinable. Then we inhale the forked method's postcondition using the generated fields on the token for arguments and old expressions. Finally, we need to set the joinable field to false, because a thread must only be joined once. If it were possible to join a thread multiple times, one could inhale it's postcondition multiple times, including any permissions it includes.

$[\![\textbf{join}\ R^+ := t]\!]_{\text{Ch}} \rightsquigarrow$
  assert $t != \text{null}$;
  assert t.joinable;
  inhale $I\left(m_{\text{post}}\right)$; //using fields of t
  t.joinable := `false`;

The joinable field is special in that the Chalice Frontend does not allow the programmer to assign to it in Chalice code. It can only be assigned through **`fork`** and `join`.

6.5 Methods

A method must set up the *thread* variable to hold `terminates` obligations, then inhale the method's precondition. This point is then marked using a statelabel called *entry*. The method body and postcondition follow before we fulfil the termination obligation and perform the method's leak check. Figure 3 contains the translation full scheme for method bodies. The translation of leakcheck is presented in section 6.7

$[\![$**method** $m\ (p)$ **requires** A_{pre}; **ensures** $A_{\text{post}}\ \{\ S\ \}\ ]\!]_{\text{Ch}}\ \rightsquigarrow$

method $m\ (p)$ **requires** true; **ensures** true {

  var thread := **new** (term_b);

  inhale $I\ (A_{\text{pre}})$;

  statelabel entry;

  S

  exhale tok $(\text{thread}, \text{term}_b)$; // fulfil thread termination

  exhale $E\ (A_{\text{post}})$;

  leakcheck

}

Figure 3: Translation scheme for method bodies

Listing 20: Chalice program with a postcondition that is not self-framing.

Well-Formedness Checks

Pre- and postconditions must be *self-framing*, that is they must come with all permissions necessary to reason about the memory locations they mention. For instance the postcondition in the method in listing 20 mentions `x.f` but does not contain `acc(x.f)`. Nonetheless, the method will pass verification using our translation scheme, because access to `x.f` will be available when the postcondition is exhaled at the end of the method. It was granted as part of the precondition, after all. Any *caller*, on the other hand, will inhale this postcondition with the statement about `x.f` without having permission to talk about `x.f`.

Silicon automatically performs some well-formedness checks for regular method postconditions. To do this, it first inhales the method's precondition, then throws away the heap, starts over with an empty one and inhales the postcondition. If the postcondition itself doesn't include enough permission, Silicon will notice.

Unfortunately, we don't have enough control over the program state on the Silver level to use the same trick. There is no way for us to "empty" the heap. What we can do, is just inhale the method's postcondition on its own, without the precondition. This is a much stricter check, because the postcondition will have to repeat some facts that are already included in the precondition.

While inconvenient, this is the same scenario as when someone inhales a postcondition during a join of a token that has been passed from another thread. In that scenario, too, the method that is inhaling the postcondition has not seen the guarantees that the precondition provides.

When we perform our version of the well-formedness check, we don't want to get in the way of the actual method verification. Therefore, we create a separate method to hold that inhale of the postcondition. To mimic the most restrictive scenario that a postcondition will ever have to be inhaled in, we act as if our well-formedness-check method received a

join-token for the original method.

$[\![$**method** $m\ (p)$ **requires** A_{pre}; **ensures** $A_{\text{post}}\ \{\ S\ \}\ ]\!]_{\text{Ch}}\ \rightsquigarrow$
method $m\ (t : Ref)$
    requires t != null && **acc** (t.joinable, **write**) && t.joinable;
    ensures true
{
    inhale $I\left(A_{\text{post}}\right)$; // using fields from token t
}

This approach breaks a number of programs that use Chalice functions inside `old` expressions because those appear as opaque fields to Silicon (see section 7).

6.6 Loops

For translation, a loop is a bit like combination between a method call and a method body implementation. The exhaling of the loop's invariant A resembles that of a method call. We remember the current amount of termination obligation, then exhale the invariant and check afterwards whether the loop has promised to terminate. If not, we need to perform a leak check.

Inside the loop, we inhale the loop's invariant mark that point using a statelabel $loopEntry_N$ (each loop in a method gets its own). Then follows the body and an exhale of the loop invariant *with lifetime checks* relative to the state on entry to the loop (section 6.8). Finally, we perform the loop's leak check (section 6.7).

Note that we don't manually fulfil the termination obligation. We expect the user's loop to correctly pass the termination obligation from each loop iteration to the next. We also do not revert the changes to the `terminates` obligation amount during the initial loop invariant exhale but instead expect the program to arrive back at the same amount by the end of the loop. Figure 4 contains the full translation scheme for loops.

6.7 Leak check

The leak check needs to make sure that, at the point where it is performed, no more obligations, bounded or unbounded, are in the current frame. We use the **`forallrefs`** expression to have the verifier expand our condition over all objects the current frame has knowledge about. We use separate assert statements so that when one of them fails, the user has an idea as to what kind of obligation they may have leaked.

$[\![\textbf{while}\ (e)\ \textbf{invariant}\ A\ \{\ S\ \}]\!]_{\text{Ch}} \rightsquigarrow$
var oldTerminates : Perm := `tokenamount` (thread.term$_b$) ;
exhale E (A) ;
if (`tokenamount` (thread.term$_b$) ≥ oldTerminates) {
    leakcheck
}
var nondet$_N$: Bool;
while (nondet$_N$) {
    inhale I (A) && $[\![e]\!]_{\text{Ch}}$;
    statelabel loopEntry$_N$;
    S
    exhale E (A) ; // (with lifetime check using E = loopEntry$_N$)
    leakcheck
}
inhale I (A) && ! $[\![e]\!]_{\text{Ch}}$
assert `tokenamount` (thread.term$_b$) == oldTerminates

Figure 4: Translation scheme for loops

checkField (f) :=
    forallrefs[f_b, f_u] r : `tokenamount` (r, f_b) + `tokenamount` (r, f_u) ≤ 0
leakcheck :=
    exhale checkField (rel)
    exhale checkField (send)
    exhale checkField (term)

6.8 Lifetime check

The purpose of the lifetime check is to ensure that a thread cannot hold on to an obligation for an unbounded amount of time. Lifetime checks happen when an obligation gets transferred away to another frame. This could be a called method or the next loop iteration.

A lifetime check is always a comparison between two states: the state at the entry into the current frame and the state at the moment of transfer. It is also a comparison between two lifetime expressions: the one from the precondition of the current frame and the one from the precondition of the frame the obligation is being transferred to.

Where to place which comparisons is entirely based on the structure of the program. When we encounter an obligation assertion while we translate for an **exhale** with lifetime checks enabled, we look for matching obligation assertions in the surrounding scope and emit a comparison.

Say we have the following very simple situation

```
method main(x:A) requires releases(x, T1)
  statelabel entry;
  ...
  call sub(x);
method sub(x:A) requires releases(x, T2)
```

Here we would assert `T2 < old[entry](T1)` just before we exhale the `releases` obligation. Naturally, we can run into situations where we have multiple releases obligations in the caller's precondition for multiple objects as in the following example

```
method main(x:A, y:A, z:B) requires x != y && y != z && x != z
    && releases(x, T1) && releases(y, T2) && releases(z, T3)
    && terminates(T4)
  statelabel E;
  ...
  call sub(temp);
method sub(v:A) requires releases(v, T5) && terminates(T6)
```

Just from the structure of the program, we cannot decide which of the three input references `x`, `y` or `z` will be passed to `sub`. We have to conservatively compare the callee's lifetime expression with all lifetime expressions found in the caller's precondition. We can exclude a pairing from the comparison just based on program structure if it concerns (1) different kinds of obligations or (2) objects of incompatible Chalice types. Therefore the last example would result in the following two lifetime checks:

① $(\text{temp} == x \Longrightarrow T5 < \mathbf{old}\,[E]\,(T1))$
$\quad \&\&\ (\text{temp} == y \Longrightarrow T5 < \mathbf{old}\,[E]\,(T2))$

② $(\text{thread} == \text{thread} \Longrightarrow T6 < T4)$

So even though lifetime checks technically require a quadratic number of object reference comparisons in the worst case, in typical programs the separation by Chalice type will hopefully cut down the number of comparisons to at most a handful per call.

Another situation to watch out for is when the obligation assertion in the caller's precondition is guarded by an implication. In that case, we cannot be sure if the caller's lifetime expression is well defined. Consider the following example:

```
method main(x:A) requires C1 => releases(x, T1)
  statelabel E;
  ...
  call sub(x)
method sub(x:A) requires C2 => releases(x, T2)
```

When we are translating the exhale of sub's precondition, we will come past `C2` on the way to `releases(x, T2)`, so we don't need to worry about that. However, we have to repeat every implication left-hand side encountered on the way to the caller's lifetime expression. Consequently the lifetime check for this example looks like this:

$$\mathbf{old}\,[E]\,(C1) \Longrightarrow (temp == x) \Longrightarrow T2 < \mathbf{old}\,[E]\,(T1)$$

Finally, we only want to perform a lifetime check for bounded obligations, that is for objects that have a positive token amount in the corresponding bounded field. We also need to ensure that the lifetime expression does not fall outside the well-founded set. The check from before would therefore be:

$$
\begin{aligned}
&\texttt{tokenamount}\,(temp.\mathrm{rel}_b) > 0 \\
&\quad \Longrightarrow 0 \leq T2 \\
&\qquad \&\&\ (\mathbf{old}\,[E]\,(C1) \\
&\qquad\quad \Longrightarrow (temp == x) \\
&\qquad\qquad \Longrightarrow T2 < \mathbf{old}\,[E]\,(T1))
\end{aligned}
$$

To implement this, we prepare set of lifetime constraints for each method precondition and loop invariant. This set of constraints can be looked up via the statelabels assigned to theses scopes: $L(E)$. Such a set consists of elements (r, τ, f, t, C) with a receiver expression r, a chalice type τ, an obligation kind f, a lifetime expression t and a sequence of implication left-hand side expressions C. Armed with this the following block of pseudo code which for some receiver expression r_1, obligation kind f_1, lifetime expression t_1 and enclosing scope E assembles lifetimeCheck (r_1, f_1, t_1, E)

procedure lifetimeCheck (r_1, f_1, t_1, E) :
var checks := $[\![\texttt{true}]\!]_{\text{silver}}$
var τ_1 := `chalice-typeof` (r_1)
`for` $(r_0, \tau_0, f_0, t_0, C)$ **in** $L(E)$ {
  if $(\tau_0 = \tau_1 \ \wedge \ f_0 = f_1)$ {
    var check := $[\![\,([\![r_0]\!]_{\text{Ch}} == [\![r_1]\!]_{\text{Ch}}) \Rightarrow [\![t_1]\!]_{\text{Ch}} < \mathbf{old}\,[E]\,([\![t_0]\!]_{\text{Ch}})]\!]_{\text{silver}}$
    `for` c **in** C {
      check := $[\![\mathbf{old}\,[E]\,([\![c]\!]_{\text{Ch}}) \Rightarrow \text{check}]\!]_{\text{silver}}$
    }
    checks := $[\![\text{checks \&\& check}]\!]_{\text{silver}}$
  }
}
$[\![\,(\texttt{tokenamount}\,([\![r_1]\!]_{\text{Ch}}.f_{1b}) > 0) \Rightarrow 0 \leq [\![t_1]\!]_{\text{Ch}} \&\&\ \text{checks}]\!]_{\text{silver}}$

The enclosing scope E is "loopEntry$_N$" of the loop for the lifetime check during the exhale of that loop's invariant at the end the loop's body and "entry" for all other lifetime checks (method calls).

6.9 Deviation from the original scheme

Our encoding of modular verification of finite blocking is not exactly what [BM14] presented. The primary difference is how we handle the lifetime of obligations.

The original verification technique would have assigned each obligation $x.f$ an integer "countdown timer". Every time an obligation enters a scope (method body, loop body),

the counters of all objects would be reduced by one. When exhaling an obligation, we would then have checked whether the lifetime value of the obligation assertion fit within the remaining time on the obligation. To implement the "ticking down" part of this scheme, we would have needed to implement something like this:

```
foreach x in Heap do {
  x.f.lifetime := x.f.lifetime - 1;
}
```

A form of "foreach" over the entire heap. This would certainly have been more difficult to implement than **forallrefs** as it manipulates the heap. Our approach with statically emitted lifetime comparisons has the advantage that we don't such a "foreach" statement in the intermediate language. It should also be easier to adapt to more interesting well-founded sets than integers since our lifetime values never need to be stored on the heap.

7 Evaluation

In this section we evaluate our implementation of the verification technique for finite blocking. To that end we have come up with a number of larger example programs to verify using our extensions.

7.1 Parallel Binary Tree Processing

Listing 21 shows an example adapted from Chalice2SIL's existing test suite ("`TreeOfWorker.chalice`"). The idea is for the method `work` to recursively descend through the tree and perform some work on the shared piece of `data` for each node. To take advantage of multi-core systems, it spawns a separate thread for each of the two sub-trees of every node.

Since we want to wait for the results of the computations on the sub-trees for each node, the method `work` must promise termination (otherwise, the spawned threads cannot be joined). We are using the tree's remaining maximum height as the termination measure. Each node must be explicitly annotated with their height within the overall tree.

As in the original version of this example, we use the predicate `valid` to reason about the properties of a correctly constructed binary tree. Using predicates is vital for recursive data structures, because they can be formulated recursively. We extend the `valid` predicate with a condition that makes sure that the node's `height` annotations decrease steadily as we move down the tree.

You will notice that we don't use the **`this.`**`height` directly in the `terminates` assertion. Because **`this.`**`height` is only accessible while the `valid` predicate is unfolded, we would have had to write

```
requires unfolding terminates(this.height)
```

but our translation does not support this. Having the obligation expression inside an **`unfolding`** expression complicates the encoding of lifetime checks. This limitation can most likely be overcome if one were to also track which predicates were unfolded in the

Listing 21: Parallel binary tree processing in Chalice.

```
class Node {
  var l: Node; var r: Node; var height: int;

  method work(data: Data, callHeight: int)
    requires rd(data.f) && valid
    requires unfolding valid in callHeight == this.height
    requires terminates(callHeight)
    ensures rd(data.f) && valid
  {
    var tkl: token<Node.work>
    var tkr: token<Node.work>

    unfold valid
    if (l != null) { fork tkl := l.work(data, this.height-1) }
    if (r != null) { fork tkr := r.work(data, this.height-1) }
    /* .. perform work on this node (using the global data: data.f) */
    if (l != null) { join tkl }
    if (r != null) { join tkr }
    fold valid
  }

  predicate valid {
    acc(l) && acc(r) && acc(height,10) && height >= 0 &&
    (l != null ==> l.valid && acc(l.height,10) && l.height == height - 1)
       &&
    (r != null ==> r.valid && acc(r.height,10) && r.height == height - 1)
  }
}
class Data { var f: int; }
```

context of each lifetime constraint in the method's precondition and repeat the **unfolding** expressions in the lifetime check as necessary. A more detailed investigation will have to determine whether such a scheme would be sound.

7.2 Producer-Consumer

Listing 22 contains a program that spawns two threads, a producer and a consumer. The producer continuously sends messages and then lets some external process determine whether to continue sending messages. Here, our external process is a fake random number generator. The consumer keeps processing messages and automatically shuts itself down once no more messages are coming. The idea is that each message consists of two parts: the data itself and a boolean flag that indicates whether at least one more message will follow. If the flag is set, the message is also accompanied by a credit to received the next message.

Unfortunately, this example will fail to verify. The verifier will complain that the lifetime

Listing 22: Producer-Consumer example in Chalice

```
class A {
  method getRandomNumber() returns (x:int)
    requires terminates(1) { x := 4; }

  method producer(d:D) requires d != null && sends(d,1,1)
  {
    var abort := false;
    while(!abort)
      invariant (!abort) ==> sends(d,1,1) // one msg per iteration
    {
      var x : int; var data: Data;
      call x := getRandomNumber();
      if(x == 77){
        abort := true;
        send d(null,false); // signal end of message stream
      } else {
        data := new Data { f := x };
        send d(data, true); // signal that more messages are to come
      }
    }
  }
  method consumer(d:D) requires d != null && credit(d,1)
  {
    var hasNext:bool := true;
    while(hasNext)
      invariant credit(d,ite(hasNext,1,0))
    {
      var data : Data;
      receive data, hasNext := d;
    }
  }
}
class Data { var f: int }
```

constraint of the channel, `sends(d,1,1)`, was violated. This makes sense and highlights a shortcoming of our handling of credits. When we send a message with a promise that at least one other message will follow, we effectively do

```
exhale sends(d, -1, _)
```

In our encoding credits always apply to the *bounded* token field. Thus, when we send that message and exhale the credit, we get a *bounded* obligation. Once we reach the lifetime check, the system sees that we have a bounded send obligation and compares the lifetime expressions. The original verification technique did not have this incompleteness, because of it encodes the lifetime check.

Workaround

Listing 23: Workaround for producer from listing 22

```
1  class A {
2      method producer(d:D)
3          requires d != null && sends(d,1,1) // promise to send at least one
4      {
5          var dNext: D := d; var abort := false; var x : int;
6          while(!abort)
7              invariant dNext != null
8              invariant (!abort) ==> sends(dNext,1,1)
9          {
10             call x := getRandomNumber();
11             var newD: D := new D;
12             if(x == 77){
13                 abort := true;
14                 send dNext(newD,false); // signal end of message stream
15             } else {
16                 send dNext(newD,true); // signal more messages are to come
17             }
18             dNext := newD;
19         }
20     }
21 }
22 channel D(c:D,b:bool) where c != null && credit(c,ite(b,1,0)) && this != c;
```

There is, however, a way around that problem in this particular instance: each time we intend to send a credit, we instead send an entirely new channel with a credit (see listing 23). When our lifetime mechanism tries to perform its check, it sees that the channel object from the beginning of the loop invariant is different from the channel object at the end of the iteration. The system is satisfied, that the obligation must be new.

7.3 Bi-Directional Channel

The workaround for the producer-consumer scenario doesn't work in all cases. Listing 24 contains a program where one "data" channel (D) is used by a pair of threads to send messages in both directions. Once a thread has sent a message on the channel, it uses the same channel to receive a message and then goes back to sending etc.

We first tried to construct such an example using only a single channel. This is, however, not possible using our obligation model. Since we cannot rely on messages being received, one can never send an obligation across a channel. Thus, one thread could never hand the other thread the obligation to reply. We thus make use of two auxiliary channels over which we receive our obligation. The idea is that each of the two threads performs a sort of "long polling" on its dedicated auxiliary channel. The other thread, by convention,

Say thread *A* has just received a message on the data channel (line 35) from thread *B*. Over the auxiliary channel it sends the credit for the message that it will send in the next loop

Listing 24: Program that uses one channel in both directions.

```
method pingpong(d:D, tIn:T, tOut:T,initial:bool)
  requires d != null && tIn != null && tOut != null && tIn != tOut;
  requires sends(tOut,1,1) && credit(tIn,1)
  requires sends(d,ite(initial,-1,1),1)
{
  var numLeft:int; var sending:bool; var receiving:bool;
  if(initial){ sending := false; receiving := true;  }
      else { sending := true;  receiving := false; }
  var tInNext:T := tIn; var dNext:D := d; var tOutNext: T := tOut;

  while(receiving || sending)
    invariant !(sending && receiving)
    invariant dNext != null && tInNext != null && tOutNext != null
    invariant tInNext != tOutNext
    invariant credit(tInNext,ite(receiving||sending,1,0))
    invariant sends(tOutNext, ite(receiving||sending,1,0),
                  ite(sending,2,1))
    invariant sends(dNext,
      ite(receiving,-1,0) + ite(sending,1,0), 1)
  {
    if(sending && !receiving){
      sending := false;
      send dNext(15);
      receive tInNext, dNext, receiving := tInNext;
      assume tInNext != tOutNext; // fails to verify without
      if(!receiving){ // close channel
        var newTOut:T := new T;
        send tOutNext(newTOut,dNext,false);
        tOutNext := newTOut;
      }
    } else if(receiving && !sending) {
      receiving := false;
      var x:int; call x := getRandomNumber();
      sending := x == 77;
      receive x := dNext;
      dNext := new D;
      var newTOut:T := new T;
      send tOutNext(newTOut,dNext,sending);
      tOutNext := newTOut;
    }
  }
}
channel T(t:T, d:D, hasNext:bool) where t != null && d != null
        && credit(t,ite(hasNext,1,0)) && credit(d,ite(hasNext,1,0))
channel D(x:int);
```

iteration to thread *B*. It is now in "sending" mode itself. In the next iteration it will send a message to *B* and then transitions into "receiving" mode. To do so, it needs to receive the necessary credit on line 24. And so on. Note how the "receiving" thread must also send a message on its auxiliary channel to signal the end of the exchange.

As with the fixed producer-consumer example in listing 23, we use the workaround to create a fresh channel every time we send a credit. This doesn't work that well in this scenario because the verifier cannot be sure that `tInNext` = tOutNext!. If the verifier needs to allow for this possibility, then our translation will compare the lifetime constraints of `tInNext` with those of `tOutNext` and fail. In the producer-consumer case, we were able to state in the channel invariant that **`this`** = c! (`c` is the transferred channel). We can't do similar thing here because the channel invariant cannot talk about the callers entire environment. The example in listing 24 thus only verifies thanks to the manually added assumption on line 25 along with the corresponding additional loop invariant.

```
assume tInNext != tOutNext
```

Unrelated to this problem, the example might also be difficult to verify if we had deadlock avoidance mechanisms in place because of how the three involved channels depend on one another.

7.4 Well-Formedness Check

Listing 25: Well-Formedness check does not detect `old(x)`.

```
1  class Test {
2    var x: int
3    var tk: token<Test.incX>
4    predicate V { acc(x) }
5
6    method incX()
7      requires V && terminates(1)
8      ensures V
9    {
10     unfold V
11     x := x + 1
12     fold V
13   }
14
15   method joinTk()
16     requires acc(tk) && tk != null && acc(tk.joinable) && tk.joinable
17     ensures V
18     ensures unfolding V in x == old(x) // ERROR: old(x) is not readable
19   {
20     join tk
21     assert V
22   }
23 }
```

While going through the existing test suite of Chalice, we came across `workitem-10194.chalice`.

A slightly abbreviated version is included in listing 25. In the postcondition of the method `joinTk`, we have the following `old` expression

```
old(x)
```

When we perform our well-formedness check this old expression appears as an opaque field and thus won't get checked by Silicon. This is one indication that it might be better to leave well-formedness checks to the verifier and use the proper mechanisms for encoding pre- and postconditions. Since we have no direct control over the program state (especially the heap) it is difficult to emit well-formedness checks in exactly the same way that Silicon can do it.

7.5 Alternating Conditions

The program in listing 26 is a particularly pathological program provided by our supervisors. It contains a loop that tries to alternate between two loop invariants in the hope of being able to trick our lifetime check. In one iteration, b is set and the termination measure is x whereas the next iteration, with `b = false`, uses y as its termination measure.

We have had a number of ideas for lifetime checks that didn't work when confronted with this program. With the lifetime check as it exists now, the verifier correctly reports that it cannot satisfy the loop invariant due to the lifetime constraint on the termination obligation. The condition that catches the error looks something like this

```
tokenamount(thread.term_b) > none ==>
  ((y >= 0)
  && old[loop_entry](b) ==>
    (old[loop_entry](thread) == thread) ==>
      y < old[loop_entry](x)
```

In other words, since we are using the old value of b to determine whether the lifetime constraint of the precondition applies, we correctly compare the current `y` to **`old`**`(x)` instead of **`old`**`(y)`.

7.6 Existing Test Suite

As part of this thesis, we have contributed a collection of 261 test methods spread across 48 files. Some of them are simplified simplified versions of others intended for debugging, though. But we have also gone through the existing test suites of Chalice2Silver to see which test cases were still working, which needed adaptation and why.

In many cases, we had to change the expected error messages because we are not using ordinary pre- and postconditions. So an error message that said "postcondition.violated" before, now reads "exhale.failed". There were a handful of cases that we had to ignore. Some because they used features that were not in scope for this work (backpointers are one example), others because our changes broke them.

Of the broken test cases, we had those that no longer worked because our well-formedness checks were inadequate. Mostly they failed to show that the postconditions in question were valid, but we also had cases where illegal postcondition were silently accepted, like in the example above. The other category of broken test cases have to do with our decision to

Listing 26: Chalice program that alternates between two loop invariants.

```
method main() {
    var b : bool := true;
    var x : int := 5; var y : int := 6;
    while(true)
        invariant b ==> terminates(x) && x == 5 && y == 6
        invariant (!b) ==> terminates(y)  && x == 5 && y == 6
    {
        if(b) { y := y - 1; x := x + 1; }
        else  { x := x - 1; y := y + 1; }
        b := !b;

        assert b ==> x == 5 && y == 6;
        assert !b ==> y == 5 && x == 6;
    }
}
```

implicitly share objects just after they are created. While the vast majority of programs affected by that change could be adapted, some were impossible change accordingly.

When we had to make changes that went beyond swapping out expected error messages, we encountered one of two scenarios. One was the aforementioned implicit sharing on object creation. To fix that, we would use initializer expressions instead of assigning fields one-by-one after the object was created. **`share`** statements needed to be removed as well. The other scenario were cases where test cases attempted to join threads. The methods that those threads were forked from of course didn't have termination annotations. In most cases, they trivially terminated and the annotation was quickly added. In some few cases, the programs tried to join threads that were never going to terminate. There we removed the **`join`** statements.

A summary of the results can be found in table 1. We have marked test cases that we had to touch with keywords. Broken test cases are marked with `BROKEN`, changed ones with `MANUAL` and ones where we only had to swap around error message with `EXPECTATIONS`.

Original test files	243			As is	185
		Changed	40	Termination	24
				Implicit Share	16
		Broken	12	Well-Formedness	8
				Implicit Share	4

Table 1: Statistics from survey of existing Chalice test suite

8 Conclusion

In this thesis, we adapted the technique for modular verification of finite blocking presented in [BM14] to be implemented on top of the Viper verification infrastructure. We designed sensible extensions to Viper's intermediate language Silver that would not only make our encoding of the technique possible but that would also be useful for users of Silver. We have implemented those extensions in the symbolic execution based verifier backend Silicon and we have implemented parts of the encoding in Chalice2Silver: obligation leak checks and lifetime checks. We have evaluated our implementation against a series of examples and the existing test suites.

8.1 Future Work

The original verification technique in [BM14] also includes a *deadlock prevention*. When we designed our extensions, we laid the groundwork in the hope that we would be able to also implement the deadlock pervention mechanism. The idea was to use the *value* of token fields to store an object's *lock level* and implement a *levelBelow* operator using the newly introduced **`forallrefs`** pseudo-quantifier.

Another extension that we had in mind, was to use a more general well-founded set for our lifetime expressions, such as something inspired by Dafny [Lei10]. Since our lifetime expressions never have to be stored in a variable or field, one could implement a form of "compile-time overloading" of the lifetime comparison operator ($<$) to allow the comparison of tuples, sequences, sets etc.

8.2 Related Work

Some of the properties we are looking for with modular verification of finite blocking, such as guaranteeing that lock release obligations cannot be leaked, can also be provided by type systems with ownership types [CPN98]. In some systems with a concept of ownership, the lock can allocate a "handle" when it is acquired. The lock will be released when this handle leaves a scope and is no longer owned. The handle is not tied to the scope it was initially created in. Ownership of it can be transferred between methods. The systems-oriented programming language *Rust* [Rus12] makes extensive use of this.

Chalice's [LMS09, LMS10] own channel implementation comes remarkably close. It has a concept of "channel debt" that can be passed around. With its working deadlock prevention mechanism, it can detect many faulty programs that our system currently cannot.

Dafny [Lei10] features a very sophisticated mechanism for specifying termination measures. It for instance supports using tuples and sequences in the termination measures, which allow the system to verify termination for more complex scenarios, such as the Ackermann function.

8.3 Acknowledgements

I would like to thank my supervisors Prof. Dr. Peter Müller and Dr. Ioannis Kassios for supporting me and making this Master's Thesis possible. I would also like to thank Malte Schwerhoff and Dr. Alex Summers for various discussions and for helping get started with the internals of Silicon. I've had the pleasure to work with a complex code basis in the form of Chalice (parser), Chalice2Silver (translator), Silver (intermediate language and Silicon (verifier) to which many people before me, both members of the Viper team and students, have contributed. Having a build system with a large and fully automated test suite is enormously helpful. Finally, I'd like to thank family and friend who have supported me during this period.

References

[BDJ+06] M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, volume 4111 of Lecture*. Springer, 2006.

[BM14] P. Boström and P. Müller. Modular verification of finite blocking in non-terminating programs. Unpublished draft, 2014.

[CPN98] D.G. Clarke, J.M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[dMB08] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008, volume 4963 of LNCS*, pages 337–340. Springer, 2008.

[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.

[Kla12] C. Klauser. Translating Chalice into SIL, 2012.

[Kob06] N. Kobayashi. A new type system for deadlock-free processes. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer Berlin Heidelberg, 2006.

[Lei10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

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

[LMS10] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In AndrewD. Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 407–426. Springer Berlin Heidelberg, 2010.

[Rus12] The Rust programming language. http://www.rust-lang.org/, 2012. [Online; accessed 30-November-2014].

[Sch11] M. Schwerhoff. Symbolic execution for Chalice. Master's thesis, Eidgenössische Technische Hochschule Zürich, 2011, 2011.

Listing 27: Chalice program that uses the `work` method from listing 21

```
class Program {
  method main()
    requires terminates(5) // make sure that work promises to terminate
  {
    var a : Node := new Node { l := null, r := null, height := 1 };
    var c : Node := new Node { l := null, r := null, height := 0 };
    var d : Node := new Node { l := null, r := null, height := 0 };
    var b : Node := new Node { l := c,    r := d,    height := 1 };
    var r : Node := new Node { l := a,    r := b,    height := 2 };

    // validate tree bottom up
    fold r.l.valid;
    fold r.r.l.valid;
    fold r.r.r.valid;
    fold r.r.valid;
    fold r.valid;

    var data : Data := new Data { f := 15 };

    call r.work(data, r.height);

    data.f := 7; // make sure we got full access to f back

    // make sure we have the valid predicate back
    unfold r.valid;
    if(r.l != null) { // we cannot be sure that work didn't change the tree
      unfold r.l.valid;
    }
  }
}
```

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Modular Verification of Finite Blocking

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Klauser	Christian

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Neuenhof, 26.11.2014	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.