Runtime Checking for Chalice

Bachelor Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zürich

By: Heinz Hegi
Supervised by: Dr. Alexander J. Summers
Prof. Dr. Peter Müller
Date: February 28, 2013

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Contents

1 Abstract

Chalice is a static verification language for concurrent programs. Before this project, there was a code generator that translated Chalice programs into C#, by looking at operationally relevant statements and ignoring all the proving constructs. The goal of this project is to extend the existing code generator to mimic the verification constructs at runtime, and thus make detailed runtime checking of output programs possible. The main focus of this report lies on the runtime representation and tracking of permissions, which are the Chalice constructs that ensure the absence of race conditions.

2 Introduction

Chalice ([LM09] [LMS10] [LMS09]) is a static verifier for concurrent programs. It ensures absence of race conditions by associating all field locations with a Permission, which is a value between 0 and 100%. Write and read access are only permitted if enough Permission is held: 100% is required for write access, and any value bigger than 0% gives read access. Permissions can be passed around through *inhaling* and *exhaling*. Inhaling is the act of gaining Permission and assuming information about the state. This takes place in several scenarios like for example the start of a new method (precondition is inhaled) or acquiring a lock (monitor invariant is inhaled). Exhaling is the act of giving Permissions away and guaranteeing facts about the state. It is symmetrical to inhaling and happens for example in a fork (precondition is exhaled) or a lock release (monitor invariant is exhaled).
Additionally, Chalice ensures the absence of deadlocks by enforcing an ordering of locks, which must be followed when acquiring multiple locks. It is realized using a "mu" field, which is associated with all shared objects and defines the placement of the associated lock in the ordering.

There already existed a code generator prior to this thesis, which translated Chalice code into C#. However, it only translated the operationally relevant code and ignored all verification constructs, which constitute a substantial part of the Chalice language.

The goal of this thesis is to expand on the existing code generation, so that these constructs have an appropriate representation at runtime and can be used to check the validity of the execution. The runtime checking should resemble the verifier's behavior, in order to provide insight into the dynamic aspects of the verified programs. Since the verifier isn't complete, it could be that a program is rejected even though it is semantically correct. In this case, the runtime checking could be used to examine a program at runtime and hopefully get a better picture of the problem at hand.
We use the .NET tool `Code Contracts` for the runtime checks in the C# output.

This report's main chapters are divided into the sections Approach and Implementation. The Approach (Section 3) describes the general idea behind the representation of Chalice constructs in C#. The Implementation (Section 4) explains the translation and functionality of the generated classes in greater detail. Finally, the code generation's testing is described in Section 5, and the results are discussed in Section 6.

3 Approach

3.1 Method Modularity versus Thread Modularity

The Chalice verifier works modularly on members, i.e. each method is verified without knowledge about the state at the caller-site. This requires exact pre- and postconditions for all methods. At runtime, we can be more permissive by tracking the permissions per-thread (instead of per-method). As a consequence, a precondition that requires too few permissions doesn't hurt, as long as it's only called by a thread that has all necessary permissions for the body. This doesn't work for forks, since the new thread will have *only* the permissions inhaled from the precondition. Also, permission is never leaked, even if a postcondition is too lax.

Another option would have been to track the permissions method-modularly at runtime as well. This would lead to inhaling and exhaling for method calls, similar to the way it works for forks now. In this case, we would also need to store the state of the caller after exhaling, so that we can pick up where we left when returning from the current method.
Additionally, permissions could be intentionally leaked if they are not mentioned in the postcondition, but this wouldn't relieve us from having to track the exact values throughout the method, since we still need to know the exact abstract amounts of permissions, which are computed dynamically. Therefore, leaking permissions would actually be more complicated than always keeping them.

3.2 Permissions

3.2.1 Permissions in Chalice

Permissions are associated with each field location of each object. A permission denotes the access level that the current thread has to this field location: write access, read access or none. Write access equals a fractional permission of 100%, also called full access. Read access is gained by having either of:

- A fractional permission α: $0\% < \alpha \leq 100\%$
- An abstract read permission π: $\pi > 0$ [HLMS13]
- A positive number of the indivisible epsilon permission. One epsilon is the smallest possible permission amount that qualifies for read access.

When exhaling an abstract read permission, it is assumed that there exists some positive fraction which is smaller than what is currently held, but an exact fraction is not picked. The only requirement is that the fractional or abstract permission held is bigger than 0. Holding nothing but epsilons is not valid, since we can't split epsilons and therefore can't branch off abstract read permissions from them.

3.2.2 Permission at Runtime

At runtime, we need a representation for permissions, so that we can emulate inhaling and exhaling. Even the abstract read permissions must be represented by a concrete value. Otherwise, when joining, we wouldn't know whether an inhaled abstract read amount equals what was exhaled earlier on, since a new thread may himself fork and join other threads. If we can't be sure if we've inhaled everything that was exhaled, then we also can't tell if we have write access again, or if there still exists another thread that has an abstract read permission to the

same field.
We also want to avoid picking an amount that is too high for the rest of the execution, or too small for the forked off thread, even if we face a setting in which the permission can get arbitrarily small, e.g. through nested or recursive exhaling.
Picking a concrete fraction for the abstract read permission would be too restrictive. A simple example of this problem is shown in Listing 1: At line 6, an abstract read permission is exhaled, but the current thread will still need a fraction of 99% for the call at line 7. Therefore, the exhaled permission at line 6 must not exceed 1%, but it must still be big enough to allow `foo` to fork off further threads.
So, we want to ensure that for all occurring abstract read permissions π_i to the same field, and for any positive fractional permission α and any epsilon permission ϵ, it holds that:

$$\alpha \gg \sum_i (\pi_i) \gg \epsilon$$

where $\gg$ means "much greater than". In order to achieve this, the permission to a field is represented by a triplet:

$$(\text{Fraction } f_1, \text{ Fraction } f_2, \text{ Integer } e)$$

Fraction is the name of the class used to represent rational numbers. The fraction f_1 represents a concrete fractional permission in Chalice (e.g. 50%). The fraction f_2 represents abstract read permission. By making it a fraction, it is ensured that there always exists an even smaller abstract read permission for exhaling. The integer e represents the number of epsilons. This representation also makes inspecting the permissions at runtime easier, because abstract and concrete amounts can be distinguished clearly.
In theory, each element of the triplet is completely independent of the others. Practically, this is not entirely true: For the multiplication of two permissions, some unit u needs to be picked for f_2. This unit is squared when two abstract read permissions are multiplied. The default value for u is $1/1000$. On the other hand, the unit of a concrete fraction f_1 is always equal to 1. Without this unit, the multiplication of two abstract fractions would yield the same result as one abstract and one concrete fraction:

$$(f_a,0,0)\cdot(0,f_b,0) = (0,f_a \cdot f_b,0) \neq (0,f_a,0)\cdot(0,f_b,0) = (0,f_a \cdot f_b \cdot u,0)$$

So now, the following equation holds:

$$(0,f_a/u,0)\cdot(0,f_b,0) = (f_a,0,0)\cdot(0,f_b,0)$$

See 4.3.2 for details about the operators on permissions. A further restriction is that due to machine precision, a Fraction cannot be split indefinitely often, since it cannot be smaller than $1/\text{maxInt}$. However, we assume that this amount is never reached.

Listing 1: Example of a read exhale in Chalice

```
method m(o : SomeObject)
  requires acc(o.f,100);
  ensures acc(o.f,100);
{
  var res : int;
  fork tk := foo(o,10);
  call res := bar(o);
  //some computation
  join res := tk;
  o.f := res; //need write access here
}

method foo(obj : SomeObject, t : int) returns (res : int)
  requires acc(obj.f, rd)
  ensures acc(obj.f, rd)
{
  var i : int;
  var s: seq<token<SomeObject.foo>>
  i := 0;
  while(i<t)
  {
    fork s[i] := foo(obj,0);
    i := i + 1;
  }
     //some computation
  // join all tokens, then return
}

method bar(obj : SomeObject) returns(res : int)
  requires acc(obj.f, 99)
  ensures acc(obj.f, 99)
{...}
```

3.3 Book-Keeping

We need to keep track of the permission that is held by a thread, in order to be able to check whether it has the necessary access level for processing the statements, and to compute a valid abstract amount for exhales. This task is carried out by the static class `Tracer`, which is located in the `Chalice.cs` file (see Section 4.1). The book-keeping is done via its thread static field `PermissionTracer`:

Listing 2: Map and PermissionTracer definition

```
using Map = //Alias
  System.Collections.Generic.Dictionary<System.Tuple<object, string>, Permission>;
...
[Pure] static public class Tracer
{
    [ThreadStatic] public static Map PermissionTracer;
    ...
}
```

The alias `Map` is used for convenience, as this type has many occurrences. A `Dictionary` is a hash map: It represents a collection of keys and values, and it provides a mapping from a specific key to it's associated value. In this case, the keys are (object,string)-tuples. The string represents the field identifier; the object is the object whose field is accessed. The values in the dictionary are the permissions associated with the field locations that are uniquely defined by the key. The `ThreadStatic` attribute ensures the desired thread modular behavior:
Each thread has it's own `PermissionTracer` dictionary, and cannot directly access the dictionary of another thread. However, this means that when a new thread is forked off, it has to initialize this field and add the correct permissions to the newly generated dictionary. See chapter 4.2 for a detailed description of the in- and exhaling process. It is important to note that this initialization has to be done before the precondition of the method is checked, since the precondition might reason about permissions. The Tracer class also provides the functionality for altering dictionary entries more conveniently, checking access requirements in-line, and `Map`-arithmetic (described in chapter 4.3.3).

3.4 Monitors

In Chalice, locking and unlocking is fairly straightforward:

Listing 3: Locking in Chalice

```
method foo()
{
  var v1 := new SomeClass;
  share v1; //makes v1 lockable
  //noncritical code
  acquire(v1);
  //critical section
  release(v1)
}
```

A read or write lock can be obtained for every shared object. The object reference is all that is needed to try to acquire a lock, so there is no extra monitor class required. The monitor invariant is defined in the class description of the object that is locked on (e.g. in `SomeClass`'s definition in Listing 3). The invariant is inhaled when acquiring a lock, and exhaled when releasing it. `share` and `unshare` can be regarded as special cases of `release` and `acquire` respectively.

In `C#` without runtime checking, this simplicity had been preserved by using the class `System.Threading.Monitor.Enter(obj) or Exit(obj)`.
In this approach, share and unshare statements were ignored. Read acquire and read release were also not supported.

For the runtime checking, we want to be able to mimic the verifier's behavior more closely, therefore we decided to use the class `System.Threading.ReaderWriterLock` instead. This gives us the possibility to support `share` and `unshare` statements, read locks (`RdAcquire` and `RdRelease` in Chalice), and theoretically up- and downgrading of a lock. However, only `Downgrade` is available in Chalice; `Upgrade` is not.
The drawback of this approach is that we need a central class that unequivocally maps objects that can be locked on to their respective `ReaderWriterLock` object. This functionality is provided by the static `MonitorHandler` class. Similarily to the `Tracer` from the Listing 2, this class uses a `Dictionary` for book-keeping:

Listing 4: definition of monitorPerms

```
public static class MonitorHandler
{
  public static System.Collections.concurrent.ConcurrentDictionary
        <object,System.Threading.ReaderWriterLock> monitorPerms;
  ...
}
```

The usage of a concurrent dictionary instead of a normal one ensures that `monitorPerms` can be accessed by different threads without leading to race conditions or related errors. This wasn't necessary for the `Tracer`'s dictionary, since it is `ThreadStatic`.
The key is the object that is associated with the lock (`v1` in Listing 3). The value is the corresponding `ReaderWriter` object. The dictionary simultaneously keeps track of the availability of the lock (shared or not) and, if shared, provides the required object-to-lock mapping.
Other than the dictionary, the `MonitorHandler` class provides methods to facilitate access to it. There is one method for each lock-relevant Chalice statement, except the `Lock` statement. This reduces the code needed at the caller site to one call (plus the in- and exhaling checks). In other words, the complexity that is newly generated by this approach is handled in the example-independent `Chalice.cs` file. Details about the `MonitorHandler` methods can be found in Section 4.3.4.
The Chalice construct `Lock(obj, body, isRead)` acquires a lock for executing a block of statements. It is translated as consecutively acquiring the lock of `obj`, then executing the statements in `body`, and finally releasing the lock again. The boolean `isRead` determines whether a read lock or a write lock is to be acquired/released. Since we use the same constructs as for acquiring and releasing locks, no extra inhaling and exhaling mechanisms or `MonitorHandler` methods are needed for this construct.

We do not deal with the deadlock avoidance constructs in the scope of this project. These are part of the possible future work (Section 6.2).

3.5 Inhale and Exhale

3.5.1 Calls

A Chalice call looks like this:

Chalice call

```
call r1,r2, ... := someObj.foo(p1,p2,...);
```

`r1, r2, ...` are the return values of `foo`, and `p1, p2, ...` are the parameters. In Chalice, the verifier checks and exhales `foo`'s precondition, and inhales its postcondition. Note that Chalice does not have to enter `foo`'s body or check its postcondition here. At runtime, we do not need to inhale or exhale anything, since we stay in the same thread. The precondition is checked, then the execution of the body starts. On return, the postcondition is checked.

3.5.2 Forks and Joins

A fork in Chalice looks like this:

Chalice fork

```
fork tk := someObj.foo(p1,p2,...);
```

Here, `tk` is the token that is associated with the fork. The current thread has to check that the precondition can be satisfied, then it exhales it. The new thread inhales the method's precondition first, then starts its execution. In Chalice, the verifier would just proceed with the next statement, without actually forking anything off, since it verifies modularly and statically. A join looks like this:

Chalice join

```
join r1, r2, ... := tk;
```

The postcondition of the forked method is inhaled at this point in Chalice. At runtime, the whole `PermissionTracer` content is inhaled, not just the postcondition. That means that at runtime, even the permissions that aren't mentioned in the postcondition are inhaled again, so no permission is leaked. Thus, we can safely ignore the postcondition for inhaling at runtime, since all we need to know is stored in the Tracer (the postcondition is still checked though). The implementation of inhaling and exhaling is explained in Section 4.2.2.

3.5.3 The Main Method

Chalice does not need main methods, since it works modularly on methods without having the need for a definite starting point of the program. This is different at runtime, as we need a entry point to start the execution. C# requires a `Main` method (capitalized) without arguments for that purpose. The execution will start with the default constructor of the class that defines the main method, and then continues by calling `Main()`. We initialize the `MonitorHandler` and the `permissionTracer` in this constructor, so that they can be used thenceforwards. Since the constructor might be used again later on in the program, we have to check that they aren't already initialized first.
If Chalice defines a lowercase `main` method without arguments, the corresponding inhaling method (see Section 4.2.2) will be the first one to be called by the actual C# `Main` method. Therefore, the `main` method's precondition will be inhaled at the start of the program execution, so the start of the C# program is in that respect similar to the start of a new thread. If Chalice does not define a `main` method, then no `Main` is generated and the output program will not be executable without first adding it manually. That's why the `main` methods have been added to the examples for testing (Section 5).

3.5.4 Monitor Invariants

Locks in Chalice can be obtained for any object. This object can define a monitor invariant, or just let it be `true`.

Chalice acquire

```
acquire obj;
```

When a write lock is acquired, the permissions in the invariant are inhaled. When a lock is released or shared, the permissions are exhaled. When acquiring or releasing a read lock, the same happens, but the inhaled and exhaled permissions are abstract.
At runtime, these operations are performed in two steps. For shares and releases, the invariant is exhaled first, then the lock operations takes place. For acquires, the lock operations are performed before inhaling the invariant. Here is an example of a write lock acquire with runtime checks:

C# write lock acquire

```
MonitorHandler.acquireWriteLock(obj);
Tracer.PermissionTracer = Tracer.mapAdd(Tracer.PermissionTracer, obj.collectInvPerm());
```

The lock operations are performed by the `MonitorHandler`, they are explained in detail in Section 4.3.4. The `collectInvPerm` method is explained in Section 4.2.4.

4 Implementation

4.1 Structure

4.1.1 Code Generation without Runtime Checks

In this subsection, we describe the code generation (without runtime checking) that existed at the start of this project. The basic code generation uses two files:
`Chalice.cs` and `ChaliceToCSharp.scala`.
The C# source file `Chalice.cs` is used for auxiliary classes whose definitions are independent of the Chalice code. It contains three class definitions: `ImmutableList<E>`, `ChannelBuffer<E>`, and `ChalicePrint`. As the name suggests, `ImmutableList<E>` provides basic functionalities for a list that cannot be altered after creation. The generic type parameter `E` dictates the type of the list's elements, which also must implement the interface `IEnumerable<E>`. `ChannelBuffer<E>` consists of the private field `Queue<E>`, and the thread-safe methods `public void Add(E e)` and `public E Remove()`. `ChalicePrint` just outputs integers or booleans to the console.
`ChaliceToCSharp.chalice` defines the `ChaliceToCSharp` class. Its top level method `convertProgram` takes a list of Chalice classes and channels stored as ASTs, and converts it to C# code and stores it in the file `out.cs`.
`Chalice.scala` is the file that implements the Chalice console command, which is mostly used to verify Chalice source files. For example, the command

```
chalice dining-philosophers.chalice
```

will start the verifying process for the `dining-philosophers.chalice` source file. To activate C# code generation, the parameter `gen` must be added:

```
chalice /gen dining-philosophers.chalice
```

With this command, the verifier will perform its task as before, and as soon as it has successfully finished, the code generation will be initiated:

Scala code for C# code generation

```
val converter = new ChaliceToCSharp();
writeFile("out.cs", converter.convertProgram(program));
```

The parameter `program` is the output of the parser. However, if the verification fails, then the code generation does not take place.

4.1.2 Code Generation with Runtime Checks

It was decided that it should still be possible to generate the basic program without runtime checking as before, so the changes made should not affect the previous code generation.
A new file `ChaliceToCSharpRTC.scala` defines the `ChaliceToCSharpRTC` class (RTC stands for runtime checking). It works similarly to the `ChaliceToCSharp` class, but the grammar rules have changed so as to include the checking mechanisms. The `ChaliceToCSharpRTC` class inherits from `ChaliceToCSharp`, but only few definitions could actually be reused.
The runtime checking uses the same `Chalice.cs` file as the basic code generation. The three classes from before remain unaltered, but four new classes have been added: `Fraction`, `Permission`, `Tracer`, `MonitorHandler`. The purpose of these classes is explained in Section 3, implementation details can be found in 4.3. This change does not affect the behavior of the original code generation, since it only uses the three classes from before.
`Chalice.scala` also needs to be changed. Firstly, the command line parameter `genRTC` is added, so that the new code generator can be enabled from the command line:

```
chalice /genRTC dining-philosophers.chalice
```

Before starting the verification, it is checked whether `genRTC` is enabled. If so, the code generation is started before verifying the program. This means that the code is generated even if the verification fails, which makes sense because the failing cases are more interesting to examine at runtime. The code generation with runtime checks is started by this:

Scala code for C# code generation with runtime checks

```
val converterRTC = new ChaliceToCSharpRTC();
writeFile("out.cs", converterRTC.convertProgram(program));
```

Note that the original parameter `gen` is still checked after the verification process, so if both `/gen` and `/genRTC` are used, the `out.cs` file is first written with runtime checks, and will be overwritten after the verification exactly if the verification is successful.

4.2 Inhale and Exhale

4.2.1 Forks and Joins without Runtime Checking

In order to support forking and joining at runtime, two extra members need to be defined: a delegate `foo_delegate(paramType p1, paramType p2, ...)`, and a class `Token_foo`. Because lookahead would be quite complicated, those two definitions need to be added for all methods defined in Chalice, even if they never occur in forks.
The delegate is used to encapsulate the method, which is necessary for the asynchronous call in C# . Its parameters must match the parameters of the encapsulated method.
The Token has a field for every return value of the method `foo`. Additionally, it has a field `del` for storing an instance of the method's delegate, and a `System.IAsyncResult` field `async`, which represents the status of the asynchronous call. So the number of fields is the number of return values plus two.

For forking without runtime checking, a delegate and a token are instantiated, the delegate is stored in the token and then the actual fork takes place:

C# fork

```
someObj.Token_foo tk = new someObj.Token_foo(); //uses a locally unique identifier
tk.del = new foo_delegate(someObj.foo);
tk.async = tk.del.BeginInvoke(p1, p2, ..., out tk.r1, out tk.r2, ..., null, null);
```

The delegate constructor takes the method handle as an argument. `BeginInvoke` takes the following arguments: One normal argument per Chalice method argument (here: `p1, p2, ...`), one out-argument per return value (here: `r1, r2, ...`), and two extra null arguments. The null arguments are for callbacks; they are never used in our generated code.
Joining is much simpler:

C# join

```
tk.del.EndInvoke(out r1, out r2, ..., tk.async);
```

`EndInvoke` has one out-argument per return value, plus the `async` field that was assigned by the fork.

4.2.2 Forks and Joins with Runtime Checking

For forking with runtime checking, we have to check and exhale the precondition, and we need to let the newly generated thread inhale its precondition with the correct abstract read fractions. Two extra methods need to be defined for each actual Chalice method in order to be able to do this: `initializeThread_foo` and `collectPrecond_foo`
`collectPrecond_foo` collects all the permission restrictions that occur in the precondition, stores them in a object of type `Map` and then returns it. This is needed because the parent thread needs to know what he has to exhale.
The Tracer needs to be initialized before entering the original method body, otherwise the precondition would not be satisfied. This initialization is performed by `initializeThread_foo`, which wraps the forked method: First it inhales the precondition by adding the permissions to the Tracer, then it calls the original method. In order to support the abstract read permissions, it can take a Map as an argument: `Map persIn`. The parent thread can store the exhaled permissions in there, so that the current thread knows exactly what amounts he has inhaled. This is necessary since the read amount can vary from fork to fork, but the sum of the permissions in circulation should still be 100% for each field location. If a thread does not provide a Map (i.e., `persIn` is null), then the precondition is inhaled with default values of $1/10$ for the abstract read amounts.

Furthermore, we need to be able to inhale the correct permissions when joining a thread again. This is made difficult by the fact that the new thread can fork off further threads, and thus end up with permission amounts that cannot be known from the pre- and postcondition alone. Just as the parent thread needed to pass the exhaled amounts to the new thread, the new thread needs to pass his Tracer's Map to the thread that's joining it, so that the joining thread can inhale it. For this, each Chalice method and its `InitializeThread` method get a new out parameter: `out Map pers`. At the end of each method body, the current `PermissionTracer` is assigned to this parameter. This is necessary because we do not know in advance which methods are forked, so we do not know whether a return is actually a join or not.

The delegate needs to be changed in order to match the signature of the `InitializeThread_foo` method instead, because we want this to be the first method that is executed by the new thread. This means that the parameters `out Map pers` and `Map persIn` need to be included:

C# with runtime checks: delegate definition

```
public delegate void foo_delegate(type1 p1, type2 p2, ...,
       out resultType1 r1, out resultType2 r2, ..., out Map pers, Map persIn);
```

In the token, the fields `Fraction rdFrac`, `Map pers`, and `bool joinable` are added. The field `rdFrac` is used for the `rd(tk)` expression in Chalice, which represents the abstract read fraction that was picked when forking off the thread associated with the token `tk`. The boolean `joinable` states whether the token can be joined in Chalice. It prevents tokens from being joined twice, which is already ensured by `tk.async` at runtime, but we need it anyway because contracts in Chalice may talk about it.

C# with runtime checks: Token definition

```
public class Token_foo {
  public main foo_delegate del;
  public System.IAsyncResult async;
  public Map pers;
  public Fraction rdFrac;
  public bool joinable;
  public resultType1 r1;
  public resultType2 r2;
  ...
  }
```

Finally, a fork and join now look like this:

C# with runtime checks: fork

```
Token_foo tk = new Token_foo(); //locally unique identifier
Tracer.addElement(tk, "joinable",100);
tk.joinable = true;
Map dict; //locally unique identifier
dict = someObj.collectPrecond_foo(params);
tk.rdFrac =Tracer.scale(dict); //scales the abstract read permissions
Tracer.subtract(dict); //exhale
tk.del = new foo_delegate(someObj.initializeThread_foo);
tk.async =
       tk.del.BeginInvoke(p1, p2, ...,
              out r1, out r2, ..., out tk.pers, Tracer.copyMap(dict), null, null);
```

C# with runtime checks: join

```
Contract.Assert(Tracer.TryGetValue((object)tk, "joinable") &&
       Tracer.tempPerm ≥ new Permission(100) && tk.joinable);
tk.joinable = false;
tk.del.EndInvoke(out r1, out r2, ..., out tk.pers, tk.async);
```

Details for the Tracer methods can be found in Section 4.3.3. The copy constructor for Maps is used for passing `dict` to ensure that it cannot be altered after the fork, which would be very unsound.

4.2.3 Predicates

In Chalice, a predicate definition looks like this:

```
predicate pred{acc(f1) && f1!=null}
```

A predicate is defined in a class, the identifiers it mentions belong to the fields of this class. `pred` is the name of the predicate, and `f1` is a member of the same class as `pred`. An occurrence of the predicate in a contract looks like this:

```
requires obj.pred;
```

Where `obj` belongs to the class that defines `pred`. A predicate can be folded and unfolded again. Folded means that the predicate is looked at as one package. For evaluating parts of the predicate, it needs to be unfolded.
We decided to always evaluate the predicates at runtime, without looking at the current folding state. Statements concerning the folding of a predicate are just translated to comments in C#. Another approach would have been to keep track of the current folding state, which is a possible

extension for future work.
For each predicate definition `pred`, two methods are generated in the C# code:

```
public Map collectPredPerm_pred()
public bool checkPredicate_pred()
```

`checkPredicate` is the one used in C# contracts. It simply evaluates the predicate as a boolean expression.
`collectPredPerm` is used when the permissions for a inhale or exhale are collected (e.g. in `collectPrecond_foo`). Like in `collectPrecond`, the permissions are added to a Map which is then returned. The caller can then add the returned Map to his own:

```
dict = Tracer.mapAdd(dict, obj.collectPredPerm_pred());
```

This output is generated when a `MemberAccess(obj,pred)` occurs in an inhaled contract or in another predicate. An `Access(ma, perm)`, in which `ma` is a predicate, is translated to this:

```
dict = Tracer.mapAdd(dict, Tracer.permTimesMap(perm, obj.collectPredPerm_pred()));
```

In this case, `perm` is a permission (e.g. 50%). `dict` is some Map identifier; it could even be the `PermissionTracer`.
The default value of `PredicateEpsilon`, which represents the abstract read fraction that is multiplied with the predicate in a `rd(pred)` expression, is $1/100$.

4.2.4 Monitor Invariants

Again, two methods are used:

```
public Map collectInvPerm()
public bool checkInvariant()
```

Again, `checkInvariant` is the one used in C# contracts. `collectInvPerm` is used when a monitor invariant is inhaled or exhaled. Additionally, `checkRdInvariant` is defined, which works like `checkInvariant` but only requires read access to all field locations mentioned in the invariant. No permission multiplication is performed for checking the read invariant, the only requirement for the check to evaluate to true is that the current thread holds some permission amount $> 0\%$ for the mentioned field locations, independent of the actual amount that is stated. Note that this doesn't allow releasing less than what was previously inhaled, since the release of the read lock relies on the `collectInvPerm`, not on the `checkRdInvariant`. This approach was taken due to the general weakness of in-line checks, as explained in Section 4.4.3.
There can only be one monitor invariant per class. All invariant-members of a Chalice class will be concatenated and translated into those two C# methods. For the read acquire and read release, the result of `collectInvPerm` is just multiplied with the `MonitorEpsilon` permission to obtain an abstract read fraction instead of the full amount. The default value of a `MonitorEpsilon` permission is $(0,1/100,0)$.

4.3 Operations

4.3.1 The Fraction Class

A Fraction represents a rational number $q = a/b$ using the two integers `numerator` (a) and `denominator` (b). It implements the `IComparable` interface of C#, which is used to compare objects with one another and is useful for relational operators. However, its function `CompareTo` is only defined for arguments of type integer or Fraction. The constructor of Fraction either takes two integers (a and b), or another Fraction (copy constructor).
Several arithmetic and relational operators are defined statically for (Fraction, Fraction) and

(Fraction, int) pairs. Their implementation follows the mathematical interpretation as rational numbers.
Additionally, the methods `cancel`, `divides` are defined, and `ToString`, `GetHashCode` and `Equals` are overridden.
`void cancel()` simplifies the quotient by dividing both the numerator and the denominator by their greatest common divisor. This method is used by some operators to normalize arguments and results.
`bool divides(int n)` checks whether the argument `n` is divisible by `this`, i.e. it returns true iff $n \cdot {}^{a}/_{b} \in \mathbb{Z}$. This must be true if a permission is mutiplied by some number n of epsilon permission.
`string ToString()` is defined by object, it is overridden in Fraction in order to improve readability when inspecting a Permission when debugging. `GetHashCode` and `Equals` are overridden so that they match the `==` and `!=` operator's definition.

4.3.2 The Permission Class

A Permission represents a Chalice permission as a (Fraction, Fraction, integer) tuple. The reasoning behind this is explained in Section 3.2. It implements the `IComparable` interface with the methods `CompareTo(Permission other)` and `CompareTo(int percent)`. The Permission's constructor can either take two Fractions and an integer (the whole triplet), just an integer denoting the percentage for the first element, or another Permission (copy constructor). There is also a constructor that takes a boolean, which is used to set the boolean field `isRdLiteral`. This field is used as a workaround for checking that $> 0\%$ is held, even if the translation uses a $\geq$ instead of a $>$ operator: The method for $\geq$ checks this field, and returns the result from > 0 instead if it's true. Picking the right operator during translation would require to look into the right hand side of $\geq$, which is complicated. Neither the boolean constructor nor the field are used in any other context.
Also, it overrides the `Equals`, `GetHashCode` and `ToString` methods for the same reasons as in the Fraction class.
Again, the arithmetic and relational operators are defined statically.
`static Fraction getReadFractionUnit()` defines the unit u of the abstract read permission, which is used by the Permission multiplication. The default value is ${}^{1}/_{1000}$.
For the operators, let p_A, p_B be Permissions, so that $p_A = (a_1, a_2, a_3)$ and $p_B = (b_1, b_2, b_3)$ with the types (Fraction, Fraction, integer).

$$
\begin{aligned}
\oplus &: p_A + p_B \coloneqq (a_1 + b_1, a_2 + b_2, a_3 + b_3) \\
\ominus &: p_A - p_B \coloneqq (a_1 - b_1, a_2 - b_2, a_3 - b_3) \\
\otimes &: p_A \times p_B \coloneqq \big(a_1 \times b_1, (a_1 \times b_2) + (b_1 \times a_2) + (a_2 \times b_2 \times u), b_1 \times a_3 + a_1 \times b_3\big) \\
\oslash &: \text{undefined} \\
== &: p_A = p_B \coloneqq (a_1 = b_1) \wedge (a_2 = b_2) \wedge (a_3 = b_3) \\
\neq &: p_A \neq p_B \coloneqq !(p_A = p_B) \\
\geq &: p_A \geq p_B \coloneqq (a_1 > b_1) \vee \Big((a_1 = b_1) \wedge (a_2 > b_2)\Big) \vee \Big((a_1 = b_1) \wedge (a_2 = b_2) \wedge (a_3 \geq b_3)\Big) \\
\leq &: p_A \geq p_B \coloneqq (a_1 < b_1) \vee \Big((a_1 = b_1) \wedge (a_2 < b_2)\Big) \vee \Big((a_1 = b_1) \wedge (a_2 = b_2) \wedge (a_3 \leq b_3)\Big) \\
> &: p_A > p_B \coloneqq !(p_A \leq p_B) \\
< &: p_A < p_B \coloneqq !(p_A \geq p_B)
\end{aligned}
$$

For the multiplication ($\otimes$) to be defined, $(b_1 \times a_3 + a_1 \times b_3)$ must be an integer. The following two requirements guarantee this:
$a_3 \neq 0 \implies \big(b_2 = 0 \wedge b_3 = 0 \wedge b_1.\text{divides}(a_3)\big)$
$b_3 \neq 0 \implies \big(a_2 = 0 \wedge a_3 = 0 \wedge a_1.\text{divides}(b_3)\big)$
An abstract read permission times an epsilon is never allowed, hence the $b_2 = 0$ and $a_2 = 0$ conjuncts above. The result of such a multiplication couldn't be an integer because we assume that abstract read fractions are always much smaller than concrete fractions, and therefore the result would be smaller than one.

4.3.3 The Tracer Class

The Tracer has the following fields:

```
public [ThreadStatic] static Map PermissionTracer;
public [ThreadStatic] static Permission tempPerm;
```

The PermissionTracer is explained in Section 3.3. The field `tempPerm` is used as the out-parameter for `TryGetValue`, which is explained below. `tempPerm` prevents methods from having to declare a suitable Permission variable beforehand, which would make it impossible to use `TryGetValue` in preconditions. The `ThreadStatic` attribute prevents other threads from interfering.
There are three methods for accessing the PermissionTracer:

```
public bool TryGetValue(object obj, string str, out Permission value)
public void addElement(object obj, string str, Permission value)
public void subtract(Map dict)
```

`TryGetValue` first creates a (object, string) tuple from the first two arguments, which represent the associated field location. The object is the target of the field dereference, and the string is the field name (which is unique in the scope of the object).
Then, `TryGetValue` uses that tuple to query the PermissionTracer for the Permission associated with it, which is then stored in the out parameter. This method is overloaded, so that the out parameter can be left out. In this case, the field `tempPerm` is used as the out parameter for the query.
The method returns true if the query was successful, i.e., if an entry with this tuple as a key exists. If no such entry exists, null is assigned to the out parameter and false is returned, which usually leads to an assertion failure.
The purpose of `TryGetValue` is to facilitate in-line checks, which otherwise would need to create the tuple first themselves, which again would make preconditions infeasible.
The Chalice statement `requires acc(obj.f, 50);` can now be translated to:

```
Contract.Requires(Tracer.TryGetValue((object)obj, f)
       && Tracer.tempPerm ≥ new Permission(50));
```

`addElement` creates a (object, string) tuple from the first two arguments, and then adds the Permission `value` to the PermissionTracer entry with the tuple as a key. If no such entry exists yet, it is created. If the entry ends up with a Permission of more than 100%, an assertion failure occurs.
`subtract` takes a Map and subtracts it from the Tracer, which is used for exhaling. Internally, it iterates over the Map's entries and subtracts them from the corresponding entries in the Tracer. If the Map's value is bigger than the Tracer's, or if the Tracer doesn't have the entry at all, then an assertion failure occurs. This is the expected behavior, since it means that the current thread does not own the necessary Permission for the operation it's about to perform.

In addition, several auxiliary methods for Map-related operations are defined in the Tracer.
`Map mapAdd(Map target, object obj, string str, Permission per)` is similar to `addElement`, but this time the Permission is added to the `target` instead of the Tracer. After adding the Permission, the updated Map is returned. The `target` Map is altered by this method.
`Map mapAdd(Map a, Map b)` implements the addition for two Maps. The result is a new Map; the arguments remain unaltered. Again, no entry may exceed 100%.
`Map permTimesMap(Permission perm, Map dict)` implements the multiplication of a Permission with a Map. Each of dict's values is multiplied by `perm` and then stored in the result Map. The same rules as for the Permission multiplication operator (defined in Section 4.3.2) apply. Again, the argument Map remains unaltered.
`Map copyMap(Map other)` creates a copy of the argument and returns the copy. This method can be used like a copy constructor.

The last three methods are used to scale the abstract read fractions of a given Map, so that they are smaller than what the current thread holds:
`Permission scaleOne(Permission old, Permission refr)` scales the `old`'s abstract read Fraction down until it is smaller than the one in `refr`, then it returns the result. The other two elements of the Permission triplet are ignored. This method is used by the `scale` method (defined below).
`Permission scaleTo(Permission old, int goalDenom)` scales the abstract read Fraction of `old` so that its denominator is equal to `goalDenom`. The resulting Permission is returned. This method too is used by `scale`.
`Fraction scale(Map dict)` first decides on a denominator `maxDenom` for the abstract read Permissions. This is achieved by iterating over all entries and calling `scaleOne(entry.Value, p)`, where `p` is the value of the same entry in the Tracer. If the Tracer is missing an entry that is in `dict`, an assertion failure occurs. `maxDenom` is then chosen to be the maximum abstract read Fraction's denominator of all the results of `scaleOne`.
Then, the entries of `dict` are scaled so that their abstract read Fraction's denominator is equal to `maxDenom`. For this, `scaleTo(entry.Value, maxDenom)` is used. When the scaling of `dict` is complete, the Fraction $1/\text{maxDenom}$ is returned. Note that this method alters the argument `dict`.

4.3.4 The MonitorHandler Class

The MonitorHandler class is used for lock-related statements. In particular, it maintains a mapping between source program objects and monitor objects. Its only field is `monitorPerms`, which is explained in Section 3.4. All of its methods are static, and every method corresponds to one Chalice statement or expression, except the `Lock` expression (as explained in Section 3.4). Note that the inhaling, exhaling and checking of the monitor invariant is not performed by these methods. The caller must do that manually (see Sections 3.5.4 and 4.2.4 for details).

`void shareElement(object obj)` handles the `share(obj)` statement from Chalice. It requires that the object is not shared already, and the thread must have write access to the `obj.mu` field. The mu field is used by Chalice for deadlock avoidance; it is implicitly defined for every Chalice class. If the requirements are met, then a new entry is added to `monitorPerms`. The value of the new entry is a new `ReaderWriterLock`, the key is the object `obj`.
`void unshareElement(object obj)` handles the `unshare(obj)` statement from Chalice. It removes the entry with key `obj` from `monitorPerms`. It requires write access to the mu field and that the write lock associated with `obj` is held by the current thread.

`void acquireReadLock(object obj)` corresponds to `RdAcquire(obj)` in Chalice. It requires read access to mu, and `obj` must be shared, i.e., an entry with key `obj` must exist in `monitorPerms`. Internally, it tries to acquire a read lock that is associated with `obj`. The current thread is blocked until the lock is acquired.
`void acquireWriteLock(object obj)` corresponds to `Acquire(obj)` in Chalice. It requires read access to mu, and `obj` must be shared. It works analogously to `acquireReadLock`, but is exclusive, i.e., no other write or read lock of the same object may be held or acquired while the write lock is held by some thread.
`void releaseReadLock(object obj)` corresponts to `RdRelease(obj)`. It releases a read lock. It requires that the object is shared and that the current thread actually holds a read lock of it.
`void releaseWriteLock(object obj)` corresponds to `Release(obj)` and requires that `obj` is shared and the current thread actually holds the write lock of it.
`bool isReadLockHeld(object obj)` corresponds to the `RdHolds(obj)` expression and requires that `obj` is shared. It returns true if the current thread holds a read lock of obj, and false otherwise.
`bool isWriteLockHeld(object obj)` corresponds to `Holds(obj)` and works analogously.

4.4 Checks

4.4.1 Read and Write Checks

Whenever a field is assigned to, we have to make sure that the current thread has write access to it. Consider a Chalice statement of the form `obj.f := someObj;`
When translating that, it is first checked whether the Permission in the Tracer that belongs to the (obj,f) tuple equals 100%, and only afterwards does the actual assignment takes place. The C# output for this statement looks like this:

```
Contract.Assert(Tracer.TryGetValue((object)obj, f)
       && (Tracer.tempPerm == new Permission(100)));
obj.f = someObj;
```

Apart from that, similar write checks are performed for the `joinable` field in joins (See 4.2.2) and the mu field for some lock operations (4.3.4).

Read checks happen far more often, as they are necessary for every field that is dereferenced or otherwise read from. Because of the pervasiveness of reads, we have to scan each expression of every statement for field read occurrences before translating the statement itself. This scan is done during the translation process, not at runtime. For each field read `obj.f` that is found, an assertion is outputted:

```
Contract.Assert(Tracer.TryGetValue((object)obj, f)
       && (Tracer.tempPerm > new Permission(0)));
```

Additionally, the scanner looks out for implications and `&&` expressions for short-circuiting, because otherwise a check like `(f != null && acc(f.g))` would end in a failure. Pre- and postconditions do not generate read checks, because the support for short-circuiting would be very cumbersome, especially since no other non-contract statements (like an if-then-else block) are allowed in C# before the contract block has ended.
The actual translation of the statement happens after all the read access checks have been generated.

4.4.2 Loop Invariants

Loop invariants are translated to a set of assertions. All these assertions are first checked before entering the loop, and then again at the end of the loop body, so that they are checked after each iteration.
Note that the read checks generated from the invariants and the guard belong to this set of assertions too, so that they are always checked together with the invariant.

4.4.3 Pre- and Postconditions

Pre- and postconditions are translated for all methods and functions. However, permission expressions are not accumulated like they are for inhaling and exhaling, so the checks done here may actually be too weak in that respect. Accumulating the permissions to be checked would require additional collector methods. This weakness of in-line checks is also the reason why read checks for monitor invariants only check for positive amounts, and not some exact value defined by permission multiplication like for the inhaling and exhaling (see Section 4.2.4).
Pre- and postconditions appear outside of the method body in Chalice. For C#, they are moved to the very start of the body:

Contract location in C#

```
public void bar(){
Contract.Requires(true);
Contract.Ensures(true);
//computation
}
```

Pre- and postconditions need to be translated separately, because the grammar for postconditions differs from the one preconditions and assertions use:
Because the postcondition occurs at the start of the method body, the compiler might think that it talks about unassigned out parameters and thus mark it as an error.
The `Contract.ValueAtReturn` method solves this problem by wrapping the out parameters.
Here is an example showing the translation:

Example Chalice method

```
method foo(i : int) returns (res : int)
requires (i > 0)
ensures (res > i)
{//computation
}
```

Corresponding C# output

```
public void foo(int i, out int res){
Contract.Requires(i > 0);
Contract.Ensures(Contract.ValueAtReturn(out res) > i);
//computation
}
```

4.5 Other Design Choices

Negative permission amounts are usually unsound, but we decided not to forbid them in general, which could easily be done in the Permission class. However, we do check for negative values in the Tracer methods `addElement`, `mapAdd` and `subtract`. This should be enough to prevent

unsound Map modifications, but it still allows to have negative intermediate values in contracts. Here is an example of a precondition that thus is valid at runtime:
`requires(acc(o.f, -rd) && acc(o.f, rd) && acc(o.f,100))`

We decided not to support inhaling and exhaling for channels yet, but we defined the mu-field and the `ChannelEpsilon` so that contracts can talk about it. The abstract `ChannelEpsilon` fraction is by default $1/10$.

The `Eval` statement is not supported, but it doesn't lead to an exception while translating; only a warning is displayed.
An occurrence of the Eval statement is not that tragic; it should still be possible to inspect the rest of the program at runtime. The only information missing is the outcome of Eval, which only occurs in contracts. In order to be able to check the rest of the contract, the converter will just fill in a dummy value. However, the Eval statement could actually appear nested in other expressions that expect a different value, and thus lead to runtime errors. The warning is there to remind the programmer that he might have to change the dummy value, so that no error originates from Eval.

In general, the translation doesn't stop when an unsupported construct appears, it just displays a warning so that the programmer can fix the problem in the output code manually. Since such unsupported constructs appear mostly in contracts, the usual fix will be to just remove the problematic part of the statement.

`Downgrade` is also not supported. The ReaderWriterLock used for monitors would be capable of handling up- and downgrading, but only allows a reader lock being upgraded to a writer lock and then being downgraded again. It does not allow downgrading without upgrading, and since Chalice doesn't support upgrading, this requirement could never be met.
To release a write lock and then try to acquire a read lock might not have the expected behavior, namely that no other thread can acquire a write lock during the downgrade. Therefore, we decided to not support downgrade at all.

An `Assume` statement translates to a assertion at runtime, since a wrong assume would make the rest of the verification unsound and therefore shouldn't occur. A comment is added to point out that this assert actually stems from an assume in Chalice.
The code generation without runtime checks translated `Assume(false)` statements to an `Assert(false)`, because reaching this statement means that a wrong turn has been taken. Other assumes were ignored. So, nothing has changed for the `Assume(false)` case in the new code generation.

5 Testing

For testing the code generation, the examples in `tests\examples` and the custom example `nestedRecursive` were used. Many of those examples were missing an appropriate main method to be a C# starting point, and thus had to be altered (See Section 3.5.3). The table below shows the name of the test, how it was modified and whether the verification was successful. The modification concern the main methods in the `.chalice` file only.
For the verification, the newly added main methods didn't always verify, which isn't important for testing the code generation. If the verification only failed because of newly generated code, this is indicated with an asterisk (Succeeds*).
The Runtime column says whether the program behaved correctly at runtime, which requires absence of assertion failures and unexpected results.
The last column indicates whether the runtime behavior is what is expected. Note that a program might be expected to run correctly even if it didn't verify, which may be due to the thread-modularity (see Section 3.1) or simply because the problematic cases marked by the verification process didn't occur in the given setting. In other words: just because a contract is unproven, that doesn't mean that it actually has to fail at runtime.
The problems with some of the test cases are explained below.

File	Modification	Verifier	Runtime	Correct
AssociationList	Added main with Init()	Fails	Fails	Yes
AssociationList	Added main without Init()	Succeeds	Succeeds	Yes
AVLTree.iterative	Added main	Succeeds*	Succeeds	Yes
AVLTree.nokeys	Added main	Succeeds*	Succeeds	Yes
BackgroundComputation	Unaltered	Succeeds	Succeeds	Yes
cell	Added calls to main1 - main4	Fails	Succeeds	No
CopyLessMessagePassing	Added main	Succeeds*	Succeeds	Yes
CLMP-with-ack	Added main	Succeeds*	Succeeds	Yes
CLMP-with-ack2	Added main	Succeeds*	Succeeds	Yes
dining-philosophers	Unaltered	Succeeds	Succeeds	Yes
FictionallyDisjointCells	Added main	Succeeds*	Succeeds	Yes
ForkJoin	Calls to main1 and main2	Succeeds	Fails	No
HandOverHand	Main renamed	Succeeds	Succeeds	Yes
iterator	Unaltered	Succeeds	Fails	No
iterator2	Unaltered	Succeeds	Fails	No
linkedList	Added main	Succeeds*	Succeeds	Yes
OwickiGries	Main renamed	Succeeds	Succeeds	Yes
PetersonsAlgorithm	Main renamed	Succeeds	Succeeds	Yes
ProdConsChannel	Main renamed	Succeeds	Fails	No
Sieve	Start renamed	Succeeds	Succeeds	Yes
Solver	Added main	Succeeds	Succeeds	Yes
swap	Added main	Succeeds*	Succeeds	Yes
RockBand	Main rename	Fails	Succeeds	Yes
TreeOfWorker	Added main	Succeeds*	Succeeds	Yes
UnboundedThreads	Added main	Fails	Fails	No
nestedRecursive	Custom	Fails	Succeeds	Yes

`cell.chalice`: The runtime checker can't find an error because the `Free`-statement is not supported.

`ForkJoin.chalice`: Fails at compile time due to the unsupported `Eval` expressions. If they are removed in the output, the runtime checking succeeds.
The examples `iterator`, `iterator2`, `producer-consumer` and `UnboundedThreads` fail at compile time because quantification is not supported.
`ProdConsChannel` fails due to the missing channel support (inhaling and exhaling).

In summary, there are seven unsuccessful test cases, which fail due to unsupported statements or constructs. The remaining eighteen test cases either don't use unsupported constructs, or the unsupported constructs don't affect the execution (like some `Eval` expressions).

6 Results

6.1 Conclusion

In the course of this project we have found an appropriate Permission-model for runtime checking. Using this model, we have extended the previously existing code generation to represent and track permissions at runtime.
Permissions are represented as a (Fraction, Fraction, integer) triple, and are associated with (object, string) tuples denoting target and name of the field location. The tracking of the Permissions at runtime is done thread-modularly, unlike the static verifier which works method-modularly. We support inhaling and exhaling of Permissions for forks, joins, predicates and monitor invariants. Call related in- and exhaling is not needed due to the thread modular tracking strategy.
Predicates are directly evaluated at runtime, we do not keep track of the folded/unfolded status. We also track the `mu` field of channels and monitors, but we do not support inhaling and exhaling for channels, nor do we support the deadlock avoidance mechanisms of the monitors.
Additionally, we added support for reader-writer locks and connected statements.

The implemented constructs allow us to insert read and write checks before executing operationally relevant statements. Also, we can perform extended checks in pre- and postconditions, loop and monitor invariants and in assert statements.
The new code generation has been tested and behaves as expected for the supported constructs. If unsupported constructs are present, it is often possible to make runtime checking work anyway, either through dummy values or by manually removing the problematic statements in the output code.

6.2 Future Work

The next step for improving the runtime checking would probably be to implement the deadlock avoidance mechanisms. This could be done by using a globally accessible tree to keep track of the locking order. However, to ensure the absence of loops, this tree would need to be locked entirely every time a lock is inserted, which could lead to performance issues.
The permission checks for contracts and monitor invariants could be made tighter by collecting all the permissions first, as is done for inhaling and exhaling. This could be achieved by adding a "collectAndCheck" method to the Tracer class. This method would take the result of the correct `collectPrecond` or `collectInvPerm` as an argument, and check that the `permissionTracer` holds at least the same permission amounts. This would be similar to exhaling the Map, only that it isn't subtracted from the Tracer.
Another possibility would be to just use a strict, linear order of all locks by associating them with a integer. However, this approach would be too restrictive, as it is possible to have multiple

locks that are independent in the ordering in Chalice, i.e. without one having a bigger priority than the other.
So, a decision must be made to either prioritize performance or expressiveness.

The support of inhaling and exhaling over channels is a pending, but likely straightforward task.
Then, there are several expressions that are currently not supported: `Eval`, quantification statements, `Free`, `Credit` and `AccessSeq` are probably the most important ones. Especially `Eval` occurs quite often, but supporting it is rather involved since we needed to keep track of different states all the time.
Finally, one could implement support for keeping track of the state of predicates, instead of just evaluating them. But the benefit of this feature is rather small compared to the other possible improvements mentioned above.

References

[HLMS13] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Abstract read permissions: Fractional permissions without the fractions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Lecture Notes in Computer Science. Springer-Verlag, 2013.

[LM09] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *European Symposium on Programming (ESOP)*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer-Verlag, 2009.

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

[LMS10] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. CW Reports CW573, Department of Computer Science, K.U.Leuven, January 2010. Extended version of the ESOP 2010 paper.