Verifying Separation Logic in Chalice

Daniel Jost

Bachelor Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zürich

January 30, 2013

Supervised by:
Dr. Alexander J. Summers
Prof. Dr. Peter Müller

Abstract

In this bachelor thesis we build a tool that translates programs written in VeriFast to Chalice. It is based on the theory from the paper "The Relationship Between Separation Logic and Implicit Dynamic Frames"[7] by Matthew J. Parkinson and Alexander J. Summers. Our main contribution, besides implementing the translation, consists of enhancing the methodology to translate the parameterised predicates of VeriFast to parameterless predicates and heap-functions in Chalice.

Contents

1 Introduction

1.1 Background

1.1.1 Separation Logic and Implicit Dynamic Frames

While standard Hoare logic has been widely accepted as a method for reasoning about programs, it lacks support for two crucial aspects of modern software engineering: modularity and concurrency. Ideally, one would like for modularity to be able to add additional properties to a Hoare triple, also described by the *framing rule*

$$\frac{\{P\}C\{Q\}}{\{P \wedge R\}C\{Q \wedge R\}}$$

This rule allows one to apply a proof that was made in a local setting to be reused in a larger (whole program) setting where additional fields and properties (R) exist. Even in a sequential setting, this rule needs a side-condition that the command C doesn't alter the meaning of R (particularly, does not modify any free variable of R). But, in the presence of a heap, which gives rise to aliasing, such a side-condition is infeasible; hence, proofs in Hoare logic are not modular by default. One common solution is to extend Hoare logic with some form of *access permissions*; a method might only read and modify the fields it has access to. Usually, different levels (e.g., full and read-only) of access permissions are considered, for instance fractional permissions [1] where 1 means full access and every non-zero fraction read access. Access permissions give rise to a framing rule for additional properties relying only on field locations which the command does not have full access to (and thus, cannot modify). In addition, it naturally extends to the case of concurrency if the access permissions on the same field can somehow be split, as for instance with fractional permissions. We will now discuss two different approaches which have been pursued in order to tackle these problems and introduce some sort of access permissions.

Separation Logic Separation logic is an extension to Hoare logic originally developed for languages with manual memory management, where dealing with aliasing plays a crucial role. It is built on the idea of disjoint heap fragments that describe the memory layout. Those disjoint fragments now give rise to a framing rule [8] in separation logic, where the * (conjunction of disjoint fragments) replaces the regular conjunction:

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}$$

In addition, the points-to assertions (written $e.f \mapsto v$) [6], which assert that the field f of the object denoted by the expression e has the value v, can be interpreted as a permission to the corresponding field. Also, separation logic has been extended with fractional permissions (written $e.f \overset{\pi}{\mapsto} v$ to represent the fraction π), which make it flexible in the context of concurrency.

Implicit Dynamic Frames The idea behind implicit dynamic frames [9] is somewhat different. Rather than tackling the framing problem by splitting the heap, one allows method specifications to directly declare which portion of the (total) heap they may modify. That portion is called the *frame* of the method and the declarations are expressed as functions of the heap; hence, the frames are "dynamic" in the sense that they may change when the heap gets modified. The frame specifications are achieved by extending the first-order logic with so called "accessibility predicates" (written $acc(o.f, \pi)$ to represent the fractional permission π on the field f of object o) which precisely specify which locations might be read or written to.

1.1.2 VeriFast

VeriFast is a semi-automated verification tool for C and Java programs that uses separation logic for preconditions and postconditions. It was originally developed by Bart Jacobs, Jan Smans, and Frank Piessens at the Katholieke Universiteit Leuven [4]. It provides a rich set of specification features such as inductive data-types, pure functions over those ADTs, lemma functions to help the verification, and predicates. Especially, VeriFast allows the verification of the following properties:

- Whether a method satisfies its specification, given by its precondition and postcondition
- The absence of data races in concurrent programs
- The absence of illegal memory accesses (and memory leaks) in C. For Java programs, that property is directly part of the language.

1.1.3 Chalice

Chalice is a tool for the verification of concurrent object-oriented programs which is based on the idea of implicit dynamic frames [5]. In addition, Chalice is a minimalistic programming language which is inspired by object-oriented languages but lacks some of their essential features. The name Chalice is used for both the programming language and the tool that verifies those programs. The specifications (such as preconditions

and postconditions) are first class citizens of the language and provide a wide variety of features to support the verification of both sequential and concurrent programs. More specifically, the language supports objects, threads, mutual-exclusion, monitor invariants, and fine-grained locking. Permissions are handled analogously to fractional permissions which allows splitting and joining of permissions. The verifier is based on Boogie, which does a first-order encoding of the program, and checks whether the stated properties are always satisfied.

1.1.4 Relationship between Separation Logic and Implicit Dynamic Frames

In recent work, Matthew Parkinson and Alex Summers showed how the assertions from separation logic can be translated into those of implicit dynamic frames [7]. Hence, they relate the partial heaps of separation logic to the implicit dynamic frames with its explicit accessibility predicates. In order to achieve this, they have developed a *total heap semantics for separation logic*, which they then used to build a logic that serves as a superset of both separation logic and implicit dynamic frames, called *total heaps permission logic*. This semantics is based on the idea of replacing the partial heaps of separation logic with a total heap combined with *permission masks*, which specify the locations in the heap which belong to the original fragment. Then they have shown that the total heap semantics for separation logic correctly preserves the original semantics of separation logic and, finally, that it can be reduced to the implicit dynamic frames subset.

In short, that relationship is based on the idea that the part of the heap we have access to (due to the permission mask) corresponds to a partial heap fragment in the original separation logic. In addition, the points-to construct $e.f \overset{\pi}{\mapsto} v$ can be translated to $acc(e.f, \pi)\,\&\&\, e.f == v$ which is part of the implicit dynamic frames assertion language.

What the paper did not cover was the issue of relating the predicates of separation logic and the version of implicit dynamic frames implemented in Chalice. While the predicates in separation logic and the implicit dynamic frames described by the original work by Jan Smans [9] do have arguments, the predicates in Chalice do not, due to the first-order encoding of Chalice. Hence, a straightforward translation of this feature is not possible.

1.2 Project description

In our project, we now aim for a translation from VeriFast programs to Chalice. Hence, we explore the practical aspects of the relationship between those two concepts and especially how to translate the missing pieces such as predicates and the programming language itself.

1.2.1 Core

The basic goal of this project is to implement a translation from VeriFast code (separation logic) to Chalice code (implicit dynamic frames). In the first phase, a suitable subset of VeriFast to be supported should be chosen. This will involve the analysis of existing VeriFast examples in order to identify commonly used features of the VeriFast assertion-language. In addition, working out by hand how to translate common cases of generally non-trivial language constructs (like parameterized predicates) will be a substantial part of this phase. While aiming for a general translation is out of scope for the core part, there were some pre-existing ideas for how to translate them by hand by replacing the parameters with functions; thus, one expects that when manually looking at the examples the translation will often be obvious.

In a second step, this translation should be implemented. This, first, requires parsing of the VeriFast assertions to an abstract syntax tree. Second, those assertions can then also be viewed as having been written in total heaps permission logic, whose assertions are a superset of separation logic. Third, the mapping of the separation logic part of total heaps permission logic to the Chalice subset can be performed. Finally, Chalice assertions should be generated, which then can be handed to the Chalice tool.

Finally, the implementation should be used to evaluate how well the Chalice tool can handle the corresponding examples. Possible shortcomings in both the translation formalism and the Chalice tool should be identified.

1.2.2 Extensions

Extra work can be performed in re-evaluating the supported subset of VeriFast in order to handle a broader class of examples. This might lead to additional insights into how general the translation is at the moment and how many of the existing VeriFast examples actually rely on features not yet covered by the translation.

At the start of the project, we considered that the formalism might be extended to handle VeriFast's recursive predicate definitions and to figure out how to translate parameterized and recursive predicates to chalice. Those are of particular interest, as they show up in most of the existing examples; recursive predicates are especially used whenever the program is dealing with recursive data structures such as linked-lists. However, the mapping from parameterized predicates in VeriFast to parameterless predicates and heap-dependent functions in Chalice is tricky. During the project we have worked out how to make it practical, and as generally applicable as possible.

Additional language constructs from VeriFast such as fixpoint functions have also been tackled. Closely related in VeriFast are also abstract data types, which currently are not implemented in Chalice and which we did not deal with in our project. In the beginning, we discussed that one might either try to translate them to Chalice as well, or more likely

go ahead and implement them in Chalice, as that would not only eliminate the need for a translation but also help on Chalice in general.

1.2.3 Goals

More explicitly, our project had the following goals:

Core

- Identify a suitable subset of the VeriFast assertions to be handled
- Write a compiler that translates VeriFast programs to Chalice programs
- Trying out by hand how to translate common cases of generally non-trivial language constructs like parameterized predicates
- Evaluate how well Chalice can handle translated separation logic examples

Extensions

- Generalising the by-hand experience for handling some cases of parameterized predicates automatically, including their folding and unfolding.
- Evaluate the practical feasibility of the extended methodology and translation
- Handle fixpoint functions in the translation
- Implement abstract datatypes for Chalice

2 Translation

2.1 Technical overview

We have seen in the introduction that VeriFast and Chalice differ in quite a few ways and, yet, they try to solve similar problems. We have also seen that the translation can theoretically be done for the core of the logic except for predicates. However, the programming languages also differ in many aspects; on one side, there is VeriFast which supports C and Java programs, and on the other side, there is Chalice with its own minimalistic language. For the sake of simplicity, we have chosen to only focus on translating Java programs, since Chalice is also object oriented.

In summary, we have to build a tool which can deal with the translation from Java to Chalice and can handle some common cases of predicates, in order to obtain a practically useful tool. We will now continue to give a brief overview of the technical implementation of our tool and elaborate some of the design decisions we have made.

2.1.1 Programming language and tool-chain

We decided to build our tool in Scala. This was quite a natural choice since Chalice itself as our translation target is also written in Scala and Scala is quite popular in the software verification research area, with good library support for things like parsing. In addition, using Scala will also allow us to reuse the pretty-printer from Chalice and, hence, directly target the AST of Chalice rather than having to deal with emitting correct Chalice code ourself.

2.1.2 Semi or fully automated

One of the design decisions we had to make was whether our tool should be semi or fully automatic. Due to the large number of differences between VeriFast and Chalice, it was pretty clear from the beginning that not every example can be translated automatically, and that in some cases user interaction could help; nevertheless, we decided to go for a fully automatic tool. This decision was based mainly on the additional program complexity that a GUI would require. A GUI that would allow some flexible user interaction during the translation would definitely be non-trivial and require a lot of effort without providing much additional insight. As a consequence, we decided that user interaction

should be made, whenever required, by manual modification of the VeriFast source and then simply restarting the translation. We believe that this gives enough flexibility to the user to make our tool usable without complicating our project. However, this obviously requires that our tool produces meaningful error messages, so that the user knows which part he must modify. On the other hand, understandable error messages should be a core feature of every tool that is supposed to fail in certain cases; therefore, that should be part of our tool anyway.

2.1.3 Overall design

We will now briefly discuss the overall technical design of our translator before diving into the interesting part of the details during the next chapters.

As a first step, the parsing of the VeriFast program has to be performed. Since parsing is well known and has good support in Scala we will not discuss that aspect any further. The parser should be able to parse almost the complete Java 7 language and definitely a superset of what VeriFast can; however, not all the VeriFast annotations are parseable. We decided to leave out some of the VeriFast annotations that we definitely cannot translate, since one might easily extend the parser later. Please have a look at the appendix for additional details.

As a second step, we do resolution and type-checking. Theoretically, one could assume that the original VeriFast code is correct - not necessarily that it verifies in VeriFast but that it is syntactically correct and type checks - however, we will need precise type information and binding for any non-trivial transformation anyway. Type resolution should be reasonably complete compared to VeriFast, except that it will fail when using unsupported built-in features of VeriFast, or external libraries.

The actual translation is then further subdivided into smaller parts. The first and most important part is a reduction in our own AST to a subset that is then directly translatable to Chalice. That reduction is composed of multiple steps, each of them reducing one specific unsupported feature to the supported ones. However, while these steps work on different parts of VeriFast's language, the order matters and they are not totally independent. We have chosen this approach of working with our own AST for as long as possible to gain more flexibility and keep the tool extensible, compared to trying to do a single-pass translation; also, having a separate intermediate representation would not have been worth the effort.

Note especially, that our tool can fail in every single phase of the translation; there is no unique point where we check whether all used features are supported or whether the program cannot be translated. Rather, our translation fails at the first point when an unsupported feature is detected.

It is important to notice that most of the following chapters will be dedicated to that reduction and not the final translation. In addition, this requires our AST to include

all the features of the total separation logic and some Chalice-specific features as well. Therefore, the terminology will be a mixture between the one from VeriFast and Chalice henceforth.

Finally, we do the translation from our AST to the one of Chalice and invoke the pretty-printer of Chalice. This step is mostly trivial, since we rely on the reduction having taken care of all difficult cases; however, the translation might still fail in this step when there is an unsupported feature for which no reduction step exists at all. As a rule of thumb, unsupported features for which we can handle special cases will fail during the reduction and the other ones during the final translation.

2.2 Handling of Parametrized Predicates

Both VeriFast and Chalice support predicates to abstract over assertions. However, VeriFast has a more general support with some key advantages which make a translation to Chalice highly non-trivial:

- VeriFast supports predicates with parameters while Chalice does not
- VeriFast supports static predicates in the global scope, while in Chalice every predicate needs a receiver

On the other hand, Chalice's functions may depend on the heap, whereas VeriFast's functions don't.

Since predicates are a key abstraction in both of the tools, almost no VeriFast example exists that does not use them. Therefore, being able to translate predicates is crucial for any tool that ought to be usable in practice; otherwise, every example would first need to be rewritten by hand before one can apply the translation. In summary, the handling of predicates is at the same time one of the most important and one of the most difficult parts of a translation from VeriFast to Chalice.

In the following section we describe how the advantage of having heap dependent functions is used in our tool to support some of VeriFast's predicates and discuss the practical feasibility of the approach. A more general evaluation is included later in section 3 about the evaluation of our tool.

2.2.1 Introduction

At the beginning of our project, Alexander J. Summers had some ideas on how it might be possible to use heap-functions to replace the predicate parameters. Especially, he had two ideas:

1. Replacing the out-parameters, which are uniquely determined by the heap, with a corresponding getter-function extracted from the predicate body.

   As an example, consider the excerpt from a 'Counter'-class with a predicate and a method, shown in figure 2.1. In the translated Chalice code at the bottom, the predicate parameter has been replaced by a getter-function 'getVal', which represents how the out-parameter is defined inside the VeriFast predicate. Whenever the predicate is then mentioned (as for the contract of the method), the parameter can then be replaced with calls to the getter, as shown in the contract of the 'get'-method.

```
predicate valid(int val) = this.value |-> val;

int get()
    //@ requires valid(?x);
    //@ ensures valid(x) && result == x;
{
    //@ open valid(x);
    return this.value;
    //@ close valid(x);
}
↪
predicate valid() { acc(this.value, 100) }

function getVal(): int
    requires valid;
{ unfolding valid in this.value }

method get() returns (res: int)
    requires valid;
    ensures valid && getVal() = old(getVal()) &&
            res = getVal();
{
    unfold valid;
    res := this.value;
    fold valid;
}
```

Figure 2.1: Example of a 'Counter'-class where the predicate parameter gets replaced by a getter-function

2. Replacing some cases of in-parameters, which are not uniquely determined by the heap, by storing the originally passed-in parameter value in a ghost-field, right before the folding of the predicate.

   An application of this idea is shown in figure 2.2, where the out-parameter 'len' is replaced by a ghost-field in Chalice. Notice, how this idea is only applicable when the predicate instance is unique for a given receiver, which is in this example guaranteed by having full-access on the *next* field.

```
predicate linkedListSegment(int len) requires {
    next |-> _ &*&
    (len > 0 ==> next != null &*&
                 next.linkedListSegment(len-1))
};

// folding that predicate:
//@ close linkedListSegment(x)
↪
ghost var len: int;
predicate linkedListSegment {
    acc(next) &&
    (this.len > 0 ⇒ next.linkedlistSegment)
}

// folding that predicate:
len := x;
fold linkedListSegment;
```

Figure 2.2: A segmented linked-list predicate with an example of how to fold it

The key contribution of our project is the automation of those ideas. Especially, this included figuring out how such getters can be automatically extracted, how to automatically determine whether the use of ghost-state is applicable, and figuring out all the details required to make it work that popped up during the project.

As a motivation as to why the automation is non-trivial, let's look at our linked-list example (figure 2.3) and see what one can learn by reasoning about the predicate by hand. This will guide us to the level of reasoning our tool needs to support.

```
(static) predicate linkedList(LinkedList jList, list<int> aList)
  requires switch(aList) {
    case nil: return jList==null;
    case cons(aX,aNext): return jList!=null
        &*& jList.value |-> ?jX
        &*& jList.next |-> ?jNext
        &*& linkedList(jNext,aNext)
        &*& aX==jX;
  };
```

Figure 2.3: Predicate of our linked-list example (in VeriFast)

We can make the following observations:

- 'jList' is null iff 'aList' is nil. This follows from the observation that when 'aList' is nil then 'jList' is null and conversely in the other branch.
- if 'jList' is null, the predicate does not hold any permissions.
- if 'jList' is non-null, we have full permission to the 'value' and 'next' field of 'jList'.
- 'aList' is uniquely determined by the heap (and the parameter 'jList').

Using all those facts, one can conclude that we can actually make this static predicate a member of the 'LinkedList' class; whenever 'jList' is null the predicate does not convey additional information or permission. How we deal with static predicates will be explained in the dedicated section 2.3.1. Also, the second parameter 'aList' can be reconstructed from 'jList'; hence, there is no need to store it explicitly in the predicate instance, and we can apply the idea of getter-functions.

After looking at the reasoning involved to translate such a predicate, we do not believe that there exists one general algorithm that can deal with all predicates from VeriFast; therefore we tried to keep our tool close to the way a human tackles the predicates and also implement the same tricks a human applies. This entails some degree of semantic reasoning about the problem; any approach that just tries to work on a syntactic level is probably doomed to fail. We, therefore, have taught our tool parts of the semantics of assertions: what does a conditional statement mean and how can we reason on them by cases, how is the conjunction (&&) of two expressions evaluated, and so on.

In the end, we have logically factored the handling of predicates into three tasks:

- Analyze a predicate definition and obtain knowledge about it.
- Use that knowledge to translate it.
- Simplify the resulting predicate definition to minimize redundancy and keep it readable.

In the code, we have split those tasks into two components: a part that actually does the transformations and an analyzer, which handles the first and the third tasks. The transformations are based on the information provided by the analyzer to actually translate the predicates. We will now start by discussing the transformations and postponing the details of the analyzer to a later subsection. The analyzer is somewhat custom tailored to allow the handling of predicates; therefore, when discussing the project in that order, the requirements and motivation for the analyzer will evolve naturally.

2.2.2 Eliminating Parameters

Let's now look into to issue of replacing the parameters in depth. As we have already discussed, the idea is to either replace the parameters by a getter-function or the ghost-state to store the parameter value. Actually, getters are used in both cases: in the first case we will speak of *pure getters*, as shown in figure 2.4 , and in the second case of *getters backed by ghost state*, as shown in figure 2.5 . Hence, the idea is to turn every parameter into a getter and instead of passing a value v to the parameter, the result of the getter must be compared to v.

```
function getLinkedListSegEnd(): LinkedList
    requires acc(linkedListSeg, 100);
{
    unfolding acc(linkedListSeg, 100) in (
        aList[1..] ≠ [] ?
            (this.next ≠ null ?
                this.next.getLinkedListSegEnd() :
                null
            ) :
            this.next
    )
}
```

Figure 2.4: Example: a pure getter with a function body

```
ghost var aList: seq<int>;

function getLinkedListSegAList(): seq<int>
    requires acc(linkedListSeg, 100);
{
    unfolding acc(linkedListSeg, 100) in aList
}
```

Figure 2.5: Example: a getter backed by ghost state

Note that these getters always require the predicate they belong to as a precondition; hence, they are only callable when the predicate is folded. This nicely matches the original semantics in VeriFast where the parameters belong to a predicate instance.

Feasability

The above ideas of using getter functions to mock the parameters is less general than having the real parameters; hence, our approaches are not applicable for every parameterized predicate. In the following we will describe the conditions which are necessary for the translation to work.

Pure getters For the first kind of getters (pure functions) to work, there must obviously exist a functional dependency from the receiver and the other parameters to that parameter. In addition, for this approach to work the predicate analyzer must be able to extract a function definition from the predicate body. For instance, in figure 2.6 the parameter of the first predicate is uniquely determined by the heap and our analysis can extract the pure getter, while in the second case no such heap-function exists. In the third case, the parameter would actually be uniquely determined by the heap (test on $y > 1$); however, our analyzer will fail to extract the corresponding function (discussed later in section 2.2.6 about the analyzer).

```
predicate p1(boolean x) requires {
    x ? this.f |-> 1 :
        this.f |-> 2
};

predicate p2(boolean x) requires {
    x ? this.f |-> 1 :
        this.g |-> 2
};

predicate p3(boolean x) requires {
    x ? this.f |-> 1 :
        (this.f |-> ?y &*& y > 1)
};
```

Figure 2.6: Examples of predicates where some of the parameters can be replaced by a pure-getter and others cannot

In addition, some measures have to been taken to prevent cyclic getters. Consider for instance a predicate which simply asserts the equality between two parameters:

```
predicate p(int a, int b) = { a == b}
```

We cannot hope to handle both parameters by pure functions, as they are not determined by the heap. Nevertheless, for both of the parameters, the assertion-analyzer will extract a getter which consists just of a call to the other getter. To prevent such cases, our tools

builds a dependency graph for all the extracted getters and then does a topological sort; the calls represent the directed edges. Whenever there is no getter which does not depend on any other, the cycle is broken by choosing one of the getters, throwing away its functions, and falling back to ghost state instead.

Ghost state The second approach we use, is to actually store the value passed as argument in a ghost field. Therefore, instead of having it stored in the predicate instance as in VeriFast, we store it externally in a ghost field in the enclosing class of the predicate.

In order for the introduction of ghost state to work, the instance of the predicate must be uniquely determined by the heap, especially by the receiver. Only one instance can be closed on the same receiver, as we only have one field per receiver. Our tool checks, that there exists a field, for which the predicate holds full permission in all branches. In the context of the translation of predicates, we will refer to this field as *guard*. That check is sufficient but not necessary as no two instances can both simultaneously hold full permission on the very same field. Technically, having more than 50 percent would also be sufficient for having mutual exclusion; however, we need full permission an a field for dealing with the permission to the ghost field as explained in the next section.

As a very simple example, consider the excerpt from VeriFast's 'ArraysManual' example:

```
//@ predicate person(int minAge, Person person; int age) =
    person.age |-> age &*&
    minAge <= age;
```

As we will discuss in section 2.3.1 about the handling of static predicates, 'person' will become the new receiver which leaves 'minAge' and 'age' as parameters. The parameter 'age' directly maps to the corresponding field and, hence, can easily be implemented with a pure getter. On the other hand, there is no way of extracting 'minAge' from the heap; the predicate can be closed with any value smaller or equal to age. However, that predicate can only be closed once for every person, due to requiring full access on the 'age'-field; therefore, we can introduce a ghost-field in the 'Person' class and just store 'minAge' there as well.

One can think of more complex properties which might ensure the mutual exclusion for specific predicates, but our approach turned out to general enough to deal with all examples to which our use of ghost state applies. Furthermore, we limit ourselves to cases where the receiver uniquely determines the predicate instance while in theory the whole heap could be considered. However, this would introduce the additional non-trivial burden of determining where we can store the ghost-fields.

Access permissions for ghost state

When introducing ghost state the question about when to hold permission to those fields arises. We want full permission inside the predicate, but what if the predicate is unfolded, then the permissions are split and later folded at another location? This problem, for instance, showed up in our segmented linked-list example, where one can switch between the total-list predicate and the segmented one. Especially, we must not lose permission to the ghost-state when converting to the other predicate, otherwise that predicate can never be converted back again.

Our tool implements a shadowing technique where permission to the guard, on which we know to have full permission on, is shadowed. Whenever the VeriFast code mentions permissions to the guard with some fraction, permission to the ghost field with the same fraction is added as well. The only exception is the defining predicate itself, where we introduce full permission in the beginning, rather than just after the guard, so that we can freely mention the ghost field inside the body just as we could the original parameter. Outside the predicate body, that ghost field is obviously never referred to as it replaces a local parameter; hence, the exact place where permission to this field is mentioned within a contract or another predicate does not matter. This technique allows the predicate to be unfolded, the permissions to be split and later for the predicate to be folded again.

For instance, consider the situation in figure 2.7, where the parameter of the 'valid' predicate is replaced by the ghost field 'p', which shadows 'value'. Note that permission to 'p' has been introduced with the same fraction as to 'value' inside the other predicate 'A.readValid' and in the contract of 'B.g'. So whenever we regain full permission to 'A.value', we will also regain full permission to 'A.p' and the predicate can be folded again. The only drawback of this method is that it breaks abstraction up to some point by exposing the existence of that field in contracts while that field only exists for implementation purposes and is abstracted by the corresponding getter.

Relation to in- and out-parameters in VeriFast

VeriFast has the concept of distinguishing between in- and out-parameters. We will now discuss how the distinction between getters with functions and those with ghost state maps to VeriFast's concept. One might expect that in-parameters map to ghost state and out-parameters to functions. In practice our tool does not care too much about VeriFast's distinction. It turned out that most of the VeriFast examples do not mark parameters as out-parameters, since most of them were written before that concept of marking out-parameters was introduced. Therefore, relying on that information would require us to introduce unnecessary ghost state in many cases. However, our assertion-analyzer is general enough to be always able to extract a function whenever a parameter is marked as an out-parameter; this is due to the fact that VeriFast has some quite restrictive checks on when a parameter can be marked as out-parameter. The only case when our tool does care about the differentiation is when it comes to breaking cyclic

```
class A {
    var value: int
    ghost var p: int

    predicate valid {
        acc(this.p, 100) && acc(this.value, 100)
    }

    predicate readValid {
        rd(this.value) && rd(this.p)
    }
}

class B {
    method m(a: A)
        requires a ≠ null && a.valid
        ensures a ≠ null && a.valid
    {
        unfold a.valid
        fold a.readValid
        g(a)
        unfold a.readValid
        fold a.valid
    }

    method g(a: A)
        requires a ≠ null && acc(a.value, 50) && acc(a.p)
        requires a ≠ null && acc(a.value, 50) && acc(a.p)
    {
        a.value := 5
    }
}
```

Figure 2.7: The permission to the field *value* is shadowed for the ghost field p

dependencies: extracted getters for in-parameters have higher priority when needing to throw away a getter. In summary, our tool handles out-parameters in a more general way than VeriFast does by often transforming the predicate to an equivalent one where a parameter can actually be an out parameter.

Once more, have a look at our linked-list predicate from figure 2.3: we have already reasoned that the 'aList' parameter is uniquely determined by the heap and, hence, is an out-parameter. However, VeriFast does not allow having an inductive-switch on an

out-parameter.

In our tool, we achieve making 'aList' an out-parameter by transforming the inductive-switch to a conditional in the following way:

```
switch(aList) {
        case nil: return jList == null;
        case cons(aX, aNext): return jList != null;
}
↪
aList == (jList==null ? nil : cons(aX, aNext))
```

This transformation relies on the fact that the assertions of the branches are pairwise exclusive; concretely, 'jList' actually determines which branch of the switch has to be taken. Hence, we can use a conditional on 'jList' instead of the switch. VeriFast, on the other hand, conservatively checks whether a parameter can be used as an out-parameter.

Handling the call-site

There exist three different ways in which how the parameters can be used at the call site.

- A value can be passed as an argument to the predicate; thus, the parameter is used as an in-parameter.
- The value can be bound to a local variable; the parameter is used as an out-parameter.
- The caller does not care about the value, denoted by an underscore in VeriFast. The parameter is treated like an out-parameter but the value cannot be referred to.

The translation of those usages is dependant on the kind of the caller. Some care has to be paid especially to the point at which point out-parameters get replaced by retrieving the value from the corresponding getter. Note that for a predicate with a maybe-null receiver, a conditional of the form obj != null ? obj.getVal() : c_{val} (where c_{val} denotes the constant value that replaces the parameter in case the predicate is not held) replaces any call to the getters; yet we will only talk of calling the getter for conciseness.

Precondition When rewriting the parametrized predicates that occur in preconditions, the following transformations are applied:

- In-parameters get replaced by an equality check between the corresponding getter and the passed-in value.

- Out-parameters: the usage of the introduced variable is handled depending on the place it gets mentioned. In general, VeriFast's semantics are that the out-parameter introduces a local variable with constant value during the whole method. We, therefore, need to mock that capturing semantics:
  - Inside the precondition, the usage will just be replaced by a call to the getter.
  - At the beginning of the body of the method the precondition belongs to, the value of the getter gets stored in a local variable. All references to the bound variable inside the method now become mentions of that local variable. Obviously, the obtained value at the beginning of the method cannot yet differ from the one in the precondition and the predicate is still folded at that point.

    A naïve approach might try to replace every mention of the bounded variable to a getter-call; however, this would neither reflect the correct semantics as the value might have changed further up since the beginning of the method, nor be always valid as the predicate might have been unfolded meanwhile.
  - Inside the postcondition the getter is used, but in the 'old' context which ensures that we will get the same value as in the precondition.

Postcondition In postconditions we handle the parameters basically the same way as in preconditions. We can just omit the whole part about when to bind the value and just always call the getter directly. Furthermore, we must not put it in the old-context.

Unfolding

- In-parameters are neglected when translating an open statement to an unfold statement in Chalice. In VeriFast only a predicate that matches those arguments can be opened, and since we know the predicate is unique for a given receiver, that unique instance ought to have those values.
- Out-parameters get their values stored in matching local variables right before the open statement.

Folding

- In-parameters are handled depending on whether the getter is backed by ghost state or not. For getters backed by ghost state, the passing of a value gets replaced by an assignment to the ghost-state. The other in-parameters get ignored as for the unfold statement.
- As for the unfold statement, out-parameters get bound to a local variable; however, that here that must occur just after the folding since the getter is only valid once the predicate is folded.

Dealing with recursion

Recursive predicates are a common class of predicates with parameters. Any kind of recursive data type like lists or trees naturally leads to a recursive predicate. Hence, correct handling of recursive predicates is crucial for many examples to verify.

Conceptionally, there is nothing special about recursive predicates and they are treated just like any other predicate with parameters. However, some technical details had to be considered in order to let them work nicely. Those tricks mainly result from the limitations of the asssertion-analyzer: it only considers equalities when trying to extract the getters. Hence a predicate invocation of the form next.valid(vnext) cannot be handled whereas next.valid && next.getPNext() == vnext can be handled. Hence when trying to extract the recursive function for the parameter 'pnext', the recursive call must already have been rewritten to the getter syntax. Our tool handles that case by rewriting the call-site early on and binding them to stub-getters before trying to extract the actual getter.

2.2.3 Analyzing predicates

In the previous section we have seen several places were the handling of the predicates required some form of knowledge about the predicate. Some further use-cases will also show up in section 2.3 about the handling of static methods and predicates. We now focus on the predicate analyzer which is supposed to obtain that knowledge. Especially, we have seen that the anaylzer must be able to answer the following questions (some result from section 2.3.1 too):

- Determine whether one of the parameters of a static predicate is guaranteed to be non-null by the predicate body so that we can make it the new receiver.
- Determine whether a predicate holds no permission in case a given parameter is known to be null to apply the "not-holding" trick introduced in section 2.3.1.
- Determine whether a predicate holds full permission to at least one field so that we can use ghost-state to replace the parameters of the predicate.
- Extracting getters for the predicate parameters from the predicate body.
- Rewriting the predicate body after the getters have been extracted. We will look into this point in some more detail.

Furthermore, the analyzer also has some closely related use-cases outside of the handling of predicates:

- The handling of static methods also relies on choosing one of the parameters to be the new receiver. Therefore, we must be able to determine whether one of the formal parameters of a static method is guaranteed to be non-null by the precondition.

- Our dead-code elimination relies on constant folding of expressions, which is done by the analyzer. However, here we just care about expressions and not assertions in general. As a consequence, we have split the analyzer into an assertion- and an expression-analyzer; the assertion-analyzer internally uses the expression one.

In summary, we have seen that the analyzer must be able to reason about values and permissions. Additionally, it performs constant folding which can be used to simplify predicates.

2.2.4 Semantics of knowledge

Let us first focus on the knowledge we want to extract; especially, we first take a quick look at what we actually want to model when talking about 'knowledge':

We are basically analyzing a predicate body, which is an assertion. In addition, the body of a predicate is only of interest when an instance of the predicate is currently folded an all of our use-cases are of the form: if the predicate is held, does the following property hold? Hence, when talking about the extracted 'knowledge', it is always assumed that the predicate holds. For instance, consider the assertion $1 == 2$ which if used as the body of a predicate, will cause our analyzer to extract that one equals to two. Nevertheless, this is not a problem, since such a predicate can never be folded.

Also, we want our representation of knowledge to be more lightweight and abstract than the expressions and assertions VeriFast uses; otherwise, we could just extract the body itself as the most precise form of knowledge and, therefore, not gain anything. In the end it boils down to a trade-off between having simple and usable facts and being as expressive as possible. Having a more restricted set of constructs, however, also implies that we cannot model every detail of the original assertion; hence, the assertion cannot be reconstructed from the knowledge in general.

Formally, this is modelled as an implication: holding an assertion 'a' implies the knowledge of this assertion to be true. Note that the converse, therefore, does not necessarily hold: $\neg a \nrightarrow \neg knowledge$. In some cases, however, we will see that we do care about having the negation at hand. Then, however, we must ensure that our knowledge is equivalent to the assertion; hence, the implication becomes an equivalence. In this case we say that the knowledge is *precise*.

Additionally, we only care about knowledge for boolean expressions like comparison and not for arbitrary expressions like '1+1'. Boolean expressions are a subset of assertions and our assumption that the expressions implies the knowledge is only meaningful for expressions that evaluate to true or false.

2.2.5 Representation of knowledge

Value Facts

In our tool, two basic forms of facts about values exist:

- Equalities
- Inequalities

These facts are used to reason about the values of local variables (usually defined by formal parameters) or fields. However, the left and right hand sides of a basic fact are general expressions, which e.g., allows the representation of conditional knowledge of the form

$$v \equiv cnd\,?a : b$$

Reasoning always happens with **conjunctions** of basic value facts. This encoding was chosen since all our use-cases of the analyzer require that we know some value-fact for sure, and if we would encode disjunctions, they would needed to be simplified at the end. As a consequence, we have chosen to ignore them completely for the sake of simplicity; therefore, disjunctions need to be encoded as a conditional expression inside the basic fact, which turned out to be expressive enough when it comes to extracting pure getters.

One of the main drawbacks of not being able to encode disjunctions is that we cannot properly negate facts. In some situations, however, taking the negation of a fact is desirable, e.g. one would like to assume $\neg c$ inside the else-branch of an expression of the form c ? a : b. As discussed, even if the condition c does not contains any disjunction, taking the negation might not possible since our facts are less expressive than the expressions of VeriFast and therefore the semantics of our knowledge k for the expression c is defined to be $c \Rightarrow k$. However, the conditions are in practice often very simple expressions like a single equality or inequality, which can be easily expressed by an equivalent fact; in those cases, our facts will be marked as precise.

In summary, the negation of a conjunction of facts is only meaningfully representable if the conjunction is a precise singleton set. Otherwise, the negation is the empty knowledge set (treated as imprecise).

Permission Facts

The analyzer reasons not only about values, but also about the permissions held by a assertion. As we have seen, we are especially interested to know whether an assertion either holds no permission at all or holds full permission to a specific field.

Therefore, for representing permissions in the analyzer we have chosen the following three kinds of *permission facts*:

ϕ **(no permission)** The assertion is known to not hold any permissions

ψ **(unknown)** The assertion might hold permissions of an unknown fraction on one or more fields. ψ is used in two situations: when the assertion contains a predicate-assertion (which wasn't unfolded automatically during the analysing), and when an exact fraction of the permission could not be determined, as for $[?p]this.f \mapsto v$ or $[\_]this.f \mapsto v$ or when the fraction is not a constant expression.

$\pi(e.f, p)$ **(known fraction p on f)** The assertion holds at least a known fraction on a single specific field location.

Two operations over those basic permission facts are defined. For the **addition**, we have the following simplification rules:

$$\begin{aligned}
\phi + \phi &:= \phi \\
\phi + \psi &:= \psi \\
\phi + \pi(e.f, p) &:= \pi(e.f, p) \\
\psi + \psi &= \psi \\
\psi + \pi(e.f, p) &:= \pi(e.f, p) \\
\pi(e.f, p_1) + \pi(e.f, p_2) &:= \pi(e.f, p_1 + p_2)
\end{aligned}$$

Note that since fractions are always positive, adding an unknown fraction ψ to $\pi(f, p)$ results in at least a fraction p as well. Strictly speaking, one could even model that case as strictly-larger and, hence, introduce a fourth kind of permission-fact. However, since we only really care in ϕ and π we omitted that case.

Furthermore, the **disjunction** $\vee$ over basic permission facts (and additions of basic facts) is defined. This has been done, since in contrast to the value-facts, we do not want to have general expressions inside the facts. The addition is extended over those disjunctions in the natural way, too.

$$(a \vee b) + (c \vee d) := (a + c) \vee (a + d) \vee (b + c) \vee (b + d)$$

Internally, the assertion-analyzer always represents permission facts in DNF. In contrast to the value facts, however, we do have disjunctions as well as conjunctions (addition). This was chosen because we cannot model conditionals inside single permissions facts, as it is the case for value facts.

2.2.6 Analyzing and folding expressions

In this section we will describe how the analyzer generates and propagates knowledge through expressions. The process of constant folding the expressions (simplifying the expressions) is well known and will not be discussed in detail. Note that while the extracted knowledge is subject to the implication semantics of the knowledge, the simplifications done by the constant folding are conservative and valid without further assumptions.

Subexpressions are evaluated and folded recursively. During the recursive process, the analyzer keeps track of a set of assumptions primarily gained from branch conditions.

Boolean literal For a boolean literal 'v', a fact of the form $true == v$ is generated and marked as precise. This is due to the fact that the knowledge is only valid when the expression 'v' actually evaluates to *true*, as we have modelled with the implication.

Equality expression (==) As for every non-atomic expressions, the sub-expressions get analysed and folded recursively beforehand. Then, the equality fact $old(lhs) == old(rhs)$ is generated and treated as precise. "Old" denotes here the subexpression before the recursive folding. We could theoretically also generate a fact about the folded equality; however in the end we want the knowledge to be about the original assertion.

Also note that the knowledge from the sub-expressions is not reused, since that knowledge only applies whenever the expression evaluates to true. This cannot be assumed for the operands just assuming the equality holds.

Constant folding:

- if the left- and right-hand-sides are structurally equal, the result becomes *true*
- if the analyzer can prove the lhs and the rhs to be equal from the current assumption set, then the result becomes *true* as well. Here 'proving' means that, either the two terms are structurally equivalent, or that the analyser can derive it from the equality-facts in the current assertion set.
- otherwise the expression remains unchanged.

Disequality expression (!=) Similar to the equality, one inequality fact is generated. Also, similar to equality-expressions, the current assumptions are used to try to prove the inequality.

Conditional (ternary operator) The condition is evaluated under the same assumptions as the expression itself. For recursively evaluating the then-branch, the knowledge of the condition is added to the assumptions. For evaluating the else-branch, the negation is added to the assumptions.

- If the branch-condition has been folded to *true* or *false*), the knowledge of the according branch (*then* for *true*, *else* for *false*) is taken.

  In addition, the constant folding can rewrite the conditional expression to just that branch.

- If exactly one branch got folded to false, the knowledge of the other branch is taken. In addition, we have learned which value the branch condition must evaluate to and can therefore either add the knowledge of the branch condition (if the else-branch evaluates to false) or its negation to the knowledge about the conditional. Since the knowledge about the condition basically models the condition itself, this is nothing different than recording to which value the condition evaluates.

  Once more, remember that our knowledge is modelled by an implication; hence, is only meaningful when the conditional-expression evaluates to *true*. Therefore, knowing that one branch evaluates to false for sure implies that this branch cannot be the one which will be evaluated. Those semantics are motivated by the fact that we mainly analyze assertions in a context where they hold, as when a predicate is folded. This is a form of backward-reasoning which one frequently does when looking at a predicate body by hand. For expressions, this reasoning might look surprising, since for example in an expression b == (c ? false : true) where b is *false* we will generate the knowledge for the right-hand side that c will be *false* as well. This is obviously not true when looking at the top-level expression. However, that knowledge will not be part of the knowledge about the equality. Hence, this reasoning is still sound for sub-expressions, although not of great use.

- Otherwise we have to 'condition' our knowledge to represent the uncertainty of the branch. As discussed, we cannot represent the disjunctions directly but must put them inside the expressions. Hence, facts with a common left- or right-hand-side are collected (e.g., $f == g$ and $f == h$) and a new fact of the form f == cnd ? g:h (where *cnd* denotes the condition of the expression we are analysing) is generated.

The knowledge about that conditional expression is marked as precise iff the knowledge of the used branches and the conditional is precise.

Logical Not For the knowledge about a negation $\neg n$, we take the negation of the knowledge of the expression n. As already discussed, the negation of the knowledge is only meaningful if it is a precise singleton-set, otherwise we discard all knowledge.

Constant folding is only applied if n has already been folded to (or originally was) a boolean literal itself.

And

- The conjunction of the knowledge from the subexpressions is taken. It is marked as precise iff both of the subexpressions are precise. Since we can directly express conjunctions in our knowledge, this is the straightforward way of dealing with it.
- The constant folding will replace the conjunction by 'true' iff both of its subexpressions have been folded to 'true' as well; if one of them got folded to 'false', we can fold the whole conjunction to 'false'.

Or The intersection of the knowledge from the subexpressions is taken; remember that knowledge is represented as a set of basic facts which model conjunctions of them. The generated knowledge is marked imprecise unless both the sets of the subexpressions were identical and precise.

Non boolean expressions Other expressions remain unchanged and we have said that we do not define the knowledge for them. For instance, we cannot extract any knowledge from a numeric expression like $3 + 1$, since our knowledge covers only equalities and inequalities.

2.2.7 Analyzing assertions

Assertions get analyzed equivalently to expressions; we recursively extract the knowledge and constant-fold them. Especially, when arriving at an expression in the recursive process, that expression gets analyzed the way we discussed before.

In addition, when analyzing assertions we can now also extract knowledge about the permissions held by the assertions.

Empty Assertion (emp in VeriFast) VeriFast distinguishes between the 'true'-assertion and the empty assertion. However, in Java programs that distinction is not of any importance. Hence, we cannot extract any value facts, but know that this assertion does not hold any permissions; thus, we can add the permission fact ϕ

Separation Assertion In terms of value-facts and constant-folding, a separation assertion 'a &*& b' is treated just like the conjunction 'a && b'. Furthermore, the permission held by the separation assertion are known to be the addition of the ones held by a and b.

Points-To Assertion In terms of value-facts, a points-to assertion $[p]obj.f \mapsto v$ generates an implicit $obj \neq null$ and an equality $obj.f = v$ to match the semantics of VeriFast.

Depending on whether p is actually passed in (for instance $[1/2]obj.f \mapsto v$), hence an expression, or a binding of the fraction (such as $[?x]obj.f \mapsto v$), either the permission-fact $\pi(f, p)$ or ψ is generated. Whenever the fraction is bound, we do not know any lower bound on it, we just now for sure from the semantics of VeriFast that the fraction is non-zero which is represented by ψ.

Access Permission Access permissions of the total separation logic are treated exactly the same way as points-to assertion, except for the value-equality not being present.

Our tool internally uses this superset during the simplifications, and supporting both points-to assertions and access permissions results in greater flexibility in which order the simplifications are executed.

Access-All Permission Currently, we do not model access-all permissions (access to all fields of an object) directly. The only place were they might occur is in the precondition of the constructor that we generate ourself and since there is no application of our analyzer that deals with constructors we do not care about it. For correctness, we nevertheless insert generate a ψ to encode that we do have some permission, as we sometimes are interested to know that an assertion holds no permission for sure.

Predicate Assertion Preconditions of methods are often expressed in terms of predicate assertions; therefore, when analyzing such preconditions (e.g., to determine how to transform a static method into a non-static one) automatic unfolding of predicate assertions is crucial. For predicate assertions we, hence, do automatic unfolding during the analysing and basically take the knowledge we got from analysing the unfolded predicate.

We have to be careful, however, not to mix knowledge about variables inside the predicate and those outside with equal names. Also, we have to limit the recursion depth in order to prevent infinite unfolding. For our cases, a maximal depth of one turned out to be enough.

Note that if we want to use the analyzer to extract the getter-functions, it is important to rewrite the predicate-assertions, to the form where the parameters have been replaced with getters, ahead of running the analyzer. Only once we have rewritten the predicate-assertion from $obj.pred(e)$ to $obj.pred\,\&*\&\,obj.getV() == e$, the analyzer can extract the equality about the passed in argument e and the formal parameter v. In the form with arguments, our analyzer cannot establish any equality-fact on the passed-in value. We have already elaborated this aspect in the section about the handling of recursive predicates (section 2.2.2).

Implication Assertion When analyzing an implication assertion $cnd \Rightarrow thn$ we obviously have to match the semantics of the implication:

- if *cnd* got folded in the recursive call to *true*, we know that the *thn*-part must evaluate to *true* in order for the whole assertion to evaluate to *true*. Hence, we can assume both the knowledge and the permissions of the *thn*-part.
- if *cnd* has been folded to *false*, we cannot assume anything about the *thn*-part. Therefore, we just know that *!cnd* holds and the corresponding knowledge can be added. In addition, we know that the implication does not hold any permissions since in VeriFast the *cnd* is a pure expression which cannot contain permissions. The implication-assertion can, furthermore, be simplified (fold) to just *true*, matching the natural semantics of implications with *false* as a condition.
- if the *thn*-part has been folded to *false* we know that the *cnd* must also evaluate to *false*; hence the appropriate knowledge gets added and we know that the implication does not hold any permissions. Also, we can simplify the implication-assertion to *!cnd*.
- Otherwise we cannot assume anything for the value-facts since our encoding of value-facts does not allow to express a disjunction or implication; however, we can add the disjunction for the permissions.

Conditional Assertion Conditional assertions get treated almost identically to conditional-expressions. The basic difference is that for the permission-facts we can properly encode the disjunction in the case that we cannot rule out one of the branches.

Inductive-Switch Assertion Inductive-Switch assertions are treated quite similarly to conditional-assertions. If the structure of the expression e on which the switch acts on ($switch(e)\,\{...\}$) is known (e.g., either *cons* or *nil*), we can simplify the switch to the according branch. This includes replacing the bound variables from the case-statement with the corresponding subexpression of e. For instance, if e is $cons(1, x)$ and the case-statement of the form $case\, cons(h, t) : [body]$, the variable h will be replaced with 1 inside the case-block, and t with x correspondingly.

Otherwise, if only one branch is satisfiable (not has been folded to false), we cannot just simplify the assertion (for folding), but at least can just take the knowledge of that branch. This is due to the fact, that whenever the assertion holds, this branch must be satisfied. Furthermore, we can add an equality-fact on the structure of e to our knowledge.

Last, if neither of the previous cases applies, we need to "switchify" our equality-facts. This means that we put the switch inside the right-hand side of equality facts which share the same left-hand side, by using conditionals. This is a generalisation of what the analyser does when conditioning facts as described for conditional-expressions.

2.2.8 Solving equalities

We have now seen how we can extract knowledge from assertions. For values, that knowledge is represented by a set of equalities and inequalities which technically we now need to solve for the variable we are interested in.

We first tried an approach where we assumed that the equations are always already solved for the variable we are interested in; hence, we only considered value-facts where either the left-hand or right-hand side was exactly that variable. While this assumption might look quite strong at first sight it actually works out for many examples. For instance, consider a points-to assertion $this.f \mapsto v$ where v is the variable we are interested in. When analyzing that points-to assertion we will produce the equation $f == v$, which obviously doesn't need to be solved. Actually, it turned out that for many predicates the parameters just get directly mapped to a field; hence, for those examples there is no need of equation solving.

However, at some point it became clear that for many recursive predicates there exist both a predicate definition that does not need equation solving and an almost identical one that does. As an example, let us look at a simple (non segmented) linked list predicate in two versions, as shown in figure 2.8 and 2.9.

```
predicate linkedlist(List l, int length, list<int> items) = {
    l != null &*&
    l.next ↦ ?n &*&
    l.val ↦ ?v &*&
    (n == null ?
        length == 1 &*& item = cons(v, nil) :
        linkedlist(n, length-1, ?i) &*& items == cons(v, i)
    )
}
```

Figure 2.8: A predicate where a definition for *length* cannot be extracted without solving an equation

```
predicate linkedlist(List l, int length, list<int> items) = {
    l != null &*&
    l.next ↦ ?n &*&
    l.val ↦ ?v &*&
    (n == null ?
        length == 1 &*& item = cons(v, nil) :
        linkedlist(n, ?len, ?i) &*&
        lenght == len + 1 &*&
        items == cons(v, i)
    )
}
```

Figure 2.9: A predicate where a definition for *length* can be simply extracted

Both of them look reasonable and there seems no reason to believe that a programmer might have a strong preference for one of them. Hence, we would like to support both of them (or fail for both), but not support one and fail for the other. However, for the first one the analyzer will extract the equation $length - 1 == l.next.getLength()$ which does need solving for *length*, whereas for the second one the equation $length == l.next.getLength() + 1$ will be generated.

As a consequence we realized that some sort of minimalistic equation solver was needed. Since we do not expect considerably more complicated equations than the $length - 1 == v$ above, we kept it simple and made the following assumptions:

- The solver only supports the basic arithmetic operations, namely: addition, subtraction, multiplication, and division.
- The variable we are solving for is only allowed to be mentioned once inside the equation.

Since we deal with multiplication we have to account for the case that we might generate a division by zero; therefore, some side conditions might have to be added. For instance, when solving $a * b == c$ for a, the equation $a == \frac{c}{b}$ is only valid if b is non-zero and unless b is a compile-time constant we cannot rule that case out during the translation. The remaining question is: where do we place the side-condition? We have considered two options:

- Only as a precondition to the getter-function.

  Remember, that we mainly extract the knowledge to built the getter-functions that replace the predicate parameters. Hence, the division will only occur inside the getter; thus, it makes sense to add the side condition as a precondition. However, Chalice would then have to figure out that the side-condition holds for every call-site. This is problematic since the required knowledge to prove the side condition

is usually encoded in the original predicate, which is folded when the getter gets called (and so the information may not be "seen" by the verifier, as predicates are not unfolded automatically).

- Inside the predicate body.

  We have just discussed that the necessary knowledge to prove the side condition is usually encoded in the predicate itself. Hence, it may make sense to put the side condition right there as well. We would then carry that knowledge with the predicate instance and get it for free inside the getter which unfolds the predicate. However, in the end that would just shift the problem of proving the side condition from the call-site to the folding of the predicate.

Actually, we believe that for many cases the side condition will not be directly derivable from the predicate, anyway. For instance, in figure 2.10 the reasoning has to be done inductively (base-case: $l_0 = 1$; inductive case: $l_n = l_{n-1} + 1$), which Chalice cannot perform automatically. In fact, a stronger induction hypothesis will have to be added to the predicate body.

```
predicate ListAverage(int a, int l) requires {
    this.value |-> ?v &*&
    this.next |-> ?n &*&
    (n != null ==> (n.ListAverage(?b, ?m) &*&
                    m == l-1 &*&
                    b*m+v == a*l)
    ) &*&
    (n == null ==> (a == v &*& l == 1))
}

function getA(): int
    requires ListAverage &&
             getL() ≠ 0; // side-condition
{ (b*m + v)/getL() }
```

Figure 2.10: A predicate where the extracted getter for the parameter a will need a side-condition, which is not directly derivable from the predicate body

As a consequence, the user will be forced to fix the translated code manually. Hence, we opted for the easier solution and just included it in precondition; the user will then need to fix it manually by adding a suitable condition to the predicate and possibly remove the precondition, as shown in figure 2.11.

```
predicate ListAverage(int a, int l) requires {
    // insert such a condition in the translated Chalice code
    length != 0 &*&  code
    [...]
}

function getA(): int
    requires ListAverage; // side-condition removed!
{ (b*m + v)/getL() }
```

Figure 2.11: The example of figure 2.10 manually fixed

2.2.9 Simplifying predicates

We have now seen all the techniques required to construct the pure getter-functions from the predicate using our analyzer: the predicate is analyzed to extract the knowledge, and then the equality-facts are tried to be solved for the parameter we are replacing. However, we actually not only want to construct the getters but literally extract them; thus, removing the information it covers from the original predicate body itself. This both has an ideological and a technical reason:

- Remove redundancy: extracting some part of the predicate into a getter implies that we extract parts of the information provided by the predicate; the equivalent information in the predicate body is then redundant. It is not just our goal to create Chalice code that verifies but also code that is concise and readable.
- Since we want to remove the parameter, we cannot refer to it inside the predicate body any longer. In addition, we cannot simply replace it by a call to the corresponding getter because our getter requires the predicate to be folded when called. Finally, just removing every expression that has an occurrence of the parameter might not work; not every occurrence necessarily contributed to the getter definition.

So how do we remove the part covered by the getter? We plug in the definition (body) of the getter for every occurrence of the parameter the predicate and let the constant folding of the analyzer take care of the redundancy.

As an example, let us once again look at the non-segmented linked list. This time, we have also added an assertion that the abstract list has a non-zero length for the second case, for the purpose of adding some information about aList which is not used in the getter.

```
predicate linkedList(LinkedList jList, list<int> aList)
  requires switch(aList) {
    case nil: return jList==null;
    case cons(aX,aNext): return jList!=null
        &*& jList.value |-> ?jX
        &*& jList.next |-> ?jNext
        &*& linkedList(jNext,aNext)
        &*& aX==jX
        &*& length(aList) > 0;
  };
```

Figure 2.12: The original predicate in VeriFast with parameters

After an initial rewrite that makes the predicate non-static and binds the arguments of the recursive predicate call to the corresponding getters, we get:

```
predicate linkedList(list<int> aList)
  requires switch(aList) {
    case nil: return this==null;
    case cons(aX,aNext): return this!=null
        &*& acc(this.value)
        &*& acc(this.next)
        &*& this.next.linkedList && this.next.getAList() == aNext
        &*& aX==this.value
        &*& length(aList) > 0;
  };
```

Now, our getter for aList can be extracted from the body:

```
function getAList(): seq<int>
    requires acc(linkedList, 100);
{
    unfolding acc(linkedList, 100) in
    cons(this.value, this.next ≠ null ?
                        this.next.getAList() : [])
}
```

If we now plug the getter definition back into the predicate and remove the parameter we get a predicate that can directly be translated to Chalice, as shown in figure 2.13.

However, this new predicate contains a lot of redundancy which we would like to eliminate; therefore, we now apply the constant folding. As a first step, we remove the nil-case, since with the non-static predicate that is no longer possible. In addition, we can bind

```
predicate linkedList()
  requires switch(cons(this.value, this.next ≠ null ?
                          this.next.getAList() : [])) {
    case nil: return this=null;
    case cons(aX,aNext): return this≠null
        &*& acc(this.value)
        &*& acc(this.next)
        &*& this.next.linkedList
        &*& this.next.getAList() = aNext
        &*& aX=this.value
        &*& length(cons(this.value, this.next ≠ null ?
                            this.next.getAList() : [])) > 0;
  };
```

Figure 2.13: The predicate in which the parameter has been eliminated

```
predicate linkedList()
  return this≠null
    &*& acc(this.value)
    &*& acc(this.next)
    &*& this.next.linkedList
    &*& this.next.getAList() =
            (this.next ≠ null ? this.next.getAList() : [])
    &*& this.value=this.value
    &*& length(cons(this.value, this.next ≠ null ?
                          this.next.getAList() : []
              )) > 0;
};
```

Figure 2.14: The predicate where the inductive-switch has been removed

the values of 'aX' and 'aNext' to the corresponding expressions, as we can structurally match the cons-pattern to the expression we do the switch on (figure 2.14).

Furthermore, we know that $this != null$ is generally true. After analyzing the first such inequality, we also add it to the assumption set; hence, for the second and third occurrence we could actually assume it even if it were not an expression that trivially holds, as shown in figure 2.15.

```
predicate linkedList()
  return acc(this.value)
    &*& acc(this.next)
    &*& this.next.linkedList
    &*& this.next.getAList() = this.next.getAList()
    &*& this.value = this.value
    &*& length(cons(this.value, this.next.getAList())) > 0;
};
```

Figure 2.15: The predicate after simplifying expressions of the form $this != null$

Finally, we know that the two remaining equalities trivially hold since they have a structurally equivalent right-hand and left-hand side. Now we have our simplified predicate that can easily be translated to Chalice:

```
predicate linkedList()
  return acc(this.value)
    &*& acc(this.next)
    &*& this.next.linkedList
    &*& length(cons(this.value, this.next.getAList())) > 0;
};
```

Figure 2.16: The translated predicate

Note that the part about the length was not removed during the simplification as that part is not included in the extracted getter. Strictly speaking that part is also redundant; however, it was in the original predicate as well and our analyzer does not know about the semantics of 'length'.

In summary, the approach of using the constant folding to remove the introduced redundancy works well, because the control structure (conditional expressions, switches) of the extracted getter corresponds to the one from the predicate. Hence, using the branch conditions from the predicate body, the according part from the getter definition can be picked which leads to trivial equations.

2.2.10 Possible extensions

While we have implemented all fundamental methodologies to handle predicates we could think of, especially the analyzer and equation solver could still be improved in various ways.

First, the transitivity of equality is currently not taken into account when trying to prove facts. Some care would need to be applied, in order to prevent infinite recursion. So far, this never turned out to be a problem; however, one can construct artificial examples where this would start to play a role.

Second, one could consider to integrate range analysis for integers. Enhancing value facts to include information about ranges would generalize the reasoning, and especially allow reasoning about inequalities which play a role for lengths and sizes of for instance lists and sets.

2.3 Static and global context

In order for our tool to be of practical use, it must be able to deal with a wide variety of VeriFast programs. Static fields and methods are of those parts which are frequently used in Java and thus VeriFast; however, do not have a direct equivalent in Chalice.

In the following section we will explore how, and to what extent, we have dealt with static methods, fields, lemma-methods, predicates, and fixpoint-functions.

2.3.1 Handling static predicates

The basic idea is to select one of the reference parameters and make it the new receiver; hence, move the predicate to the corresponding class.

In order for this to work, the predicate must have at least one reference parameter of a user defined class which meets the requirement that the body of the predicate ensures that parameter to be non null. Then we can rely on having an object of the corresponding class at every caller of the predicate. For knowing whether a parameter is guaranteed to be non-null, we once again rely on our predicate analyzer from section 2.2.6.

In some cases, however, we can also select a parameter which might be null. The idea is then to simply not hold / require the predicate in those cases. Requiring $p.pred$ becomes $p \neq null \Rightarrow p.pred$. That case applies if the following points are fulfilled:

- Whenever the chosen parameter is null, the predicate must not hold any permissions. Hence, we do not lose any permissions by simply not holding the predicate. In addition, not holding the predicate implies that the predicate cannot be unfolded; therefore, the unfolding must be skipped. If a predicate which holds permission were skipped, we would lack those permissions after skipping the unfolding.
- Whenever the chosen parameter is null, there must be a functional dependency for all other parameters; thus, all others parameters $p_2, ..., p_n$ must take known constant values $c_2, ..., c_n$ whenever the predicate is closed with p_1 being null.

  We can then simply extract those constant values to the call-site and write instead of $p(v_1, v_2, ..., v_n)$ an equivalent assertion

  $$\texttt{p} \neq \texttt{null} \ ? \ v_1.p(v_2, ..., v_n) \ : \ c_2 = v_2 \ \&\& \ \ldots \ \&\& \ c_n = v_n$$

  Note that a limitation in the current implementation of the tool is that it will not retry with another parameter when this point is not fulfilled; the translation will fail in that case. However if needed, this restriction could easily be lifted with some refactoring.

Again, checking all of those conditions and extracting the constant values is the duty of the predicate-analyzer.

This "not-holding the predicate" trick is correct since a predicate can only talk about the heap and its parameters. Whenever the predicate does not have any permissions to the heap, therefore, all of its information deals with what valid values for parameters are. Then, since those are also known constants that equality is equivalent to the predicate body.

While those requirements for applying the "not-holding" trick might sound quite restrictive, it turned out that it is still general enough to handle some practical cases. For instance, reconsider our linked-list example shown in figure 2.17: the next field being null encodes the end of the list, and therefore closing on the null-list forms the base case of the recursive predicate.

```
(static) predicate linkedList(LinkedList jList, list<int> aList)
  requires switch(aList) {
        case nil: return jList==null;
        case cons(aX,aNext): return jList!=null
                [...]
  };
```

Figure 2.17: Our static linked-list predicate: our "not-holding" trick applies

If we now look at that predicate by hand, it becomes obvious that the predicate does not hold any permissions whenever *jList* is null and, in addition, aList is also uniquely identified to being *nil* in that case; hence, our trick applies. We have seen also other examples of predicates where this trick is able to deal with the base-case of a recursive predicate definition (e.g. Stack.java) and, hence, believe that it should be general enough to handle most of such cases.

2.3.2 Static (lemma) methods

The overall approach is to convert a static method to an equivalent non-static one by picking one of the parameters to be the new receiver. Obviously, this requires a reference parameter which must be non-null at all call-sites; otherwise we would try to call a method on null. In practice, this approach turned out to work quite well; many methods expect their parameters to be non-null, and for most of the remaining static methods we do have some fall-back techniques that we will describe later.

In the implementation, we use our assertion-analyzer infrastructure from section 2.2.6 to try to prove that the precondition implies one of the reference parameters to be non-null. If multiple parameters satisfy that condition, always the first one is picked. That

```
void swap(Counter c2)
    //@ requires this.value |-> ?v1 &*& c2.value |-> ?v2;
    //@ ensures this.value |-> v2 &*& c2.value |-> v1;
{
    int tmp = this.value;
    this.value = c2.value;
    c2.value = tmp;
}
```

Figure 2.19: The translated method of the one in figure 2.18

parameter then gets removed from the list of formal-parameter declarations, all its usages inside the method replaced by 'this', and the method gets moved to the corresponding class.

```
static void swap(Counter c1, Counter c2)
    //@ requires c1.value |-> ?v1 &*& c2.value |-> ?v2;
    //@ ensures c1.value |-> v2 &*& c2.value |-> v1;
{
    int tmp = c1.value;
    c1.value = c2.value;
    c2.value = tmp;
}
```

Figure 2.18: the 'swap' method from VeriFast's 'Counter' example

As an example, consider the static 'swap' from figure 2.18. This static method contains two reference parameters, both of type 'Counter'. Note that the assertion of the form $c.value \mapsto ?v$ in the precondition implies c to be non-null. Hence, both $c1$ and $c2$ are known to be non-null at every invocation. As a consequence, we can pick $c1$ to be the receiver of the generated instance method and turn the method into the one shown in figure 2.19 inside the 'Counter' class.

While the technical implementation works in many cases, moving a method to another class might be problematic from the point of view of information hiding; the static method might have been an implementation detail of a certain class which now has been moved to another class. Since Chalice does currently not support any form of visibility modifiers, however, that does not cause any problems in the verification. If Chalice did support such a feature in a future version, of course, all such transformed static methods could still be marked as *public* during the translation. However, there seems to be no way of preserving the desired information hiding when moving a method to a completely

different class; if this would be desired, another approach for handling the static methods would need to be taken.

More interesting from our point of view is the question: for which kinds of methods is this rewriting even applicable? Thus, how many methods cannot be handled this way? In practice it turned out that for some methods, none of the parameters can be guaranteed to be non-null; however the method does a null-check inside the body. This was especially common with lemma-methods belonging to potentially-null predicates, as discussed in the section about dealing with static predicates before, also refereed as the "not-holding" trick. Therefore, we would like to apply a similar trick: not calling the method whenever the new receiver is null.

The "not-holding" trick for predicates, where the predicates is not held in situations when the new receiver of the generated instance predicate would be null, often applies for base cases such as the empty list segment. Our similar trick for lemma methods is now motivated by the fact that lemma methods are often closely related to the predicate instances, since lemma methods are intended to transform the abstract state of the program (abstracted by the predicate). In some cases the static lemma methods even takes the "same" parameter as the one from the predicate that could not derived to be non-null; hence, there is little chance for this parameter to be now guaranteed to be non-null in case of the lemma method. However, since such cases tend to represent e.g. base-cases, not only does the predicate not assert anything non-trivial (and can be left out), but also the lemma method tends to either do nothing or return a constant. This becomes more evident, when considering that in case the lemma method requires the predicate in the precondition, the unfolding of the predicate will be guarded with a null-check inside the lemma-method, as there exists no predicate instance otherwise. In short, such newly introduced null-checks often give raise to a situation were the lemma-method body becomes empty in case of the parameter being null.

As an example of such a "not-calling" situation, consider the lemma-method of a segmented linked-list from figure 2.20, which converts the segmented predicate to the non-segmented equivalent.

- Assume that for the segmented-list predicate, and the non-segmented version as well, the potentially-null trick applies for the first parameter. Whenever the first parameter is null, in which case the predicate denoted the empty-list segment represented by $linkedListSeg(null, null, nil)$, the predicate will not be held in the corresponding translated Chalice code.
- Hence, we cannot simply expect 'jList' to be derivable to be non-null from the predicate mentioned in the precondition. However, whenever 'jList' is null then we know, from back when we applied the not-holding trick, that 'aList' is empty.
- In addition, due to applying the potentially-null trick on the predicate, the open- and close statements on the first respectively last line get guarded by a null-check on 'jList'.

- Furthermore, one can also easily see, that 'aList' being nil leads the switch to be an empty statement.
- As a consequence, the whole method basically becomes a no-op whenever 'jList' is null and we might apply our 'not-calling' trick.

```
lemma void seg2Tot(LinkedList jList)
  requires linkedListSeg(jList, null, ?aList);
  ensures linkedList(jList, aList);
{
  open linkedListSeg(jList, null, aList);

  switch(aList){
    case nil:

    case cons(x,aR):
        if(aR != nil) {
            seg2Tot(jList.next);
        }
        else {
            close linkedList(null, nil);
        }
  }

  close linkedList(jList, aList);
}
```

Figure 2.20: A static lemma from the segmented linked-list example

In order to automatically deduce that a method does not perform any operation, we have implemented a dead code elimination described in section 2.4.2. Whenever our dead code elimination can eliminate the whole body of a method under the assumption that a parameter is null, we known that we can apply the "not-calling" trick. A drawback is that, however, we then no longer get the postcondition of that method. Even though the postcondition of a method with empty body should follow directly from the precondition, it might guide the verifier; hence, the verifier might not try to derive that property automatically. One possible solution might be to assert, or even assume, the postcondition whenever a call is skipped due to a null-receiver.

In addition, the same trick also applies for methods which return a constant expression in case of a null receiver. While we have not met an example benefitting from this, it seems like a natural extension. Also, it should cover most cases of legacy code that does null checking inside the method rather than requiring it in the precondition. For instance, the following two methods can both be handled this way:

```
int foo(A a, int x) {
    if (a != null) {
        int tmp = a.x;
        a.x = x;
        return tmp;
    }

    return -1;
}

class B {
    int v;
    boolean compare(B other) {
        if (other == null) {
            return false;
        }

        return v == other.v;
    }
}
```

Figure 2.21: Examples of static methods that can be handled by the "not-calling" trick

Fallback: inlining One important category of static methods usually cannot be handled in the way discribed above: factory methods. Factory methods tend to return a reference instead of taking them as parameters and might not take any parameters at all. On the other hand, factory methods are rarely recursive. Their non-recursive nature makes them suitable for inlining, which is performed by our tool as a fallback strategy. Obviously, our tool has to check for the recursiveness and the translation will fail whenever the static method can neither be turned into a non-static one, nor inlining is applicable. Some care has also to be taken not to create naming collisions with existing variable and parameter names at the call-site; this is achieved by renaming all parameters and local variables of the inlined method and putting them into a separate block to limit the scope of the local variables. In addition, both the pre- and the postcondition get turned into assertions.

Nevertheless, information hiding and encapsulation are violated by this transformation; however as Chalice does not have any access modifiers, this is currently more of a ideological objection rather than a technical limitation; although, Chalice might implement information hiding in the future.

As an example of a static method were we need our fallback, consider the factory method of figure 2.22 from VeriFast's 'AmortizedQueue' example. In figure 2.23, the translated

code of a call-site of this method is shown were inlining has been applied.

```
public static LinkedList New()
    //@ requires true;
    //@ ensures result != null &*& linkedlist(result, nil);
{
    LinkedList l = new LinkedList();
    //@ close linkedlist(l, nil);
    return l;
}
```

Figure 2.22: A static (factory) method without parameter

Notice how the result variable was renamed to *tmp_24*, and *l* to *tmp_25*, to avoid naming conflicts. Furthermore, the whole implementation was put into its own scope (curly brackets) to prevent the local variables from escaping. Also you can see how the contracts have been turned into assertions. In the last line, the result of the method invocation is used, which became a simple access of the temporary variable.

```
[...]
var tmp_24: LinkedList;
{
    // the precondition gets asserted
    assert true;
    {
        // the actual inlined body
        var tmp_25: LinkedList := new LinkedList;
        call tmp_25._init();
        fold acc(tmp_25.linkedlist, 100);
        tmp_24 := tmp_25;
    }

    // the postconditions gets asserted
    assert tmp_24 ≠ null &&
           acc(tmp_24.linkedlist, 100) &&
           tmp_24.getVs() = [];
}
// later...
[var e: LinkedList := tmp_24;]
```

Figure 2.23: Call-site where the method from figure 2.22 got inlined

Other possible approaches We have considered some other approaches to the handling of static methods as well:

- Firstly, simply declare the method as non-static and require the call-site to create a new instance of the class containing the method at every invocation. This approach would relieve us from having to pick one of the parameters as the receiver is always an additional, known to be non-null, instance. However, it would demand all of our classes with static methods to provide a parameter-less constructor; hence, we could no longer enforce any invariants on instances of those classes.
- As a second alternative one could think of a global 'static-context' class, which hoists all static methods of the whole program. While such a class would obviously not need any non-trivial constructors or invariants, it would impose some other problems such as naming-conflicts and break modularity.
- Last, the probably most practical solution would maybe be to have some kind of Scala-like companion class for every class containing static methods. We would then always be able to create such a helper object, but also remain modular. In retrospect, we would probably try to implement that approach, since it looks much simpler than the current implementation. Owing to the fact that our current implementation was so far able to deal with all static methods we have dealt so far, we did not have an incentive to replace the current implementation.

2.3.3 Static fields

The current implementation of the tool cannot handle any VeriFast program containing static fields. Therefore, the translation of such a program will fail. Fortunately, very few of the existing VeriFast samples use static fields.

Static fields can be viewed as a form of global state; they are available at every point of the program. Without having support for any similar feature in Chalice, the emulation of static fields would imply to having to pass some object around the whole program to every single method and function. That object could then be used to represent the global context. However, this still doesn't address the question of how the permissions to this special object should be handled, especially in a concurrent setting where we could not simply pass around write-access along with the special global object to every method.

We have discussed during the project whether static fields could be emulated by having a second class for every class in the original VeriFast program that basically mocks the class-objects of Java. However, the question of how one could make those classes singletons or at least being able to get hold of a specific instance remained unanswered.

A more feasible approach for future extension might be to only try to cope with static fields that represent compile time constants. Those could easily be propagated to the call-sites and would cover most of the existing VeriFast examples using static fields.

In the long term, however, extending Chalice to have something like static fields or singletons will possibly be the approach to pursue.

2.3.4 Static fixpoint functions

In VeriFast, fixpoint functions are quite often declared in the global scope; thus, are implicitly static.

Whereas lemma methods can be handled the same way as regular methods, we had to come up with a different treatment for static fixpoint functions. In contrast to lemma-methods they do often not operate on reference types (the actual object), but rather on the abstract data-types. As a consequence they usually don't have reference parameters which could become the new receiver. One could then think of trying to introduce a fake receiver, hence just declaring them non-static in an arbitrary class. However, fixpoint functions are most often used in contracts where one cannot simply create a new object. In addition, inlining defeats the whole purpose of abstracting the contract specification and the static functions are very often recursive which prevents inlining altogether.

Hence, it became obvious that a different approach has to be taken for supporting static fixpoint functions. In the following we will present what approach was first taken, why that approach failed, and how we now support static functions.

First approach In the first approach we nevertheless stuck to the idea of having a fake receiver, since every function in Chalice needs a receiver.

Since we cannot hope for having a reference parameter and a new receiver cannot be created for every use, we have chosen to use 'this' as the fake receiver, as shown in figure 2.24. Owing to not having a static context in Chalice, 'this' is always a valid reference parameter. This, however, implies that the function definition needs to be copied to every class where that function is called. Hence, there is the potential problem of having multiple technically equivalent functions but Chalice treating them as different ones. This will let the verification fail whenever a contract is specified in terms of such a function, but Chalice only knows that the property holds for the actually equivalent function.

```
static fixpoint int foo() { [...] }
class A {
    int bar()
        //@ ensures result == foo();
    { [...] }
}
↪
class A {
    function foo() : int { [...] }
    method bar() returns (res: int)
        ensures res = this.foo();
    { [...] }
}
```

Figure 2.24: A static function with a usage, and its translated version

Limitation of that approach Whereas that approach was able to translate any of VeriFast's fixpoint-functions, it hindered the verification of the program, not only due to the issue of duplicated definitions, but also due to the fake receiver itself.

Once again, consider the example of a non-empty linked-list. Especially, we focus on the recursive implementation of a 'contains'-method and its abstract counterpart 'aContains' shown in figure 2.25. The problem is the recursive call: we get an assertion of the form

```
next.aContains(old(next.getAList()), x)
```

from the postcondition. However, the postcondition of the current call requires the assertion with *this* as a fake receiver. Since those receivers are "dummy", and thus, the value of the function doesn't depend on them, this equivalence would actually hold. However, Chalice would need to prove the following equivalence:

```
next.aContains(old(next.getAList()), x) =
this.aContains(old(next.getAList()), x)
```

In order to avoid this problematic situation and prove that the value of the abstract function does not depend on the choice of the receiver of a recursive function, Chalice would need to do an inductive proof, which it obviously does not attempt. This is understandable, as for general functions there is no reason to include parameters they do not depend on.

Second attempt As a consequence it became clear that Chalice needs some basic support for static functions as well. That especially made sense because there are plans

to extend Chalice with inductive datatypes, which would require some form of static functions anyway.

However, after some evaluation we decided to make a minimal implementation that still relies on a fake receiver, and leave the addition of real static functions for future work. Our current implementation supports static functions of the following form:

- The 'static' keyword can be added to functions
- Those functions are not allowed to access the non-static context (hence mention 'this'). The only exception is for another fake receiver as in described in the next point.
- However, they still get called on a fake receiver of the type they are declared in.

This allows for a very simple implementation: in the translation from Chalice to Boogie, the receiver of a function is just transformed to a parameter of the Boogie-function, which now gets dropped for static functions. Minimal changes had to be made to the parser and the AST, which basically just required the keyword to be parsed and stored. In the typechecker, an additional check that a static function does not implicitly or explicitly access 'this' had to be added.

Support for real static functions would have required much deeper modifications to the parser and especially the resolution since they currently do not expect an expression of the form 'A.f' where A is a class name. While in theory doable, those changes proved to be difficult, as the parser and resolution are quite entangled.

The current implementation proved to be good enough to let most of our programs verify. However, the drawback of duplicated functions still exists, and caused the verification to fail in one of our examples.

```
method contains(x: int) returns (res: bool)
    requires acc(this.linkedList, 100);
    ensures acc(this.linkedList, 100) &&
            this.getAList() = old(this.getAList()) &&
            res = this.aContains(old(this.getAList()), x);
{
    unfold acc(this.linkedList, 100);

    // does the first element match?
    if (value = x) {
    fold acc(this.linkedList, 100);
    res := true;
    } else {
        // does the list have more elements?
        if (next = null) {
            fold acc(this.linkedList, 100);
            res := false;
        } else {
            call res_next := next.contains(x);
            fold acc(this.linkedList, 100);
            res := res_next;
        }
    }
}

function aContains(l: seq<int>, x: int): bool
{
    (|l| = 0 ?
        false :
        (l[0] = x ?
            true :
            (l[1..] = [] ? false : this.aContains(l[1..], x))
        )
    )
}
```

Figure 2.25: Example of a case were the dummy receiver prevents the verification

2.4 Control Flow

There are several language differences between Java (as used by VeriFast) and Chalice we have to overcome in the translation. One of them is the lack of early exits with return-statements; in Chalice there exists a return variable which can be assigned to, but the control flow always leaves a method regularly by executing its statements to the end.

Nevertheless, we want to translate return-statements with assignments to the result variable; thus, we need to overcome the problem of early returns. Therefore, we need to modify the body of the method when replacing return-statements with assignments to the result-variable. This is done by introducing additional if-statements that guard on a additional local variable which encodes whether the method would (in the presence of real return-statements) already have exited or not. In short, that variable encodes whether an assignment to the return variable has already happened and whenever it is set, no more code of the method should be executed, to match the early-return semantics of Java.

In addition, we want the resulting Chalice code to be as readable as possible, and in some cases that guard-variable is redundant. Therefore, some optimisations such as dead code elimination have been introduced.

2.4.1 Basic methodology

As we have already briefly explained, we replace every return-statement with an assignment to the result variable and then only conditionally execute the remaining parts of the method. Conditionally executing the remaining parts is obviously important whenever the return is inside a branch and not at the end of the method, which is often the case. For instance, consider the following example of a Java method with an early return, and the corresponding Chalice version.

```
public boolean foo(int a)
{
  if (a == 0) {
        return false;
  }

  bar();
  return true;
}
```

Figure 2.26: A Java method with an early return

```
method foo(a: int) returns (res: boolean)
{
    var isResultSet: boolean := false;
    if (a = 0) {
        res := false;
        isResultSet := true;
    }

    if (!isResultSet) {
        call bar();
        res := true;
        isResultSet := true;
    }
}
```

Figure 2.27: The translated Chalice code of Figure 2.26

Note in particular that the call to 'bar' might have side-effects and must not be executed whenever the initial branch is taken. Note that the Chalice code of Figure 2.27 does not directly correspond the one actually produced by our tool; in the next subsection we will explain some techniques which have been applied to make the produced Chalice code more readable.

Note that there is actually one caveat when making the execution of the rest of the method conditional: VeriFast allows pure statements, such as predicate folding, directly after a return statement that get treated as an epilogue, meaning that those statements still get executed right after the return statement and before the method is left. In regular Java, such code would be unreachable, which would also be reflected by introducing a guard right after the assignment. Hence, guards are only introduced after the current scope (e.g. the current if-statement, while-loop, block, or the whole method) has been left.

One drawback is that this could cause our tool to make truly unreachable code (impure statements after a return-statement) reachable again; however, VeriFast itself rejects such programs.

Similarly to if-statements, we also need to handle returns in while-loops: the rest of the loop body should no longer get executed, the loop condition should not be evaluated, and no further iteration should be performed. Therefore, we extend the loop-condition to include a test on the result-set variable first.

As discussed in Section 2.5, Chalice only allows pure expressions as loop conditions and for instance not method calls. Hence, the loop condition might have to be "pulled out" and evaluated before the loop and at the end of the loop-body. Therefore, we

```
if (a = 0) {
    res := false;
    isResultSet := true;
    // note: we do not guard on !isResultSet here
    // the following code is an epilogue and not unreachable
    close(this.valid);
}

// here we must now guard...
if (!isResultSet) {
    ...
}
```

Figure 2.28: Example of a method with an 'epilogue'

have to make sure that our return-statement elimination here does not interfere with that transformation of impure expressions, which is done in an earlier stage in the translation.

On the one hand, that "pulling out" makes our life easier by ensuring that the new loop-condition is pure and we do not have to worry about executing it too often; hence, we can just add the guard to the loop condition as shown in Figure 2.29. On the other hand, that "pulling out" has just placed the actual evaluation of the loop condition at the end of the body. In the case of a loop that returns unconditionally, as shown in Figure 2.29, this evaluation of the loop-condition (call to 'c') is, therefore, actually dead code and should never be executed. We now have to ensure that we do not treat that evaluation as an epilogue of the return-statement; otherwise, we would execute one more time than desired.

Break-statements can be handled in a similar way and continue-statements are currently unsupported by VeriFast. In addition, other loops get converted to while-loops in an earlier stage; hence, our tool can handle all kinds of loops.

```
// first evaluation of the loop condition, pulled out
call loopCnd: boolean := c()

// loop with a guarded condition
while (!isResSet && loopCnd) {
    [...]

    // this loop always returns in the first iteration
    res := false;
    isResSet := true;

    // not an epilogue of the return-statement
    // but dead code introduced by our translation
    // Dead code elimination will remove this...
    if (!isResSet) {
        // actual evaluation of the loop condition
        call loopCnd := c()
    }
}
```

Figure 2.29: Correct handling of the evaluation of the loop-condition

2.4.2 Optimizations

We will now discuss some of the optimizations our tool performs in order to keep the code concise and readable. First, reconsider the previous example from Figure 2.26. We notice that the return-statement occurs in an if-branch without a corresponding else-branch; therefore, instead of guarding on the result-set variable, we can simply put the remaining code in the else-branch.

Second, we have implemented a generic dead code elimination procedure to deal with redundancy and keep the generated code readable. While dead code elimination is important to remove unnecessarily-introduced guarding variables and branches during the rewriting of the return-statements, it is also used in numerous other places in our tool. As we have stated previously, it is not only our goal to generate code that verifies in Chalice but also code that is readable.

There are quite a few places in our tool where we introduce some additional variables or branches, and often they are needed to handle the general cases but can be omitted in the common cases, like methods that do not return early. Rather than optimizing the generated code at every stage of the translation and trying to be clever when to introduce local variables and branches, we just always emit them and then rely on having the generic dead code elimination to remove the redundancy as a post-processing step.

```
method foo(a: int) returns (res: boolean)
{
        var isResultSet: boolean := false;
    if (a = 0) {
        res := false;
        isResultSet := true;
    } else {
        call bar();
        res := true;
        isResultSet := true;
    }
}
```

Figure 2.30: Simplified version of Figure 2.27

This is useful because it allows many parts of the translation to be greatly simplified, and to postpone the caring about readability to one place.

Note that this approach can alter not only the code introduced by our tool but also code from the original VeriFast program; however, since our dead code elimination is conservative and does not perform any unsound modifications this should never affect correctness. Also, the original code gets modified quite strongly during the translation anyway, and finally, the original code usually does not contain lots of dead code or redundancy.

Our dead code elimination procedure is based on single static assignment (SSA) and performs the following tasks:

- Forward propagation of constants. This is for instance important when dealing with our 'isResSet' variable which first gets initialized to false and then assigned to true; those values can often be forward propagated.
- Constant folding of guard expressions of if-statements and loops. This is important to determine whether the branch (respectively the loop body) is reachable or not. Especially, when 'isResSet' has been constant propagated, the negation in the guarding can be fold.
- Removal of unreachable branches, where the branch-condition got either folded to 'true' or 'false'. In this case, the unreachable branch can be completely removed, and also, if-statements which do not have an else-branch and the condition is known to evaluate to false, can be completely removed. Conversely, if the condition got folded to true, the if statement is superfluous and the body can be pulled out. Notice, how the first three points play together in order to eliminate those simple guards on the *isResSet* guard.

- Removal of local variable writes if the assigned value is never read. Especially, when the written value is a constant it might have been propagated and then the write itself is redundant. Note in particular, that writes to fields are never optimized in any way, to stay conservative.
- Removal of unused local variable declarations. Most of them result from the other optimizations where the reads got replaced by constants and all the writes were removed.

Our dead code elimination procedure currently does not handle some cases which could easily be identified by hand:

- It does not take branch conditions into account when constant folding the guards of nested if-statements. Therefore, the inner branch of the following example cannot be deduced to be unreachable:

```
method bar(a: int) returns (res: boolean)
{
    if (a = 0) {
        if (a ≠ 0) {
            // unreachable
            [...]
        }
    }
}
```

Figure 2.31: Example of the limitations of the dead code elimination

  For the current use-cases this particular optimization never turned out to be crucial and some care would have to be taken to use correctly use the SSA form when there are writes to 'a' inside the outer if-statement. Especially, the constant folding is done by our expression-analyzer described in section 2.2.6, which currently does not understand the SSA annotations.

- The dead code elimination so far does not track any initialization properties; particularly, newly created objects are not known to be non-null. This can lead to some redundant null-checks right after creating a new object.

In our original example of Figure 2.26, the first optimization removes the guarding on the result-set variable (Figure 2.30). The dead code elimination can then remove the writes to, and also the declaration of, the result-set variable. Finally, our example becomes much more concise and therefore easier to read:

```
method foo(a: int) returns (res: boolean)
{
    if (a = 0) {
        res := false;
    }
    else {
        call bar();
        res := true;
    }
}
```

Figure 2.32: The actual translated Chalice code of Figure 2.26

2.5 Impure Expressions

When bridging the gap between the language features of Java and Chalice, expressions with side-effects play a big role. On one side, there is Java, which supports a wide variety of expressions including side-effects, such as assignments, method calls, object creation, and increment operations. On the other hand, there is Chalice where all expressions are pure. Therefore, during the translations all side-effects have to be removed from expressions.

On a high level, our technique is to "pull" the side-effects out from expressions and put the part of the evaluation which has side-effects as separate statements before the expressions. This gives rise to the following two questions: How does the extraction work, and where exactly do we place the additional statements?

When extracting side-effects we can basically distinguish three kinds of expressions:

1. Those which have inherent side effects
2. Those which affect the evaluation of sub-expressions and, therefore, the side-effects of them.
3. Those which just propagate side effects from the sub-expressions

The first kind includes the following types of expressions: assignment, object creation, method invocation, and prefix/postfix increment/decrement. The extraction is mostly straightforward:

Assignment The whole assignment gets placed before the actual expression. The assigned variable can be then used to obtain the value in the expression.

```
foo(x = 3);
↪
x := 3;
foo(x);
```

Figure 2.33: A method-call where the passed argument contains an assignment (on top). On bottom: the assignment was extracted to its own statement in the translated version.

Object creation (constructor invocation) An additional variable is introduced and gets assigned a new object. That variable can be used in lieu of the new-statement in the expression.

Method invocation Similar to object creation, an additional variable is used to store the result of the invocation. In addition, we must obey the evaluation order of the receiver and the arguments, which themselves have to be evaluated before the method invocation itself.

Prefix increment / decrement Prefix operators can be treated like assignments. The new value is then used.

Postfix increment / decrement The old value has to be saved to an additional variable before incrementing, as shown in figure 2.34

```
y = x++;
↪
var tmp := x;
x := x + 1:
y := tmp;
```

Figure 2.34: The handling of a postfix increment

The second kind of expression includes the conditional-expressions, as well as the short-cut and- and or-operator.

Conditional expression The extracted side-effects of the then- and else-branch have to be put into an equivalent if-statement to prevent the evaluation of both branches.

```
z = (x == y) ? foo() : 0;
↪
var tmp: int;
if (x = y) {
        call tmp := foo();
}
// The else-branch does not contain side-effects
// Hence, we do not need to evaluate it beforehand
z := (x = y) ? tmp : 0;
```

Figure 2.35: Evaluation of the branches of a conditional-expression

&&-operator The right hand side should only be evaluated whenever the left hand side did evaluate to 'true'. Hence, the extracted side-effects need to be wrapped into an if-statement guarding on the left hand side, as shown in Figure 2.36

```
y = (x == 3) && foo();
↪
var tmp: bool;
if (x = 3) {
        // only evaluate (call foo) if the LHS is true
        call tmp := foo();
}
y := (x = 3) && tmp;
```

Figure 2.36: Evaluation of an &&-expression

||-operator Similar to the and-operator, an if-statement needs to be introduced.

Finally, there are expressions which neither directly have side-effects nor alter the way subexpressions get evaluated. Nevertheless, care has to be taken to get the evaluation order right. Consider the following example of the binary expression $x + (x = 3)$: one might think that we can just pull the side-effects of the left- and right-hand-side out and place them one after another. However, x by itself does not have any side-effects, yet we need to evaluate it before executing the assignment $x = 3$. Therefore, we need to take a snapshot of x before updating it; hence, we actually need to evaluate both sub-expressions beforehand (of the evaluation of the plus) to preserve evaluation order, as shown in Figure 2.37.

```
y = x + (x = 3);
↪
var old_x := x; // snapshot of LHS
x := 3;         // evaluation of RHS
y := old_x + x; // usage of the expression
```

Figure 2.37: Example of taking a snapshot for binary-expressions

Applying that approach in general, every subexpression needs to get evaluated in its own statement in the right order, before the actual expression can be evaluated. This might introduce a lot of unnecessary variables when the subexpressions do not affect each other as for instance in $y + x$ where both variables technically need to be evaluated outside. Hence, some adhoc optimizations have been performed in our tool to prevent us from introducing too many unneeded temporary variables.

The second question which remains to be answered, is where to put the extracted evaluation. Since we only have to deal with expressions from the Java code and not the

VeriFast annotations, which are pure anyway, we can rely on having every expression embedded in some statement. In addition, statements almost always occur within a block, and otherwise, the statement can be replaced with a block containing just that statement. Hence, when an expression e is embedded in a statement s, we put the extracted evaluation right before s in the same block.

One important exception is the placement of the loop-condition of while-loops. Not only must the evaluation happen before the first iteration, but also before every consecutive iteration. Hence, we have to place the extracted evaluation both before the while-loop and at the end of the loop body:

```
[evaluation of e]
while (e) {
        // loop body
        // ...
        [evaluation of e]
}
```

Figure 2.38: Evaluation of a loop-condition with side-effects

2.6 Constructors

In contrast to Java, Chalice does not have constructors. Whenever you create a new object in Chalice, you just get full permissions on all fields of the new object. Furthermore, the fields of a freshly created object are uninitialized. When translating the Java code we, therefore, have to do the initialization done by the constructor manually.

Our translation generates an 'init'-method for every constructor, or one for the default constructor. We then just call the initialization-method right after creating a new object, which can be done since object creation is a statement and not an expression in Chalice anyway (the issue of using 'new' as an expressions in VeriFast has already been handled in section 2.5). For instance, consider the following constructor of a class:

```
public int x;
public int y = 1;

public LinkedList(int aX)
//@ requires aX >= 0;
//@ ensures x == aX &*& y == 1;
{
    this.x = aX;
}
```

Figure 2.39: Example of a constructor in VeriFast

and the corresponding initialization method in Chalice:

```
var x: int;
var y: int;

method _init(aX: int)
        requires acc(this.*, 100) && aX ≥ 0;
        ensures x = aX && y = 1;
{
        this.x = 0;
        this.y = 1;
        this.x = aX;
}
```

Figure 2.40: Translated constructor of figure 2.39

That initialization method requires access to all fields in the precondition (expressed by $acc(this.*, 100)$), additionally to the original precondition of the constructor. The body first initializes all fields with either the default (as for variable x) of the corresponding type or the value which was given at the field declaration (as for y); then, the body of the constructor follows. Finally, as a postcondition we just use the one of the constructor.

2.7 Lemma Methods

Currently we treat lemma-methods exactly like regular methods and translate them to regular Chalice methods, owing to the fact that no closer counterpart exists. However, this has two pitfalls:

Conceptional: The purpose of lemma-methods is to modify the abstract state; they are not meant to be executable. This distinction is completely lost in the current translation.

Practical: The control-flow of lemma-methods can depend on ghost-variables. If we directly translate such code, Chalice surprisingly still accepts it, although the execution of a (real) method should obviously not depend on the ghost state. There is also a translation from Chalice to C#, and we expect this translation either to fail or produce incorrect code in such a case.

2.8 Abstract Data Types and Inductive Switches

For both abstract data types (ADT) and inductive switches, we have only very limited support. This is on the one hand owing to the fact that they are not that frequent in our VeriFast samples and on the other hand that have no matching counterpart in Chalice and are also not easily emulatable.

The only ADT that is supported is the built-in 'list' of VeriFast. This is the only built-in ADT of VeriFast and by far the most common one that is used. In addition, it is the only one that has some sort of built-in equivalent in Chalice: the sequence type 'seq'.

There is no proper way to emulate other, user-defined, ADTs since the basic intention of them is to abstract over actual classes and data types. Hence, emulating them by using regular classes would simply defeat the purpose of that abstraction. In addition, ADTs can be freely mentioned inside specifications without having to have access permission, since they are immutable, and one does not need to construct them explicitly. If one were to try to replace them with regular classes, the objects would need to be constructed outside of the specification and either be passed to the method or be returned by the method; also, access permissions would need to be carried around. In short, one would probably need to extend Chalice at some point to support ADTs as well; however, this turned out to be outside the scope of this project.

As we have already mentioned, the 'list' ADT of VeriFast can be more or less directly be translated to the 'seq' type of Chalice. Construction of a list then becomes *[]* for *nil* and *[x] ++ [tail]* for $cons(x, tail)$. Reversely extraction of variables of 'cons' can be translated to $l[0]$ for $cons(?x, \_)$ and $l[1..]$ for $cons(\_, ?t)$; both of the expressions replacing their respective bound variables (x and t). Inductive switches on the 'list' can be replaced with an if-else statement testing on the length of the list; the extraction for the cases can then be handled as described before.

```
fixpoint list<int> remove(list<int> l, int x) {
  switch (l) {
    case cons(v,n): return x==v ?
                           n :
                           cons(v, remove(n,x));
    case nil: return nil;
  }
}

function remove(l: seq<int>, x: int): seq<int> {
    |l| > 0 ?
        (x = l[0] ?
            l[1..] :
            [(l[0])] ++ remove(l[1..], x)
        ) :
        []
}
```

Figure 2.41: Example of a function using the built-in 'list' ADT and the translated counterpart

Other inductive switches are currently not supported and there is no real incentive as they would work on the unsupported ADTs.

2.9 Subtyping

Subtyping is commonly considered as one of the core features of object oriented languages. Nevertheless, Chalice currently does not support any form of subtyping even though it is an OOP-like language with classes. An ongoing Master's Thesis [2] aims to address this topic, but is not available at the time of writing this report.

Subtyping has many facets that we would need somehow to translate: on the Java side there are features like interfaces, inheritance, dynamic dispatch, and casting; on the specification side there exist features such as predicate-families in VeriFast to accompany subtyping.

Our tool does not support any form of subtyping and, hence, the translation will fail for all the examples making use of it. Of course, we could try to work around it and apply some sort of clever tricks like we did for other features such as static methods. Some possible ideas we discussed during the project were:

- Remove interfaces that are only implemented by a single class and use that class directly. This obviously comes at the cost of breaking both abstraction and extendibility since one assumes that all the possible implementations are known. However, for the Java side, this would allow some of the examples to be translated.
- Expand interfaces to wrappers that hold the actual instance in a field and then delegate all the invocations. This would allow the retention of some of the intended abstraction and clearly be more general than the first approach.

  However, this approach also relies on knowing all possible implementations, since one will need one field per possible implementation and do case splitting for delegation. The key advantage would be that most of the idioms of subtyping like dynamic dispatch and (explicit) casting would follow quite naturally from that approach.

```
interface Counter {
    int get();
        //@ requires valid(?value);
        //@ ensures valid(value) &*& result == value;
}

class C1 implements Counter {...}
class C2 implements Counter {...}

class Counter {
    instC1: C1
    instC2: C2

    method get() returns (res: int)
        requires valid;
        ensures valid && old(getValue()) = getValue() &&
                res = getValue();
    {
        if (instC1 ≠ null) {
            call res := instC1.get();
        }
        else if (instC2 ≠ null) {
            call res := instC2.get();
        }
    }
}
```

Figure 2.42: Excerpt from an interface-example of VeriFast (on top) and the proposed translation with explicit case-splitting (on bottom)

- For direct subclasses one could either introduce an additional interface or try to do the trick described above directly in the superclass. In any case, having dealt with interfaces the other features such as inheritance would not cause any real additional troubles.

So why did we not implement the second approach? First, it only describes how to deal with the Java part; how to deal with predicate families (section 2.9) is even less clear. Second, while subtyping is common, there are still many interesting examples which do not require it in contrast, e.g., to predicates and static methods which are used all over the place. Therefore, we did not try to deal with subtyping, and require the user to manually apply such tricks on the original source code by hand. Finally, we hope that eventually some form of subtyping will be implemented in Chalice itself; however, this is clearly out of scope for our project.

2.10 Loops

In Java, there exist three kinds of loops: while-loops, for-loops, and do-while-loops. On the contrary, only while-loops are supported in Chalice. Yet, transforming a for-loop to a while-loop is straight-forward: we just put the initialization before the loop and move the update to the end of the loop body. However, one needs to ensure that the definition scope loop variable does not escape the loop; otherwise, having two for-loops both using i as the loop variable would cause a name collision. To our rescue, Chalice allows the use of extra blocks to limit the scope of variables; therefore, we can put the transformed for-loop into its own block.

```
for (int i = 0; i < n; i = i+1) {
    [body]
}
↪
{
    int i = 0;
    while (i < n) {
        [body]
        i = i+1;
    }
}
```

Figure 2.43: Example of a for-loop and its equivalent while-loop. Note the additional brackets around the while-loop and the initialization to prevent i from escaping.

Similarly a do-while-statement can be translated to a while-statement by introducing an additional boolean guard that encodes whether the first iteration has been completed, like shown in figure 2.44. The short-cut semantics of the ||-operator prevent the original condition *cnd* from being evaluated before the first iteration.

```
do { [...] } while (cnd);
↪
{
   var first := true;
   while (first || cnd) { [...]; first = false; };
}
```

Figure 2.44: The translation of a do-while loop

2.11 Overloading

Translating from Java to Chalice includes the removal of any overloading, since Chalice requires names to be unique in a given scope. This does not only affect methods, but also fields, predicates, and functions which all have to have different names. However, Chalice is just as expressive, even though it does not support overloading, since overloading is just pure syntactic convenience and is completely resolved at compile time. Hence, we can simply rename overloaded names: we add an unique number to every overloaded member. Of course we also need to adapt all the callsites; however, since our resolution completely binds them this becomes a trivial task.

```
void add(int x) { }
void add(Counter x) { }
↪
method add_1(x: int) { }
method add_2(x: Counter) { }
```

Figure 2.45: Elimination of overloading by renaming the conflicting methods

2.12 Exceptions

There is some basic support for exceptions in VeriFast; notably, VeriFast supports a special kind of postcondition for checked-exceptions[1], which applies in the case the method left by throwing an exception.

```
void throwsMyException(boolean thrw)
    throws MyException /*@ ensures thrw == true; @*/
    //@ requires true;
    //@ ensures thrw == false;
{
    if(thrw) {
        throw new MyException();
    }
}
```

Figure 2.46: A VeriFast example with an exception-contract

[1] http://docs.oracle.com/javase/specs/jls/se7/html/jls-11.html#jls-11.2.3

Chalice, currently, does not support any kind of exceptions or exceptional control-flow. As a consequence our tool does not support exceptions, as emulating this non-structured control-flow in a reasonable manner (producing readable code) seems almost impossible, although theoretically possible. In addition, in the VeriFast examples only the 'Exception' test-case uses exceptions.

2.13 Permissions

In the assertion language, one of the challenging differences is that Chalice works with percentages of permissions, whereas VeriFast uses real fractional permissions. While every percentage can obviously be expressed as an fraction, the reverse is not true: even simple fractions like $\frac{1}{3}$ do not have a counterpart in Chalice.

When motivating our approach, one has to be aware that fractional permissions are rarely used in the VeriFast examples; the only occurrences are in the various 'spouse'-examples which use the extremely simple fraction $\frac{1}{2}$. We, therefore, settled for a simple approach and leave a more ambitious translation up to further improvement of the tool. Such an extended transition could, for instance, take advantage of the fact that reals have recently been added to Chalice, although, they are not used by default.

As a consequence, our tool only supports fractions which can be expressed as a percentage, otherwise the translation fails. We first support some basic arithmetic on fractions, including addition, subtraction, multiplication, and division. Then, we multiply the resulting fraction by 100 and try to cancel it in order to obtain an integer number. Furthermore, that approach also prevents us from supporting fractions which are general expressions, i.e. refer the variables, since there is no way to ensure $100 * e$ is an integer for a general expression e.

What we do support, on the other hand, are "unknown" fractions of the form $[\_]\,\mathrm{f} \mapsto \mathrm{v}$ from VeriFast. That concept of having an arbitrary but non-zero permission translates nicely to the read-star (written as rd$*$(f)) assertions of Chalice. This kind of permission turned out to be more frequently used than concrete fractions, too.

Several ideas for more sophisticated approaches have been discussed during the project; however, they were left for possible future extension. Especially, it can be noticed that the exact amount of the fractions are not of great importance; in the end it just matters whether we have no, read, or write permission. Hence, instead of splitting write-access into three equal fractions of $\frac{1}{3}$ one could also split them into one fraction of $\frac{1}{2}$ and two of $\frac{1}{4}$. One could therefore try to rewrite all fractions to different ones are expressible as percentage. The difficult part is that once those fractions are joined again to form $\frac{1}{1}$, that has also to be the case in the rewritten form; hence, some careful analysis of the whole program would have to be done to figure out the constraints under which we can rewrite the fractions.

As an other alternative, the new permission model of Chalice [3] (read, read-star, and write), which was particularly motivated by the fact that concrete fractions are often irrelevant, could be used. However, the same restrictions of splitting up and later rejoining also apply.

2.14 Assertions

The translation from VeriFast's assertions to these of Chalice is (with the exception of predicates) straight forward. It has been shown [7] that the semantics are preserved when we simply replace the points-to-assertion $[p]o.f \mapsto v$ with access to the field followed by an equality assertion **acc**(o.f, p) && f =v.

```
public void foo()
    //@ requires this.head |-> 0;
    //@ ensures true;
{ [...] }
↪
method foo()
    requires acc(this.head, 100) && this.head = 0;
    ensures true;
{ [...] }
```

Figure 2.47: Example of a contract containing a points-to-assertions and its translation

Furthermore, there is a difference in the semantics of preconditions between VeriFast and Chalice. Namely, when we have a points-to-assertion $o.f \mapsto v$ in a precondition in VeriFast it implicitly requires the target object o to be non-null. In Chalice that requirement must be stated explicitly, otherwise the verification will complain about the precondition itself since it accesses a field on a potentially null reference. Therefore, we must add $o \neq null$ in front of every points-to-assertion in preconditions.

3 Evaluation

When performing the evaluation of our tool, we wanted to answer a number of questions for each test-case in order to evaluate the tool itself, the feasibility of the translation, and in particular how well Chalice does on verifying the translated programs:

- Can the program by translated?
    1. Why could it not automatically be translated?
    2. Could it (theoretically) be translated by hand?
    3. If it is impossible by hand, which features of VeriFast or Java prevent the translation altogether?
    4. If it can be done by hand but not by our tool, how would one need to improve our tool?
    5. How could the test-case be modified to allow either an automatic or manual translation?
- Can the translated test-case be verified?
    1. If not, did the translation introduce the problems by not conserving some properties or facts or even introducing new flaws?
    2. How can the translated Chalice code be enhanced to allow the translation?
    3. Is it an unexpected difficulty for Chalice or was it obvious that Chalice could not verify such a program?

To conduct our evaluation we mainly used the Java test-cases from VeriFast. We restricted ourself to the single-file test-cases since our tool currently only handles those and because it is very unlikely for a bigger test-case to not contain any unsupported features such as inheritance. In addition, we have written our own (simple) linked-list example and also a segmented linked-list.

3.1 Initial evaluation with the unmodified test-cases

We first conducted an evaluation without modifying the test-cases; secondly, we modified some of the test-cases and re-evaluated them. The results of an initial run are summarised in Table 3.1, where the underlying problems are denoted as a foot-note.

Table 3.1: Evaluation with unmodified test-samples

Testcase	**T**	**V**	**Error**	**Problem**
AbstractClasses	failed		failure in final translation	1
Account	ok	ok		
AmortizedQueue	ok	failed		*
ArrayList	failed		parser error	8
ArraysManual	failed		parser error	5
Automation	failed		parser error	6
Bag	failed		failure in typechecker	14
Compounds-Assignment	ok	ok [1]		
Comprehensions	failed		parser error	6
ConstantExpression	failed		failure in final translation	9
Constants	failed		failure in typechecker	2
Contrib	failed		failure in parser	4
Counter	ok	ok		
DefaultCtor	failed		failure in final translation	10, 1
Division	failed		failure in final translation	15
DoWhile	failed		error in typechecker	11
Downcast	failed		parser error	7
Exceptions	failed		parser error	12
FieldInitializers	failed		failure in final translation	3
InstanceOf	failed		error in typechecker	2
InterfaceLemmas	failed		parser error	6
Iterator	failed		parser error	6
map	failed		parser error	7
MonitorExample	failed		parser error	4
NetedExprTest	ok	ok		
Recell	failed		failure in final translation	1
Spouse	ok	ok		
Spouse2	ok	ok[1]		
SpouseFinal	ok	ok[1]		
Stack	failed		failure in simplifications	*
StaticFields	failed		parser error	7, 9
SuperCalls	failed		parser error	7, 1
SuperConstructorCall	failed		failure in final translation	1
ThreadRun	failed		parser error	13, 6
Tree	failed		failure in simplification	*
LinkedList	ok	ok		*
LinkedListSeg	failed		Cannot handle predicate	*

[1] inheritance [2] interfaces [3] super-constructor invoc. [4] predicate constructors
[5] higher-order predicate [6] abstract predicate [7] predicate families
[8] auto lemmas [9] static fields [10] use of short / byte
[11] external libraries [12] exceptions [13] concurrency [14] built in pair not supported
[15] unsupported operators (division, bitwise-and, ...) [*] discussed separately

3.2 Discussion of selected test-cases

For some of the test-cases we could modify the original VeriFast example to make the translation work nevertheless, or modify the generated Chalice program to let the verification pass. We will now discuss those test-cases in detail and discuss why the changes were necessary. In addition, we will discuss why some test-cases failed and cannot be fixed, and discuss some that were verified but are nonetheless worth discussing.

AmortizedQueue For the 'AmortizedQueue' our translation works and produces a syntactically correct Chalice program; however, the verification fails. When inspecting the failure, it became clear that the failure resulted from our implementation of the (pseudo) static functions in Chalice. We do get a duplication of the list-reversal function and there is a method in the class 'AmortizedQueue' looking like this:

```
method _init_1(front: LinkedList, rear: LinkedList)
    requires [...];
    ensures [...]this.reverse_int(old(rear.getVs()))
{
    [...]
    call f := rear.reverse();
    [...]
}
```

Figure 3.1: The problem of duplicated functions

Notably, the 'reverse_int' in the postcondition refers to the version from the 'AmortizedQueue' class; however the actual reversing is done by calling the according method on the linked-list which expresses its postcondition in terms of the duplicated reversal-function there. As a consequence, Chalice would need to prove the equivalence between those reversal-functions which is infeasible due to their recursive nature.

Nevertheless, hand-modification can easily be done to allow the program to verify (except some termination checks) by replacing the postcondition with rear. reverse_int (old(rear.getVs()) and completely deleting the version in the 'AmortizedQueue' class.

Stack The 'Stack' test-case originally failed in the translation, since it tried to use a discarded predicate argument as the new receiver. However, the test-case can easily modified by hand to indicate the desired receiver of the predicate as shown in Figure 3.2, in which the modification is shown in a comment.

```
/*@
predicate nodes(Node n0; int count) =
    n0 == null ?
        count == 0
    :
        n0.value |-> _ &*& n0.next |-> ?next &*&
        nodes(next, ?ncount) &*&
        count == 1 + ncount;

@*/

[...]

//@ predicate valid(int count) = head |-> ?h &*& nodes(h, count);

int pop()
    //@ requires valid(?count) &*& 0 < count;
    //@ ensures valid(count - 1);
{
    //@ open valid(count);
    //@ open nodes(head, _); // instead of: nodes(_, _)
    int result = head.value;
    head = head.next;
    //@ close valid(count - 1);
    return result;
}
```

Figure 3.2: Excerpt of the hand-modified Stack test-case indicating the modified line

By inspecting the 'valid' predicate that was unfolded, it becomes immediately clear that only 'head' can be the first argument of the 'nodes' predicate. However, doing such an analysis automatically is out of scope for our current tool as it must reason that the only way to get a 'nodes' predicate at that location of the program is from the unfolding before. Note that the first parameter of the 'nodes'-predicate is actually an in-parameter and discarding it makes the predicate instance non-unique. Hence, the discarding in such a situation is only allowed because VeriFast implements some heuristic, which picks an arbitrary instance from the matching ones.

The resulting program, obtained by making the receiver explicit, translates and also verifies without any further problem.

LinkedList For the 'LinkedList' example we wrote at the very beginning of our project, we just had to replace our own inductive list-type by the built-in one of VeriFast. Our translation can currently only handle the built-in list ADT.

After circumventing a known parser bug of Chalice (function-calls not allowed directly inside an 'acc'-expression[2]), the program verifies with two errors. Yet, those errors result from the termination check of functions, which in Chalice currently only takes the size of the heap into account. However, since the static functions we contributed to Chalice usually don't rely on the heap any longer at all, but rather depend on the length of an ADT list as a variant, that termination criterion is useless in these cases. The termination checks can be turned off in Chalice and then the verification succeeds as expected.

LinkedListSeg Here we had to manually rewrite the segmented predicate to a version where the empty segment has the canonical representation "linkedListSeg(nil, nil)" in order for our translation to work. Otherwise, an arbitrary number of such predicates could be closed on the same receiver (the segment start) and one could not store the segment end in ghost-state; obviously there is also no functional dependency between the segment start and end, in contrast to the non-segmented version where the start and the heap determines the whole list. We have already discussed this issue in section 2.2 as well as why we cannot do such a transformation automatically.

With the rewritten predicate, the test-case can both be translated and verified, except for the usual termination check problems. In summary, we have written a segmented linked list in VeriFast that can be fully automatically translated and that verifies in Chalice, while it was unclear whether a segmented linked list can even be encoded in Chalice at the beginning of our project.

Tree In this test-case our tool tries to inline a (global) static, recursive lemma method which operates on an inductive data-type (tree). Lemma methods that act on ADTs are intrinsically recursive and, most of the time, do not take any reference parameters. However, since the ADT itself is unsupported, this class of static functions (that we cannot handle), is not of great relevance. If Chalice, however, gets extended to include ADTs then also one needs to come up with a way of dealing with such static methods.

Spouse The three test-cases 'Spouse', 'Spouse2', and 'SpouseFinal', which verify with no problems, provide some further evidence that (apart from inheritance) our tool is able to cope with non-trivial programs. While the first and the last version use a parameterized predicate with fractional permissions, the 'Spouse2' even uses multiple nested, parametrized predicates (Figure 3.3) to express the validity of a marriage. The translated

[2] `http://boogie.codeplex.com/workitem/10226`

version (Figure 3.4) not only reflects that nesting but also gets the access permissions in the getter-functions right, by invoking getValid0Spouse from getValidSpouse.

```
protected predicate ticket(Person spouse) =
        [1/2]this.spouse |-> spouse &*&
        spouse != null;

protected predicate valid0(Person spouse) =
        [1/2]this.spouse |-> spouse &*&
        spouse == null ?
                [1/2]this.spouse |-> null
                : emp;

public predicate valid(Person spouse) =
        this.valid0(spouse) &*&
        spouse != null ?
                spouse.ticket(this)
                : emp;
```

Figure 3.3: The predicates of 'SpouseFinal' (VeriFast)

3.3 Discussion of the results

On the subject of the feasibility of the translation, it became clear that a large portion of the test-cases are not translatable by our tool; however, for most of them the reason is simply the use of subtyping, inheritance, or a related specification feature such as predicate families and constructors. We somewhat hoped in the beginning that the lack of subtyping in Chalice would not be so big a deal, since most of the test-cases are rather small and such small samples have the tendency to only consist of a single class. However, we were proven wrong and the results indicate that most of VeriFast's Java examples do use some sort of subtyping. Unfortunately, subtyping is not only a problem for our automatic translation but, since Chalice lacks subtyping at all, it is also unclear how one could translate those examples by hand without completely changing their intent.

For the remaining few examples which our tool could not handle, no particular culprit could be identified. Almost every one of them failed for a different unsupported feature such as ADTs, external interfaces, and arrays. In theory one could probably try to handle those cases in our tool with some effort; however, since they only affect a single example and at least some of them contain inheritance as well it was not considered worth the effort.

```
predicate ticket {
    acc(this.spouse, 50) &&
    this.spouse ≠ null
}

predicate valid0 {
    acc(this.spouse, 50) &&
    (this.spouse = null ? acc(this.spouse, 50) : true)
}

predicate valid {
    acc(this.valid0, 100) &&
    (this.getValid0Spouse() ≠ null ?
        this.getValid0Spouse().ticket &&
        this.getValid0Spouse().getTicketSpouse() = this
        : true
    )
}

function getTicketSpouse(): Person
  requires acc(ticket, 100); {
    unfolding acc(ticket, 100) in this.spouse
}

function getValid0Spouse(): Person
  requires acc(valid0, 100); {
    unfolding acc(valid0, 100) in this.spouse
}

function getValidSpouse(): Person
  requires acc(valid, 100); {
    unfolding acc(valid, 100) in this.getValid0Spouse()
}
```

Figure 3.4: The translated predicates of 'SpouseFinal'

More interesting is to note that except for our own two linked-lists, not a single example failed due to an untranslatable predicate. While our treatment of predicates is highly incomplete and can only handle certain kinds of predicates, it seems to be complete enough to handle the common cases. The same applies for our handling of static methods and fixpoint-functions as well: whilst incomplete, we only saw a single example where they caused an abort in the translation, and there it is closely related to the use of user-defined ADTs.

Considering the evaluation of Chalice, it is worthwhile to note how well Chalice does on the translated examples. Several test-cases verified without a single modification, once translated. Also, this is not limited to toy-examples but also highly non-trivial test-cases such as the 'Spouse' ones which contain for instance multiple nested predicates with parameters. The 'AmortizedQueue' example was the only one that our tool was able translate but some additional tweaks were required to get it verified. It was a bit surprising that Chalice ran for more than one hour on the original example whereas mostly it is done within seconds on test-cases of similar size. Overall, we noticed that Chalice verified all the working examples quite fast; however, back when the translation of the fixpoint-functions was broken, also other examples (LinkedList, LinkedListSeg, Stack) were falsified rather slowly by Chalice.

Finally, our own segmented linked-list turned out to be the test-case which provided the most insight. On the one hand, it contained the only predicate which our tool could not handle and, nevertheless, is translatable by hand; we have already discussed in Section 2.2 why doing such a translation automatically would be challenging. On the other hand, it was one of the examples where the need for static functions became very clear originally, and once they were added to Chalice, this allowed the translation.

Acknowledgements

I would like to thank my supervisor Dr. Alexander J. Summers for all his helpful support through-out the whole bachelor thesis. Furthermore, I would like to thank Professor Peter Müller for giving me the opportunity to gain exciting insight into the research area of program verification.

Finally, I would like to thank Professor Bart Jacobs for providing the VeriFast source-code.

Bibliography

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

[2] Andres Bühlmann. Supporting subclassing and traits in syxc. Master's thesis, ETH Zürich, 2013.

[3] 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. To appear.

[4] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 304–311, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] K. Rustan Leino, Peter Müller, and Jan Smans. Foundations of security analysis and design V. chapter Verification of Concurrent Programs with Chalice, pages 195–222. Springer-Verlag, Berlin, Heidelberg, 2009.

[6] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, UK, 2001. Springer-Verlag.

[7] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, pages 439–458, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

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

A Unsupported features and limitations

A.1 Parser

Unsupported Java features

- Class literals (including void.class)
- Generic wildcards
- Generic bounds
- Native, transient, and stricfp modifier
- Anonymous inner classes
- Super constructor invocation
- Resources try-blocks introduced in Java SE 7.0

Unsupported VeriFast features

- Predicate families
- Predicate constructors
- Higher order predicates
- Auto lemmas
- Exception contracts

A.2 Resolution

- For inner classes, the resolution does not take members defined in enclosing classes into account, unless they are qualified by a 'this' expression explicitly referring to the enclosing class (e.g., C.this.f where 'C' denotes the name of the outer class).
- Resolution does not support referring to members of the generic type parameter (e.g., x.f where x is of type T inside MyList$\langle T \rangle$)
- Referring to types defined in external libraries
- Static field access and static method invocation on expressions

- Constructors with generic parameters
- Inductive switch statements on a inductive types except the built in list

A.3 Simplifications

Fractional permissions Only fractions expressible as an integer percentage are supported. Other fractions like $\frac{1}{3}$ will cause the simplification to fail.

In addition the expression denoting the fraction must be a compile time constant; hence, may especially not refer to any variables. Otherwise, the fractional value could not be safely converted into a percentage. The constant must be composed only from integers as well as addition, subtraction, multiplication, and division; other operations like shifting are not supported.

Control flow

- Exceptional control flow (exceptions) are unsupported
- break statements and continue statements are unsupported

Static methods Static methods are only supported if at least one of the following criteria holds:

- There exists a formal parameter of a user defined reference-type which is guaranteed by the precondition to be non-null. Then this parameter is made the receiver of the instance method.
- There exists a formal parameter, such that when it is null the method body is a no-op (for methods without return type) or a constant expression. This allows this parameter to be made the receiver and the method to be simply not called whenever that parameter is null.
- The method is not recursive, to allow inlining.

Predicates Static predicates defined in the global scope are only supported when one of the following conditions hold:

- There exists a parameter of a user defined reference-type which is guaranteed to be non-null by the predicate body.
- There exists a parameter of a user defined type for which, when null, the following conditions hold, such that the predicate can be erased:
  - The predicate does not hold any permissions

  - All other parameters become constant expressions, hence there exists a functional dependency from that parameter to all others.
  - That parameter is only referred to as an out parameter and never bound ($p(?x)$) or used in an existential form like $p(\_)$

Predicate with parameters (except the one used as the receiver of static predicates) are only supported when one of the following conditions hold:

- There exists a functional dependency from the other parameters and the heap; hence, the parameter can be expressed as a pure function. In addition, for multiple parameters those functions may not contain recursion among each other.
- The predicate holds full permission to at least one field. Note that this field must be statically determined and an existential of the form $c?acc(f1) : acc(f2)$ is not sufficient.

Inductive switches Inductive switches (expressions, assertions, and statements) are only supported on the built in inductive list.

A.4 Final translation

Some features unsupported by the tool are not currently handled by the intermediate simplifications and, therefore, will cause the final translation to fail.

- Any kind of inheritance such as inheriting from a class, implementing or interface, and calling a super constructor.
- Declaring an enum or using one
- Declaring an interface
- Declaring a custom Java annotation class
- The float, double, and char data types
- Shift operators and bitwise operators (negation, xor)
- Static fields: both declaration and access
- Any kind of generics except for generic fixpoint methods
- All flavors of exceptions: try-catch blocks, the throw statement, and throws declarations
- Synchronized blocks
- Initializer blocks; both static and non-static

- Arrays
- Boxed types
- Java switches
- Inductive data types except for the built in list
- Auto lemma methods