An automatic encoding from VeriFast Predicates into Implicit Dynamic Frames

Daniel Jost and Alexander J. Summers

ETH Zurich, Switzerland
`dajost@ethz.ch, alexander.summers@inf.ethz.ch`

Abstract. VeriFast is a symbolic-execution-based verifier, based on separation logic specifications. Chalice is a verifier based on verification condition generation, which employs specifications in implicit dynamic frames. Recently, theoretical work has shown how the cores of these two verification logics can be formally related. However, the mechanisms for abstraction in the two tools are not obviously comparable; VeriFast employs parameterised recursive predicates in specifications, while Chalice employs recursive predicates without parameters, along with heap-dependent abstraction functions.
In this paper, we show how to relate a subset of VeriFast, including many common uses of separation logic recursive predicates, to the implicit dynamic frames approach. In particular, we present a prototype tool which can translate a class of VeriFast examples into Chalice examples. Our tool performs several semantic analyses of predicate definitions, and determines which of a selection of novel techniques can be applied to infer appropriate predicate and function definitions, as well as corresponding code instrumentation in a generated program. The tool is automatic, and produces programs which can themselves be directly handled by the automatic Boogie/Z3-based Chalice verifier.

1 Introduction

Separation logic [3, 8] is a well-established approach for the verification of heap-based imperative programs; many verifiers have been built using separation logic as their specification language. VeriFast [5, 4] is a mature verification tool for C and Java programs, which handles separation logic specifications by internally maintaining a representation of the current program state (symbolic execution), while passing queries off to an SMT solver about arithmetical problems and other theories. The key primitive features of separation logic are the *points-to assertions* $x.f \mapsto v$, which provide the only means of dereferencing heap locations in assertions, and the *separating conjunction* $*$, whose semantics can be used to divide ownership of heap locations between assertions. The ability to specify unbounded heap structures is provided by recursive *abstract predicates* [9].

Implicit dynamic frames [12] is a more-recently-introduced specification logic, which is designed to facilitate implementations based not only on symbolic execution but also on verification condition generation (i.e., encoding the entire verification problem, including heap information, to an SMT solver). It separates the notion of having permission to access a heap location from the means of actually referring to the location's value. The key primitives here are *accessibility predicates* $\mathbf{acc}(x.f)$, which represent permission to access a heap location, and a conjunction (also written $*$ in this paper) which acts multiplicatively on accessibility predicates (i.e., sums the permissions from the two conjuncts), while not enforcing a strict separation between the heap locations actually dereferenced in expressions. Instead, a concept of *self-framing* assertions is imposed on those assertions used in pre/post-conditions etc., which essentially requires that the assertion only reads from heap locations for which it also requires permission via accessibility predicates. For example, $x.f = 5$ is an implicit dynamic frames assertion, but is not self-framing, while $\mathbf{acc}(x.f) * x.f = 5$ is.

Chalice [6, 7] is a verifier which handles a small object-oriented language (with many concurrency-related primitives) annotated with implicit dynamic frames specifications. It works by verification condition generation; as such, certain design decisions in the language have been made in order to facilitate the encoding to SMT. In particular, although recursively-defined predicates are available in the specification logic, in contrast to VeriFast (and most similar tools), such predicates cannot take parameters (other than the implicit this receiver). Compared with VeriFast predicates, Chalice predicates by themselves are therefore significantly less expressive. However, Chalice specifications *can* include (parameterised) recursive *functions*, whose evaluations can depend on the heap, in contrast to separation logic based tools. Thus, the mechanisms in the two tools for handling recursion in specifications are not directly comparable.

It has been recently shown that separation logic and implicit dynamic frames can be formally related, and that it is possible to encode from separation logic specifications into equivalent specifications in implicit dynamic frames [10, 11]. Using the relationship defined, the IDF assertion $\mathbf{acc}(x.f) * x.f = 5$ is shown to be equivalent to the separation logic assertion $x.f \mapsto 5$; indeed, is it shown that a large fragment of separation logic can be encoded into IDF. This hints at the possibility of encoding programs annotated for, say, VeriFast, into programs annotated instead for Chalice. However, the cited work only applied to the "core" fragments of the two logics; in particular, recursive predicates/functions were not treated in those papers. Since almost all interesting separation logic examples employ predicates in some form or other, this limitation is a serious obstacle to relating the two approaches in practice.

In this paper, we tackle the problem of making this relationship practical. In particular, we present a novel technique for translating VeriFast programs which include parameterised predicate definitions, into Chalice programs (which cannot). Our work helps with understanding the two approaches and their relationships/differences, and potentially provides a platform for future comparative studies on issues such as performance and annotation overhead.

Our approach involves the introduction of heap-dependent functions and ghost state/annotations, and relies crucially on a custom-made assertion analyser, which is used to extract simple semantic information about predicate definitions, without the need to invoke a background prover. While our techniques cannot handle all possible predicate definitions, they are fully automatic, and produce code which can be handled by the Chalice tool without modifications. Our preliminary experiments indicate that non-trivial examples can be handled by our techniques, and we have many ideas for extending their applicability. To our knowledge, this also presents the first method for verifying separation-logic-annotated code solely via verification condition generation to a first-order SMT solver (Z3 [2]). Our approach is implemented, and available to download [1].

2 Background

In this section, we give a swift introduction to the important aspects of VeriFast and Chalice. For more details, we refer the reader to the papers [5, 4, 6, 7]

2.1 VeriFast Predicates, In and Out Parameters

VeriFast source files can declare *predicate definitions*; a predicate has a name, a sequence of formal parameters, and a body, which is a VeriFast assertion. Predicate definitions may occur outside of class definitions (*static predicates*), or inside a class definition (*instance predicates*), in which case they also have an implicit this parameter. For example, an instance predicate describing (non-empty) linked lists, can be defined as follows:

```
predicate linkedlist(list<int> elems) =
  this.value |-> ?v &*& this.next |-> ?n &*&
  (n == null ? elems = cons(v,nil) :
    n.linkedlist(?rest) &*& elems = cons(v,rest))
```

The &*& syntax denotes the separating conjunction ($*$) of separation logic. The ?v syntax indicates a binding of a (logical) variable to a value (which must be uniquely determined by the context); occurrences of the same variable name afterwards refer to this value. The same syntax can be used with predicate instances in, e.g., method specifications:

```
void add(int x)
//@ requires this.linkedList(?xs);
//@ ensures this.linkedList(cons(x,xs));
{ ... }
```

When handling a call to such a method, the verifier matches the variable xs with the actual parameter to the currently-held predicate instance. Such a matching is only guaranteed to be deterministic because the elems parameter is uniquely determined by the predicate body, in any given state. Such a parameter is called an *out parameter* of the predicate, in VeriFast. Conversely, some predicate parameters are used to *determine* the meaning of the predicate body; for example, the parameter end in the famous list segment predicate (a static predicate, here):

```
predicate lseg(LinkedList start, LinkedList end ;) =
  ((start == null || start == end) ? true :
     (start.value |-> _ &*& start.next |-> ?n &*& lseg(n,end)));
}
```

The _ syntax here, represents a wildcard value - essentially, the particular value is anonymously existentially quantified. In this predicate, both start and end parameters must be known before the meaning of the predicate body can be determined. VeriFast calls these *in parameters*. For in parameters, it is not possible to use the ? binding (as in the add declaration above); the values of the parameters are not determined by holding a predicate instance.

In both Verifast and Chalice, ghost unfold and fold statements are used to direct the verifier to replace a predicate instance with its defined body, and vice versa. For example, an instance of the above predicate could be obtained via a VeriFast source statement fold lseg(null,null). When a predicate instance is held, the permissions (points-to assertions) and other constraints given by its definition are not directly available to the verifier; an unfold statement makes them available. This guidance tames the problem of reasoning statically about unbounded recursive definitions; and isorecursive semantics is used [13].

2.2 Chalice Predicates and Functions

Chalice allows a restricted form of predicate definitions, compared with VeriFast. Predicates can only be instance predicates, and cannot take parameters (other than the implicit this receiver). Predicate definitions can still be recursive: for example, the following predicate definition includes the same permissions as the analogous VeriFast example in the previous subsection (the analogous connective to separating conjunction is written &&, in the Chalice tool):

```
predicate linkedlist {
  acc(this.value) && acc(this.next) &&
  (this.next != null ==> this.next.linkedlist)
}
```

The reason for the above restrictions is to simplify the bookkeeping of permissions held by the current thread, for the verification condition generation. Nonetheless, Chalice includes an additional mechanism for abstraction/recursion: the ability to define heap-dependent *functions*. These play a role analogous with pure methods, as often used in contract languages; they can be used to abstract over values represented by the underlying heap data structure. A Chalice function definition includes a *pre-condition*, which must require permissions to (at least) the heap locations on which the function's evaluation depends. Function invocations in assertions do not themselves represent these permissions, but must occur within an assertion in which the permissions are required. For example, the following declaration defines a function which extracts the elements from a linked list structure:

```
function elems() : list<int>
  requires this.linkedlist
  {
    unfolding this.linkedlist in
    (next == null ? [value] : [value] ++ next.elems())
  }
```

The unfolding expression in Chalice permits the definition of expressions which access heap locations whose permissions are currently folded inside a predicate instance; they do not affect the expression's value, but help the verifier to check that appropriate permissions are held. The notion of *self-framing* assertions is extended to check function pre-conditions. For example, this.elems() == [4] is not a self-framing assertion (it does not contain sufficient permissions to satisfy the function's pre-condition), but this.linkedlist && this.elems() == [4] is.

2.3 Running Example

In this paper, we will use as a running example an adapted list segment predicate, in which the list elements are also exposed as a predicate parameter. Our predicate is not quite analogous to the typical lseg; we only model non-empty list segments, with this definition. Our tool can actually handle a more general definition (in which non-empty list segments can also be represented), but we explain the relevant limitations (and how we plan to lift them) in Section 6. The VeriFast definition for our running example is:

```
predicate listSeg(List start, List end, list<int> elems) =
  start != null &*& start.value |-> ?x &*& start.next |-> ?n &*&
  (n != end ? listSeg(n, end, ?nextElems) &*&
              elems == cons(x, nextElems)
            : elems == cons(x, nil));
```

3 Approach

We base our approach around two main ideas: replacing out parameters with abstraction functions, and replacing in parameters with ghost fields; these are detailed in the next two subsections. Note that we do not stick to the VeriFast notions of in/out parameters, but instead try to infer that as many parameters as possible can be treated as out parameters. In the following, we will first outline those two main ideas in detail. Second, we will show how those abstract ideas are used when translating VeriFast predicates (and programs) to Chalice; for concreteness, we show how they apply to our running example (from Section 2.3). Finally, we will motivate the analysis presented in Section 4.

3.1 From Out Parameters to Abstraction Functions

The observation that out parameters can be determined by the underlying heap (along with the in parameters of a predicate definition) led us to an encoding in which such out parameters can be replaced by abstraction functions. The constraints determining the value of the predicate parameter can be re-encoded as a function which computes the value itself. Uses of the predicate parameter can, in general, then be replaced by invocations of the function. For example, the elems parameter of our linkedlist predicate can be replaced by an elems() function, providing the same abstraction of the underlying data structure.

Abstraction functions introduced in this way take the (translated version of the) original predicate as a pre-condition; this provides the appropriate permissions to the heap locations on which the function's evaluation depends. An instance of the original predicate can then be replaced by an occurrence of the new predicate, conjoined with a fact relating the abstraction function's value to the original parameter value. For example an instance this.linkedlist(l) of the parameterised list predicate, is replaced by this.linkedlist * this.elems()=l.

Where the original predicate is recursive, the body of the predicate will usually relate the parameters of the original and recursive predicate instance via some constraint; this results in a recursive definition of the extracted abstraction function. For example, in the body of the parameterised predicate linkedlist, we find that, if this.next = null holds, then elems=nil is required, while if this.next != null then we have that elems = v:rest is required, where rest is the corresponding parameter of the *recursive* linkedlist predicate instance. This gives rise to a natural function definition, as shown at the end of Section 3.4, in which rest corresponds to a recursive call to the function.

3.2 From In Parameters to Ghost State

While out parameters can be naturally handled as abstraction functions, it is clear that the same trick cannot be applied to all predicate parameters. In particular, if the value of a parameter cannot be uniquely determined from the predicate body, but is instead used to *decide* the meaning of the predicate body (for example, the end parameter of the lseg predicate), then it must necessarily be provided for each predicate instance. We handle this situation by introducing additional *ghost fields* to represent the values of the in parameters of a currently-held predicate instance. In particular, a *fold* of the original parameterised predicate definition is handled by instead first writing to the ghost field(s) (with the values that were originally provided for the in parameters), and then folding the translated predicate definition. When the resulting predicate instance is unfolded, the ghost fields can be used in place of any occurrences of the original parameters. For instance, when folding a linked list segment predicate taking the start as receiver and the end as parameter, fold(start.lseg(end)) gets replaced by start.end = end; fold(start.lseg).

This handling of in parameters using ghost state comes with a clear limitation: since a (ghost) field can only have a single value at any one time, it is only possible for us to encode uses of predicates for which it is never required to hold multiple instances of the same predicate, for the same receiver but for different values of the other parameters. This problematic situation could arise in two (related) ways. Firstly, it could really be that the in parameter is used to select between two different views of the same data structure. For example, while it is not possible to hold lseg(this,x) * lseg(this,y) in a scenario where this, x and y are all distinct references (since this would require too much permission to e.g., the field this.next), it *is* possible to hold these two predicate instances if, e.g., x=this holds. In this case, the first instance of the predicate holds no permissions at all, but is still a valid instance. In this particular case, it is possible for our tool to often provide a further workaround, as described in Section 6. In the presence of list segments involving *fractional permissions* (denoted for the fraction p in VeriFast by $[p]e.x{\mapsto}y$ or $[p]pred$ for predicates, and in Chalice by $\mathbf{acc}(e.x, p)$), this problematic case can even arise in the former scenario. If it is possible to express list segments which require partial permissions, such as [1/2]lseg(this,x) * [1/2]lseg(this,y) for different x and y; essentially, this allows for two overlapping (and read-only) "views" on different portions of the same list. Our ghost-state-based approach is not able to handle this case, which nonetheless has not yet arisen in the examples we have looked at so far.

Since our ghost field approach involves writing to the ghost fields before a fold statement, we need write permission to the ghost-fields at these points. In addition, we need to put at least some permission to this field inside the predicate body, so that we can refer to its value. However, how to distribute the permission throughout arbitrary code, is less obvious. Our solution is to attempt to determine a field to which a predicate definition always requires full permission; we then *mirror* the permissions to that field throughout the entire program; whereever some permission to the mirrored field occurs, we conjoin the same amount of permission to the ghost field. In particular, this guarantees that whenever the predicate is foldable we also have full permission to the ghost-field (and so, may write to it).

3.3 Initial Translation

In the following subsections we will describe the steps performed by our tool to translate VeriFast programs to Chalice. Note that this translation consists of multiple steps, each of which can potentially fail, aborting the translation; our tool cannot handle every VeriFast program.

We begin with the body (assertion) of a predicate definition, and firstly apply the following translation recursively throughout: every points-to assertion $[p]x.f{\mapsto}v$ (in which v is neither bound using $?y$, nor the wildcard $\_$ expression), is replaced by the assertion $x \neq null * \mathbf{acc}(x.f, p) * x.f == v$. The first conjunct reflects the implicit non-nullity guarantee that VeriFast bakes into points-to assertions, while the latter two reflect the basic encoding from separation logic into implicit dynamic frames [11]. In the case of a bound variable or a wildcard, we translate $[p]x.f{\mapsto}\_$ as $x \neq null * \mathbf{acc}(x.f, p)$; for a bound variable $?v$, subsequent occurrences of v get replaced by $x.f$. The non-nullity conjunct $x \neq null$ is also omitted for the special this reference (since this $\neq null$ is implicit in Chalice predicate definitions, as in VeriFast).

Turning our attention to our running example from Section 2.3, we note that this VeriFast predicate is static; Chalice, in contrast, only supports instance predicates. In order to turn this static predicate into an instance predicate we need to pick one of the reference parameters and make it the new receiver[1]. This only works if the parameter is guaranteed to be non-null: in our running example the predicate body includes the assertion start ≠ null, indicating that start would be a valid choice. Our tool must be able to make this selection automatically; this is the first of several use-cases for an analysis of predicate definitions, capable of extracting (dis)equalities of interest. The technical details of this analysis will be provided in Section 4. In fact, our tool makes further use of our analysis to deal with a wider range of static predicates, for which the new receiver cannot necessarily proven to be non-null; we will outline this idea in Section 6.

By the end of a successful translation of a VeriFast program, each recursive predicate instance will correspond to an instance of the corresponding Chalice predicate in our translated program. Furthermore, as we will detail in the next two subsections, both of our techniques for replacing predicate parameters (described informally in the previous two subsections) result in the introduction of a Chalice function, which retrieves the corresponding value. Therefore, for a predicate p with formal parameters $y_1, y_2, \ldots$, we replace each predicate instance $x.p(t_1, t_2, \ldots)$ with an assertion $p * x.y_1()$ == $t_1 * x.y_2()$ == $t_2 \ldots$ in which $y_1(), y_2()$ etc. are now *function applications*[2]. The following two subsections describe how we find the definitions for the functions.Having applied the steps detailed in *this* subsection, the predicate definition of our running example looks as follows:

```
predicate listseg(LinkedList end, list<int> elems) =
    this != null
    &*& acc(this.value)
    &*& acc(this.next)
    &*& (this.next != end
            ? this.next.listseg &*& this.next.end() == end &*&
              elems == cons(this.value, this.next.elems())
            : elems == cons(this.value, nil));
```

3.4 Inferring Abstraction Functions

After the initial translation steps described in the previous subsection, we attempt to identify predicate arguments to be replaced by abstraction functions (cf. Section 3.1). In order to come up with abstraction functions, we *extract equality facts* about the predicate body; this is another motivation for the underlying analysis we designed, presented in the next section). Applied to our running example, our analysis is able to extract just a single equality fact:

[1] We considered an encoding with a "dummy" receiver object. However, representing that this receiver is the same for all occurring instances of the predicate is difficult.

[2] In fact, we only conjoin equalities for those predicate parameters which we concretely specified in the original predicate instance; those which were bound with ?y or _ syntax are omitted in the resulting assertion.

elems = (next≠end?cons(value, next.elems()):cons(value, nil)). This fact is deduced by combining information from the branches of the conditional expression. Note that no such fact is generated concerning end; this is because information about the value of end is not present in both branches.

Having extracted those equality facts, the approach is quite simple: for each predicate parameter v, we search for an equality fact $v = e$ for some arbitrary expression e (or the symmetric case). In order to make this strategy more robust, we have also implemented a very simple equation solver which is able to solve (some) equalities for v rather than relying on v being already one side of the equation. Furthermore, solving an equation for v can also introduce a side-condition, such as preventing zero division. In case a suitable expression e was found, we can now generate a new function definition for the parameter in question; in case we have extracted more than one equality for v, we pick an arbitrary one. The function takes the original predicate, as well as the side-condition a, as pre-condition, and the body of the function is **unfolding** this.p **in** e where p is the predicate under analysis.

In our running example, we cannot extract any equality for the parameter end, but we have one for elems. Thus, we generate the following new function definition:

```
function elems(): list<int>
  requires acc(this.listseg) {
  unfolding acc(this.listseg) in
    this.next != end ? cons(this.value, this.next.elems())
                     : cons(this.value, nil)
}
```

We then substitute the function's *body* for occurrences of the original parameter in the predicate body[3]. This can typically introduce trivial equalities, and so we simplify the resulting assertion yielding an updated predicate definition:

```
predicate listseg(LinkedList end) =
    acc(this.value);
    &*& acc(this.next);
    &*& (this.next != end
            ? this.next.listseg
              &*& this.next.end() == end()
            : true;
    };
```

Note that, in general, we have to take some extra steps to avoid introducing cyclic function definitions, here. For example, given the (non-sensical) example of a predicate defined by: $p(x, y) \overset{def}{=} (x == y)$, a naïve approach might be to define a function for each parameter, each calling the other function directly in

[3] Note that we cannot substitute a call to the function itself, since the function requires the predicate under analysis as a pre-condition, and these occurrences are inside the predicate body.

its body. Our tool detects and breaks such cycles; this means that at most one parameter will be replaced with a function, in this example.

3.5 Introducing Ghost Fields

Any remaining predicate parameters are handled by introducing extra ghost fields (one per parameter). As described in Section 3.2, this requires us to identify a suitable field in the original program to which full permission is guaranteed to be held whenever the predicate itself is held. Therefore, we require another analysis of the predicate body, able to extract information about the *permissions* held by the predicate. Applied to our running example this results in the knowledge that the predicate body holds full permission on this.value and this.next. In our running example, the predicate parameter end remains to be dealt with and from our learnt knowledge about the permissions it is clear that either field value or field next will suffice. Therefore, picking the first, we will have the permissions to the newly-introduced ghost field end mirror those to value in our output code. In particular, these permissions are included in the assertion under analysis.

We also provide a Chalice function to access the ghost field's value when the predicate is folded. This means that all predicate parameters, whether backed by abstraction functions (as described in the previous subsection) or by ghost fields, can be accessed uniformly (cf. Section 3.3). For our running example, this results in a final set of definitions as follows:

```
ghost LinkedList end;

predicate listseg =
  acc(this.value) &*& acc(this.end) &*& acc(this.next)
  &*& (this.next != this.end ? this.next.listseg
        &*& this.next.end() == this.end
       : true; };

function end(): LinkedList
  requires acc(listseg) {
    unfolding acc(listseg) in this.end;
}

function elems(): list<int>
  requires acc(this.listseg) {
    unfolding acc(this.listseg) in
      this.next != end ? cons(this.value, this.next.elems())
                       : cons(this.value, nil)
}
```

3.6 Translating Programs

While the above sections explain the details of our analysis of predicates, we explain here how we adapt our approach to translating entire programs. The initial translation described in Section 3.3 above is applied also to the rest of the program specifications; this results in the elimination of points-to assertions, and every predicate instance being replaced by the unparameterised predicate, plus appropriate equalities with function invocations.

The translation from Section 3.4 and 3.5 is then applied to each predicate definition in the program; if both of them fail for any definition, then the overall translation fails. For each additional ghost field introduced, the field which was found for "mirroring" is also recorded. The mirroring of permissions is then applied throughout the program text; in our running example, any accessibility predicate of the form $\mathbf{acc}(e.value, p)$ is replaced by a conjunction $\mathbf{acc}(e.value, p)$ && $\mathbf{acc}(e.end, p)$.

Furthermore, when encoding method signatures, if a predicate parameter value is bound inside the pre-condition of a method (e.g. ?xs as shown in Section 2.1), then, in the original code, that variable may be referred in both the method body and the post-condition. Occurrences in the post-condition are replaced by the appropriate function call wrapped inside an old expression (referring to the state of the pre-condition). If the variable is used in the method body, an additional ghost-variable capturing the value is introduced in the beginning of the method body, and used in place of the original occurrences.

Every fold statement of the original program which concerns a predicate for which ghost fields have been introduced, is translated into a sequence of field updates to the ghost fields, followed by a fold of the new predicate. For example, fold this.listseg(x, xs) could be translated to this.end := x; fold this.listseg. However, we should also reflect explicitly-provided values for parameters which have been replaced by abstraction functions in our translation. Where values are *not* concretely provided (i.e. with ?xs or _ syntax) in the input program, we do nothing extra. However, if a concrete expression *is* provided in the original program, we can reflect the correct semantics by adding an assert statement to check that the supplied parameter matches the value of the corresponding abstraction function. Thus, our translation of fold this.listseg(x, xs) actually produces this.end := x; fold this.listseg; assert this.elems() == xs.

Many small syntactic differences between VeriFast and Chalice syntax are trivially handled by the Chalice pretty-printer of our internal AST; we do not detail these syntactic differences here.

3.7 Usage of Predicate Analysis

We have now presented the main ideas behind our translation from VeriFast predicates to Chalice predicates and functions. As remarked throughout, in order to automate the translation, we require an analysis of predicate bodies, able to infer equalities and disequalities (as in Section 3.4 and for handling static predicates) and permissions guaranteed by the predicate body (cf. Section 3.5). Finally, the ability to simplify assertions/expressions during translation helps to keep our output code readable (and also permits our analysis to work more simply). In the next section, we present the semantic analysis of predicate bodies that we developed to tackle all of these problems.

4 Core Analysis

4.1 Our Core Analysis

The semantic analysis of predicates is the main component of our tool and is used in various places as outlined in Section 3. It is designed to follow the reasoning we performed by hand when extracting information from an assertion. In this section we will present the technical details of the so-far informal concepts of extracting (dis)equality facts, subsequently called *value facts*, and extracting *permission facts*, which approximate the permissions making up the footprint of an assertion; both of the two kinds collectively referred to as *analysis facts*.

Our analysis aims to attack questions of the form: what do we know about an assertion/expression under a certain set of assumed facts? Our approach reasons in terms of limited semantic information about assertions, but does not involve external interactions with any kind of theorem prover/SMT solver; we employ a somewhat limited but simple and efficient set of procedures in our tool for accumulating and making use of information extracted from the input program. The use of simplifications/rewriting throughout our analysis is partly to aid code readability, but also limits the impact of our simple representation of facts. We present our algorithms for a representative subsyntax of the assertions which our implementation handles, as follows:

Definition 1 (Assertions and Boolean Expressions for Analysis). *We assume a set of (unspecified, here) unanalysed expressions, ranged over by e. We assume the syntax of e to (at least) include Chalice function applications. Our analysis handles the following syntax of boolean expressions, ranged over by b, and assertions, ranged over by a (in which p represents a permission amount):*

$$\begin{aligned} b &::= \mathit{true} \mid \mathit{false} \mid e_1 = e_2 \mid e_1 \neq e_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid (b?b_1{:}b_2) \\ a &::= b \mid \mathbf{acc}(e.f, p) \mid a_1 * a_2 \mid (b?a_1{:}a_2) \mid e_1.P \end{aligned}$$

Note that this syntax does not include VeriFast-specific assertions; in particular, no points-to assertions or parameterised predicate instances. These will be handled in our tool before invoking our main analysis, as explained in Section 3.3.

4.2 Value facts

To represent heap information described by assertions, our analysis works with equalities and disequalities of expressions, which we call *value facts.* We extract sets of value facts by syntactically traversing an input assertion/expression. The extracted set is constructed to satisfy the property that *if* the input assertion/-boolean expression is true in a state, *then* the conjunction of the set of value facts is also guaranteed to be true. In some cases, it can be useful to know if the reverse implication also holds; this depends on whether the set of value facts is sufficiently rich to precisely characterise when the input assertion holds. When the reverse implication is also guaranteed, we call the set of value facts *invertible*, and track this status with a boolean flag on each set.

Definition 2 (Value Facts and Contexts). Value facts*, ranged over by v, are the subset of boolean expressions generated by the following grammar:* $v ::= (e_1{=}e_2) \mid (e_1{\neq}e_2)$*. Value facts are always treated modulo symmetry; i.e., we implicitly identify* $(e_1{=}e_2)$ *with* $(e_2{=}e_1)$ *when considering them as value facts.*
Value fact sets*, ranged over by V, are sets of value facts, i.e.,* $V ::= \{\overrightarrow{v_i}\}$ *for some value facts* $\overrightarrow{v_i}$*. We write* $\varnothing$ *for the empty value fact set.*
Value fact contexts*, ranged over by Γ, consist of a value fact set along with a boolean constant, i.e.,* $\Gamma ::= V^B$ *for* $B ::= true \mid false$*. When* $B = true$*,* Γ *is called* invertible*.*

The use of implicit symmetries for value facts simplifies several of the following definitions. For example, when we write $(e_1{=}e_2) \in V$, this criterion is insensitive to the order of the two expressions. Similarly, $\{(e_1{=}e_2)\} \cap \{(e_2{=}e_1)\} \neq \varnothing$, with this interpretation; this can avoid discarding such equalities unnecessarily.

Note that value fact contexts are always interpreted via conjunction; we have no way to directly represent disjunctions of sets of value facts. This makes our analysis much simpler, and partly reflects its use cases; as we have seen in Section 3.4, we are typically interested in extracting a single equality fact from our value sets, which can be used as the basis for a new definition. Generalisations are certainly possible, but so far we have not found this to be a serious limitation in our examples. Note that, while we do not directly represent disjunctions, we can still employ conditional expressions as operands to value facts; this can in some cases replace a disjunction between facts, and is useful when analysing conditionals from within a predicate body.

We next define a number of operations on our value fact contexts, that are used in our analysis. The conditional merge is used in the analysis of conditionals, to combine facts about the branches; the other operations are more familiar.

Definition 3 (Value Fact Context Operators). *The* union *of two value fact contexts is defined (where & denotes the boolean conjunction function on two boolean constants) by:* $V_1^{B_1} \cup V_2^{B_2} = (V_1 {\cup} V_2)^{B_1 \& B_2}$ *The* intersection *of two value fact contexts is defined as follows:*

$$\begin{aligned} \Gamma \cap \Gamma &= \Gamma \\ V_1^{B_1} \cap V_2^{B_2} &= (V_1 {\cap} V_2)^{false} \quad \textit{(otherwise)} \end{aligned}$$

The negation *of a value fact context is defined as follows:*

$$\begin{aligned} (neg\ \{(e_1{=}e_2)\}^{true}\ ) &= \{e_1{\neq}e_2\}^{true} \\ (neg\ \{(e_1{\neq}e_2)\}^{true}\ ) &= \{e_1{=}e_2\}^{true} \\ (neg\ \Gamma\ ) &= \varnothing^{false} \qquad \textit{(otherwise)} \end{aligned}$$

The conditional merge *of two value fact contexts (over a boolean expression b) is defined by:*

$$V_1^{B_1} \overset{b}{\curlywedge\!\!\curlywedge} V_2^{B_2} = \{(e_0{=}(b?e_1{:}e_2)) \mid (e_0{=}e_1) \in V_1 \ \textit{and}\ (e_0{=}e_2) \in V_2\}^{false}$$

These operations treat "invertibility" status of contexts very conservatively. Intersection of value fact contexts never results in an invertible value fact context, unless the original contexts are identical, while merging two contexts never returns an invertible context. Furthermore, even when we have an invertible value fact context, we only actually define the negation of the context in a meaningful way for singleton sets of value facts; this results from our choice not to represent disjunctions explicitly in our value facts. It is clear that these operations could be made much more general with a richer treatment of value facts; nonetheless, the above definitions have been sufficiently expressive for our experiments so far.

4.3 Analysis of Boolean Expressions

Having presented our notions of value facts, we can define the analysis we perform on boolean expressions. We define a function *analyseE* which takes as parameters a boolean expression, and a value fact context (representing information that is currently *assumed* in our analysis), and returns a similar pair; the resulting expression is equivalent to the input expression, and the resulting value context contains information about the facts learned in the analysis.

Definition 4 (Analysis of Boolean Expressions).

$$\begin{array}{lrl}
\mathit{analyseE}\ v\ \Gamma & & = ((\mathit{tryEval}\ v\ \Gamma\ ), \{v\}^{\mathit{true}}) \\
\mathit{analyseE}\ \neg b\ \Gamma & & = (\neg b', (\mathit{neg}\ \Gamma'\ )) \\
\quad \textsf{where} & (b', \Gamma') & = \mathit{analyseE}\ b\ \Gamma \\
\mathit{analyseE}\ b_1 \wedge b_2\ \Gamma & & = (b'_1 \wedge b'_2, \Gamma_1 \cup \Gamma_2) \\
\quad \textsf{where} & (b'_1, \Gamma_1) & = \mathit{analyseE}\ b_1\ \Gamma \\
 & (b'_2, \Gamma_2) & = \mathit{analyseE}\ b_2\ \Gamma \cup \Gamma_1 \\
\mathit{analyseE}\ b_1 \vee b_2\ \Gamma & & = (b'_1 \vee b'_2, \Gamma_1 \cap \Gamma_2) \\
\quad \textsf{where} & (b'_1, \Gamma'_1) & = \mathit{analyseE}\ b_1\ \Gamma \\
 & (b'_2, \Gamma'_2) & = \mathit{analyseE}\ b_2\ \Gamma \\
\mathit{analyseE}\ (b_0 ? b_1 : b_2)\ \Gamma & & = \\
 & \textsf{if}\ b'_0 = \mathit{true} : & (b'_1, \Gamma_0 \cup \Gamma_1) \\
 & \textsf{else if}\ b'_0 = \mathit{false} : & (b'_2, (\mathit{neg}\ \Gamma_0\ ) \cup \Gamma_2) \\
 & \textsf{else if}\ b'_1 = \mathit{false} : & ((b'_0 ? b'_1 : b'_2), (\{b_0 == \mathit{false}\}^{\mathit{true}} \cup (\mathit{neg}\ \Gamma_0\ ) \cup \Gamma_2)) \\
 & \textsf{else if}\ b'_2 = \mathit{false} : & ((b'_0 ? b'_1 : b'_2), (\{b_0 == \mathit{true}\}^{\mathit{true}} \cup (\mathit{neg}\ \Gamma_0\ ) \cup \Gamma_1)) \\
 & \textsf{else} : & ((b'_0 ? b'_1 : b'_2), (\Gamma_1 \overset{b'_0}{\wedge\!\!\!\wedge} \Gamma_2)) \\
\quad \textsf{where} & (b'_0, \Gamma_0) & = \mathit{analyseE}\ b_0\ \Gamma \\
 & (b'_1, \Gamma_1) & = \mathit{analyseE}\ b_1\ \Gamma \cup \Gamma_0 \\
 & (b'_2, \Gamma_2) & = \mathit{analyseE}\ b_2\ \Gamma \cup (\mathit{neg}\ \Gamma_0\ )
\end{array}$$

The definition above is designed such that if $\mathit{analyseE}\ b\ \Gamma\ = (b', \Gamma')$, then, in any state in which the conjunction of the value facts in Γ holds, the following properties are also guaranteed. Firstly, $b' \Leftrightarrow b$. Secondly, in any state in which b is true, the conjunction of the value facts in Γ' is true. Thirdly, *if* Γ' is invertible, then in any state in which b is false, the conjunction of the value facts in Γ' is also false. The function $\mathit{tryEval}\ v\ \Gamma$ implements a simple (conservative) attempt to determine whether the inequality v is guaranteed to be either true or false, assuming the value facts in Γ. If either can be shown, then *true* or *false* are returned, otherwise v is returned unchanged.

The ability to simplify conditional expressions in four cases above gives more precise information about the assertion; only in the case that neither can the condition b_0 be simplified, nor can either of the two assertions a_1 and a_2 be rewritten to *false*, does the $⩕$ operator have to be applied (typically losing information). Note that the if-conditionals in the analysis of conditional expressions compare for *syntactic* equality of boolean expressions. These conditionals also propagate simplifications throughout the structure of expressions, where possible.

4.4 Permission Facts

In the case of analysing assertions (rather than just boolean expressions), we also require information about the *permissions* required by a particular assertion. We do not require very precise permission accounting (which would be difficult in the presence of aliasing questions), since ultimately we are only concerned with two particular outcomes - whether or not an assertion is known to guarantee *no* permissions (e.g., a simple boolean expression), and whether or not it guarantees *full* permission to some field location. As we have seen, knowing that a predicate body holds full permission to at least one field is crucial when trying to replace in-parameters; knowing that a predicate body holds no permissions at all is beneficial when applying some tricks to deal with static predicates. Guided by these goals, we choose a simple representation of permission facts for our analysis.

Definition 5 (Permission Facts and Operations). *Permission facts, ranged over by Π, are defined by $\Pi ::= \psi \mid \pi$, where the symbol ψ is called the* unknown permission fact *(written ψ), and where π is a* known permission fact set*: a (possibly empty) set of tuples of the form (e, f, p), satisfying the constraint that no pair of e, f occurs in more than one tuple.*
Addition *of permission facts is defined as follows:*

$$\begin{array}{ll} \psi + \psi = \psi & \psi + \varnothing = \varnothing + \psi = \psi \\ \pi_1 + \pi_2 = \pi_1 \uplus \pi_2 & \psi + \pi = \pi + \psi = \pi \quad \textit{otherwise} \end{array}$$

where $\uplus$ takes the union of the two sets, except that when the same e, f occur in a tuple of each set, the resulting set has a tuple with the sum *of the two permission amounts: e.g., $\{(x, f, p_1), (y, f, p_2)\} \uplus \{x, f, p_3\} = \{(x, f, p_1 + p_3), (y, f, p_2)\}$*
Intersection *of permission facts is defined by:*

$$\psi \cap \Pi = \Pi \cap \psi = \psi \quad \pi_1 \cap \pi_2 = \{(e, f, min(p_1, p_2)) \mid (e, f, p_1) \in \pi_1 \textit{ and } (e, f, p_2) \in \pi_2\}$$

The unknown permission fact is employed in our analysis whenever we are forced to be approximate conservatively, either because an exact fractional permission is not know, or because an assertion contained a further predicate instance (we do not unfold predicate definitions recursively, and this would in general not terminate).

We are now ready to define our analysis of assertions: we define a function *analyse* , which takes an assertion and value fact context (assumed knowledge) as input, and produces a triple of (possibly simplified) assertion, value fact context, and permission fact, as output.

Definition 6 (Analysis of Assertions).

$$
\begin{array}{lrl}
\multicolumn{2}{l}{\mathit{analyse}\ e\ \varGamma} & = (e', \varGamma', \varnothing) \\
\quad \textsf{\textit{where}} & (e', \varGamma') & = \mathit{analyseE}\ e\ \varGamma \\
\multicolumn{2}{l}{\mathit{analyse}\ a_1 * a_2\ \varGamma} & = (a_1' * a_2', \varGamma_1 \cup \varGamma_2, \varPi_1 + \varPi_2) \\
\quad \textsf{\textit{where}} & (a_1', \varGamma_1, \varPi_1) & = \mathit{analyse}\ a_1\ \varGamma \\
 & (a_2', \varGamma_2, \varPi_2) & = \mathit{analyse}\ b_2\ \varGamma \cup \varGamma_1 \\
\multicolumn{2}{l}{\mathit{analyse}\ \mathbf{acc}(e.f, p)\ \varGamma} & = (\mathbf{acc}(e.f, p), \varnothing^{\mathit{false}}, \{(e, f, p)\}) \\
\multicolumn{2}{l}{\mathit{analyse}\ (b_0 ? a_1 : a_2)\ \varGamma} & = \\
 & \textsf{\textit{if}}\ b_0' = \mathit{true}: & (b_1', \varGamma_0 \cup \varGamma_1, \varPi_1) \\
 & \textsf{\textit{else if}}\ b_0' = \mathit{false}: & (b_2', ((\mathit{neg}\ \varGamma_0\ ) \cup \varGamma_2), \varPi_2) \\
 & \textsf{\textit{else if}}\ b_1' = \mathit{false}: & ((b_0' ? b_1' : b_2'), (\{b_0 == \mathit{false}\}^{\mathit{true}} \cup (\mathit{neg}\ \varGamma_0\ ) \cup \varGamma_2), \varPi_2) \\
 & \textsf{\textit{else if}}\ b_2' = \mathit{false}: & ((b_0' ? b_1' : b_2'), (\{b_0 == \mathit{true}\}^{\mathit{true}} \cup (\mathit{neg}\ \varGamma_0\ ) \cup \varGamma_1), \varPi_1) \\
 & \textsf{\textit{else}}: & ((b_0' ? b_1' : b_2'), (\varGamma_1 \overset{b_0'}{\mathbin{/\!\!\wedge\!\!\backslash}} \varGamma_2), \varPi_1 \cap \varPi_2) \\
\quad \textsf{\textit{where}} & (b_0', \varGamma_0) & = \mathit{analyseE}\ b_0\ \varGamma \\
 & (a_1', \varGamma_1, \varPi_1) & = \mathit{analyse}\ a_1\ \varGamma \cup \varGamma_0 \\
 & (a_2', \varGamma_2, \varPi_2) & = \mathit{analyse}\ a_2\ \varGamma \cup (\mathit{neg}\ \varGamma_0\ )
\end{array}
$$

Note that, as in Definition 4, the cases for conditional expressions allow us to retain more-precise information and simpler assertions, where possible. In particular, the $\cap$ and $\mathbin{/\!\!\wedge\!\!\backslash}$ operators are not applied if an earlier case applies.

The analysis rules above are defined to guarantee similar properties to those for Definition 4. Specifically, if $\mathit{analyse}\ a\ \varGamma\ = (a', \varGamma', \varPi)$, then, in any state in which the conjunction of $\varGamma$ holds:

1. a and a' are equivalent assertions (typically, a' is syntactically simpler).
2. Whenever a is true, the conjunction of the value facts in $\varGamma'$ is true.
3. If $\varGamma'$ is invertible and a is false, the conjunction of $\varGamma'$ is false.
4. If $\varPi = \pi$ for some known permission fact set π, then a logically entails the iterated conjunction of all recorded permissions: $\ast\{\mathbf{acc}(e.f, p) \mid (e, f, p) \in \pi\}$

We make use of the above analysis at every stage of our translation: in particular, to discover equalities suitable for generating function definitions (as detailed further in the next subsection), and identify suitable fields and parameters for the handling of ghost field permissions and static predicates, as discussed in Section 3. We also constantly simplify the assertions we are working with, as a side-effect of our algorithm above.

4.5 Equation Solver

In addition to the main analysis described above, we have defined a simple equation solver, with the aim of rewriting equalities into the form $x = e$ for some chosen variable x. This is needed, for example, when we wish to extract new function definitions (to replace predicate parameters), or when dealing eliminating bound variables from VeriFast expressions. Our solver takes the left-hand and the right-hand sides of an equation, and a variable to solve for. It either fails (indicated by a special $\perp$ return value), or returns a result expression and an additional side-condition (assertion), expressing for instance that no zero-division occurs when solving an equation containing multiplications.

Definition 7 (Equation Solver). *We write $occs(x, e)$ for the number of occurrences of the variable x in expression e. We define an operation $solve\ e\ e'\ x$, which is only defined when $occs(x, e) = 1 \wedge occs(x, e') = 0$, by the rules below. Rules using the meta-variable e_1 have a side-condition: that $occs(x, e_1) = 1$ (i.e., the rule only applies if x occurs in this sub-expression).*

$$
\begin{array}{lll}
solve\ x\ e_3\ x & = (e_3, true) \\
solve\ e_1{+}e_2\ e_3\ x & = solve\ e_1\ e_3{-}e_2\ x \\
solve\ e_2{+}e_1\ e_3\ x & = solve\ e_1\ e_3{-}e_2\ x \\
solve\ e_1{-}e_2\ e_3\ x & = solve\ e_1\ e_3{+}e_2\ x \\
solve\ e_2{-}e_1\ e_3\ x & = solve\ e_1\ e_2{-}e_3\ x \\
solve\ e_1{*}e_2\ e_3\ x & = (e', e_2 \neq 0 \wedge c') \\
\quad \text{where} \quad (e', c') & = solve\ e_1\ e_3/e_2\ x \\
solve\ e_2{*}e_1\ e_3\ x & = (e', e_2{\neq}0 \wedge c') \\
\quad \text{where} \quad (e', c') & = solve\ e_1\ e_3/e_2\ x \\
solve\ e_1/e_2\ e_3\ x & = solve\ e_1\ e_3{*}e_2\ x \\
solve\ e_2/e_1\ e_3\ x & = (e', e_3{\neq}0 \wedge c') \\
\quad \text{where} \quad (e', c') & = solve\ e_1\ e_2/e_3\ x \\
solve\ e_1\ e_3\ x & = (\perp, false) \\
\quad otherwise &
\end{array}
$$

Note that the last case can match arbitrary expression syntax; for example, if x occurs as the parameter to a function application. The aim of our solver is not to apply deep reasoning to resolve such scenarios, but just to apply simple rewrites where possible.

Based on the *solve* function, we can then define *findExpressionFor* , which takes a value fact context and a variable name, and returns a set of expressions e' known to be equivalent to x, paired with corresponding side-conditions c. Furthermore, the function takes a set of “forbidden” variables vs that are not allowed to occur in the identified expressions. This expression e' along with the potential side-condition c can then directly be used to build the new abstraction function as outlined in Section 3.4.

$$
\begin{array}{l}
findExpressionFor\ V\ x\ vs\ = \{(e', c) \mid (e', c) \in s \wedge e' \neq \perp \wedge vars(e') \cap vs = \varnothing\} \\
\quad \text{where} \\
s = \{(solve\ e_1\ e_2\ x\ ) \mid (e_1 == e_2) \in V\ \ and\ \ occs(x, e_1) = 1\ \ and\ \ occs(x, e_2) = 0\}
\end{array}
$$

5 Results and Evaluation

We have performed some preliminary experiments to gain some insight about the feasibility of our approach. Rather than as a full evaluation, it should be understood more as a starting point for further investigation and work. Furthermore, we were also interested in how well Chalice does on the translated examples and whether some general observations can be made when comparing Chalice to VeriFast.

Our experiments indicate so far that practical predicates can be handled by our analysis. From the (Java) examples provided with VeriFast, not a single one failed due to an untranslatable predicate; however only a handful of the test cases can actually be translated by our tool and the others failed due to a number of features not supported by Chalice at the time the tool was written, such as subtyping, general abstract data types, and unsupported language features. Nonetheless, the following examples from VeriFast could be translated: *Spouse*, *Spouse2*, *SpouseFinal*, *AmortizedQueue*, and *Stack*; they can be found together with the translated Chalice code in our examples [1]. We also included two of our own hand-written examples: *LinkedList*, *LinkedListSeg_simple*, and the full running-example (*LinkedListSeg*). While those are only few examples, several contain non-trivial recursive predicates, with both in and out parameters.

All the examples above translated without any modification, except for *Stack* which needed a single modification (as commented) in the original file[4]. Furthermore, they all verify in Chalice except for *AmortizedQueue* which needed an additional tweak (due a somewhat prototypical current support for static functions in Chalice) unrelated to the handling of the predicates. Notice that some of the examples will produce warnings from Chalice regarding termination checks for recursive functions which operate purely on abstract data types; this is due to a lack of support for general termination measures.

The only predicate that our approach could not handle is in *LinkedListSeg*, due to the predicate not being unique for a given receiver as the empty segment can be fold arbitrary often on any receiver. However, we will outline in section 6 some ideas of how we could extend our analysis to handle such cases.

In addition, our experiments indicate that Chalice can sometimes manage with fewer ghost annotations than VeriFast needs. In particular, several VeriFast examples contained consecutive unfold / fold pairs on exactly the same predicate instance; often they serve only for the purpose of binding the arguments to a variable, and in this case we can just call the corresponding getter function without having to unfold the predicate in Chalice. Interestingly, removing those superfluous unfold / fold pairs seems to speed up the verification in Chalice significantly for harder examples; we intend further investigation to disclose the underlying reason for this. Furthermore, many of the built in lemma methods of VeriFast (especially for lists) are not required in Chalice; while adding an equivalent assertion helps in terms of verification speed (probably by pointing the verifier into the right direction), the verification still succeeds without them.

6 Conclusions and Future Work

In this paper, we have presented the first implemented encoding from a subset of separation logic to a verifier based on first-order-verification-condition-generation (Chalice). To achieve this for interesting examples, we have presented a number of novel ideas for eliminating predicate parameters, and employing alternative verification features available. In particular, we have presented a sim-

[4] VeriFast permits the wildcard _ to be used even for in parameters; in this case, the symbolic heap is searched for any appropriate predicate instance. We provided the (obviously unique, in this example) value explicitly.

ple but flexible automatic analysis of predicate definitions, which enables us to rewrite such definitions without user intervention.

Our analysis is currently limited in several ways, largely as an engineering trade-off between simplicity and expressiveness. However, our approach is easily extensible in many ways which do not affect the overall approach. Firstly, our value facts could be easily extended to capture more precise information about the entailment between the assertion and our knowledge; the same holds also for our permission facts. One can construct predicates in which reasoning in terms of *inequalities* is desirable. In addition, disjunctions could be added to value-fact contexts, making the negation of a context more often expressible. Both extensions would enable more examples in general, but also make the analysis itself, and particularly the entailment checking (built into *tryEval* of Section 4.3) much more complex; indeed, it is likely that a prover would be required for serious reasoning about inequalities. The *tryEval* function, as well as our equality solver (Section 4.5) could be made arbitrarily more sophisticated; at present, not even transitivity of equalities is taken into account, but in principle complex theories could be incorporated, with the aid of suitable prover support.

The main limitation of our current approach is that whenever a predicate has an in parameter, predicate instances must be unique per receiver; many separation logic predicates (such as the classical lseg definition) do not satisfy that restriction. On the other hand, the cases for which a predicate describing a recursive data structure can be held for the same receiver in many different ways, often coincide with the predicate instance not holding any permissions. In such cases, the predicate "degenerates" to a boolean expression referring only to the parameter values; for example, the base case of the lseg predicate in Section 2. Our tool currently has the ability to deal with a specific kind of these predicates: if the designated receiver r of a static predicate is potentially null, we check whether the predicate does not hold any permissions when $r = null$. If, in addition, all other predicate parameters can be uniquely determined in this case, we can drop the problematic predicate instances (in these cases) in our output code, by replacing all the occurrences of the predicate with e.g., $r \neq null \Rightarrow r.p$) and replacing all calls to getter functions (which would not make sense on a null receiver) with constant expressions: e.g., $r \neq null\ ?\ r.f() : \mathsf{const\_val}$. This limited trick is already implemented, but in future work, we would like to generalise the technique to deal with cases where the condition for not needing to hold the predicate might be arbitrary (but still determinable from the in parameters). The classical list segment predicate would need $\mathsf{start} \neq \mathsf{end}$ as the condition; we wish to deduce this automatically and then apply a similar approach.

In trying our tool out on several VeriFast examples, we have established the feasibility of an encoding, in which separation logic examples are handled entirely by an SMT solver. This has also provided us with several new and interesting test cases for the Chalice verifier, and the ability to experiment with comparisons between the two approaches. While Chalice is generally able to handle the output code efficiently, one example (the translation of the AmortizedQueue.java VeriFast example) takes many minutes to verify (while VeriFast handles all examples in a few seconds). Interestingly, we found that we can delete many of the resulting fold/unfold statements in this example by hand: the verification still succeeds, and faster (presumably because the SMT encoding involves representing fewer states). We also found that we can delete several assertions which correspond to calls to lemma methods in the original code: these are also not required for the Chalice verifier to succeed, although do seem to speed up the verification. We would like to investigate these issues further, and believe that our tool opens up new and interesting possibilities for comparing the two approaches.

Acknowledgements

We are very grateful to Bart Jacobs for many useful and detailed discussions of predicates in VeriFast, and providing us with his source code to study. We would also like to thank the anonymous reviewers, whose feedback significantly improved the presentation of our work.

References

1. Verifast2chalice (online). `http://www.pm.inf.ethz.ch/research/chalice/verifast2chalice.zip`.
2. L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of TACAS (ETAPS 2008)*, pages 337–340, 2008.
3. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26. ACM Press, 2001.
4. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical report, Katholieke Universiteit Leuven, August 2008.
5. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java - invited paper. In *NFM*, 2011.
6. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *European Symposium on Programming (ESOP)*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer-Verlag, 2009.
7. K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In A. D. Gordon, editor, *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, 2010.
8. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, London, UK, 2001. Springer-Verlag.
9. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM Press, 2005.
10. M. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. In *ESOP*, 2011.
11. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.
12. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 5653, pages 148–172, July 2009.
13. A. J. Summers and S. Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *ECOOP*, volume 7920, pages 129–153, 2013.