Quantified Permissions for Random Access Data Structures

Korbinian Breu
Supervisors: Dr. Yannis Kassios, Dr. Alex Summers

October 4, 2013

1 Verifiers for Concurrent Programs Lack Random Access

Most automated verifiers for concurrent programs, e.g. VeriFast [JP08] or Chalice [LMS09], use abstract predicates to specify access permissions to data structures [PB05]. Abstract predicates are named assertions that abstract over access permissions, which represent the right of a thread to access a given field. For recursive data structures, such as trees, these predicates are also recursive. In those verifiers, abstract predicates are currently the only way to specify properties over a statically unknown number of fields.

Abstract predicates implicitly dictate a specific traversal of the data structure. The nature of this implicitly dictated traversal complicates proofs of algorithms that traverse in a different direction. Consider an algorithm performing binary search on a sequence: It compares with the center element, then with the center element of the upper or lower half, and proceeds to access elements randomly. It is designed to access elements in this way and does not play well with a recursive data structure. Thus, currently, designing and specifying such algorithms is hard and unnatural. We aim at reducing the specification programmer's burden and allowing for a more natural specification style. Thus, we provide a new mechanism to specify properties over random access structures, i.e. sets, multisets, and sequences: We extend an automatic verifier by universal quantifiers over access permissions. Our approach will be implemented in Silicon, a verifier for the intermediate verification language SIL based on the concepts of Implicit Dynamic Frames, Symbolic Execution, and Fractional Permissions [Sch11].

Consider the example in listing (1), which illustrates a use case of the proposed feature. The sample specifies a sorting algorithm in the style of SIL. SIL, like Chalice, uses $acc(o.f)$ to denote the permission to access field f of object o [LMS09]. The data structure consists of a sequence of cells C, each cell with its value stored in field *value*. Lines 6,7, and 9 show how the proposed feature could look. In line 6, the specification quantifies over the elements in the sequence to gain access to all of the values. It hands them all back in line 8, and line 9 ensures that the sequence is now sorted. Our goal is that within the implementation, which is omitted here, access to the elements of the sequence can now be random.

```
1  var C: Seq[Ref]
2  var value: Int
3
4  method sort(this:Ref)
5  requires acc(this.C)
6  requires ∀i:Int :: 0≤i ∧ i<|this.C|−1 ⟹ acc(c.value)
7  ensures acc(this.C)
8  ensures  ∀i:Int :: 0≤i ∧ i<|this.C|−1 ⟹ acc(c.value)
9  ensures  ∀i:Int :: 0≤i ∧ i<|this.C|−1 ⟹ this.C[i].value ≤
       this.C[i+1].value
10 ensures ...
11 {
12   // implementation
13   ...
14 }
```

Listing 1: This specification of a sorting algorithm makes use of universally quantified permissions by getting access to all values of the elements in the collection C that is to be sorted.

2 Core

Ideally, it would be possible to state unrestricted universally quantified expressions

$$\forall x_1 : T_1, x_2 : T_2, ..., x_n : T_n \ : \ Q(x_1, x_2, ..., x_n) \tag{1}$$

in SIL. More precisely, in the language of Implicit Dynamic Frames, by stating (1) we mean that Q has to hold on a separate part of the heap for each instantiation. Thus, equation (1) is equivalent to

$$\bigwedge_j Q(x_1^j, \ldots, x_n^j) :\Longleftrightarrow Q(x_1^1, \ldots, x_n^1) \ * Q(x_1^2, \ldots, x_n^2) \ * \ldots \tag{2}$$

where $*$ denotes the separating conjunction[1] and x_i^j is the value of x_i in the jth instantiation[2].

However, there are a lot of open questions: How to inhale and exhale universally quantified expressions? How to represent them on the heap? How to encode them for Z3 if necessary? Therefore, to make the problem more manageable, we plan to solve a restricted variant first.

The first part of the core consists of extending Silicon by universal quantifiers over permissions for sets instead of multisets. We also restrict ourselves to constant permissions. We expect this restricted version of the problem to be substantially easier to solve. In the second part, we explore lifting these restrictions to allow multisets and arbitrary permissions, and evaluate the results.

2.1 Restrictions of the problem

We impose the following restrictions on the first core part:

1. **Quantify only over one variable**

$$\forall x : T \ : \ Q(x) \tag{3}$$

   Restricting ourselves to one variable still allows for interesting examples and simplifies the mathematical representation for now.

2. **The expression under the quantifier must be an implication** of the form

$$\forall x : T \ : \ \underline{B(x) \implies acc(e_1(x).f, e_2(x))} \tag{4}$$

   where B is a pure boolean expression without permissions, $e_1(x)$ is an expression that evaluates to a receiver, f is the accessed field or an abstract predicate, and $e_2(x)$ evaluates to a fractional permission. This may be more of a simplification than a restriction, as based on the structure of our fragment of Implicit Dynamic Frames, all expressions can be stated in this form.

   We now simplify the permission expression e_2.

3. **Only allow constant permissions** of the form

$$\forall x : T \ : \ B(x) \implies acc(e_1(x).f, \underline{constant}) \tag{5}$$

   where *constant* is either *write*, *read*, or a constant fraction. We thereby forbid local variables, method parameters, and expressions dependent on the heap.

[1] Note that in Silicon the separating conjunction is written "&&".

[2] Using the logical conjunction instead of the separating conjunction would reduce the quantification to quantification over sets instead of multisets. We impose this restriction differently, by fixing the receiver, in section (2.1).

4. **Fix the receiver to be a quantified variable**

   To restrict further, we take a look at the receiver expression e_1. In its current unrestricted form, it could evaluate to the same receiver object more than once. This case of aliasing would require us, for example, to ensure that write permissions are not given twice. To avoid having to deal with aliasing at first, it may therefore be beneficial to restrict e_1:

   $$\forall x : T \ : \ B(x) \Longrightarrow acc(\underline{x}.f, constant) \tag{6}$$

   The receiver x is now fixed to a quantified variable. Permissions to access $x.f$ are given exactly once or not at all. This restriction essentially reduces the problem to sets, a problem which may be substantially more tractable than the previous, unrestricted version in equation (1).

We will implement this restricted case in SIL/Silicon and test it on examples.

2.2 Relax the Restrictions

The second core part of the project explores lifting restrictions on our journey to supporting the unrestricted case given in equation (1).

1. **Examine the multiset case**. Once we no longer restrict the receiver, the receiver expression e_1 might evaluate to the same object twice or more often. Therefore, we have to deal with aliasing. For example, when we assume and give permissions according to

   $$\forall x : T \ : \ x \in C \Longrightarrow acc(\underline{x.a.f}, write) \tag{7}$$

   and there are two or more objects in C that point to the same object in field a, we need to be careful not to give away write permission more than once.

   We will analyze the implications and experiment with changes to Silicon's current heap model.

2. **Try to allow unrestricted permissions**, e.g. abstract permissions, non-constant permissions, and permission expressions that depend on the heap. It might be interesting to distinguish the following three cases of permission expressions:

   - *write*
   - *read* with known upper bound on the number of reads
   - *read* with with no such known upper bound

3. **Explore examples and evaluate our implementation**. We have some examples that already use quantified permissions in SIL, but they have to be checked for errors and quite possibly debugged. Moreover, there might be many more interesting examples, also smaller ones, that could employ universal quantification in a useful way. We plan to evaluate our implementation on them: Is it convenient and does it allow a natural proof style? Is the performance satisfactory?

2.3 Deliverables

The following deliverables constitute a successful core:

- An **implementation of quantified permissions over sets** is created as an extension of the existing Silicon verifier. The usage resembles the notation in the sample. Additionally, an **implementation of the multiset case** is developed. The required syntax and the capabilities of the implementation are well documented.
- A **description of the formal design** as part of the report. This description gives an abstract view of the involved operations, e.g. inhaling/exhaling, as well as the modifications to Silicon's core, e.g. the representation of heap chunks and their lookup. It points out restrictions that are still imposed and clearly state to what extent they simplify the problem. The description also argues for the soundness of the chosen approach.
- An **evaluation of the implementation** as part of the report. The evaluation exhibits samples to show the usefulness and functional correctness of the implementation. It also compares the performance of the modified Silicon with the previous version.

3 Extensions to be Explored

We propose the following extensions:

1. **Generalize the insights gained to other verification concepts**. For example, think about how the concepts used in Silicon and Symbolic Execution apply to Verification Condition Generation. How do the problems relate, how do they differ?
2. **Support aggregates**, e.g. *sums*, the *max* function, or the *avg* function. Aggregates are useful to specify collections and recursive data structures. Universal

quantifications help to create self-framing assertions about aggregates, where self-framing means an assertion includes permissions to all heap locations it accesses [Kas06]. The assertion

$$(\forall x : T \ : \ x \in C \implies acc(x.f, read)) \wedge \sum_{y}^{y \in C} y.f \geq 0 \tag{8}$$

is self-framing, as the universally quantified accessibility predicate gives access to all heap locations needed in the sum.

3. **Find efficient triggering strategies** for the underlying prover, which is in our case Z3. As we cannot predict Z3's strength when dealing with quantified expressions, it might be fruitful to tune the performance by providing appropriate triggers.

4. **Check if our new technique helps to specify recursive data structures more succinctly**. The Composite pattern is a recursive data structure, and was proposed as a verification challenge in [LLM07]. Its invariant is specified from top to bottom, but adding an element breaks the invariant at the bottom and then fixes it from bottom to top. When relying on the conventional recursive predicates, proving the correctness of this pattern results in lengthy solutions: The VeriFast team's proof consists of lemma methods that require five times as many lines as the original code [JSP08]. Our approach may be well suited to reduce the amount of lemma methods.

4 Time Plan

We plan to conduct this project in sprints, where each sprint consists of design, analysis, implementation, and documentation of a milestone. These milestones will allow us to integrate changes into the Silicon development release incrementally and iteratively. We separated the project into three sprints:

16.09.–04.10.2013 Finish proposal, prepare and give initial presentation

7.10.–08.11.2013 **The restricted set case**, 5 weeks: 4 weeks requirements, design, and implementation – 1 week documentation and integration. Evaluate different approaches for the set case and implement a feasible approach in Silicon.

11.11.–13.12.2013 **The multiset case**, 5 weeks: 3 weeks requirements, design, and implementation – 1 week documentation and integration – 1 week prepare presentation. Explore the multiset case.

Mid of November Intermediate presentation: Gather feedback and ideas.

16.12.–20.12.2013 **Evaluation** of the implementation

06.01.–24.01.2014 Final testing, bugfixing and integration, buffer time

27.01.–28.02.2014 Writeup and final presentation

References

[JP08] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, 2008.

[JSP08] Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the Composite Pattern using Separation Logic. In *SAVBCS*, 2008.

[Kas06] Ioannis T. Kassios. Dynamic Frames: Support for Framing , Dependencies and Sharing without Restrictions. *FM*, 4085(1):268–283, 2006.

[LLM07] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, April 2007.

[LMS09] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of Concurrent Programs with Chalice. pages 1–29, 2009.

[PB05] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *ACM SIGPLAN Notices*, 2005.

[Sch11] Malte Schwerhoff. Symbolic Execution for Chalice. Master's thesis, 2011.