Verifying Separation Logic Contracts in Chalice

Project description

Daniel Jost
dajost@students.ethz.ch

April 26, 2012

1 Background

One of the most demanding aspects of formal program verification is dealing with the heap. Especially, one is interested in framing (what portions of the heap might get changed by a method call) to make verification modular. Another crucial aspect is sound reasoning about concurrency, which is commonly facilitated by introducing some form of access permissions.

Among others, two different approaches which have been pursued in order to tackle these problems are: On one side, separation logic, an extension of Hoare logic based on the idea of combining partial heaps and having permissions to heap location. This allows to reason about framing and concurrency. Plenty of examples exists using separation logic. On the other side, there are implicit dynamic frames, based on a total heap and explicit accessibility predicates to heap locations as part of the assertions. The framing problem is dealt with by specifying the portion of the heap a function is allowed to modify.

In recent work, Matthew Parkinson and Alex Summers showed how the assertions from separation logic can be translated into those of implicit dynamic frames. In order to achieve that, they introduce the total heaps permission logic, which is a syntactical superset of both languages, showed the semantical equivalence of the separation logic part to the original separation logic and then introduced a projection to the implicit dynamic frame subset. Many tools based on separation logic have been implemented, including VeriFast. The purpose of this project is to exploit this translation and reuse existing examples from VeriFast in Chalice, a tool based on separation logic.

2 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, 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.

3 Extensions

Extra work can be performed in reevaluating the supported subset of VeriFast in order to handle a broader class of examples. This might lead to additional insight 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.

Explicitly, 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 predicated are especially used whenever the program is dealing with recursive data structures like linked-lists. However, the mapping from parameterized predicates in VeriFast to parameterless predicates and heap-dependent functions in Chalice is tricky and no general methodology has yet been developed.

Additional language constructs from VeriFast like fixpoint functions might also be tackled. Closely related in VeriFast are also abstract data types, which currently are not implemented in Chalice. So one might either try to translate them to Chalice as well, or more likely go ahead and implement them in Chalice, as that would be

4 Goals

Core

- Identify a suitable subset of the VeriFast assertions to be handled
- Write a compiler that translates VeriFast assertions to Chalice assertions
- Working 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

- Extending the methodology for parameterized predicates
- Extending the translation for recursive predicates and their folding respectively unfolding
- Evaluate the practical feasibility of the extended methodology and translation
- Handle fixpoint functions in the translation
- Implement abstract datatypes for Chalice