A HIP and SLEEK Verification System

Wei-Ngan Chin, Cristina David, Cristian Gherghina
Department of Computer Science, National University of Singapore

Abstract

The HIP/SLEEK systems are aimed at automatic verification of functional correctness of heap manipulating programs. HIP is a separation logic based automated verification system for a simple imperative language, able to modularly verify the specifications of heap-manipulating programs. The specification language allows user defined inductive predicates used to model complex data structures. Specifications can contain both heap constraints and various pure constraints like arithmetic constraints, bag constraints. Based on given annotations for each method/loop, HIP will construct a set of separation logic proof obligations in the form of formula implications which are sent to the SLEEK separation logic prover. SLEEK is a fully automatic prover for separation logic with frame inferring capability.

1. Description

HIP is a separation logic based automated verification system for a simple imperative language, able to modularly verify heap-manipulating programs.

The system can handle programs with complex data structures. HIP accepts abstract descriptions for such structures in the form of inductive predicates. Predicates are represented as separation logic formulae that describe the shape of data structures together with their derived properties, such as length, height and bag of values.

Given annotations for each method/loop with one or more pre/post conditions[2], the HIP verifier constructs a set of obligations in the form of implication checks between pairs of formulae which are then sent to the SLEEK prover to be discharged. The specification language allows rich specifications[5] that contain both heap constraints expressed as separation logic formulae and several different logic fragments like Presburger arithmetic, bags, lists for the pure constraints[1]. By making use of set/bag solvers, the user can also encode reachability conditions as a set/bag of values that can be collected from some given data structure. Such conditions are then automatically discharged by HIP.

HIP relies on the SLEEK prover in order to discharge the verification conditions. SLEEK[7] is a fully automatic prover for separation logic with frame inferring capability. It takes as input two heap states represented by separation formulae, and checks if one formula (the antecedent) entails the other (the consequent). The antecedent may cover more heap states than the consequent, so a residual heap state which represents the frame condition can be returned by the prover. This residual heap state will include the pure state of the antecedent. SLEEK also supports instantiation of logical variables that appear during the entailment as existential variables in the consequent. As part of the implication check, SLEEK discharges the heap obligations, the obligations pertaining to the shape of data structures and translates the remaining pure obligations to pure constraints that can be discharged by theorem provers. The list of possible pure provers includes Omega (a Presburger prover), MONA (based on monadic second-order logic), CVC Lite, Z3, and Isabelle (a general purpose theorem prover that supports some automatic tactics).

2. Examples

In the current demo, we are going to present a suite of examples handling mutable data structures. Each example will contain the heap predicates describing the data structures used by the example, together with a few methods performing operations on the data structures. Each method is decorated with pre/post conditions. The task of the HIP verifier is to check each method implementation against the given specification. If the specification is not met then it reports the obligation that could not be discharged and the source of it. Furthermore in the process of verifying a method, pointer safety guarantees like null dereferencing are naturally enforced. Similarly, bounds checks, can be encoded and verified as well.

Some of the examples presented in the demonstration are enumerated below:

- Examples involving an acyclic linked list (that terminates with a `null` reference). The heap predicate describing the

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

list can be described by:

$$\texttt{root} :: \texttt{ll}\langle \texttt{n} \rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{n}{=}0) \vee (\exists \texttt{i}, \texttt{m}, \texttt{q} \cdot \texttt{root}{::}\texttt{node}\langle \texttt{i}, \texttt{q} \rangle * \texttt{q}{::}\texttt{ll}\langle \texttt{m} \rangle \wedge \texttt{n}{=}\texttt{m}{+}1)$$
$$\textbf{inv}\ \texttt{n} \geq 0$$

The parameter `n` captures a *derived* value that denotes the length of the acyclic list starting from `root` pointer. The above definition asserts that an `ll` list can be empty (the base case `root=null`) or consists of a head data node (specified by $\texttt{root}{::}\texttt{node}\langle \texttt{i}, \texttt{q} \rangle$) and a separate tail data structure which is also an `ll` list ($\texttt{q}{::}\texttt{ll}\langle \texttt{m} \rangle$). The $*$ connector ensures that the head node and the tail reside in disjoint heaps. We also specify a default invariant $\texttt{n} \geq 0$ that holds for all `ll` lists. (This invariant can be verified by checking that each disjunctive branch of the predicate definition always implies its stated invariant. In the case of `ll` predicate, the disjunctive branch with $\texttt{n} = 0$ implies the given invariant $\texttt{n} \geq 0$. Similarly, the $\texttt{n} = \texttt{m} + 1$ branch together with $\texttt{m} \geq 0$ from the invariant of $\texttt{q}{::}\texttt{ll}\langle \texttt{m} \rangle$ also implies the given invariant $\texttt{n} \geq 0$.) Our predicate uses existential quantifiers for local values and pointers, such as $\texttt{i}, \texttt{m}, \texttt{q}$.

With the `ll` predicate we can, for example, specify the expected behaviour of a list append operation as follows:

```
void append(node x, node y)
  requires x :: ll⟨n1⟩* y :: ll⟨n2⟩ ∧ n1>0
  ensures x :: ll⟨n1 + n2⟩;
```

This specification guarantees that if two lists of sizes n_1 and n_2 are appended then the result will be a list of size $n_1 + n_2$. Depending on the properties of interest and the precision required, the user provided specifications can be easily refined further. The specification language allows the user to supply varied granularities for the precision of the specification, by default coarser specifications can be used and more refined ones can be added only when required. For example, the previous specification does not require that the resulting list contains all the initial elements however if such a constraint is needed it can easily be added.

- Examples regarding sorted linked list operations.
- Examples involving AVL tree structures.
- We also investigate the benefits of immutability guarantees for allowing more flexible handling of aliasing, as well as more precise and concise specifications. Our approach supports finer levels of control that can localize and mark parts of a data structure as being immutable through the use of selective annotations on predicate and data declarations. Additionally, we support either partial or total immutability on each predicate. By using such annotations to encode immutability guarantees, we expect to obtain better specifications that can more accurately describe the intentions, as well as prohibitions, of the method. Ultimately, our goal is improving the precision of the verification process, as well as making the specifications more readable, more precise and as an enforceable program documentation.
- Conventional specifications typically have a flat structure that is based primarily on the underlying logic. Such specifications lack structures that could have provided better guidance to the verification process. In this demo we will also investigate three new structures to our specification framework for separation logic to achieve a *more precise* and *better guided* verification for pointer-based programs. The newly introduced structures empower users with more control over the verification process in the following ways: (i) case analysis can be invoked to take advantage of disjointedness conditions in the logic. (ii) early, as opposed to late, instantiation can minimize on the use of existential quantification. (iii) formulae that are staged provide better reuse of the verification process.
- Examples of entailments of separation heap constraints using SLEEK.

3. About the speaker

Cristina David is a PhD student at National University of Singapore, working under the supervision of Wei-Ngan Chin. During her studies, she participated in developing the HIP and SLEEK systems, and co-authored a few of the papers based on these systems [3, 4, 6].

References

[1] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties. In *ICECCS*, pages 307–320, 2007.

[2] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Multiple pre/post specifications for heap-manipulating methods. In *HASE*, pages 357–364, 2007.

[3] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular oo verification with separation logic. In *POPL*, pages 87–99, 2008.

[4] Cristina David, Cristian Gherghina, and Wei-Ngan Chin. Translation and optimization for a core calculus with exceptions. In *PEPM*, pages 41–50, 2009.

[5] Cristian Gherghina, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Structured specifications for better verification of heap-manipulating programs. In *FM*, 2011.

[6] H. H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, Nice, France, January 2007.

[7] Huu Hai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In *CAV*, pages 355–369, 2008.