HIPimm: Verifying Granular Immutability Guarantees

Andreea Costea Asankhaya Sharma
Department of Computer Science
National University of Singapore
{andreeac, asankhs}@comp.nus.edu.sg

Cristina David
Department of Computer Science
University of Oxford
cristina.david@cs.ox.ac.uk

Abstract

HIPimm, an extension of the HIP/SLEEK automatic verification system, offers immutability guarantees on top of ensuring functional correctness for heap-manipulating programs. The extra capability of HIPimm, as compared to its precursor, is the ability to reason about immutability guarantees in a granular manner. For this purpose, we enhance the specification language with immutability annotations which provide the means to assert whether the annotated heap can be mutated or whether is inaccessible. As part of user defined predicates, these annotations are integrated at the data field level offering granular immutability guarantees. An immediate result of this new functionality is a finer level of precision in the verification process of programs involving heap data structures. That is, we enable the verification of program properties such as preservation of data structures shapes and/or values, flexible aliases, and information leakage prevention.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords specification, granular immutability, heap

1. Introduction

Heap-manipulating programs are notoriously hard to verify, and thus it is crucial to have automatic tools capable of assisting the programmer in their verification. For this purpose, we introduce HIPimm, a separation logic [14, 15] based automated verification system for a simple imperative language, aimed to modularly verify the functional correctness of heap manipulating programs. What differentiates HIPimm from its precursor, the HIP/SLEEK verification system [13], is the emphasis on *immutability guarantees*. Such guarantees over heap allocated data structures (or *data nodes*) ensure that certain attributes (or *fields*) of the corresponding memory locations do not change.

Special attention is given in HIPimm to allowing a *fine-grained granularity of the immutability annotations*. Thus, HIPimm allows associating different annotations to each field of a data node (please see Fig 2 for an example of a data declaration and its usage). We refer to this capability as the *field-level immutability guarantee*, and stress that it is more precise than *node-level guarantee*, which associates the same immutability annotation with the entire data node, i.e., all the fields of a data node have identical annotations (this was the case in HIP/SLEEK).

For illustration, consider the data declaration in Fig 2. When using the corresponding data node in a specification, we adopt the notation $\texttt{x::node}\langle \texttt{v}, \texttt{p}\rangle$ denoting the fact that x is a pointer to the data node, and v and p are variables corresponding to the two fields, `val` and `next`, respectively. Each of these two fields may be specified as *mutable*, *immutable* or *absent* through the following annotations:

- @M means the associated field may be mutated, e.g., if `val` is mutable, it can be updated via the field dereference `x.val`;
- @L denotes an immutable field, meaning that it can only be read, but not updated;
- @A implies that the field cannot be accessed at all neither for reading, nor for writing (this annotation may be used to prevent information leakage as shown in Sec3.4).

With field-level immutability, we can describe scenarios such as $\texttt{x::node}\langle \texttt{v@M}, \texttt{p@L}\rangle$, i.e., only the value field is allowed to change, whereas the reference one is immutable, or $\texttt{x::node}\langle \texttt{v@A}, \texttt{p@L}\rangle$, i.e., the value field cannot be accessed at all, while the reference field can only be read.

In this paper, we report on two encouraging consequences of the fine-grained immutability granularity:

- *improved flexibility* of the specifications, allowing for more detailed properties to be specified.
- *better precision* of the verification process itself, which enabled the successful verification of properties such as preservation of data structures shapes and/or values and information leakage prevention (detailed in Sec 3).

All the examples can be tried through the web based interface of our tool available at the following url: `http://loris-7.ddns.comp.nus.edu.sg/~project/HIPimm/`

2. Overview

2.1 HIPimm architecture

HIPimm is a separation logic based automatic verifier which ensures the safety of heap-manipulating programs. Adopting a Hoare-style program verification, each method/loop is accessorized with a set of pre- and postconditions. The expressivity of its specification mechanism is shaped around features such as support for user-definable predicates, means to handle bag operations [3], structured specifications [8] and fine-grained immutability guarantees.

As shown in Fig 1, HIPimm includes two sub-systems, both of which have been enhanced to deal with immutability guarantees at the field-level: (i) an automated verification system for a simple

PEPM '14, January 20–21, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2543728.2543743

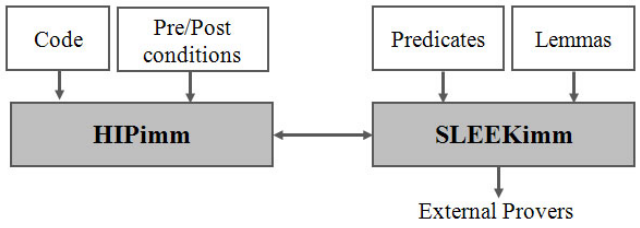

Figure 1. Architecture of HIPimm

```
/* DATA DECLARATION : */
data node{int val; node next; }

/* PREDICATE DECLARATIONS : */
/* allowed in both HIP/SLEEK and HIPimm */
pred ll⟨n⟩ ≡ (self=null ∧ n=0) ∨
   (self::node⟨_, p⟩ * p::ll⟨n−1⟩)
   inv n≥0.

/* only supported by HIPimm */
pred llA⟨n, a1, a2⟩ ≡ (self=null ∧ n=0) ∨
   (self::node⟨_@a1, p@a2⟩ * p::llA⟨n−1, a1, a2⟩)
   inv n≥0.
```

Figure 2. Snippet with data and predicate declarations for capturing linked lists

imperative language (HIPimm) and (ii) a fully automatic separation logic prover with frame inferring capabilities (SLEEKimm).

Based on the given set of pre- and postconditions, HIPimm constructs a set of separation logic proof obligations in the form of separation logic entailments, which are then sent to SLEEKimm. SLEEKimm and the entailment proving part will be later discussed in Sec 4. Our tool supports user defined predicates to express various data structures and user specified lemmas to enable proof search during entailment.

2.2 A First Glimpse of HIPimm

To understand how the pre-post specifications are used in the verification context, let us explore some very basic examples first. For that, we consider the declarations in Fig.2 which are used by most of the examples provided in this subsection and subsequent ones.

As already mentioned in Sec 1, the data declaration in Fig.2 represents a data node comprising of a data field `val` and a pointer field `next` for pointing to the next node in a singly linked list. A linked-list of length `n` ending with `null`, is recursively defined by the shape predicate `ll`, which, besides the size parameter `n`, also uses an implicit parameter `self` to capture the initial pointer into the specified heap structure. In the base case, `self` points to an empty list, while in the inductive case, it points to the head of a list consisting of at least one element. To ensure that all the nodes in the list reside in disjoint heaps, the inductive case uses the separating conjunction $*$ from separation logic to link the lists' head with its tail. `llA` describes a similar heap structure, with the essential difference that now the data fields are associated to the `a1` and `a2` immutability annotations, which are set as explicit parameters.

Underscore _ is used instead of certain data fields, to denote an anonymous value. Free variables in the body of the shape definition are treated as being existentially quantified, e.g., `p`.

The trivial examples in Fig.3 are meant to illustrate the key difference between a verification process guided by a specification without immutability annotations (Fig.3 (a)), and one which enhances precision through field-level immutability guarantees (Fig.3 (b)).

```
void set(node x, int v)
 requires  x::node⟨_, p⟩
 ensures   x::node⟨v, p⟩;
{x.val=v; }

int get(node x)
 requires  x::node⟨v, p⟩
 ensures   x::node⟨v, p⟩ ∧
        res = v;
{return x.val; }
```

(a)

```
void set(node x, int v)
 requires  x::node⟨_, p@A⟩
 ensures   x::node⟨v, p@A⟩;
{x.val=v; }

int get(node x)
 requires  x::node⟨v@L, p@A⟩
 ensures   res = v;

{return x.val; }
```

(b)

Figure 3. `set` and `get` methods specified (a) without the use of annotations and (b) using immutability annotations

For the `set` method, the verification can be guided to check that for a given node `x`, out of its two fields, only the `val` field is being updated (Fig.3 (a)), while its `next` field remains the same. A stronger specification is obtained by annotating the `next` field with `@A`, i.e, the annotated `next` field cannot be accessed at all inside the method's body (Fig.3 (b)).

A more concise and precise specification (Fig.3 (b)) is obtained with the usage of immutability annotations to specify `get`, a method which is not supposed to mutate, but only to read and return the field `val` of parameter `x`. If its precondition is satisfied, that is, the `val` field is read-only and `next` is not accessed within this method, then there is no need for the postcondition to specify anything about the state of the heap pointed by `x` as it did not change.

Next, we consider a method which computes the length of a linked list. The implementation of the `length` method, whether using a recursive call, or a loop, is standard, so we omit to give its full code here. It is worth mentioning that `length` should perform no mutation on the given list. In Fig.4 (a), the specification of `length` says that "assuming `x` points to a linked list of size `n` at the beginning of this method, it will point to a list of the same size at the end of the method and the output is the length of the list". If it is important to verify that the list also preserves the same set of values and addresses, then there is a need for a stronger specification. With the usage of a new predicate in (b), which also captures the bag of address and values inside a list, the aforementioned problem is solved. An even stronger information is captured in (c). With the help of immutability annotations attached to the nodes of the linked list pointed by `x`, it is now possible to specify that the list also preserves the same sequence of addresses and that the `val` field is not needed for the computation of the length.

3. Applicability

3.1 Shape Immutability

Sec 2.2 presented a convenient way to annotate `set`, a method updating the `val` field of the node received as argument without accessing its `next` pointer. Possible scenarios where such a method would be useful include the marking of each visited node of a data structure during a traversal, or an in-situ sorting where the value fields get swapped around, whereas the pointer fields are left unchanged.

One common property of these algorithms is the fact that they mutate the value fields of a certain data structure, while leaving the pointer fields unchanged. We refer to this type of immutability as *shape immutability*, i.e., the links of the data structure are invariant. HIPimm makes it possible for its users to specify and verify such shape immutability constraints.

```
int length(node x)
 requires  x::ll⟨n⟩
 ensures   x::ll⟨n⟩ ∧res = n;
{...}
```
(a)

```
int length(node x)
 requires  x::llA⟨n, @A, @L⟩
 ensures   res = n;
{...}
```
(c)

```
pred llB⟨n, V, P⟩ ≡ (self=null ∧ n=0 ∧ V={} ∧ P={}) ∨
          (self::node⟨v, p⟩ * p::llB⟨n−1, Vr, Pr⟩ ∧
                    V=union(v, Vr) ∧ P=union(p, Pr))
   inv n≥0.

int length(node x)
 requires  x::llB⟨n, V, P⟩
 ensures   x::llB⟨n, V, P⟩ ∧ res = n;
{...}
```
(b)

Figure 4. Specifying the `length` method: (a) without annotations, (b) without annotations, but using the `llB` predicate whose usage guarantees that the same sets of values and addresses are preserved and (c) using immutability annotations to additionally guarantee the preservation of addresses sequence

3.2 Data Immutability

As opposed to the scenarios presented in the previous subsection, there are operations on the heap structures that modify the shape of the structure without affecting its data fields. This type of immutability referred to as *data immutability* is complementary to shape immutability. Operations exhibiting data immutability include list reversing, list partition, insertion of a node in a sorted linked list, etc.

An application where it is important to offer data immutability guarantees comes in the form of garbage collection mechanisms. In particular, the well-known Deutsch-Schorr-Waite marking algorithm for binary trees, marks all reachable nodes so that the unreachable ones can be collected and re-used. The algorithm uses a complicated pointer manipulation scheme mixing two different data structures, without awareness on the informational content of the tree's nodes. Thus, it is advisable to have a specification of this algorithm which guarantees that the verified garbage collector supports pointer manipulation without having any access over the data stored inside these nodes. The verification of this algorithm is known to be a complex task, and existing formalisms mostly focus on functional correctness. To tackle this problem, we not only prove functional correctness, but on top of that we also offer immutability guarantees by specifying that the informational content is absent during tree marking (we annotate data fields with `@A`) while the pointers can freely change according to the Schorr-Waite algorithm (pointers are marked as `@M`).

3.3 Flexible Aliasing

Next, we show how immutability annotations allow HIPimm to reason about conjunctive heap formulas with arbitrary aliasing. In Fig.5, the conjunctive operator over heap formulas, ∧, indicates that the nodes pointed by `x` and `y` are may aliases. When first introduced in our specification logic, this operator could only involve immutable data structures [5]. However, field-level immutability guarantees enable HIPimm to relax the immutability requirement by allowing mutation inside may aliased nodes, under the strict condition that their fields are updated in a mutually exclusive manner.

For illustration, in the precondition of `sum`, the `next` field of `x` can be mutated because its `next` field counterpart in `y` is guaranteed to be absent. Note that, in the postcodition of `sum`, the value field of

```
int sum(node x, node y)
 requires  x::node⟨a@L, _@M⟩ ∧ y::node⟨b@L, _@A⟩
 ensures   res=a + b ∧ x::node⟨a@A, _@M⟩;
{return x.val+y.val; }
```

Figure 5. Summation of the values stored in may-aliases nodes

```
data user{char * name; char * pass; }

int login(ref user u, char * psswd)
 requires  u::user⟨n, p⟩
 ensures   u′::user⟨n, p@A⟩;
{ if(check_pass(u, psswd) != 0) return 1;
  else return 0;
}

int check_pass(ref user u, char * psswd)
 requires  u::user⟨n, p⟩
 ensures   u′::user⟨n, p@A⟩;
{ return strcmp(psswd, u.pass); }
```

Figure 6. Adaptation of Unix password-authentication of Cyrus' Internet Message Support Protocol daemon

the node pointed by `x` is marked with `@A`. This is due to the fact that a caller always retains ownership of the fields marked as immutable in the callee's precondition. Thus, as ownership does not need to be returned to the caller, those fields are marked as absent in the callee's postcondition.

Although currently HIPimm only allows this flexible aliasing scheme between data nodes, we work on lifting it to heap predicates, e.g., expressing may aliasing between two lists. Our conjecture is that this enhancement will allow reasoning in scenarios involving overlaid data structures. These data structures are frequently identified in low-level code which requires accessing the same set of nodes with respect to different criteria. For example, a certain part of the heap may be accessible either by traversing a list, or a tree, depending on certain optimization principles.

3.4 Information Leakage

We claim that field-level immutability annotations can be successfully used for preventing information leakage for a series of security related applications. They are particularly useful in scenarios where certain data structure containing both confidential as well as public information are involved in computations outside a secured code. Annotating the confidential fields with @A triggers verification failures for those methods attempting to do read or write operations on the annotated heap.

The code in Fig 6 is an adaptation of the Unix password-authentication of Cyrus' Internet Message Support Protocol daemon, known to represent a security risk due to the possible leakage of user's confidential information [10]. It illustrates a scenario where @A proved to be a simple and elegant solution towards the prevention of such risks. The sensitive information which needs to be protected in this case is the users' password. Once it was checked to match against the input string `psswd` using `check_pass` method, the verifier triggers a failure for any attempt to access the users' password field by dereferencing `u` or subsequently created aliases. However, the verifier offers no guarantees for aliases created prior to calling `check_pass`.

4. Technical Background

Program verification is typically formalised using Hoare triples of form $\{pre\}code\{post\}$, where pre and post are the initial and final states of the program code in some logic (separation logic in

our case). We use P to denote the program being checked. With pre/post conditions declared for each method in P, we can now apply modular verification to its body using Hoare-style triples $\vdash \{\Delta_1\}\, e\, \{\Delta_2\}$, where e stands for the verified expression. These are forward verification rules as we expect Δ_1 to be given before computing Δ_2.

There are several places during forward verification when proof obligations are generated:

- at each method call site, the program state must entail the precondition of the method being called.
- at the end of each method verification, the program state must entail the method's postcondition.
- at the point of each field access/update, a heap entailment is generated in order to check that the memory location being accessed is not null.

In order to discharge these proof obligations given as separation logic entailments, SLEEKimm in Fig 1 is being called. Due to space constraints we just give the overview of the entailment procedure. Full details of both the entailment and verification rules are provided in [4]. Given formulas Φ_1 and Φ_2, our entailment prover checks if Φ_1 entails Φ_2, that is if in any heap satisfying Φ_1, we can find a sub-heap satisfying Φ_2. Formally, the entailment relation is written as follows: $\Phi_1 \vdash^{\kappa}_{V} \Phi_2 * \Phi_R$, where κ is the history of nodes from the antecedent which have been used to match nodes from the consequent and, V is the list of existentially quantified variables from the consequent. Note that κ and V are derived. The entailment checking procedure is initially invoked with $\kappa = \texttt{emp}$ and $V = \emptyset$. Besides determining if the entailment relation holds, our entailment prover also infers the residual heap of the entailment, that is a formula Φ_R such that $\Phi_1 = \Phi_2 * \Phi_R$.

The entailment rules are extended to handle separation logic formulas with annotated heap structures. For brevity reasons, we only explain the intuition behind discharging constraints related to immutability annotations:

- having defined a subtyping relation between the immutability annotations, i.e., $@M <: @L <: @A$, this must always hold between the left hand side and its corresponding right hand side annotation in an entailment;
- for a pair of matching nodes, there is always an annotation residue left on the left hand side of the entailment, after consuming the right hand side heap:

$$residue(@a1,\, @a2) = \begin{cases} @a1 & \text{if @a2=@A or @a2=@L;} \\ @A & \text{if @a2=@M.} \end{cases}$$

5. Discussions

Fine grained immutability guarantees can provide useful information for various compiler optimizations [16], safe parallelism [9] and ease formal reasoning about programs [18]. Similar to [5] we have focused on adding immutability annotations to the specification logic. When compared to [5] we support more granular immutability as the annotations can be used on field level, while in [5] they can only be used at object level. Our solution supports modular specification and verification of data immutability, shape immutability, flexible aliasing and information leakage. Granular immutability annotations make the verification more precise and concise with finer control over access to resources. The following table shows the results of some preliminary experiments we conducted using HIPimm for verification.

Program	*LOC*	*Time [secs]*	$@L[\%]$	$@A[\%]$
Big Int	204	7.71	65.6	0
Get Set	62	0.12	25	50
Info Leakage	29	0.16	0	66.7
List Insert	30	0.26	100	0
List Length	17	0.14	50	50
Schorr Waite List	37	13.49	0	100
Sorted List	118	1.03	83.3	16.7

The first column in the table shows the list of the programs. The list includes all the programs presented in paper. The second column shows the total number of lines of code in the program. We have currently applied the technique for verification of programs up to 200 lines of code. The third column shows the time taken in seconds for HIPimm to verify the corresponding program. In our benchmark, all the examples are verified within seconds; moreover, there is no significant increase in time taken when compared to the same example without immutability annotations. In order to evaluate the usefulness of the granular immutability annotations, we show in column 4 and 5 the percentage of specifications using the @L and @A annotation respectively. We argue that the values in the latter two columns, indicate the percentage of the verified program specifications which are more precise (offer more information about the heap structure) than in the cases where, for the same set of specifications, the annotations are being ignored. In order to reach the same level of precision, thus being able to verify the same set of properties without using the immutability annotations, the specification language should provide support for additional structures (e.g. such as bags or sequences to show shape preservation), which further demands the usage of dedicated theorem provers in supporting these structures. Our approach, on the other hand, discharges proofs which can be handled by most of the automatic theorem provers (e.g. Omega[11], Redlog[7], Z3[6]).

Currently in HIPimm the immutability annotations are provided by the user. In order to alleviate the burden of writing annotations we would like to provide an inference mechanism in future. The inference would be based on the program under consideration and would infer the most general immutability annotations under which the program can be verified using HIPimm. In addition we are also looking at ways to enable transfer of mutability and immutability in heap, specially in the presence of aliasing. This will enable verification of a wider range of programs that use sharing in data structures.

Comparison with fractional permissions. Ever since Boyland introduced them in order to check computation interference, fractional permissions [2] have been proved useful for a broad range of applications. These applications include, but are not limited to, type systems for safe memory deallocation [17], implicit dynamic frames based verification of multi-threaded programs [12], etc.

In separation logic, fractional models based on permissions [1] have been proposed to provide shared read and exclusive write access to the heap. Although useful for the verification of concurrent programs, we argue that, in a sequential setting, the fractional models can be efficiently replaced by the granular immutability guarantees mechanism. We claim the following advantages of our proposed solution over the fractional permission model: the constraints generated due to the usage of the immutability guarantees can be handled by most of existing theorem provers (there is no extra need for fraction capabilities inside the verifier, or inside the prover); we allow @A in our proposed system, which proved useful for the information leakage examples; simplified ownership tracking as in the case of @L and @A the ownership does not need to return to the caller.

Moreover, fraction based shared read permissions no longer preserve disjointness for $*$, while the usage of immutability annotations still guarantees that the separation logic conjunction expresses disjoint heaps. Lastly, it was already shown that immutability annotations help reduce the size of specifications and preserve cut-points into the data structures [5].

Acknowledgments

We would like to thank the reviewers of PEPM'14 for their useful feedback.

References

[1] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.

[2] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer Berlin Heidelberg, 2003.

[3] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

[4] C. David. *Enhanced Specification Expressivity for Verification with Separation Logic*. PhD thesis, Department of Computer Science, National University of Singapore, 2011.

[5] Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. In *OOPSLA*, 2011.

[6] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[7] A. Dolzmann and Thomas Sturm. Redlog: computer algebra meets computer logic. *SIGSAM Bull.*, 31:2–9, June 1997.

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

[9] Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA*, pages 21–40, 2012.

[10] Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 261–269. ACM, 2010.

[11] P. Kelly, V. Maslov, W. Pugh, and et al. *The Omega Library Version 1.1.0 Interface Guide*, 1996.

[12] K. Rustan Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. pages 251–266, January 2007.

[14] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), 1999.

[15] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. 2002.

[16] Alexandru Doru Sălcianu. *Pointer analysis for Java programs: Novel techniques and applications*. PhD thesis, Massachusetts Institute of Technology, 2006.

[17] Kohei Suenaga and Naoki Kobayashi. Fractional ownerships for safe memory deallocation. In *APLAS*, pages 128–143, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, et al. Object and reference immutability using java generics. In *ESEC FSE*, pages 75–84, 2007.