SMT-Based Modular Analysis of Sequential Systems Code

Shuvendu K. Lahiri

Microsoft Research

Abstract. In this paper, we describe a few challenges that accompany SMT-based precise verification of systems code (device drivers, file systems) written in low-level languages such as C/C++. First, the presence of pointer arithmetic and untrusted casts make type checking difficult; we show how to formalize C type safety checking and exploit the types for disambiguation of addresses in the heap. Second, the prevalence of explicit manipulation of pointers in data structures using dereference and address arithmetic precludes abstract reasoning about data structures. We provide an expressive and efficient theory for reasoning about linked lists, which comprise most data structures in systems code. We discuss extensions to standard SMT solvers to tackle these issues in the context of the HAVOC verifier.

1 Introduction

A majority of systems software (device drivers, file systems etc.) continue to be written in low-level languages such as C and C++. These languages offer developers the potential to obtain raw performance by low-level control over object layout and object management. However, the gains come at the expense of lack of type and memory safety, lack of modularity and large bloated monolithic components with several hundred thousands of lines. These factors impose additional challenges for the analysis of systems code, in addition to those posed by higher level languages such as Java and C#.

In this work, we discuss our experience with applying *satisfiability modulo theories* (SMT) solvers [7] for *predictable* analysis of systems software, namely in the context of the HAVOC verifier [4]. Predictable analysis constitutes *precise* and *efficient* checking of assertions across loop-free and call-free program fragments.

- By precision, we denote an assertion logic (for writing pre/post conditions, loop invariants) expressive enough to be closed under *weakest liberal preconditions* [3] across a bounded code fragment.
- By efficient, we imply the complexity of the decision problem for the assertion logic. Since many efficiently solvable SMT logics (Boolean satisfiability (SAT), integer linear arithmetic, theory of arrays) have NP-complete decision problems, we consider logics with NP-complete decision problems to be efficiently decided in practice.

The use of such predictable verifiers can be extended to whole programs by combining them with user-supplied or automatically inferred procedure contracts, and loop invariants. We do not focus on the issue of inferring such annotations in this work.

We focus on two main aspects of analysis of systems software in this paper:

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 21–27, 2011.

1. *Lack of type-safety*: We discuss the challenges in checking type-safety of these low-level programs and the implications for modular property checking. We show how to formalize the type-safety of C programs as state assertions, and augmenting SMT solvers with a theory of low-level C types. Details of this work can be found in an earlier paper [2].
2. *Low-level lists*: Linked lists form a majority of linked data structures in systems code; we show the difficulty of employing abstractions on top of such lists given explicit manipulation of addresses and links. We present an SMT theory of lists that allows stating many interesting invariants for code manipulating such lists. Details of this work can be found in the following works [4,6].

In the next few sections, we briefly summarize the issues and the solutions in a semi-formal fashion to enable quick reading. Interested readers are encouraged to refer to the detailed works for more elaborate treatment on each topic.

2 Basic Memory Model

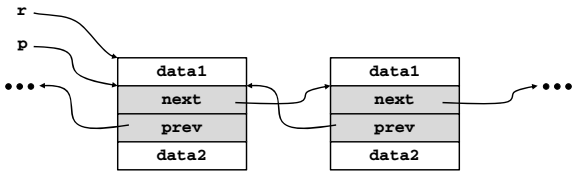
For the sake of illustration in this paper, we will assume a simplified subset of C programs where the only primitive type consists of *integers* int. Addresses and integer values are treated as integers. We ignore the issue of sub-word access, where an integer may be split up into 4 characters, or 2 shorts. The state of the heap is modeled using an mutable array $\mathsf{Mem} : \mathsf{int} \rightarrow \mathsf{int}$ that maps an address to a value or another address.

Variables whose addresses are taken (using $\&$) and structures are allocated on the heap. Read from a pointer $*e$ is modeled as $\mathsf{Mem}[||e||]$, a lookup into the array Mem at the location corresponding to the value of the C expression e (denoted by $||.||$). Similarly a write $*e = x$ is modeled as $\mathsf{Mem}[||e||] := ||x||$, an update to Mem. Field accesses $e \rightarrow f$ are compiled as pointer accesses with a field offset, $*(e + \mathrm{Offset}(f))$, where $\mathrm{Offset}(f)$ is the (static) offset of the field f in the structure pointed to by e. The different operations (arithmetic, relational) are translated as appropriate operations on integers.

3 Types

Consider the method `init_record` in Figure 1. The parameter `p` is a pointer to a `list` structure. In most systems program, it is common to use a structure similar to `list` to define a generic doubly-linked list. This structure can be embedded in any structure (such as `record`) as a field to create a list of such structures. The programming paradigm is uniformly used as the list manipulation routines (insertion, deletion, test for emptiness) are defined once on the `list` structure. A structure can have multiple fields of type `list` if the same structure can be part of multiple lists.

However, this also poses several challenges for type-safety as can be seen from the example. First, the type of the enclosing structure is not evident from the signature of the parameter of `init_record`. Second, programs need to use a macro like `CONTAINING_RECORD` that obtains the pointer to the enclosing structure from the address of an internal field. This involves non-trivial pointer arithmetic and type casts, the safety of which is not easy to justify.




```
struct list   { list *next; list *prev; }
struct record { int data1; list node; int data2; }

#define CONTAINING_RECORD(x, T, f) ((T *)((int)(x) - (int)(&((T *)0)->f)))

void init_record(list *p) {
    record *r = CONTAINING_RECORD(p, record, node);
    r->data2 = 42;
}

void init_all_records(list *p) {
  while (p != NULL) {
    init_record(p);
    p = p->next;
  }
}
```

Fig. 1. Example C code. The diagram shows two `record` structures in a linked list, with the embedded `list` shown in gray.

To create a sound analysis, one can completely disregard the types and field names in the program. However, this poses two main issues:

- The presence of types and checking for well-typed programs may guarantee the absence of some class of runtime memory safety errors (accesses to invalid regions in memory).
- Types also provide for disambiguation between different parts of the heap, where a read/write to pointers of one type cannot affect the values in other types/fields. For instance, any reasonable program analysis will need to establish that the value in `data1` field in any structure is not affected by `init_record`.

3.1 Formalizing Types

We address these problems by formalizing types as predicates over the program state along with an explicit type-safety invariant [2]. We introduce a map $\mathsf{Type} : \mathsf{int} \rightarrow$ type that maps each allocated heap location to a type, and two predicates Match and HasType. The Match predicate lifts Type to types that span multiple addresses. Formally, for address a and type t, $\mathsf{Match}(a, t)$ holds if and only if the Type map starting at address a matches the type t. The HasType predicate gives the meaning of a type. For a word-sized value v and a word-sized type t, $\mathsf{HasType}(v, t)$ holds if and only if the value v has type t.

The definitions of Match and HasType are given in Figure 2. For Match, the definitions are straightforward: if a given type is a word-sized type (int or $\mathsf{Ptr}(t)$ where Ptr is a pointer type constructor), we check Type at the appropriate address, and for

Definitions for Int

$$\mathsf{Match}(a, \mathsf{Int}) \triangleq \mathsf{Type}[a] = \mathsf{Int} \quad (A)$$
$$\mathsf{HasType}(v, \mathsf{Int}) \triangleq \mathsf{true} \quad (B)$$

Definitions for $\mathsf{Ptr}(t)$

$$\mathsf{Match}(a, \mathsf{Ptr}(t)) \triangleq \mathsf{Type}[a] = \mathsf{Ptr}(t) \quad (C)$$
$$\mathsf{HasType}(v, \mathsf{Ptr}(t)) \triangleq v = 0 \vee (v > 0 \wedge \mathsf{Match}(v, t)) \quad (D)$$

Definitions for $\texttt{type t} = \{f_1 : \sigma_1; \ldots; f_n : \sigma_n\}$

$$\mathsf{Match}(a, \mathsf{T}) \triangleq \bigwedge_i \mathsf{Match}(a + \mathrm{Offset}(f_i), T(\sigma_i)) \quad (E)$$

Fig. 2. Definition of HasType and Match for a, v of sort int and t of sort type

structure types, we apply Match inductively to each field. For HasType, we only need definitions for word-sized types. For integers, we allow all values to be of integer type, and for pointers, we allow either zero (the null pointer) or a positive address such that the allocation state (as given by Match) matches the pointer's base type. HasType is the core of our technique, since it explicitly defines the correspondence between values and types.

Now that we have defined HasType, we can state our type safety invariant for the heap:

$$\forall a : \mathsf{int}.\mathsf{HasType}(\mathsf{Mem}[a], \mathsf{Type}[a])$$

In other words, for all addresses a in the heap, the value at $\mathsf{Mem}[a]$ must correspond to the type at $\mathsf{Type}[a]$ according to the HasType axioms. Our translation enforces this invariant at all program points, including preconditions and postconditions of each procedure. We have thus reduced the problem of type safety checking to checking assertions in a program.

The presence of the Type also allows us to distinguish between pointers of different types. In fact, we provide a refinement of the scheme described here to allow names of word-sized fields in the range of Type. This allows to establish that writes to the `data2` field in `init_record` does not affect the `data1` field of any other objects.

3.2 SMT Theory for Types

By using standard verification condition generation [1], the checking of the type safety assertions in a program reduces to checking a ground formula. The formula involves the application of Mem, Type, Match and HasType predicates, in addition to arithmetic symbols. The main challenge is to find an assignment that respects the definition of Match and HasType from Figure 2 and satisfies the type safety assertion; all of these can be expressed as quantified background axioms. We show that it suffices to instantiate these quantifiers at a small number of terms (with at most quadratic blowup) to produce an equisatisfiable ground formula, where the predicates Match and HasType are completely uninterpreted. This ensures that the type safety can be checked for low-level C programs in logics with NP-complete decision problem.

4 Low-Level Data Structures

Consider the procedure `init_record` in Figure 1. To ensure type safety, we need the following precondition for this procedure:

$$\mathsf{HasType}(p-1, \mathsf{Ptr}(\mathsf{Record}))$$

to indicate that $p-1$ is a pointer to a `record` structure. However, to prove this precondition at the call site in `init_all_records`, we need a loop invariant that asserts that for any pointer $x \in \{p, *(p+0), *(*(p+0)+0), \ldots\}$, $x-1 \neq \texttt{null}$ and $\mathsf{HasType}(x-1, \mathsf{Ptr}(\mathsf{Record}))$ holds. This set represents the set of pointers reachable from p using the `next` field. Similarly, the set of pointers obtained by following the `prev` field is $\{p, *(p+1), *(*(p+1)+1), \ldots\}$, because the `prev` field is at an offset of 1 inside the `list` structure.

Unlike most high level languages, where the internal details of a list are hidden from the clients, most systems program explicitly use the fields `next` and `prev` to iterate over lists. The lists are not well encapsulated and clients can have multiple pointers inside a list. In addition, the presence of pointer arithmetic (as described above) to obtain enclosing objects make it difficult to reason about such lists abstractly.

4.1 Reachability Predicate

We define a logic that can describe properties of a set of pointers in a list [4]. The main idea is to introduce a set constructor $Btwn : (\mathsf{int} \rightarrow \mathsf{int}) \times \mathsf{int} \times \mathsf{int} \rightarrow 2^{\mathsf{int}}$ that takes a map, and two integer addresses such that $Btwn(f, x, y)$ returns the pointers $\{x, f[x], f[f[x]], \ldots, y\}$ (between x and y) when x reaches y by dereferencing f, or $\{\}$ otherwise. The salient points of the assertion logic are:

1. The logic can model singly and doubly linked lists, including cyclic lists.
2. Formulas in this logic are closed under the weakest liberal precondition transformer with respect to the statements in a language with updates to the maps (corresponding to updating fields such as `next` or `prev`). In other words, the formula $Btwn(f[a := b], x, y)(z)$ can be expressed in terms of $Btwn(f, w_1, w_2)(w_3)$, where $w_i \in \{a, b, x, y\}$. This allows for precise reasoning for a loop-free and call-free fragment of code annotated with assertions in this logic.
3. The logic provides for expressing quantified facts about all elements in a list such as
$$\forall x : \mathsf{int} \in Btwn(f, a, b) \;::\; \phi(x)$$
as long as $\phi(x)$ maintains a *sort restriction* [4]. Although the logic can express a variety of useful invariants for lists (initialization, sortedness, uniqueness), it restricts the use of terms such as $f[x]$ inside $\phi(x)$. Intuitively, this may result in generating an unbounded set of terms of a list $x, f[x], f[f[x]], \ldots$ when instantiating the quantified assertions.
4. The decision problem for the resulting logic (when combined with other SMT logics such as arrays, arithmetic, equality) still remains NP-complete. We encode the decision procedure as a set of rewrite rules using *triggers* supported for quantifier reasoning in SMT solvers.

4.2 Modeling Low-Level Lists

As described earlier, elements in most lists in systems programs are obtained by first incrementing pointers by a constant offset before dereferencing the heap. For example, the list of pointers reachable through the `prev` fields are

$$\{p, \mathsf{Mem}[p+1], \mathsf{Mem}[\mathsf{Mem}[p+1]+1], \ldots, \texttt{null}\}$$

In HAVOC, we model this list as $Btwn(shift_1(\mathsf{Mem}), p, \texttt{null})$, where $shift_c : (\mathsf{int} \rightarrow \mathsf{int}) \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$ satisfies the following properties:

$$\forall x : \mathsf{int} :: shift_c(f)[x] = f[x+c]$$
$$\forall x : \mathsf{int}, v : \mathsf{int} :: shift_c(f[x+c := v]) = shift_c(f)[x := v]$$

The first axiom states that the content of $shift_c(f)$ is shifted by c with respect to the contents of f. The second axiom eliminates updates $f[x+c := v]$ nested inside $shift$. These axioms are carefully guarded by triggers such that they fire only for updates to the appropriate linking field such as `next` or `prev`. These extensions prevent us from guaranteeing a decision procedure for assertions about lists in C programs. However, we have observed very few unpredictable behavior in practice, even when reasoning about lists in programs several thousand lines large [6].

5 Other Challenges

In this paper, we have highlighted two issues that make precise verification difficult for systems programs. There are several other issues that pose interesting challenges. A few of them are:

- Since systems program rely on the user to perform object management, the problem of *double free* can lead to variety of undesired behaviors. When freeing objects over an unbounded data structures, we need to capture that pointers have unique references to them. These facts can often be captured as part of a module invariant, but may be broken temporarily inside the module. We describe *intra-module inference* to address similar issues in the context of large code bases [6].
- Since all the lists in a program share the `next` and `prev` fields, a modification to the field in one list may potential affect a completely disjoint list. It becomes more challenging when the procedure modifying the `next` field of a list does not have the other lists in scope. We discuss this imprecision in the context of *call invariants* [5] that offers a mechanism to leverage intraprocedural analysis to deal with interprocedural reasoning precisely.

References

1. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Program Analysis For Software Tools and Engineering (PASTE 2005), pp. 82–87 (2005)
2. Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. In: Principles of Programming Languages (POPL 2009), pp. 302–314 (2009)

3. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18, 453–457 (1975)
4. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: Principles of Programming Languages (POPL 2008), pp. 171–182 (2008)
5. Lahiri, S.K., Qadeer, S.: Call invariants. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 237–251. Springer, Heidelberg (2011)
6. Lahiri, S.K., Qadeer, S., Galeotti, J.P., Voung, J.W., Wies, T.: Intra-module inference. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 493–508. Springer, Heidelberg (2009)
7. Satisfiability Modulo Theories Library (SMT-LIB), `http://goedel.cs.uiowa.edu/smtlib/`