Heaps and Data Structures: A Challenge for Automated Provers

Sascha Böhme[1] and Michał Moskal[2]

[1] Technische Universität München, boehmes@in.tum.de
[2] Microsoft Research Redmond, michal.moskal@microsoft.com

Abstract. Software verification is one of the most prominent application areas for automatic reasoning systems, but their potential improvement is limited by shortage of good benchmarks. Current benchmarks are usually large but shallow, require decision procedures, or have soundness problems. In contrast, we propose a family of benchmarks in first-order logic with equality which is scalable, relatively simple to understand, yet closely resembles difficult verification conditions stemming from real-world C code. Based on this benchmark, we present a detailed comparison of different heap encodings using a number of SMT solvers and ATPs. Our results led to a performance gain of an order of magnitude for the C code verifier VCC.

1 Introduction

Among the applications of automatic provers, software verification is one of the most prominent and challenging driving forces [15]. Successful automatic code verifiers triggered and influenced the development of previously unseen improvements in reasoning systems. Yet, there is ample space for further advances, both in code verifiers and their underlying reasoning systems.

In practice, the development of automatic provers is usually driven by benchmarks (see [3,28] for large collections), taken from examples of existing applications. Current benchmarks stemming from the program verification world, large in size as they may be, are typically shallow (i.e., their solution touches only a small fraction of the search space), sometimes have consistency problems in their axiomatizations, and in many cases do not fall into the domain of an entire class of strong automatic provers. Shallowness is due to the prevalent use of model checkers in program verification, whose applications rarely cover deep problems. Inconsistencies in axiomatizations remain frequently unnoticed, due to the "reliably" incomplete quantifier reasoning of the applied provers (interestingly, the provers still correctly discover bugs in programs). Inapplicability of provers is caused by the lack of support for specific theories, especially for integer arithmetic, which is frequently required by benchmarks. Thus, there is a demand for new and complex benchmarks with the potential to initiate further improvements in the world of automatic provers and code verifiers.

We consider two classes of automatic provers relevant for verifying complex code. The first class is that of resolution-based provers such as E [27], SPASS [30] or Vampire [26], which we will refer to as atomatic theorem provers (ATPs). They are powerful in logical reasoning, but lack decision procedures for theories such as arithmetic. The second class is formed by satisifiability modulo theories (SMT) solvers, e.g., CVC3 [4], Yices [13] and Z3 [11]. They combine SAT solvers with decision procedures for various theories (e.g., equality and linear arithmetic) and quantifier instantiation heuristics.

1.1 The Quest for Fast Automated Provers

Automatic code verifiers demand an interactive feedback-driven style of verifying code: The user annotates the code and invokes the verifier, which, in turn, asks the automatic back-end prover to check the verification condition (VC) corresponding to the annotated code. In almost all cases, the initial annotation does not comply with the code and the user has to modify either of them and run the verifier again. Only after typically several repetitions of this feedback loop, the verifier will be satisfied. Especially for complex code (in particular in the domain of data structures), annotations are usually extensive and hence mostly added in small steps, requiring tens or hundreds of iterations.

Clearly, the response time of the code verifier (and, in particular, of the automatic prover running in the back-end) is of crucial importance to its adoption. In the Hypervisor verification project [18], for example, we have found verification times per entity (VCs corresponding to single C functions in that case) of over 30 seconds to be severely impeding productivity. Considerably longer return times made it virtually impossible to progress with verification. In fact, verification turnaround times (followed by fickleness of quantifier instantiation heuristics guiding the underlying SMT solver and difficulty in understanding verification errors) were reported as the main limiting factor during the Hypervisor verification project.

1.2 The Challenge

Verifying dynamic data structures is one of the challenges in code verification. This was also true for the aforementioned Hypervisor verification project (consisting of about $100,000$ lines of C code), which used the VCC [8] verifier. In part, this might be due to the heavy methodology imposed by the verifier. However, we found some evidence that Dafny [19], a verifier for a type-safe language, performs much better than VCC on equivalent benchmarks of complex data structures. We suspected the heap encoding to be the culprit here. To confirm our guess, we have developed a series of benchmark programs and tested them against several different heap encodings (Sect. 2), including versions of VCC's and Dafny's heap models. These initial benchmarks have shown the VCC model to be much less efficient, but have also shown surprising differences between superficially similar heap encodings. Some of these results carried over to prototypical re-implementations of the heap encoding in VCC, but others did not.

Consequently, we devised a benchmark family of multiply-linked lists (Sect. 3) as an archetypical example of a recursive data structure. We implemented and specified the benchmarks in Boogie [2], a minimal intermediate verification language used by Dafny and VCC, among other verifiers. The axiomatization required is tiny, giving good guarantees of soundness, and does not require arithmetic. We have also reimplemented the multiply-linked list benchmarks, or *multi-list* benchmarks for short, in VCC and Dafny (both of which have large background axiomatizations using arithmetic).

1.3 Contributions

We propose a new family of benchmarks for both automatic code verifiers and automatic provers, named multi-lists (Sect. 3). These benchmarks can be arbitrarily scaled in two dimensions (showing trends instead of only scattered data-points), chopped into pieces (one per assertion), and run against any of the six heap encodings we consider (none of which is novel, but see Sect. 2). This gives a wealth of benchmarks, which, we believe, will be found useful by the community and will foster improvements of automatic provers.

Using our benchmarks, we compare several ATPs and SMT solvers (Sect. 4). This is the first comprehensive comparison of how well different heap encodings perform on different reasoning systems. The experiments also show that the proposed benchmarks are relevant in that our results using the rather low-level Boogie system carry over to VCC and Dafny and, moreover, behave comparable to benchmarks of real-world data structures. Since the benchmarks respond, under changes of the heap encoding and options to the back-end prover (Z3 in our case), similarly in all considered code verifiers (Boogie, VCC and Dafny), we believe that our results also apply to other verification systems.

2 Heap Encodings

It is customary to represent the heap as a mapping from indices to values and to provide functions read() and write() for accessing and updating it. Below we summarize the read-over-write axioms to characterize these functions [7]. Here, as well as in the rest of the paper, we use the syntax of the Boogie [2] intermediate verification language. The predicate disjoint() abstracts from the disequality test between two indices.

```
axiom (∀ H, p, v • read(write(H, p, v), p) = v);
axiom (∀ H, p, q, v • disjoint(p, q) ⇒ read(write(H, p, v), q) = read(H, q));
```

Throughout this section, we assume a language with a two-tiered heap, i.e., where heap access and update require a pointer and a field. Note that this assumption is a generalization of plain pointer-indexed heaps, because a pseudo-field can always artificially be constructed, if necessary. Moreover, we use (unbounded) integers to represent values stored on the heap, an over-approximation of real memory. Our assumptions apply to many languages such as Java, C#, or Dafny, just to name a few. Applying such a model to C is also possible, but requires additional work (Sect. 2.1).

Two-tiered heaps can be modeled in (at least) the following six logically equivalent ways (we abbreviate each heap model name by a short Boogie representation and refer to heap models by their abbreviation later on):

Linear heap H[dot(p,f)] The heap is addressed by pointers only. It corresponds to the model sketched above with pointer disequality instantiating the disjointness predicate. Pointer-field pairs are coerced to pointers by means of the free constructor dot() axiomatized with its projections base() and field():

```
axiom (∀ p, f • base(dot(p,f)) = p ∧ field(dot(p,f)) = f);
axiom (∀ p • dot(base(p), field(p)) = p);
```

State-based linear heap H[dot2(p,f)] In some language formalizations (see Sect. 2.1 for more details), the above projections base() and field() only exist for certain pointers in certain states. We model that by making them dependent on the heap:

```
axiom (∀ H, p, f • base(H, dot2(p, f)) = p ∧ field(H, dot2(p, f)) = f);
axiom (∀ H, p • dot2(base(H, p), field(H, p)) = p);
```

Synchronous heap H[p,f] The heap is simultaneously accessed by pointer and field. Its axiomatization is as follows:

```
axiom (∀ H, p, f, v • read(write(H, p, f, v), p, f) = v);
axiom (∀ H, p, q, f, g, v • p ≠ q ∧ f ≠ g ⇒
    read(write(H, p, f, v), q, g) = read(H, q, g));
```

Two-dimensional heap H[p][f] and H[f][p] The heap is laid out in two dimensions, each addressed by either pointer or field. The only difference between H[p][f] and H[f][p] is the way values are obtained: For H[p][f], the heap is first addressed by a pointer and then by a field; for H[f][p], this is vice versa. Its axiomatization consists of two pairs of read-over-write axioms where disjointness is disequality of pointers and fields, respectively. For example, the heap model H[p][f] is axiomatized as follows:

```
axiom (∀ H, p, h • read(write(H, p, h), p) = h);
axiom (∀ H, p, q, h • p ≠ q ⇒ read(write(H, p, h), q) = read(H, q));
axiom (∀ h, f, v • read'(write'(h, f, v), f) = v);
axiom (∀ h, f, g, v • f ≠ g ⇒ read'(write'(h, f, v), g) = read'(h, g));
```

Field heaps F[p] Instead of a single heap, there are several distinct heaps, one for each field. For each such heap, there are distinct functions read() and write(), axiomatized using read-over-write axioms with pointer disequality as disjointness predicate.

To clarify the description of each of these encodings, let us consider an example program—a sequence of assignments—and its translation into each of the described heap encodings. In the translations, the heap, which is only implicit in the program, gets explicit, which results in ordinary assignments turned into explicit heap assignments. The different instances of the heaps are numbered starting from 0 (the index of the initial heap).

Program (without explicit heap)

```
p.f := 3; p.g := p.f;
```

Linear heap H[dot(p,f)] (similar for H[dot2(p,f)])

```
H1 := write(H0, dot(p, f), 3); H2 := write(H1, dot(p, g), read(H1, dot(p, f)));
```

Synchronous heap H[p,f]

```
H1 := write(H0, p, f, 3); H2 := write(H1, p, g, read(H1, p, f));
```

Two-dimensional heap H[p][f]

```
H1 := write(H0, p, write'(read(H0, p), f, 3));
H2 := write(H1, p, write'(read(H1, p), g, read'(read(H1, p), f)));
```

Two-dimensional heap H[f][p]

```
H1 := write(H0, f, write'(read(H0, f), p, 3));
H2 := write(H1, g, write'(read(H1, g), p, read'(read(H1, f), p)));
```

Field heaps F[p]

```
F1 := write(F0, p, 3); G1 := write(G0, p, read(F1, p));
```

2.1 Type-Safe C

In general, C does not comply with our initial assumptions of a two-tiered heap. Instead, C heap is understood as a linear heap of bytes addressed solely by pointers. Even the access to structure fields is reduced to plain pointers using address arithmetic. Moreover, individually addressable entities spanning more than one byte each (e.g., integer values or floating point numbers) may overlap on the heap. Verifying complex data structure algorithms in such a setting is out of reach, because complex invariants will be hidden by layers of pointer arithmetic and numerous disjointness checks.

Fortunately, most C programs are written in a nearly type-safe manner, avoiding unrestricted casting and pointer arithmetic most of the time. This type-safety intuition led to the memory model implemented in the current version of VCC [9], which we will refer to as VCC2 from now on. This model maintains, throughout the execution of a program, a set of *valid pointers* and an invariant stating that they do not address overlapping portions of memory unless required by the type system. For instance, C's type system mandates that fields of a structure overlap with the entire structure, but not with each other. Each valid pointer p has a unique *embedding*, i.e., another valid pointer to the structure in which p is directly contained, as well as a unique field that identifies p within its embedding. The definition of the embedding and the field of a pointer depend on the current set of valid pointers. Hence, the heap model underlying VCC2 can be approximated by H[dot2(p,f)], although the actual axioms include premises about p being valid which slows down reasoning.

Along with our experiments (Sect. 4), we created a new memory model for VCC, dubbed VCC3, which separates the concepts of pointers from that of fields and is amenable to most of the heap models presented in Sect. 2 (except for F[p]), as it makes the embedding state-independent. The key idea is to treat C pointers as *fat pointers* for specification purposes, i.e., entities consisting of another fat pointer (to the embedding) and a field. For simple type-safe field accesses, there is just one field object per field of a source-level structure, corresponding exactly to our assumption about two-tiered heaps. VCC3 restricts memory accesses to a set of valid pointers, in much the same way as in the VCC2 model. Valid pointers occupy disjoint memory locations, and thus separate writes to valid pointers do not interfere. The details, in particular pointer arithmetic, arrays and casted pointers, are tricky, and will be described in an upcoming paper.

3 Multiply-Linked List Benchmark Family

Algorithms modifying data structures are among the most complex verification challenges. A simple, yet already sufficiently hard problem is inserting an element into a

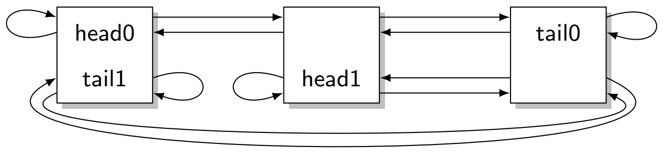

Figure 1. An example of a generalized list with degree 2. The labels in the nodes mark the heads and tails of the two link sequences.

doubly-linked list. We render this problem even harder by generalizing the list datatype into a multiply-linked list, or *multi-list* for short. Instead of one bi-directional link (consisting of pointers next and prev per node) between two nodes, we allow *n* such links and call *n* the degree of the multi-list. Consequently, the nodes of a list are not only connected by one sequence of links, but there are *n* sequences, each imposing a (potentially different) order on the list nodes. Figure 1 gives an example of a multi-list with degree 2 and consisting of three nodes.

For specifying multi-lists, we assume that the following types and functions are already specified. Their exact semantics is given by the choice of the underlying heap model (Sect. 2). Here, we turned the heap into a global variable to make the specification more readable. Moreover, we require a function read_ptr() which reads and coerces heap values into pointers and a function of_ptr() which coerces pointers back into integers.

```
type heap; type ptr; type field;
var H: heap;
function read(H: heap, p: ptr, f: field): int;
procedure write(p: ptr, f: field, v: int) modifies H;
function read_ptr(H: heap, p: ptr, f: field): ptr;
function of_ptr(p: ptr): int;
```

All of the previously described heap models except for F[p] can easily provide semantics to these declarations. For field heaps, we provide several global heaps (one for each field) as well as copies of the heap-accessing functions.

A multi-list is characterized by its nodes, represented as a set nodes of pointers, as well as *n* head and *n* tail nodes, one head and tail for each link sequence (e.g., head0 and tail0 for the first link sequence). The predicate is_node() tests pointers for membership in a multi-list. We require the following properties (collectively referred to as the invariant of a multi-list) to be fulfilled for valid multi-lists (with examples for the first link sequence).

– Heads and tails are nodes of the list.

```
is_node(H, list, read_ptr(H, list, head0)) ∧ is_node(H, list, read_ptr(H, list, tail0))
```

– Each node's predecessors and successors are again nodes of the list.

```
(∀ p: Pointer • is_node(H, list, p) ⇒
    read_ptr(H, list, next0) ≠ null ∧ is_node(H, list, read_ptr(H, p, next0)) ∧
    read_ptr(H, list, prev0) ≠ null ∧ is_node(H, list, read_ptr(H, p, prev0)))
```

– All heads are their own predecessors, tails are their own successors.

```
read_ptr(H, read_ptr(H, list, head0), prev0) = read_ptr(H, list, head0) ∧
read_ptr(H, read_ptr(H, list, tail0), next0) = read_ptr(H, list, tail0)
```

– Each inner node connects bi-directionally with its predecessor and successor. An inner node's successor and predecessor is always distinct from the node.

```
(∀ p: Pointer • is_node(H, list, p) ∧ p ≠ read_ptr(H, list, tail0) ⇒
    read_ptr(H, read_ptr(H, p, next0), prev0) = p ∧
    read_ptr(H, p, next0) ≠ p) ∧
(∀ p: Pointer • is_node(H, list, p) ∧ p ≠ read_ptr(H, list, head0) ⇒
    read_ptr(H, read_ptr(H, p, prev0), next0) = p ∧
    read_ptr(H, p, prev0) ≠ p)
```

– Each node of the list contains *m* data fields, and every such data field fulfills an abstract property good_val(). We abbreviate this entire condition by the predicate node_valid().

Given a multi-list with these properties, insertion of a new node into the multi-list works as follows. First, the data fields of the new node are set in such a way that they then adhere to the predicate good_val(). Second, for each link sequence of the list, the new node is, based on nondeterministic choice, prepended to the head node or appended at the tail node. A nondeterministic choice may either be implemented as a random decision or simply by testing the (arbitrary) value of extra boolean arguments to the insertion function. Finally, the new node is added to the set nodes of the list. The verification condition for this function essentially corresponds to showing that the multi-list invariant also holds for the extended multi-list.

The purpose of nondeterministic choice is to simulate multiple paths, typical for data structure operations (e.g., red-black tree rotations have four cases). The number of paths is exponential in the number of links, but some provers may reuse parts of the proof between paths. Note that if the VC generator splits the paths before passing them to the prover, this will result in highly symmetric VCs—exponentially many in the number of links–being generated. Therefore, we also consider cases without nondeterministic choice, where we prepend to each list.

To better illustrate the mutual pointer updates, we give here the code which prepends a new node p in front of the head of the first link sequence (head0) of the multi-list d.

```
call write(p, next0, read(H, d, head0));
call write(p, prev0, of_ptr(p));
assume node_valid(H, read_ptr(H, d, head0));
call write(read_ptr(H, d, head0), prev0, of_ptr(p));
call write(d, head0, of_ptr(p));
assert node_valid(H, read_ptr(H, read_ptr(heap, d, head0), next0));
```

The case of $n = 1$ and $m = 1$ corresponds to a doubly-linked list with one data item per node. We consider cases where the degree of a mult-list $n \leq 3$ and the number of data fields $m \leq 10$ practically relevant, e.g., tree datatypes with several data fields fall well into this category with respect to the effort needed to verify corresponding functionality. For studying the impact of the coefficients n and m, we will also consider greater values in our experiments. It is clear that the burden on automatic provers increases drastically

Automated prover	Configuration
E 1.2	−l5 −tAutoDev
SPASS 3.7	−Auto −PGiven=0 −PProblem=0 −Splits=0 −FullRed=0 −DocProof −VarWeight=3
Vampire 0.6 (r903)	−−mode casc
CVC3 2011-01-27	
Fx7 r1074	−o:MaxInstRounds=30,MaxQuantIters=2000
Yices 1.0.29	
Z3 2.18	AUTO_CONFIG=false
$Z3_{+3p}$ 2.18	CASE_SPLIT=3 DELAY_UNITS=true SORT_AND_OR=false QI_EAGER_THRESHOLD=100 RESTART_STRATEGY=0 RESTART_FACTOR=1.5 AUTO_CONFIG=false

Table 1. Automated provers used in the experiments and their configuration. $Z3_{+3p}$ is Z3 (version 2.18), but invoked with the configuration used by Boogie.

by increasing n. This is far the less the case when increasing m. Hence, scaling n yields a good coarse-grained criterion, whereas scaling m provides smooth, fine-grained trends.

We use the naming scheme n/m to refer to the multi-list insertion benchmark with n link sequences and m data fields. Benchmarks without nondeterministic choice are indicated by a minus superscript (e.g., $3/9^-$).

4 Experiments

We compare seven different automatic provers as well as all heap models described in Sect. 2 (our benchmarks and results can be obtained from `http://research.microsoft.com/~moskal/multilist.aspx`). We used the ATPs E [27], SPASS [30], and Vampire [26], and we applied the SMT solvers CVC3 [4], Fx7 [22], Yices [13], and Z3 [11]. See Table 1 for the exact prover versions and their configurations. The configurations for the ATPs are the same as used in the most recent version of Isabelle/HOL [25], namely Isabelle2011, which was our best guess as to how to configure them. CVC3 and Yices do not seem to offer options that could be relevant. Fx7 is the historical entry from SMTCOMP 2007, run with the same options as in the competition. The results for Z3 should be taken with a grain of salt as its auto-configuration features fail due to a bug in the current version. This, however, does not affect the $Z3_{+p}$ and $Z3_{+p3}$ configurations, which were used in more extensive experiments described later (starting with Sect. 4.2).

4.1 Comparing ATPs with SMT Solvers

We used benchmarks $1/1^-$ and $2/2^-$, and generated one proof obligation per assertion. The check of the multi-list invariant at the end was split into separate assertions, one per conjunct. This yields 33 proof obligations per model. Table 2 summarizes the results of this experiment.

Among all models, F[p] is the most efficient: Most VCs are proved with this model, and mostly even in the shortest average time. However, it is unclear how to implement

Model	CVC3	E	Fx7	SPASS	Vampire	Yices	Z3	$Z3_{+p}$	Total
H[dot2(p,f)]	$2_{.04}$	$4_{.09}$	$2_{2.04}$	$5_{112.99}$	$20_{2.08}$	$2_{.08}$	$3_{187.68}$	$33_{.43}$	$203_{9.09}$
H[p][f]	$9_{37.92}$	$5_{.15}$	$16_{44.09}$	$6_{35.46}$	$20_{11.67}$	$27_{9.19}$	$2_{.03}$	$32_{.43}$	$249_{10.58}$
H[dot(p,f)]	$12_{27.09}$	$4_{.09}$	$24_{22.64}$	$9_{19.47}$	$20_{1.64}$	$31_{52.16}$	$26_{20.34}$	$33_{.37}$	$291_{12.33}$
H[p,f]	$11_{3.23}$	$9_{3.76}$	$27_{21.59}$	$9_{50.67}$	$25_{6.31}$	$32_{3.80}$	$33_{1.58}$	$33_{.29}$	$311_{5.83}$
H[f][p]	$9_{.49}$	$11_{.59}$	$33_{9.37}$	$10_{21.69}$	$22_{1.91}$	$33_{1.06}$	$33_{1.64}$	$33_{.12}$	$316_{2.45}$
F[p]	$18_{.07}$	$15_{22.03}$	$33_{1.02}$	$23_{17.83}$	$33_{1.21}$	$33_{.04}$	$33_{.04}$	$33_{.04}$	$353_{2.43}$
Total	$61_{11.60}$	$48_{7.76}$	$135_{16.14}$	$62_{32.84}$	$140_{3.91}$	$158_{12.80}$	$130_{9.23}$	$197_{.28}$	$1723_{6.68}$

Table 2. Number of assertions solved and, in subscript, the average time of successful proofs in seconds for multi-list benchmarks $1/1^{-}$ and $2/2^{-}$. The timeout was set to 600 seconds, and the tests run on a 2.8 GHz Windows 7 PC.

this model when field names are not known statically, which is the case for virtually all C programs. Second best in terms of proved VCs is H[f][p] for nearly all systems, followed by H[p,f] (only for Vampire, the order of these two models is swapped). Note that both H[f][p] and H[p,f] do not suffer from implementation problems as F[p]. We will see similar results in more complex benchmarks run by Z3 in Sect. 4.3 below.

The general trend is that SMT solvers are faster and solve more problems than ATPs, but Vampire is competitive, when presented with fragmented VCs, with the leading SMT solvers used in their default configurations. As we will see in the next section, it does not perform so well when one VC per problem is generated. Reasoning with quantifiers is a field in which ATPs usually excel SMT solvers, but when we manually supply additional hints on quantified formulas, in the form of patterns, SMT solvers easily outperform ATPs (see, e.g., $Z3_{+p}$ in the last column of Table 2).

4.2 Guiding SMT Solver with Patterns

A pattern [24] for a quantified formula $\forall \overline{x}.\, \psi$ is a set of terms $\{t_0, \ldots, t_n\}$, typically subterms of ψ. The solver can instantiate ψ with the substitution σ by adding a tautology $(\forall \overline{x}.\, \psi) \Rightarrow \sigma(\psi)$ to the logical context. It will do so if $\sigma(t_0), \ldots, \sigma(t_n)$ have an interpretation in the currently considered partial ground model.

Patterns are the standard approach for handling quantifiers in SMT solvers. All solvers used in our evaluation come with their own inference algorithms to automatically derive patterns. Except for Yices, all considered SMT solvers also provide syntax to manually add patterns to problems, effectively overriding the solver's internal pattern inference and providing direct user control over how solvers perform quantifier instantiations. While patterns are sometimes dismissed as a poor man's substitute for proper quantifier reasoning methods, they are often used to effectively program a custom decision procedure in the SMT solver [10,23]. Moreover, they give SMT solvers significant edge over ATPs in software verification scenarios.

Table 3 summarizes runs of ATPs as well as different SMT solvers with (indicated by the suffix $_{+p}$) or without pattern annotations. As opposed to Table 2, the benchmarks are not split by assertion, but each is given as one VC to the solvers. We see that, when provided with explicit patterns, Z3 is clearly the most successful and fastest solver.

Model	CVC3$_{+p}$	Fx7	Fx7$_{+p}$	Vampire	Yices	Z3	Z3$_{+p}$	Z3$_{+p3}$	Total
H[p][f]	$2_{69.61}$	0	$7_{53.83}$	0	$1_{4.92}$	0	$6_{3.62}$	$9_{29.52}$	$34_{26.15}$
H[dot2(p,f)]	$3_{34.78}$	0	$5_{69.96}$	0	0	0	$7_{53.66}$	$10_{2.04}$	$35_{24.96}$
H[dot(p,f)]	$6_{150.85}$	$2_{153.15}$	$4_{18.89}$	0	$1_{79.26}$	$2_{294.07}$	$10_{40.18}$	$10_{1.47}$	$45_{53.06}$
H[p,f]	$3_{178.28}$	$2_{94.47}$	$7_{27.90}$	0	$2_{43.21}$	$6_{126.86}$	$8_{13.07}$	$10_{.60}$	$48_{39.19}$
H[f][p]	$6_{97.21}$	$6_{180.64}$	$10_{17.65}$	0	$6_{28.11}$	$6_{30.69}$	$6_{12.87}$	$10_{.18}$	$60_{37.95}$
F[p]	$7_{11.25}$	$10_{29.01}$	$10_{9.25}$	$3_{7.65}$	$10_{.27}$	$10_{13.27}$	$10_{.24}$	$10_{.10}$	$80_{7.80}$
Total	$27_{86.87}$	$20_{93.46}$	$43_{29.45}$	$3_{7.65}$	$20_{17.10}$	$24_{69.42}$	$47_{20.92}$	$59_{5.25}$	$302_{29.58}$

Table 3. Number of benchmarks solved by different systems. Systems not mentioned in the table timeout on all benchmarks. Benchmarks: $1/1^{-}$, $1/10^{-}$, $2/2^{-}$, $2/10^{-}$, $3/3^{-}$, $1/1$, $1/10$, $2/2$, $2/10$, and $3/3$.

Prover	Common	Unique	T/O	Prover	Common	Unique	T/O	Ratio
Z3	$153_{18.12}$	$1_{92.67}$	$91_{600.00}$	Z3$_{+p}$	$153_{.70}$	$91_{10.25}$	$1_{600.00}$	35.1
CVC3	$61_{11.60}$	0	$164_{600.00}$	CVC3$_{+p}$	$61_{.54}$	$164_{26.49}$	0	22.6
Fx7	$155_{26.12}$	0	$86_{600.00}$	Fx7$_{+p}$	$155_{1.81}$	$86_{15.35}$	0	34.8
Z3$_{+p}$	$244_{4.26}$	0	$13_{600.00}$	Z3$_{+p3}$	$244_{.28}$	$13_{19.45}$	0	27.6

Table 4. The first row shows that Z3 can solve 153 benchmarks both with and without patterns in average time per benchmark of 18.12 seconds and 0.70 seconds, respectively. There are 91 benchmarks that Z3 timeouts on (labelled T/O), and which Z3$_{+p}$ solves in an average time of 10.25 seconds, and the single benchmark that Z3$_{+p}$ timeouts on is solved by Z3 in 92.67 seconds. If we add up the three columns for each prover, Z3$_{+p}$ is 35.1 times faster (another approach would be to ignore timeouts and divide 18.12 by 0.70; this yields results in the same ballpark).

Boogie runs Z3 with specific options (see Table 1), indicated by the suffix $_{+p3}$. Separate experiments showed that among these options, which configure Z3's SAT solver and quantifier heuristics, CASE_SPLIT=3 is most important. This causes Z3 not to use the common SAT case-split selection strategy (based on variable activity), but to try to satisfy the formula from left to right. Our results show that this configuration outperforms all provers, even the standard configuration of Z3.

To compare the relative performance impact of patterns and case-split selection we compared the cumulative run times of solvers when solving the proof obligations of Table 2 and those of Table 3 in all six heap encodings ($(33+10) \times 6$ encodings $=$ 258 benchmarks). When only one solver of the pair could solve the benchmark, we took the time of the other to be 600 seconds (the timeout value). Table 4 summarizes the findings. The pattern speedups for Z3, CVC3 and Fx7 are roughly 1.5 order of magnitude, and would be larger had we decided to penalize timeouts more. Moreover, Z3$_{+p3}$ is 27.8 times faster than Z3$_{+p}$. Thus, a custom case-split selection strategy gives another 1.5 order of magnitude over just using patterns.

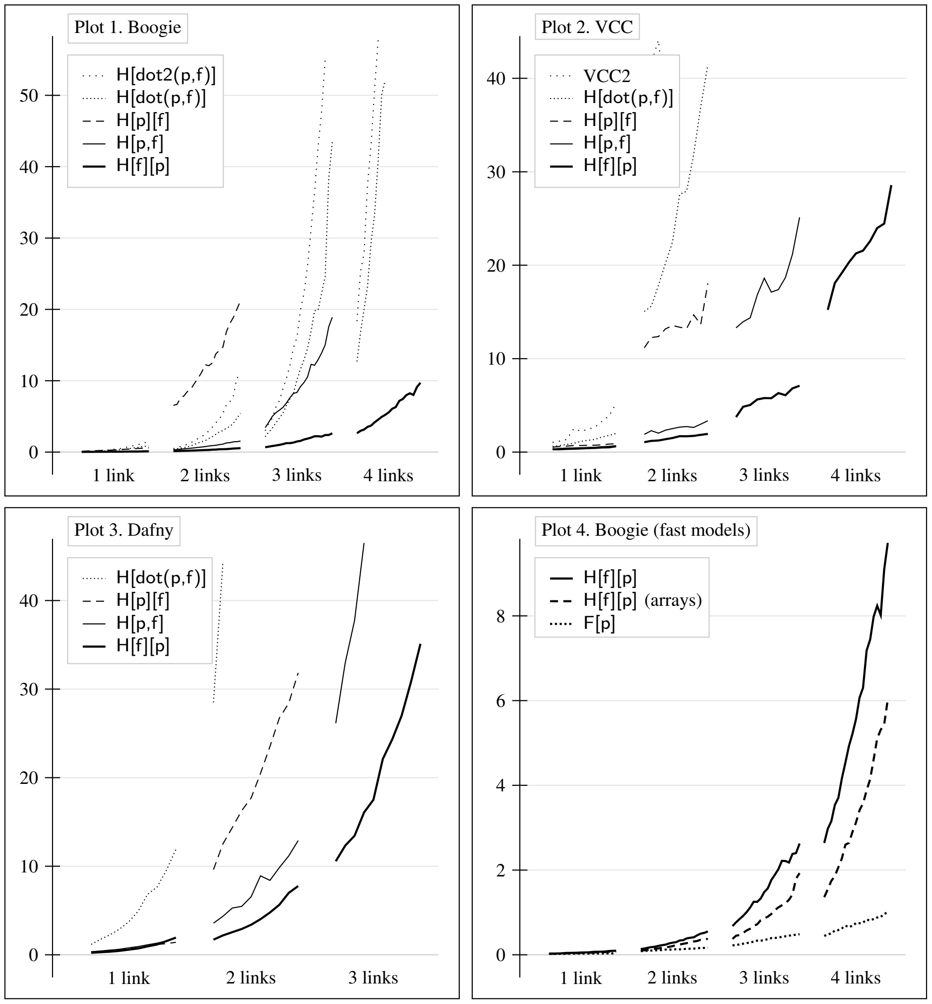

Figure 2. Multi-list benchmark using different heap encodings and different programming languages. The x axis shows the complexity of the benchmark in order $1/1$, $1/2$, ..., $1/20$, $2/1$, ..., $2/20$, ..., $4/20$. The y axis shows the median runtime (in seconds) of Z3 run with 6 different random seeds.

4.3 Scaling up

We subsequently compared different heap encodings in different languages using the fastest solver from the previous section: $Z3_{+p3}$, i.e., Z3 using patterns and the options used by Boogie. Plots 1, 2, and 3 in Fig. 2 show run times of our benchmarks in Boogie, VCC, and Dafny. We were unable to supply some of the pattern annotations to Dafny, which explains its poor performance relative to VCC. Still, it shows similar trends. We omitted all timed-out data points. We see similar results in terms of order, in particular the H[f][p] representation is always the fastest, and collapsing p and f into a single entity, especially using VCC2's dot2() function, performs very badly. This similarity gives us some confidence that experiments with other models would also carry over from Boogie to different verification systems.

Plot 4 shows the results of experiments with representations which are not implemented in VCC3, namely the F[p] and H[f][p] with the built-in array theory [12]. The F[p] representation brings about an order of magnitude speedup over H[f][p]. It is, however, known to be tricky to handle soundly and modularly: Often the accessed field is not known statically. Additionally, functions generally need to be allowed to write freshly allocated objects, which means they potentially write all fields in the heap, and thus modeling function calls reduces the benefits of F[p]. One can use different splits of the heap into field components in different functions, and maintain some consistency invariants between them, which is something we want to explore in the future.

The array decision procedure shows some speedup, but except for the biggest benchmark it is small (within 20%). It does so by avoiding certain instantiations of read-over-write axioms, which can lead to incompleteness when patterns are used to guide instantiations of user-provided axioms. Thus, in the context of VCC, using it is unlikely to be a good idea.

4.4 Interpretation of the Results

The two linear models, H[dot2(p,f)] and H[dot(p,f)], perform poorly because the prover needs to instantiate several axioms to prove disjointness of heap accesses. This is particularly painful when there are also a lot of other axioms to instantiate (i.e., in VCC or Dafny), and the prover will often instantiate those other axioms first. Especially the dot2() axioms are complicated, much more than the dot() axioms.

As for H[f][p] compared with H[p,f] or H[p][f], consider a write to x.a and subsequent reads of y.b and z.c. To find that writing x.a did not clobber the value of y.b in H[f][p], the prover can establish $a \neq b$, which is usually immediate, and moreover if $b = c$ then the value of z.c will be known without further quantifier instantiations. Similar reasoning holds for H[p][f] and $x \neq y$, but the pointer comparison usually involves complex reasoning (e.g., x was valid at some point, whereas y was not). Finally, in H[p,f] the prover can prove disjointness of either fields or pointers, but there is no reuse of instantiations for different heap accesses.

Difficult benchmarks typically have many invariants quantifying over pointers. For example, proving each of the quantified invariants in our artificial benchmark introduces a new Skolem constant n_{sk} and in turn $\mathsf{n}_{sk}.\mathsf{prev}_i$, $\mathsf{n}_{sk}.\mathsf{next}_i$, etc. Thus, difficult benchmarks are likely to use many more pointers than fields, making the reuse in H[f][p] much more significant.

The H[p][f] behaves surprisingly poorly. Initially we thought it was a Z3-specific problem (it seems to be generating very large conflict clauses in this case, which talk about distinctness of pointers), but it obviously also occurs in other provers.

4.5 Checking Real-World Verification Examples

Table 5 lists results of applying VCC with different heap encodings on verification of common data structures stemming from the VACID-0 benchmark suite [20]: binomial

Testcase	VCC2	VCC3	Ratio
Heap.c: Heap_adm	0.14 ±0.01	0.10 ±0.00	1.3×
Heap.c: extractMin	15.55 ±4.40	7.95 ±1.63	2.0×
Heap.c: heapSort	7.11 ±3.84	0.27 ±0.01	26.4×
Heap.c: heapSortTestHarness	1.03 ±0.11	0.14 ±0.00	7.6×
Heap.c: init	0.14 ±0.00	0.10 ±0.00	1.4×
List.c: InitializeListHead	0.20 ±0.01	0.14 ±0.00	1.5×
List.c: InsertHeadList	8.12 ±0.64	1.02 ±0.05	8.0×
List.c: InsertTailList	8.45 ±0.69	0.95 ±0.07	8.9×
List.c: IsListEmpty	0.10 ±0.00	0.07 ±0.00	1.4×
List.c: RemoveEntryList	4.64 ±0.23	0.53 ±0.03	8.8×
List.c: RemoveHeadList	4.75 ±0.12	0.49 ±0.03	9.6×
List.c: RemoveTailList	4.12 ±0.12	0.52 ±0.07	8.0×
List.c: _LIST_MANAGER_adm	0.22 ±0.01	0.15 ±0.01	1.5×
RedBlackTrees.c: Tree_adm	0.59 ±0.01	0.40 ±0.01	1.5×
RedBlackTrees.c: left_rotate	108.40 ±22.74	12.90 ±4.43	8.4×
RedBlackTrees.c: right_rotate	97.06 ±14.07	13.44 ±0.77	7.2×
RedBlackTrees.c: tree_find	0.14 ±0.00	0.13 ±0.01	1.0×
RedBlackTrees.c: tree_init	0.13 ±0.00	0.11 ±0.00	1.2×
RedBlackTrees.c: tree_insert	94.17 ±9.53	3.14 ±0.24	30.0×
RedBlackTrees.c: tree_lookup	0.12 ±0.00	0.08 ±0.00	1.5×

Table 5. Detailed comparison of different heap models in VCC3 against each other and VCC2. Times are median for 6 runs with different random seeds, given in seconds.

heaps, doubly-linked lists, and red-black trees. VCC3 (using the H[f][p] model) shows, in comparison with the current model of VCC (VCC2), about an order of magnitude of speedup on non-trivial functions.

We also conducted comparisons of heap models for the VACID-0 benchmarks. Due to overheads of the VCC axiomatizations, the differences between the heap encodings are not as dramatic as with our artificial multi-list benchmarks. Nevertheless, we found similar trends as in the other experiments, with H[f][p] being the best model. Thus, we consider the multi-list benchmark to be representative for real-world benchmarks.

5 Related Work

Many existing program verifiers support the selection of automatic back-end provers [2, 14, 16, 17], but so far, none has reported a thorough comparison between the automatic provers. Notably Jahob [31] uses both ATPs (E and SPASS) and SMT solvers (CVC3 and Z3) as background provers, among others.

None of the presented heap models is novel, and in fact, most of them are standard in program verifiers. For example, the state-based linear heap is used by VCC [9], the synchronous heap is underlying Dafny's memory model [19], and field heaps, due to [5, 6], are used by Jahob [31].

Challenges for program verifiers have been posed before [20], but we know of only one further proposal [1] of scalable benchmarks to evaluate the behaviour of automatic provers.

6 Conclusion

We have proposed a scalable and challenging benchmark taken from the domain of data structures, and tested it on a number of heap encodings and verification systems (Boogie, VCC and Dafny). The experiments gave similar results in terms of tendency (i.e., indicating a clear order of encodings with respect to performance) for the three systems. Additionally, testing realistic C benchmarks with a subset of the heap encodings yielded similar results. We thus believe that the multi-list benchmark is a good representative of benchmarks stemming from modular software verification, while being simple (no arithmetic required and no soundness problems) and scalable.

We have confirmed the folklore that splitting the heap by fields performs best, but have also put concrete numbers on that claim. We have also tested the performance impact of using a dedicated decision procedure for the array theory.

We have found that the performance of ATPs is comparable with that of SMT systems when software verification problems are provided without any additional hints and in small fragments. This may be particularly useful for parts of VCs without explicitly engineered patterns, such as those coming from the user of the verification tool. Processing the bigger chunks of the VC at once will likely be needed, if ATPs are to meet the performance requirements of today's verification demands.

Pattern annotations give SMT solvers a huge advantage over ATPs (at least 1.5 orders of magnitude). It would be very desirable for the ATPs to take advantage of these (e.g., via hints [29] as implemented in Otter [21]), maybe to guide term ordering heuristics. Similarly, a custom case split strategy further improves Z3's performance by 1.5 orders of magnitude.

References

1. A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic*, 10(1), 2009.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
3. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.SMT-LIB.org`, 2010.
4. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, volume 4590 of *LNCS*, pages 298–302. Springer-Verlag, 2007.
5. R. Bornat. Proving pointer programs in Hoare Logic. In *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 102–126. Springer, 2000.
6. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
7. R. Cartwright and D. Oppen. The logic of aliasing. *Acta Informatica*, 15:365–384, 1981.

8. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
9. E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. *ENTCS*, 254:85–103, 2009.
10. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, pages 302–314. ACM, 2009.
11. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
12. L. M. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design*, pages 45–52. IEEE, 2009.
13. B. Dutertre and L. de Moura. The Yices SMT solver, 2006. Available at `http://yices.csl.sri.com/tool-paper.pdf`.
14. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, Mar. 2003.
15. C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
16. P. James and P. Chalin. Faster and more complete extended static checking for the Java Modeling Language. *Journal of Automated Reasoning*, 44:145–174, 2010.
17. V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.
18. D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Formal Methods*, volume 5850 of *LNCS*, pages 806–809. Springer, 2009.
19. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNAI*, pages 348–370. Springer, 2010.
20. K. R. M. Leino and M. Moskal. VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.
21. W. McCune. *OTTER 3.3 Reference Manual*. Mathematics and Computer Science Division, Argonne National Laboratory, Technical Memorandum No. 263, 2003.
22. M. Moskal. Fx7 or in software, it is all about quantifiers. In *Satisfiability Modulo Theories Competition*, 2007.
23. M. Moskal. Programming with triggers. In *SMT 2009*, pages 20–29. ACM, 2009.
24. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC, 1981.
25. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
26. A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Comm.*, 15(2–3):91–110, 2002.
27. S. Schulz. System description: E 0.81. In D. Basin and M. Rusinowitch, editors, *IJCAR*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.
28. G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
29. R. Veroff. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *Journal of Automated Reasoning*, 16:223–239, 1996.
30. C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *Automated Deduction*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.
31. K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *Programming Language Design and Implementation*, pages 349–361. ACM, 2008.