Fat Pointers, Skinny Annotations: A Heap Model for Modular C Verification

Sascha Böhme[1] and Michał Moskal[2]

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

Abstract. Folklore has it that verification of functional properties of C programs is intractable without making compromises on soundness of the heap model or coverage of C features. In contrast, we present a heap model for deductive verification that achieves (1) soundness by explicitly keeping track of runtime type assignment of pointers, (2) completeness and precision by accounting for changes to that type assignment with little additional annotations, and (3) performance by making verification conditions easy for SMT solvers in the common case of type-safe heap accesses. Our encoding of a particular heap access operation is independent of the rest of the program, which makes the model particularly suitable for modular verification. We have implemented this model in the VCC program verifier making it an order of magnitude faster than the previous version.

1 Introduction

Ironically, the C programming language is often used in scenarios where reliability is of crucial importance. Among the many tools providing means to enforce certain quality levels, formal verification systems give the strongest guarantees about correctness of the code under consideration. Still, the world-wide fraction of fully verified C code in such scenarios is negligible. To improve this situation, the costs of formal verification need to be brought down.

The VCC verification system [2] is a step in that direction. VCC verifies that a concurrent C program adheres to contracts in the form of pre- and post-conditions and object invariants. These contracts are specified in the program text in a language that extends the C expression language with first-order quantification and user-defined predicates. Verifying a C program with VCC is much alike programming and compiling: users write an initial version of the specification, run the verifier (which reports any inconsistency of the specification with respect to the code), they then fix or add annotations, and repeat. The interaction with VCC is at the level of C constructs — the user does not need to interact with the underlying theorem prover (usually Z3 [5]). Thus, the VCC verification cost depends on the amount of annotations and the time it takes to develop each of them.

VCC has been applied to the kernel of Microsoft Hyper-V hypervisor, which is a thin layer of software allowing several operating systems to share the same machine. The Hypervisor is a small (about 100 000 lines of C) operating system, complete with scheduler, complex page management, etc. Some more details are provided in the VCC

```
struct S {
  int f;
  int a[5];
  int g;
};
```

```
void foo(struct S *s) {
  s->f = 10;
  s->g = 20;
  _(assert s->f == 10)
}
```

```
void bar(int *p) { *p = 10; }
void baz(struct S *s)
{
  bar(&s->f); bar(&s->g);
}
```

Fig. 1. Accessing fields of a C struct directly and indirectly.

paper [2]. In the Hypervisor project, we found the VCC annotation overhead, for functional correctness of the verified parts, to be about 2:1 (two lines of annotations per line of code). In a comparable effort [8] to verify the L4 micro-kernel, using the interactive theorem prover Isabelle/HOL [13], annotation overhead was reportedly 20:1. This makes the more automatic VCC approach look favorable on the annotation amount front. However, the verification cost also depends on the annotation development time. This clearly depends on the expertise of the user, but also critically on the performance of verification tool as the user is likely to run multiple verification cycles for each line of annotation. Run times of less than five seconds are desirable and up to 20 seconds still acceptable. Our experience with the Hypervisor verification shows that longer run times greatly decrease productivity; in particular, we have never seen a completed verification when the response time during development would be more than one hour.

In fact, verification turnaround times (followed by fickleness of quantifier instantiation heuristics and difficulty in understanding verification errors) were reported as the main limiting factors during the Hypervisor verification. Our previous experience and experimental data in Sect. 5.1 show that the choice in modeling the heap is extremely important for the end-to-end performance of a verification tool.

The first contribution of this paper is a novel heap model (Sect. 3) for the new version of VCC dubbed VCC3, which is shown to outperform the earlier VCC model by about an order of magnitude on complex benchmarks. The model covers arbitrary casting and pointer arithmetic, as well as support for unions, arrays, and memory reinterpretation, with little-to-no additional annotations. We also provide a mechanized soundness proof (Sect. 4).

Our second contribution is an experimental evaluation of heap axiomatizations for different languages, as well as a forward-looking evaluation of further performance improvements of VCC (Sect. 5). To this end, we compare VCC3 with VCC2, Dafny [9], and plain Boogie [1] axiomatizations of heap models in different configurations. We propose a scalable, artificial benchmark, which we claim to be representative for realistic modular functional verification problems.

VCC handles concurrency, but it does so entirely at a higher abstraction layer above the heap model (through ownership, volatile data, and object invariants). Thus, this work ignores concurrency altogether.

2 The Heap Problem and Previous Solutions

In a deductive verifier, the source program is translated to a logical formula, validity of which is checked by a theorem prover (be it a SMT solver, an ATP system, or a

interactive proof assistant) and implies correctness of the program. By the heap model we refer to the way heap accesses are translated. It generally requires read and write functions. Their interplay is usually specified using some variation of the following read-over-write axioms:

```
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));
```

In the snippet above, as well as in the rest of the paper, we use syntax of the Boogie [1] intermediate verification language. We provide explanations in cases where Boogie syntax is not self-explanatory. VCC implementation [2] translates annotated C to Boogie, which in turn interfaces with Z3 [5] or other provers.

Consider function foo() from Fig. 1. If the C expression &s->f is translated to the term x, and &s->g is translated to y, then to prove the assertion, we will need to prove disjoint(x, y), meaning that that pointers do not alias or otherwise overlap in memory. A typical verification task consists of assuming a property, performing a series of updates, and checking that the property still holds, which boils down to proving that the property still holds for the unchanged part of the heap, and that the updates did not interfere with each other as to violate the property. This requires proving lots of such anti-aliasing predicates. Thus, the particular definition of disjoint() is critical to the performance of a heap model, and in turn the verifier.

Previous VCC Solutions Over the past three years, VCC has used four different heap models (including the one described in this paper), with progressively simpler definitions of disjoint().

The very first version of VCC took a precise, machine-oriented view of memory: the heap was modeled as a map from 64-bit values (pointers) to 8-bit values. Reading a 32-bit integer involved performing four reads and concatenating the bytes according to the endianness of the machine. As a consequence, deciding disjoint() of two pointers to 32-bit values involved 16 pointer equality tests, which together with the relatively expensive bitvector decision procedure made this model not very usable for proving complex properties.

Consequently, we replaced bitvectors by mathematical integers for which decision procedures are much faster. The heap was modeled as a map from pointers to integers. We used the following "macro-definition" of disjoint():

```
function disjoint(p:ptr, q:ptr) : bool
  { addr(q) + sizeof(q) ≤ addr(p) ∨ addr(p) + sizeof(p) ≤ addr(q) }
```

where the addr(p) and sizeof(p) functions yield the memory address and the statically known size of the object pointed to by p respectively. This was much more efficient, but we hit another obstacle: the verifier required plethora of disjoint() preconditions and invariants, even about completely unrelated objects. While we were able to accomplish nontrivial verification tasks, e.g., proving memory safety of the 4500 lines of assembly code in the Hypervisor (translating the assembly code to C first), complex recursive data structures were out of reach.

Again, we re-designed the heap model to reduce the inherent complexity. The next heap model [3], currently used in VCC2, is based on the observation that even C programs are typically written in a type-safe way, i.e., avoiding unrestricted casting and

pointer arithmetic most of the time. This type-safety intuition is captured as a set of *valid pointers*, with an invariant inv() maintained throughout the program execution, stating that valid pointers do not overlap unless required by the type system (e.g., pointers to fields of a struct overlap with the struct itself, but not with each other or unrelated pointers). It also implies that valid pointers have unique *embedding*, that is a valid pointer to a struct in which they are directly contained. Pointers are represented as pairs (constructed with ptr()) of a value representing the type (accessed with typ()) and an integer address (accessed with addr()). The C expression &s−>f is translated as dot2(s, f)[3], where f is a constant of type field. For every field f of type τ defined at an offset o in a struct σ, we generate:

---

const unique f : field;
axiom ($\forall$ V:[ptr]**bool**, p:ptr • inv(V) $\wedge$ V[p] $\wedge$ typ(p) $= \sigma \Rightarrow$
  dot2(p, f) = ptr(τ, addr(p) + o) $\wedge$
  V[dot(p, f)] $\wedge$ emb2(V, dot2(p, f)) = p $\wedge$ field2(V, dot2(p, f)) = f);

We use a map from pointers to Booleans (written [ptr]**bool**) to represent the set of valid pointers. The **unique** modifier on a constant definition ensures the defined constant is distinct from all other unique constants.

In this model, the predicate disjoint(p, q) is defined to be just p $\neq$ q, which is usually proven based on the uniqueness of embedding utilized by the emb2() and field2() functions axiomatized above. Given two valid pointers p and q, &p−>a and &q−>b are known to be disjoint if p and q are known to be different (by emb2() above), or if a and b are known to be different (by field2() above). This gives a type-safe view on valid pointers: fields of different valid objects never overlap.

The C cast expression $(\tau)p$ is translated to ptr(τ, addr(p)), while pointer arithmetic $p + k$ is translated to ptr(typ(p), addr(p)+ sizeof(typ(p))$*$ k). Note that &s−>g and (**int**$*$)s + 6 evaluate to the same type/address pair — pointer arithmetic works as expected. Together with the ability to update the set of valid pointers (reinterpret memory as different types at different times), this made the VCC2 model adequate for the Hypervisor verification. Compared to the previous heap model, the annotation overhead drastically decreased, and the verification times were much faster, but still unsatisfactory on complex data structure benchmarks.

Related work The field of C verifiers focusing on both full functional correctness and a high level of automation is not very crowded.

On the functional verification side the most closely related heap model is Tuch's [14] used in L4 verified project [8]. It maintains a low-level byte heap, a word-level heap, and separate heaps for struct types, all kept in sync. The user can choose the appropriate heap (and thus required level of precision) for each memory access. This seems useful in interactive setting, but rather difficult to automate.

On the automation side, a good representative is the heap model used in the HAVOC static checker [4], which uses a notion of valid pointers similar to ours. HAVOC makes the valid pointer set immutable for efficiency reasons, which leads to compromises in either soundness or precision in modelling of memory allocation. This is an excellent

[3] The new model (Sect. 3) also defines a dot() function, so to avoid confusion we call the VCC2 one dot2(). Similarly for emb() and field().

choice for efficient bug finding, but for sound verification of systems code the changes to type state need to be modelled precisely. This is why our model migrates type information into pointers themselves. This permits efficient, sound, and precise reasoning, at the expense of additional annotations for casting, unions, and pointer arithmetic. Similar heap model, aided by a static whole-program analysis is employed in Frama-C [11].

Heap models in verification tools using non-classical logics (e.g., separation logic or dynamic logic) are not directly comparable, as most of the aliasing analysis is performed outside the reasoning engine. In that area, tools closest to VCC in scope are KeY-C [12] and Verifast [7].

3 Fat Pointers

In the VCC2 heap model, the theorem prover needs to take into account changes to the set of valid pointers to reason about aliasing. In contrast, in fully type-safe models (e.g., in Dafny [9]) fields of different objects never alias, regardless of pointer validity. We have observed that Dafny (using the same Boogie and Z3 pipeline) indeed performs much better than VCC2 on our complex data structure benchmarks. We thus decided to include identity of fields in pointer representation. Such "fat" pointers to different fields are never considered equal. Clearly, the C execution model does not account for that, but the fact that one can only access valid pointers comes to rescue — Sect. 4 has a bisimulation proof between the C execution model and our fat pointer model. The dot() function combines a pointer and a field into a pair, which can be deconstructed using emb() and field().[4]

```
type ptr, field;
function dot(p:ptr, f:field) : ptr,   field(p:ptr) : field,   emb(p:ptr) : ptr;
axiom (∀ p:ptr, f:field • emb(dot(p, f)) = p ∧ field(dot(p, f)) = f);
axiom (∀ p:ptr • dot(emb(p), field(p)) = p);
```

For a top-level pointer p, not embedded in any other struct, we define emb(p)= p. Like in the VCC2 model, we will need a runtime representation of C types. The typ() function is now defined in terms of field_type().

```
type typ;
function field_type(f:field) : typ,   sizeof(t:typ) : int;
const unique t_short, t_int, t_unsigned_int, ... : typ;
axiom (∀ t:typ • sizeof(t) > 0) ∧ sizeof(t_short) = 2 ∧ sizeof(t_int) = 4 ∧ ... ;
function typ(p:ptr) : typ { field_type(field(p)) }
```

[4] The Boogie snippets from this point until the end of the paper form a program, which has been verified. The axioms from Sect. 2 are conceptually related but do not form a part of this program. The theorems are proven as assertions inside procedures. Boogie is thus, somewhat unusually, used for a meta-proof. We generally introduce one "defining" axiom per function, but we have not mechanized the proof of their consistency. The full Boogie file (including necessary quantifier instantiation hints) is available at `http://research.microsoft.com/~moskal/vcc3memory.aspx`.

The representation of the heap is a map from fat pointers to integers[5]. As in VCC2, we also maintain a set of valid pointers, for which the type-safe heap map agrees with the physical byte-array memory. For each memory read or write, we check that the pointer accessed is valid, and thus if none of these checks fails, the execution of an abstracted program is equivalent to the execution of the corresponding real program (this is formalized in Sect. 4). Both the heap and valid pointers are stored in Boogie global variables.

```
type heap = [ptr]int, ptrset = [ptr]bool;
var H:heap, V:ptrset;
function select(H:heap, p:ptr) : int { H[p] }
function store(H:heap, p:ptr, v:int) : heap { H[p := v] }
```

The heap type can be also defined as, e.g., [ptr][field]**int**. The select(H, p) would then be H[emb(p)][field(p)], so select(H, dot(p, f)) can be optimized to H[p][f]. While these models are mathematically equivalent, they have very different performance characteristics (Sect. 5.1). The C program from Fig. 1 is translated into the following declarations (def_field() is explained below):[6]

```
const unique f, a, g : field;
axiom def_field(f, t_int, 0) ∧ def_field(a, t_int, 4) ∧ def_field(g, t_int, 24);
procedure foo(s:ptr) { assert V[dot(s, f)]; H := store(H, dot(s, f), 10);
                       assert V[dot(s, g)]; H := store(H, dot(s, g), 20);
                       assert select(H, dot(s, f)) = 10; }
procedure bar(p:ptr) { assert V[p]; H := store(H, p, 10); }
procedure baz(s:ptr) { call bar(dot(s, f)); call bar(dot(s, g)); }
```

3.1 Arrays and Pointer Arithmetic

Arrays embedded in structs are handled using a field constructed using function idx(), e.g., memory access s−>a[3] is translated to select(H, dot(s, idx(a, 3))).

```
function idx_of(f:field) : int,   field_of(f:field) : field,   offset(f:field) : int;
function idx(f:field, i:int) : field;
function def_field(f:field, t:typ, o:int) : bool
  { field_of(f) = f ∧ idx_of(f) = 0 ∧ field_type(f) = t ∧ offset(f) = o }
axiom (∀ f:field, i:int • field_of(idx(f, i)) = field_of(f) ∧
   idx_of(idx(f, i)) = i + idx_of(f) ∧ field_type(idx(f, i)) = field_type(f) ∧
   offset(idx(f, i)) = offset(f) + sizeof(field_type(f)) ∗ i);
```

The function idx() is a type-safe approximation of pointer arithmetic. Because the verifier knows idx_of(idx(a, 2))≠ idx_of(idx(a, 3)), it concludes that &s−>a[2] and &s−>a[3] do not alias. Because field_of(idx(a, 0))= a and field_of(f)= f, it can conclude that also &s−>a[0] and &s−>f do alias.

What about &s−>g and &s−>a[5]? From the verifier point of view they are different pointers, however the C pointer comparison will see them as equal. Generally, &s−>a[5]

[5] Boogie type **int** refers to unbounded integers, but the VCC implementation takes care also of bounded integers and overflows.

[6] To verify the program, one needs preconditions about validity of pointers. In VCC, validity is implied by ownership, but ownership itself needs to be specified explicitly.

will be invalid, and thus impossible to read or write. The C pointer comparison operation p == q is translated to addr(p)= addr(q), where the function addr() is axiomatized as follows:

```
function addr(p:ptr) : int;
axiom (∀ p:ptr, f:field • addr(dot(p, f)) = addr(p) + offset(f));
```

The idx() function is also used for arbitrary pointer arithmetic, e.g., p + 7 is translated to dot(emb(p), idx(field(p), 7)). The addr() interpretation above and the computation of offset(idx(...)) ensure that addr() agrees with the actual addresses computed by the C program.

What if the user expects s−>a[5] to access s−>g? In VCC the user would need to supply an annotation: ∗(_(**retype**)&s−>a[5]).[7] Every annotation _(**retype**)p is translated as retype(V, p), which will "guess" the proper way to access the value of type typ(p) at addr(p). Specifically, the verifier is going to maintain the following type uniqueness property of the set of valid pointers:

```
function unique_ta(V:ptrset) : bool
  { (∀ p, q:ptr • V[p] ∧ V[q] ∧ addr(p) = addr(q) ∧ typ(p) = typ(q) ⇒ p = q) }
```

For such type-unique valid pointer sets, we can postulate the existence of the retype() function, such that:

```
function retype(V:ptrset, p:ptr) : ptr;
axiom (∀ V:ptrset, p:ptr • unique_ta(V) ⇒
  (addr(retype(V, p)) = addr(p) ∧ typ(retype(V, p)) = typ(p)) ∧
  (V[p] ⇒ retype(V, p) = p));
```

That is, retype() is a function that yields a pointer with the same type and address as the input pointer, and if there exists a valid pointer at that type and address, it yields this very pointer (which by unique_ta() is unique).

The predicate unique_ta() states that valid pointers of the same type that have the same address (e.g., because they compare equal using the C == operator) are indeed equal. For example, provided p and q are valid pointers of the same type, the following snippet will verify: **if** (p == q){ _(**assert** ∗p == ∗q)}.

Arrays not embedded in structures are treated as instances of a synthetic type, similar to the one defined in C++-like syntax below:

```
template <typename T, size_t n> struct Array<T,n> { T data[n]; };
```

The C cast expression $(\tau)p$ is translated to cast(τ, p), axiomatized as:

```
function cast(t:typ, p:ptr) : ptr;
axiom (∀ t,s:typ, p:ptr • typ(cast(t, p)) = t ∧ addr(cast(t, p)) = addr(p) ∧
        cast(typ(p), p) = p ∧ cast(t, cast(s, p)) = cast(t, p));
```

We may also axiomatize cast(t, p) to return dot(p, f) when field_type(f)= t ∧ offset(f)= 0, and similarly for emb(), so when cast() is applied to a valid pointer, it will yield a valid pointer in common cases. Otherwise, the user will be required to supply the _(**retype**) annotation.

[7] All VCC annotations are enclosed in _(...), which permits hiding them from the regular C compiler.

Generally, retype() should not be used too extensively (e.g., in the default translation of cast), as it may degrade the verification experience (e.g., (**int***)p yielding different values in different states is counterintuitive), as well as performance (by mixing the heap into previously stateless operations). Fortunately, it is not required for type-safe field and array accesses, nor for simple casting.

4 Formalization

This section justifies soundness of approximation of the C memory with a map from fat pointers to integers. We first define a low-level model mandated by the C standard [6], then define the coupling invariant inv() between the low-level memory, and the high-level heap and the set of valid pointers. The set of valid pointers can be changed using explicit memory reinterpretation. We have previously shown that by using enough reinterpretation, a similar heap model is also complete [3].

The inv() predicate, and all predicates in its definition, are only used for the meta-proofs below. We prove once and for all that writing at valid pointers and reinterpretation preserve inv(), so there is no need to assert or assume inv() explicitly in translations of C programs (cf. the HAVOC model [4]). It is used implicitly to justify using high-level heap for reading and writing, and the existence of the retype() function.

4.1 The Coupling Invariant

The C standard views the heap as a set of disjoint chunks of memory, within which pointer arithmetic is possible. The memory read operation conceptually reads a sequence of bytes stored at a particular address (which is just an integer), and combines them depending on endianess of the machine and the type read. Similarly, the write operation will update a sequence of bytes.

```
type mem;
function read(M:mem, t:typ, a:int) : int;
function write(M:mem, t:typ, a:int, v:int) : mem;
```

The C standard mandates that reading a value from previously written location yields the written value, and that writes through non-overlapping pointers commute. To define overlapping, we use a function support(t, a) yielding the set of memory addresses occupied by pointer a of type t. The predicate scalar(t) holds iff t is a scalar type (i.e., an arithmetic type or a pointer, but not a struct or union type). read() and write() are used only for scalar pointers. Because we are using Boogie maps to represent sets, Boogie lambda-expressions can be used as set comprehensions.

```
type addrset = [int]bool;
function scalar(t:typ) : bool;
function support(t:typ, a:int) : addrset { (λ q:int • a ≤ q ∧ q < a + sizeof(t)) }
axiom (∀ M:mem, t:typ, a:int, v:int • read(write(M, t, a, v), t, a) = v);
axiom (∀ M:mem, t1, t2:typ, a1, a2:int, v:int • support(t1, a1) ∩ support(t2, a2) = ∅
  ⇒ read(write(M, t1, a1, v), t2, a2) = read(M, t2, a2));
```

We are going to maintain a coupling invariant inv(M, H, V) between the low-level memory M, the heap H, and the set of valid pointers V. The invariant states that supports of scalar valid pointers are disjoint, and the values stored in M and H at scalar valid pointers agree. It also includes the predicates unique_ta() and anchored(), which will be important for the proof in the next section but are irrelevant here (ordinary writes do not depend on them, nor do they update the valid pointer set).

```
function supp(p:ptr) : addrset { support(typ(p), addr(p)) }
function scalar_only(P:ptrset) : ptrset { (λ p:ptr • P[p] ∧ scalar(typ(p))) }
function disjoint_supps(P:ptrset) : bool { (∀ p, q:ptr •
  scalar_only(P)[p] ∧ scalar_only(P)[q] ⇒ p ≠ q ⇒ supp(p) ∩ supp(q) = ∅)) }
function inv(M:mem, H:heap, V:ptrset) : bool {
  (∀ p:ptr • V[p] ∧ scalar(typ(p)) ⇒ select(H, p) = read(M, typ(p), addr(p))) ∧
  disjoint_supps(V) ∧ anchored(V) ∧ unique_ta(V) }
```

Theorem 1. *Storing values at valid scalar pointers in the low-level memory and the abstract heap maintains the coupling invariant.*

```
(∀ M:mem, H:heap, V:ptrset, p:ptr, v:int • V[p] ∧ scalar(typ(p)) ⇒
  inv(M, H, V) ⇒ inv(write(M, typ(p), addr(p), v), store(H, p, v), V))
```

Proof. Automatic in Boogie. □

By the coupling invariant and the theorem above, both reads and writes at valid pointers can be performed on the high-level heap, and the low-level memory can be discarded for verification. If all heap accesses are performed at valid pointers, the programs accessing low and high-level heaps are bisimilar. This justifies the foo/bar translation in previous section.

4.2 Reinterpretation

Sometimes programs change the type assignment, changing the set of valid pointers and reinterpreting the contents of the low-level memory as seen on the high level. One example is a memory allocator, which needs to interpret chunks of memory as byte arrays before allocation, and as some other types after allocation. Reinterpretation removes a set of pointers from V, and adds another set, which occupies the same bytes in the low-level memory. For locality of reasoning, the result of reinterpretation generally needs to be considered fresh, i.e., invalid before reinterpretation. This is the usual property required from memory allocation—writing to freshly allocated memory does not violate properties of other objects.

Allowing unrestricted reinterpretation may lead to violation of unique_ta() property, which is crucial for pointer arithmetic (Sect. 3.1). For example, consider **struct** A { **struct** B b; } *a. The pointer &a->b, and all fields below, could be reinterpreted as a byte array with **sizeof**(**struct** B) elements, leaving a itself valid. The byte array could be then reinterpreted as an instance of **struct** A, because it has the proper size. The resulting pointer will be fresh, and thus distinct from a (which was valid all along), leading to a violation of unique_ta().

To prevent this situation, we prevent at least one member of a struct from being reinterpreted, as long as the pointer to the struct itself stays valid. In the example above, one could not reinterpret &a−>b without reinterpreting a in the same operation. Formally, a set of valid pointers is *anchored* iff for any valid pointer, there exists a valid scalar pointer reachable from it via dot() operations. We express dot()-reachability using transitive closure of its inverse (emb()). embt(p, t) yields first embedding of type t if it exists.[8]

```
function embt(p:ptr, t:typ) : ptr;
function anchored(V:ptrset) : bool { (∀ p:ptr • V[p] ⇒
  (∃ q:ptr • scalar(typ(q)) ∧ V[q] ∧ embt(q, typ(p)) = p ∧ supp(q) ⊆ supp(p))) }
```

The supp_set(P) yields sum of supports of pointers in P. A *fully-anchored* set of pointers is one where each non-scalar pointer p has all addresses in its support covered by disjoint scalar pointers reachable from p via dot() function. An example would be a pointer to a struct, together with pointers to all its (transitive) members.

```
function supp_set(P:ptrset) : addrset { (λ a:int • (∃ p:ptr • P[p] ∧ supp(p)[a])) }
function fully_anchored(P:ptrset) : bool { (∀ p,q:ptr • P[p] ∧ P[q] ∧ scalar(typ(q)) ⇒
  supp(p) ∩ supp(q) = ∅ ∨ embt(q, typ(p)) = p) ∧
  supp_set(scalar_only(P)) = supp_set(P) ∧ disjoint_supps(P) }
```

Memory reinterpretation operation takes two sets, A and B, and makes pointers in A invalid and pointers in B valid. It requires that sum of supports of scalar pointers in A and B are equal, and that B is fully anchored. It also requires, that pointers in A are valid, and that (V \ A) ∪ B is anchored. In VCC2, the analog of the last condition was ensured by making sure that A is closed under emb2(), i.e., allowing reinterpretation only of top-level pointers (with an exception for unions). In VCC3, we relax this condition: if A contains p but not emb(p), then either (1) emb(p) is a union, and B contains another member of that union, or (2) there is at least one other member of emb(p) in V \ A. After reinterpretation, the newly valid pointers get their values from the low-level memory.

Theorem 2. *Reinterpretation preserves the coupling invariant.*

```
(∀ M:mem, H,H':heap, V,V':ptrset, A,B:ptrset •
  V' = (V \ A) ∪ B ∧
  H' = (λ r:ptr • if B[r] then read(M, typ(r), addr(r)) else select(H, r)) ∧
  fully_anchored(B) ∧ supp_set(scalar_only(A)) = supp_set(scalar_only(B)) ∧
  A ⊆ V ∧ anchored(V') ∧ inv(M, H, V) ⇒ inv(M, H', V'))
```

Proof. Automatic in Boogie. □

The low-level memory is unavailable in verification, so the heap reinterpretation needs to be either conservatively approximated (by giving newly valid pointers unspecified values), or specified precisely based on the high-level heap before reinterpretation, and appropriate bitvector operations.

[8] The precise definition is recursive, but is needed only in meta-argument not in Boogie proof. embt(p, t) ≡ **if** typ(p)= t ∨ emb(p)= p **then** p **else** embt(emb(p), t)

5 Experimental Evaluation

The fat pointers model has been implemented in VCC3.[9] In comparison with the previous model implemented in VCC2, it shows about an order of magnitude of speedup on common data structures stemming from the VACID-0 benchmark suite [10]: binomial heaps, doubly-linked lists, and red-black trees. Appendix A gives a detailed view of our measurements, and Plot 1 in Fig. 2 shows the results in a graphical way (with lines indicating identical run times as well as two and ten times faster run times). Note that the VACID-0 benchmarks are geared towards modular verification of full functional correctness, which happens to be our area of interest, meaning we verify a small amount of code (a single function) against a complex invariant.

The measurements were made for the optimal choice of heap representation, as described in Sect. 5.1. In all experiments, the back-end prover Z3 [5] (version 2.10) was limited to a run time of 60 seconds and used with the default set of options supplied by Boogie. We did not benchmark any other back-end prover.[10]

5.1 Heap Representations

The fat pointer model permits different representations of type-safe heaps. Some of them are easily pluggable into the VCC3 background axiomatization, but some require significant work. We wanted to understand the impact of these choices, without having to do a full-fledged VCC implementation. Moreover, we wanted to separate the performance effects of VCC background axiomatization of orthogonal C features (e.g., modular arithmetic).

In fact, even before implementing the VCC3 model, we had encoded an abstraction of the VCC2 model directly in Boogie and compared that against different variations of type-safe heaps. We have evaluated these models on a simple artificial benchmark writing n fields in m objects, and then asserting that the value read is the value written beforehand. The abstraction of VCC2 was orders of magnitude slower than simple type-safe heaps, which gave us strong motivation to develop the fat pointers model. We also observed large differences between different type-safe heap representations.

After implementing the VCC3 prototype, we have plugged some of the type-safe representations and evaluated them on VACID-0 benchmarks. While all VCC3 variations were still dramatically faster than VCC2, the differences between different type-safe heaps were much smaller than in plain Boogie, on the simple read-write benchmark. Also the relative order of these representations was different. We thus developed an artificial benchmark, based on the linked-list data structure, which "simulates" VACID-0 benchmarks (i.e., gives similar results in terms of relative order), but is scalable, simple, and amplifies the effects of heap encoding choices.

[9] VCC is available in source for academic use at `http://vcc.codeplex.com/`.

[10] The other SMT solvers with quantifier support (notably CVC3, Fx7, and Yices1) are at least two orders of magnitude slower and, except for CVC3, no longer under development (cf. the results of annual SMT competition, in the relevant AUFLIA and UFNIA divisions, `http://www.smtcomp.org/2010/`). ATP systems typically do not support arithmetic, required by VCC background axiomatization.

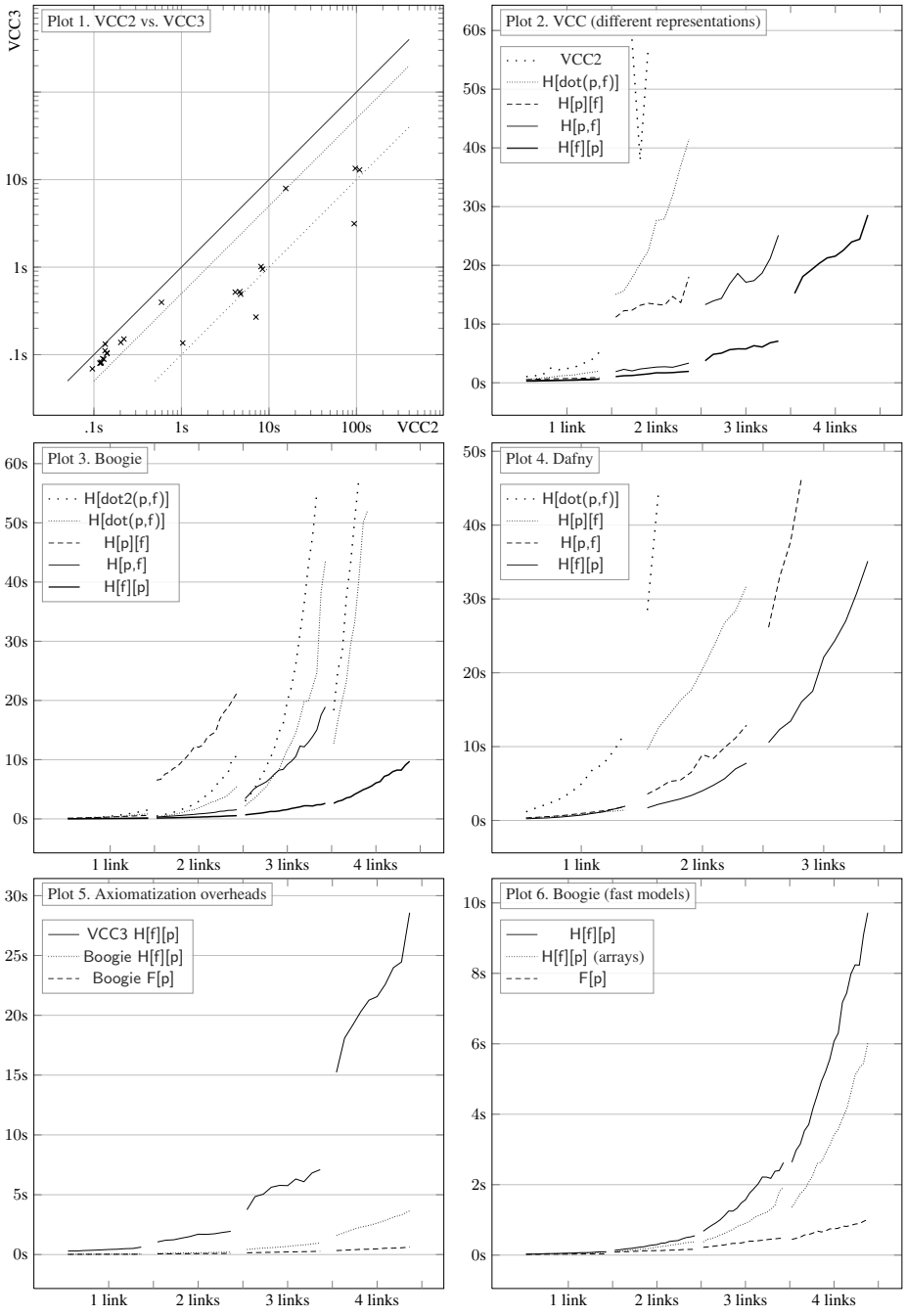

Fig. 2. Results of experimental evaluation using a 2.7GHz Intel machine.

The Representations In each representation, the heap is accessed at a specific pointer and field (p,f), which can be combined using functions dot2() and dot() axiomatized in Sect. 2 and Sect. 3 respectively, giving rise to the representations H[dot2(p,f)] and H[dot(p,f)]. Another option is to treat the heap as a map from pointers to a map from fields to values, yielding the representation H[p][f]. We can also reverse p and f getting H[f][p]. These two can be implemented either using an read-over-write axiomatization (the default choice), or using the SMT solver array theory. Yet another option is a Boogie multi-dimensional map H[p,f], axiomatized similarly to single-dimensional maps:

```
axiom (∀ H, p, f, v • H[p, f := v][p, f] = v);
axiom (∀ H, p1, p2, f1, f2, v • p1 ≠ p2 ∨ f1 ≠ f2 ⇒ H[p1, f1 := v][p2, f2] = H[p2, f2]);
```

Finally, one can use a separate heap for each field, F[p].

The Benchmark The objective of the benchmark is to verify that a list insertion function maintains the list invariant. The function operates on a designated list manager object, pointing to $2n$ list nodes (n heads and n tails). Each list node has n next pointers and n previous pointers. The manager also contains a specification field holding the set S of nodes it controls. The invariant of manager t contains the following formula for $i = 0 \ldots n-1$ (which we give in pseudo-code using p.f to denote heap access).

$$(\forall\ n \bullet n \in t.S \Rightarrow n.next_i \in t.S \wedge n.prev_i \in t.S \wedge n.next_i.prev_i = n \wedge n.prev_i.next_i = n) \wedge t.head_i \in t.S \wedge t.tail_i \in t.S$$

Additionally, each list node carries m data fields, which are required by the invariant to be positive. The x axes of the plots show the number of data fields m between 0 and 19, separately for number of links n between 1 and 4. The insertion function adds a fresh node to t.S, updates its data fields to be positive, and then for $i = 0 \ldots n-1$ it performs a non-deterministic choice to link it at either $t.head_i$ or $t.tail_i$, updating next and previous pointers accordingly.[11]

Results Plots 2, 3, and 4 show run times of our benchmark in VCC, Boogie, and Dafny[12]. 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 other verification systems. Plot 5 shows the heavy toll taken by VCC background axiomatization compared to Boogie.

Finally, Plot 6 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. 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: e.g., in function bar() from Fig. 1 the accessed field is not know without inspecting the call sites. Additionally, functions generally need to be allowed to write freshly allocated objects, which means

[11] Full implementations in Boogie, Dafny, and VCC can be found at `http://www4.in.tum.de/~boehmes/vcc3.html`.

[12] We were unable to supply some of the triggering annotations for Dafny, which explains its performance relative to VCC. Still, it shows similar trends.

they potentially write all the fields in the heap, and thus modeling function calls reduces 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 future.

The array decision procedure shows 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 triggers are used to guide instantiations of user-provided axioms. Thus, in context of VCC, using it is unlikely to be a good idea.

Rationale The two dot() models perform poorly because the prover needs to instantiate some axioms to prove disjointness of heap accesses. This is particularly painful where there is also a lot of other axioms to instantiate (i.e., in VCC or Dafny, not plain Boogie), and the prover will often instantiate those other axioms first. Moreover, the dot2() axioms are more complicated than the dot() axioms.

As for H[f][p] compared to 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. Note that the opposite case of $a = b$ is much less likely, because programs usually use more than two fields. Similar reasoning holds for H[p][f] and $x \neq y$, however 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 $n_{sk}.prev_i$, $n_{sk}.next_i$, etc. Thus, difficult benchmarks are likely to use many more pointers than fields, making the reuse in H[f][p] much more significant.

6 Conclusion

We described a novel heap model for C that is at the same time sound (Sect. 4.1), precise (Sect. 4.2), and efficient by making the type-safe accesses simple (Sect. 5). This heap model has been implemented for VCC3, a new version of a state-of-the-art C verifier. Experiments with verification of data structure examples show that VCC3 outperforms VCC2 by at least one order of magnitude, thanks to the new heap model, at the expense of additional annotation in some corner cases.

We also tested different heap encodings in three different verification systems: Boogie, Dafny, and VCC. To this end, we designed a scalable and challenging benchmark taken from the domain of data structures. Testing it on the three systems mentioned gave similar results in terms of tendency (i.e., indicating a clear order of encodings with respect to performance). We have confirmed the folklore that splitting the heap by fields performs best, but have also put concrete numbers on that claim. We believe that these results are of general interest and carry other to further verification systems.

Our measurements show that there is still a significant performance gap between VCC3 and an axiomatization of VCC3's core heap model in Boogie. Future work will address this issue.

References

1. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
2. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
3. Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. *ENTCS*, 254:85–103, 2009.
4. Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, pages 302–314. ACM, 2009.
5. Leonardo M. de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
6. International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, December 1999.
7. Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.
8. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *POPL*, pages 207–220. ACM, 2009.
9. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNAI*, pages 348–370. Springer, 2010.
10. K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.
11. Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, January 2009.
12. Oleg Mürk, Daniel Larsson, and Reiner Hähnle. KeY-C: A tool for verification of C programs. In *CADE*, volume 4603 of *LNCS*, pages 385–390. Springer, 2007.
13. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
14. Harvey Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *Journal of Automated Reasoning: Special Issue on Operating System Verification*, 42(2–4):125–187, Apr 2009.

A Detailed Comparison of VCC2 and VCC3

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 1. Detailed comparison of VCC2 and VCC3. Times are median for 6 runs with different random seeds, given in seconds. ± refers to variance of results when using different random seeds.