A Precise Yet Efficient Memory Model For C

Ernie Cohen [1]

Microsoft Corp., Redmond, WA, USA

Michał Moskal[2]

European Microsoft Innovation Center, Aachen, Germany

Stephan Tobies[3]

European Microsoft Innovation Center, Aachen, Germany

Wolfram Schulte[4]

Microsoft Research, Redmond, WA, USA

Abstract

Verification for OO programs typically starts from a strongly typed object model in which distinct objects/fields are guaranteed not to overlap. This model simplifies verification by eliminating all "uninteresting" aliasing and allowing the use of more efficient frame axioms. Unfortunately, this model is unsound and incomplete for languages like C, where "objects" can overlap almost arbitrarily. Sound verification for C therefore typically starts from an untyped memory model, where memory is just an array of bytes. The untyped model, however, adds substantial annotation burden, and reasoning in the untyped model is computationally expensive.

We propose a sound, typed semantics for C that provides the annotational and computational advantages of the typed object model while remaining sound and complete for C. We maintain a predicate identifying where the "valid" objects are, and introduce invariants and proof obligations that guarantee that the valid objects are suitably antialiased, and that (almost) all objects appearing in the program are valid. We describe the implementation of this approach in VCC (a sound verifier for C being used to verify the Microsoft Hypervisor) and the resulting performance gains.

Keywords: deductive program verification, C programming language, memory models.

1 Introduction

When writing a program verifier for an imperative language, a fundamental design decision is how to model program state. In typesafe languages like Java and C#,

[1] Email: ernie.cohen@microsoft.com

[2] Email: michal.moskal@microsoft.com

[3] Email: stephan.tobies@microsoft.com

[4] Email: schulte@microsoft.com

```
void foo(int *p, short *q)
{
  *p = 12;
  *q = 42;
  assert(*p == 12);
}
```

```
void bar(int *p, int *q)
  requires(p != q) {
  *p = 12;
  *q = 42;
  assert(*p == 12);
}
```

Fig. 1. Partial overlap of primitive pointers

the state consists of a collection of objects, each with its own fields, some of which might be pointers to objects. Thus, aliasing can arise only through two pointers (of the same type) pointing to the same object. This allows a convenient logical representation of state, e.g., as a mapping from $\langle$object, field$\rangle$ pairs to values, and easily mechanized frame axioms, where a write to a field of an object leaves the map unchanged at all other points.

C deviates from this view of state in fundamental ways. First, C has no real "objects"; types merely give a way of interpreting a chunk of memory. Thus, in C, objects can overlap arbitrarily (within the limits of object alignment). Second, in C, there is no distinction between objects and fields. A struct can contain another struct as a member, and a pointer can point to a member of a struct.

Because of these differences, we cannot soundly use the typed (object) representation directly for C programs. For example neither of the assertions in Fig. 1 is valid [5] — in each case, the parameters might point to overlapping memory blocks. In the case of function `bar` even though we explicitly rule out the possibility of the pointers being equal, because the size of the **`int`** type is bigger than one, the memory blocks pointed to by `p` and `q` might partially overlap.

An alternative, to the high-level object model, is the "official", untyped model of C, where memory is essentially a sequence of bytes [6] . The size of each type, as well as offsets of members within structs, is given by the application binary interface, so access addresses and widths can be computed from the type definitions; two objects are disjoint if they occupy disjoint memory ranges. However, this model has several disadvantages. First, object disjointness is more complicated: in the object model, two objects alias iff their addresses are the same, whereas in the C model, we have a more complex condition depending on both their addresses and their sizes. Second, it greatly increases the annotation burden on the code. For example, in the examples above, we would have to add additional assertions guaranteeing that p and q are disjoint. Moreover, doing this naively leads to a situation where the number of disjointness assertions grows quadratically with the number of objects.

The key to rescuing the typed memory model lies in the slogan, "In every untyped program, there is a typed program trying to get out" [7] . We maintain in ghost program state a set of "valid" typed pointers that point to the "real" objects of the state. Our C memory model does retain one difference from the object model, arising from the fact that C does not distinguish between objects and fields: if the state contains a valid object whose type is a struct, then the members of the struct

[5] While most modern architectures enforce alignment, the prevailing x86/x64 does not, and thus we do not take advantage of alignment in our general purpose verification tool.

[6] The memory model described in the C standard [15] is actually a collection of byte sequences, with pointer arithmetic allowed only within a single sequence. This distinction is irrelevant for our purposes.

[7] "Inside every large program, there is a small program trying to get out." (Tony Hoare)

```
struct A { short x, y; };
struct B { short z, *pz; };
struct C1 { A a1; short w1; };
struct C2 { short w2; A a2; };
void baz(A *a, B *b) {
  a->x = 1; a->y = 2;
  b->z = 3;
  assert(a->y == 2 &&
         b->z == 3);
}
```

```
void qux(C1 *c1, C2 *c2) {
  c1->a1.x = 1; c2->a2.x = 2;
  assert(c1->a1.x == 1 &&
         c2->a1.x == 2);
}
void should_fail(C1 *c1, A *a)
{
  c1->a1.x = 1; a->x = 2;
  assert(c1->a1.x == 1);
}
```

Fig. 2. Overlaps impossible in well typed C programs, unless typing allows aliasing

are also valid objects[8] Thus, our aliasing invariant is slightly weaker: if two valid objects overlap, then one is a structural "descendent" of the other.

The examples from Fig. 1 and 2 verify under the assumption that all involved pointers are valid. In VCC such validity is derived from preconditions talking about ownership of those memory locations (in this case the owner would need to be the current thread of executions). The ownership system of VCC is not a topic of this paper (but is described elsewhere [6]), we thus do not get into the precise preconditions that are needed in these examples. Anyhow, the validity of involved pointers yields the desired antialiasing properties. For example, the function `foo` verifies because `p` and `q` cannot alias because they have different types, and neither can be a structural descendent of the other (because they are both base types). The function `bar` verifies because two objects of the same type can alias only if they are identical. The function `baz` (Fig. 2) verifies because `a` and `b` cannot overlap (because the types `A` and `B` are unrelated in the type containment hierarchy). Similarly, the function `qux` verifies because `c1` and `c2` cannot overlap (hence `&c1->a1` and `&c2->a2` cannot overlap). Finally, in function `should_fail` the pointer `a` can indeed be equal to `&c1->a1`, so verification of the assertion fails.

1.1 Contribution

We introduce a typed memory model for C, where pointers to structs are interpreted as implicitly non-overlapping objects with implicitly disjoint fields (Sect. 4). This model is sound (Sect. 5.1) and complete (Sect. 5.2) with respect to the untyped (C) memory model, but places significantly lower burden on the programmer and the theorem prover (Sect. 7). We also show how other C types like unions, arrays, and bitfields are incorporated into the typed model (Sect. 6.4) and how we deal with performance issues like bit-vector reasoning which is typically too slow for verification.

In addition to making verification more convenient and efficient, this work lays the foundation for applying object-oriented verification techniques (e.g. ownership and type invariants) to C programs, as described in the companion publications [6].

2 A Toy Language

Fig. 3 lists the constructs of a toy programming language supporting pointer arithmetic and updates at arbitrary locations in the memory (but not conditionals, it-

[8] This does not include bitfields, see section 6.3.

$$
\begin{array}{lll}
 & \mathbb{T}_I & ::= \texttt{i8} \mid \texttt{u8} \mid \texttt{i16} \mid \ldots \mid \texttt{u64} \\
 & \mathbb{T}_P & ::= \mathbb{T}_I \mid \mathbb{T}* \\
S & \in \mathbb{T}_S & ::= \texttt{S}_1 \mid \ldots \mid \texttt{S}_n \\
t & \in \mathbb{T} & ::= \mathbb{T}_P \mid \mathbb{T}_S \\
f & \in \mathbb{F} & ::= f_1 \mid \ldots \mid f_m \\
e & \in \mathbb{E} & ::= \mathbb{E} \oplus \mathbb{E} \mid \mathbb{E} \rightsquigarrow \mathbb{F} \mid (\mathbb{T})\mathbb{E} \mid \mathbb{V} \mid \mathbb{N} \\
\psi & \in \Psi & ::= \mathbb{E} \otimes \mathbb{E} \\
s & \in \mathbb{S} & ::= \textbf{assert } \Psi \mid \textbf{assume } \Psi \mid *\mathbb{E} := \mathbb{E} \mid \mathbb{V} := *\mathbb{E} \mid \textbf{split } \mathbb{E} \mid \textbf{join } \mathbb{E} \\
ss & \in \mathbb{S}^* & ::= \mathbb{S}; \mathbb{S}^* \mid \epsilon
\end{array}
$$

$$
\begin{array}{lll}
tp(e_1 \oplus e_2) & = tp(e_1) & \text{where } tp(e_1) = tp(e_2) \wedge\ tp(e_1) \in \mathbb{T}_I \\
tp(e_1 \rightsquigarrow f) & = t* & \text{where } tp(e_1) = S* \wedge\ \textbf{struct } S\ \{\ldots f : t; \ldots\} \\
tp((t)e_1) & = t & \text{where } t \in \mathbb{T}_P \wedge\ tp(e_1) \in \mathbb{T}_P \\
tp(c) & = \texttt{u64} & \\
tp(v) & = \texttt{u64} &
\end{array}
$$

Fig. 3. The language

erations or procedural abstractions, extensions which can be added using standard techniques [11]). The language supports integers ($\mathbb{T}_I$), pointers (i.e., $t*$ is pointer to t), and struct types ($\mathbb{T}_S$, ranged over by S). Integer and pointer types are collectively called *primitive* types ($\mathbb{T}_P$). Expressions ($\mathbb{E}$) are side-effect free and consist of binary expressions $e_1 \oplus e_2$ (where $\oplus$ is any C binary integer operator), field address computation $e \rightsquigarrow f$ (in C it would be written `&e->f`), type casts (which allow for casting between pointers and integers and thus arbitrary pointer arithmetic), variable references, and literals. Constants and variables are restricted to unsigned 64 bit integers (but may be used to hold pointers with suitable casting). Formulas (Ψ) consist of binary relations ($\otimes$) applied to expressions. Statements ($\mathbb{S}$) consist of assertions, assumptions, memory write, memory read, and the type reinterpretation operations split and join; programs ($\mathbb{S}^*$) are sequences of statements.

Struct types are defined as part of the program environment. We use **struct** $S\ \{f_1 : t_1; \ldots; f_n : t_n\}$ as a predicate meaning that the struct S is part of the program environment and contains fields $f_1, \ldots, f_n$ of types $t_1, \ldots, t_n$. As in C, struct types are acyclic, i.e., a struct S cannot contain a field of type S at any level of nesting (but it can contain fields typed $S*$).

Typed programs $\mathbb{S}^*_{\mathbb{T}}$ are ones where every expression e occurring in the program has a defined type $tp(e)$, see Fig. 3 for the definition of tp. Statements have to be typed as well: for $*e_1 := e_2$ we require $tp(e_1) = tp(e_2)*$, for $v := *e$ we require $tp(e) = t*$ for some $t \in \mathbb{T}_P$, and for **split** e and **join** e, $tp(e) = t*$ for some t.

$$
\begin{array}{lll}
\llbracket e_1 \oplus e_2 \rrbracket_{\mathcal{E}} = \llbracket e_1 \rrbracket_{\mathcal{E}} \oplus \llbracket e_2 \rrbracket_{\mathcal{E}} & \langle \mathcal{E}, \mathcal{B}, (\textbf{assert}\ \psi;\ ss)\rangle & \rhd\ \textbf{if}\ \llbracket \psi \rrbracket_{\mathcal{E}}\ \textbf{then}\ \langle \mathcal{E}, \mathcal{B}, ss\rangle\ \textbf{else}\ \bot \\
\llbracket e_1 \rightsquigarrow f \rrbracket_{\mathcal{E}} = \llbracket e_1 \rrbracket_{\mathcal{E}} \mathbin{\widehat{+}} \mathit{offset}(f) & \langle \mathcal{E}, \mathcal{B}, (\textbf{assume}\ \psi;\ ss)\rangle & \rhd\ \textbf{if}\ \llbracket \psi \rrbracket_{\mathcal{E}}\ \textbf{then}\ \langle \mathcal{E}, \mathcal{B}, ss\rangle\ \textbf{else}\ \top \\
\llbracket (t) e_1 \rrbracket_{\mathcal{E}} = \mathit{cast}(\llbracket e_1 \rrbracket_{\mathcal{E}}, t) & \langle \mathcal{E}, \mathcal{B}, (v := *e_1;\ ss)\rangle & \rhd\ \langle \mathcal{E}[v := \mathit{read}(\mathcal{B}, \llbracket e_1 \rrbracket^{\mathbb{P}}_{\mathcal{E}})], \mathcal{B}, ss\rangle \\
\llbracket v \rrbracket_{\mathcal{E}} = \mathcal{E}(v) & \langle \mathcal{E}, \mathcal{B}, (*e_1 := e_2;\ ss)\rangle & \rhd\ \langle \mathcal{E}, \mathit{write}(\mathcal{B}, \llbracket e_1 \rrbracket^{\mathbb{P}}_{\mathcal{E}}, \llbracket e_2 \rrbracket_{\mathcal{E}}), ss\rangle \\
\llbracket c \rrbracket_{\mathcal{E}} = c & \langle \mathcal{E}, \mathcal{B}, (\textbf{split}\ e_1;\ ss)\rangle & \rhd\ \langle \mathcal{E}, \mathcal{B}, ss\rangle \\
\llbracket e_1 \otimes e_2 \rrbracket_{\mathcal{E}} = \llbracket e_1 \rrbracket_{\mathcal{E}} \otimes \llbracket e_2 \rrbracket_{\mathcal{E}} & \langle \mathcal{E}, \mathcal{B}, (\textbf{join}\ e_1;\ ss)\rangle & \rhd\ \langle \mathcal{E}, \mathcal{B}, ss\rangle \\
 & & \text{where } \llbracket e \rrbracket^{\mathbb{P}}_{\mathcal{E}} = (t, \llbracket e \rrbracket_{\mathcal{E}}) \text{ where } \mathit{tp}(e) = t*
\end{array}
$$

Fig. 4. untyped semantics

3 Untyped Semantics

Next, we define a small-step semantics of our language, where memory is modelled as a sequence of bytes (as in a conventional semantics for C).

The size of a type $|\cdot| : \mathbb{T} \to \mathbb{N}$ is the number of bytes the representation of type occupies in memory. We assume that the size is known for primitive types, e.g. $|\texttt{u8}| = 1$, $\texttt{u64}$ is the biggest primitive type and for every type t we have $|t*| = |\texttt{u64}|$. Given a **struct** $S\ \{f_1 : t_1; \ldots; f_n : t_n\}$, we define $|S| = \Sigma_{i \leq n} |t_i|$, i.e. we assume all padding has been made explicit. The size is well-defined and finite because the structs are acyclic.

Let $\mathbb{B} = 0, 1, \ldots, 255$ be the set of *bytes*, and $\mathbb{B}^*$ be the set of sequences of bytes. The function $\big[\cdot\big]_{\mathbb{N}} : \mathbb{B}^* \to \mathbb{N}$ returns the natural number represented by the given byte sequence; the function $\big[\cdot\big]^n_{\mathbb{B}} : \mathbb{N} \to \mathbb{B}^n$ for $n \geq 0$ defines the sequence of bytes encoding the lowest $8n$ bits of a natural number. These functions are defined by

$$
\begin{aligned}
\big[b_0, b_1, \ldots, b_n\big]_{\mathbb{N}} &= \Sigma^n_{i=0} b_i \cdot 2^{8i} \\
\big[\big[k\big]^n_{\mathbb{B}}\big]_{\mathbb{N}} &= k \text{ for } k < 2^{8n}
\end{aligned}
$$

A *pointer* is a pair of type and memory address, i.e. the set of pointers $\mathbb{P} = \mathbb{T} \times \mathbb{B}^{|\texttt{u64}|}$. A *primitive pointer* is one with primitive type: $\mathbb{P}_P = \mathbb{T}_P \times \mathbb{B}^{|\texttt{u64}|}$.

The function $\mathit{offset} : \mathbb{F} \to \mathbb{N}$ computes the distance of field f in bytes from the beginning of the struct containing f; the function $\cdot \rightsquigarrow \cdot : \mathbb{P} \times \mathbb{F} \to \mathbb{P}$ computes the address of a field within a struct. Given **struct** $S\ \{f_1 : t_1; \ldots; f_n : t_n\}$, we define $\mathit{offset}(f_i) = \Sigma_{j<i} |t_j|$ and $(S, r) \rightsquigarrow f_i = (t_i, r \mathbin{\widehat{+}} \mathit{offset}(f_i))$, where $r \mathbin{\widehat{+}} o = \big[\big[r\big]_{\mathbb{N}} + o\big]^{|\texttt{u64}|}_{\mathbb{B}}$.

The semantics of expressions and statements is defined with respect to an *environment* $\mathcal{E} : \mathbb{V} \to \mathbb{B}^*$ and *byte memory* $\mathcal{B} : \mathbb{P} \to \mathbb{B}^*$. Reading and writing of byte memory via a (typed) pointer (t, r) is defined by:

$$
\begin{aligned}
\mathit{read}(\mathcal{B}, (t, r)) &= \mathcal{B}(\texttt{u8}, r), \ldots, \mathcal{B}(\texttt{u8}, r \mathbin{\widehat{+}} (|t| - 1)) \\
\mathit{write}(\mathcal{B}, (t, r), (v_1, \ldots, v_{|t|})) &= \mathcal{B}[(\texttt{u8}, r) := v_1] \ldots [(\texttt{u8}, r \mathbin{\widehat{+}} (|t| - 1)) := v_{|t|}]
\end{aligned}
$$

Note that these operations assume little-endian (least significant byte first) byte order and need to be redefined for big-endian architectures.

Figure 4 defines the semantics of expressions and formulas via the function $[\![\cdot]\!]_{\mathcal{E}} : \mathbb{E} \to \mathbb{B}^*$, and predicate $[\![\cdot]\!]_{\mathcal{E}} \subset \Psi$, respectively. Note that downcasts result in taking subsequences, upcast result (potentially) in (sign) extensions and interpretations of different $\oplus$ operators return values in the proper range. See Sect. 6 for detailed description of how these operations map to the primitives supported by the theorem prover.

Fig. 4 also defines the semantics of programs by the standard transition relation $\rhd$. Given a state $(\mathcal{E}, \mathcal{B})$ and a statement s from a typed program, the new state is computed according to the $\rhd$ relation defined in Fig. 4. There are two special states of the execution: $\top$ means that the program is *stuck* (i.e., the execution was no longer possible due to some external constraints), while $\bot$ means that the program has *gone wrong* (i.e., there has been an error in the program). The relation $\rhd^*$ is the smallest transitive and reflexive relation containing $\rhd$. The state $\bot$ has a special meaning in verification: the verification conditions we generate state that the program never goes wrong, i.e. $\forall \mathcal{E}, \mathcal{B}.\ \neg((\mathcal{E}, \mathcal{B}, ss) \rhd^* \bot)$. Note that in the byte memory model, the split and join operations are no-ops.

4 Typed Semantics

Next, we present a semantics like that of the last section, but where memory is a collection of typed objects rather than a sequence of bytes.

The *support* of a pointer is a sequence of pointers to bytes where the pointer representation is stored, i.e.: $\mathit{support}(t, r) = (\mathtt{u8}, r), (\mathtt{u8}, r \mathbin{\widehat{+}} 1), \ldots, (\mathtt{u8}, r \mathbin{\widehat{+}} (|t| - 1))$.

The *extent* of a pointer $p = (t, r)$, written $\mathit{extent}(p)$, is the set of pointers that can be obtained from p by applying $\cdot \rightsquigarrow \cdot$ zero or more times:

$$\mathit{extent}(p) = \begin{cases} \{p\} \cup \bigcup_{i=1 \ldots n} \mathit{extent}(p \rightsquigarrow f_i) & \text{where } t \in \mathbb{T}_S, \textbf{struct } t\ \{f_1 : t_1; \ldots; f_n : t_n\} \\ \{p\} & \text{where } t \notin \mathbb{T}_S \end{cases}$$

Note that this is not pointer chasing: the extent is always well-defined and finite because of the acyclicity of struct containment. A set of pointers has the *disjoint roots property* iff its elements have disjoint supports. Let us take a set of pointers $\mathcal{T}$ with disjoint roots property and define $\mathcal{T}^* = \bigcup_{p \in \mathcal{T}} \mathit{extent}(p)$.

Similarly as in the byte case, in Fig. 5 we define a transition relation $\blacktriangleright$ between states consisting of the environment $\mathcal{E}$, the memory $\mathcal{M} : \mathbb{P} \to \mathbb{B}^*$ and the root pointer set $\mathcal{T} \subseteq \mathbb{P}$. This time the memory will only be read and written using primitive pointers (including u8 pointers). There is however an additional requirement, namely that memory can be only read or written at locations from $\mathcal{T}^*$. The only operations modifying $\mathcal{T}$ in $\blacktriangleright$ are the two reinterpretation functions *split* and *join*,

$$
\begin{array}{ll}
\langle \mathcal{E}, \mathcal{M}, \mathcal{T}, (\textbf{assert}\ \psi;\ ss) \rangle & \blacktriangleright\ \textbf{if}\ \llbracket \psi \rrbracket_{\mathcal{E}}\ \textbf{then}\ \langle \mathcal{E}, \mathcal{M}, \mathcal{T}, ss \rangle\ \textbf{else}\ \bot \\
\langle \mathcal{E}, \mathcal{M}, \mathcal{T}, (\textbf{assume}\ \psi;\ ss) \rangle & \blacktriangleright\ \textbf{if}\ \llbracket \psi \rrbracket_{\mathcal{E}}\ \textbf{then}\ \langle \mathcal{E}, \mathcal{M}, \mathcal{T}, ss \rangle\ \textbf{else}\ \top \\
\langle \mathcal{E}, \mathcal{M}, \mathcal{T}, (v := *e_1;\ ss) \rangle & \blacktriangleright\ \textbf{if}\ \llbracket e_1 \rrbracket^{\mathbb{P}}_{\mathcal{E}} \in \mathcal{T}^*\ \textbf{then}\ \langle \mathcal{E}[v := \mathcal{M}(\llbracket e_1 \rrbracket^{\mathbb{P}}_{\mathcal{E}})], \mathcal{M}, \mathcal{T}, ss \rangle\ \textbf{else}\ \bot \\
\langle \mathcal{E}, \mathcal{M}, \mathcal{T}, (*e_1 := e_2;\ ss) \rangle & \blacktriangleright\ \textbf{if}\ \llbracket e_1 \rrbracket^{\mathbb{P}}_{\mathcal{E}} \in \mathcal{T}^*\ \textbf{then}\ \ \langle \mathcal{E}, \mathcal{M}[\llbracket e_1 \rrbracket^{\mathbb{P}}_{\mathcal{E}} := \llbracket e_2 \rrbracket_{\mathcal{E}}], \mathcal{T}, ss \rangle\ \textbf{else}\ \bot \\
\langle \mathcal{E}, \mathcal{M}, \mathcal{T}, (\textbf{split}\ e_1;\ ss) \rangle & \blacktriangleright\ \textbf{if}\ \llbracket e_1 \rrbracket^{\mathbb{P}}_{\mathcal{E}} \in \mathcal{T}\ \textbf{then}\ \langle \mathcal{E}, \mathit{split}(\mathcal{M}, \mathcal{T}, \llbracket e_1 \rrbracket^{\mathbb{P}}_{\mathcal{E}}), ss \rangle\ \textbf{else}\ \bot \\
\langle \mathcal{E}, \mathcal{M}, \mathcal{T}, (\textbf{join}\ e_1;\ ss) \rangle & \blacktriangleright\ \textbf{if}\ \mathit{support}(\llbracket e_1 \rrbracket^{\mathbb{P}}_{\mathcal{E}}) \subseteq \mathcal{T}\ \textbf{then}\ \langle \mathcal{E}, \mathit{join}(\mathcal{M}, \mathcal{T}, \llbracket e_1 \rrbracket^{\mathbb{P}}_{\mathcal{E}}), ss \rangle\ \textbf{else}\ \bot
\end{array}
$$

Fig. 5. Typed semantics

defined as follows:

$$
\begin{aligned}
\mathit{split}(\mathcal{M}, \mathcal{T}, p) &= \langle \mathit{write}(\mathcal{M}, p, \mathcal{M}(p)), \mathcal{T} \setminus \{p\} \cup \mathit{support}(p) \rangle \\
\mathit{join}(\mathcal{M}, \mathcal{T}, p) &= \langle \mathcal{M}', \mathcal{T} \setminus \mathit{support}(p) \cup \{p\} \rangle \\
&\qquad\qquad \text{where } \mathcal{M}'(q) = \textbf{if}\ q \in \mathit{extent}(p)\ \textbf{then}\ \mathit{read}(\mathcal{M}, q)\ \textbf{else}\ \mathcal{M}(q)
\end{aligned}
$$

Intuitively *split* exchanges a pointer to its support in $\mathcal{T}$; *join* works the other way round. Note that in each case the exchanged sets of pointers have equal sum of supports, so that $\blacktriangleright$ maintains the disjoint roots property. Furthermore all the new pointers in $\mathcal{T}^*$ are given an interpretation in $\mathcal{M}$, based on values stored at the old pointers.

4.1 Usage of Split And Join

For example, one might use a join an array of bytes returned from the memory allocator, into an object of some particular type. Then one would use fields of the newly allocated object in a typed way. At some point one might want to split it back into array of bytes, use `memcpy()` to copy it somewhere, and then join it back again. Finally, one needs to split it and pass the array of bytes back to the memory allocator.

The split and join also allow for accessing individual bytes, e.g., in a floating point number representation.

On the other hand, one does not need to split or joing to access specific fields of a typed object. This is because the composite object can not be accessed as a single entity, and operations like struct assignment are modelled member-wise.

4.2 Embeddings

Given a **struct** $S\ \{f_1 : t_1; \ldots; f_n : t_n\}$ for any r the $\mathit{support}(S, r)$ is a disjoint union of $\mathit{support}((S, r) \rightsquigarrow f_i)$ for $i = 1 \ldots n$. Therefore different fields of an object can never overlap, and neither can fields of objects with disjoint supports. Figure 6 describes this graphically. In each configuration the supports for pointers in $\mathcal{T}^*$ look like a set of disjoint boxes ($\mathcal{T}$) subdivided into smaller boxes (fields at level one of nesting), subdivided into even smaller ones (fields at level two) and so on,

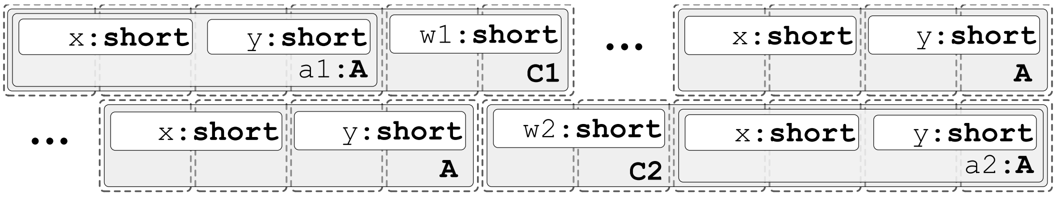

Fig. 6. Example of an embedding forest

until we get to primitive types, with single-element extents (but possibly multi-byte supports). Each box is labelled with a type. The boxes never overlap. So for each inner box, there is a single smallest (in the sense of nesting, not support size) box containing it (its *embedding*) and a single field name written on it (its *path*).

This intuition is captured by the notion of an *embedding graph* and *forest*. An *embedding graph* of $A \subseteq \mathbb{P}$ is a directed multi-graph, vertices of which are pointers drawn from A and there is an edge from p to q labelled f iff $p \rightsquigarrow f = q$.

Lemma 4.1 *If $\mathcal{T}$ has the disjoint roots property then the embedding graph of $\mathcal{T}^*$ is a forest, with at most one edge between any two vertices.*

4.3 SMT Axiomatization

Given $\mathcal{T}$ we define the function $\mathit{embedding}(\mathcal{T}, p) : \mathbb{P}$ that returns the parent of p in the embedding forest of $\mathcal{T}^*$. If $p \in \mathcal{T}$ (that is it has no parent) then $\mathit{embedding}(\mathcal{T}, p) = p$. Similarly the function $\mathit{path}(\mathcal{T}, p) : \mathbb{F} \cup \{f_\perp\}$ returns the label of the incoming edge ending in p, and $f_\perp$ for $p \in \mathcal{T}$. These function have the following property:

$$\forall p, f, \mathcal{T}.\ p \in \mathcal{T}^* \Rightarrow p \rightsquigarrow f \in \mathcal{T}^* \wedge\ \mathit{embedding}(\mathcal{T}, p \rightsquigarrow f) = p \wedge\ \mathit{path}(\mathcal{T}, p \rightsquigarrow f) = f$$

We present it as an axiom to the SMT solver. It is instantiated (triggers) whenever a term of the form $p \rightsquigarrow f \in \mathcal{T}^*$ appears, which is, by different parts of axiomatization, whenever a $p \rightsquigarrow f$ is accessed in a state, where the set of valid objects is $\mathcal{T}$.

For example, let us take accesses to $p_1 = a_1 \rightsquigarrow b_1 \rightsquigarrow c_1$ and $p_2 = a_2 \rightsquigarrow b_2 \rightsquigarrow c_2$, done in the same $\mathcal{T}$. We want to conclude $p_1 \neq p_2$. For p_1 the SMT solver computes $\mathit{embedding}(\mathcal{T}, p_1) = a_1 \rightsquigarrow b_1$ and $\mathit{path}(\mathcal{T}, p_1) = c_1$, while for p_2 is is $\mathit{embedding}(\mathcal{T}, p_2) = a_2 \rightsquigarrow b_2$ and $\mathit{path}(\mathcal{T}, p_2) = c_2$. If $c_1 \neq c_2$ we are done. Otherwise, we need to check if $a_1 \rightsquigarrow b_1 \neq a_2 \rightsquigarrow b_2$. Using similar reasoning (the axiom above will be triggered again), if $b_1 \neq b_2$, we are done, otherwise we look for $a_1 \neq a_2$, which will hopefully follow from preconditions. Thus the axiom above allows for distinguishing between pointers, if any element of the path leading to them is different.

5 Equivalence of the Untyped and Typed Semantics

Next, we show that the two semantics we have defined are equivalent, in the sense that the choice of model effect neither whether the program goes wrong nor the final state.

5.1 Soundness

Soundness states that if the computation in the untyped model goes wrong then so does the computation in the typed model.

We define the following correspondence between typed and untyped memories:

$$\mathcal{B} \approx_{\mathcal{T}} \mathcal{M} \quad \text{iff} \quad \forall p \in \mathcal{T}^*.\ \mathcal{M}(p) = \mathit{read}(\mathcal{B}, p).$$

Starting from corresponding memories, Lemma 5.1 says that a single transition in both system that doesn't get stuck or goes wrong has corresponding effects.

Lemma 5.1 *If* $\mathcal{B} \approx_{\mathcal{T}} \mathcal{M}$ *and* $\langle \mathcal{E}, \mathcal{B}, ss\rangle \rhd \langle \mathcal{E}', \mathcal{B}', ss'\rangle$, $\langle \mathcal{E}, \mathcal{M}, \mathcal{T}, ss\rangle \blacktriangleright \langle \mathcal{E}'', \mathcal{M}', \mathcal{T}', ss''\rangle$ *then* $\mathcal{E}' = \mathcal{E}''$, $ss' = ss''$ *and* $\mathcal{B}' \approx_{\mathcal{T}'} \mathcal{M}'$.

Lemma 5.2 states that if a transition in the untyped model goes wrong, then so does the corresponding transition in typed model:

Lemma 5.2 *If* $\mathcal{B} \approx_{\mathcal{T}} \mathcal{M}$ *and* $\langle \mathcal{E}, \mathcal{B}, ss\rangle \rhd \bot$ *then* $\langle \mathcal{E}, \mathcal{M}, \mathcal{T}, ss\rangle \blacktriangleright \bot$.

Let $\mathcal{T}_1 = \{\mathtt{u8}\} \times \mathbb{B}^{|\mathtt{u64}|}$. Observe that $\mathcal{B} \approx_{\mathbb{T}_1} \mathcal{B}$.

Theorem 5.3 *If* $\langle \mathcal{E}, \mathcal{B}, ss\rangle \rhd^* \bot$ *then* $\langle \mathcal{E}, \mathcal{B}, \mathbb{T}_1, ss\rangle \blacktriangleright^* \bot$.

5.2 Completeness

Completeness states that if computation in the untyped memory model terminates, then computation in the typed semantics terminates with a corresponding memory.

Let $[\cdot]_1 : \mathbb{S}^* \to \mathbb{S}^*$ be a transformation adding join/split around any memory access, and removing explicit join/splits, i.e.:

$$\begin{aligned}
[s;\, ss]_1 &= \begin{cases} \textbf{join}\ e_1;\ s;\ \textbf{split}\ e_1;\ [ss]_1 & \text{where } s \in \{*e_1 := e_2, v := *e_1\} \\ [ss]_1 & \text{where } s \in \{\textbf{join}\ e_1, \textbf{split}\ e_1\} \\ s;\ [ss]_1 & \text{otherwise} \end{cases} \\
[\epsilon]_1 &= \epsilon
\end{aligned}$$

Theorem 5.4 *If* $\langle \mathcal{E}, \mathcal{B}, ss\rangle \rhd^* \langle \mathcal{E}', \mathcal{B}', ss'\rangle$ *then* $\langle \mathcal{E}, \mathcal{B}, \mathcal{T}_1, [ss]_1\rangle \blacktriangleright^* \langle \mathcal{E}', \mathcal{B}', \mathbb{T}_1, [ss']_1\rangle$.

Proof. If $[ss]_1$ does not get stuck or go wrong, we have the correspondence from Lemma 5.1. Otherwise the only difference between ss and $[ss]_1$ are the additional conditions on memory accesses. They are however always OK since for any newly introduced **join** e_1, $\mathcal{T} = \mathcal{T}_1$ and for all other operations there was a preceding **join** e_1. □

While using $[\cdot]_1$ on programs removes any advantage of using the typed memory, it shows that when precision is needed, the typed model can be forced into a untyped model thus allowing mixed untyped and type reasoning.

6 Extensions

In this section we discuss how our core language can be extended to capture other C types and objects. But before doing so we investigate how sequences of bytes, representing primitive values, are mapped into objects from a theory understood by an automatic theorem prover. A natural candidate would be to use fixed size bit-vectors as the underlying theory. While this is very precise (all the machine arithmetic operations are modelled with bit-level precision), the resulting performance was unsatisfactory. We therefore decided to represent primitive values, and their corresponding operations, by a much weaker, but also much faster theory — *linear integer arithmetic*. We map byte sequences of length n into integers between -2^{8n-1} and $2^{8n-1}-1$ or between 0 and $2^{8n}-1$, depending if the type is signed or unsigned. We introduce function symbols for each C operator on each integer type t. Then we axiomatize all operations. Unsigned 64-bit integer addition, is for example axiomatized as follows:

$$\forall x,y.\ 0 \leq x+y \leq 2^{64}-1 \Rightarrow add_{\mathtt{u64}}(x,y) = x+y$$
$$\forall x,y.\ \ 0 \leq add_{\mathtt{u64}}(x,y) \leq 2^{64}-1$$

This axiomatization, while incomplete, seems sufficient in most cases. We use a similar trick for axiomatization of type conversions, which are only defined if the given value falls within the cast's target range.

Since many programmers miss overflows, we generate by default additional assertions before each arithmetic operation requiring that the computed value will fit into the target range. In fact, if the value fits range of `u64`, then $add_{\mathtt{u64}}$ coincides with linear arithmetic operator +, so all the usual arithmetic laws hold. The generation of these assertions can be suppressed in case the user wants to reason about overflows.

6.1 Arrays

We extend our core language to allow embedded arrays inside of structs, as in **struct** $S\ \{\ldots f : t[n]\ldots\}$. We shall treat f as n separate fields $f^{[0]} : t \ldots f^{[n-1]} : t$. Therefore we extend the set of fields and expressions as follows:

$$\begin{array}{lcl} \mathbb{F} & ::= & \ldots \mid \mathbb{F}^{[\mathbb{N}]} \qquad \mathbb{E} ::= \ldots \mid \mathbb{E}[\mathbb{E}] \\ tp(e_1[e_2]) & = & t \text{ where } tp(e_1) = t{*}, tp(e_2) \in \mathbb{T}_I \\ [\![e_1[e_2]]\!]_{\mathcal{E}} & = & [\![e_1]\!]_{\mathcal{E}} \mathbin{\widehat{+}} (\left[[\![e_2]\!]_{\mathcal{E}}\right]_{\mathbb{N}} \cdot |tp(e_1[e_2])|) \end{array}$$

The relationship between embedding and index computation is similar to the normal field address computation:

$$\forall p,q,i,f.\ p \in \mathcal{T}^* \wedge\ q = p{\rightsquigarrow}f[i] \wedge\ 0 \leq i \leq n \Rightarrow$$
$$p \in \mathcal{T}^* \wedge\ embedding(\mathcal{T},q) = p \wedge\ path(\mathcal{T},q) = f^{[i]}$$

For cases where an array is allocated outside of a struct, we introduce a parametric array type $\texttt{array} : \mathbb{T} \times \mathbb{N} \to \mathbb{T}$, and treat the array of type t with n elements, as if it was an embedded array of $\texttt{array}(t, n)$.

6.2 Unions

For unions we have the additional complication that only one of the fields should be considered typed at any given point. Therefore for union U with fields $f_1, ..., f_n$ we introduce n struct types $U_1, ..., U_n$ and use the reinterpretation operations *join* and *split* to switch between them. Note that this is only needed when a union is used in the sense of discriminating union. Another common use of unions in C code (and actually fairly common the OS code) is to interpret several fields of integer types (particularly bitfields) as one integer type. This is covered below.

6.3 Bit-fields

C allows the definition of bit-fields in structured types, which are interpreted as a signed or unsigned integer type with the corresponding number of bits. Since most architectures do not allow for direct access to arbitrary bit ranges in memory, C compilers usually merge one or more consecutive bit-fields into a single underlying field of unsigned integer type. Accesses to particular bit-fields will then be transformed into bit manipulations on the underlying field. That is why C does not allow taking the address of a bit-field. We extend our expression language to accommodate for the additional bit manipulations:

$$\begin{array}{lcl}
\mathbb{E} & ::= & \ldots \mid \mathbb{E}\langle \mathbb{N} : \mathbb{N} \rangle \mid \mathbb{E}[\langle \mathbb{N} : \mathbb{N} \rangle := \mathbb{E}] \mid e_{1 \pm \mathbb{N}} \\
tp(e_1\langle a : b \rangle) & = & \texttt{u64} \;\; \text{where } a \leq b, tp(e_1) = \texttt{u64} \\
tp(e_1[\langle a : b \rangle := e_2]) & = & \texttt{u64} \;\; \text{where } a \leq b, tp(e_1) = tp(e_2) = \texttt{u64} \\
tp(e_{1 \pm b}) & = & \texttt{i64} \;\; \text{where } b > 0, tp(e_1) = \texttt{u64}
\end{array}$$

where for simplicity we assume only 64 bit underlying fields. The operation $e_1\langle a : b \rangle$ extracts bits between a and b inclusive from e_1; the operation $e_1[\langle a : b \rangle := e_2]$ replaces bits between a and b in e_1 with e_2; the operation $e_{1 \pm b}$ performs a sign extension from b to 64 bits. Formally:

$$\begin{aligned}
[\![e\langle a : b \rangle]\!]_{\mathcal{E}} &= \left[([\![e]\!]^{\mathbb{N}}_{\mathcal{E}} \operatorname{div} 2^a) \operatorname{mod} 2^{b-a+1} \right]^8_{\mathbb{B}} \\
[\![e_{\pm b}]\!]_{\mathcal{E}} &= \textbf{if } [\![e]\!]^{\mathbb{N}}_{\mathcal{E}} < 2^{b-1} \textbf{ then } [\![e]\!]_{\mathcal{E}} \textbf{ else } \left[-2^b + [\![e]\!]^{\mathbb{N}}_{\mathcal{E}} \right]^8_{\mathbb{B}} \\
[\![e_1 \left[\langle a : b \rangle := e_2 \right]]\!]_{\mathcal{E}} &= \left[[\![e_1]\!]^{\mathbb{N}}_{\mathcal{E}} \operatorname{mod} 2^a + 2^a \cdot [\![e_2\langle 0 : b - a + 1 \rangle]\!]^{\mathbb{N}}_{\mathcal{E}} + 2^b \cdot ([\![e_1]\!]^{\mathbb{N}}_{\mathcal{E}} \operatorname{div} 2^b) \right]^8_{\mathbb{B}} \\
\text{where} \quad [\![e]\!]^{\mathbb{N}}_{\mathcal{E}} &= \left[[\![e]\!]_{\mathcal{E}} \right]_{\mathbb{N}}
\end{aligned}$$

Consider struct `X64VirtualAddress` from Fig. 7. Our translation maps all bit-fields into a single field $\texttt{bf0} : \texttt{u64}$. Here are some resulting translations:

```
struct X64VirtualAddress {
  i64 PageOffset:12; //  <0:11>
  u64 PtOffset   : 9; // <12:20>
  u64 PdOffset   : 9; // <21:29>
  u64 PdptOffset: 9; // <30:38>
  u64 Pml4Offset: 9; // <39:47>
  u64 SignExtend:16; // <48:64>
};
```

```
union X64VirtualAddressU {
  X64VirtualAddress Address;
  u64 AsUINT64;
};
union Register {
  struct { u8 l, h; } a;
  u16 ax;
  u32 eax; };
```

Fig. 7. A structure with bit-fields, a union using it and a almost-bit-field union

$$\begin{array}{lll}
\texttt{*q = p->PdOffset;} & \Rightarrow & tmp := *p \rightsquigarrow \mathtt{bf0};\ *q := tmp\langle 21:29\rangle \\
\texttt{p->PdOffset = x;} & \Rightarrow & tmp := *p \rightsquigarrow \mathtt{bf0};\ *p \rightsquigarrow \mathtt{bf0} := tmp[\langle 21:29\rangle := x] \\
\texttt{*q = p->PageOffset;} & \Rightarrow & tmp := *p \rightsquigarrow \mathtt{bf0};\ *q := tmp\langle 0:11\rangle_{\pm 12} \\
\texttt{p->PageOffset = x;} & \Rightarrow & tmp := *p \rightsquigarrow \mathtt{bf0};\ *p \rightsquigarrow \mathtt{bf0} := tmp[\langle 0:11\rangle := (\mathtt{u64})x]
\end{array}$$

To discharge formulas involving bitfields, we had been using a decision procedure for fixed size bit vector arithmetic. But as it turns out, this approach places a strong burden on the SMT solver and leads to unacceptable performance for even moderate complexity problems.

To our rescue it turns out that bit-fields are typically only used for compact storage of related information or to exactly map hardware data structures. As such, interaction between bit-fields and arithmetic is rather uncommon. (What is the point in summing up page table entries?). Thus we axiomatized bit-selection and concatenation:

$$\begin{array}{lll}
0 \le n < 2^{b-a} & \Rightarrow v\big[\langle a:b\rangle := n\big]\langle a:b\rangle & = n \\
-2^{c-b} \le k < 2^{c-b} & \Rightarrow (v[\langle b:c\rangle := (\mathtt{u64})k]\langle b:c\rangle)_{\pm c-b+1} & = k \\
b' < a \ \lor\ b < a' & \Rightarrow v\big[\langle a:b\rangle := n\big]\langle a':b'\rangle & = v\langle a':b'\rangle
\end{array}$$

These properties are essentially the same as the usual select-of-store axioms [19] used for array decision procedures – they are very suitable for modern theorem provers supporting quantification.

On top of that we provide axioms for some limited interaction with arithmetic, like special properties of 0 and bit-shifts:

$$\begin{array}{l}
0\langle a:b\rangle = 0 \\
a \ge n \Rightarrow (2^n \cdot v)\langle a:b\rangle = v\langle a-n:b-n\rangle \\
(v \operatorname{div} 2^n)\langle a:b\rangle = v\langle a+n:b+n\rangle
\end{array}$$

6.4 Bitfields and Unions

Consider the union `X64VirtualAddressU` from Fig. 7. This is very typical use of a union in operating system code: the `AsUINT64` field is used to change the value of all bit-fields at once. The struct `X64VirtualAddress` is used to access individual bits. However after having applied the transformation introduced in the previous section this struct is now also represented by a single backing field of type `u64`. As a consequence we are currently looking at a union with two fields of the same type. This allows further normalization: we simply express operations on one field in terms of the other, which in effect eliminates the struct containing the bit-field and the union altogether.

We also treat fields of small integer types used inside of unions, as if they were bit-fields. For example consider union `Register` from Fig. 7. Member operations on registers, whether they relate to the fields `l,h,a,ax` or `eax` are translated into bit-field accesses and are thus completely compiled away.

These two simple transformation cover most unions within Microsoft's Hyper-V.

6.5 Globals, Stack and Heap

Our memory models did not distinguish between heap, stack and the global memory, however it introduced locals. In fact, they correspond to C's local variables, provided they are are never accessed through the address-of operator.

C's global variables sit somewhere in memory, in a location that is typed when the program starts. Global variables have the disjoint roots property.

The C language does not really have a notion of heap — all one can do is to allocate chunks of possible varying size from the operating system (if there is one). Since our model supports arbitrary reinterpretation of data, it is fine to allocate array of bytes from the OS and then treat them as different types using *join*. Note that each successful allocation extends the program's disjoint roots.

If in C a local variable of type t is accessed via the address-of operator then the translation introduces a local variable of $t*$, which is initialized with the result of a memory allocation of size $|t|$.

6.6 Memory Protection

We have no special treatment of memory protection in our memory models. We just assume that every memory location can be accessed. However for most operating systems, let alone application programs, this is not true. If necessary this restriction is easy to enforce in the typed model — we just need to restrict the root pointer set to *allocated locations* (from C memory allocator, the operating system or some hardware memory management unit).

7 Evaluation

The aforementioned memory axiomatization has been implemented in the Verifying C Compiler (VCC), a sound C verifier being used to verify the functional correctness of the Hypervisor (the virtualization kernel of Microsoft's Hyper-V product []). VCC translates annotated C code into BoogiePL [10], an intermediate language for verification. The verification condition generator Boogie [4] takes BoogiePL as input, and feeds the generated verification conditions into the Z3 [9] SMT solver. VCC is available for academic use and can be downloaded from the Microsoft Research website.

VCC was originally built on top of the untyped memory model using the bitvector decision procedure (DP) in Z3 to perform precise split and join operations.

This resulted in very poor performance, particularly when combined with quantified sub-formulas needed to prove functional correctness.

We next decided to drop bit-vector DP and instead go to the linear integer arithmetic DP. Now we stored entire composite values in memory, but to work around soundness problems with possibly overlapping regions, memory could no longer be treated as a simple map. We introduced a Variable Sized Word (VSW) memory model, characterized by axioms saying that writes through primitive pointers commute as long as their supports are disjoint. This required extensive annotations talking about disjointness of memory regions. We were however able to verify that a C simulation of Windows based Smart Card (approx 1000 lines) runs in a sandbox [13] and the memory safety as well as partial functional correctness of "baby" hypervisor [2], which is a C simulation of a simple CPU architecture along with a hypervisor (about 1000 lines). Additionally we verified parts of the Microsoft Hypervisor, including the memory safety of approx 4500 lines of the x86 assembly code by translating it into C and making the machine state explicit [17]. On the other hand we were still unable to verify recursive data-structures, such as doubly linked lists and red-black trees. The functional part of invariants was deeply buried inside statements about disjointness of memory regions which confused both the annotator and the prover.

This led to development of the memory model described in this paper. So far we have been able to verify an implementation of doubly linked list with full functional specification (about 500 lines, no function takes longer than 20s to verify). We have also verified implementation of concurrency primitives like spin-locks and reader-writer locks [14] as well as some lock-free data structures (verification times are usually in the couple-seconds range). Additionally we have ported the test suite of the old VCC (about 10000 lines of code) to the new version. The new memory model (along with the ownership-based methodology) allowed for specification of almost all types in the Hypervisor as well as substantial parts of the executable code.

The VSW model is easier on VCC than the untyped model, because each write modifies memory at a single point. However, to check whether writes commute, the prover still needs to reason about disjointness of memory regions. Various statistics produced by the prover have shown that this was where the majority of time was spent. In the typed model we can treat memory as a simple map, which means that updates happen at a single place, and proving commutativity of writes is as easy as proving pointer inequality. Pointer inequality reasoning is simplified by the inclusion of type information in the pointer (i.e., writes through pointers of different types always commute) and by the fact that we have the $\mathit{embedding}(\cdot,\cdot)$ and $\mathit{path}(\cdot,\cdot)$ functions.

8 Related Work

Deductive verification of low level system's code has recently received much attention. Here we only discuss directly comparable verifiers. VCC follows largely the design of Spec# [5]. From Spec# we also adopted its verification machinery [4]. Havoc [1], another C verifier developed at Microsoft Research, also tries to address the verification of low level system's code. However it is not sound. The architecture of VCC is similar to the architecture of Caduceus [12] and Escher's C compiler [8]. Caduceus, like ESC/Java maps field names to separately updateable memories. This helps with antialiasing but hinders sound verification of low level, address manipulating code. KeY-C [18] is a verifier for C that uses dynamic-logic instead of our first-order framework. The L4 kernel verification [22] uses the untyped memory model (based on the embedding of C0 in HOL [20]), but uses a simulation of separation logic in HOL to achieve better alias control.

Except for the L4 kernel verifier, none of these verifiers deals with unions and bit fields. The memory model presented here is similar to the embedding of C in Coq developed as part of the ongoing certification of a moderately-optimising C compiler [16]. The SPARK programming language, a subset of Ada, has its own verifier [3]. SPARK avoids the issues with anti-aliasing and dangling pointers by disallowing allocation at run time entirely.

The architecture and memory model [7] of HAVOC are both similar to ours. The main difference is the goal: we aim at a sound verifier for complex functional properties with whatever annotations are necessary, while HAVOC aims at (unsound) property checking and bug finding with as little annotations as possible. The design choices in the memory model thus reflect that: we offer byte granularity of pointer values and precisely model partial overlaps (the HAVOC paper mentions that as a possible extension, but does not discuss further) and allow for arbitrary changes of type assignment at runtime, which is needed to prove correctness of components like memory allocator but also for something as simple as implementation of byte-copy of a struct. Additionally our modelling of embedded structs seems more natural and slightly stronger, for example in the HAVOC model one would be unable to prove the assertion in `qux` from Fig. 2. The direct performance comparison is difficult because of the unsound assumptions used in HAVOC, however as far as the memory model is concerned, the tasks for the prover are rather similar.

The recent work by Tuch [21] describes a memory model similar to ours, but in context of interactive theorem proving. The memory is also type-indexed and the model allows for reinterpretation. The major difference is that we decided not to support first class struct types. In other words, the heap only stores values of primitive types (integers, floats and pointers). Thus a C memory update corresponds to exactly one update of the modelled heap, while in Tuch's model it needs to affect heaps corresponding to all embedding structs. Such abstraction seems useful in interactive theorem proving, where the user might decide which is the relevant levels of embedding at any given point. On the other hand we do not see how the automatic SMT solver would make such decisions. Additionally, our higher-level methodology allows for reasoning at the level of entire ownership domains, which might include multiple objects, but boundaries of which are orthogonal to physical structure embedding.

Acknowledgements

Authors would like to thank Markus Dahlwaid, Leven Desmet, Mark Hillebrand, Dirk Leinenbach, Rustan Leino, Elena Petrova, Wolfgang Paul, Thomas Santen, Norbert Schirmer, Herman Venter, for help in implementing and experimentally validating the ideas described in this paper. The Hypervisor verification is part of the Verisoft XT project supported by BMBF under grant 01IS07008.

References

[1] The HAVOC property checker, 2008. http://research.microsoft.com/projects/havoc/.

[2] Eyad Alkassar and Wolfgang Paul. On the verification of a "baby" hypervisor for a RISC machine; draft 0, January 2008. http://www-wjp.cs.uni-sb.de/lehre/vorlesung/rechnerarchitektur/ws0607/layouts/hypervisor.pdf.

[3] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison Wesley, New York, NY, 2003.

[4] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[6] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, February 2009. Available from http://research.microsoft.com/pubs.

[7] Jeremy Condit, Brian Hackett, Shuvendu Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. Technical Report MSR-TR-2008-96, Microsoft Research, 2008. To appear at POPL2009.

[8] David Crocker and Judith Carlton. Verification of c programs using automated reasoning. 2007. to appear.

[9] Leonardo de Moura and Nikolaj Bjrner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[10] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

[11] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[12] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[13] Yuri Gurevich and Charles Wallace. Specification and verification of the windows card runtime environment using abstract state machines. Technical Report MSR-TR-99-07, Microsoft Research, 1999.

[14] Mark A. Hillebrand and Dirk C. Leinenbach. Formal verification of a reader-writer lock implementation in C. In *4th International Workshop on Systems Software Verification (SSV 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science B.V., 2009. To appear.

[15] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, December 1999.

[16] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM.

[17] Stefan Maus, Michal Moskal, and Wolfram Schulte. Vx86: x86 assembler simulated in c powered by automated theorem proving. In José Meseguer and Grigore Rosu, editors, *AMAST*, volume 5140 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2008.

[18] Oleg Murk, Daniel Larsson, and Reiner Hahnle. Key-c: A tool for verification of c programs. In *CADE 21*, 2007. to appear.

[19] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[20] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL.* PhD thesis, Technische Universität München, 2006.

[21] 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*, page 59, 2009. to appear.

[22] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.