On the Use of Underspecified Data-Type Semantics for Type Safety in Low-Level Code*

Hendrik Tews Marcus Völp
Technische Universität Dresden, Germany
{tews,voelp}@os.inf.tu-dresden.de

Tjark Weber
Uppsala University, Department of IT, Sweden
tjark.weber@it.uu.se

In recent projects on operating-system verification, C and C++ data types are often formalized using a semantics that does not fully specify the precise byte encoding of objects. It is well-known that such an underspecified data-type semantics can be used to detect certain kinds of type errors. In general, however, underspecified data-type semantics are unsound: they assign well-defined meaning to programs that have undefined behavior according to the C and C++ language standards. A precise characterization of the type-correctness properties that can be enforced with underspecified data-type semantics is still missing. In this paper, we identify strengths and weaknesses of underspecified data-type semantics for ensuring type safety of low-level systems code. We prove sufficient conditions to detect certain classes of type errors and, finally, identify a trade-off between the complexity of underspecified data-type semantics and their type-checking capabilities.

1 Introduction

The formalization of C with abstract-state machines by Gurevich and Huggins [5], Norrish's C++ semantics in HOL4 [12] and the operating-system verification projects VFiasco [7], l4.verified [10] and Robin [16] all use a semantics of C or C++ data types that employs (untyped) byte sequences to encode typed values for storing them in memory. An underspecified, partial function converts byte sequences back into typed values.

We use the term *underspecified data-type semantics* to refer to such a semantics of data types that converts between typed values and untyped byte sequences while leaving the precise conversion functions underspecified. With an underspecified data-type semantics, it is unknown during program verification which specific bytes are written to memory.

The main ingredients of underspecified data-type semantics are two functions — *to_byte* and *from_byte* — that convert between typed values and byte sequences. The function *from_byte* is in general partial, because not every byte sequence encodes a typed value. For instance, consider a representation of integers that uses a parity bit: $\mathit{from\_byte}^{\texttt{int}}$ would be undefined for byte sequences with invalid parity.

Underspecified data-type semantics are relevant for the verification of low-level systems code. This includes code that needs to maintain hardware-controlled data structures, e.g., page directories, or that contains its own memory allocator. Type and memory safety of such low-level code depend on its functional correctness and are undecidable in general. For this reason, type safety for such code can only be established by logical reasoning and not by a conventional type system. As a consequence, this paper focuses on data-type *semantics* instead of improving the type system for, e.g., C++.

Having to establish type correctness by verification is not as bad as it first sounds. With suitable lemmas, type correctness can be proved automatically for those parts of the code that are statically type

*This work was in part funded by the European Commission through PASR grant 104600, by the Deutsche Forschungsgemeinschaft through the QuaOS project, and by the Swedish Research Council.

Submitted to:
SSV 2012

correct [17]. Thereby, the type-correctness property can be precisely tailored to the needs of the specific verification goals, for instance, by taking assumptions about hardware-specific data types into account.

It has long been known that underspecified data-type semantics can detect certain type errors during verification, and thus imply certain type-correctness properties [6]. Because the encoding functions *to_byte* for different types T and U are a priori unrelated, programs are prevented from reading a T-encoded byte sequence with type U. Any attempt to do so will cause the semantics to become stuck, and program verification will fail.

However, additional assumptions, which are often necessary to verify machine-dependent code, easily void this property. For instance, if one assumes that the type `unsigned int` can represent all integer values from 0 to $2^n - 1$ on n-bit architectures, $\mathit{from\_byte}^{\texttt{unsigned int}}$ becomes total for cardinality reasons. Consequently, *any* sequence of n bits becomes a valid encoding of a value of this type.

Despite the widespread use of underspecified data-type semantics for the verification of systems code, a precise characterization of the type-correctness properties that these semantics can enforce is still missing.

In this paper, we investigate different kinds of type errors and different variants of underspecified data-type semantics. We provide sufficient conditions for the fact that a certain semantics can prove the absence of certain type errors and describe the trade-off between the complexity of the semantics and the type errors it can detect. One key insight is that the simple underspecified data-type semantics that we advocated before [6, 15] is only sound for trivially copyable data [9, §3.9] under strong preconditions, which are typically violated in low-level systems code. Type correctness for non-trivially copyable data in the sense of C++ requires a rather complicated semantics that exploits protected bits, see Sect. 4.3.

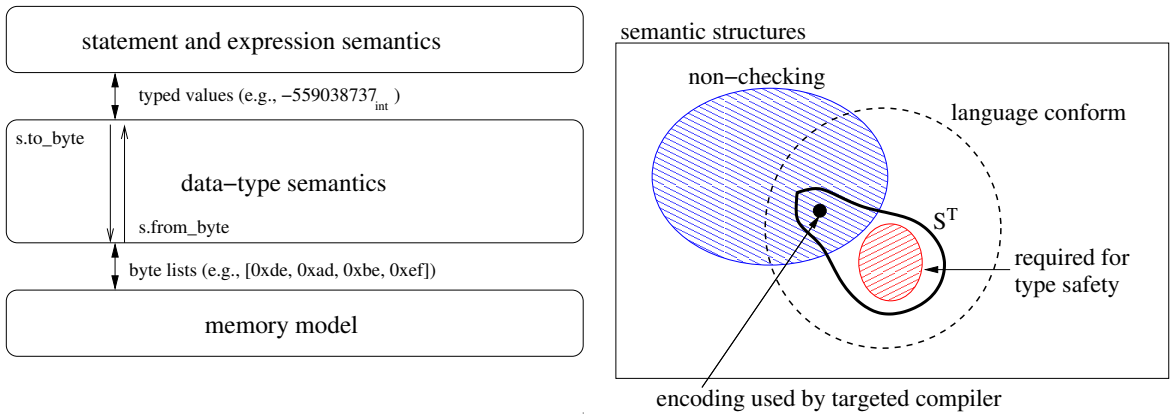
The remainder of this paper is structured as follows: in the next section, we recollect the formalization of underspecified data-type semantics. Sect. 3 describes our classes of type errors. In Sect. 4, we formally define *type sensitivity* as a new type-correctness property that rules out these errors, and discuss the type sensitivity of three different variants of underspecified data-type semantics. Sect. 5 formally proves sufficient conditions for type sensitivity and Sect. 6 discusses related work. For space reasons, a small case study that exemplifies our approach has been moved to App. A. Part of our results have been formalized in the theorem prover PVS [13]. The corresponding sources are publicly available.[1]

2 The Power of Underspecified Data Types

Type checking with underspecified data-type semantics is rooted in the observation that many programming languages do not fully specify the encoding of typed values in memory. For instance, the programming languages Java [3] and Go [2] leave language implementations (compilers and interpreters) complete freedom in how much memory they allocate, and how values are encoded in the bytes that comprise an object in memory. The standards of the programming languages C [8] and C++ [9] also leave encoding and object layout (including endianness and padding) mostly unspecified, with few restrictions: e.g., object representations must have a fixed (positive) size.

The data-type semantics associates a semantic structure s^T (defined below) with each primitive language type T. Semantic structures provide conversion functions between typed values and their memory representation: $s^T.\mathit{to\_byte}$ translates values of type T into untyped byte lists, and $s^T.\mathit{from\_byte}$ translates byte lists back into typed values. The data-type semantics thereby provides an abstract interface that connects the high-level semantics of the language's statements and expressions to a byte-wise organized

[1] At `http://os.inf.tu-dresden.de/~voelp/sources/type_sensitive.tar.gz`



(a) Semantics stack (b) Approach

Figure 1: Type checking with underspecified data-type semantics.

memory model (Fig. 1(a)). As a beneficial side effect, this abstraction allows us to omit from this paper both the details of the statement/expression semantics and the details of the memory model (which includes virtual-to-physical memory mappings and memory-mapped devices [15]).

The key idea behind ensuring type safety with underspecified data-type semantics is taken from [6], and illustrated in Fig. 1(b). The conversion functions $s^T.to\_byte$ and $s^T.from\_byte$ are typically underspecified.[2] The precise requirements on these functions depend on the type T (and possibly on additional factors such as the targeted compiler and hardware architecture). For instance, C requires that the type `unsigned char` uses a "pure binary" encoding [8, §6.2.6.1]. Let $\mathbb{S}^T$ denote the set of all semantic structures that meet these requirements.

Program verification is carried out against an arbitrary (but fixed) structure $s^T \in \mathbb{S}^T$, for each primitive type T. Therefore, verification succeeds only if it would succeed for every possible choice of structures $s^T \in \mathbb{S}^T$ for every primitive type T.

One can think of each structure s^T as a specific way a compiler implements objects of type T. The set $\mathbb{S}^T$ should contain semantic structures that correspond to realistic compilers. These structures typically perform little or no runtime type checking, but their inclusion guarantees that verification results apply to code that is compiled and run on existing hardware.

Provided that the encoding of T is sufficiently underspecified, $\mathbb{S}^T$ also contains more obscure semantic structures that may not correspond to realistic compilers, but that can detect certain type errors. Since $s^T.from\_byte$ is partial, it may be undefined for byte lists that are not of the form $s^T.to\_byte(v)$ for some value v. In this case, the semantics would get stuck when a program attempts to read an invalid representation with type T, and program verification would fail. A single $s^T \in \mathbb{S}^T$ whose *from_byte* function is undefined for the invalid representation suffices to render normal program termination unprovable.

As an example, consider the type `bool` of Booleans. A semantic structure for this type is easily obtained by imitating the encoding of values of a particular language implementation. For instance, $s^{\texttt{bool}}_{\text{gcc}}$ is a semantic structure for `bool` if we define $s^{\texttt{bool}}_{\text{gcc}}.to\_byte$ to map *false* to the byte value [`0x00`], and *true* to [`0x01`]. The GCC C compiler decodes Booleans by mapping [`0x00`] to *false*, and all other values to *true*. The corresponding $s^{\texttt{bool}}_{\text{gcc}}.from\_byte$ function is total, and as such incapable of detecting

[2] A function is underspecified if its precise mapping on values is not known. For an underspecified partial function the precise domain may also be unknown. Formally, one achieves this effect by using an arbitrarily chosen but fixed element of a suitable set of functions.

type errors: all byte values are valid representations for `bool`.

As a second example, consider the semantic structure $s_{0,1}^{\texttt{bool}}$ that agrees with $s_{\text{gcc}}^{\texttt{bool}}$, except that $s_{0,1}^{\texttt{bool}}.\mathit{from\_byte}$ is undefined for byte lists other than $[\texttt{0x00}]$ and $[\texttt{0x01}]$. This structure is able to detect as type error all modifications of Boolean variables b that store a value other than `0x00` or `0x01` at the address of b. When a program attempts to read b with type `bool`, $s_{0,1}^{\texttt{bool}}.\mathit{from\_byte}$ is undefined for the modified value, and the semantics will get stuck.

Now suppose that $\mathbb{S}^{\texttt{bool}} = \{s_{\text{gcc}}^{\texttt{bool}}, s_{0,1}^{\texttt{bool}}, s_{2,3}^{\texttt{bool}}, \dots\}$ additionally contains a structure $s_{2,3}^{\texttt{bool}}$ that performs the analogous mapping for the object representations $[\texttt{0x02}]$ (*true*) and $[\texttt{0x03}]$ (*false*). Because program verification is carried out against an arbitrary (but fixed) structure $s^{\texttt{bool}} \in \mathbb{S}^{\texttt{bool}}$, programs can be verified only if they are correct wrt. every structure in $\mathbb{S}^{\texttt{bool}}$. No fixed (constant) byte value is in the domain of all $s^{\texttt{bool}}.\mathit{from\_byte}$ functions, hence any modification that stores such a fixed value at the address of b will be detected as a type error. In contrast, reading a byte list of the form $s^{\texttt{bool}}.\mathit{to\_byte}(v)$ (for $v \in \{\mathit{true}, \mathit{false}\}$) at the address of b will never cause an error.

For practical purposes one chooses $\mathbb{S}^T$ as indicated by the thick black line in Fig. 1(b): It should contain the set of those semantic structures that are needed to ensure type safety as well as those structures that represent the encoding of the used compiler. The latter requirement ensures that the verification results apply directly to the generated code (assuming that the compiler is correct).

Additional assumptions typically constrain the set $\mathbb{S}^T$ of admissible semantic structures. They are often necessary to prove correctness of machine-dependent code. For instance, assuming that some pointer type T has the same size as type `int` makes the set $\mathbb{S}^T$ smaller, but may be required for the verification of a custom memory allocator that casts integers into type T internally. The questions of interest are therefore:

1. Which kinds of type errors can be detected with an underspecified data-type semantics, and
2. when do additional assumptions constrain $\mathbb{S}^T$ to a point where type safety is no longer guaranteed?

Giving partial answers to these questions is the central contribution of this paper. We now define semantic structures more formally.

2.1 Semantic Structures

In Sect. 4, we present three variants of semantic structures with increasing type-checking capabilities. To prepare for the two more advanced variants, the definition below contains "$\cdots$" as a placeholder for further parameters. For now, we assume no further parameters.

Definition 1 (Semantic structure)**.** Let T be a type. A semantic structure $s = (V, A, \mathit{size}, \mathit{to\_byte}, \mathit{from\_byte})$ for T consists of a non-empty set of values V, a set of addresses $A \subseteq \mathbb{N}$ (specifying alignment requirements for objects of type T), a positive integer *size* (specifying the size of object encodings), and two conversion functions:

$$\begin{aligned} \mathit{to\_byte}: &\quad V \times \cdots \to \mathit{list}[\mathit{byte}] \times \cdots \\ \mathit{from\_byte}: &\quad \mathit{list}[\mathit{byte}] \times \cdots \rightharpoonup V \end{aligned}$$

where $\mathit{list}[\mathit{byte}]$ denotes the type of byte lists, and *from_byte* is a partial function. Every semantic structure must satisfy the following properties for all $v \in V$:

$$\mathit{length}(\mathit{to\_byte}(v, \dots)) = \mathit{size} \tag{1}$$

$$\mathit{from\_byte}(\mathit{to\_byte}(v, \dots), \dots) = v \tag{2}$$

where $\mathit{length}\colon \mathit{list}[\mathit{byte}] \to \mathbb{N}$ denotes the length of a byte list.

Equation (2) requires $s.from\_byte$ to be defined on byte lists that form a valid object representation for T. Otherwise, $s.from\_byte$ may be undefined. To ensure type safety, we will exploit this fact by constructing sufficiently many semantic structures whose *from_byte* function is partial: at least one for every byte list that may have been modified by a type error. Note that eq. (2) ensures that one can always read a value that has been written with the same semantic structure. Therefore, the data-type semantics allows to verify well-typed code as expected.

The set of values that a type can hold may depend on the hardware architecture and compiler. For instance, the C++ type `unsigned int` can typically represent values from 0 to $2^n - 1$ on n-bit architectures. This set is therefore specified by each semantic structure, just like size, alignment, and encoding. For the verification of concrete programs, we generally assume a minimal set of values that can be represented by all semantic structures in $\mathbb{S}^T$.

We use bytes in the definition of semantics structures, because we assume a byte-wise organized memory model that resembles real hardware. This is sufficient to support the bit fields of C++, because the C++ standard specifies that the byte is the smallest unit of memory modifications [9, §1.7(3)–(5)]. By using lists of bits and bit-granular addresses instead, one could support more general architectures with more general bit fields.

3 Type Errors

Type errors are undesirable behaviors of a program that result from attempts to perform operations on values that are not of the appropriate data type. The causes for these errors are diverse. Buffer overflows, dangling or wild pointers, (de-)allocation failures, and errors in the virtual-to-physical address translation can all lead to type errors in low-level code.

Formally, we say that memory m is *modified* relative to memory m' at address a if it cannot be proven that m and m' are identical at a. A memory modification (at address a) is a state transition starting with memory m and yielding memory m' such that m' is modified relative to m (at address a). A read-access to an object in memory at address a with type T is *type correct*, if the content at a is provably the result of a write-access to a with type T. A program is *type correct* if all its memory-read accesses are type correct. It turns out that the strength of underspecified data-type semantics to detect an incorrectly typed read access depends on the kind of memory modification that happened before the read access in the relevant memory region. Therefore, we define *type error* as a memory modification that causes an incorrectly typed read access. To be able to view all missing variable initialization as type error, we assume a memory initialization that overwrites the complete memory with arbitrary values.

Our notion of type correctness has specific properties that are needed for the verification of systems code. Firstly, it permits to arbitrarily overwrite memory whose original contents will not be accessed any more. Secondly, it depends on the presence and strength of additional assumptions. If one assumes, for instance, that the range of $to\_byte^{\texttt{void*}}$ is contained in the domain of $from\_byte^{\texttt{unsigned}}$, then reading the value of a `void` pointer into an `unsigned` variable *is* type correct.

For the analysis in the following sections, we define the following classes of type errors (which are formally sets of memory modifications). These classes are only used with respect to a specific read access at some address a with some semantic structure s^T (for some type T). The memory modifications contained in some class may therefore depend on s^T and a. Note that the memory modifications in these classes may be caused not just by writing variables in memory, but also by hardware effects, such as changes of memory mapped registers or DMA access by external devices.

1. **Unspecified memory contents:** memory may contain arbitrary values; for instance, when the program reads a location that has not been initialized before. Formally, this class contains all memory modifications such that the modified value is indeterminate.
2. **Constant byte values:** memory locations may contain specific (constant) byte values; for instance, newly allocated memory may, in some cases, be initialized to `0x00` by the operating system or runtime environment. This class contains all memory modifications where the modified value is a constant.
3. **Object representation of a different type:** a T-typed read operation may find an object representation for a value of a different type U in memory. Typed reads of differently typed values result in implicit casts. Such an implicit cast happens, for instance, when the program attempts to read an inactive member of a `union` type, or when a pointer of type `T*` is dereferenced that actually points to an object of type `U`.

   For a given structure s^T (for some type T) and an address a, this class contains all memory modifications that write a complete object representation of some structure s^U (for a type $U \neq T$ with $s^U.size = s^T.size$) at a.
4. **Parts of valid object representation(s):** a special form of implicit cast occurs when the read operation accesses part of a valid object representation. For instance, `*(char*)p` reads the first character of the object pointed to by `p`. A single read of a larger object may also span several valid object representations (or parts thereof) simultaneously.

   Although this error class shows many similarities to Class 3, it illustrates the need for type-safety theorems capable of ruling out undesired modifications at the minimal access granularity of memory (typically one byte on modern hardware architectures).

   For a given structure s^T (for some type T) and an address a, this class contains all memory modifications that overwrite the memory range $[a, a + s^T.size)$ with (some slice of) consecutive object representations of structures s^{U_1}, s^{U_2}, ... (for arbitrary types U_i). For $i = 1$, this reduces to Class 3, and we require $U_1 \neq T$—otherwise there is no type error.
5. **Bitwise copy of valid object representations:** objects may perform operations on construction and destruction; for instance, they might register themselves in some global data structure. A bitwise copy of such an object does not preserve the semantics associated with the object. Consequently, any attempt to access the bitwise copy with the object's type may lead to functional incorrectness. We consider this a type error.

   For a given structure s^T (for some type T), this class contains all memory modifications that write at least one bit of an object representation of $s^{T'}$ (for an arbitrary type T').

We have presented these error classes in order of increasing detection difficulty. Class 5 is particularly challenging, because the invalid copy is bitwise indistinguishable from a valid object representation. The classes were developed during our investigation for a sound data-type semantics for non-trivially copyable types. We make no claim about their completeness. In the next section, we discuss how the different error classes may be detected with suitable variants of underspecified data-type semantics.

4 Type Sensitivity with Semantic Structures

In this section, we introduce the notion of *type sensitivity* to capture the requirement that no type errors occur as a result of memory modifications.

Definition 2 (Type Sensitivity)**.** A data-type semantics for a type T is *type sensitive with respect to a class $\mathscr{C}$ of memory modifications* if normal program termination implies that memory read with type T was not changed by modifications in $\mathscr{C}$.

Applied to our approach, type sensitivity means that for every memory modification in some class $\mathscr{C}$, and for every subsequent read of a modified object with type T, there must be a suitable semantic structure $s \in \mathbb{S}^T$ that can detect the modification as an error. More precisely, s is suitable if $s.from\_byte$ is undefined for the modified object representation.

In the remainder of this section, we introduce three different variants of underspecified data-type semantics: *plain object encodings*, *address-dependent object encodings*, and *external-state dependent object encodings*. These variants are type sensitive with respect to increasingly large classes of modifications.

4.1 Plain Object Encodings

Plain object encodings for semantic structures are inspired by trivially copyable C++ data types. An object of trivially copyable type T can be bitwise copied into a sufficiently large ($\geq s^T.size$) `char` array and back, and to any other address holding a T-typed object, without affecting its value [9, §3.9(2)]. Examples of plain encodings for integers include two's complement and sign magnitude, but also numeration systems augmented with, e.g., cyclic redundancy codes. The semantic structures for plain object encodings are as described in Def. 1, i.e., with no additional parameters.

Plain object encodings can detect reads from uninitialized memory (Class 1) and reads of constant data (Class 2), as exemplified in Sect. 2. Plain object encodings can also detect implicit casts of a differently typed object (Class 3) or parts of it (Class 4), provided $\mathbb{S}^T$ is sufficiently rich, i.e., type-sensitive with respect to the relevant class. We shall return to this condition in Sect. 5.

Plain object encodings cannot detect errors from Class 5.

4.2 Address-Dependent Object Encodings

To prevent errors of Class 5, copies by wrong means must be detected on non-trivially copyable data types [9, §3.9(2)]. Plain object encodings cannot detect these copies, because eq. (2) in Def. 1 requires $s.from\_byte$ to be defined for all byte lists that are equal to a valid object representation.

Address-dependent object encodings are able to recognize most (but not all) object copies obtained by bitwise memory copy operations. For address-dependent object encodings we augment the two conversion functions with an additional address parameter a, and adjust the left-inverse requirement of eq. (2) accordingly:

$$\forall a \in A.\ from\_byte(to\_byte(v,a),a) \ = \ v \tag{3}$$

Address-dependent encodings generalize plain object encodings by allowing a different encoding for each address. They can therefore detect all errors from Classes 1–4. Errors from Class 5 can be detected as long as the bitwise copy is located at an address that is different from the address of the original object. This includes type errors caused by aliasing between different virtual addresses. However, eq. (3) prevents address-dependent object encodings from detecting those errors of Class 5 that overwrite memory with a bitwise copy of an object previously stored at the same address.

4.3 External-State Dependent Object Encodings

External-state dependent object encodings are the most complex data-type semantics that we consider in this paper. They can detect type errors from all classes discussed in Sect. 3, but require further additions to the definition of semantic structures.

4.3.1 Exploiting Protected Bits.

In general, error detection is easy if a part of the object representation is protected and cannot be overwritten by erroneous operations. One only has to make sure that the set $\mathbb{S}^T$ contains semantic structures that store some kind of hash in the object representation. Then, when the unprotected part of the object representation is changed, the hash is wrong and *from_byte* will fail. External-state dependent object encodings develop this observation to the extreme. We will first see that it is sufficient to protect one bit only. After that, we will enrich the definition of semantic structures to make sure that there is always one protected bit.

Consider a type T and a set of semantic structures $\{s_v^a \mid a \in A, v \in V\}$ that all have the same set of values V and addresses A and that all use the same object encodings, except for the first bit. The first bit of $s_v^a.to\_byte(v', a')$ is 1 if $a = a'$ and $v = v'$ and 0 otherwise. The function $s_v^a.from\_byte$ fails if the first bit is different from what was specified for *to_byte*. That is, every s_v^a protects just the value v at address a by setting the first bit of the object representation and leaves all other value/address combinations unprotected.

Consider now a memory copy operation that copies the object representation of v from address a to a different address a' but leaves the first bit at address a' intact. If this bit is 0 then $from\_byte(a', \ldots)$ from structure $s_v^{a'}$ will fail. If the first bit at a' was 1, $s_v^{a'}$ will succeed, but all other structures will fail. In case a and a' are the same address, the memory remains (provably) unchanged, so there is no error to detect. However, if the value v at a' is overwritten with the object representation of a value v' that was previously stored there, either $s_{v'}^{a'}$ or $s_v^{a'}$ will detect the error in case the first bit at a' remains unchanged.

We can conclude that a sufficiently large set $\mathbb{S}^T$ can detect all type errors from all classes, provided there is at least one protected bit that no erroneous memory modification can change.

4.3.2 Protecting Bits in External State.

We will now enrich semantic structures such that every object representation can potentially contain one additional bit. With a clever use of underspecification this will require only one additional bit of memory per program. In a last step we will protect this one bit by making its location unknown.

We first enrich semantic structures with a partial function *protected_bit*:

$$\begin{aligned} protected\_bit: &\quad A \rightharpoonup B \\ to\_byte: &\quad V \times A \to list[byte] \times bit \\ from\_byte: &\quad list[byte] \times A \times bit \rightharpoonup V \end{aligned}$$

Here, B is the set of bit-granular addresses of the underlying memory and *bit* is the type of bits. The idea is as follows: If $s.protected\_bit(a) = b$ then the structure s uses an additional bit of object representation at address b for values stored at address a. In this case *to_byte* returns this additional bit and *from_byte* expects it as third argument. If $protected\_bit(a)$ is undefined, no additional bit is used and *to_byte* returns a dummy bit. The consistency requirement of semantic structures (eq. (2) in Def. 1) is changed in the

obvious way. In the verification environment (Fig. 1(a)) the memory model must of course be adapted to handle the additional bit appropriately.

There are of course problems if the address returned by *protected_bit* is already in use. We solve this in several steps. We first require that *protected_bit* is defined for at most one address for every structure s. This restriction does not hurt because $\mathbb{S}^T$ can still contain one structure for every address a such that *protected_bit*(a) is defined. Next, recall from Sect. 2 that for each primitive type T a fixed but arbitrarily chosen s^T is used. We refine this choice such that there is at most one primitive type T for which s^T.*protected_bit* is defined for one address. Again, this latter restriction does not limit the checking powers, because for each type every address still can potentially use an additional bit.

As a last step consider the set AF of free, unused bit granular addresses.[3] The memory model is enriched with a constant $r \in AF$ that is used precisely when the only additional bit that is used by the current choice of semantic structures is outside of AF. In this case, the memory model silently swaps the contents of r and the additional bit.

The changes for using protected bits are rather complex. However, if AF is not empty and if the sets $\mathbb{S}^T$ are sufficiently large, then there is for each type T and each address a a choice of semantic structures such that values of type T at address a use an additional bit in the object representation. If, for every accessible bit address b, every $\mathbb{S}^T$ contains a structure that uses b as additional bit, then the location of the additional bit is de facto unknown. Under these circumstances protected bits can detect all type errors from all classes of Sect. 3 as long as it is not the case that the complete memory is overwritten.

There are two points to note about external-state dependent object encodings. Firstly, the protected bit in these encodings is not write protected in a general sense. Type-correct operations that use the chosen semantic structure do in fact change the protected bit. Secondly, we used single bits and bit-granular addresses here only because we assume a memory model that resembles real hardware. The same idea can be applied to more abstract memory models.

5 Towards a Type-Sensitivity Theorem

In Sect. 4, we carved out type sensitivity as the key property that ensures there are sufficiently diverse semantic structures to identify all type errors. We now take a closer look at the delicate interplay between compiler intelligence, additional assumptions, and type sensitivity. We give sufficient conditions for type sensitivity for the error classes discussed in Sect. 3. These entail construction guidelines for sufficiently rich sets $\mathbb{S}^T$.

The relationship between semantic structures and type errors that are ruled out by verification turns out to be intricate. Intuitively, one might expect type sensitivity to be monotone: more semantic structures can detect more type errors. Unfortunately, more semantic structures also give rise to more program executions, and can therefore cause undetected type errors.

For instance, consider a memory-mapped device that overwrites memory at an address a. A program that performs a read access of type T will be unaffected by this modification if alignment requirements ensure that objects of type T are never located at a. Relaxing these alignment requirements, however, might lead to a type error in certain program executions: namely in those that read at a. To remain type sensitive, the data-type semantics would then need to admit a semantic structure that can detect the modification *and* allows alignment at a.

To be able to detect a memory modification as a type error (without resorting to external state), we have to assume some degree of independence between the modification and the semantic structures fixed

[3] For practical purposes one can use a safe approximation of AF.

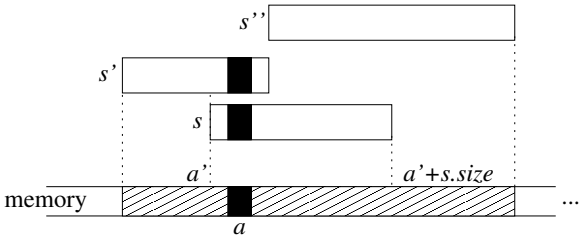


Figure 2: Visibility of addresses to semantic structures: a is visible to s and s' but not to s''.

for program verification. The classification in Sect. 3 describes different degrees of data independence. In this section, we additionally assume that modifications occur at fixed addresses, independent of the choice of semantic structure for T.

We say that an address a is *visible to a semantic structure* s if there exists $a' \in s.A$ such that $a' \leq a < a' + s.size$; see Figure 2 for illustration. In other words, a is visible to s if s might read memory at a. We say that a is *visible* if there is a semantic structure $s \in \mathbb{S}^T$ such that a is visible to s.

Lemma 3 (Unspecified Memory)**.** *Assume that for every visible address a, there is a semantic structure $s \in \mathbb{S}^T$ and an address $a' \in s.A$ (with $a' \leq a < a' + s.size$) such that for every sequence of bytes $(b_i)_{i=0}^{\infty}$, there is a byte value b such that $s.from\_byte$ is undefined for the byte list $[b'_{a'}, \ldots, b'_{a'+s.size-1}]$ given by $b'_a := b$, $b'_i := b_i$ for $i \neq a$. Then $\mathbb{S}^T$ is type sensitive wrt. unspecified memory contents (Class 1).*

Proof. Assume that an unspecified byte value at address a is read with type T. Because a is visible, there is a semantic structure $s \in \mathbb{S}^T$ as postulated in the lemma. This structure might read at address a'. Let $(b_i)_{i=0}^{\infty}$ be the memory contents at the time of the read. Since b_a is unspecified, it might be equal to b. Hence $s.from\_byte$ might read the byte list $[b'_{a'}, \ldots, b'_{a'+s.size-1}]$, for which it is undefined. Therefore, normal program termination is no longer provable. □

For instance, $\mathbb{S}^{\texttt{bool}} := \{s^{\texttt{bool}}_{0,1}\}$ (with $s^{\texttt{bool}}_{0,1}$ as defined on page 3) is type sensitive wrt. unspecified memory contents, because $s^{\texttt{bool}}_{0,1}.from\_byte$ is undefined for some (in fact, for all but two) byte lists of length one.

The following lemmas have similarly straightforward proofs, which we omit for space reasons. For constant byte values (Class 2), the only difference to Lemma 3 is that *any* byte value b must now be detected as an error. In particular, any semantics that is type sensitive wrt. constant byte values is also type sensitive wrt. unspecified memory contents.

Lemma 4 (Constant Bytes)**.** *Assume that for every visible address a, and for every byte value b, there is a semantic structure $s \in \mathbb{S}^T$ and an address $a' \in s.A$ (with $a' \leq a < a' + s.size$) such that for every sequence of bytes $(b_i)_{i=0}^{\infty}$, $s.from\_byte$ is undefined for the byte list $[b'_{a'}, \ldots, b'_{a'+s.size-1}]$ given by $b'_a := b$, $b'_i := b_i$ for $i \neq a$. Then $\mathbb{S}^T$ is type sensitive wrt. constant byte values (Class 2).*

Sect. 2 exemplifies how a sufficiently rich set $\mathbb{S}^T$ can be obtained by inclusion of sufficiently many semantic structures such that no byte list is in the domain of all *from_byte* functions.

For Class 3, we restrict ourselves to those semantic structures $s^T \in \mathbb{S}^T$ that read exactly one object representation produced by a semantic structure $s^U \in \mathbb{S}^U$ for some type U. Partial overlaps between object representations are covered by Class 4. We assume that the object representation for U does not depend on the choice of semantic structure for T.

Lemma 5 (Implicit Casts)**.** *Assume that for every semantic structure $s^T \in \mathbb{S}^T$, every address $a \in s^T.A$, and every byte list $[u_a, \ldots, u_{a+s^T.size-1}]$ that is the result of $s^U.to\_byte(v, \ldots)$ for some $s^U \in \mathbb{S}^U$, $v \in s^U.V$, there is a semantic structure $s \in \mathbb{S}^T$ and an address $a' \in s.A$ such that for every sequence of bytes $(b_i)_{i=0}^{\infty}$, $s.from\_byte$ is undefined for the byte list $[b'_{a'}, \ldots, b'_{a'+s.size-1}]$ given by $b'_i := u_i$ for $a \leq i < a + s^T.size$, $b'_i := b_i$ otherwise. Then $\mathbb{S}^T$ is type sensitive wrt. implicit casts from type U (Class 3).*

To construct a set $\mathbb{S}^T$ that fulfills the assumptions of the preceding lemma, one can include a set S of non-total semantic structures that are closed wrt. permutation of undefined object representations. Given a non-total semantic structure s where $s.from\_byte(bl, \ldots)$ is undefined, we can construct such a set S if we include for all byte lists bl' the semantic structure s' that is identical to s except that $s'.from\_byte = \Pi_{bl,bl'} \circ s.from\_byte$ and $s'.to\_byte = s.to\_byte \circ \Pi_{bl,bl'}$. Here, $\Pi_{bl,bl'}$ is the permutation function that just exchanges bl with bl'.

It is straightforward to generalize Lemma 5 to parts of valid object representations (Class 4) by allowing $[u_a, \ldots, u_{a+s^T.size-1}]$ to be (an arbitrary slice of) a concatenation of object representations for other types U_i. We omit the formal statement of this lemma.

To detect Class 5 errors, we have to further relax our independence requirements between type errors and semantic structures by considering also copies of object representations for T at visible addresses a. We say that two semantic structures s_1 and s_2 are *equivalent*, $s_1 \sim s_2$, if they differ at most in their *to_byte*, *from_byte* functions. Equivalent semantic structures produce and interpret object representations of the same size and at the same set of addresses.

Lemma 6 (Bitwise Copy)**.** *Assume that for every semantic structure $s \in \mathbb{S}^T$ and every $a \in s.A$ there exists an equivalent semantic structure $s' \in \mathbb{S}^T$ such that for any byte list $bl := [b_a, \ldots, b_{a+s.size-1}]$ where b_i, $i \in [a, a+s.size)$ may be comprised of copies of bit value of an object representation $s.to\_byte(v, \ldots)$ for some value $v \in s.V$, the result of $s'.from\_byte(bl, \ldots)$ is undefined if we replace the copied bits with the respective value of $s'.to\_byte(v, \ldots)$. Then $\mathbb{S}^T$ is type-sensitive wrt. bitwise copies of a non-trivially copyable object (Class 5).*

Clearly, if the copy is exact in the sense that $bl = s'.to\_byte(v, \ldots)$ for some value $v \in s.V$, eq. (2) rules out the existence of a semantic structure s' for which $s'.from\_byte(v, \ldots)$ is undefined. For the same reason, there can be no semantic structure with an address dependent encoding that detects Class 5 errors if bl is a valid object representation for the read address a.

In Sect. 4.3, we described external-state dependent encodings that are able to fulfill the assumptions of Lemma 6. The proof that external-state dependent encodings are type-sensitive wrt. all error classes is lengthy but not difficult. It builds on the fact that for every address a there exists a choice of semantic structures such that values at address a are protected with one additional bit of object representation, see Sect. 4.3.

6 Related Work

In spirit, the work presented here is very similar to runtime type checking, as it is present in dynamically typed programming languages such as Lisp and Perl. The runtime system of such languages attaches type tags to all values, and uses them for type checking at runtime. There are also tools that perform extended static or dynamic type checking for C and C++ programs by source or object code instrumentation [1, 11]. One can view each element of $\mathbb{S}^T$ as a runtime system that performs a particular type check. While runtime type checking can practically only be done for a limited number of program runs, this paper analyzes verification techniques that apply to all possible program runs.

There are several proposals to enhance the type safety of C. Cyclone [4] introduces additional data types such as safe pointers. BitC [14] augments a type-safe dialect of C with explicit placement and layout controls to reduce the number of situations where low-level code has to break the type system. A strength of underspecified data-type semantics is the ability to re-establish type safety when such a break is inevitable.

As mentioned in Sect. 1, several similar data-type semantics for C or C++ have been discussed in the literature. The formalization of C with abstract state machines by Gurevich and Huggins [5] and Norrish's C++ semantics in HOL4 [12] both rely on partial functions to convert byte lists to typed values.

The idea to reflect the underspecification of the C++ standard in the data-type semantics, and to exploit this underspecification for type checking, was first proposed in the context of the VFiasco project [7] by Hohmuth and Tews [6]. This idea has then been independently further developed in the operating-system verification projects l4.verified [10] and Robin [16].

For l4.verified, Tuch et al. built a typed memory on top of untyped memory [17]. This typed view on memory can be used to automatically discharge type-correctness conditions for type-safe code fragments.

7 Conclusions

In this paper, we explored the ability of underspecified data-type semantics to enforce type-correctness properties in verification settings that rely on untyped byte-wise organized memory. We have identified five different classes of type errors, and proved sufficient conditions for recognizing all type errors from each class. This required increasingly complex data-type semantics. Notably, simple underspecified data-type semantics are unsound for non-trivially copyable types. Bitwise copies of such types can only be detected with external-state dependent object encodings. The trade-off between using such complex data-type semantics or dealing with errors from class 5 by other means must be decided for each verification individually.

Although our analysis is inspired by C and C++, our results are largely programming-language independent. They apply to all programs that cannot be statically type-checked. To demonstrate the practical relevance of our analysis, we verified the type safety of a small code fragment from an operating-system kernel in PVS (see App. A). Our PVS files are publicly available (see footnote 1 on page 2).

Giving a fully accurate, sound data-type semantics for the verification of C and C++ code remains a challenge. The language standards make few guarantees in order to permit efficient implementations on a wide range of hardware architectures. Yet the type systems are complex, there are subtle constraints on memory representations and type domains, and the typed and untyped views on memory interact in intricate ways.

References

[1] Michael Burrows, Stephen N. Freund & Janet L. Wiener (2003): *Run-Time Type Checking for Binary Programs*. In Görel Hedin, editor: *12th International Conference on Compiler Construction, LNCS* 2622, Springer, pp. 90–105, doi:10.1007/3-540-36579-6_7.

[2] (2012): *The Go Programming Language Specification*. Available at `http://golang.org/doc/go_spec.html`. Retrieved June 15, 2012.

[3] James Gosling, Bill Joy, Guy L. Steele Jr. & Gilad Bracha (2005): *The Java Language Specification (3rd ed.)*. Addison-Wesley.

[4] D. Grossman, M. Hicks, T. Jim & G. Morrisett (2005): *Cyclone: A Type-Safe Dialect of C*. *C/C++ User's Journal* 23(1).

[5] Yuri Gurevich & James K. Huggins (1992): *The Semantics of the C Programming Language*. In Egon Börger, Gerhard Jäger, Hans Kleine Büning, Simone Martini & Michael M. Richter, editors: *Computer Science Logic, CSL '92*, *LNCS* 702, Springer, pp. 274–308, doi:10.1007/3-540-56992-8_17.

[6] M. Hohmuth & H. Tews (2003): *The Semantics of C++ Data Types: Towards Verifying low-level System Components*. In David Basin & Burkhart Wolff, editors: *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003. Emerging Trends Proceedings*, Universität Freiburg, pp. 127–144.

[7] M. Hohmuth & H. Tews (2005): *The VFiasco approach for a verified operating system*. In Andreas Gal & Christian W. Probst, editors: *Proc. 2nd ECOOP Workshop on Programming Languages and Operating Systems (ECOOP-PLOS 2005)*.

[8] ISO/IEC JTC1/SC22/WG14 C Standards Committee (2011): *Programming Languages—C*. ISO/IEC 9899:2011.

[9] ISO/IEC JTC1/SC22/WG21 C++ Standards Committee (2011): *Programming Languages—C++*. ISO/IEC 14882:2011.

[10] Gerwin Klein et al. (2010): *seL4: formal verification of an operating-system kernel*. *Commun. ACM* 53(6), pp. 107–115, doi:10.1145/1743546.1743574.

[11] Alexey Loginov, Suan Hsi Yong, Susan Horwitz & Thomas W. Reps (2001): *Debugging via Run-Time Type Checking*. In Heinrich Hußmann, editor: *Fundamental Approaches to Software Engineering, FASE 2001*, *LNCS* 2029, Springer, pp. 217–232, doi:10.1007/3-540-45314-8_16.

[12] Michael Norrish (2008): *A Formal Semantics for C++*. Technical Report, NICTA. Available from `http://nicta.com.au/people/norrishm/attachments/bibliographies_and_papers/C-TR.pdf`. Retrieved June 15, 2012.

[13] Sam Owre & Natarajan Shankar (2008): *A Brief Overview of PVS*. In Otmane Aït Mohamed, César Muñoz & Sofiène Tahar, editors: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008*, *LNCS* 5170, Springer, pp. 22–27, doi:10.1007/978-3-540-71067-7_5.

[14] Jonathan Shapiro (2006): *Programming language challenges in systems codes: why systems programmers still use C, and what to do about it*. In Christian W. Probst, editor: *Proc. 3rd Workshop on Programming Languages and Operating Systems (PLOS 2006)*, ACM, p. 9.

[15] Hendrik Tews, Marcus Völp & Tjark Weber (2009): *Formal Memory Models for the Verification of Low-Level Operating-System Code*. *Journal of Automated Reasoning: Special Issue on Operating Systems Verification* 42(2–4), pp. 189–227.

[16] Hendrik Tews, Tjark Weber, Marcus Völp, Erik Poll, Marko van Eekelen & Peter van Rossum (2008): *Nova Micro–Hypervisor Verification*. Technical Report ICIS–R08012, Radboud University Nijmegen.

[17] Harvey Tuch, Gerwin Klein & Michael Norrish (2007): *Types, Bytes, and Separation Logic*. In Martin Hofmann & Matthias Felleisen, editors: *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007*, ACM, pp. 97–108, doi:10.1145/1190216.1190234.

A Verifying Safe and not so Safe Kernel Code

To demonstrate our approach, we have verified termination and hence type safety of a small piece of microkernel code (Fig. 3). The code in lines 21 to 24 is part of the scheduler. Upon preemption, it inserts the thread control block (TCB) of the currently running thread at the back of the doubly-linked priority list. Also shown but not verified is a simplified version of the copy routine of the inter-process communication path. To demonstrate the error checking capabilities, we modified the call to `memcpy` in line 16 to copy the first `cnt` bytes from the sender TCB rather than from its message buffer `mr`.

```
1 class TCB : public list<TCB> {
2 public:
3   unsigned char priority;
4   Msg_Buffer mr;
5   ...
6   static inline TCB * current(){
7     unsigned long dummy;
8     asm volatile ("mov␣%%esp,␣%0␣\n\t" : "+m" (dummy) ::);
9     return reinterpret_cast<TCB*> (dummy & ~(1 << L2_TCB_SIZE));
10  }
11 };
12 list<TCB> prio_list[Max_Prio];
13 void
14 copy(TCB * dest, unsigned long cnt){
15   TCB * src = TCB::current();
16   memcpy(src, dest, cnt);
17 }
18
19 ...
20 // preempt current thread
21 TCB * current = TCB::current();
22 unsigned char p = current→priority;
23
24 prio_list[p].push_back(current);
25 ...
```

Figure 3: Excerpt of a simple IPC send operation and the kernel code that is executed when the current thread is preempted.

The verification is based on an excerpt of the Robin statement and expression semantics for C++ [16] extended with the C++ instance of our data-type semantics. We will first focus on lines 21 to 24, as they demonstrate the normal use of our data-type semantics. After that, we dive into the function `TCB::current()`, which extracts the TCB pointer of the current thread from the processor's kernel stack pointer, and look at the interplay between `list<TCB>::push_back` and the erroneous call to `memcpy`.

A.1 Preempt Current Thread

For our example, we use sets $\mathbb{S}^T$ that are rich enough to fulfill the respective preconditions of Lemmas 3 to 6, for all used types T. We assume that the compiler inlines the call to `push_back` in line 24, which therefore expands to the usual update of the `prev` and `next` pointers of double-linked list inserts. By inheriting from `list<TCB>`, `class TCB`-typed objects include these pointers in their representation. In the course of updating these pointers, the value of `current` must be read to obtain the addresses of these members. This value can only be obtained by reading the byte list at the address of `current` and interpreting it using $s^{TCB*}.\mathit{from\_byte}$. The assignment to `p` in line 22 or hardware side effects (e.g., when reading `current→priority`) may modify this byte list in which case our data-type semantics prevents any verification. We therefore make the (sensible) assumption that the objects at `current` and `p` are allocated at disjoint address regions, which are not changed by any side effect.[4] Then our rewrite engine

[4] These two assumptions are only made to simplify the case study. In a real verification, the disjointness would be implied by the functional correctness of the memory allocator. A suitable type-safety invariant would imply that side effects occur only in other address regions.

simplifies the typed read of the `current` pointer to

$$s^{TCB*}.\mathit{from\_byte}(s^{TCB*}.\mathit{to\_byte}(v, \texttt{current}), \texttt{current})$$

which eq. (2) collapses to v where v is the result of `TCB::current()`.

When compared to other approaches, the qualitative difference is that our approach demands either a proof of disjointness, or additional assumptions that connect the object representations of `unsigned char` and `TCB*`. For the same reason, Lemma 6 demands for a fix of the call to `memcpy` because only then the list invariant can be maintained that running threads are never in the priority list. An erroneous `memcpy` of the characters of not trivially-copyable type `list<TCB>` prevents the proof of such an invariant.

A.2 TCB::current()

The verification of `TCB::current()` (lines 6 to 9) demonstrates the inclusion of additional assumptions without restricting $\mathbb{S}^T$ to a point where type sensitivity is no longer given.

Co-locating the kernel stack next to sufficiently aligned objects is a common programming pattern in microkernels to quickly retrieve pointers to these objects. `TCB::current()` reads the stack pointer value in `esp` as an `unsigned long` (line 8), rounds it to the object alignment (line 9) and casts it into the respective pointer type (line 9). When verifying the first operation, it is tempting to fix the encoding of `esp` as the four-byte little endian representation of machine words and to require that the word values of stack addresses are valid object representations for `unsigned long`. Under these assumptions, underspecified data-type semantics can detect modifications that cause the `esp` value to point to non-stack addresses. However, modifications that cause the `esp` to point to other (possibly unallocated) stacks remain undetected. An elegant way to circumvent these problems is to introduce a semantic structure also for the `esp` register. The verification is then performed against a whole family of processors that differ in their choice of semantics structure for register `esp`.