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

Hendrik Tews[1], Marcus Völp[1], Tjark Weber[2]

[1]Technische Universität Dresden, Germany

[2]Uppsala University, Sweden

Systems Software Verification Conference, November 29, 2012

Motivation

Find a common denominator in

- Gurevich and Huggins ASM semantics of C
- Norrish's C++ semantics in HOL4
- C semantics in l4.verified
- C++ semantics in VFiasco/Robin

Motivation

Find a common denominator in

- Gurevich and Huggins ASM semantics of C
- Norrish's C++ semantics in HOL4
- C semantics in l4.verified
- C++ semantics in VFiasco/Robin

They all encode typed values in an untyped, byte-wise organised memory

$$to\_byte : V \to byte\ list$$
$$from\_byte : byte\ list \rightharpoonup V$$

- V are the values of some type
- *from_byte* might fail on byte lists that do note represent a value from V
- the object encoding and the domain of *from_byte* is usually not specified

Underspecified data-type semantics refers to this kind of semantics

Motivation

Find a common denominator in

- Gurevich and Huggins ASM semantics of C
- Norrish's C++ semantics in HOL4
- C semantics in l4.verified
- C++ semantics in VFiasco/Robin

They all encode typed values in an untyped, byte-wise organised memory

$$to\_byte : V \to byte\ list$$

$$from\_byte : byte\ list \rightharpoonup V$$

- V are the values of some type
- *from_byte* might fail on byte lists that do note represent a value from V
- the object encoding and the domain of *from_byte* is usually not specified

Underspecified data-type semantics refers to this kind of semantics

Summary of the talk / paper

Underspecified data-type semantics can detect type errors

- *from_byte* fails on objects of the wrong type

Main questions

- Which type errors can be detected?
- Under which preconditions?

This paper makes progress on the topic, providing partial answers

- describe external state-dependent encodings
  for detecting most subtle type errors
- trade-off between
  - complexity of the object encodings
  - and the different kinds of type errors
- sufficient conditions on the encoding functions
  for detecting certain type errors

Summary of the talk / paper

Underspecified data-type semantics can detect type errors

- *from_byte* fails on objects of the wrong type

Main questions

- Which type errors can be detected?
- Under which preconditions?

This paper makes progress on the topic, providing partial answers

- describe external state-dependent encodings
  for detecting most subtle type errors
- trade-off between
  - complexity of the object encodings
  - and the different kinds of type errors
- sufficient conditions on the encoding functions
  for detecting certain type errors

Outline

- Introduction
- Background / Basics
- Type Errors
- Stronger Object Encodings
- Type Sensitivity
- Conclusion

Underspecification

A function f is underspecified if

- its precise mapping on values is not known
- for partial f: its domain is not known

Technically,

- let F be a suitable set of candidate functions
- choose $f \in F$ arbitrarily but fixed
- $\vdash P(f)$ only if $\vdash \forall f \in F \,.\, P(f)$

How to detect type errors with underspecified data-type semantics

Consider `bool`

s₁: $\texttt{false} \longleftrightarrow \texttt{0x00} \quad \texttt{true} \longleftrightarrow \texttt{0x01}$

$\mathrm{dom}(\mathit{from\_byte}_1) = \{\texttt{0x00}, \texttt{0x01}\}$

s₂: $\texttt{false} \longleftrightarrow \texttt{0x02} \quad \texttt{true} \longleftrightarrow \texttt{0x03}$

$\mathrm{dom}(\mathit{from\_byte}_2) = \{\texttt{0x02}, \texttt{0x03}\}$

- $\mathbb{S} = \{s_1, s_2\}$
- *from_byte* can read whatever *to_byte* wrote, because the choice $s \in \mathbb{S}$ is fixed

```
boolean b = true; *(p + x) = y
```

- if $\texttt{y}$ writes something $> \texttt{0x02}$, $\mathit{from\_byte}_1$ will fail
- otherwise $\mathit{from\_byte}_2$ will fail
- proof assistant *cannot* prove normal program termination

$\mathbb{S}$ detects type errors

Type checking capabilities can easily get lost

Consider `unsigned` **and** `void *`**. Assume**

- `unsigned` can represent everything from 0 to $2^{32} - 1$
- you can cast between `unsigned` and `void *` without loosing bits
- `void *` fits in 4 bytes

$\texttt{from\_byte}^{\texttt{void*}}$ **must be total on lists of length 4**

- because of cardinality reasons
- every 4 bytes form a valid object representation
- no type checking

What is all this good for?

type checkers can automatically detect all type errors

... while underspecified data-type semantics can detect *some* type errors only during *verification*

... but not for low-level code, which

- contains its own memory allocation
- must break the type system for specific hardware registers
- manages the virtual address mapping of itself

For low level code

- type correctness depends on functional correctness
- simple type correctness properties are undecidable
- there exists no static type checker

Verification of low-level code necessarily includes some type checking

What is all this good for?

type checkers can automatically detect all type errors

. . . while underspecified data-type semantics can detect *some* type errors only during *verification*

. . . but not for low-level code, which

- contains its own memory allocation
- must break the type system for specific hardware registers
- manages the virtual address mapping of itself

For low level code

- type correctness depends on functional correctness
- simple type correctness properties are undecidable
- there exists no static type checker

Verification of low-level code necessarily includes some type checking

What is all this good for?

type checkers can automatically detect all type errors

... while underspecified data-type semantics can detect *some* type errors only during *verification*

... but not for low-level code, which

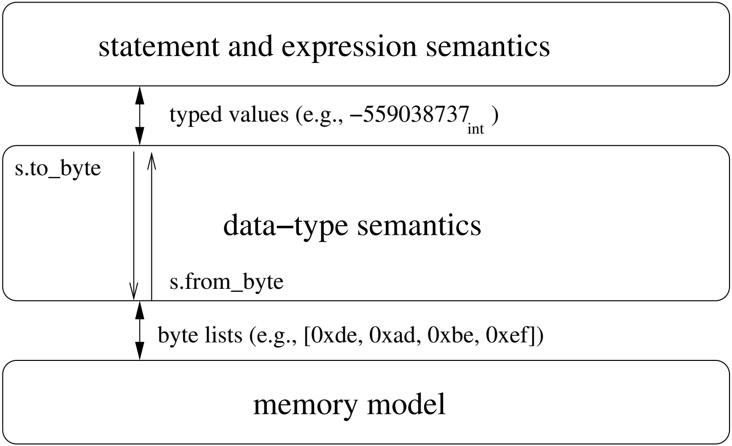
- contains its own memory allocation
- must break the type system for specific hardware registers
- manages the virtual address mapping of itself

For low level code

- type correctness depends on functional correctness
- simple type correctness properties are undecidable
- there exists no static type checker

Verification of low-level code necessarily includes some type checking

Background for this talk



statement and expression semantics

typed values (e.g., -559038737_{int})

s.to_byte

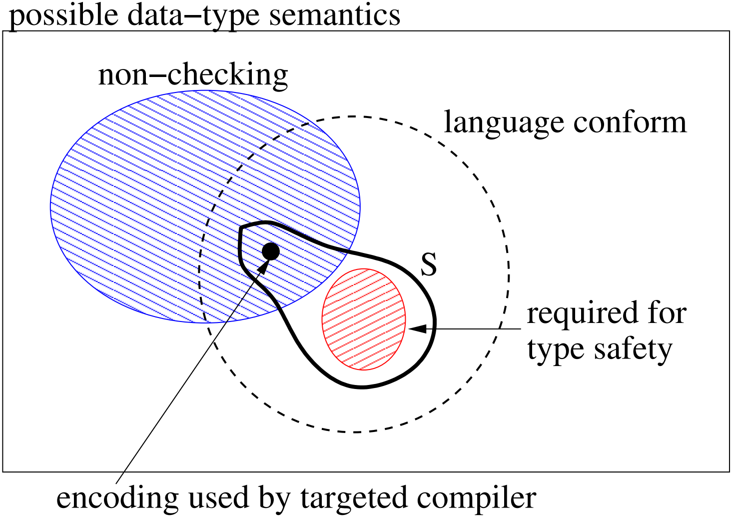
data–type semantics

s.from_byte

byte lists (e.g., [0xde, 0xad, 0xbe, 0xef])

memory model

General approach



Semantic Structures

Definition (Semantic structure)

A semantic structure for a type T is a tuple $(V, A, size, to\_byte, from\_byte)$ with

V	set of values
A	set of addresses $A \subseteq \mathbb{N}$
$size$	size of object encodings (in bytes)
to_byte	$V \times \cdots \rightarrow byte\ list \times \cdots$
$from\_byte$	$byte\ list \times \cdots \rightharpoonup V$

such that

$$\begin{aligned} length(to\_byte(v, \ldots)) &= size \\ from\_byte(to\_byte(v, \ldots), \ldots) &= v \end{aligned}$$

Outline

Introduction

Background / Basics

Type Errors

Stronger Onject Encodings

Type Sensitivity

Conclusion

Type-Error Classification I

1. **Unspecified memory contents**
   - arbitrary, uninitialised values
2. **Constant values**
3. **Object of different type**
   - a read of type T finds a (complete) value of type U
   - implicit cast
     - read inactive member of a union
     - read after wrong pointer arithmetic
4. **Parts of valid objects**
   - a read of type T finds some bytes of an object of type U
   - copy one byte from an U-object into a T-object

Non-trivially copyable Data in C++

Trivially copyable data

- can be copied with `memcpy`
- afterwards the destination holds the same value as the source

Non-trivially copyable data

- might have a constructor/destructor that ensures some global invariant
- a virtual function table that cannot be copied with `memcpy`
- such types cannot be copied with `memcpy`

Type-Error Classification II

5. Bitwise object copies

- copy at least one bit of a valid object
- restore a backup copy of some object at the same address

Outline

Introduction

Background / Basics

Type Errors

Stronger Onject Encodings

Type Sensitivity

Conclusion

Address dependent encodings

Enhance semantic structures with addresses

V	set of values
A	set of addresses $A \subseteq \mathbb{N}$
size	size of object encodings (in bytes)
to_byte	$V \times A \rightarrow \mathit{byte\ list}$
from_byte	$\mathit{byte\ list} \times A \rightharpoonup V$

such that

$$\begin{aligned} \mathit{length}(\mathit{to\_byte}(v, a)) &= \mathit{size} \\ \mathit{from\_byte}(\mathit{to\_byte}(v, a), a) &= v \end{aligned}$$

Can detect bitwise object copies (class 5)

- if source and destination have a different address

Address dependent encodings

Enhance semantic structures with addresses

V	set of values
A	set of addresses $A \subseteq \mathbb{N}$
size	size of object encodings (in bytes)
to_byte	$V \times A \rightarrow \mathit{byte\ list}$
from_byte	$\mathit{byte\ list} \times A \rightharpoonup V$

such that

$$\begin{aligned} \mathit{length}(\mathit{to\_byte}(v, a)) &= \mathit{size} \\ \mathit{from\_byte}(\mathit{to\_byte}(v, a), a) &= v \end{aligned}$$

Can detect bitwise object copies (class 5)

- if source and destination have a different address

External-state dependent encodings

Outline of the next slides

- error detection is easy, if some part of the object remains unchanged
    - unchanged part could contain hash
- 1 unchanged bit suffices
- enrich semantic structures to ensure that there is always 1 additional bit
- 1 free bit suffices to protect everything

External-state dependent encodings

Outline of the next slides

- error detection is easy, if some part of the object remains unchanged
    - unchanged part could contain hash
- 1 unchanged bit suffices
- enrich semantic structures to ensure that there is always 1 additional bit
- 1 free bit suffices to protect everything

1 bit per object is enough

Consider $\{s_v^a \mid a \in A, v \in V\}$ such that

- they use the same object encoding, except for the first bit
- for the first bit: $s_v^a.to\_byte(v', a') = 1$ iff $a = a'$ and $v = v'$
- $s_v^a.from\_byte$ fails if the first bit is different

Assume that an object at address a is changed

- the first bit remains intact
- the remaining bits encode v
- $s_v^a.from\_byte$ will fail if the first bit is 0
- $s_{v'}^{a'}.from\_byte$ will fail if the first bit is 1
- regardless where the bits for v come from

Object encodings with external state

Enhance semantic structures with protected bits

V	set of values
A	set of addresses $A \subseteq \mathbb{N}$
size	size of object encodings (in bytes)
protected_bit	$A \rightharpoonup BA$
to_byte	$V \times A \rightarrow \mathit{byte\ list} \times \mathit{bit}$
from_byte	$\mathit{byte\ list} \times A \times \mathit{bit} \rightharpoonup V$

- if *protected_bit* is defined,
  one bit of the object representation is to be stored there
- memory model must be suitably adapted
- problems if protected bit is already in use (wait for next slide)
- the result of *protected_bit* is completely unspecified
- need to overwrite the complete memory to overwrite the protected bit

Ensure the protected bit is unused

Restrict the choice of semantic structures

- $s.protected\_bit$ is defined for at most one address
- have to choose one s^T for each primitive type T
- choose such that there is one protected bit for at most one primitive type T
- have to deal with at most one protected bit at any time
- adapt memory model to silently exchange the protected bit with a free bit

One free bit suffices to protect all objects of all types

- for every primitive type T, every address a and every bit address ba, there is a choice of semantic structures for the primitive types, such that

$$s^T.protected\_bit(a) = ba$$

Outline

Introduction

Background / Basics

Type Errors

Stronger Oject Encodings

Type Sensitivity

Conclusion

Type sensitivity

Definition (Type Sensitivity)

The set $\mathbb{S}^T$ of semantic structures for T is

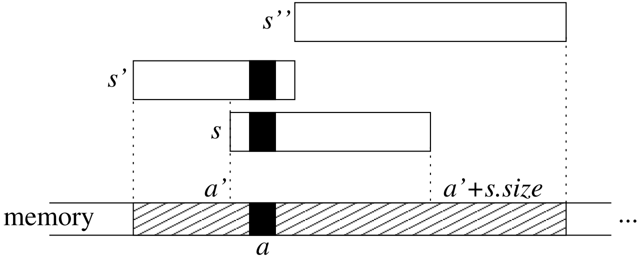
type sensitive with respect to a class $\mathcal{C}$ of type errors

if normal termination implies that no T-object was affected by errors in $\mathcal{C}$.

Type sensitivity permits to distinguish between

- sufficient conditions on the semantics $\mathbb{S}^T$, and
- the construction of $\mathbb{S}^T$
- additional assumptions necessary for the verification

Visible addresses



Address a is visible in s and s' but not in s''

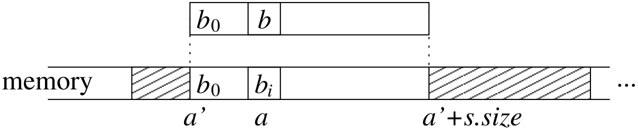
Type sensitivity for unspecified memory

Lemma

Assume that

- *for every visible address* a
- *there is a semantic structure* $s \in \mathbb{S}^T$ *and an address* $a' \in s.A$ *such that*
- $a' \leq a < a' + s.size$ *and*
- *for every* $[b_0, \ldots, b_{size-1}]$
- *there is a* b*, such that*
- $s.from\_byte([b_0, \ldots, b, \ldots, b_{size-1}]) = undef$

Then $\mathbb{S}^T$ *is type sensitive wrt. unspecified memory contents (Class 1).*



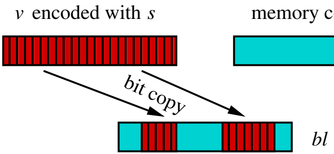
Type sensitivity for bitwise copy

Lemma

Assume that

- *for every structure $s \in \mathbb{S}^T$, $v \in s.V$ and every visible address a*
- *there exists a semantic structure $s' \in \mathbb{S}^T$ such that*
- *s and s' differ only in to_byte and from_byte and*
- *for every byte list bl, comprising $s.to\_byte(v, \ldots)$,*
- *$s'.from\_byte(bl') = undef, \quad$ where bl' equals bl but with $s'.to\_byte(v, \ldots)$ substituted for $s.to\_byte(v, \ldots)$.*

Then $\mathbb{S}^T$ is type sensitive wrt. bitwise object copies (Class 5).



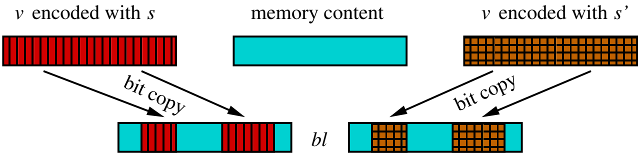
Type sensitivity for bitwise copy

Lemma

Assume that

- *for every structure $s \in \mathbb{S}^T$, $v \in s.V$ and every visible address a*
- *there exists a semantic structure $s' \in \mathbb{S}^T$ such that*
- *s and s' differ only in to_byte and from_byte and*
- *for every byte list bl, comprising $s.to\_byte(v, \ldots)$,*
- *$s'.from\_byte(bl') = undef$, where bl' equals bl but with $s'.to\_byte(v, \ldots)$ substituted for $s.to\_byte(v, \ldots)$.*

Then $\mathbb{S}^T$ is type sensitive wrt. bitwise object copies (Class 5).



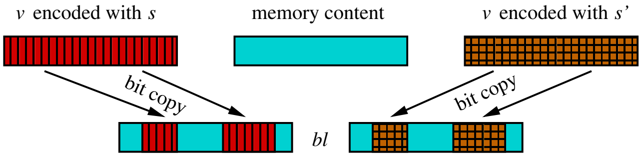
Type sensitivity for bitwise copy

Lemma

Assume that

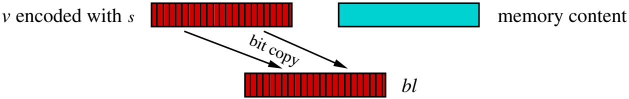
- *for every structure* $s \in \mathbb{S}^T$, $v \in s.V$ *and every visible address* a
- *there exists a semantic structure* $s' \in \mathbb{S}^T$ *such that*
- s *and* s' *differ only in to_byte and from_byte and*
- *for every byte list* bl*, comprising* $s.to\_byte(v, \ldots)$,
- $s'.from\_byte(bl') = undef$, *where* bl' *equals* bl *but with* $s'.to\_byte(v, \ldots)$ *substituted for* $s.to\_byte(v, \ldots)$.

Then $\mathbb{S}^T$ *is type sensitive wrt. bitwise object copies (Class 5).*



Type sensitivity for bitwise copy II

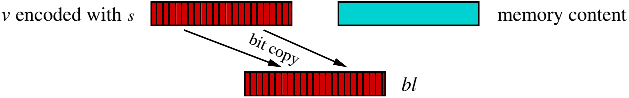
Assumptions are impossible for the case



because $s'.from\_byte(s'.to\_byte(v, \ldots), \ldots)$ must be equal to v

Type sensitivity for bitwise copy II

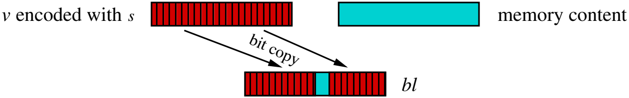
Assumptions are impossible for the case



because $s'.from\_byte(s'.to\_byte(v, \ldots), \ldots)$ must be equal to v

With external state dependent encodings

there is always one original bit left



unless the whole memory is overwritten.

Outline

Introduction

Background / Basics

Type Errors

Stronger Oject Encodings

Type Sensitivity

Conclusion

Conclusion

Underspecified data-type semantics

- can detect type errors
- verification of low-level code necessarily contains some type checking
- inspired by C/C++, applicable to other languages as well

Introduce

- external-state dependent object encodings
- type sensitivity

Trade-off between

- more difficult classes of type errors
- the complexity of the semantics for detecting these errors

Disclaimer

Notion of type error depends on

- the language
- the verification goals

External-state dependent encodings

- might not be well-suited for verification