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

Hendrik Tews\textsuperscript{1}, Marcus Völп\textsuperscript{1}, Tjark Weber\textsuperscript{2}

\textsuperscript{1}Technische Universität Dresden, Germany
\textsuperscript{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

\[
\begin{align*}
to\_byte &: V \rightarrow \text{byte list} \\
from\_byte &: \text{byte list} \rightarrow V
\end{align*}
\]

- \( V \) are the values of some type
- \( from\_byte \) might fail on byte lists that do not 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

\[
\text{to}_\text{byte} : V \to \text{byte list} \\
\text{from}_\text{byte} : \text{byte list} \to V
\]

- \( V \) are the values of some type
- \( \text{from}_\text{byte} \) might fail on byte lists that do not represent a value from \( V \)
- the object encoding and the domain of \( \text{from}_\text{byte} \) 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

- \textit{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

- \textit{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`

\[ \begin{align*}
\text{s}_1 &: \text{false} \leftrightarrow 0x00 \quad \text{true} \leftrightarrow 0x01 \\
\text{dom}(\text{from\_byte}_1) &= \{0x00, 0x01\} \\
\text{s}_2 &: \text{false} \leftrightarrow 0x02 \quad \text{true} \leftrightarrow 0x03 \\
\text{dom}(\text{from\_byte}_2) &= \{0x02, 0x03\} \\
\Rightarrow \quad &\mathbb{S} = \{\text{s}_1, \text{s}_2\} \\
\Rightarrow \quad \text{from\_byte} \text{ can read whatever to\_byte wrote, because the choice } s \in \mathbb{S} \text{ is fixed}
\end{align*} \]

boolean \( b = \text{true}; *(p + x) = y \)

- if \( y \) writes something \( > 0x02 \), `from\_byte_1` will fail
- otherwise `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

from_byte\(^{\text{void}*}\) 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., \(-559038737\) )

s.to_byte

data–type semantics

s.from_byte

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

memory model
possible data–type semantics

non–checking

language conform

required for type safety

encoding used by targeted compiler
**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)
- $to\_byte : V \times \cdots \rightarrow byte\ list \times \cdots$
- $from\_byte : byte\ list \times \cdots \rightarrow V$

such that

\[
\text{length}(to\_byte(v, \ldots)) = size
\]

\[
from\_byte(to\_byte(v, \ldots), \ldots) = v
\]
Outline

Introduction

Background / Basics

Type Errors

Stronger Object 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

```
all live objects  ───────→  ───────→  ───────→  ... 
```

Tews, Völp, Weber
Underspecified Data-Type Semantics
SSV 2012
Outline

Introduction

Background / Basics

Type Errors

Stronger Object Encodings

Type Sensitivity

Conclusion
Address dependent encodings

Enhance semantic structures with addresses

\[ V \text{ set of values} \]
\[ A \text{ set of addresses } A \subseteq \mathbb{N} \]
\[ \text{size } \text{ size of object encodings (in bytes)} \]
\[ \text{to}_\text{byte} \ V \times A \rightarrow \text{byte list} \]
\[ \text{from}_\text{byte} \ \text{byte list} \times A \rightarrow V \]

such that

\[ \text{length}(\text{to}_\text{byte}(v, a)) = \text{size} \]
\[ \text{from}_\text{byte}(\text{to}_\text{byte}(v, a), a) = v \]

Can detect bitwise object copies (class 5)

- if source and destination have a different address
Address dependent encodings

Enhance semantic structures with addresses

\[ V \text{ set of values} \]
\[ A \text{ set of addresses } A \subseteq \mathbb{N} \]
\[ \text{size} \text{ size of object encodings (in bytes)} \]
\[ \text{to}_{\text{byte}} \text{ } V \times A \to \text{ byte list} \]
\[ \text{from}_{\text{byte}} \text{ } \text{byte list} \times A \to V \]

such that

\[ \text{length}(\text{to}_{\text{byte}}(v, a)) = \text{size} \]
\[ \text{from}_{\text{byte}}(\text{to}_{\text{byte}}(v, a), a) = v \]

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
Consider \( \{s^a_v \mid a \in A, v \in V\} \) such that

- they use the same object encoding, except for the first bit
- for the first bit: \( s^a_v.to\_byte(v', a') = 1 \) iff \( a = a' \) and \( v = v' \)
- \( s^a_v.from\_byte \) 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 \( v \)
- \( s^a_v.from\_byte \) will fail if the first bit is 0
- \( s^{a'}_v.from\_byte \) 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)

\textit{protected\_bit} \( A \rightarrow BA \)

\textit{to\_byte} \( V \times A \rightarrow \text{byte list} \times \text{bit} \)

\textit{from\_byte} \( \text{byte list} \times A \times \text{bit} \rightarrow V \)

- if \textit{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 \textit{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 Object Encodings

Type Sensitivity

Conclusion
Type sensitivity

**Definition (Type Sensitivity)**

The set $S^T$ of semantic structures for $T$ is 

\[ \text{type sensitive with respect to a class } C \text{ of type errors} \]

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

**Type sensitivity permits to distinguish between**

- sufficient conditions on the semantics $S^T$, and
- the construction of $S^T$
- additional assumptions necessary for the verification
Visible addresses

Address \( a \) is visible in \( s \) and \( s' \) but not in \( s'' \)
Lemma
Assume that

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

Then $S^T$ is type sensitive wrt. unspecified memory contents (Class 1).
Type sensitivity for bitwise copy

Lemma
Assume that

1. for every structure \( s \in S^T \), \( v \in s.V \) and every visible address \( a \)
2. there exists a semantic structure \( s' \in S^T \) such that
3. \( s \) and \( s' \) differ only in to\_byte and from\_byte and
4. for every byte list \( bl \), comprising \( s\).to\_byte(\( v, \ldots \))
5. \( s'.from\_byte(bl') = \text{undef} \), where \( bl' \) equals \( bl \) but with
   \( s'.to\_byte(v, \ldots) \) substituted for \( s\).to\_byte(v, \ldots).

Then \( S^T \) is type sensitive wrt. bitwise object copies (Class 5).
**Type sensitivity for bitwise copy**

**Lemma**

Assume that

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

Then $S^T$ 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 $\text{to	extunderscore byte}$ and $\text{from	extunderscore byte}$ and
- for every byte list $bl$, comprising $s.\text{to	extunderscore byte}(v, \ldots)$,
- $s'.\text{from	extunderscore byte}(bl') = \text{undef}$, where $bl'$ equals $bl$ but with $s'.\text{to	extunderscore byte}(v, \ldots)$ substituted for $s.\text{to	extunderscore byte}(v, \ldots)$.

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

\begin{figure}
\centering
\includegraphics[width=\textwidth]{bitwise-copy-diagram}
\caption{Diagram illustrating the bit copy process.}
\end{figure}
Type sensitivity for bitwise copy II

Assumptions are impossible for the case

\[ v \text{ encoded with } s \quad \text{memory content} \]

because \( s'.\text{from}\_\text{byte}(s'.\text{to}\_\text{byte}(v, \ldots), \ldots) \) must be equal to \( v \)
Type sensitivity for bitwise copy II

Assumptions are impossible for the case

\[ v \text{ encoded with } s \quad \text{memory content} \]

\[ \text{bit copy} \]

\[ b \]

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

\[ v \text{ encoded with } s \quad \text{memory content} \]

\[ \text{bit copy} \]

\[ b \]

unless the whole memory is overwritten.
Outline

Introduction

Background / Basics

Type Errors

Stronger Object 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