The Semantics of C++ Data Types: Towards Verifying low-level System Components

Michael Hohmuth Hendrik Tews

Dresden University of Technology
Department of Computer Science

vfiasco@os.inf.tu-dresden.de

July 3, 2003

In order to formally reason about low-level system programs one needs a semantics (for the programming language in question) that can deal with programs that are *not statically type-correct.* For system-level programs, the semantics must deal with such heretical constructs like casting integers to pointers and converting pointers between incompatible base types.

In this paper we describe a formal semantics for the data types of the C++ programming language that is suitable for low-level programs in the above sense. This work is part of a semantics for a large subset of the C++ programming language developed in the VFiasco project. In the VFiasco project we aim at the verification of substantial properties of the Fiasco microkernel, which is written in C++.

1 Introduction

The VFiasco [21] project aims at the verification of substantial properties of the Fiasco [9] microkernel for x86 PC hardware (more precisely for IA32-based systems). Fiasco is a real-time microkernel operating system. It has been developed in the context of the DROPS project [7] and supports the flexible construction of applications with security or quality of service requirements. As a microkernel, Fiasco has a minimal interface and supports only

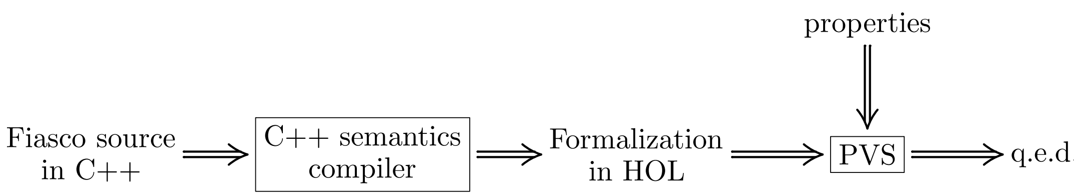
This work was supported by the Deutsche Forschungsgemeinschaft (DFG) through DFG grant Re 874/2-1.

the absolutely necessary operating-system functionality. There are, for instance, no device drivers included in Fiasco. For legacy applications it is possible to boot Linux on top of Fiasco [8].

Fiasco is almost entirely written in C++ [20]. Only those operations that cannot be performed in C++ (like accessing CPU control registers) are written in inline assembler. The properties that we attempt the prove in the VFiasco project include the following:

- Fiasco's internal page-fault handler terminates for all kernel page faults
- Fiasco's internal memory allocation works correctly

For the verification, we plan to employ a state-of-the-art general-purpose theorem prover. Our current development focuses on PVS [17]. For the verification, the C++ source code of Fiasco will be fed into a semantics compiler that generates the semantics of the input as shallow embedding[1] in the higher-order logic of PVS. Specifications and properties will be formulated directly in the logic of PVS. The whole verification process can be depicted as follows:



The semantics compiler is currently developed on the basis of the OpenC++ package [4]. For the translation of Fiasco's source code we need a semantics for a substantial part of C++.

In this paper we concentrate on the built-in types and the type constructions (like structs and unions) of C++. Other details of our semantics of C++ will be described elsewhere, see [10, 11]. We base our formalisation of the data types on the C++ ISO Standard [5]. In the following we call this document simply *the standard*. The notation §$n.m(k)$ refers to Section $n.m$, paragraph k in the standard. In the standard the data types are mainly described in §3.9 and §9. We have the following requirements for our semantics:

1. The semantics is suitable for reasoning about low-level systems programs, especially operating systems written in C++, like Fiasco.
2. The semantics scales well to the formalisation of a specific C++ implementation that determines the behaviour of certain language constructs that is unspecified in the standard.

[1] In a shallow embedding an external tool translates the sources into their semantics. In contrast, in a deep embedding one can represent phrases of the source in the logic and the semantic function is expressed inside the logic.

3. The semantics supports popular unsafe C++ programming idioms (like allocating a `char` array and casting its base address into a specific pointer type) together with programming idioms that are necessary for operating-systems programming (like casting unsigned integers to pointers).

4. The semantics is sensible enough to detect subtle programming errors, like reading data from uninitialised memory.

The motivation for these requirements is clear from the context of this work: After all we want to use the semantics in a concrete verification project. Let us now discuss the implications of these requirements.

The first consequence of Requirement 1 (reasoning about low level code) is that we cannot use the traditional way of modelling dynamically allocated structures. For instance in [3, 15] dynamically allocated structures are modelled as functions of the form $heap : address \longrightarrow value\text{-}type$. In this approach the heap is implicitly typed, making it impossible to describe programs which possibly contain type errors (like reading an integer from a memory location that contains a string). However, one of the goals of the VFiasco project is to prove the absence of such typing errors. This is impossible in an implicitly-typed semantics. Our model of the heap must support operations like pointer-type conversion and also interpreting the same data in different types.

For the second consequence of the first requirement recall again our long term goal: Verifying Fiasco. For that we are currently developing an x86 hardware model in PVS. The hardware model gives an abstraction of the execution environment of x86 compatible processors. A state of the hardware model will be an abstraction of the state of real-world PC. Such a state contains a snapshot of the physical memory together with some important control registers on the CPU (like `cr3`, which determines the base address of the current page directory for the virtual-memory address translation). For the verification, the semantics of the Fiasco sources will be interpreted on the hardware model. In order to ensure that the verification results apply to the real-world Fiasco, it is necessary that our hardware model models x86 hardware — and nothing more. The conclusion is that the semantics of data types must not add information to the states of the hardware model, like, for instance, associating type tags to memory locations.

The second requirement (scalability of the semantics) originates from the following: The C++ standard is very vague about almost every C++ construct. For the built-in integer types it does not specify which values they can represent, for signed integer types it is left open if they can represent negative numbers. One of the few things that are required is that all built-in integer types are at least 7 bits wide.[2] A specific C++ implementation can define things that are left open in the standard. For instance the gcc compiler uses 8 bits for `char` and 32 bits for `int`. It is impossible to write (and verify!) an operating system without such implementation-specific knowledge. For instance, Fiasco relies on the fact that the standard conversion from any pointer type to `unsigned` and back is the identity

[2]Because they all can represent the 96 characters from the *basic source character set*, see §§ 2.2.1, 3.9.1.2

function on the underlying bit representation. The data-type semantics in this paper will easily scale to (possibly different) implementations. The base version of the semantics formalises the C++ standard. From the base version one can derive an implementation-specific version by just adding the additional properties.

Requirement 3 (support for unsafe idioms) is (again) motivated by our long-term goal in the VFiasco project. The semantics must support all the program idioms that are necessary for writing an operating system. This includes casting (seemingly) arbitrary unsigned integers into pointers and dereferencing these pointers.

Last but not least we want the semantics to catch all possible programming errors (Requirement 4. Consider the following C++ statement[3]

```
T a = * reinterpret_cast<T *>(50)
```

It accesses an object of type `T` that is located at address 50. At the time where this statement is executed there might or might not be a valid object of type `T` stored at address 50. Statements of this kind are necessary in an operating system, for instance in the code that traverses page directories. Therefore we need a semantics for data types that permits the above statement (with the expected semantics) provided one can prove that there is a valid object of type `T` stored at address 50. However, we also want to catch programming errors where the above statement is executed before the memory at address 50 is properly initialised. Therefore, without any knowledge about the memory contents one should not be able to derive anything about the above statement (not even that it does not crash the hardware).

For a second example about which programming errors we want to catch, consider the following piece of code:

```
unsigned a[2] = { 1, 2 };
unsigned b[3];
memcpy(static_cast<char *>(static_cast<void *>(b)) + 1, a, sizeof(a));
unsigned c = * static_cast<int *> (static_cast<void *>
                        (static_cast<char *> (static_cast<void *> (b)) + 1));
unsigned d = b[1];
```

The `memcpy` function copies the two integers 1 and 2 in array `a` into array `b`. Note that the copy is displaced by an offset of 1. According to the C++ standard such a `memcpy` is legitimate. Moreover, it specifies that the fourth statement (which reads the integer 1 at its new address) initialises `c` correctly with 1 (§3.9.3) provided the alignment requirements are met. For any compiler we know, also the last statement will execute without problems. On x86 hardware (with little endian byte ordering) it will combine the most significant byte of 1 (which is 0) together with the three less significant bytes of 2 to a result of 512 (decimal). However, the C++ standard leaves the behaviour of the last statement open (it

[3]The C++ standard says that the behaviour of this statement is undefined. However for gcc it is well defined.

could potentially crash the machine). We would like to view the evaluation of `b[1]` as an error (because it reads an integer where none has been stored before). In the semantics we present here, one cannot even prove that the last program does not crash.

Related Work Traditionally, program verification focuses on well-typed programming languages (see for instance [2]), which have a relatively simple semantics. However, as explained before, in the VFiasco project we *need* to reason about an unsafe programming language. Recently, in the work on proof-carrying code, type systems and semantics have been developed for assembly languages [16]. However, the type systems developed for proof-carrying code are not well suited for complex analysis. With our present work we try to find a compromise between reasoning on an abstract level and the ability to treat possibly illtyped programs.

A semantics for C and C++ has also been described in the framework of abstract state machines [6, 23]. There exist simulators for abstract state machines, but, to our knowledge, no theorem-proving support.

Our work is very much inspired by the work in the LOOP project for reasoning about Java programs [22]. Jacobs studies in [14] the integral types of Java. The main difference to our work is (apart from the source languages) that for Java's integral types *nothing* is left unspecified. Therefore Jacobs can model his semantics as a definitional extention in PVS. However, with Jacobs semantics one cannot reason about the absence of over- or underflows.

Acknowledgements We would like to thank Sarah Hoffmann, Matthias Daum, and Shane G. Stephens for many discussions on the subject.

2 General Properties of C++ Data Types

In this section we analyse the general properties of data types in C++ and explain the general approach of our semantics. In the standard one can distinguish three types of descriptions of the behaviour of an operation:

1. The standard leaves the behaviour (explicitly or implicitly) unspecified (permitting the compiler to reject the program, the program to crash, or to continue with the most sensible result).
2. The standard says that the evaluation terminates normally but the results are unspecified (the program must not crash but one cannot rely on the result).
3. The standard describes the behaviour in detail.

For the first two points the standard distinguishes between *implementation defined* and unspecified behaviour. This distinction is not important for us.

For the development of the semantics we use the following approach: For a description of Type 1 or 2 we use an undetermined axiomatic specification with uninterpreted constants.

For Type 2 we add appropriate additional axioms that ensure termination. For type 3 we use a definition. The effect is as follows: The available proof power correlates precisely to what one knows about an execution of the program on an arbitrarily chosen C++ implementation (including all hypothetical ones). Consider, for instance, a program that uses an unsigned-integer operation that is only guaranteed to terminate. The result of that operation will subsequently be divided by 2 and stored in some variable v. For such a program one can prove termination under the precondition that the unspecified integer operation does not return zero. Further, one can derive that the value in v is less than or equal to maximal unsigned integer divided by 2. Any attempt to prove something more specific will fail.

The standard divides all types into POD (*plain old data*) types and non-POD types (see §§3.9 (10), 9 (4)). Basically, POD types are those types that are compatible with C. That is, all arithmetic types, structs, and unions are POD, provided they do not contain any virtual functions, constructors, destructors and the like.

The standard describes the C++ memory and object model in §1.7 and §1.8. The interesting point for us is that objects (i.e., memory representations of values of C++ types) have an address, a type, and occupy some memory. Objects of POD types are stored contiguously in memory and they can be copied or moved (§3.9 (2)). Objects of non-POD type cannot be moved and one has to use the copy constructor or assignment operator to copy them. Some common aspects of all types are described in §3.9, the built-in types (called *fundamental types*) are described in §3.9.1. The properties of classes, structures, unions, and bit fields are laid out in §9.

For any type T the standard distinguishes between its *value type* (i.e., the set of values of T), the *value representation* (i.e., the bit string that represents a value), and the *object representation* (i.e., the bit string that is stored in memory). There are special requirements for the three character types that we discuss below. For all other types the object representation can be different from the value representation (which is indeed the case on all little-endian machines). The mapping from the value representation to the values might not be injective (for instance it is possible to use one-complement encoding for integers with two different representations of zero: $+0$ and -0).

The value representation is only used in the description of the bit-wise operations in §§5.8, 5.11–5.13. However, the standard either defines the bitwise operations precisely in terms of the argument values or leaves the result undefined. We therefore decided to ignore the value representation. In our semantics we model values and their object representation.

To relate object and value representation the standard states in §3.9 (4):

> *For POD types, the value representation is a set of bits in the object representation that determines a value . . .*

It is a fundamental observation that the preceding citation it *the only* requirement that the standard puts on the connection of object and value representation. In particular, the object representation might contain more bits than the value representation. These additional bits can be used for arbitrary purposes. The C++ runtime system can, for instance, use them to encode the object type into the object representation and to perform Lisp-like runtime type checking. For non-POD types one could even encode the memory location of the object into the object representation to detect (at runtime) if the object has been illegally moved with `memcpy`. We can therefore derive the following observation.

Observation 1 *A C++ program that does not crash on any C++ implementation must be type correct in the following sense:*

- *For any type T which is not a character type, it accesses objects of type T only at memory locations that contain a correctly initialised object of type T.*
- *It accesses members of any non-POD type T only at memory locations that have been initialised by a constructor or assignment operator for T.*

To summarise: For the semantics of a data type we need a value type and functions that translate values into their object representation and back. These functions (and if necessary the value type) must be undetermined to a certain extent to model all possible C++ implementations. To prove type correctness of a C++ program it is then sufficient to prove that the program does not crash.

3 Common PVS Formalisation

This section describes the common infrastructure that we use for all types. Let us fix some notions before we turn to the PVS sources. Consider a C++ data type T. Any C++ implementation defines a value type and an object representation for T. We call such an implementation of a type T a *model of* T. As semantics of T we develop a specification for all possible models of T.

Our semantics will be independent of the underlying (model of the) memory. This makes it possible to reuse the same data-type semantics for all the memory abstractions that we are developing in the VFiasco project [12]. However, it may be helpful to imagine an oversimplified memory model, consisting of a type State containing all memory states, and two functions (in PVS syntax):

```
memory_read : [State, Address -> Byte]
memory_write : [State, Address, Byte -> State]
```

Here, Byte is the type of all values that can be stored in a byte and Address is the type of addresses (as a subtype of natural numbers). The precise definition of these two types does not matter for the contents of this paper.

These memory-access functions can be extended in the obvious way to

```
memory_read_list : [State, Address, nat -> list[Byte]]
memory_write_list : [State, Address, list[Byte] -> State]
```

In our semantics, every C++ data type T is modelled in PVS by its value type together with a record that contains the basic common operations. The operations in the record depend on the value type. Its definition in PVS is:

```
Data_type_structure : Type = [#
  size : nat,
  to_byte : [Data, Address -> list[Byte]],
  from_byte : [list[Byte], Address -> lift[Data]]     #]
```

The types Address and Byte are as before. The type parameter Data stands for the value type of T. The first field size contains the length of the object representation (in bytes). It corresponds to the `sizeof` operator of C++. The functions to_byte and from_byte convert values into their object representation and back. As an additional argument they take the address of the object representation in the memory. This way the function to_byte could encode the address in the object representation and from_byte can check it to prevent illegal copying. Naturally from_byte is a partial function that is only defined on valid object representations. We use the traditional PVS approach and represent a partial function $X \rightharpoonup Y$ as $X \to \mathsf{lift}[Y]$ in PVS.[4]

For any model of a C++ type we require some basic properties. They are combined in a predicate as follows:

```
data_type? : PRED[Data_type_structure] = Lambda(ct : Data_type_structure) :
  (Forall(d : Data, a : Address) : length(to_byte(ct)(d,a)) = size(ct)) And
  (Forall(d : Data, a : Address) : up?(from_byte(ct)(to_byte(ct)(d,a), a))) And
  (Forall(d : Data, a : Address) : down(from_byte(ct)(to_byte(ct)(d,a), a)) = d ) And
  (Forall(l : list[Byte], a : Address) : up?(from_byte(ct)(l, a)) Implies length(l) = size(ct))
```

This predicate requires that

- the object representation of all values is size(ct) bytes long.
- from_byte is a left inverse of to_byte (i.e., from_byte is defined on all results of to_byte and yields the original value).
- from_byte fails on objects of the wrong size

In contrast to the standard that requires all objects to occupy some memory (§1.8 (5)) we do not require the size field to be positive. A size of zero makes sense for `void`, see Subsection 4.5. Inhabitants of the predicate subtype (data_type?[Data]) are models of C++ types with Data as value type.

For objects of a POD-type the standard requires that one can copy the object to a different memory location. Therefore we strengthen the data_type? predicate for POD types:

```
pod_data_type? : PRED[Data_type_structure] =
  Lambda(ct : Data_type_structure) : data_type?(ct) And
    (Forall(d : Data, a1,a2 : Address) : up?(from_byte(ct)(to_byte(ct)(d,a1), a2)))
```

[4] The type constructor lift corresponds to option in Isabelle. In PVS (the representation of) a partial function returns bottom if it is undefined and up(-) otherwise.

This addition ensures that the from_byte function is successful regardless of the address on which we find a valid object representation.[5]

For any model dt of type (data_type?[Data]) one can now define two functions that attempt to read or write a value into the memory:

```
read_data(dt)(s : State, addr : Address) : lift[Data] =
  from_byte(dt)(memory_read_list(s, size(dt), addr), addr)

write_data(dt)(s : State, addr : Address, data : Data) : State =
  memory_write_list(s, addr, to_byte(dt)(data, addr))
```

We can now precisely define what we mean with the semantics and a model of a data type. Note that in PVS any constant or function definition[6] determines a specific value, even if the axiom of choice is used in the definition. Therefore the only way to achieve the needed undeterminedness in our semantics is to use declarations in an axiomatic specification. This approach requires special efforts to maintain soundness.

Definition 2 (Semantics) The semantics of a C++ type T in PVS consists of

- a type definition Semantics_T for the value type
- a constant declaration of type Data_type_structure providing the common operations; for non-POD types this constant must be in data_type?[Semantics_T] and for POD types in pod_data_type?[Semantics_T]
- a finite number of axioms stating additional properties

Note that one can add further axioms to any existing semantics of a type T to obtain a new semantics of T. This feature ensures scalability in the sense of Requirement 2 from the introduction. We will exploit this feature in the following way: First we develop a semantics for the data types as described in the standard. In a second step we derive a specific semantics for the GNU gcc compiler by adding more axioms.

For some types the standard leaves the value type implementation-defined. A semantics of such a type will fix the the super type of all value types of all models. The value type Semantics_T can then be defined as a predicate subtype that involves an uninterpreted constant (such that the precise range of Semantics_T stays undefined). We use this technique for all signed integer types, see Subsection 4.2 below.

Definition 3 (Model) A model of a C++ type T in PVS consists of a type definition Model_T and a constant definition of type (data_type?[Model_T]), or, if T is POD, (pod_data_type?[Model_T]) such that these two definitions

- do not involve uninterpreted constants or types

[5] We ignore alignment issues here, see Subsection 6.

[6] Here we mean *interpreted constant definitions* in the sense of the PVS Language Reference [19]. We refer to *Uninterpreted constant definitions* as *declarations*.

- fulfil the axioms from the semantics of T

In principle the model relationship can be established with the notion of theory interpretations [18] that have been introduced in PVS recently. A theory interpretation provides definitions for uninterpreted constants. Axioms involving the constants reappear as proof obligations. With theory interpretations it is possible to establish the soundness of an axiomatic specification *whithin* PVS. However, in the current PVS version there remain a few issues[7] to be resolved before theory interpretations can be applied in our context.

To maintain consistency we develop a model for every data type together with its semantics. We also plan that our C++ semantics compiler will generate *both* the semantics and a model for every used data type.

Closely related with the data types are the C++ standard conversions. Here we treat the semantics of data types and that of the conversions separately although they are very closely related: Sometimes the standard specifies a property of a data type in an indirect way by putting requirements on the conversion functions.

The semantics of a conversion function is an appropriately typed constant declaration. It is accompanied with axioms in case the standard restricts the behaviour of the conversion. As one expects, a model of a conversion is a function definition.

4 Fundamental Types

In this section we describe the semantics of the fundamental types of C++.

4.1 Booleans

For booleans the standard is very clear: "Values of type bool are either true or false." (§3.9.1 (6)). For the semantics of `bool` we set

```
Semantics_bool: Type = bool
dt_bool_exists : Axiom Exists (x: (pod_data_type?[Semantics_bool])): true
dt_bool : (pod_data_type?[Semantics_bool])
```

We need the axiom on the second line to discharge the existence TCC[8] for the declaration dt_bool. Note that the semantics does not stipulate that the value **true** is represented as a non-zero byte. The standard only says that the result of *converting* **true** into an integer type is one.

[7] See PVS bug reports 757–760 on `http://pvs.csl.sri.com/cgi-bin/pvs/pvs-bug-list/`.

[8] A *Type check condition* (TCC) is a proof obligation generated by the type checker when it cannot decide whether an expression is type-correct or not. For PVS TCC's are necessary because the type system is not decidable.

For a model of `bool` we only have to define dt_bool (in a different theory):

```
dt_bool : (pod_data_type?[Semantics_bool]) = (#
  size := 1,
  to_byte := Lambda(b : bool, a : Address) : IF b Then (: 1 :) Else (: 0 :) Endif,
  from_byte := Lambda(bl : list[Byte], a : Address) :
                 IF bl = (: 1 :) Then up(true)
                 Elsif bl = (: 0 :) Then up(false)
                 Else bottom   Endif    #)
```

Note that for this definition PVS generates a TCC that requires us to prove that the POD data-type properties hold. Now one can use theory interpretations to show that the semantics of booleans is sound. A more simple-minded approach is to copy the only axiom from the semantics of booleans and prove it as a lemma:

```
dt_bool_exists : Challenge Exists (x: (data_type?[Semantics_bool])): true
```

4.2 Signed Integers

There are four signed integer types in C++: `signed char`, `short int`, `int`, and `long int`. Character types are treated separately in Subsection 4.4 because of their special requirements. For the other signed integer types the standard does not say much. For instance it is not required that they can hold negative numbers. We only show the semantics of `int` here, the semantics of the other types is very similar.

```
Cxx_Int : Theory
Begin
  int_bits: posnat
  semantics_int_pred : PRED[int]

  Importing Cxx_Sshort
  int_longer : Axiom sshort_bits <= int_bits
  int_contains_sshort : Axiom subset?(semantics_sshort_pred, semantics_int_pred)

  Semantics_int : Type = (semantics_int_pred)
  Importing Abstract_Data[Semantics_int]
  dt_int_exists : Axiom Exists (x: (pod_data_type?[Semantics_int])): True
  dt_int : (pod_data_type?[Semantics_int])
End Cxx_Int
```

The identifiers with sshort refer to the corresponding items from the semantics of `signed short`. First we declare the size of the value representation, this becomes important for the unsigned integer types, see below. We define the value type Semantics_int as a predicate subtype of the PVS integer type int. The axioms int_longer and int_contains_sshort formalise the requirement that "[short int] provides at least as much storage as [int]" (§3.9.1 (2)).

The standard further requires that the value representation uses a *pure binary numeration system* (§3.9.1 (7)). However, it is unclear to us in which way a program could rely on the use of a pure binary numeration system. Programs that do use integers usually rely on the fact that at least a certain interval can be represented in the integer types. Most C++ implementations specify the value type of the integer types. Therefore we do not bother to axiomatise pure binary numeration systems. Instead we rely on additional assumptions on the value type of the integers. To obtain the integer type of the GNU C++ compiler on x86 one could use the following theory:

```
Gnu_IA32_Int : Theory
Begin
  Importing Cxx_Int
  int_bits_ia32 : Axiom int_bits = 32
  int_pred_ia32 : Axiom semantics_int_pred = { i : int | −2^31 <= i And i < 2^31}
End Gnu_IA32_Int
```

The models for the integer types are the obvious ones. We skip their presentation here.

4.3 Unsigned Integers

For each signed integer there is a corresponding unsigned integer. For the unsigned integer types the standard specifies arithmetic modulo 2^n, where n is the number of bits in the value representation. The formalisation of the type `unsigned` is as follows:

```
Cxx_Unsigned : Theory
Begin
  unsigned_bits: posnat
  semantics_unsigned_pred : PRED[nat] = { n : nat | n < 2^unsigned_bits }

  Importing Cxx_Int, Cxx_Ushort
  unsigned_same_size : Axiom unsigned_bits = int_bits
  unsigned_inclusion : Axiom
    Forall(i : Semantics_int) : i >= 0 Implies semantics_unsigned_pred(i)
  unsigned_longer : Lemma ushort_bits <= unsigned_bits
  unsigned_contains_ushort : Lemma
        subset?(semantics_ushort_pred, semantics_unsigned_pred)

  Semantics_unsigned : Type = (semantics_unsigned_pred)
  Importing Abstract_Data[Semantics_unsigned]
  dt_unsigned_exists : Axiom Exists (x: (pod_data_type?[Semantics_unsigned])): True
  dt_unsigned : (pod_data_type?[Semantics_unsigned])
End Cxx_Unsigned
```

For unsigned integers the value type depends only on the number of bits in the value representation. The two axioms unsigned_same_size and unsigned_inclusion formalise that the value representation of `int` and `unsigned int` has the same size and that a positive value of `int` is also a value of `unsigned int`. Other requirements of the standard follow now as lemma: for instance that the value representation of `unsigned int` is longer than that of `unsigned short`.

4.4 Character Types

There are three different character types in C++, `unsigned char`, `signed char`, and `char`. The value type of `char` coincides with either `signed char` or `unsigned char`. Note that character types are integer types, that is, their values are integers and not characters. The special property of the character types is that one can copy every POD object into an array of a character type of sufficient length (see §3.9 (2) and issue 350 in [1]). This must work even if the memory of the source object has not been correctly initialised. The semantics of signed characters is as follows:

```
1  Cxx_Schar : Theory
2  Begin
3    schar_bits: posnat
4    semantics_schar_pred : PRED[int]
5    Semantics_schar : Type = (semantics_schar_pred)
6    Importing Abstract_Data[Semantics_schar]
7    dt_schar_exists : Axiom Exists (x: (pod_data_type?[Semantics_schar])): True
8    dt_schar : (pod_data_type?[Semantics_schar])

9    dt_schar_from_byte_total: Axiom Forall (l: list[Byte], a : Address):
10     length(l) = size(dt_schar) Implies up?(from_byte(dt_schar)(l, a))
11   dt_schar_injective: Axiom Forall (l: list[Byte], a : Address):
12     length(l) = size(dt_schar) Implies
13       to_byte(dt_schar)(down(from_byte(dt_schar)(l, a)), a) = l
14 End Cxx_Schar
```

Up to line eight we have the usual semantics for signed integer types. The remainder formalises the ability to copy POD objects. The axiom on line nine expresses that one can interpret every piece of memory as signed character (the from_byte function never fails). The last axiom ensures that the data does not change when it is copied (the from_byte function is injective on byte lists of the right length).

Both properties need not hold for any non-character type T: There might be a bit pattern that does not describe a value of T, such that from_byte fails. There also might be two bit patterns that describe the same value (like $+0$ and -0 in a one's-complement representation), such that from_byte is not injective.

Our axiomatisation of signed characters implies that there is a bijective correspondence between the value type (Semantics_schar) and the object representation (byte lists of length size(dt_schar)). This can be proved within PVS (where List_len(l) is the type of lists of length l):

```
schar_iso : Lemma
  Exists(f : [Semantics_schar -> List_len[Byte](size(dt_schar))]) : bijective?(f)
```

The semantics of `unsigned char` is very similar to the one of signed characters. It only fixes the value type (like all other unsigned types). For the semantics of `char` we add the following to the usual setup:

signed_or_unsigned : **Axiom**
(semantics_char_pred = semantics_uchar_pred) Xor
(semantics_char_pred = semantics_schar_pred)

char_same_size : **Axiom**
(semantics_char_pred = semantics_uchar_pred **Implies** size(dt_char) = size(dt_uchar)) **And**
(semantics_char_pred = semantics_schar_pred **Implies** size(dt_char) = size(dt_schar))

These axioms require that the value type of `char` coincides with either `unsigned char` or `signed char`. Further the object representation must have the right size. With these assumptions it is possible to prove in PVS that also `char` has the ability to copy POD objects:

dt_char_from_byte_total: **Lemma Forall** (l: list[Byte], a : Address):
length(l) = size(dt_char) **Implies** up?(from_byte(dt_char)(l, a))

dt_char_injective: **Lemma Forall** (l: list[Byte], a : Address):
length(l) = size(dt_char) **Implies** to_byte(dt_char)(down(from_byte(dt_char)(l, a)), a) = l

The proof is not trivial: Consider the function to_byte and fix the second argument of type Address. The data_type? predicate implies that this function is injective with a finite codomain (byte lists of a fixed length). Therefore also its domain, the value type of `char`, must be finite. The Lemma schar_iso and the two axioms for `char` imply that the value type and the object representation of `char` have the same cardinality. So the to_byte function (with fixed second argument) must also be surjective. The preceding two lemmas follow now because from_byte is a left inverse of to_byte.

4.5 Void

The type `void` is very special. In C++ it is used as return type for functions that only produce side effects. Besides that, any value can be converted into type `void`, so there are expressions of type `void`. Nevertheless the standard specifies `void` as an *empty* type (§3.9.1.(9)). For a set-theoretic interpretation all this does not make much sense.

For the semantics of `void` we see the following alternative:

- Model `void` as the empty type (which does exist in PVS). In this case all functions and conversions with a codomain of `void` must be specially treated such that they do not create inhabitants in the empty type.
- Model `void` as an one-element type. In this case it is difficult to stay consistent with most C++ implementations, which optimise away any value of type void. One possible way is to allow that the only inhabitant of `void` does not occupy any memory.

We opt for the second alternative and explicitly permit models of `void` with empty object representation (i.e., size(dt_void) = 0).

4.6 Standard Conversions

The formalisation of the standard conversions is straigthforward. Here we show only three examples:

```
cnv_sc2b(sc : Semantics_schar) : Semantics_bool = sc /= 0
cnv_b2sc(b : Semantics_bool) : Semantics_schar = IF b Then 1 Else 0 Endif

cnv_uc2sc(uc : Semantics_uchar) : Semantics_schar
cnv_uc2sc_prop: Axiom Semantics_schar_pred(uc) Implies cnv_uc2sc (uc) = uc
```

The first two conversions from `signed char` to `bool` and back are completely specified in §4.7 (4) and §4.12 (1). Therefore we define the semantics of these conversions as functions. The behaviour of the third conversion from `unsigned char` to `signed char` is only partially specified in §4.7 (3), so we use a declaration together with an axiom.

5 Structures

Recall that we aim at a shallow embedding of C++ in PVS. Therefore we cannot formalise the semantics of all structures or all union types. We can only describe rules how to generate a semantics for a specific compound type. These rules will be compositional in the sense that the semantics of a compound type T that contains a member of type T_m, relies on the semantics of T_m. Our semantics compiler will implement these rules and generate a semantics for every compound type in the source. In this section we describe the semantics of structures. The other compound types remain future work, see Section 6.

The C++ version of the cartesian product is called `struct`. A structure combines several members of different types. Here is a typical example:

```
struct Z { int x; char y; };
```

It defines a new type Z. Any object of that type contains an integer and a character. Both can be accessed independently. There are several specific things for C++ structures: First one can access the whole structure or individual fields. The access to one field is independent of whether the other fields are initialised or not. It is perfectly legitimate to work with partly initialised structures, as long as one accesses only the initilized fields. Further, structures may contain padding, that is, there might be unused memory between any two fields of a structure (for instance, to satisfy alignment requirements). However, for POD structures, there is no padding at the beginning (§9.2 (17)). If there is no intervening access specifier the members are allocated in the same order as they are declared in the source code.

Note that classes in C++ are structures with a different default for the access specifier (**private** for classes, **public** for structures). In the semantics we do not distinguish between classes and structures.

In the following we present the semantics of the example structure Z. The general semantics for structures becomes clear from that. First we define the value type for the structure Z as a record of its member types. Next we define a second record for keeping the offsets of the individual members in the structure:

```
Semantics_Z : Type = [# x: Semantics_int, y: Semantics_char #]
Offsets_Z : Type =    [# x_offs: nat, y_offs: nat #]
```

Next we have the usual declarations for the common data type structure together with a constant offsets_Z that provides the indices of the members in the object representation of Z:

```
dt_Z_exists : Axiom Exists (x: (pod_data_type?[Semantics_Z])): true
dt_Z : (pod_data_type?[Semantics_Z])
offsets_Z : Offsets_Z
```

The semantics is completed with three axioms. The first one describes the order of the indices in offsets_Z and requires that the individual objects in the structure do not overlap:

```
offs_order_Z: Axiom 0 = x_offs(offsets_Z) And
  x_offs(offsets_Z) + size(dt_int) <= y_offs(offsets_Z) And
  y_offs(offsets_Z) + size(dt_char) <= size(dt_Z)
```

If the declaration of the structure contains access specifiers, then this axiom changes slightly and fixes only the order of members that are not separated by an access specifier (§9.2 (12)).

The second axiom states that one can access the individual fields if one can access the whole structure. Further, it does not matter whether one accesses one field or accesses the whole structure and selects that field in the value type.

```
from_byte_whole_Z: Axiom
  Forall(bl: list[Byte], a : Address):    up?(from_byte(dt_Z)(bl,a)) Implies
    Let x = down(from_byte(dt_Z)(bl,a)),
        lift_a = from_byte(dt_int)(sublist(x_offs(offsets_Z), size(dt_int))(bl),
                                   a + x_offs(offsets_Z)),
        lift_b = from_byte(dt_char)(sublist(y_offs(offsets_Z), size(dt_char))(bl),
                                    a + y_offs(offsets_Z))
    IN
        up?(lift_a) And down(lift_a) = x(x) And
        up?(lift_b) And down(lift_b) = y(x)
```

The function sublist(offs, len)(l) cuts out the sublist of length len of l that starts at offs. The third axiom is kind of inverse: If one can access all fields then one can access the whole structure and get the expected result:

```
from_byte_parts_Z: Axiom
  Forall(bl: list[Byte], a : Address):     length(bl) = size(dt_Z) Implies
    Let lift_x = from_byte(dt_Z)(bl, a),
        lift_a = from_byte(dt_int)(sublist(x_offs(offsets_Z), size(dt_int))(bl),
                                                      a + x_offs(offsets_Z)),
        lift_b = from_byte(dt_char)(sublist(y_offs(offsets_Z), size(dt_char))(bl),
                                                      a + y_offs(offsets_Z))
    IN
      up?(lift_a) And up?(lift_b) Implies
        up?(lift_x) And down(lift_a) = down(lift_x)`x And down(lift_b) = down(lift_x)`y
```

To prove consistency one can use a model that stores all members one after each other in one byte string of sufficient length. The required proofs are easy once one has the appropriate rewrite lemmas about sublist.

In this simple example we have neglected a few issues because their treatment is rather obvious: Inheritance of structures becomes a substructure hierarchy in the semantics. If sharing occurs (virtual base classes) one has axioms that state that different substructures are identical. Static member functions are not part of the structure, they are modelled as part of the program. The semantics of a structure with virtual member functions contains an additional field that holds its dynamic type. This field is used for the semantics of late binding. The details will be described in [10].

6 Future Work

In this section we remark on those parts of the semantics that have not been fully worked out yet.

Unions Unions are like structures with the important difference that all members are stored in it with offset zero. So in a union only one of its members can be active at any time. For the semantics of unions we need to express the value type of a union in the PVS type system. For this we cannot use disjoint unions because in the usual model of unions one cannot decide which member is active. The construction of a type constructor that is suitable for the value types remains future work.

Pointers The standard guarantees only very few things about pointers. One can, for instance, convert any pointer into a void pointer and back, without loosing information. For all the other pointer conversions the standard only requires that they preserve the null pointer. For the efficient implementation of an operating system one needs much more properties. For the verification of Fiasco it makes sense to identify the address type of the memory abstraction with the value type of all pointer types. In this case the semantics cannot detect if a pointer moves past the array bounds. For a general semantics of C++

it would be nice to permit also smart pointers as models of the pointer types. A smart pointer keeps information about its associated array and its type to detect errors of pointer arithmetic. To permit smart pointers it is necessary to leave the value type for pointers open.

Constant Objects In C++ one can qualify any object as constant. Intuitively a const object should never be changed after its initialisations (but compare the open issue 290 in [1]). The general setup of our semantics cannot faithfully model constant objects. There are the following two possible solutions.

- The semantics compiler rejects any program that casts a constant type into a non-constant type. For the remaining program one can check statically if they treat constant objects correctly.
- One enriches the memory interface to permit an operation that changes memory areas to read-only at runtime. In this case the semantics is able to detect an attempt to write to a constant object. One can then verify programs that need to cast away the `const` modifier because some arguments of a library function are (erroneously) not declared as constant.

For the VFiasco project the first solution is suffient.

Alignment Alignment describes the inability of some CPU's to access all data on all possible addresses. For instance on a sparc architecture a 2 byte memory access is only possible on even addresses. The x86 architecture has no alignment requirements. The framework presented in this paper does not support alinment requirements. An easy way to support alignment is to make the to_byte function partial. It can then fail if alignment requirements are not met. At the moment it is not clear to us how to model alignment requirements such that it applies to all possible architectures.

Float and Double The standard says almost nothing about the floating point types (apart from that they exist). For the VFiasco project the floating point types are not necessary. For the verification of C++ floating point programs one needs more assumptions than the standard provides. For this one can reuse one of the several floating point formalisations, for instance [13].

Miscellaneous A complete semantics for the C++ data types must also treat references, bit fields, and volatile objects. For the VFiasco project only bit fields are relevant.

7 Conclusion

In this paper we describe a semantics of the data types of the C++ programming language. The semantics can deal with dynamically allocated objects, pointer conversion, and even with typing errors. The semantics can easily be adjusted to model additional features of a specific C++ implementation. This paper reports on work-in-progress. Here, we describe the general setup, the semantics of the fundamental or builtin types, and that of structures. We only comment on the other data type constructions.

References

[1] JTC1/SC22 Working Group 21. C++ standard core language active issues, revision 26, April 2003. available via http://std.dkuug.dk/JTC1/SC22/WG21/.

[2] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, Berlin, 1991.

[3] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland., 1972.

[4] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, October 1995.

[5] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, September 1998.

[6] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[7] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[8] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[9] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[10] M. Hohmuth and H. Tews. The C++ object model: A semantics for late binding and pure virtual functions. In preparation.

[11] M. Hohmuth and H. Tews. A semantics for C++ statements: Formalising harmful goto's, Duff's device and other monstrosities. In preparation.

[12] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project (extended abstract). In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[13] C. Jacobi. Formal verification of a theory of ieee rounding. In R.J. Boulton and P.B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, Informatics Research Report EDI-INF-RR-0046, Edinburgh, UK, 2001.

[14] B. Jacobs. Java's integral types in pvs, 2003. Manuscript.

[15] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, Lecture Notes in Computer Science. Springer, 2003.

[16] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[17] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, Berlin, 1996.

[18] S. Owre and N. Shankar. Theory interpretations in pvs. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.

[19] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.4*. SRI International, Menlo Park, CA, December 2001.

[20] B. Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[21] H. Tews, H. Härtig, and M. Hohmuth. VFiasco — towards a provably correct μ–kernel. Technical Report TUD-FI01-1 – January 2001, Dresden University of Technology, Department of Computer Science, 2001. Available via http://wwwtcs.inf.tu-dresden.de/~tews/science.html.

[22] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, 2001.

[23] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.