Formal verification of object layout for C++ multiple inheritance

Tahina Ramananandro, Gabriel dos Reis, Xavier Leroy

Formal Verification of Object Layout for C++ Multiple Inheritance

Tahina Ramananandro
INRIA Paris-Rocquencourt
Tahina.Ramananandro@inria.fr

Gabriel Dos Reis *
Texas A&M University
gdr@cs.tamu.edu

Xavier Leroy †
INRIA Paris-Rocquencourt
Xavier.Leroy@inria.fr

Abstract

Object layout — the concrete in-memory representation of objects — raises many delicate issues in the case of the C++ language, owing in particular to multiple inheritance, C compatibility and separate compilation. This paper formalizes a family of C++ object layout schemes and mechanically proves their correctness against the operational semantics for multiple inheritance of Wasserrab *et al*. This formalization is flexible enough to account for space-saving techniques such as empty base class optimization and tail-padding optimization. As an application, we obtain the first formal correctness proofs for realistic, optimized object layout algorithms, including one based on the popular "common vendor" Itanium C++ application binary interface. This work provides semantic foundations to discover and justify new layout optimizations; it is also a first step towards the verification of a C++ compiler front-end.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—*Correctness proofs*; D.3.3 [*Programming Languages*]: Language Constructs and Features—*Classes and objects*; D.3.4 [*Programming Languages*]: Processors—*Compilers*; E.2 [*Data storage representations*]: Object representation; F.3.3 [*Logics and meanings of programs*]: Studies of program constructs—*Object-oriented constructs*

General Terms Languages, Verification

1. Introduction

One of the responsibilities of compilers and interpreters is to represent the data types and structures of the source language in terms of the low-level facilities provided by the machine (bits, pointers, etc.). In particular, for data structures that must be stored in memory, an appropriate *memory layout* must be determined and implemented. This layout determines the position of each component of a compound data structure, relative to the start address of the memory area representing this structure.

Many programming languages are implemented using straightforward memory layouts. In Fortran and C, for example, compound data structures are laid out consecutively in memory, enumerating their members in a conventional order such as column-major ordering for Fortran arrays or declaration order for C structures. Padding is inserted between the members as necessary to satisfy the alignment constraints of the machine. Higher-level languages leave more flexibility for determining data representations, but practical considerations generally result in simple memory layouts quite similar in principle to those of C structures, with the addition of tags and dynamic type information to guide the garbage collector and implement dynamic dispatch and type tests.

This paper focuses on data representation for objects in the C++ language. C++ combines the many scalar types and pointer types inherited from C with a rich object model, featuring multiple inheritance with both repeated and shared inheritance of base classes, object identity distinction, dynamic dispatch, and run-time type tests for some but not all classes. This combination raises interesting data representation challenges. On the one hand, the layout of objects must abide by the semantics of C++ as defined by the ISO standards [8]. On the other hand, this semantics leaves significant flexibility in the way objects are laid out in memory, flexibility that can be (and has repeatedly been) exploited to reduce the memory footprint of objects. A representative example is the "empty base optimization" described in section 2.

As a result of this tension, a number of optimized object layout algorithms have been proposed [6, 7, 13, 17], implemented in production compilers, and standardized as part of application binary interfaces (ABI) [3]. Section 2 outlines some of these algorithms and their evolution. These layout algorithms are quite complex, sometimes incorrect (see Myers [13] for examples), and often difficult to relate with the high-level requirements of the C++ specification. For example, the C++ "common vendor" ABI [3] devotes several pages to the specification of an object layout algorithm, which includes a dozen special cases.

The work reported in this paper provides a formal framework to specify C++ object layout algorithms and prove their correctness. As the high-level specification of operations over objects, we use the operational semantics for C++ multiple inheritance formalized by Wasserrab *et al* [19], which we have extended with structure fields and structure array fields (section 3). We then formalize a family of layout algorithms, independently of the target architecture, and axiomatize a number of conditions they must respect while leaving room for many optimizations (section 4). We prove that these conditions are sufficient to guarantee semantic preservation when the high-level operations over objects are reinterpreted as machine-level memory accesses and pointer manipulations (section 5). Finally, we formalize two realistic layout algorithms: one based on the popular "common vendor" C++ ABI [3], and its ex-

* Partially supported by NSF grants CCF-0702765 and CCF-1035058.

† Partially supported by ANR grant Arpège U3CAT.

POPL '11 January 26–28, Austin, TX, USA

tension with one further optimization; we prove their correctness by showing that they satisfy the sufficient conditions (section 6).

All the specifications and proofs in this paper have been mechanically verified using the Coq proof assistant. The Coq development is available online [15].

The contribution of this paper is twofold. On one hand, it is (to the best of our knowledge) the first formal proof of semantic correctness for realistic, optimizing C++ object layout algorithms, one of which being part of a widely used ABI. Moreover, we hope that large parts of our formalization and proofs can be reused for other, present or future layout algorithms. On the other hand, just like the subobject calculus of Rossie and Friedman [16] and the operational semantics for multiple inheritance of Wasserrab *et al* [19] were important first steps towards a formal specification of the semantics of (realistic subsets of) the C++ language, the work presented in this paper is a first step towards the formal verification of a compiler front-end for (realistic subsets of) C++, similar in principle and structure to earlier compiler verification efforts for other languages such as Java [9], C0 [10], and C [11].

2. Overview

2.1 The object layout problem

Generally speaking, an object layout algorithm is a systematic way to map an abstract, source-level view of objects down to machine-level memory accesses and pointer operations. At the source level, a C++ object is an abstract entity over which various primitive operations can be performed, such as accessing and updating a field, converting ("casting") an object or an object descriptor to another type, or dispatching a virtual function call. At the machine level, an object descriptor is a pointer p to a block of memory containing the current state of the object. The object layout scheme determines, at compile-time, how to reinterpret the source-level object operations as machine instructions:

- Accessing a field f defined in the class C of the object becomes a memory read or write at address $p + \delta$, where the constant offset δ is determined by the layout algorithm as a function of C and f.
- Dispatching a virtual function call is achieved by querying the dynamic type information attached to the object (typically, a virtual function table), a pointer to which is stored at a fixed offset relative to p.
- Converting to a base class (super-class) D of C is also achieved by adding an offset δ to the pointer p, making it point to the subobject of class D contained within. The offset δ can be statically determined in many cases, but owing to multiple virtual inheritance, it may have to be determined at run-time by querying the dynamic type information of the object.
- Accessing a field f defined in a base class D of C is achieved by converting p to D, then accessing f at a fixed offset from the resulting pointer.

For the generated machine operations to correctly implement the C++ semantics of object operations, the object layout scheme must satisfy a number of correctness conditions. First, when a scalar field is updated, the values of all other fields must be preserved. This immediately translates to the following requirement:

> *Field separation*: two distinct scalar fields, reachable through inheritance and/or fields from the same object, map to disjoint memory areas.

Moreover, the hardware architecture can impose *alignment constraints* on memory accesses: for example, loading a 32-bit integer is possible only from an address that is a multiple of 4. This leads to a second requirement:

> *Field alignment*: for any field f of scalar type t, the natural alignment of type t evenly divides its memory address.

Besides containing fields, C++ objects also have an *identity*, which can be observed by taking the address of an object and comparing it (using the == or != operators) with addresses of other objects of the same type. The C++ semantics specifies precisely the outcome of these comparisons, and this semantics must be preserved when the comparisons are reinterpreted as machine-level pointer comparisons:

> *Object identity*: two pointers to two distinct (sub)objects of the same static type A, obtained through conversions or accesses to structure fields, map to different memory addresses.

This requirement is further compounded by the fact that C++ operates under a simplistic separate compilation model inherited from C, and the fact that every class can be used independently to create complete objects, and every subobject in isolation is a potential target of most operations supported by any complete object of the same type.

Furthermore, some classes are considered to be *dynamic*: those classes that need dynamic type data to perform virtual function dispatch, dynamic cast, access to a virtual base, or other dynamic operations. The concrete representation for objects of these dynamic classes must include dynamic type data (usually as a pointer to a data structure describing the class), and such data must be preserved by field updates.

> *Dynamic type preservation*: any scalar field maps to a memory area that is disjoint from any memory area reserved to hold dynamic type data.

The separation conditions between two areas holding dynamic type data are weaker than the separation conditions for fields. Indeed, most layout algorithms arrange that the dynamic type data for a class C is shared with that of one of its dynamic non-virtual direct bases, called the *non-virtual primary base* of C and chosen during layout, in a way that preserves the semantics of virtual function dispatch and other dynamic operations.

> *Dynamic type data separation*: for any two dynamic classes, the memory areas of their respective dynamic type data must be disjoint, unless one of the classes is the non-virtual primary base of the other.

2.2 The historic layout scheme

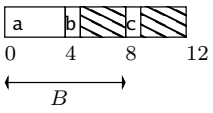
Early C++ compilers such as Cfront (Stroustrup's C++-to-C translator) used a simple object layout algorithm that obviously satisfies the requirements listed above. In effect, all components of a class (fields, base classes, and dynamic data if needed) are laid out contiguously following a conventional order, inserting padding (unused bytes) between components and after the last component as necessary to satisfy alignment constraints. In this approach, each component (base or field) of a class C has its own memory area, disjoint from those of all other components of C. Moreover, each class is represented by at least one byte of data, even if it contains no fields nor dynamic type data. This trivially enforces the object identity requirement.

2.3 Tail padding optimization

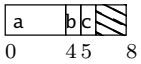
The simple layout scheme just described is inefficient in space, because it never attempts to reuse padding to store useful data. Consider:

```
struct A     { int a; };
struct B : A { char b; };
struct C : B { char c; };
```

Assume the size and natural alignment of type **int** are 4 bytes. To ensure proper alignment of field a, 3 bytes of padding are inserted after field b in B, and 3 more bytes of padding are inserted after field c in C. (To understand why, consider arrays whose elements have type B or C.) Consequently, B has size 8 while C has size 12.



However, it is perfectly legitimate to reuse one of the 3 bytes of padding present at the end of B to hold the value of field c of C. In this case, C occupies only 8 bytes, for a nice space saving of $1/3$. This layout trick is known as the *tail padding optimization*, and is used by many C++ compilers.



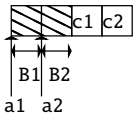
2.4 Empty base class optimization

Empty classes offer another opportunity for reusing space that is unused in a base class. Consider:

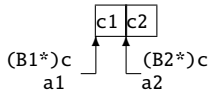
```
struct A            {};
struct B1 : A       {};
struct B2 : A       {};
struct C  : B1, B2 { char c1; char c2; };

C   c;
A * a1 = (A *) (B1 *) &c;
A * a2 = (A *) (B2 *) &c;
```

Here, the classes A, B1 and B2 are *empty*: they contain no field and need no dynamic type information. It is tempting to give them size 0, so that their instances occupy no memory space at all. However, this would violate the "object identity" requirement: as depicted to the right, the pointers a1 and a2 (to the two instances of A logically contained within C) would compare equal, while C++'s semantics mandate that they are different.

The layout algorithm must therefore insert (at least) one byte of padding in A, resulting in A, B1 and B2 having size 1. Following the naive approach outlined in section 2.2, C therefore occupies 4 bytes.



However, it is unnecessary to keep the fields c1 and c2 disjoint from the subobjects of types B1 and B2: the padding inserted in the latter to satisfy the object identity requirement can be reused to hold the values of fields c1 and c2, as shown to the right.



This technique is known as the *empty base class optimization*. It is implemented by many C++ compilers, but often in restricted cases only. Here is an example where GCC 4.3 misses an opportunity for reusing the tail padding of an empty base class.

```
struct A            {};
struct B1 : A       {};
struct B2 : A       {};
struct C  : B1, B2 { char c; };
struct D  : C       { char d; };
```

As in the previous example, B1 and B2 must be laid out at different offsets within C, to allow distinguishing the two A contained in C. Thus, C must have size at least 2. This lower bound can be achieved by placing c at offset 0, as explained above.

What about D? GCC places d at offset 2, resulting in a size of 3 for D. However, the second byte of C is just padding introduced to preserve the identity of empty base classes, and it can be reused to hold data such as field d. This optimized layout fits D in just two bytes.

Why empty classes matter Over the years, successful C++ software, such as the Standard Template Libraries (STL), has become dependent on C++'s ability to deliver efficient code based on simple techniques such as empty classes and inlining. Part of the success of the STL is based on its archetypical use of function objects: these are objects of classes with the function call operator overloaded. These objects typically do not carry any runtime data. Rather, their semantics is their static types. As an example, consider sorting an array of integers. The STL provides the following template:

```
template<typename Ran, typename Comp>
  void sort(Ran first, Ran last, Comp cmp);
```

The comparator cmp could be a pointer to function (like the qsort function in C), but in idiomatic C++ it is any object whose type overloads the function call operator.

```
struct MyGreater {
   typedef int first_argument_type;
   typedef int second_argument_type;
   typedef bool result_type;
   bool operator()(int i, int j) const {
      return i > j;
   }
};
```

The sort template can, then, be invoked as sort(t, t + n, MyGreater()). The comparator object constructed by MyGreater() is of interest not for its runtime data (it carries none), but for its type: the comparison function is not passed to the sorting routine through data, but through the type of the object. Consequently, it is directly called, and in most cases inlined since the body is a simple integer comparison, which typically reduces to a single machine instruction. This simple technique, a cornerstone of the STL success, is effective. One can think of it as a simulation of dependent types, where data is encoded at the level of types, therefore making data flow obvious.

The function object technique just described is at the basis of a composable component of the STL. Composition implies a protocol that parts being composed should adhere to. For example, if we want to combine two unary function objects, we need a mechanism to ensure that the result type of one function object agrees with the argument type of the other. To that end, the STL requires the existence of certain nested types, such as first_argument_type, second_argument_type and result_type in the MyGreater class above. To reduce clutter, the STL provides ready-to-use classes that define those types. In this case, idiomatically, one would write

```
struct MyGreater :
  std::binary_function<int,int,bool> {
    bool operator()(int i, int j) const {
       return i > j;
    }
};
```

The sole purpose of std::binary_function<int,int,bool> is to provide those nested types. It is an empty base class that introduces no data and no virtual functions. Its semantics is purely static. This usage, while not object-oriented programming by most popular definitions, is very common in modern C++ programs.

The pattern is not restricted to only one empty base class: a class can simulatenously inherit from several empty base classes. This is the case of bidirectional_input_iterator_tag, which inherits both from forward_iterator_tag and output_iterator_tag. A compiler that effectively supports C++ should not have to allocate space for such empty classes since their runtime semantics are completely irrelevant. While an optimization in C++03, the empty base optimization is considered so important that it is required in the next version dubbed C++0x [4].

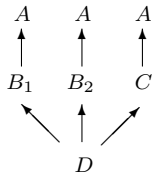
3. Semantic model

To capture the expected behavior of C++ object operations, we build on the model of multiple inheritance introduced by Rossie and Friedman [16] and further developed and mechanized by Wasserrab *et al.* [19] using the Isabelle proof assistant. In this section, we briefly recall the main aspects of this model, referring the reader to [19] for full details, then describe how we extended it to handle fields that are themselves structures or arrays of structures, and finally use the model to given an operational semantics to a simple calculus of C++ objects.

3.1 Modeling multiple inheritance

A C++ class can inherit from several base classes. Consequently, a class D can inherit from another class A through several different ways. Consider for example:

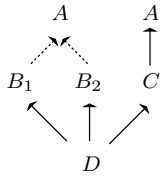
```
struct A     { int a; };
struct B1: A          {};
struct B2: A          {};
struct C : A          {};
struct D : B1, B2, C {};
```



An instance of `D` contains as many copies of `A` as ways to inherit. Each copy of `A` is called a *subobject* of `D`, of static type `A`. In this example, `D` has three different subobjects of static type `A`, obtained through the base classes `B1`, `B2` and `C`. This is called *non-virtual inheritance* by Stroustrup [5], *replicated inheritance* by Rossie and Friedman [16], or *repeated inheritance* by Wasserrab *et al* [19].

However, the programmer can elect to declare some base classes as *virtual*, as in the following example:

```
struct A     { int a; };
struct B1: virtual A {};
struct B2: virtual A {};
struct C : A         {};
struct D : B1, B2, C {};
```



In this case, `B1` and `B2` contribute only one subobject of static type `A` to class `D`: the paths `B1::A` and `B2::A` designate the same subobject. This is called *virtual inheritance* [5] or *shared inheritance* [19]. Note, however, that `C` contributes another, distinct subobject `A` to `D`, since `C` inherits from `A` in a non-virtual manner.

Following Wasserrab *et al* [19], we capture this notion of subobject and these two flavors of inheritance as follows. A base class subobject of a given object is represented by a pair $\sigma = (h, l)$ where h is either $\mathsf{Repeated}$ or Shared, and l is a path in the non-virtual inheritance graph (the directed graph having classes as nodes and edges $U \to V$ if and only if V is a direct non-virtual base of U). More formally, we write $C \prec\langle\sigma\rangle^{\mathcal{I}}_{\hookrightarrow} A$ to mean that σ designates a base class subobject of class C, this subobject having static type A. This predicate is defined inductively:

$$C \prec\langle(\mathsf{Repeated}, C :: \epsilon)\rangle^{\mathcal{I}}_{\hookrightarrow} C$$

$$\frac{B \text{ direct non-virtual base of } C \qquad B \prec\langle(\mathsf{Repeated}, l)\rangle^{\mathcal{I}}_{\hookrightarrow} A}{C \prec\langle(\mathsf{Repeated}, C :: l)\rangle^{\mathcal{I}}_{\hookrightarrow} A}$$

$$\frac{B \text{ virtual base of } C \qquad B \prec\langle(h, l)\rangle^{\mathcal{I}}_{\hookrightarrow} A}{C \prec\langle(\mathsf{Shared}, l)\rangle^{\mathcal{I}}_{\hookrightarrow} A}$$

The fields of a full instance of a class C, then, are the pairs (σ, f) where $C \prec\langle\sigma\rangle^{\mathcal{I}}_{\hookrightarrow} A$ and f is a field defined in class A. In the first example above, the fields of `D` are $\{((\mathsf{Repeated}, \texttt{D} :: \texttt{B1} :: \texttt{A} :: \epsilon), \texttt{a});\ ((\mathsf{Repeated}, \texttt{D} :: \texttt{B2} :: \texttt{A} :: \epsilon), \texttt{a});\ ((\mathsf{Repeated}, \texttt{D} :: \texttt{C} :: \texttt{A} :: \epsilon), \texttt{a})\}$, while in the second example (involving virtual inheritance), they are $\{((\mathsf{Shared}, \texttt{A} :: \epsilon), \texttt{a});\ ((\mathsf{Repeated}, \texttt{D} :: \texttt{C} :: \texttt{A} :: \epsilon), \texttt{a})\}$.

3.2 Structure fields and structure array fields

In the original formalization of Wasserrab *et al* [19], fields are restricted to scalar types (arithmetic types or pointer types). We extended this formalization to support fields that are themselves structures or arrays of structures. To simplify presentation, we only consider arrays of structures, treating a structure field or variable `C x;` as a one-element array `C x[1];`.

In our formalization, a full object is always an element of a (possibly one-element) array of structures, which is either bound to a program variable, or dynamically created using **`new`**, or appears as a structure array field of a larger object. To designate a subobject of an array of structures, we therefore proceed in three steps:

1. select an array of structures contained in this array;
2. select an element of this sub-array;
3. select a base class subobject of this element (in the sense of section 3.1).

To select an array of structures contained in a larger array, we introduce the notion of an *array path* α from an array of n structures of type C to an array of n' structures of type C', written $C[n] \prec\langle\alpha\rangle^{\mathcal{A}}_{\hookrightarrow} C'[n']$. Array paths are defined inductively by the following rules:

$$\frac{i \le n \qquad C \prec\langle\sigma\rangle^{\mathcal{I}}_{\hookrightarrow} A}{C[n] \prec\langle i, \sigma\rangle^{\mathcal{CI}}_{\hookrightarrow} A} \qquad \frac{n' \le n}{C[n] \prec\langle\epsilon\rangle^{\mathcal{A}}_{\hookrightarrow} C[n']}$$

$$\frac{F = (f, D, m) \text{ is a structure field defined in } A \qquad C[n] \prec\langle i, \sigma\rangle^{\mathcal{CI}}_{\hookrightarrow} A \qquad D[m] \prec\langle\alpha\rangle^{\mathcal{A}}_{\hookrightarrow} C'[n']}{C[n] \prec\langle(i, \sigma, F) :: \alpha\rangle^{\mathcal{A}}_{\hookrightarrow} C'[n']}$$

Then, a *generalized subobject* of type A of a full object of type $C[n]$ is a triple (α, i, σ) where α is an array path from $C[n]$ to some $C'[n']$ and $i \in [0, n')$ is an index in the array of type $C'[n']$ and σ is a base class subobject of C' of type A. We write $C[n] \prec\langle(\alpha, i, \sigma)\rangle\!\rightarrow A$ and formally define this relation by:

$$\frac{C\,[n] \prec\langle\alpha\rangle^{\mathcal{A}}_{\hookrightarrow} C'\,[n'] \prec\langle i, \sigma\rangle^{\mathcal{CI}}_{\hookrightarrow} A}{C\,[n] \prec\langle(\alpha, i, \sigma)\rangle\!\rightarrow\ A}$$

Consider for example:

```
struct Z              {};
struct A : Z          {};
struct B              { A a[4]; };
struct C1: virtual B {};
struct C2: virtual B {};
struct D : C1, C2     {};
struct E              { D d[5]; };
E e[7];
```

The expression `(Z *) &(e[2].d[0].a[3])` denotes the generalized subobject (α, i, σ) within `e`, where

$$\alpha = (2, (\mathsf{Repeated}, \texttt{E} :: \epsilon), \texttt{d}) :: (0, (\mathsf{Shared}, \texttt{B} :: \epsilon), \texttt{a}) :: \epsilon$$
$$i = 3$$
$$\sigma = (\mathsf{Repeated}, \texttt{A} :: \texttt{Z} :: \epsilon)$$

The static type of a generalized subobject (α, i, σ) is the static type of the base class subobject σ. This generalized subobject denotes a full instance of class C if and only if σ is the trivial subobject of C of static type C, that is, $\sigma = (\mathsf{Repeated}, C :: \epsilon)$.

3.3 An operational semantics

Building on these notions of subobjects, we now give an operational semantics for a small 3-address intermediate language featuring a C++-style object model. The syntax of this language is as follows. (x ranges over variable names.)

$Stmt ::= x := op(x_1, \ldots, x_n)$	arithmetic
$\mid x := \texttt{nullptr}$	null pointer
$\mid x := x'\texttt{->}_C field$	field access
$\mid x\texttt{->}_C field := x'$	field assignment
$\mid x := \&x_1[x_2]_C$	array indexing
$\mid x := x_1 == x_2$	pointer equality test
$\mid x := \texttt{static\_cast}\langle C'\rangle_C(x_1)$	static cast
$\mid x := \texttt{dynamic\_cast}\langle C'\rangle_C(x_1)$	dynamic cast

Object operations are annotated with static types C (subscripted) as determined during type-checking. For instance, in field operations, C is the class that defines the field being accessed; in conversions, C is the static type of the argument of the conversion, and C' is the destination type. Our Coq formalization also includes some control structures (sequence, conditional, loops), which we omit for simplicity. Also omitted are virtual function calls, object creation and object destruction, which we have not fully formalized yet.

The operational semantics uses the following semantic objects:

$v ::= \mathsf{Atom}\ baseval$	base value
$\mid \mathsf{Ptr}\ \varnothing$	null pointer
$\mid \mathsf{Ptr}\ (objid, \alpha, n, \sigma)$	pointer to a subobject
$V ::= x \mapsto v$	variable environment
$\omega ::= (\alpha, n, \sigma, (f, t)) \mapsto v$	values of scalar fields
$H ::= objid \mapsto (C, n, \omega)$	object heap

The state of the execution is a pair (V, H) of an environment V mapping variable names to scalar values (either base values or pointers), and a heap H mapping top-level object identifiers $objid$ to their type $C[n]$ and their contents ω. The contents ω is, in turn, a mapping from scalar field designators (pairs of a generalized subobject (α, n, σ) and a field (f, t)) to scalar values.

The operational semantics is given as a transition relation $Stmt \vdash (V, H) \to (V', H')$ relating the state (V, H) before and the state (V', H') after executing $Stmt$. We show some representative rules.

$$\frac{V(x') = \mathsf{Ptr}(o, \alpha, i, \sigma) \qquad H(o) = (C, n, \omega) \\ C[n] \dashv\langle(\alpha, i, \sigma)\rangle\!\rightarrowtail A \qquad F \text{ is a scalar field of } A}{x := x'\texttt{->}_A F \vdash (V, H) \to (V\{x \leftarrow \omega(\alpha, i, \sigma, F)\}, H)}$$

$$\frac{V(x) = \mathsf{Ptr}(o, \alpha, i, \sigma) \\ H(o) = (C, n, \omega) \qquad C[n] \dashv\langle(\alpha, i, \sigma)\rangle\!\rightarrowtail A \\ F \text{ is a scalar field of } A \qquad \omega' = \omega\{(\alpha, i, \sigma, F) \leftarrow V(x')\}}{x\texttt{->}_A F := x' \vdash (V, H) \to (V, H\{o \leftarrow (C, n, \omega')\})}$$

$$\frac{V(x') = \mathsf{Ptr}(o, \alpha, i, \sigma) \qquad H(o) = (C, n, \omega) \\ C[n] \dashv\langle(\alpha, i, \sigma)\rangle\!\rightarrowtail A \qquad F = (f, A', n') \text{ is an array field of } A \\ p' = (o, \alpha{+\!\!+}(i, \sigma, F) :: \epsilon, 0, (\mathsf{Repeated}, A' :: \epsilon))}{x := x'\texttt{->}_A F \vdash (V, H) \to (V\{x \leftarrow \mathsf{Ptr}\ p'\}, H)}$$

$$\frac{V(x_1) = \mathsf{Ptr}(o, \alpha, i, (\mathsf{Repeated}, A :: \epsilon)) \\ H(o) = (A', n', \omega) \\ A'\,[n'] \dashv\langle\alpha\rangle\!\rightarrowtail^{\mathcal{A}} A\,[n] \qquad 0 \le i < n \qquad V(x_2) = \mathsf{Atom}\ j \\ 0 \le i + j < n \qquad p' = (o, \alpha, i + j, (\mathsf{Repeated}, A :: \epsilon))}{x := \&x_1\,[x_2] \vdash (V, H) \to (V\{x \leftarrow \mathsf{Ptr}\ p'\}, H)}$$

For reads and writes over scalar fields (first two rules above), the content map ω of the top-level object o being accessed is consulted or updated at the path (α, i, σ) to the subobject and the accessed field. In contrast, accessing a structure array field (third rule) and addressing an array element (fourth rule) just synthesize the appropriate subobject from that given in the object pointer. In the case of a structure array field, the given subobject (α, i, σ) is refined into the sub-subobject $(\alpha{+\!\!+}(i, \sigma, F) :: \epsilon, 0, (\mathsf{Repeated}, A' :: \epsilon))$ designating the structure array. ($+\!\!+$ stands for list concatenation.) For array addressing, only the i part of the given subobject is modified. By lack of space, we omit the rules for static and dynamic casts, which are complicated but similar to those given by Wasserrab *et al* [19].

4. Formalization of a family of layout algorithms

In this section, we formalize the interface to a family of layout algorithms and sufficient conditions ensuring that they are semantically correct.

4.1 Platform-dependent parameters

Our formalization is independent of the characteristics of the hardware platform. It is parameterized by a set of scalar types, comprising arithmetic types (e.g. **`int`**, **`short`**, **`char`**, **`float`**, **`double`**, etc.) and pointer types. For each scalar type t, we take as parameters its size scsize_t and its natural alignment $\mathsf{scalign}_t$. The only assumptions we make about size and alignment is that they are positive and that all pointer types have the same size and alignment. The unit of size is not even specified: a natural choice is bytes, but bits could be used as well. Likewise, natural alignments are not required to be powers of 2.

As additional parameters, we also assume given a positive size $\mathsf{dtdsize}$ and a positive natural alignment $\mathsf{dtdalign}$ for dynamic type data. For simplicity, we assume that dynamic type data has the same size for all classes. This was not always the case in early compilers [17], but nowadays the trend is to store only a pointer to a class descriptor, thus incurring a constant access overhead (one or two more indirections) but significant improvements on the size of objects.

4.2 Interface of a layout algorithm

Let C be a class. We introduce the following notations:

- $\mathsf{dnvbases}_C$ is the set of direct non-virtual bases of C.
- vbases_C is the set of (direct or indirect) virtual bases of C.
- $\mathsf{scfields}_C$ is the set of scalar fields declared in C. Each such field is of the form (f, t) where f is an identifier and t is a scalar type.
- $\mathsf{stfields}_C$ is the set of structure array fields declared in C. Each such field is of the form (f, B, n) where f is an identifier, B is the structure type of the field, and $n > 0$ is the number of elements in the array.
- $\mathsf{fields}_C = \mathsf{scfields}_C \cup \mathsf{stfields}_C$ is the set of all fields declared in C.

For every class C in the program, a layout algorithm is expected to compute each of the following (as depicted in figure 1):

- pbase_C is the primary base of C, if any. More precisely, either $\mathsf{pbase}_C = \varnothing$, or $\mathsf{pbase}_C = \{B\}$ for some direct non-virtual base B of C.
- $\mathsf{dnvboff}_C : \mathsf{dnvbases}_C \to \mathbb{N}$ assigns offsets to the non-virtual direct bases of C.
- $\mathsf{fboundary}_C \in \mathbb{N}$ is an offset within C representing the boundary between non-virtual base data and field data.
- $\mathsf{foff}_C : \mathsf{fields}_C \to \mathbb{N}$ assigns offsets to the fields declared in C.
- $\mathsf{nvdsize}_C \in \mathbb{N}$ is the non-virtual data size of C: the data size of its non-virtual part.
- $\mathsf{nvsize}_C \in \mathbb{N}$ is the non-virtual size of C: the total size of its non-virtual part.

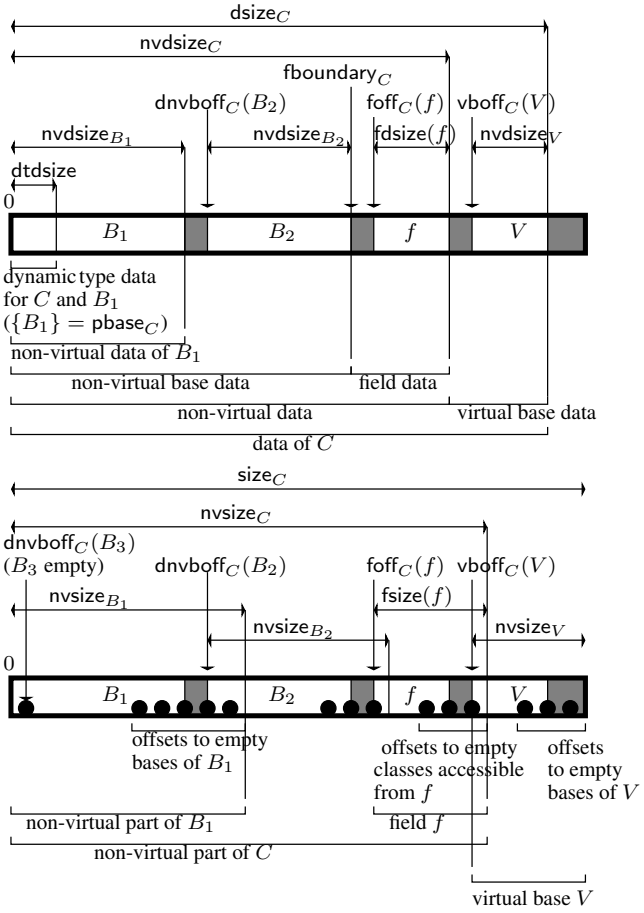


Figure 1. A proposed layout for a full instance of a dynamic class C having a primary non-virtual base B_1, a non-empty non-virtual base B_2, an empty non-virtual base B_3, a field f and a virtual base V. Grayed areas represent padding. Bullets represent offsets to (inheritance or field) subobjects of C of empty types. The top part depicts the layout parameters related to data; the bottom part, those related to object identity.

- $\mathsf{vboff}_C : \mathsf{vbases}_C \to \mathbb{N}$ gives the offsets of the virtual bases of C.
- $\mathsf{dsize}_C \in \mathbb{N}$ is the total data size of C.
- $\mathsf{size}_C \in \mathbb{N}$ is the total size of C. This is the quantity that **`sizeof`**(C) evaluates to.

For the purpose of our formalization, C can be considered as a virtual base of itself. Thus we introduce the notion of a class B being a *generalized virtual base* of C: either $B = C$, or B is a direct or indirect virtual base of C. Thanks to this notion, we can state that the layout of a full instance of C is composed of the layout of the non-virtual parts of the generalized virtual bases of C for this instance, with the non-virtual part of C starting at offset 0. Thus, we may safely extend the domain of vboff_C by taking $\mathsf{vboff}_C(C) = 0$.

4.3 Empty classes and dynamic classes

Our formalization leaves partially specified the notions of empty classes and dynamic classes, leaving their exact definitions to the layout algorithm, subject to the following conditions:

- An empty class must not contain any scalar field.
- An empty class must not contain any structure field of a non-empty class type.
- An empty class must not have any non-empty base.
- A dynamic class is not empty.
- If a class C has a virtual base, then C is dynamic
- If a class C has a non-virtual primary base B, then both C and B are dynamic.

In our formalization, the data size and the non-virtual data size of a class are of interest only if the class is not empty. Our way of classifying empty classes and dynamic classes can be understood as follows: the data of a class is composed of all its reachable scalar fields and dynamic type data; an empty class is a class that requires no data; and a dynamic class is a class that requires dynamic type data.

4.4 Computing offsets and compiling object operations

Once the quantities listed in section 4.2 have been computed by a layout algorithm, all the offsets mentioned in section 2.1 and required during compilation can be computed as follows. The offset $\mathsf{nvsoff}(C :: l)$ of a non-virtual, base class subobject $(\mathsf{Repeated}, C :: l)$ within the low-level representation of a subobject of static type C is:

$$\mathsf{nvsoff}(C :: \epsilon) = 0$$
$$\mathsf{nvsoff}(C :: B :: l') = \mathsf{dnvboff}_C(B) + \mathsf{nvsoff}(B :: l')$$

The offset $\mathsf{soff}_C(h, l)$ of the base class subobject (h, l) of C within the low-level representation of a full instance of C is computed as follows:

$$\mathsf{soff}_C(\mathsf{Repeated}, C :: l') = \mathsf{nvoff}(C :: l')$$
$$\mathsf{soff}_C(\mathsf{Shared}, B :: l') = \mathsf{vboff}_C(B) + \mathsf{nvsoff}(B :: l')$$

If l is an array path from $C[n]$ to $C'[n']$, then the offset $\mathsf{aoff}_C(l)$ of (the first element of) the array of n' elements of type C' designated by l within the low-level representation of an array of n elements of type C is determined by $\mathsf{aoff}_C(\epsilon) = 0$ and

$$\mathsf{aoff}_C((i, s, (f, A, m)) :: \alpha)$$
$$= i \cdot \mathsf{size}_C + \mathsf{soff}_C(s) + \mathsf{foff}_B(f, A, m) + \mathsf{aoff}_A(\alpha)$$
(where B is the static type of s)

Finally, the offset $\mathsf{off}_C(\alpha, i, s)$ of a generalized subobject (α, i, s) within an array of structures of type C is computed as follows:

$$\mathsf{off}_C(\alpha, i, s) = \mathsf{aoff}_C(\alpha) + i \cdot \mathsf{size}_B + \mathsf{soff}_B(s)$$
(where $B[m]$ is the destination type of α)

Using these offsets, we now outline a compilation scheme for the C++-like intermediate language of section 3.3. The target language is a conventional low-level intermediate language, featuring a flat, byte-addressed memory, pointer arithmetic, and explicit $\mathtt{load}(n, p)$ and $\mathtt{store}(n, p, x)$ operations to read and write n-byte quantities at address p. For field and array accesses, we have:

$$[\![x := x'\texttt{->}_C F]\!] = x := \mathtt{load}(\mathsf{scsize}_t, x' + \mathsf{foff}_C(F))$$
(if $F = (f, t)$ is a scalar field of C)
$$[\![x\texttt{->}_C F := x']\!] = \mathtt{store}(\mathsf{scsize}_t, x + \mathsf{foff}_C(F), x')$$
(if $F = (f, t)$ is a scalar field of C)
$$[\![x := x'\texttt{->}_C F]\!] = x := x' + \mathsf{foff}_C(F)$$
(if F is a structure array field of C)
$$[\![x := \&x_1[x_2]_C]\!] = x := x_1 + \mathsf{size}_C \times x_2$$
$$[\![x := x_1 == x_2]\!] = x := x_1 == x_2$$

For static casts, we have two cases depending on whether inheritance is virtual or not. If $C \dashv\!\langle(\mathsf{Repeated}, l)\rangle\!\xrightarrow{\mathcal{I}} A$ for a uniquely-

defined path $(\mathsf{Repeated}, l)$ (conversion to or from a non-virtual base), the static cast is achieved by adjusting the pointer by a constant offset:

$$[\![x := \texttt{static\_cast}\langle A\rangle_C(x')]\!] = x := x' + \mathsf{nvsoff}(l)$$
$$[\![x := \texttt{static\_cast}\langle C\rangle_A(x')]\!] = x := x' - \mathsf{nvsoff}(l)$$

If $C \dashv\langle(\mathsf{Shared}, B :: l)\rangle^{\mathcal{I}}\!\!\rightarrow A$ and $(\mathsf{Shared}, B :: l)$ is unique (conversion through a virtual base), the offset of the virtual base B of C must be looked up in the dynamic type data:

$$[\![x := \texttt{static\_cast}\langle A\rangle_C(x')]\!] =$$
$$t := \texttt{load}(\mathsf{dtdsize}, x'); x := x' + \texttt{vboff}(t, B) + \mathsf{nvsoff}(l)$$

Finally, the code $[\![x := \texttt{dynamic\_cast}\langle A\rangle_C(x')]\!]$ for a dynamic cast is:

$$t := \texttt{load}(\mathsf{dtdsize}, x');$$
$$\textbf{if}\ (\texttt{dyncastdef}(t, A))\ \ \texttt{then}\ x := x' + \texttt{dyncastoff}(t, A)$$
$$\textbf{else}\ x := \texttt{NULL}$$

As shown in the last two cases, the target language features three operations over the run-time representation t of dynamic type data: $\texttt{vboff}(t, B)$ returns the offset appropriate for virtual base B; $\texttt{dyncastdef}(t, A)$ returns 1 if a dynamic cast to A is possible, 0 otherwise; and $\texttt{dyncastoff}(t, A)$ returns the offset appropriate for a dynamic cast to A.

In the semantics of our target language, we formalize these operations at an abstract level through queries to an "oracle". The actual concrete implementation (virtual table, dictionary, etc.) is left unspecified. However, we have formally proved that such an oracle can always be constructed from a well-founded class hierarchy.

4.5 Soundness conditions

We now state a number of soundness conditions over the results of the layout algorithm. Section 5 shows that these conditions are sufficient to guarantee semantic preservation.

Sizes The first set of conditions ensures that the total non-virtual size and total size of a class C are large enough to enclose all corresponding components of C.

(C1) $\mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B \leq \mathsf{nvsize}_C$ if B direct non-virtual base of C

(C2) $\mathsf{foff}_C(f) + \mathsf{fsize}(f) \leq \mathsf{nvsize}_C$ if f field of C

(C3) $\mathsf{vboff}_C(B) + \mathsf{nvsize}_B \leq \mathsf{size}_C$ if B generalized virtual base of C

(C4) $\mathsf{dsize}_C \leq \mathsf{size}_C$

(C5) $0 < \mathsf{nvsize}_C$

The first two conditions ensure that the total non-virtual size of C is large enough to hold the non-virtual part of any non-virtual base of C (C1) as well as any field of C (C2). We write $\mathsf{fsize}(f)$ for the size of the field f, computed as follows:

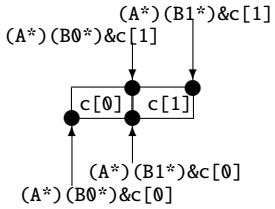
$$\mathsf{fsize}(f) = \mathsf{scsize}_t \qquad \text{if } f \text{ has scalar type } t$$
$$\mathsf{fsize}(f) = n \cdot \mathsf{size}_B \qquad \text{if } f \text{ has array type } B[n]$$

Likewise, the total size of C must be large enough to hold a full instance of C, including any virtual base of C (C3), the non-virtual part of C itself (C3 again), and all data of C excluding its tail-padding (C4).

By contrast, it is not required that $\mathsf{nvdsize}_C \leq \mathsf{nvsize}_C$. More precisely, the non-virtual size of a class is an upper bound of the interval in which offsets to fields or subobjects within the non-virtual part of the class must appear. The non-virtual size plays no role in the correctness of data access. However, (C4) is used to show that the data of two disjoint cells of a structure array (or of two cells of distinct structure arrays) are disjoint, which guarantees that data access within an object does not affect other objects. Conversely, the non-virtual data size, and data sizes in general, play no role in the correctness of subobject identification.

The C++ standard [8] mandates that the total size of a class be positive. This is necessary to ensure that distinct elements of an array of structures have different offsets. We require a stronger condition, imposing that the total *non-virtual size* of a class be positive (C5). This is necessary to preserve object identity, as the following example demonstrates.

```
struct A           {};
struct B0: A       {};
struct B1: A       {};
struct C : B0, B1 {};
C[2] c;
A * a0 = (A*) (B1*) &c[0];
A * a1 = (A*) (B0*) &c[1];
```



If the non-virtual size of `A` were 0, then (C1) would allow the non-virtual sizes of `B0` and `B1` to be 0, so that the non-virtual size of `C` could be 1. Thus, (C3) would allow the size of `C` to be 1. But in that case, assuming `B0` at offset 0 and `B1` at offset 1 in `C`, the pointers `a0` and `a1` would compare equal, even though they refer to logically different subobjects.

Field separation We now present a set of conditions sufficient to guarantee separation between scalar fields and, therefore, the "good variable" property (updating a field preserves the values of all other fields). These conditions are pleasantly local: they only involve the fields defined in the class C under consideration, as well as the layout quantities for its direct bases, but do not need to consider inherited fields nor indirect bases. They focus on *relevant* components of the class C under consideration. A base class is not relevant if, and only if, it is of an empty class type. A field is not relevant if, and only if, it is an array of structures of an empty type. We write $\mathcal{S} \# \mathcal{T}$ to say that two sets are disjoint ($\mathcal{S} \cap \mathcal{T} = \varnothing$).

(C6) $[\mathsf{dnvboff}_C(B_1), \mathsf{dnvboff}_C(B_1) + \mathsf{nvdsize}_{B_1}) \# [\mathsf{dnvboff}_C(B_2), \mathsf{dnvboff}_C(B_2) + \mathsf{nvdsize}_{B_2})$ if B_1, B_2 distinct non-empty non-virtual direct bases of C

(C7) $\mathsf{dnvboff}_C(B) + \mathsf{nvdsize}_B \leq \mathsf{fboundary}_C$ if B non-empty non-virtual direct base of C

(C8) $\mathsf{fboundary}_C \leq \mathsf{foff}_C(f)$ if f relevant field of C

(C9) $[\mathsf{foff}_C(f_1), \mathsf{foff}_C(f_1) + \mathsf{fdsize}(f_1)) \# [\mathsf{foff}_C(f_2), \mathsf{foff}_C(f_2) + \mathsf{fdsize}(f_2))$ if f_1 and f_2 are distinct relevant fields of C

(C10) $\mathsf{foff}_C(f) + \mathsf{fdsize}(f) \leq \mathsf{nvdsize}_C$ if f relevant field of C

(C11) $\mathsf{fboundary}_C \leq \mathsf{nvdsize}_C$

(C12) $[\mathsf{vboff}_C(B_1), \mathsf{vboff}_C(B_1) + \mathsf{nvdsize}_{B_1}) \# [\mathsf{vboff}_C(B_2), \mathsf{vboff}_C(B_2) + \mathsf{nvdsize}_{B_2})$ if B_1, B_2 distinct non-empty generalized virtual bases of C

(C13) $\mathsf{vboff}_C(B) + \mathsf{nvdsize}_B \leq \mathsf{dsize}_C$

Condition (C6) states the absence of overlap between the data of two distinct, *non-empty*, non-virtual direct bases of a class. To separate the data of direct non-virtual bases of C from the data of the fields of C, our formalization introduces a *boundary* $\mathsf{fboundary}_C$ such that every *non-empty* direct non-virtual base of C has its data laid out below this boundary (C7), and every *relevant* field f of C is laid out above this boundary (C8). It is however possible for the tail padding of a direct non-virtual base of C to straddle the data, or even the tail padding, of a field of C.

Define the *data size* $\mathsf{fdsize}(f)$ of a relevant field f as:

$$\mathsf{fdsize}(f) = \mathsf{scsize}_t \qquad \text{if } f \text{ has scalar type } t$$
$$\mathsf{fdsize}(f) = (n-1) \cdot \mathsf{size}_B + \mathsf{dsize}_B \qquad \text{if } f \text{ has array type } B[n]$$

In other words, the tail padding of a structure array field is that of the last element of the array, while the tail padding of the first $n-1$ elements is part of the data area for this field (so that this data area is contiguous in memory). Then, condition (C9) states that the *data* areas of two distinct *relevant* fields f_1 and f_2 of C are disjoint. Note, however, that the tail padding of a field of C can be reused to hold some of the next fields of C.

The non-virtual data size of C, like the non-virtual size and the size of C, is not computed directly. Instead, it is constrained as follows: the data of any relevant field f of C is included in the non-virtual data of C (C10), and the non-virtual data of any direct non-virtual bases of C is embedded in the non-virtual data of C (C11). This condition is redundant if C has at least one relevant field.

Finally, as regards virtual inheritance, two distinct, non-empty, generalized virtual bases B_1 and B_2 of C are laid out in such a way that they do not overlap (C12). Recall that a generalized virtual base of C is C itself or a direct or indirect virtual base of C. Then, if B is a non-empty virtual base of C, then C is not empty (as it has a non-empty base), so the above condition holds for generalized virtual bases B and C, which guarantees that the data of the non-virtual part of the virtual base B does not overlap the data of the non-virtual part of C.

Finally, the data of C (excluding its tail-padding) contains the non-virtual data of any generalized virtual base B of C (C13). By contrast, since neither any scalar value nor any virtual function can be accessed from an irrelevant component, the data of an irrelevant component need not be disjoint from other fields or bases of the same type. Likewise, it is not required that an irrelevant field starts at the field boundary: it may also straddle non-virtual base data.

Field alignment The next set of conditions ensures that every field is laid out at an offset that is naturally aligned with respect to its type, even if the field appears in a base class or an array. Define the natural alignment $\mathsf{falign}(f)$ of a field f as $\mathsf{falign}(f) = \mathsf{scalign}_t$ if f has scalar type t, and $\mathsf{falign}(f) = \mathsf{align}_B$ if f has type $B[n]$.

Besides its alignment align_C, a class C has a *non-virtual alignment* $\mathsf{nvalign}_C$ used whenever C is laid out as a base of another class. This distinction allows the non-virtual part of C to be laid out under weaker alignment constraints than for a full instance of C, especially if the alignment of a full instance of C is constrained by that of a virtual base.

The conditions related to alignment are as follows. We write $p \mid q$ to mean that p evenly divides q.

(C14) $(\mathsf{falign}(f) \mid \mathsf{foff}_C(f))$ and $(\mathsf{falign}(f) \mid \mathsf{nvalign}_C)$ if f field of C

(C15) $(\mathsf{nvalign}_B \mid \mathsf{dnvboff}_C(B))$ and $(\mathsf{nvalign}_B \mid \mathsf{nvalign}_C)$ if B non-virtual base of C

(C16) $(\mathsf{dtdalign} \mid \mathsf{nvalign}_C)$ if C is dynamic

(C17) $(\mathsf{nvalign}_B \mid \mathsf{vboff}_C(B))$ and $(\mathsf{nvalign}_B \mid \mathsf{align}_C)$ if B is a generalized virtual base of C

(C18) $(\mathsf{align}_C \mid \mathsf{size}_C)$

In particular, (C17) implies that $(\mathsf{nvalign}_C \mid \mathsf{align}_C)$, as expected. (C16) ensures correctly-aligned accesses to the dynamic type data of a class. Finally, (C18) is used to show that an access to an element of an array of type C is correctly aligned. It is, however, not necessary for non-virtual sizes, as there are never any arrays composed only of non-virtual parts of a class.

Dynamic type data Similarly to compilers such as GCC, we chose to store the dynamic type data of a subobject at the beginning of the subobject. This leads to the following conditions for a dynamic class C:

(C19) $\mathsf{dtdsize} \leq \mathsf{fboundary}_C$

(C20) $\mathsf{pbase}_C = \varnothing \;\Rightarrow\; \mathsf{dtdsize} \leq \mathsf{dnvboff}_C(B)$ if B is a non-empty non-virtual direct base of C

(C21) $\mathsf{pbase}_C = \{B\} \;\Rightarrow\; \mathsf{dnvboff}_C(B) = 0$

Any relevant field must be laid out after the dynamic type data (C19). If C is dynamic but has no primary base, (C20) ensures that all non-empty non-virtual direct bases of C start after the dynamic type data of C. Finally, the primary base of a class is laid out at offset 0, so that the class and its primary base can share the same storage for their dynamic type data (C21).

As a consequence of (C20), (C21) and (C6), it follows that non-empty, direct, non-virtual bases of C other than its primary base are laid out at offsets at least $\mathsf{dtdsize}$.

Identity of subobjects The final set of conditions guarantees that two different subobjects of a class C of the same type B map to different offsets in memory. This identity requirement is achieved in two completely different ways, depending on whether B is empty or not. If B is non-empty, the requirement follows immediately from

(C22) C non-empty $\Rightarrow 0 < \mathsf{nvdsize}_C$

In practice, actual layout algorithms define empty bases in such a way that this condition automatically holds (a non-empty base contains at least one byte of field data or some dynamic type data), but we still need to include it in the specification.

If B is an empty base of C, neither condition (C22) nor the field separation conditions suffice to show that two different B subobjects map to distinct memory offsets. We therefore need specific conditions for empty bases.

Unfortunately, we found no local condition to ensure that two different subobjects of the same empty type are mapped to distinct offsets: we need to involve not only the layout parameters of the direct bases of the class C under consideration, but also those of all classes transitively reachable from C by inheritance or structure fields. Indeed, if we took a local interval-based condition similar to the one used for field separation – for instance, if we required the whole non-virtual zones (not only the data) of non-virtual bases to be disjoint –, then we would lose tail padding optimization, and fall back to a naive implementation similar to that of section 2.2.

Therefore, we have to compute, for any class C, the set eboffs_C of empty base offsets, i.e. the offsets of all empty classes reachable from C through inheritance or structure fields.

Similarly, we compute, for any class C, the set $\mathsf{nveboffs}_C$ of non-virtual empty base offsets, i.e. the offsets of all empty classes that are subobjects of direct non-virtual bases, or that are reachable through a field of C or a field of a non-virtual base. Define:

$$\omega + \mathcal{S} =_{\text{def}} \{(A, \omega + o) \mid (A, o) \in \mathcal{S}\}$$

Then, the sets $\mathsf{nveboffs}_C$ of the *non-virtual empty base offsets* of C and eboffs_C of *empty base offsets* of C are defined as follows:

$$\mathsf{eboffs}_C =_{\text{def}} \bigcup_{B \in \mathsf{vbases}_C \cup \{C\}} \mathsf{vboff}_C(B) + \mathsf{nveboffs}_B$$

$$\mathsf{nveboffs}_C =_{\text{def}} \text{if } C \text{ is empty then } \{(C,0)\} \text{ else } \varnothing \cup \bigcup_{B \in \mathsf{dnvbases}_C} \mathsf{dnvboffs}_C(B) + \mathsf{nveboffs}_B \cup \bigcup_{\substack{(f,B,n) \in \mathsf{stfields}_C \\ 0 \leq i < n}} \mathsf{foff}_C(f,B,n) + i \cdot \mathsf{size}_B + \mathsf{eboffs}_B$$

For the purposes of the Coq proofs, those sets are described as (mutually) inductive predicates and intended to be computed separately by algorithms. From an implementation point of view, we can assume that those sets are computed incrementally: when it comes to computing those offsets for C, they are assumed to be computed for all of the bases and structure field types of C.

Now, we impose that the unions defining $\mathsf{nveboffs}_C$ and eboffs_C be disjoint:

(C23) $(\mathsf{dnvboff}_C(B_1) + \mathsf{nveboffs}_{B_1}) \# (\mathsf{dnvboff}_C(B_2) + \mathsf{nveboffs}_{B_2})$ if B_1, B_2 distinct non-virtual bases of C

(C24) $(\mathsf{dnvboff}_C(B_1) + \mathsf{nveboffs}_{B_1}) \# \bigcup_{0 \le j < n} \mathsf{foff}_C(f, B_2, n) + j \cdot \mathsf{size}_{B_2} + \mathsf{eboffs}_{B_2}$ if B_1 non-virtual base of C and (f, B_2, n) structure field of C

(C25) $\bigcup_{0 \le j_1 < n_1} \mathsf{foff}_C(f_1, B_1, n_1) + j_1 \cdot \mathsf{size}_{B_1} + \mathsf{eboffs}_{B_1} \# \bigcup_{0 \le j_2 < n_2} \mathsf{foff}_C(f_2, B_2, n_2) + j_2 \cdot \mathsf{size}_{B_2} + \mathsf{eboffs}_{B_2}$ if (f_1, B_1, n_1) and (f_2, B_2, n_2) distinct structure fields of C

(C26) $(\mathsf{vboff}_C(B_1) + \mathsf{nveboffs}_{B_1}) \# (\mathsf{vboff}_C(B_2) + \mathsf{nveboffs}_{B_2})$ if B_1, B_2 distinct generalized virtual bases of C

In practice, whenever the algorithm tries to lay out a base or a field, it can check its empty base offsets against the empty base offsets laid out so far for the main class; whenever this check fails, the offset is incremented by the alignment of the component. As the algorithms in section 6 show, these checks can be simplified by imposing tighter layout constraints, at the cost of losing some tail padding optimizations.

5. Correctness proof

We now show that the conditions stated in section 4 are sufficient to guarantee the various separation, alignment and disjointness properties discussed in section 2.1, these properties being, in turn, sufficient to ensure semantic preservation during compilation. All the results below have been mechanically verified using the Coq proof assistant. We therefore only sketch the proofs, giving an idea of how the various conditions are used.

5.1 Field separation

Consider a generalized subobject p of static type A from an array of structures of type C, and a field f defined in class A. Define $\mathsf{Foff}_C(p, f)$ to be the offset of the field f of the subobject designated by p, within the representation of the array of type C:

$$\mathsf{Foff}_C(p, f) = \mathsf{off}_C(p) + \mathsf{foff}_A(f)$$

Theorem 1. *If p_1, p_2 are two generalized subobjects of static type A_1, A_2 within an array of structures of type C, and if f_1, f_2 are scalar fields of A_1, A_2 of types t_1, t_2 respectively, such that $(p_1, f_1) \neq (p_2, f_2)$, then the memory areas associated with these fields are disjoint:*

$$[\mathsf{Foff}_C(p_1, f_1), \mathsf{Foff}_C(p_1, f_1) + \mathsf{scsize}_{t_1}) \# [\mathsf{Foff}_C(p_2, f_2), \mathsf{Foff}_C(p_2, f_2) + \mathsf{scsize}_{t_2})$$

Proof. First we show that two distinct scalar fields reachable through base class subobjects s_1 and s_2 of a class C are disjoint. There are two cases. If $s_1 = s_2$, then $f_1 \neq f_2$ and (C9) concludes. Otherwise, we show that, if A_i is the static type of s_i, then each f_i is included (C8, C7, C11, C10) in the *field data zone* of A_i (the memory zone between offsets $\mathsf{fboundary}_{A_i}$ and $\mathsf{nvdsize}_{A_i}$ within A_i) such that the two field data zones of s_1 and s_2 are disjoint.

Then we show that if s_1 and s_2 are two distinct base class subobjects of C through non-virtual inheritance only, then their field data zones are disjoint. There are two cases. If one subobject, say s_1, is a subobject of the other s_2, then, A_2 denoting the static type of s_2, field f_2 is on the right-hand side of the field boundary of A_2 (C8), whereas f_1 is included in a direct non-virtual base of A_2, which is on the left-hand side of the field boundary of A_2 (C7), which concludes. Otherwise, s_1 and s_2 are subobjects of some classes A_1 and A_2 that are distinct direct non-virtual bases of some class A, so a similar inclusion scheme, combined with (C6), concludes.

Then, if s_1 and s_2 are distinct base class subobjects of C, we consider the generalized virtual bases of which s_1 and s_2 are non-virtual subobjects. If they are equal, then the non-virtual case above can be reused. Otherwise, (C12) concludes.

Finally, we can show our main theorem by induction on the length of the array path of, say, p_1. Then, for each i, (p_i, f_i) can be decomposed into (j_i, s_i, f'_i, P'_i) where j_i is an integer less than the size n of the array of type $C[n]$ from which p_1 and p_2 start, and s_i is a base class subobject of C of some static type B_i, and f'_i is a field of A_i such that either $(f'_i, P'_i) = (f_i, \varnothing)$ (no structure fields are traversed), or f'_i is a structure field of type $A_i[n_i]$ and $P'_i = \{p'_i\}$ for some generalized subobject p_i from the array $A_i[n_i]$. In that case, the array path of p'_1 is one element shorter than the one of p_1, which allows induction. There are five cases. If $j_1 \neq j_2$, then the scalar fields are included in distinct cells of the initial array. As two distinct fields of an array are necessarily disjoint, we conclude, additionally using (C13, C4). If $s_1 \neq s_2$, then fields f'_1 and f'_2 are included in disjoint field data zones; as fields f_1 and f_2 are included in the data zones of f_1 and f_2, this concludes. If $f'_1 \neq f'_2$, then their data zones are disjoint; as the data zone of each f_i is included in the data zone of f'_i (which may be the same, if $f'_i = f_i$). Otherwise, $f'_1 = f'_2$ is a structure field, and we may use the induction hypothesis.

It is legal to use all those conditions, as all considered classes are non-empty: as f_1 and f_2 are scalar fields, the classes defining them are non-empty, so are their derived classes, the structure fields containing them, and so on. □

5.2 Alignment

Theorem 2. *If p is a generalized subobject of static type A within an array of structures of type C, and if f is a scalar field of A of type t, then the access to f via p is correctly aligned:*

$$(\mathsf{scalign}_t \mid \mathsf{Foff}_C(p, f)) \text{ and } (\mathsf{scalign}_t \mid \mathsf{align}_C)$$

Proof. Easy induction on the path, using transitivity of the "evenly divides" relation, alignment conditions (C14, C15, C17), and additionally (C18) for array cell accesses. Access to the i-th cell of an array of structures of type B corresponds to shifting by $i \cdot \mathsf{size}_B$, which requires an alignment condition for size_B. □

Theorem 3. *If p is a generalized subobject of static type A within an array of structures of type C, and if A is dynamic, then the access to the dynamic data of p is correctly aligned:*

$$(\mathsf{dtdalign} \mid \mathsf{off}_C(p)) \text{ and } (\mathsf{dtdalign} \mid \mathsf{align}_C)$$

Proof. Same reasoning as above, additionally using (C16). □

5.3 Identity of subobjects

Theorem 4. *If p_1, p_2 are two different generalized subobjects of the same static type A within an array of structures of type C, then their corresponding offsets are distinct:*

$$p_1 \neq p_2 \Rightarrow \mathsf{off}_C(p_1) \neq \mathsf{off}_C(p_2)$$

Proof. There are two radically different sub-proofs depending on whether A is empty. If it is not empty, then the proof is similar to that of theorem 1, thanks to condition (C22). If A is empty, all those conditions no longer apply. By induction on the length of the array path of p_1, there are two cases. If p_1 and p_2 originate from the same cell of the array of structures of type C, then, (C23, C24, C25, C26) and the induction hypothesis conclude. Otherwise, we know that the two cells are entirely disjoint (not only their data), so we simply have to show that each p_i is included in its own cell, using (C1, C2, C3) and also (C5). □

5.4 Preservation of dynamic type data

The following theorem ensures that writing a scalar field does not change the dynamic type data of subobjects.

Theorem 5. *If p_1, p_2 are two generalized subobjects of static type A_1, A_2 within an array of structures of type C, and if A_1 is a dynamic class and if f_2 is a scalar field of A_2 of type t_2, then the memory area occupied by the dynamic type data of A_1 is disjoint from the memory area occupied by field f_2:*

$$[\mathit{off}_C(p_1), \mathit{off}_C(p_1) + \mathit{dtdata}) \\ \# \quad [\mathit{Foff}_C(p_2, f_2), \mathit{Foff}_C(p_2, f_2) + \mathit{scsize}_{t_2})$$

Proof. The proof resembles that of theorem 1, except that data inclusions (not only field data inclusions) must be checked. (C19) ensures that dynamic type data is included in the data of a subobject, but it also ensures that it is disjoint from the field data zones if p_1 is a generalized subobject that is reachable from p_2 by inheritance and/or scalar fields. Conversely, if p_2 is reachable from p_1, then (C20, C21) must be additionally used. □

A dynamic class C may share its dynamic type data with one of its dynamic non-virtual bases B: the primary base of C. But the semantics of C++ guarantees that dynamic operations (e.g. dynamic cast, dynamic function dispatch) on a base class subobject s_2 of C give the same result as on a base class subobject s_1 of C if s_2 is reachable from s_1 (i.e. if s_1 *dominates* s_2 [19]). As the primary base B is a base of C, this sharing is semantically sound.

However, initialization of dynamic data has still to be checked. Even without a detailed semantics of object construction and destruction, which is out of the scope of this paper, we still have to know whether initializing the dynamic data of a subobject does not change the dynamic data of other subobjects. We proved formally that this is the case, except for two subobjects s_1 and s_2 such that s_2 can be reached by s_1 through primary bases only.

A *primary path* is a path in the primary path graph (where classes are the vertices, and (U, V) is an edge if and only if V is the primary base of U). Any non-virtual path $B :: l$ can be decomposed into a *reduced path* $\mathsf{red}(B :: l)$ such that:

- if $B :: l$ is primary, then $\mathsf{red}(B :: l) =_{\text{def}} B :: \epsilon$;
- otherwise, $B :: l$ can be rewritten as $l_1 + B_1 :: B_2 :: l_2$ where B_2 is a non-virtual base of B_1 that is not the primary base of B_1, and $B_2 :: l_2$ is a primary path. Then, $\mathsf{red}(B :: l) =_{\text{def}} l_1 + B_1 :: B_2 :: \epsilon$.

In other words, the reduced path of a non-virtual path l' is obtained by truncating the longest primary path on the right of l'.

Theorem 6. *Let $p_1 = (\alpha_1, i_1, (h_1, l_1))$, $p_2 = (\alpha_2, i_2, (h_2, l_2))$ be two generalized subobjects of static types A_1 and A_2 respectively, from an array of structures of type C. Assume A_1 and A_2 are dynamic classes. If $(\alpha_1, i_1) \neq (\alpha_2, i_2)$ or $\mathsf{red}(l_1) \neq \mathsf{red}(l_2)$, then the dynamic type data of the two subobjects are disjoint:*

$$[\mathit{off}_C(p_1), \mathit{off}_C(p_1) + \mathit{dtdata}) \\ \# \quad [\mathit{off}_C(p_2), \mathit{off}_C(p_2) + \mathit{dtdata})$$

Proof. The proof proceeds like that of theorem 5, but replacing each l'_i with its reduced path $\mathsf{red}(l'_i)$. □

5.5 Semantic preservation

To conclude this section and exercise the theorems above, we now sketch a proof of semantic preservation for the compilation scheme given in section 4.4.

The semantics of the target language is given in terms of states (V', M), where V' maps variables to target values and M is a machine-level, byte-addressed memory state in the style of Tuch [18]. Target values v' are the union of base values $\mathsf{Atom}\ bv$, data pointers $\mathsf{Ptr}\ i$ (where i, an integer, is the address of a byte in memory), and pointers to dynamic type data $\mathsf{Vptr}(C, \sigma)$. Memory states are presented abstractly through the two partial operations $\mathtt{load}(i, n, M)$ and $\mathtt{store}(i, n, v', M)$ where i, an integer, is the address of a byte, and n is a byte count. By lack of space, we omit the operational semantics of the target language, referring the reader to the online supplement [15] instead.

We now define a predicate $(V, H) \triangleright (V', M)$ relating the execution states of the program before and after compilation. We start with the relation $H \vdash v \triangleright v'$ between source values and target values:

$$H \vdash \mathsf{Atom}\ bv \triangleright \mathsf{Atom}\ bv \qquad H \vdash \mathsf{Ptr}\ \varnothing \triangleright \mathsf{Ptr}\varnothing$$

$$\frac{H(o) = (C, n, \omega) \qquad C[n] \dashv\langle(\alpha, i, \sigma)\rangle\rightarrow A}{H \vdash \mathsf{Ptr}(o, \alpha, i, \sigma) \triangleright \mathsf{Ptr}(\mathsf{objoff}(o) + \mathsf{off}_C(\alpha, i, \sigma))}$$

In the last rule, objoff is a partial mapping from source-level object identifiers o to memory addresses. It represents the initial placement of top-level objects in memory.

Agreement $(V, H) \triangleright (V', M)$ between a source state and a target state is, then, the conjunction of the following conditions:

1. Variable agreement: $H \vdash V(x) \triangleright V'(x)$ for all $x \in \mathrm{Dom}(V)$.
2. Alignment of top-level objects: $(\mathsf{align}_C \mid \mathsf{objoff}(o))$ if $H(o) = (C, n, \omega)$.
3. Separation between top-level objects:
   $[\mathsf{objoff}(o_1), \mathsf{objoff}(o_1) + n_1 \times \mathsf{size}_{C_1}) \ \#$
   $[\mathsf{objoff}(o_2), \mathsf{objoff}(o_2) + n_2 \times \mathsf{size}_{C_2})$
   if $H(o_1) = (C_1, n_1, \omega_1)$ and $H(o_2) = (C_2, n_2, \omega_2)$ and $o_1 \neq o_2$.
4. Correct values for scalar fields: $H \vdash \omega(\alpha, i, \sigma, (f, t)) \triangleright \mathsf{load}(\mathsf{objoff}(o) + \mathsf{off}_C(\alpha, i, \sigma) + \mathsf{foff}_A(f, t), \mathsf{scsize}_t, M)$ if $H(o) = (C, n, \omega)$ and $C[n] \dashv\langle(\alpha, i, \sigma)\rangle\rightarrow A$ and (f, t) is a scalar field of A.
5. Correct values for dynamic data: $\mathsf{load}(\mathsf{objoff}(o) + \mathsf{off}_C(\alpha, i, \mathsf{red}(\sigma)), \mathsf{dtdsize}, M) = \mathsf{Vptr}(B, \mathsf{red}(\sigma))$ if $H(o) = (C, n, \omega)$ and $C[n] \dashv\langle\alpha\rangle\overset{\mathcal{A}}{\rightarrow} B[m] \dashv\langle i, \sigma\rangle\overset{\mathcal{CI}}{\rightarrow} A$ and A is a dynamic class.

Theorem 7. *Every execution step of the source program is simulated by one or several execution steps of the compiled code: if $(V_1, H_1) \triangleright (V'_1, M_1)$ and $\mathit{Stmt} \vdash (V_1, H_1) \rightarrow (V_2, H_2)$, then there exists a target state (V'_2, M_2) such that $[\![\mathit{Stmt}]\!] \vdash (V'_1, M_1) \overset{+}{\rightarrow} (V'_2, M_2)$ and $(V_2, H_2) \triangleright (V'_2, M_2)$.*

Proof. If Stmt is an assignment to a scalar field, the existence of M_2 follows from theorem 2, which guarantees that the store is properly aligned and therefore succeeds. Theorem 1, combined with the "good variable" properties of the target memory model, shows part (4) of the agreement between the final states. Part (5) likewise follows from theorem 5 and the good variable properties.

If Stmt is a pointer comparison, theorem 4 ensures that the address comparison generated by the compilation scheme produces the same boolean outcome.

The remaining cases follow more or less directly from the hypothesis $(V_1, H_1) \triangleright (V_1', M_1)$. □

6. Verification of representative layout algorithms

We now put our formalization into practice by using it to prove the correctness of two realistic, optimizing C++ object layout algorithms.

6.1 The common vendor C++ ABI

The first layout algorithm we study is the one specified in the "common vendor" C++ ABI [3], first introduced for Itanium and now used by the GCC compiler on many other platforms. Our algorithm is faithful to the ABI specification with one exception: we do not allow the use of nearly empty virtual bases as primary bases, an optimization of dubious value discussed in section 7.

Following the ABI, we define a class C to be dynamic if, and only if, it has a virtual function, or a non-virtual dynamic base, or a virtual base. In the latter case, C is considered dynamic as its dynamic type data is actually used to find the offsets of the virtual bases of C. Likewise, we define a class to be empty if, and only if, all the following conditions hold: it has no virtual function; it has no non-static fields; it has no virtual bases; all its direct non-virtual bases are empty. In particular, an empty class cannot be dynamic.

The layout algorithm is summarized below. (See [15] for the full pseudocode.) It takes as input a class C and the layout of all classes B mentioned in C (i.e. B is a direct or indirect base of C, or B is the type of a field of C, or depends on a base of C or of the type of a field of C). Besides the layout parameters listed in section 4.2, it also computes the set eboffs'_C of the offsets of all subobjects of an empty virtual base of C or an empty non-virtual direct base of C.

1. Start from $\mathsf{nvdsize}_C = 0$ and $\mathsf{nvalign}_C = \mathsf{nvsize}_C = 1$.
2. Arbitrarily choose a dynamic non-virtual direct base B, if any. This will be the primary base. Give it offset 0 within C, and update $\mathsf{nveboffs}_C$, $\mathsf{nvdsize}_C$, nvsize_C and $\mathsf{nvalign}_C$ to their B counterparts.
3. If there is no primary base but the class is dynamic, then reserve some space for the dynamic type data: update $\mathsf{nvdsize}_C$ and nvsize_C to $\mathsf{dtdsize}$, and $\mathsf{nvalign}_C$ to $\mathsf{dtdalign}$. Start at offset equal to the size of dynamic type data.
4. Then, for each non-primary non-virtual direct base B of C:
   - Try to give it an offset $\mathsf{dnvboff}_C(B)$ within C, starting from the least multiple of $\mathsf{nvalign}_B$ no less than $\mathsf{nvdsize}_C$. If there is a type conflict with empty base offsets, then try at a further offset by increasing $\mathsf{dnvboff}_C(B)$ by $\mathsf{nvalign}_B$, knowing that for $\mathsf{dnvboff}_C(B) \geq \mathsf{nvsize}_C$ there will be no conflict. However, if B is empty, first try 0 before trying any offset from $\mathsf{nvdsize}_C$.
   - Update nvsize_C to $\max(\mathsf{nvsize}_C, \mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B)$, and $\mathsf{nvalign}_C$ to $\mathrm{lcm}(\mathsf{nvalign}_C, \mathsf{nvalign}_B)$.
   - If B is not empty, also update $\mathsf{nvdsize}_C$ to $\mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B$: in this case, note that the whole non-virtual part of B (not only its data) is included in the data of C
5. Set $\mathsf{fboundary}_C$ to $\mathsf{nvdsize}_C$.
6. Then for each field, try to lay it out. Starting from $\mathsf{nvdsize}_C$, find a correctly aligned offset (incrementing by $\mathsf{falign}(f)$), and set $\mathsf{nvdsize}_C$ to $\mathsf{foffset}_C(f) + \mathsf{fsize}(f)$, so that the next field starts after the end of the whole previous field (not only its data). This explains why any class having a field, even an irrelevant one, is not empty.
7. Then for each virtual base, try to lay out its non-virtual part, the same way as for non-virtual bases, but updating align_C, size_C, etc. instead of $\mathsf{nvalign}_C$, nvsize_C, etc.

Type conflicts with empty base offsets are detected as follows. When trying to lay out an empty base B, check between B's non-virtual empty base offsets and the whole set of empty base offsets so far. When trying to lay out a non-empty base B, or a field of type $B[n]$, check between the base's or the field's non-virtual empty base offsets and the offsets of C so far reachable by inheritance only through an empty direct non-virtual base or an empty virtual base. In other words, ignore those offsets of bases reachable through fields, or through non-empty bases of C. Indeed, those offsets are guaranteed to be laid out in the data zone of C, as this algorithm ensures that a field, or a non-empty base, is wholly included (not only its data) in the data of C. As the data zone of already laid out relevant components is disjoint from the data zone of the current relevant component, there is no need to check whether those offsets are present in the empty base offsets of the current component.

These details about type conflict resolution were not present in the ABI specification [3]: it gave no clue about which offsets to empty subobjects need to be checked.

We proved that this algorithm satisfies all the conditions stated in section 4. Thus, it is semantically correct with respect to field access, dynamic operations and subobject identification.

6.2 A more efficient layout: empty member optimization

The previous algorithm misses several opportunities for saving space. Consider:

```
struct A0    {};
struct A: A0 {};
struct B     { char f; A a; };
struct C     { B b; char f; };
struct D: A  { A a; };
struct E: D  { char f; };
```

Running this example through GCC 4.3 (which follows the common vendor ABI), we obtain `sizeof(C) == 3` and `sizeof(E) == 3`. This indicates that field `f` of `C` is laid out completely disjointly from `b`, even though `b.a` contains no data (`A` is empty). The space for `b.a` could be reused, giving `sizeof(C) == 2`. Likewise, field `f` of `E` is laid out completely disjointly from the subobject `D`, which is considered as not empty even though `D::a` contains no data. The space for `D` could be reused, resulting in `sizeof(E) == 2`. This optimisation is justified insofar as an object of empty class has no source of observable behavior other than its address: it carries no runtime data. Consequently a conforming C++ implementation can systematically compile assignments to such objects as no-ops.

We propose an algorithm that performs these space optimizations by refining the notion of empty classes. Say that a class is empty if, and only if, all the following conditions hold:

- it has no virtual functions;
- it has no scalar fields;
- it has no virtual bases;
- all its direct non-virtual bases are empty;
- all its structure fields are of an empty type.

In particular, an empty class cannot be dynamic. (This definition of empty classes is the smallest that satisfies the conditions from section 4.3.) Then, the previous algorithm is modified as follows:

- When laying out an irrelevant component, all correctly aligned offsets starting from 0 are tried, until the offsets to empty subobjects do not clash, or nvsize_C is reached
- A field can start in the tail padding of the previous field: when a relevant field is being laid out, the data size is updated to $\mathsf{foff}_C(f) + \mathsf{fdsize}(f)$ instead of $\mathsf{foff}_C(f) + \mathsf{fsize}(f)$. Similarly for a non-empty base.

This requires a modification when checking for type conflicts with empty base offsets: the whole sets of offsets must be considered every time. Indeed, as there are no more conditions about the sizes of the fields or the non-virtual sizes of the bases, there are no more guarantees that an empty subobject reachable from an already laid out component will not clash with the current component being laid out. Similar modifications occur for fields, and virtual bases (in the latter case, checking against eboffs_C). As all offsets are checked, the auxiliary set eboffs'_C is no longer useful.

We proved this algorithm to be correct. In fact, the proof is easier than for the previous algorithm, as this algorithm is closer to the conditions stated in Section 4.

7. Limitations and extensions

Bit fields In our formalization, we did not specify the size unit, so that sizes and offsets can be expressed in bits instead of bytes. However, this may not be enough to fully implement bit fields, as they require specific alignment constraints. Consider:

```
struct A { int i: 29; int j: 2; int k: 3; }
```

On a 32-bit platform, fields `i` and `j` can be packed into a single 32-bit integer, and retrieved by shifting and masking from this integer. However, the field `k` should not be packed adjacent to `j`, otherwise two correctly-aligned memory accesses would be required to recover the value of `k`. Our alignment constraints over field offsets would have to be strengthened accordingly.

POD Our work does not consider the specificities of POD (*Plain Old Data*), and treats them in an ordinary way, not distinguishing them from general C++ structures. Roughly speaking, a POD structure is a structure with no inheritance, and no non-POD fields: as such, it is roughly equivalent to a C structure. The Standard mandates the layout of POD structures to be compatible with C, allowing in particular bitwise assignment between POD structures using memcpy. By contrast, our work considers that structures are always assigned to by memberwise assignments of fields. However, we suspect that constraining $\mathsf{nvdsize}_C = \mathsf{nvsize}_C = \mathsf{size}_C$ whenever C is a POD could help us prove the correctness of bitwise copy, even if a POD is inherited by another class.

Unions Just like structures, unions offer opportunities for reusing tail padding that is common to all of its members. The practical benefits are low: unions are used infrequently enough that treating them like POD is a reasonable choice. The main difficulty in extending our formalism to unions comes from the operational semantics, which must be instrumented to enforce the policy that, at any time, the only member of an union that can be accessed is the one most recently initialized.

Virtual primaries In our work, a class can share its dynamic type data only with a non-virtual primary base. However, the common vendor ABI [3] allows a class to share its dynamic type data with one of its virtual bases, as long as the latter has no fields. In practice, the layout of such a virtual base (called a *nearly empty virtual base*) is reduced to its dynamic type data, so that it can be shared with the dynamic type data of its derived class. However, such a layout would break the schema of laying out the non-virtual part of a class separately from the non-virtual parts from its virtual bases, as some non-empty virtual bases may be actually laid out in the non-virtual part of the class. Moreover, indirect primary virtual bases may appear several times in a class layout because they have been chosen as primary bases by different non-virtual bases. We do not know how to solve this layout ambiguity. In practice, the latter case also actually poses efficiency problems in most compilers, so much so that the "virtual primary" optimization is described as "an error in the design" of the common vendor ABI [3].

Alternate layout of dynamic type data Our work assumes that dynamic type data is laid out at the beginning of a class. Some C++ compilers such as Cfront or Compaq's compiler elect to place it at a different offset. That layout scheme has the property that a pointer to the complete object is also a pointer to the data of the primary base class subobject, which may not need dynamic type data. How frequently that situation occurs in practice is unclear. Moreover, it can lead to alignment padding that is more difficult to reuse in derived classes.

Bidirectional layout In our work as in production C++ compilers, object representation can only be extended on one side in derived classes. There are, however, some layouts [7] that may extend an object representation to both sides, so that for instance a class may share its dynamic type data with two of its bases. However, an efficient implementation of such layout would need to know the derived classes of a class in advance, which would impede separate compilation.

Reusing padding holes Our work assumes that the non-virtual data of a class is a contiguous memory zone. Consequently, holes arising from inter-field padding cannot be detected and reused, as in the following example:

```
struct A0    { }
struct A: A0 { char x; int y; };
struct B: A  { char z; };
```

There is some alignment padding between `A::x` and `A::y`, but in the layout of `B`, our formalization does not allow `z` to be stored in this unused space, as the data of `A` is considered to be contiguous.

Virtual table layout Our work focused so far on the representation of objects, leaving unaddressed the concrete representation of dynamic type data. Future work includes formalizing the layout of virtual tables as studied by Sweeney and Burke [17], as well as **`this`** pointer adjustment during virtual function dispatch. This would lead to mechanized verification of the implementation of dynamic operations such as virtual function dispatch and dynamic cast.

Object construction and destruction A related piece of future work is the formalization of object construction and destruction, especially the updates to the dynamic type data of objects that take place during this process.

8. Related work

Obviously, any C++ compiler includes an object layout algorithm. However, to the best of our knowledge, none has been reported as formally verified. Stroustrup [5] extensively discusses object layouts found in earlier C++ compilers. Although he explicitly stated "Objects of an empty class have a nonzero size" [5, p. 164], he did not consider empty base class optimization. Nor was that optimization implemented in Cfront. The nonzero size requirement was later clarified by the C++ standard [8, paragraph 10/5].

The empty base optimization, as a basis for efficient programming technique, was popularized by Myers [13]. He credited Jason Merrill for the possibility of optimizing empty *member* subobjects, as we consider in this paper. However, there was no proposed algorithm, and as we observed such an optimization requires careful assumptions about PODs (*Plain Old Data*). The empty base class optimization is explicitly mandated by the common vendor ABI [3], which is a practical basis of our formalization and improvements.

Our present work is based on the algebraic model of inheritance by Rossie and Friedman [16], and the operational semantics of Wasserrab *et al* [19]. These foundational papers do not consider concrete object layout algorithms. Rather, they purposefully focus

on abstract object semantics independent of concrete machine representations.

There are other formalizations of aspects of the compilation of C++, but none of them considers concrete data representation. Chen [2] proposed a typed intermediate language for compiling multiple inheritance. This work formally describes which pointer adjustments are necessary for virtual function calls and conversions, including offset computations and thunks, and proves type soundness results about them. However, Chen's formalization, focusing only on an intermediate language, leaves largely unspecified the concrete object layout. The delicate issue of subobject identity is not addressed either.

Luo and Qin [12] proposed a separation logic-based formalism for reasoning about C++-style multiple inheritance. This paper does not address the thorny issues of concrete object layout. Rather, it relies on a syntactic form of field resolution close to the algorithm of Ramalingam and Srinivasan [14], itself a reformulation of the calculus of Rossie and Friedman [16]. Furthermore, Luo and Qin restricted themselves to non-virtual multiple inheritance only, considered a rather abstract storage model, and defined the semantics of field access through substitution.

In a completely different direction, Gil and Sweeney [6] proposed an arguably space and time-efficient bidirectional object layout scheme for multiple inheritance. The core of their algorithms is based on the assumption that the compiler knows the complete class hierarchy. That assumption holds only in special cases (e.g. in a closed world with whole program analysis), and is inadequate for most realistic C++ compilers at large, which must cope with a separate compilation model inherited from C. This theoretical work was followed up by an quantitative study by Sweeney and Burke [17]. They developed a formalism to characterize when compiler artifacts (to support runtime semantics of C++-style inheritance) are required. Their work influenced design choices in the memory layout of the IBM Visual Age C++ V5 compiler. The core of the bidirectional layout was later refined by Gil, Pugh, Weddell and Zibin [7] to a "language-independent" object layout algorithm for multiple inheritance. However, they explicitly exclude C++-style non-virtual multiple inheritance on the ground that it is "a rarity or [...] an abomination". It is not rare, and we cannot afford such assumption for formally verified layout algorithms in real world compilers for ISO standard C++.

Leaving C++ objects for the simpler world of C structures, Tuch [18] axiomatized a field separation property of structure layout and used it in a separation logic able to verify low-level system C code. The Clight formal semantics of Blazy and Leroy [1] defines a simple structure layout algorithm; the field separation, field alignment and prefix compatibility properties were mechanically verified.

9. Conclusions

C++ object layout is one of those seemingly-simple implementation issues that turn out to require a 18 000 line Coq development to start making formal sense. Between the few compilers in the past that over-optimized the layout, losing object identity in the process, and the great many contemporary compilers that err on the side of caution and miss opportunities for saving space, our formalization and mechanized verification delineate a design space of trustworthy, yet efficient layout algorithms, bringing more confidence in commonly-used ABIs and suggesting further safe optimizations.

This work is also a first step towards formally verifying a compiler front-end for a realistic subset of C++, even though much work remains to be done on other aspects of the C++ object model (virtual dispatch, construction and destruction, etc).

Acknowledgments

We are grateful to the anonymous reviewers for their very helpful suggestions to improve our results and their presentation. We thank Bjarne Stroustrup and Lawrence Rauchberger for their feedback, which helped us to keep our formalization connected to the realistic practices of C++ compilers.

References

[1] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[2] J. Chen. A typed intermediate language for compiling multiple inheritance. In *34th symp. Principles of Programming Languages*, pages 25–30. ACM, 2007.

[3] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ ABI, 2001. URL `http://www.codesourcery.com/public/cxx-abi`.

[4] B. Dawes. POD's Revisited; Resolving Core Issue 568 (Revision 2). Technical report, ISO/IEC SC22/JTC1/WG21, March 2007. URL `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2172.html`.

[5] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[6] J. Gil and P. F. Sweeney. Space and time-efficient memory layout for multiple inheritance. In *14th conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999)*, pages 256–275. ACM, 1999.

[7] J. Y. Gil, W. Pugh, G. E. Weddell, and Y. Zibin. Two-dimensional bidirectional object layout. *ACM Trans. Program. Lang. Syst.*, 30(5): 1–38, 2008.

[8] *International Standard ISO/IEC 14882:2003. Programming Languages — C++*. International Organization for Standards, 2003.

[9] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Prog. Lang. Syst.*, 28(4):619–695, 2006.

[10] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *Int. Conf. on Software Engineering and Formal Methods (SEFM 2005)*, pages 2–11. IEEE, 2005.

[11] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[12] C. Luo and S. Qin. Separation Logic for Multiple Inheritance. *Electron. Notes Theor. Comput. Sci.*, 212:27–40, 2008.

[13] N. Myers. The empty member C++ optimization. *Dr Dobb's Journal*, Aug. 1997. URL `http://www.cantrip.org/emptyopt.html`.

[14] G. Ramalingam and H. Srinivasan. A member lookup algorithm for C++. In *Programming Language Design and Implementation (PLDI'97)*, pages 18–30. ACM, 1997.

[15] T. Ramananandro. Formal verification of object layout for C++ multiple inheritance – Coq development and supplementary material, 2010. URL `http://gallium.inria.fr/~tramanan/cxx/`.

[16] J. G. Rossie and D. P. Friedman. An algebraic semantics of subobjects. In *10th conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1995)*, pages 187–199. ACM, 1995.

[17] P. F. Sweeney and M. G. Burke. Quantifying and evaluating the space overhead for alternative C++ memory layouts. *Software: Practice and Experience*, 33(7):595–636, 2003.

[18] H. Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *Journal of Automated Reasoning*, 42(2):125–187, 2009.

[19] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *21st conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 345–362. ACM, 2006.