An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++

Daniel Wasserrab
Universität Passau
wasserra@fmi.uni-passau.de

Tobias Nipkow
Technische Universität München
nipkow@in.tum.de

Gregor Snelting
Universität Passau
snelting@fmi.uni-passau.de

Frank Tip
IBM T.J. Watson Research Center
ftip@us.ibm.com

Abstract

We present an operational semantics and type safety proof for multiple inheritance in C++. The semantics models the behavior of method calls, field accesses, and two forms of casts in C++ class hierarchies exactly, and the type safety proof was formalized and machine-checked in Isabelle/HOL. Our semantics enables one, for the first time, to understand the behavior of operations on C++ class hierarchies without referring to implementation-level artifacts such as virtual function tables. Moreover, it can—as the semantics is executable—act as a reference for compilers, and it can form the basis for more advanced correctness proofs of, e.g., automated program transformations. The paper presents the semantics and type safety proof, and a discussion of the many subtleties that we encountered in modeling the intricate multiple inheritance model of C++.

1. Introduction

We present a operational semantics and type safety proof for the multiple inheritance model of C++ in all its complexity, including both repeated and shared (virtual) inheritance. This semantics enables one—for the first time!—to fully understand and express the behavior of operations such as method calls, field accesses, and casts in C++ programs without referring to compiler data structures such as virtual function tables (v-tables) [27].

Type safety is a language property which can be summarized by the famous slogan "Well-typed programs cannot go wrong" [13]. Cardelli's definition of type safety [6] demands that no untrapped errors may occur (although controlled exceptions are allowed). The type safety property that we prove is the fact that the execution of a well-typed, terminating program will deliver a result of the expected type, or end with an exception. The semantics and proof are formalized and machine-checked using the Isabelle/HOL theorem prover [14].

One of the main sources of complexity in C++ is a complex form of multiple inheritance, in which a combination of shared ("virtual") and repeated ("nonvirtual") inheritance is permitted. Because of this complexity, the behavior of operations on C++ class hierarchies has traditionally been defined informally [28], and in terms of implementation-level constructs such as v-tables. We are only aware of a few formal treatments—and of no operational semantics—for C++-like languages with shared and repeated multiple inheritance. The subobject model by Rossie and Friedman [20], upon which our work is based, formalizes the object model of C++. Rossie and Friedman defined the behavior of method calls and member access using this model, but their definitions do not follow C++ behavior precisely, they do not consider the behavior of casts, and they do not provide an operational semantics. In 1996, Rossie, Friedman, and Wand [21] stated that "In fact, a provably-safe static type system [...] is an open problem", and to our knowledge this problem has remained open until today.

The CoreC++ language studied in this paper features all the essential elements of the C++ multiple inheritance model (while omitting many features not relevant to operations involving class hierarchies). The semantics of CoreC++ were designed to mirror those of C++ to the maximum extent possible. In previous versions of the semantics [36], we explored a number of variations, and we will briefly discuss these in §8.

Our interest in formalizing the semantics of multiple inheritance was motivated by previous work by two of the present authors on: (i) restructuring class hierarchies in order to reduce object size at run-time [33], (ii) composition of class hierarchies in the context of an approach for aspect-orientation [24], and (iii) refactoring class hierarchies in order to improve their design [25, 23], In each of these projects, class hierarchies are *generated*, multiple inheritance may arise naturally, and additional program transformations are then used to replace multiple inheritance by a combination of single inheritance and delegation. We plan to exploit our formal semantics for a correctness proof of these transformations.

In summary, this paper makes the following contributions:

- We present a formal semantics and machine-checked type safety proof for multiple inheritance in C++. This enables one, for the first time, to understand and express the behavior of operations involving C++ class hierarchies without referring to compiler data structures.
- We discuss some subtle ambiguities concerning the behavior of member access and method calls in C++ that were uncovered in the course of designing the semantics.
- By formalizing the complex behavior of C++ multiple inheritance, we extend the applicability of formal semantics and theorem prover technology to a new level of complexity.

Thus the message to language semanticists is that the much maligned C++ system of multiple inheritance contains a perfectly sound core.

2. Multiple inheritance

2.1 An intuitive introduction to subobjects

C++ features both *nonvirtual* (or *repeated*) and *virtual* (or *shared*) multiple inheritance. The difference between the two flavors of inheritance is subtle, and only arises in situations where a class Y indirectly inherits from the same class X via more than one path in the hierarchy. In such cases, Y will contain *one* or *multiple* X-"subobjects", depending on the kind of inheritance that is used. More precisely, if only shared inheritance is used, Y will contain a single, shared X-subobject, and if only repeated inheritance is used, the number of X-subobjects in Y is equal to N, where N is the number of distinct paths from X to Y in the hierarchy. If a combination of shared and repeated inheritance is used, the number of X-subobjects in an Y-object will be between 1 and N (a more precise discussion follows). C++ hierarchies with only single inheritance (the distinction between repeated and shared inheritance is irrelevant in this case) are semantically equivalent to Java class hierarchies.

Fig. 1(a) shows a small C++ class hierarchy. In these and subsequent figures, a solid arrow from class C to class D denotes the fact that D repeated-inherits from C, and a dashed arrow from class C to class D denotes the fact that D shared-inherits from C. Here, and in subsequent examples, all methods are assumed to be `virtual` (i.e. dynamically dispatched), and all classes and inheritance relations are assumed to be `public`.

In Fig. 1(a), all inheritance is repeated. Since class `Bottom` repeated-inherits from classes `Left` and `Right`, a `Bottom`-object has one subobject of each of the types `Left` and `Right`. As `Left` and `Right` each repeated-inherit from `Top`, (sub)objects of these types contain distinct subobjects of type `Top`. Hence, for the C++ hierarchy of Fig. 1(a), an object of type `Bottom` contains *two distinct subobjects* of type `Top`. Fig. 1(b) shows the layout used for a `Bottom` object by a typical compiler, given the hierarchy of Fig. 1(a). Each subobject has local copies of the subobjects that it contains, hence it is possible to lay out the object in a contiguous block of memory without indirections.

Fig. 2(a) shows a similar C++ class hierarchy in which the inheritance between `Left` and `Top` and between `Right` and `Top` is *shared*. Again, a `Bottom`-object contains one subobject of each of the types `Left` and `Right`, due to the use of repeated inheritance. However, since `Left` and `Right` both shared-inherit from `Top`, the `Top`-subobject contained in the `Left`-subobject is *shared* with the one contained in the `Right`-subobject. Hence, for this hierarchy, a `Bottom`-object will contain *a single subobject* of type `Top`. In general, a shared subobject may be shared by arbitrarily many subobjects, and requires an object layout with indirections (typically in the form of *virtual-base pointers*) [27, p.266] [1]. Fig. 2(b) shows a typical object layout for an object of type `Bottom` given the hierarchy of Fig. 2(a). Observe, that the `Left`-subobject and the `Right`-subobject each contain a pointer to the single shared `Top`-subobject.

2.2 The Rossie-Friedman Subobject Model

Rossie and Friedman [20] proposed a subobject model for C++-style inheritance, and used that model to formalize the behavior of method calls and field accesses. Informally, one can think of the Rossie-Friedman model as an abstract representation of object layout. Intuitively, a *subobject*[2] identifies a component of type D that is embedded within a complete object of type C. However, simply defining a subobject type as a pair (C, D) would be insufficient, because, as we have seen in Fig. 1, a C-object may contain multiple D-components in the presence of repeated multiple inheritance. Therefore, a subobject is identified by a pair $[C, Cs]$, where C denotes the type of the "complete object", and where the *path* Cs consists of a sequence of class names $C_1 \cdot \cdots \cdot C_n$ that encodes the transitive inheritance relation between C_1 and C_n. There are two cases here: For *repeated* subobjects we have that $C_1 = C$, and for *shared* subobjects, we have that C_1 is the least derived (most general) shared base class of C that contains C_n. This scheme is sufficient because shared subobjects are unique within an object (i.e. there can be at most one *shared* subobject of type S within any object). More formally, for a given class C, the set of its subobjects, along with a containment ordering on these subobjects, is inductively defined as follows:

(i) $[C, C]$ is the subobject that represents the "full" C-object.

(ii) if $S_1 = [C, Cs.X]$ is a subobject for class C where Cs is any sequence of class names, and X shared-inherits from Y, then $S_2 = [C, Y]$ is a subobject for class C that is accessible from S_1 through a pointer.

(iii) if $S_1 = [C, Cs.X]$ is a subobject for class C where Cs is any sequence of class names, and X repeated-inherits from Y, then $S_2 = [C, Cs.X.Y]$ is a subobject for class C that is directly contained within subobject S_1.

Fig. 1(c) and Fig. 2(c) show *subobject graphs* for the class hierarchies of Fig. 1 and Fig. 2, respectively. Here, an arrow from subobject S to subobject S' indicates that S' is directly contained in S or that S has a pointer leading to S'. For a given subobject $S = [C, Cs.D]$, we call C the *dynamic class* of subobject S and D the *static class* of subobject S. Associated with each subobject are the members that occur in its static class. Hence, if an object contains multiple subobjects with the same static class, it will contain multiple copies of members declared in that class. For example, the subobject graph of Fig. 1(c) shows two subobjects with static class `Top`, each of which has distinct fields `x` and `y`.

Intuitively, a subobject's dynamic class represents the type of the "full object" and is used to resolve dynamically dispatched method calls. A subobject's static class represents the declared type of a variable that points to an (subobject of the full) object and is used to resolve field accesses. In this paper, we use the Rossie-Friedman subobject model to define the behavior of operations such as method calls and casts as functions from subobjects to subobjects. As we shall see shortly, it will be necessary in our semantics to maintain full subobject information even for "static" operations such as casts and field accesses.

Multiple inheritance can easily lead to situations where multiple members with the same name are visible. In C++, many member accesses that are seemingly ambiguous are resolved using the notion of *dominance* [28]. A member m in subobject S' *dominates* a member m in subobject S if S is contained in S' (i.e. S' occurs below S in the subobject graph). Member accesses are resolved by selecting the unique dominant member m if it exists; otherwise an access is ambiguous[3]. For example, in Fig. 2, a `Bottom`-object sees two declarations of `f()`, one in class `Right` and one in class `Top`. Thus a call `(new Bottom())->f()` seems ambiguous. But it is not, because in the subobject graph for `Bottom` shown in Fig. 2(c), the definition of `f()` in [`Bottom`,`Bottom.Right`] dominates the one in [`Bottom`,`Top`]. On the other hand, the subobject graph in Fig. 1(c) con-

[1] An alternative implementation mechanism is to store the offsets to shared subobjects in vtables.

[2] In this paper, we follow the terminology of [20] and use the term "subobject" to refer both to the label that uniquely identifies a component of an object type, as well as to components within concrete objects that are identified by such labels. In retrospect, the term "subobject label" would have been better terminology for the former concept.

[3] In some cases, C++ uses the static class of the receiver for further disambiguation. This will be discussed shortly.

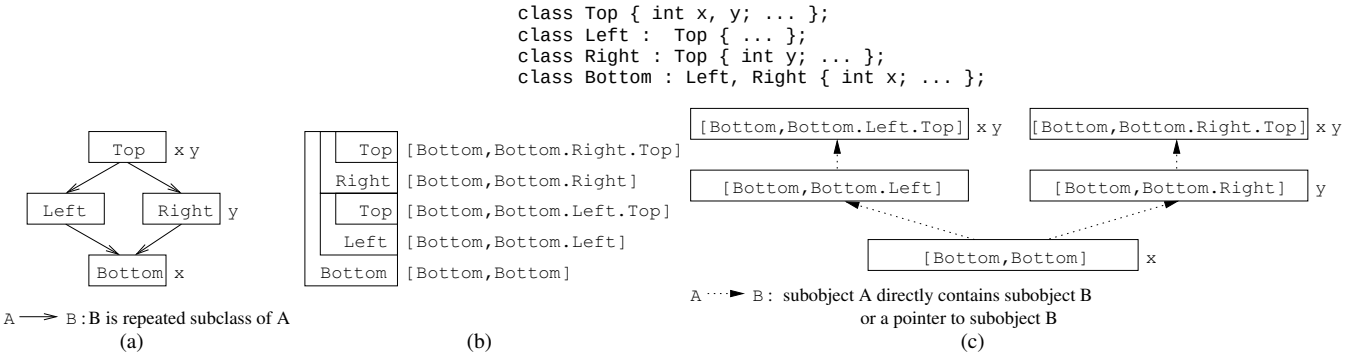

Figure 1. The repeated diamond

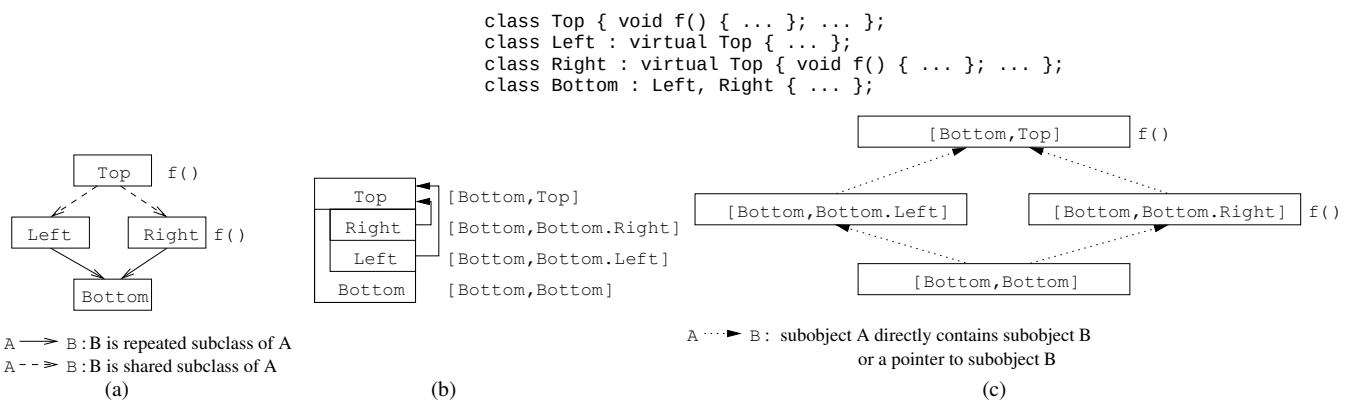

Figure 2. The shared diamond

tains three definitions of `y` in [`Bottom`,`Bottom.Right`], [`Bottom`,`Bottom.Right.Top`], and [`Bottom`,`Bottom.Left.Top`]. As there is no unique dominant definition of `y` here, a field access `(new Bottom())->y` is ambiguous.

2.3 Casts in C++

C++ has three cast operators for traversing class hierarchies, each of which has significant limitations[4]. Most commonly used are so-called C-style casts. C-style casts may be used to cast between arbitrary unrelated types, although some static checking is performed on up-casts (e.g., a C-style up-cast is statically rejected if the receiver's static type does not contain a unique subobject whose static class is the type being casted to), but no runtime checks. C-style casts cannot be used to down-cast along a shared inheritance relation, as it is not possible to "go back" along the indirection pointers in the object. When used incorrectly, C-style casts may cause run-time errors.

The `static_cast` operator only performs compile-time checks (e.g., to ensure that a unique subobject of the target type exists) and disallows casting between unrelated types. `static_cast` cannot be used to down-cast along a shared inheritance relation. When used incorrectly, `static_cast` may cause run-time errors.

The `dynamic_cast` operator is the recommended cast operator in C++. It has the desirable property that failing casts result in controlled exceptions (when the target of the cast is a reference) or the special value `NULL` (when the target is a pointer). Unlike the previous two operators, down-casting along shared inheritance relations is allowed, and `dynamic_cast` may be used to cast between unrelated types. However, a subtle limitation exists: A `dynamic_cast` is statically incorrect when applied to an expression whose declared type does not declare virtual methods.

In the semantics, we implemented two different casting operators: a static type safe casting operator analogously to `static_cast` and a generalization on `dynamic_cast` that is not restricted to casting types with declared virtual methods. It would be simple to add this restriction to our type system but this would weaken our type soundness result, which is completely independent of this matter.

2.4 Examples

We will now discuss several examples to illustrate the subtleties that arise in the C++ inheritance model.

Example 1. Dynamic dispatch behavior can be counterintuitive in the presence of multiple inheritance. One might expect a method call always to dispatch to a method definition in a superclass or subclass of the type of the receiver expression. Consider, however, the "shared diamond" example of Fig. 2, where a method `f()` is defined in classes `Right` and `Top`. Now assume that the following C++ code is executed (note the implicit up-cast to `Left` in the assignment):

```
Left* b = new Bottom(); b->f();
```

One might expect the method call to dispatch to `Top::f()`. But in fact it dispatches to `f()` in class `Right`, which is neither a superclass nor a subclass of `Left`. The reason is that up-casts do not switch off dynamic dispatch, which is based on the receiver object's dynamic class. The dynamic class of `b` remains `Bottom`

[4] The remaining two cast operators in C++, `const_cast` and `reinterpret_cast` are irrelevant for the issues studied in this paper.

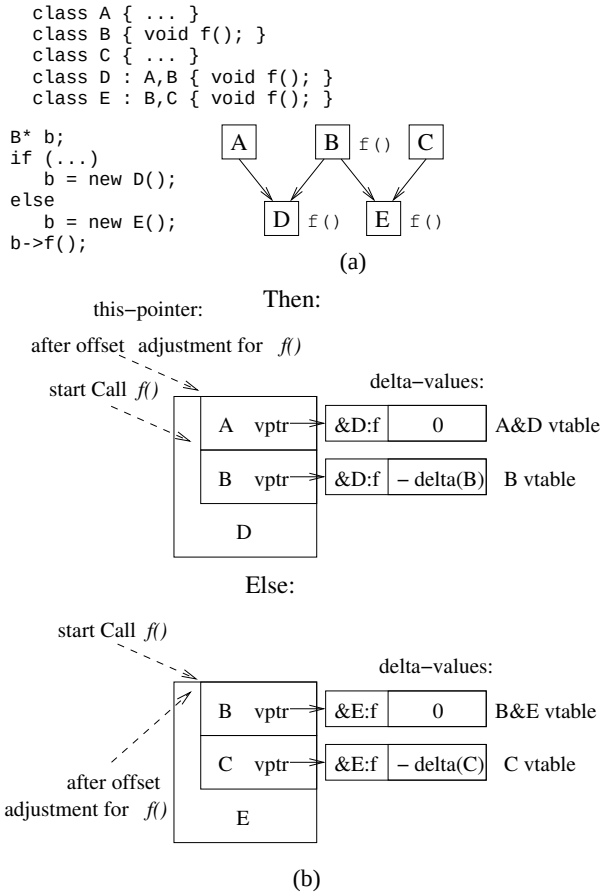

Figure 3. C++ fragment demonstrating dynamically varying subobject context

after the cast, and since `Right::f()` dominates `Top::f()`, the former is called.

This makes sense from an application viewpoint: Imagine the top class to be a "Window", the left class to be a "Window with menu", the right class to be a "Window with border", the bottom class to be a "Window with border and menu", and `f()` to compute the available window space. Then, a "Window with border and menu" object which is casted to "Window with menu" pretends not to have a border anymore (border methods cannot be called). But for the area computation, the hidden border must be taken into account, thus `f()` from "Window with border" must be called.

Example 2. The next example illustrates the need to track some subobject information at run-time, and how this complicates the semantics. Consider the program fragment in Fig. 3(a), where `b` points to a `B`-subobject. This subobject occurs in two different "contexts", namely either as a `[D, D.B]` subobject (if the then-case of the `if` statement is executed), or as an `[E, E.B]` subobject (if the else-case is executed). Note that executing the assignments `b = new D()` and `b = new E()` involves an implicit up-cast to type `B`. Depending on the context, the call `b->f()` will dispatch to `D::f()` or `E::f()`. Now, executing the body of this `f()` involves an implicit assignment of `b` to its `this` pointer. Since the static type of `b` is `B`, and the static type of `this` is the class containing its method, an implicit down-cast (to `D` or to `E`, depending on the context) is needed. At compile time it is not known which cast will happen at run-time, which implies that the compiler must keep track of some additional information to determine the cast that must be performed.

In a typical C++ implementation, a cast actually implies changing the pointer value in the presence of multiple inheritance, as is illustrated in Fig. 3(b). The up-cast from `D` to `B` (then-case, upper part of Fig. 3(b)) is implemented by adding the offset $delta(\texttt{B})$ of the `[D,D.B]`-subobject within the `D` object to the pointer to the `D` object. Afterwards, the pointer points to the `[D,D.B]`-subobject. As we discussed, the subsequent call `b->f()` requires that the pointer be down-casted to `D` again. This cast is implemented by adding the negative offset $-delta(\texttt{B})$ of the `[D,D.B]`-subobject to the pointer. The else-case (lower part of Fig. 3(b)) is analogous, but involves a different offset, which happens to be 0. In other words, the offsets in the then- and else-cases are different, and we do not know until run-time which offset has to be used. To this end, C++ compilers typically extend the virtual function table (vtable) [27] with "delta" values, that, for each vtable entry, record the offset that has to be added to the `this`-pointer in order to ensure that it points to the correct subobject after the cast (Fig. 3(b), left part).[5]

Our semantics correctly captures the information needed for performing casts, without referring to compiler data structures such as vtable entries and offsets.

Example 3. The following example shows how C++ resolves ambiguities by exploiting static types. In the "repeated diamond" of Fig. 1, let us assume that we have declared a method `f()` in class `Top`, and execute the following code:

```
Left* b = new Bottom(); b->f();
```

Note that the assignment performs an implicit up-cast to type `Left`, and that the method call is statically correct because a single definition of `f()` is visible.

However, at run-time the dynamic class of the subobject [`Bottom`,`Bottom.Left`] associated with `b` is used to resolve the dynamic dispatch. The dynamic class of `b` is `Bottom`, and `b` has *two* `Top` subobjects containing `f` (and `x`). As neither definition of `f()` dominates the other, the call to `b->f()` appears to be ambiguous.

Note that the code for `f` exists only once, but this code will be called with an ambiguous `this`-pointer at run-time: is it the one pointing to the [`Bottom`, `Bottom.Left.Top`] subobject, or the one pointing to the [`Bottom`,`Bottom.Right.Top`] subobject? Each of these subobject has its own field `x`, and these `x`'s may have different values at run-time when referenced by `f()`, leading to ambiguous program behavior.

C++ uses the static type of `b` to resolve the ambiguity and generate a unique vtable entry for `f()`. As `b`'s static type is `Left`, the "delta" part of the vtable entry will cause the dynamic object of type `Bottom` (and thus the `this`-pointer) to be cast to [`Bottom`,`Bottom.Left.Top`], and *not* to [`Bottom`,`Bottom.Right.Top`].

While this may seem to be a "natural" way to resolve the ambiguity, it makes the result of dynamic dispatch—which, intuitively, is based *solely* on an object's *dynamic* type—additionally dependent on the object's static type. During the evolution of our semantics, for a long time we considered this a flaw in the design of C++, and our first semantics [36] (for a language then called C+) did not resolve the ambiguity using the static type, but threw an exception instead. This viewpoint was inspired by Rossie and Friedman, who also considered this situation to be ambiguous. Now we stick exactly to C++, even though this makes the semantics more complex (see discussion in §8).

[5] An alternative to delta entries in vtables are so-called "trampolines", which use additional machine code for pointer adjustment.

Example 4. C++ allows method overriding with *covariant* (i.e. more specific) return types. Unrestricted covariance can however lead to ambiguities. In the context of the repeated diamond of Fig. 1, consider:

```
class A { Top* f(); }
class B : A { Bottom* f(); } //not allowed

A* a = new B();
Top* t = a->f();
```

Statically, everything seems fine: because the type of `a` is `A`, the type of `a->f()` is `Top`. However, if we allowed the redefinition of `f()`, at run-time `a->f()` evaluates to a `Bottom` object. C++ implicitly casts to the return type of the statically selected method (which would be `Top`); but this cast is ambiguous, as a `Bottom` object has two different `Top` subobjects in the repeated diamond. Hence this redefinition is statically incorrect. C++ requires *unique covariance*: if the return type of the statically selected method is C and the return type of the dynamically selected one is D, then there must exist a unique path from D back to C.

Example 5. C++ does not allow method overriding with *contravariant* (i.e. less specific) parameter types, and one reason for this is again the possibility of ambiguities. In the context of the repeated diamond of Fig. 1, consider:

```
class A { void f(Left* l); }
class B : A { void f(Top* t); } //no redefinition
                                //in C++!
A* a = new B();
a->f(new Bottom());
```

Here, the actual parameter must be cast from `Bottom` to `Top`, but again this cast is ambiguous.

Clearly, the semantics of method calls, field accesses, and casts are quite complicated in the presence of shared and repeated multiple inheritance. Typical C++ compilers rely on implementation-level artifacts such as v-tables and subobject offsets to define the behavior of these constructs. We will now present a formalization that relies solely on subobjects and paths, which enables us to demonstrate type-safety.

3. Formalization

Our semantics builds on the multiple inheritance calculus developed by Rossie and Friedman [20], but goes well beyond that work by providing an executable semantics and a type-safety proof. Rossie and Friedman merely provide the subobject model but no programming language, they do not model casts and their notion of method dispatch does not model C++ precisely (see Example 3 above).

The starting point for our formal semantics was Jinja [10], a model of a Java-like language defined in higher-order logic (HOL) in the theorem prover Isabelle/HOL. However, because of the many intricacies of C++, CoreC++ has really outgrown its parent. As an indicator for this see the fact that the size of the formal specification and associated proofs more than doubled.

Our meta-language HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types with their primitive operations.

3.1 Basic notation — The meta language

Types include the basic types of truth values, natural numbers and integers, which are called *bool*, *nat*, and *int* respectively. The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t::\tau$ means that HOL term t has HOL type τ.

Pairs come with the two projection functions $fst :: 'a \times 'b \Rightarrow 'a$ and $snd :: 'a \times 'b \Rightarrow 'b$. We identify tuples with pairs nested to the right: (a, b, c) is identical to $(a, (b, c))$ and $'a \times 'b \times 'c$ is identical to $'a \times ('b \times 'c)$.

Sets (type $'a$ *set*) follow the usual mathematical convention.

Lists (type $'a$ *list*) come with the empty list $[]$, the infix constructor $\cdot$, the infix @ that appends two lists, and the conversion function *set* from lists to sets. Variable names ending in "s" usually stand for lists and $|xs|$ is the length of *xs*. The standard function *map*, which maps a function to every element in a list, is also available.

Function update is defined as follows:
$f(a := b) \equiv \lambda x.$ *if* $x = a$ *then* b *else* $f\,x$
where $f :: 'a \Rightarrow 'b$ and $a :: 'a$ and $b :: 'b$.

datatype $'a$ *option* $=$ *None* $|$ *Some* $'a$
adjoins a new element *None* to a type $'a$. All existing elements in type $'a$ are also in $'a$ *option*, but are prefixed by *Some*. For succinctness we write $\lfloor a \rfloor$ instead of *Some a*. Hence *bool option* has the values $\lfloor True \rfloor$, $\lfloor False \rfloor$ and *None*.

Partial functions are modeled as functions of type $'a \Rightarrow 'b$ *option*, where *None* represents undefinedness and $f\,x = \lfloor y \rfloor$ means x is mapped to y. Instead of $'a \Rightarrow 'b$ *option* we write $'a \rightharpoonup 'b$, call such functions **maps**, and abbreviate $f(x:=\lfloor y \rfloor)$ to $f(x \mapsto y)$. The latter notation extends to lists: $f([x_1,\ldots,x_m]\ [\mapsto]\ [y_1,\ldots,y_n])$ means $f(x_1 \mapsto y_1)\ldots(x_i \mapsto y_i)$, where i is the minimum of m and n. The notation works for arbitrary list expressions on both sides of $[\mapsto]$, not just enumerations. Multiple updates like $f(x \mapsto y)(xs\,[\mapsto]\,ys)$ can be written as $f(x \mapsto y,\ xs\ [\mapsto]\ ys)$. The map $\lambda x.$ *None* is written *empty*, and *empty*$(\ldots)$, where $\ldots$ are updates, abbreviates to $[\ldots]$. For example, *empty*$(x \mapsto y,\ xs[\mapsto]ys)$ becomes $[x \mapsto y,\ xs\ [\mapsto]\ ys]$. The domain of a map is defined as $dom\ m \equiv \{a \mid m\ a \neq None\}$. Function *map-of* turns an list of pairs into a map:

map-of $[] = empty$
map-of $(p \cdot ps) = map\text{-}of\ ps(fst\ p \mapsto snd\ p)$

3.2 Names, paths, and base classes

Type *cname* is the (HOL) type of class names. The (HOL) variables *C* and *D* will denote class names, *Cs* and *Ds* are paths. We introduce the type abbreviation

$$path = cname\ list$$

Programs are denoted by *P*. For the moment we do not need to know what programs look like. Instead we assume the following predicates describing the class structure of a program:

- $\boldsymbol{P} \vdash \boldsymbol{C} \prec_R \boldsymbol{D}$ means D is a direct repeated base class of C.
- $\boldsymbol{P} \vdash \boldsymbol{C} \prec_S \boldsymbol{D}$ means D is a direct shared base class of C.
- $\preceq^*$ means $(\prec_R \cup \prec_S)^*$.
- ***is-class P C*** means class C is defined in *P*.

3.3 Subobjects

We slightly change the appearance of subobjects in comparison with Rossie-Friedman style: we use a tuple with a class and a path component where a path is represented as a list of classes. So e.g., a Rossie-Friedman subobject `[Bottom,Bottom.Left]` is translated into `(Bottom,[Bottom,Left])`.

The subobject definitions are parameterized by a program *P*. First we define $Subobjs_R\ P$, the subobjects whose path consists only of repeated inheritance relations:

$$\frac{is\text{-}class\ P\ C}{(C, [C]) \in Subobjs_R\ P}$$

$$\frac{P \vdash C \prec_R D \qquad (D, Cs) \in Subobjs_R\ P}{(C, C \cdot Cs) \in Subobjs_R\ P}$$

Now we define $Subobjs\ P$, the set of all subobjects:

$$\frac{(C, Cs) \in Subobjs_R\ P}{(C, Cs) \in Subobjs\ P}$$

$$\frac{P \vdash C \preceq^* C' \qquad P \vdash C' \prec_S D \qquad (D, Cs) \in Subobjs_R\ P}{(C, Cs) \in Subobjs\ P}$$

We have shown that this definition and the one by Rossie and Friedman (see §2.2) are equivalent. Ours facilitates proofs because paths are built up following the inductive nature of lists.

3.4 Path functions

Function *last* on lists returns the topmost class in a path (w.r.t. the class hierarchy), *butlast* chops off the last element.

Function $@_p$ appends two paths assuming the second one is starting where the first one ends with. If the second path only contains repeated inheritance, it starts with the same class the first one ends, so we can append both of them via @ (taking care to just use the common class once). If the second path begins with a shared class, the first path just disappears (because we lose all information below the shared class):

$Cs\ @_p\ Cs' \equiv$ *if last* $Cs = hd\ Cs'$ *then* $Cs\ @\ tl\ Cs'$ *else* Cs'

The following property holds under the assumption that program P is well-formed.

If $(C, Cs) \in Subobjs\ P$ and $(last\ Cs, Ds) \in Subobjs\ P$
then $(C, Cs\ @_p\ Ds) \in Subobjs\ P$.

A well formed program requires certain natural constraints of the program such as the class hierarchy relation to be irreflexive.

An ordering on paths is defined as follows:

$$\frac{(C, Cs) \in Subobjs\ P \qquad (C, Ds) \in Subobjs\ P \qquad Cs = butlast\ Ds}{P,C \vdash Cs \sqsubset^1 Ds}$$

$$\frac{(C, Cs) \in Subobjs\ P \qquad P \vdash last\ Cs \prec_S D}{P,C \vdash Cs \sqsubset^1 [D]}$$

The reflexive and transitive closure of $\sqsubset^1$ is written $\sqsubseteq$. The intuition of this ordering is subobject containment: $P,C \vdash Cs \sqsubseteq Ds$ means that subobject (C,Ds) lies below (C,Cs) in the subobject graph.

4. Abstract syntax of CoreC++

We do not define a concrete syntax for CoreC++, just an abstract syntax. The translation of the C++-subset corresponding to CoreC++ into abstract syntax is straightforward and will not be discussed here.

In the sequel, we use the following (HOL) variable conventions: V is a (CoreC++) variable name, F a field name, M a method name, e an expression, v a value, and T a type.

In addition to *cname* (class names) there are also the (HOL) types *vname* (variable and field names), and *mname* (method names). We do not assume that these types are disjoint.

4.1 References

A **reference** refers to a subobject within an object. Hence it is a pair of an **address** that identifies the object on the heap (see §6.1 below) and a path identifying the subobject. Formally:

$$reference = addr \times path$$

The path represents the dynamic context of a subobject as a result of previous casts (as explained in §2.4), and corresponds to the result of adding "delta" values to an object pointer in the standard "vtable" implementation. Note that our semantics does not emulate the standard implementation, but is more abstract.

Note: CoreC++ references are not equivalent to C++ references, but are more like C++ pointers.

As an example, consider Fig. 3. Let us assume that the `else` statement is executed, then `b` will have the reference value $(a, [E, B])$ where a is the memory address of the new E object, and path $[E, B]$ represents the fact that this object has been up-cast to B and `b` in fact points to the B subobject.

4.2 Values and Expressions

A CoreC++ **value** (abbreviated *val*) can be

- a boolean *Bool b*, where $b :: bool$, or
- an integer *Intg i*, where $i :: int$, or
- a reference *Ref r*, where $r :: reference$, or
- the null reference *Null*, or
- the dummy value *Unit*.

CoreC++ is an imperative but an expression-based language where statements are expressions that evaluate to *Unit*. The following **expressions** (of HOL type *expr*) are supported by CoreC++:

- creation of new object: `new` C
- static casting: `stat_cast` $C\ e$
- dynamic casting: `dyn_cast` $C\ e$
- literal value: `Val` v
- binary operation: $e_1 \ll bop \gg e_2$ (where *bop* is one of + or =)
- variable access `Var` V and variable assignment V `:=` e
- field access $e.F\{Ds\}$ and field assignment $e_1.F\{Ds\}$ `:=` e_2
  (where Ds is the path to the subobject where F is declared)
- method call: $e.M(es)$
- block with locally declared variable: $\{V{:}T; e\}$
- sequential composition: e_1 `;` e_2
- conditional: `if` $(e)\ e_1$ `else` e_2
  (do not confuse with HOL's *if b then x else y*)
- while loop: `while` $(e)\ e'$

The constructors `Val` and `Var` are needed in our meta-language to disambiguate the syntax. There is no return statement because everything is an expression and returns a value.

The annotation $\{Ds\}$ in field access and assignment is not part of the input language but is something that a preprocessor, e.g., the type checking phase of a compiler, must add.

To ease notation we introduce an abbreviation:

$$ref\ r \quad \equiv \quad \texttt{Val}(Ref\ r)$$

4.3 Programs

The abstract syntax of programs is given by the type definitions in Fig. 4, where *ty* is the HOL type of CoreC++ types.

A CoreC++ program is a list of class declarations. A **class declaration** consists of the name of the class and the class itself. A **class** consists of the list of its direct superclass names (marked shared or repeated), a list of field declarations and a list of method declarations. A **field declaration** is a pair of a field name and its type. A **method declaration** consists of the method name and the method itself, which consists of the parameter types, the result type, the parameter names, and the method body.

Note that CoreC++ (like Java, but unlike C++) does not have global variables. Method bodies can access only their *this*-pointer and parameters, and return a value.

We refrain from showing the formal definitions (see [10]) of the predicates like $P \vdash C \prec_R D$ introduced in §3 as they are straightforward. Instead we introduce one more access function:

- ***class P C***: the class (more precisely: *class option*) associated with C in P.

$$
\begin{array}{lcllcl}
\mathit{prog} & = & \mathit{cdecl\ list} & \mathit{cdecl} & = & \mathit{cname} \times \mathit{class} \\
\mathit{class} & = & \mathit{base\ list} \times \mathit{fdecl\ list} \times \mathit{mdecl\ list} & \mathit{fdecl} & = & \mathit{vname} \times \mathit{ty} \\
\mathit{method} & = & \mathit{ty\ list} \times \mathit{ty} \times \mathit{vname\ list} \times \mathit{expr} & \mathit{mdecl} & = & \mathit{mname} \times \mathit{method} \\
\textbf{datatype}\ \mathit{base} & = & \mathit{Repeats\ cname} \mid \mathit{Shares\ cname} & & &
\end{array}
$$

Figure 4. Abstract program syntax

5. Type system

CoreC++ types are either primitive (*Boolean* and *Integer*), class types *Class C*, *NT* (the type of *Null*), or *Void* (the type of *Unit*). The set of these types (i.e. the corresponding HOL type) is called *ty*. The first two rules of the subtype relation $\leq$ are straightforward:

$$P \vdash T \leq T \quad P \vdash NT \leq \mathit{Class}\ C$$

To relate two classes, we have to take care that we can use an object of the smaller type wherever an object of the more general type can occur. This property can be guaranteed by requiring that a static cast between these two types can be performed, resulting in the premise[6]:

$$P \vdash \mathit{path}\ C\ \mathit{to}\ D\ \mathit{unique} \equiv \exists!Cs.\ (C, Cs) \in \mathit{Subobjs}\ P \wedge \mathit{last}\ Cs = D$$

This property ensures that the path from class *C* leading to class *D* exists and is unique ($\exists!$ is unique existence).

This leads to the third subtyping rule:

$$\frac{P \vdash \mathit{path}\ C\ \mathit{to}\ D\ \mathit{unique}}{P \vdash \mathit{Class}\ C \leq \mathit{Class}\ D}$$

The pointwise extension of $\leq$ to lists is written $[\leq]$.

5.1 Typing rules

The core of the type system is the judgment $P,E \vdash e :: T$, where E is an **environment**, i.e. a map from variables to their types. We call T the **static** type of e.

We will discuss the typing rules (see Fig. 5) construct by construct, concentrating on object-orientation. The remaining rules can be found elsewhere [10]. For critical constructs we will also consider the question of type safety: does the type system guarantee that evaluation cannot get stuck and that, if a value is produced, it is of the right type.

Values are typed with their corresponding types, e.g., *Bool* as *Boolean*, *Intg* as *Integer*. However, there is no rule to type a *reference*, so *explicit references cannot be typed*. CoreC++, like Java or ML, does not allow explicit references for well known reasons.

5.1.1 Cast

Typing static casts is non-trivial in CoreC++ because the type system needs to prevent ambiguities at run-time (although it cannot do so completely). When evaluating `stat_cast` *C e*, the object that *e* evaluates to may have multiple subobjects of class *C*. If it is an up-cast, i.e. if $P,E \vdash e :: \mathit{Class}\ D$ and *D* is a subclass of *C*, we have to check if there is a unique path from *D* to *C*.

Two examples will make this clearer: if we want to cast `Bottom` to `Top` in the repeated diamond in Fig. 1, we have two paths leading to possible subobjects: [`Bottom`,`Left`,`Top`] and [`Bottom`,`Right`,`Top`]. So there is no unique path, the cast is ambiguous and the type system rejects it. But the same cast in the shared diamond in Fig. 2 is possible, as there is only one possible path, namely [`Top`].

For down-casts we need to remember (§2.3) that we have chosen to model a type safe variant of `static_cast` (which means we throw an exception where C++ produces a run time error), for which C++ has fixed the rules as follows:: down-casts may only involve repeated inheritance. To enforce this restriction we introduce the predicate

$$P \vdash \mathit{path}\ C\ \mathit{to}\ D\ \mathit{via}\ Cs \equiv (C, Cs) \in \mathit{Subobjs}\ P \wedge \mathit{last}\ Cs = D$$

Combining the checks for up- and down-casts in one rule and requiring the class to be known we obtain WT1 (see Fig. 5). Remember that $(C, Cs) \in \mathit{Subobjs}_R\ P$ means that *Cs* involves only repeated inheritance.

As an example of an ambiguous down-cast, take the repeated diamond in Fig. 1 and extend it with a shared superclass `C` of `Top`. Casting a `Bottom` object of a static class `C` to `Top` is ambiguous because there are two `Top` subobjects.

Dynamic casts are non-trivial operations at run-time but statically they are as simple as can be: rule WT2 only requires that the expression is well-typed and the class is known. This liberality is not just admissible (because dynamic casts detect type mismatches at run-time) but even necessary. We come back to this point when we discuss the semantics in §6.3.2.

5.1.2 Variable assignment and binary operators

The assignment rule WT3 is completely straightforward as the expression on the right hand side has to be a subtype of the variable type on the left hand side, which we get by consulting the typing environment.

Rule WT4 for binary operators: Addition is unsurprising. In the equality test, we assume that both operands have the same type, i.e. that all necessary casts are performed explicitly. This simplifies the presentation without loss of generality.

5.1.3 Field access and assignment

The typing rule for field access WT5 is straightforward. It can either be seen as a rule that takes an expression where field access is already annotated (by $\{Cs\}$), and the rule merely checks that the annotation is correct. Or it can be seen as a rule for computing the annotation. The latter interpretation relies on the fact that predicate $P \vdash C\ \mathit{has\ least}\ F : T\ \mathit{via}\ Cs$ can compute *T* and *Cs* from *P*, *C* and *F*. So it remains to explain $P \vdash C\ \mathit{has\ least}\ F : T\ \mathit{via}\ Cs$: it checks if *Cs* is the least (w.r.t. $\sqsubseteq$) path leading from *C* to a class that declares an *F*. First we define the set *FieldDecls P C F* of all (Cs, T) such that *Cs* is a valid path leading to a class with an *F* of type *T*:

$$
\begin{array}{l}
\mathit{FieldDecls}\ P\ C\ F \equiv \\
\{(Cs, T) \mid \\
\ (C, Cs) \in \mathit{Subobjs}\ P \wedge \\
\ (\exists\ Bs\ fs\ ms.\ \mathit{class}\ P\ (\mathit{last}\ Cs) = \lfloor(Bs, fs, ms)\rfloor \wedge \mathit{map\text{-}of}\ fs\ F = \lfloor T \rfloor)\}
\end{array}
$$

Then we select a least element from that set:

$$
\begin{array}{l}
P \vdash C\ \mathit{has\ least}\ F : T\ \mathit{via}\ Cs \equiv \\
(Cs, T) \in \mathit{FieldDecls}\ P\ C\ F \wedge \\
(\forall\ (Cs', T') \in \mathit{FieldDecls}\ P\ C\ F.\ P,C \vdash Cs \sqsubseteq Cs')
\end{array}
$$

If there is no such least path, field access is ambiguous and hence not well-typed. We give an example. Once again we concentrate on the repeated diamond in Fig. 1 and assume that a field *x* is defined in class `Bottom` and class `Top`. When type checking *e.x*, where *e* is of class `Bottom`, the path components in *FieldDecls P Bottom x* are [`Bottom`], [`Bottom`,`Left`,`Top`] and [`Bottom`,`Right`,`Top`].

[6] For more information about static casts, see §5.1.1

$$\frac{P,E \vdash e :: \mathit{Class}\ D \qquad \mathit{is\text{-}class}\ P\ C \qquad P \vdash \mathit{path}\ D\ \mathit{to}\ C\ \mathit{unique} \vee (\forall\, Cs.\ P \vdash \mathit{path}\ C\ \mathit{to}\ D\ \mathit{via}\ Cs \longrightarrow (C, Cs) \in \mathit{Subobjs}_R\ P)}{P,E \vdash \texttt{stat\_cast}\ C\ e :: \mathit{Class}\ C}\ \text{WT1}$$

$$\frac{P,E \vdash e :: \mathit{Class}\ D \qquad \mathit{is\text{-}class}\ P\ C}{P,E \vdash \texttt{dyn\_cast}\ C\ e :: \mathit{Class}\ C}\ \text{WT2}$$

$$\frac{E\ V = \lfloor T \rfloor \qquad P,E \vdash e :: T' \qquad P \vdash T' \leq T}{P,E \vdash V := e :: T}\ \text{WT3}$$

$$\frac{P,E \vdash e_1 :: T_1 \qquad P,E \vdash e_2 :: T_2 \qquad \mathit{case}\ \mathit{bop}\ \mathit{of} = \Rightarrow T_1 = T_2 \wedge T = \mathit{Boolean} \mid + \Rightarrow T_1 = \mathit{Integer} \wedge T_2 = \mathit{Integer} \wedge T = \mathit{Integer}}{P,E \vdash e_1 \ll \mathit{bop} \gg e_2 :: T}\ \text{WT4}$$

$$\frac{P,E \vdash e :: \mathit{Class}\ C \qquad P \vdash C\ \mathit{has\ least}\ F : T\ \mathit{via}\ Cs}{P,E \vdash e.F\{Cs\} :: T}\ \text{WT5}$$

$$\frac{P,E \vdash e_1 :: \mathit{Class}\ C \qquad P \vdash C\ \mathit{has\ least}\ F : T\ \mathit{via}\ Cs \qquad P,E \vdash e_2 :: T' \qquad P \vdash T' \leq T}{P,E \vdash e_1.F\{Cs\} := e_2 :: T}\ \text{WT6}$$

$$\frac{P,E \vdash e :: \mathit{Class}\ C \qquad P \vdash C\ \mathit{has\ least}\ M = (Ts, T, m)\ \mathit{via}\ Cs \qquad P,E \vdash es\ [::]\ Ts' \qquad P \vdash Ts'\ [\leq]\ Ts}{P,E \vdash e.M(es) :: T}\ \text{WT7}$$

Figure 5. The typing rules

The least element of the path components in this set is `[Bottom]`, so the x in class `Bottom` will be accessed. Note that if no x in `Bottom` is declared, then there is no element with a least path in *FieldDecls* and the field access is ambiguous and hence illegal.

Field assignment works analogously as shown in WT6.

The following example [19] shows that many compilers treat dominance incorrectly and thus have problems with field access/ assignment (as well as method call):

```
class A { int x; }
class B { int x; }
class C : virtual A, virtual B { int x; }
class D : virtual A, virtual B, C {}

(new D())->x = 42;
```

Both g++ and the Intel compiler reject the left hand side of `(new D())->x = 42` as ambiguous while our type system correctly says that the least declaration of `x` is the one in `C`.

5.1.4 Method call

In the call typing rule WT7 the class C of e is used to collect all declarations of M and select the least one. The set of all definitions of method M from class C upwards is defined as

$$\begin{array}{l}\mathit{MethodDefs}\ P\ C\ M \equiv \\ \{(Cs, \mathit{mthd}) \mid \\ (C, Cs) \in \mathit{Subobjs}\ P \wedge \\ (\exists\, Bs\ fs\ ms.\ \mathit{class}\ P\ (\mathit{last}\ Cs) = \lfloor (Bs, fs, ms) \rfloor \wedge \mathit{map\text{-}of}\ ms\ M = \lfloor \mathit{mthd} \rfloor)\}\end{array}$$

This set pairs the method (of type *method*, see Fig. 4) with the path Cs leading to the defining class. Among all definitions the least one (w.r.t. the ordering on paths) is selected:

$$\begin{array}{l}P \vdash C\ \mathit{has\ least}\ M = \mathit{mthd}\ \mathit{via}\ Cs \equiv \\ (Cs, \mathit{mthd}) \in \mathit{MethodDefs}\ P\ C\ M \wedge \\ (\forall\, (Cs', \mathit{mthd}') \in \mathit{MethodDefs}\ P\ C\ M.\ P,C \vdash Cs \sqsubseteq Cs')\end{array}$$

Unfortunately, the absence of static ambiguity of method lookup is not sufficient to avoid ambiguities at run-time. Even if the call is well-typed, e may evaluate to a class below C from which there is no least declaration of M. We presented this problem in Example 3 and will discuss it in detail in §6.3.6.

In the third premise of WT7, the relation $[::]$ is the pointwise extension of $::$ to lists.

$$\begin{array}{lcl}\mathit{state} & = & \mathit{heap} \times \mathit{locals} \\ \mathit{locals} & = & \mathit{vname} \rightharpoonup \mathit{val} \\ \mathit{heap} & = & \mathit{addr} \rightharpoonup \mathit{obj} \\ \mathit{obj} & = & \mathit{cname} \times \mathit{subo\ set} \\ \mathit{subo} & = & \mathit{path} \times (\mathit{vname} \rightharpoonup \mathit{val})\end{array}$$

Figure 6. The type of CoreC++ program states

5.2 Well-formed programs

A well-formed CoreC++ program (*wf-C-prog P*) must obey all the usual requirements (every method body is well-typed and of the declared result type, the class hierarchy is acyclic, etc — for details see [10]). Additionally, there are CoreC++-specific conditions concerning method overriding:

(i) covariance in the result type combined with the uniqueness of paths from the new result class to *all* result classes in previous definitions of the same method (see Example 4). This requirement is easily formalized by means of the *path-unique* predicate introduced in §5.

(ii) invariance in the argument types (see Example 5)

(iii) for every method definition a class C sees via path Cs, the corresponding subobject (C,Cs) must have a least overrider as explained in §6.3.6 (otherwise the corresponding C++ program would not be able to construct a unique vtable entry for this method call and the program would be rejected at compile time)

6. Big Step Semantics

The big step semantics is a (deterministic) relation between an initial expression-state pair $\langle e,s \rangle$ and a final expression-state pair $\langle e',s' \rangle$. The syntax of the relation is $P,E \vdash \langle e,s \rangle \Rightarrow \langle e',s' \rangle$ and we say that e *evaluates* to e'. The rules will be such that *final* expressions are always values (`Val`) or exceptions (`throw`), i.e. final expressions are completely evaluated.

6.1 State

The set of states is defined in Fig. 6. A **state** is a pair of a **heap** and a **store** (*locals*). A store is a map from variable names to values. A heap is a map from addresses to objects. An **object** is a pair of a class name and its subobjects. A **subobject** (*subo*) is a pair of a

$$\frac{\textit{new-Addr}\ h = \lfloor a \rfloor \qquad h' = h(a \mapsto (C, \textit{init-obj}\ P\ C))}{P,E \vdash \langle \texttt{new}\ C,(h, l)\rangle \Rightarrow \langle \textit{ref}\ (a, [C]),(h', l)\rangle}\ \text{BS1}$$

$$\frac{P,E \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a, Cs),s_1\rangle \qquad P \vdash \textit{path last}\ Cs\ \textit{to}\ C\ \textit{via}\ Cs' \qquad Ds = Cs\ @_p\ Cs'}{P,E \vdash \langle \texttt{stat\_cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a, Ds),s_1\rangle}\ \text{BS2}$$

$$\frac{P,E \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a, Cs\ @\ [C]\ @\ Cs'),s_1\rangle \qquad \textit{is-class}\ P\ C \qquad C \notin \textit{set}\ Cs'}{P,E \vdash \langle \texttt{stat\_cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a, Cs\ @\ [C]),s_1\rangle}\ \text{BS3}$$

$$\frac{P,E \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a, Cs),(h, l)\rangle \qquad h\ a = \lfloor (D, \_)\rfloor \qquad P \vdash \textit{path}\ D\ \textit{to}\ C\ \textit{via}\ Cs'}{P,E \vdash \langle \texttt{dyn\_cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a, Cs'),(h, l)\rangle}\ \text{BS4}$$

$$\frac{\begin{array}{c}P,E \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a, Cs),(h, l)\rangle \\ h\ a = \lfloor (D, S)\rfloor \qquad \neg\, P \vdash \textit{path}\ D\ \textit{to}\ C\ \textit{unique} \qquad \neg\, P \vdash \textit{path last}\ Cs\ \textit{to}\ C\ \textit{unique} \qquad C \notin \textit{set}\ Cs \vee \neg\ \textit{distinct}\ Cs\end{array}}{P,E \vdash \langle \texttt{dyn\_cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{null},(h, l)\rangle}\ \text{BS5}$$

$$\frac{P,E \vdash \langle e,s_0\rangle \Rightarrow \langle \texttt{Val}\ v,(h, l)\rangle \qquad E\ V = \lfloor T \rfloor \qquad P \vdash T\ \textit{casts}\ v\ \textit{to}\ v' \qquad l' = l(V \mapsto v')}{P,E \vdash \langle V\ \texttt{:=}\ e,s_0\rangle \Rightarrow \langle \texttt{Val}\ v',(h, l')\rangle}\ \text{BS6}$$

$$\frac{P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle \texttt{Val}\ v_1,s_1\rangle \qquad P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle \texttt{Val}\ v_2,s_2\rangle \qquad \textit{binop}\ (bop, v_1, v_2) = \lfloor v \rfloor}{P,E \vdash \langle e_1 \ll bop \gg e_2,s_0\rangle \Rightarrow \langle \texttt{Val}\ v,s_2\rangle}\ \text{BS7}$$

$$\frac{P,E \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a, Cs'),(h, l)\rangle \qquad h\ a = \lfloor (D, S)\rfloor \qquad Ds = Cs'\ @_p\ Cs \qquad (Ds, fs) \in S \qquad fs\ F = \lfloor v \rfloor}{P,E \vdash \langle e.F\{Cs\},s_0\rangle \Rightarrow \langle \texttt{Val}\ v,(h, l)\rangle}\ \text{BS8}$$

$$\frac{\begin{array}{c}P,E \vdash \langle e_1,s_0\rangle \Rightarrow \langle \textit{ref}\ (a, Cs'),s_1\rangle \qquad P,E \vdash \langle e_2,s_1\rangle \Rightarrow \langle \texttt{Val}\ v,(h_2, l_2)\rangle \\ h_2\ a = \lfloor (D, S)\rfloor \qquad P \vdash \textit{last}\ Cs'\ \textit{has least}\ F : T\ \textit{via}\ Cs \qquad P \vdash T\ \textit{casts}\ v\ \textit{to}\ v' \qquad Ds = Cs'\ @_p\ Cs \\ (Ds, fs) \in S \qquad fs' = fs(F \mapsto v') \qquad S' = S - \{(Ds, fs)\} \cup \{(Ds, fs')\} \qquad h_2' = h_2(a \mapsto (D, S'))\end{array}}{P,E \vdash \langle e_1.F\{Cs\}\ \texttt{:=}\ e_2,s_0\rangle \Rightarrow \langle \texttt{Val}\ v',(h_2', l_2)\rangle}\ \text{BS9}$$

$$\frac{\begin{array}{c}P,E \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a, Cs),s_1\rangle \qquad P,E \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle \textit{map}\ \texttt{Val}\ vs,(h_2, l_2)\rangle \\ h_2\ a = \lfloor (C, \_)\rfloor \qquad P \vdash \textit{last}\ Cs\ \textit{has least}\ M = (\_, T', \_, \_)\ \textit{via}\ Ds \\ P \vdash (C, Cs\ @_p\ Ds)\ \textit{selects}\ M = (Ts, T, pns, body)\ \textit{via}\ Cs' \\ |vs| = |pns| \qquad P \vdash Ts\ \textit{Casts}\ vs\ \textit{to}\ vs' \qquad l_2' = [this \mapsto \textit{Ref}\ (a, Cs'), pns\ [\mapsto]\ vs'] \\ \textit{new-body} = (\textit{case}\ T'\ \textit{of Class}\ D \Rightarrow \texttt{stat\_cast}\ D\ body \mid \_ \Rightarrow body) \\ P,E(this \mapsto \textit{Class}\ (\textit{last}\ Cs'), pns\ [\mapsto]\ Ts) \vdash \langle \textit{new-body},(h_2, l_2')\rangle \Rightarrow \langle e',(h_3, l_3)\rangle\end{array}}{P,E \vdash \langle e.M(ps),s_0\rangle \Rightarrow \langle e',(h_3, l_2)\rangle}\ \text{BS10}$$

Figure 7. The Big Step rules

path (leading to that subobject) and a field table mapping variable names to values.

The naming convention is that h is a heap, l is a store (the *l*ocal variables), and s a state.

Note that CoreC++, in contrast to C++, does not allow stack-allocated objects: variable values can only be pointers (CoreC++ references), but not objects. Objects are only on the heap (as in Java). We do not expect stack based objects to interfere with multiple inheritance.

Remember further that a reference contains not just an address but also a path. This path selects the current subobject of an object and is modified by casts (see below).

6.2 Exceptions

CoreC++ supports exceptions. They are essential to prove type soundness as certain problems can occur at run-time (e.g., a failing cast) which we cannot prevent statically. In these cases we throw an exception so the semantics does not get stuck. Three exceptions are possible in CoreC++: *OutOfMemory*, if there is no more space on the heap, *ClassCast* for a failed cast and *NullPointer* for null pointer access. We will explain in the text exactly when an exception is thrown but will omit showing the corresponding rules; the interested reader can find them in the appendix.

6.3 Evaluation

Remember that $P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ is the evaluation judgment, where P denotes the program and E the type environment. The need for E will be explained in §6.3.3.

For a better understanding of the evaluation rules it is helpful to realize that they preserve the following heap invariant: for any object (C, S) on the heap we have

- S contains exactly the paths starting from C:
  $\{Ds \mid \exists\, fs.\ (Ds, fs) \in S\} = \{Ds \mid (C, Ds) \in \textit{Subobjs}\ P\}$,
- S is a (finite) function:
  $\forall\, (Cs,fs), (Cs',fs') \in S.\ Cs = Cs' \longrightarrow fs = fs'$

Furthermore, if an expression e evaluates to $\textit{ref}\ (a, Cs)$ then the heap maps a to $\lfloor (C, S)\rfloor$ such that

- Cs is the path of a subobject in S: $(Cs, fs) \in S$ for some fs.
- $\textit{last}\ Cs$ is equal to the class of e inferred by the type system.

We will now discuss the evaluation rules construct by construct, concentrating on object-orientation, as shown in Fig. 7. The remaining rules can be found elsewhere [10].

6.3.1 Object creation

Rule BS1 shows the big step rule for object creation. The result of evaluating `new` C is a reference $\textit{Ref}\ (a, [C])$ where a is some unallocated address returned by the auxiliary function *new-Addr* (which returns *None* if the heap is exhausted, in which case we throw an *OutOfMemory* exception). As a side effect, a is made to point to the object (C, S), where $S = \textit{init-obj}\ P\ C$ is the set of all subobjects (Cs, fs) such that $(C, Cs) \in \textit{Subobjs}\ P$ and $fs :: \textit{vname} \rightharpoonup \textit{val}$ is the field table that contains every field declared in class *last Cs* initialized with its default value (according to its type). We omit the details.

Note that C++ does not initialize fields. Our desire for type safety requires us to deviate from C++ in this minor aspect.

6.3.2 Cast

Casting is a non-trivial operation in C++, in contrast to Java. Remember that any object reference contains a path component identifying the current subobject which is referenced. A cast changes this path, thus selects a different subobject. Hence casting must adjust the path component of the reference. This mechanism corresponds to Stroustrup's adjustment of pointers by "delta" values. We consider it a prime example of the fact that our semantics does not rely on run-time data structures but on abstract concepts.

Let us first look at the static up-cast rule BS2: After evaluating e to a reference with path Cs, that path is extended (upward) by a (unique, if the the cast is well-typed, §5.1.1) path Cs' from the end of Cs up to C, which we get by predicate *path-via*. So if we want to cast `Bottom` to `Left` in the repeated diamond in Fig. 1, the appropriate path is [`Bottom`,`Left`], casting `Right` to `Top` in the shared diamond in Fig. 2 uses path [`Top`].

Rule BS3 models the static down-cast which forbids down-casts involving shared inheritance. This means that class C must occur in the path component of the reference, or the cast is "wrong". Moreover, C may not occur again further right in the path and it has to be a class in the given program P.

If neither of these two rules applies, the static cast throws a *ClassCast* exception.

Now consider `dyn_cast` which models `dynamic_cast` in C++. If possible, `dyn_cast` tries to behave like the static cast: there are two rules (not shown) that look almost like BS2 and BS3, except that they evaluate `dyn_cast` $C\ e$ rather than `stat_cast` $C\ e$. In the presence of multiple inheritance, not only up and down-casts are possible but also cross-casts: A reference $(a, [Bottom, Left])$ to the `Left` subobject of a `Bottom` object (in either the shared or repeated diamond) can be cast to the `Right` subobject resulting in the reference $(a, [Bottom, Right])$.

Luckily, dynamic up, down and sideways-casts are all subsumed by rule BS4. After evaluating e to a reference to address a, we look up the class D of the object at address a. If D has a unique C subobject, that is the one the reference must now point to.

If BS4 is inapplicable, i.e. if there is either no path or no unique path from the dynamic class, and a static cast fails as well, we return the null pointer, i.e. the value *null* (see BS5). This is exactly how C++ handles failing `dynamic_cast`s.

We now come back to the point raised in the discussion of the typing rule for dynamic casts in §5.1.1. Rule WT2 needs to be as liberal as it is because even if there is no relationship between C and the static class of e (call it B), e may evaluate to an object of a subclass of both C and B and the cast could succeed. Does that mean we should at least require that C and B have a common subclass (or maybe superclass)? Not even that: since inheritance is all about permitting later extensions with new subclasses, the common subclass of C and B need not yet exist when `dyn_cast` C e is type checked.

6.3.3 Variable assignment

Assignment is straightforward except that it requires an up-cast of the expression to the static type T of the variable. Hence we need the environment E to look up T (by $E\ V = [T]$). The up-cast is inserted implicitly by the semantics and defined via

$$\frac{\forall\, C.\ T \neq \mathit{Class}\ C}{P \vdash T\ \mathit{casts}\ v\ \mathit{to}\ v}$$

$$\frac{P \vdash \mathit{Class}\ C\ \mathit{casts}\ \mathit{Null}\ \mathit{to}\ \mathit{Null} \qquad P \vdash \mathit{path}\ \mathit{last}\ Cs\ \mathit{to}\ C\ \mathit{via}\ Cs' \qquad Ds = Cs\ @_p\ Cs'}{P \vdash \mathit{Class}\ C\ \mathit{casts}\ \mathit{Ref}\ (a, Cs)\ \mathit{to}\ \mathit{Ref}\ (a, Ds)}$$

6.3.4 Binary operators

The evaluation rule for binary operators BS7 is based on a function *binop* taking the operator and its two argument values and returning an optional (in order to deal with type mismatches) result. The definition of *binop* for our two binary operators = and + is straightforward:

$$\begin{array}{lcl} \mathit{binop}\ (=, v_1, v_2) & = & \lfloor \mathit{Bool}\ (v_1 = v_2) \rfloor \\ \mathit{binop}\ (+, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) & = & \lfloor \mathit{Intg}\ (i_1 + i_2) \rfloor \\ \mathit{binop}\ (\_, \_, \_) & = & \mathit{None} \end{array}$$

In the first equation, equality on the left hand side is the CoreC++ equality operator, equality in the middle is definitional equality, and equality on the right hand side is the test for equality. Logically, the latter two are the same.

Addition only yields a value if both arguments are integers. We could also insist on similar compatibility checks for the equality test, but that leads to excessive case distinctions that we want to avoid for reasons of presentation. In particular, = does not perform any implicit casts.

6.3.5 Field access and assignment

Let us first look at field access in rule BS8. There are two paths involved. Cs is (if the expression is well-typed, §5.1.3) the path from the class of e to the class where F is declared. Cs' is the path component of the reference that e evaluates to. As we have discussed in §6.3, *last* Cs' is equal to the static class of e. To obtain the complete path leading to the subobject in which F lives, we just have to concatenate via $@_p$ the two paths. The resulting path Ds is the path to the subobject we are looking for. If e doesn't evaluate to a reference, but to a null pointer, we throw a *NullPointer* exception.

Field assignment (rule BS9) is similar, except that we now have to update the heap at a with a new set of subobjects. The up-cast is inserted implicitly, analogously to BS6. Note that the functional nature of this set is preserved.

6.3.6 Method call

Rule BS10 is lengthy:

- evaluate e to a reference (a, Cs) and the parameter list ps to a list of values vs;
- look up the dynamic class C of the object in the heap at a;
- look up the method definition used at type checking time (*last* Cs is the static class of e) and note its return type T and the path Ds from *last* Cs to this definition;
- select the dynamically appropriate method (see below) and note its parameter names pns, parameter types Ts, body $body$, and path Cs' from C to this definition;
- check that there are as many actual as formal parameters;
- cast the parameter values vs up to their static types Ts by using $P \vdash Ts\ \mathit{Casts}\ vs\ \mathit{to}\ vs'$, the pointwise extension of casts to lists, yielding vs';
- evaluate the body (with an up-cast to T, if T is a class) in an updated type environment where *this* has type *Class* (*last* Cs') (the class where the dynamically selected method lives) and the formal parameter names have their declared types, and where the local variables are *this* and the parameters, suitably initialized.

The final store is the one obtained from the evaluation of the parameters; the one obtained from the evaluation of $body$ is discarded – remember that CoreC++ does not have global variables. If e evaluates to a null pointer, we throw a *NullPointer* exception.

Method selection is performed by the judgment $P \vdash (C, Cs)\ \mathit{selects}\ M = \mathit{mthd}\ \mathit{via}\ Cs'$, where (C,Cs) is the subobject where the method lives that was used at type checking time. Hence there is

```
class Top { void f(); }
class Right2 : Top { ... }
class Right : virtual Right2 { void f(); }
class Left : Top { void f(); }
class Bottom : Left, Right { ... }

((Right2*)(new Bottom()))->f();
```

Figure 8. Example illustrating static resolution of dynamically ambiguous method calls

at least one definition of M visible from C. There are two possible cases. If we are lucky, we can select a unique method definition based solely on C:

$$\frac{P \vdash C \text{ has least } M = mthd \text{ via } Cs'}{P \vdash (C, Cs) \text{ selects } M = mthd \text{ via } Cs'}$$

Otherwise we need static information to disambiguate the selection as Example 3 already demonstrated. To appreciate the full intricacies of this mechanism, let us consider the example in Fig. 8, where a subobject (`Bottom`,[`Right2`]) calls method f: the path components in *MethodDefs P Bottom f* are [`Bottom`,`Left`], [`Bottom`,`Left`,`Top`], [`Bottom`, `Right`] and [`Right2`,`Top`]. None of these paths is smaller than all of the others, so we cannot resolve the method call purely dynamically. So another approach is taken: we select the minimal paths in *MethodDefs P Bottom f*, which leaves us with [`Bottom`,`Left`] and [`Bottom`,`Right`]. Now we have to find out which of these two paths will select the method to call. This is done by considering the statically selected method call (i.e. the least one seen from the static class *Right2*), yielding path [`Right2`,`Top`], which is guaranteed to be unique by the type system. Now we append this 'static' path to the path component of the subobject, which results in the path where the dynamic class sees the statically selected method definition, namely [`Right2`]$@_p$[`Right2`,`Top`] = [`Right2`,`Top`]. Finally we select a path from the above set of minimal paths that is smaller than the composed path, which results in [`Bottom`,`Right`]. The uniqueness of this path is guaranteed by the well-formedness of the program (see §5.2 (iii)).

Abstractly, $P \vdash (C, Cs)$ *selects* $M = mthd$ *via* Cs' selects that Cs' from the set of minimal paths from C to definitions of M that lies on Cs, i.e. that lies below the statically selected method definition Cs. The minimal elements are collected by *MinimalMethodDefs*,

$$\begin{array}{l} MinimalMethodDefs\ P\ C\ M \equiv \\ \{(Cs, mthd) \mid \\ (Cs, mthd) \in MethodDefs\ P\ C\ M \wedge \\ (\forall\, (Cs', mthd') \in MethodDefs\ P\ C\ M.\ P,C \vdash Cs' \sqsubseteq Cs \longrightarrow Cs' = Cs)\} \end{array}$$

the ones that override the definition at Cs, i.e. are below Cs, are selected by *OverriderMethodDefs*,

$$\begin{array}{l} OverriderMethodDefs\ P\ R\ M \equiv \\ \{(Cs, mthd) \mid \\ \exists\, Cs'\ mthd'. \\ \quad P \vdash ldc\ R \text{ has least } M = mthd' \text{ via } Cs' \wedge \\ \quad (Cs, mthd) \in MinimalMethodDefs\ P\ (mdc\ R)\ M \wedge \\ \quad P,mdc\ R \vdash Cs \sqsubseteq snd\ R\ @_p\ Cs'\} \end{array}$$

and selection of a least overrider is performed as follows:

$$\begin{array}{l} P \vdash R \text{ has overrider } M = mthd \text{ via } Cs \equiv \\ (Cs, mthd) \in OverriderMethodDefs\ P\ R\ M \wedge \\ card\ (OverriderMethodDefs\ P\ R\ M) = 1 \end{array}$$

Note that *OverriderMethodDefs* returns a singleton set (card is the cardinality of a set) if the program is well-formed (see §5.2 (iii)). Hence the second defining rule for *selects* is

$$\frac{\forall\, mthd\ Cs'.\ \neg\, P \vdash C \text{ has least } M = mthd \text{ via } Cs' \qquad P \vdash (C, Cs) \text{ has overrider } M = mthd \text{ via } Cs'}{P \vdash (C, Cs) \text{ selects } M = mthd \text{ via } Cs'}$$

6.4 Small Step Semantics

Big step rules are easy to understand but cannot distinguish nontermination from being stuck. Hence we also have a *small step* semantics where expression-state pairs are gradually reduced. The reduction relation is written $P,E \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle$ and its transitive reflexive closure is $P,E \vdash \langle e,s\rangle \rightarrow^* \langle e',s'\rangle$.

We do not show the rules (for lack of space) but emphasize that we have proven the equivalence of the big and small step semantics (for well-formed programs):

$$P,E \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle = (P,E \vdash \langle e,s\rangle \rightarrow^* \langle e',s'\rangle \wedge final\ e').$$

7. Type Safety Proof

Type safety, one of the hallmarks of a good language design, means that the semantics is sound w.r.t. the type system: *well-typed expressions cannot go wrong*. Going wrong does not mean throwing an exception but arriving at a genuinely unanticipated situation. The by now standard formalization of this property [38] requires proving two properties: *progress* (well-typed expressions can be reduced w.r.t. the small step semantics if they are not final yet — the small step semantics does not get stuck) and *preservation* or *subject reduction*: reducing a well-typed expression results in another well-typed expression whose type is $\leq$ the original type.

In the remainder we concentrate on the specific technicalities of the CoreC++ type safety proof. We do not even sketch the actual proof, which is routine enough, but all the necessary invariants and notions without which the proof is very difficult to reconstruct. For a detailed exposition of the Jinja type safety proof, our starting point, see [10]. For a tutorial introduction to type safety see, for example, [18].

7.1 Run-time type system

The main complication in many type safety proofs is the fact that well-typedness w.r.t. the static type system is *not* preserved by the small step semantics. The fault does not lie with the semantics but the type system: for pragmatic reasons it requires properties that are not preserved by reduction and are irrelevant for type safety. Thus a second type system is needed which is more liberal but closed under reduction. This is known as the *run-time type system* [7] and the judgment is $P,E,h \vdash e : T$. Please note that there is no type checking at run-time: this type system is merely the formalization of an invariant which is not checked but whose preservation we prove. Many of the rules of the run-time type system are the same as in the static type system. The ones which differ are shown in Fig. 9.

Rule RT3 takes care of the fact that small step reduction may introduce references values into an expression (although the static type system forbids them, see §5.1). The premise $P \vdash typeof_h\ v = \lfloor T \rfloor$ expresses that the value is of the right type; if $v = Ref\ (a, Cs)$, its type is $Class\ (last\ Cs)$ provided $h\ a = \lfloor (C, \_) \rfloor$ and $(C, Cs) \in Subobjs\ P$.

The main reason why static typing is not preserved by reduction is that the type of subexpressions may decrease from a class type to a null type with reduction. Because of this, both cast rules only require the expression to cast to have a reference type ($is\text{-}ref T\ T$), which means either a class or the null type. None of the checks that are needed for the static cast are important for the run-time type system.

Rule RT4 takes care of $e.F\{Cs\}$ where the type of e has reduced to *NT*. Since this is going to throw an exception, and exceptions can

$$\frac{P,E,h \vdash e : T \qquad \textit{is-refT}\ T \qquad \textit{is-class}\ P\ C}{P,E,h \vdash \texttt{dyn\_cast}\ C\ e : \textit{Class}\ C}\ \text{RT1}$$

$$\frac{P,E,h \vdash e : T \qquad \textit{is-refT}\ T \qquad \textit{is-class}\ P\ C}{P,E,h \vdash \texttt{stat\_cast}\ C\ e : \textit{Class}\ C}\ \text{RT2}$$

$$\frac{P \vdash \textit{typeof}_h\ v = \lfloor T \rfloor}{P,E,h \vdash \texttt{Val}\ v : T}\ \text{RT3} \qquad \frac{P,E,h \vdash e : NT}{P,E,h \vdash e.F\{Cs\} : T}\ \text{RT4}$$

$$\frac{P,E,h \vdash e_1 : NT \qquad P,E,h \vdash e_2 : T' \qquad P \vdash T' \leq T}{P,E,h \vdash e_1.F\{Cs\}\ \texttt{:=}\ e_2 : T}\ \text{RT5}$$

$$\frac{P,E,h \vdash e : NT \qquad P,E,h \vdash es\ [:]\ Ts}{P,E,h \vdash e.M(es) : T}\ \text{RT6}$$

Figure 9. Run-time type system

have any type, this expression can have any type, too. Rules RT5 and RT6 work similarly for field assignment and method call.

We have proved that $P,E \vdash e :: T$ implies $P,E,h \vdash e : T$. Heap h is unconstrained as the premise implies that e does not contain any references.

7.2 Conformance and Definite Assignment

Progress and preservation require that all semantic objects *conform* to the type constraints imposed by the syntax. We say that a value v conforms to a type T (written $P,h \vdash v :\leq T$) if the type of v equals type T or, if T is a class type, v has type NT. A heap conforms to a program if for every object (C, S) on the heap

- if $(Cs, fs) \in S$ then $(C, Cs) \in \textit{Subobjs}\ P$ and if F is a field of type T declared in class $\textit{last}\ Cs$ then $fs\ F = \lfloor v \rfloor$ and the type of v (in the sense of rule RT1) conforms to type T.
- if $(C, Cs) \in \textit{Subobjs}\ P$ then $(Cs, fs) \in S$ for some fs.

In this case we write $P \vdash h\ \surd$. A store l conforms to a type environment E iff $l\ V = \lfloor v \rfloor$ implies $E\ V = \lfloor T \rfloor$ such that v conforms to T. In symbols: $P,h \vdash l\ (:\leq)_w\ E$. We also need conformance concerning the type environment: $\textit{envconf}\ P\ E$ states that for every variable that maps to a type in environment E, the type is a valid type in program P.

$$\textit{envconf}\ P\ E \equiv \forall\ V\ T.\ E\ V = \lfloor T \rfloor \longrightarrow \textit{is-type}\ P\ T$$

If $P \vdash h\ \surd$, $P,h \vdash l\ (:\leq)_w\ E$ and $\textit{envconf}\ P\ E$ then we write $P,E \vdash (l, h)\ \surd$ and say that state (h,l) conforms to the program and the environment.

For the proof we need another conformance property, which we call *type-conf*. It simply describes that given a certain type, an expression has that type in the run-time type system. However, if this given type is a class type, the run time type system may also return the null type for the expression.

$$\begin{aligned} \textit{type-conf}\ P\ E\ (\textit{Class}\ C)\ h\ e &= P,E,h \vdash e : \textit{Class}\ C \vee P,E,h \vdash e : NT \\ \textit{type-conf}\ P\ E\ \textit{Void}\ h\ e &= P,E,h \vdash e : \textit{Void} \end{aligned}$$

The rules for *Boolean*, *Integer* and *NT* are analogous to the rule containing *Void*.

From Jinja we have inherited the notion of *definite assignment*, a static analysis that checks if in an expression every variable is initialized before it is read. This constraint is essential for proving type safety. Definite assignment is encoded as a predicate $\mathcal{D}$ such that $\mathcal{D}\ e\ A$ (where A is a set of variables) asserts the following property: if initially all variables in A are initialized, then execution of e does not access an uninitialized variable. For technical reasons A is in fact of type *vname set option*. That is, if we want to execute e in the context of a store l we need to ensure $\mathcal{D}\ e\ \lfloor \textit{dom}\ l \rfloor$. Since $\mathcal{D}$ is completely orthogonal to multiple inheritance we have omitted all details and refer to [10] instead.

7.3 Progress

Progress means that any (run-time) well-typed expression which is not yet not fully evaluated (i.e. final) can be reduced by a rule of the small step semantics. To prove this we need to assume that the program is well-formed, the heap and the environment conforms, and the expression passes the definite assignment test:

If $\textit{wf-C-prog}\ P$ and $P,E,h \vdash e : T$ and $P \vdash h\ \surd$ and $\textit{envconf}\ P\ E$ and $\mathcal{D}\ e\ \lfloor \textit{dom}\ l \rfloor$ and $\neg\ \textit{final}\ e$ then $\exists\ e'\ s'.\ P,E \vdash \langle e,(h,l)\rangle \rightarrow \langle e',s'\rangle$.

This theorem is proved by a quite exhausting rule induction on the (run-time) typing rules, where most cases consist of several more case distinctions, like e being final or not. So some cases can get quite long (e.g., the proof for method call has about 150 lines of proof script).

7.4 Preservation

To achieve type safety we have have to show that all of the assumptions in the Progress theorem above are preserved by the small steps rules.

First, we consider the heap conformance:

If $\textit{wf-C-prog}\ P$ and $P,E \vdash \langle e,(h,l)\rangle \rightarrow \langle e',(h',l')\rangle$ and $P,E,h \vdash e : T$ and $P \vdash h\ \surd$ then $P \vdash h'\ \surd$.

We proof this by induction on the small step rules. Most cases are straightforward, the only work lies in the rules which alter the heap, namely the ones for creation of new objects and field assignment.

Next, we need a similar rule for the conformance of the locals. To prove this, we need to assume that the program is well formed, the environment conforms to it and the expression is well typed in the runtime type system:

If $\textit{wf-C-prog}\ P$ and $P,E \vdash \langle e,(h,l)\rangle \rightarrow \langle e',(h',l')\rangle$ and $P,E,h \vdash e : T$ and $P,h \vdash l\ (:\leq)_w\ E$ and $\textit{envconf}\ P\ E$ then $P,h' \vdash l'\ (:\leq)_w\ E$.

Here, the interesting cases from the small step rule induction are those that change the locals, namely variable assignment and blocks with locally declared variables.

Furthermore, also definite assignment needs to be preserved from the semantics. The corresponding lemma is quite easily proved by induction on the small step rules:

If $\textit{wf-C-prog}\ P$ and $P,E \vdash \langle e,(h,l)\rangle \rightarrow \langle e',(h',l')\rangle$ and $\mathcal{D}\ e\ \lfloor \textit{dom}\ l \rfloor$ then $\mathcal{D}\ e'\ \lfloor \textit{dom}\ l' \rfloor$.

Finally we have to show that the semantics preserves well-typedness. Preservation of well-typedness here means that the type of the reduced expression is equal to that of the original expression or, if the original expression had a class type, the type may reduce to the null type. This is formalised via the *type-conf* property from §7.2:

If $\textit{wf-C-prog}\ P$ and $P,E \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle$ and $P,E \vdash s\ \surd$ and $P,E,hp\ s \vdash e : T$ then $\textit{type-conf}\ P\ E\ T\ (hp\ s')\ e'$.

where $hp\ s$ is the heap component of s. This proof is quite lengthy because the most complicated cases (mostly method call and field assignment) of the 68 small step rules can have up to 80 lines of proof script each (the snapshot in Fig. 10 shows the first case of the proof).

7.5 The type safety proof

All the preservation lemmas only work 'one step'. We have to extend them from $\rightarrow$ to $\rightarrow^*$, which is done by induction (because of the equivalence of big and small step semantics mentioned in §6.4, all these lemmas now also hold for the big step rules).

Now combining type preservation with progress yields the main theorem:

If *wf-C-prog P* and $P,E \vdash s\ \surd$ and $P,E \vdash e :: T$ and
$\mathcal{D}\ e\ \lfloor dom\ (lcl\ s)\rfloor$ and $P,E \vdash \langle e,s\rangle \rightarrow^* \langle e',s'\rangle$ and
$\neg\ (\exists e''\ s''.\ P,E \vdash \langle e',s'\rangle \rightarrow \langle e'',s''\rangle)$ then
$(\exists v.\ e' = \mathtt{Val}\ v \wedge P,hp\ s' \vdash v :\leq T) \vee$
$(\exists r.\ e' = Throw\ r \wedge the\text{-}addr\ (Ref\ r) \in dom\ (hp\ s'))$.

If the program is well formed, state s conforms to it, e has type T and passes the definite assignment test w.r.t. $dom\ (lcl\ s)$ (where *lcl s* is the store component of s) and its $\rightarrow$-normal form is e', then the following property holds: either e' is a value of type T (or *NT*, if T is of type class) or an exception *Throw r* such that the address part of r is a valid address in the heap.

8. Evolution of the Semantics

The semantics presented in this paper has gone through several stages. This section will discuss a few example steps in the evolution of the specification.

8.1 Addresses, references and object structure

From the beginning, it was clear that objects in the heap have to comprise an object's dynamic class, a subobject, and the values stored in the object's fields. We initially thought that pointers to objects could be identified by just an address. However, by studying the behaviors of static casts and field operations, we soon realized that we need to keep track of the subobject that is currently being pointed to. Our first attempt was to incorporate this information in the object description itself, so objects became a triple with a path (the only way to uniquely identify a subobject) as the third component:

$$obj = cname \times path \times (path \rightharpoonup vname \rightharpoonup val)$$
$$Addr\ addr, \text{ where } addr = nat$$

However, in the presence of multiple pointers to some object o, each of these pointers may point to a different subobject of o, and hard-coding subobject information in o itself is clearly insufficient. Realizing this, we removed the path component from the *object* and included it with the *pointer* (which we now call a *reference*), which is similar to how C++ works. Moreover, for technical reasons, we replaced the mapping from paths to the variable maps by a set of tuples with these two components. Thus, we arrived at the object representation that we are using now:

$$obj = cname \times (path \times (vname \rightharpoonup val))\ set$$
$$Ref\ reference, \text{ where } reference = addr \times path \text{ and } addr = nat$$

8.2 Eliminating exceptions by using static type information

A big issue was how to handle method calls that become ambiguous at run-time. As already stated in the discussion of example 3 in §2.4, we initially considered the use of static information to resolve dynamically dispatched calls contrary to the idea of dynamic dispatch. Following this line of reasoning, we argued that a method call that is ambiguous at runtime should not be resolved but should throw a *MemberAmbiguousException* instead. So the rule looked as follows:

$$\frac{\begin{array}{c}P \vdash \langle e,s_0\rangle \Rightarrow \langle ref\ (a,Cs),s_1\rangle \\ P \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,l_2)\rangle \qquad h_2\ a = Some(C,S) \\ \forall\ Ts\ T\ pns\ body\ Cs'.\ \neg\ P \vdash C\ has\ least\ M = (Ts,T,pns,body)\ via\ Cs'\end{array}}{P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle THROW\ MemberAmbiguous,(h_2,l_2)\rangle}$$

A similar issue arose in the presence of overridden methods with covariant return types. Consider, for example, a situation where the result of a method call (a reference) is assigned to a variable, and where there exists an overriding definition of the method under consideration with a "smaller" return type. Then, by assigning the returned reference to the variable, the reference may receive a supertype to its actual type (given by the last class in its path component). Because of this it was possible to have references with a "gap" between the last class in its path component and the static class given by the (run time) type system. In the field access and field assignment rules one needed to fill this gap by introducing in the rules a third path. We could not always guarantee this third path to be unique, and also threw the *MemberAmbiguousException* when this was not the case.

However, realizing that the introduction of a new exception takes us away from the semantics of C++, we adopted the use of static information in both cases to eliminate the *MemberAmbiguousException* exception. To this end, we introduced the term of an *overrider* which enabled us to use static information to make a dynamically ambiguous method call unique. Of course, the resulting method call rule is quite intricate and requires auxiliary predicates. To close the "gap" between the last class of a reference and the class computed by the type system we extended assignment and method call rules with explicit casts to the static type. Thus the need for the exception disappeared.

9. Working with Isabelle

This section is written for the benefit of readers unfamiliar with automated theorem provers. So far they may have gotten the impression that, given all the definitions and the statement of a lemma, Isabelle proves it automatically. Unfortunately, formal proofs still require much effort by an expert user, a limitation Isabelle shares with all such proof systems. A proof is an interactive process, a dialogue where the user has to provide the overall proof structure and the system checks its correctness but also offers a number of tools for filling in missing details. Chief among these tools are the simplifier (for simplifying formulae) and the logical reasoner (for proving predicate calculus formulae automatically).

Most of the proofs in the present paper are written in *Isar* [37], a language of structured and stylized mathematical proofs understandable to both machines and humans. This proof language is invaluable when constructing, communicating and maintaining large proofs.

Fig. 10 shows a snapshot of *Proof General* [1], Isabelle's GUI, which turns the XEmacs editor into a front end for Isabelle that supports interactive proof construction. In the main window the reader can see a fragment of an Isar proof text. Other windows show the context, e.g. assumptions currently available, and diagnostic information, e.g. if a proof step succeeded or failed.

Isabelle also supports the creation of LaTeX documents (such as this paper) based on Isabelle input files: LaTeX text may contain references to definitions and lemmas in Isabelle files and Isabelle will automatically substitute those references by pretty printed and typeset versions of the respective formulae. This is similar to and has all the advantages of "literate programming".

10. Related work

There is a wealth of material on formal semantics of object-oriented languages, but to our knowledge, a formal semantics for a language with C++-style multiple inheritance has not yet been presented. We distinguish several categories of related work.

10.1 Semantics of Multiple Inheritance

Cardelli [5] presents a formal semantics for a form of multiple inheritance based on structural subtyping of record types, which also extends to function types. Another early paper that claims to give a semantics to multiple inheritance for a language (PCF++) with

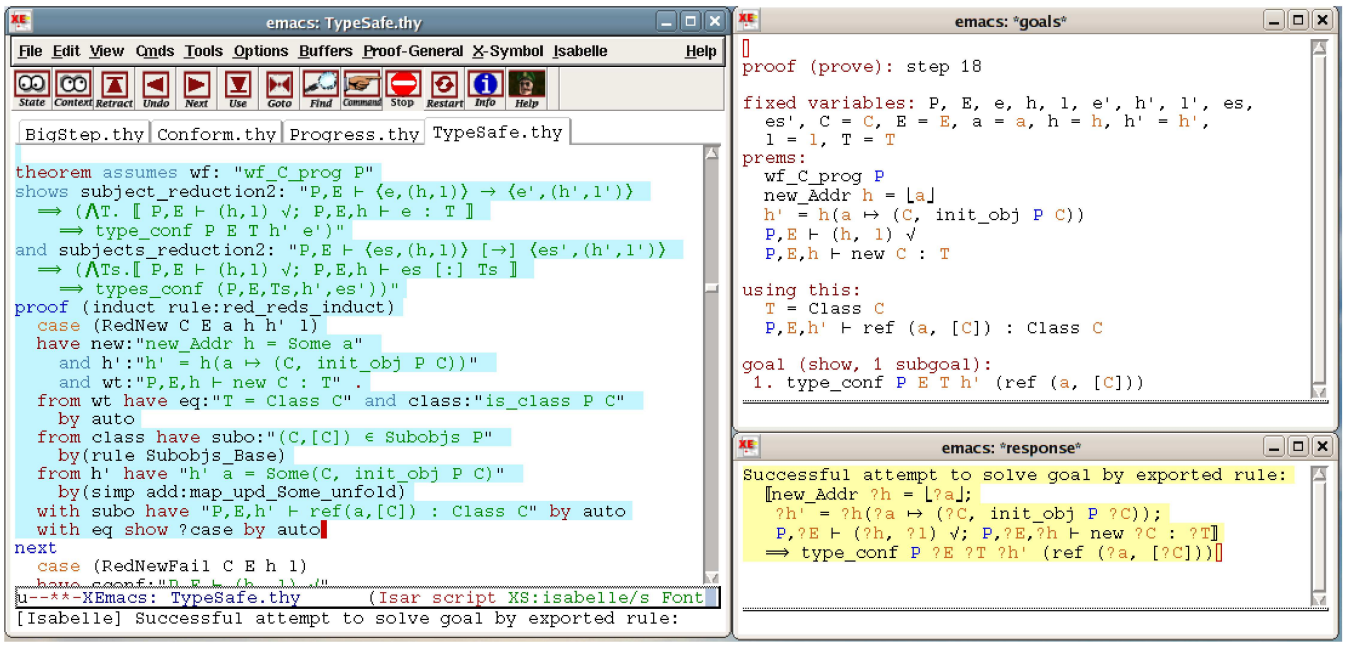

Figure 10. Snapshot of Isabelle in the Proof General GUI

record types is [4]. It is difficult to relate the language constructs used in each of these works to the multiple inheritance model of C++.

10.2 C++ Multiple Inheritance

Wallace [35] presents an informal discussion of the semantics of many C++ constructs, but avoids all the crucial issues. The natural semantics for C++ presented by Seligman [22] does not include multiple inheritance or covariant return types. Most closely related to our work is [8], where some basic C++ data types (including structs but excluding pointers) are specified in PVS; an object model is "in preparation".

The complexities introduced by C++-style multiple inheritance are manifold, and have to our knowledge never been formalized adequately or completely. In the C++ standard [28], the semantics of operations such as method calls and casts that involve class hierarchies are defined informally, while several other works (see, e.g., [26]) discuss the implementation of these operations in terms of compiler data structures such as virtual function pointer tables ("vtables").

Rossie and Friedman [20] were the first to formalize the semantics of operations on C++ class hierarchies in the form of a calculus of subobjects, which forms the basis of our previous work on semantics-preserving class hierarchy transformations that was already mentioned in §1 [33, 23, 24, 25].

Ramalingam and Srinivasan [19] observe that a direct implementation of Rossie and Friedman's definition of member lookup can be inefficient because the size of a subobject graph may be exponential in the size of the corresponding class hierarchy graph. They present an efficient member lookup algorithm for C++ that operates directly on the class hierarchy graph. However, like Rossie and Friedman, their definition does not follow C++ precisely in cases where static information is used to resolve ambiguities (see Example 3 in §2.4).

It has long been known that inheritance can be modeled using a combination of additional fields and methods (a mechanism commonly called "delegation") [11]. Several authors have suggested independently that multiple inheritance can be simulated using a combination of interfaces and delegation [32, 31, 34]. Nonetheless, all of these works stop well short of dealing with the more intricate aspects of modeling multiple inheritance such as object initialization, implicit and explicit type casts, instanceof-operations, and handling shared and repeated multiple inheritance.

Multiple inheritance also poses significant challenges for C++ compiler writers because the layout of an object can no longer reflect a simple linearization of the class hierarchy. As a result, a considerable amount of research effort has been devoted to the design of efficient object layout schemes for C++ [30, 29, 39].

10.3 Other Languages with Multiple Inheritance

Various models of multiple inheritance are supported in other object-oriented languages, and we are aware of a number of papers that explore the semantic foundations of these models.

The work by Attali *et al.* [2] is similar to ours in spirit but treats Eiffel rather than C++, whose multiple inheritance model differs considerably. Eiffel uses shared inheritance by default; repeated inheritance is not possible, instead repeated members must be uniquely renamed when inherited.

In several recent languages such as Jx [15] and Concord [9], multiple inheritance arises as a result of allowing classes to override other classes, in the spirit of BETA's virtual classes [12]. In Jx [15], an outer class A_1 can declare a nested class $A_1.B$, which can be overridden by a nested class $A_2.B$ in a subclass A_2 of A_1. In this case, $A_2.B$ is a subclass of $A_1.B$. Shared multiple inheritance arises when $A_2.B$ also has an explicitly defined superclass. Member lookup is defined quite differently than in C++ (implicit overriding inheritance takes precedence over explicit inheritance when selecting a member), but appears to behave similarly in practice. Nystrom et al. present a type system, operational semantics and soundness proof for Jx, although the latter is not machine-checked.

Concord [9] introduces a notion of *groups* of classes, where a group g may be extended by a subgroup g'. An implicit form of inheritance exists between a class $g.X$ declared in group g that is further bound by a class $g'.X$ in subgroup g', giving rise to a simi-

lar form of shared multiple inheritance as in Jx. Two important differences, however, are the fact that further binding does not imply subtyping: $g'.X$ is not a subtype of $g.X$, and explicit inheritance takes precedence over implicit overriding when resolving method calls. Jolly et al. present a type system and soundness proof (though not machine-checked) for Concord. Because repeated multiple inheritance is not supported in either Jx or Concord, the semantics for these languages can represent the run-time type of an object as a simple type, and there is no need for the subobject and path information required for modeling C++.

Scala [16] provides a mechanism for symmetrical mixin inheritance [3] in which a class can inherit members from multiple superclasses. If members are inherited from two mixin classes, the inheriting class has to resolve the conflict by providing an explicit overriding definition. Scala side-steps the issue of shared vs. repeated multiple inheritance by simply disallowing a class to (indirectly) inherit from a class that encapsulates state more than once (multiply inheriting from abstract classes that do not encapsulate state—called traits—is allowed, however). The semantic foundations of Scala, including a type system and soundness proof can be found in [17].

11. Conclusion

We have presented an operational semantics and type-safety proof for multiple inheritance in C++. The semantics precisely models the behavior of method calls, field accesses and two forms of casts in C++ class hierarchies, and allows one—for the first time—to understand the behavior of these operations without referring to implementation-level data structures such as virtual function pointer tables (v-tables). The type-safety proof was formalized and machine-checked using Isabelle/HOL.

The paper discusses C++ features in the light of the formal analysis, discusses a number of subtleties in the design of C++ that we encountered during the construction of the semantics, and provides some background about its evolution. Trying to put C++ on a formal basis has been interesting but quite challenging at times. It was great fun figuring out what C++ means at an abstract level, and this exercise has demonstrated that its mixture of shared and repeated multiple inheritance gives rise to a lot of additional complexity at the semantics level. Our work opens the door to machine-checked correctness proofs of transformations such as the automated elimination of multiple inheritance from C++ programs.

References

[1] David Aspinall. Proof General — a generic tool for proof development. In S. Graf and M.I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, TACAS 2000*, volume 1785 of *Lect. Notes in Comp. Sci.*, pages 38–42. Springer-Verlag, 2000.

[2] Isabelle Attali, Denis Caromel, and Sidi Ould Ehmety. A natural semantics for eiffel dynamic binding. *ACM TOPLAS*, 18(6):711–729, 1996.

[3] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. of OOPSLA/ECOOP'90*, pages 303–311, 1990.

[4] V. Breazu-Tannen, C. A. Gunter, and A. Scedrov. Computing with coercions. In *Proc. ACM Conf. LISP and functional programming*, pages 44–60. ACM Press, 1990.

[5] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[6] Luca Cardelli. Type systems. In *The Computer Science and Engineering Handbook*. 2 edition, 2004.

[7] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In *Proc. of ECOOP'97*, volume 1241 of *Lect. Notes in Comp. Sci.*, pages 389–418, 1997.

[8] Michale Hohmuth and Hendrik Tews. The semantics of C++ data types: Towards verifying low-level system components. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, Emerging Trends Proc.*, pages 127–144. Universität Freiburg, 2003. Tech. Rep. 187.

[9] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *Proc. of FTfJP'05*, 2005.

[10] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM TOPLAS*. To appear.

[11] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. of OOPSLA'86*, pages 214–223, 1986.

[12] Ole Lehrmann Madsen and Birger Moeller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proc. of OOPSLA'89*, pages 397–406, 1989.

[13] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[14] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* 2002. `http://www.in.tum.de/~nipkow/LNCS2283/`.

[15] Nathaniel Nystrom, Stephen Chong, and Andrew. C. Myers. Scalable extensibility via nested inheritance. In *Proc. of OOPSLA'04*, pages 99–115, 2004.

[16] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2004. Available from `scala.epfl.ch`.

[17] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. of ECOOP'03*.

[18] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[19] G. Ramalingam and Harini Srinivasan. A member lookup algorithm for c++. In *Proc. of PLDI '97*, pages 18–30, 1997.

[20] Jonathan G. Rossie, Jr. and Daniel P. Friedman. An algebraic semantics of subobjects. In *Proc. of OOPSLA'95*, pages 187–199. ACM Press, 1995.

[21] Jonathan G. Rossie, Jr., Daniel P. Friedman, and Mitchell Wand. Modeling subobject-based inheritance. In *Proc. of ECOOP'96*, volume 1098 of *Lect. Notes in Comp. Sci.*, pages 248–274, 1996.

[22] Adam Seligman. *FACTS: A formal analysis for C++*. Williams College, 1995. Undergraduate thesis.

[23] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM TOPLAS*, pages 540–582, 2000.

[24] Gregor Snelting and Frank Tip. Semantics-based composition of class hierarchies. In *Proc. of ECOOP'02*, volume 2374 of *Lect. Notes in Comp. Sci.*, pages 562–584, 2002.

[25] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with kaba. In *Proc. of OOPSLA'04*, pages 315–330, 2004.

[26] Bjarne Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4), 1989.

[27] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994.

[28] Bjarne Stroustrup. *The C++ Standard: Incorporating Technical Corrigendum No. 1*. John Wiley, 2 edition, 2003.

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

[30] Peter F. Sweeney and Joseph Gil. Space and time-efficient memory layout for multiple inheritance. In *Proc. of OOPSLA'99*, pages 256–275, 1999.

[31] Ewan Tempero and Robert Biddle. Simulating multiple inheritance in Java. *Journal of Systems and Software*, 55:87–100, 2000.

[32] Krishnaprasad Thirunarayan, Günter Kniesel, and Haripriyan Hampapuram. Simulating multiple inheritance and generics in Java. *Computer Languages*, 25:189–210, 1999.

[33] Frank Tip and Peter Sweeney. Class hierarchy specialization. *Acta Informatica*, 36:927–982, 2000.

[34] John Viega, Bill Tutt, and Reimer Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical Report CS-98-3, University of Virginia, 1998.

[35] Charles Wallace. The semantics of the C++ programming language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[36] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An Operational Semantics and Type Safety Proof for C++-like Multiple Inheritance. Technical Report RC23709, IBM, 2005.

[37] Markus Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.

[38] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, (115):38–94, 1994.

[39] Yoav Zibin and Joseph Gil. Two-dimensional bi-directional object layout. In *Proc. of ECOOP'03*, volume 3013 of *Lect. Notes in Comp. Sci.*, pages 329–350, 2003.