RC23709 (W0508-162) August 31, 2005
Computer Science

IBM Research Report

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

Daniel Wasserrab[1], Tobias Nipkow[2], Gregor Snelting[1], Frank Tip[3]
[1]Universität Passau
Germany

[2]Technische Universität
München, Germany

[3]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

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

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, for the first time, an operational semantics and a type system for a C++-like object-oriented language with both shared and repeated multiple inheritance, together with a machine-checked proof of type safety. The formalization uncovered several subtle ambiguities in C++, which C++ compilers resolve by ad-hoc means or which even result in uncontrolled run-time errors. The semantics is formalized in Isabelle/HOL.

1. Introduction

Java has been a favorite target of language specifiers for close to a decade. C++ has received much less attention, perhaps due to the much greater complexity of the language. One of the main sources of this complexity is the fact that C++ allows 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 [25], and in terms of implementation-level constructs such as virtual function tables (v-tables) [24]. We are only aware of a few formal treatments—and of no operational semantics—for C++-like languages with shared and repeated multiple inheritance. In 1996, Rossie, Friedman, and Wand [18] 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 main contribution of this paper is a formal and executable operational semantics for a language with C++-like multiple inheritance, with a machine-checked type-safety proof. This semantics frees programmers and language implementors from the need to discuss program behavior in terms of implementation-level constructs such as v-tables. Type safety is a language property which can be summarized by the famous slogan "Well-typed programs can't go wrong". Cardelli's definition of type safety [5] demands that no untrapped errors may occur (but controlled exceptions are allowed). Our type safety proof is completely formalized and machine-checked by the Isabelle/HOL theorem prover [12].

Our semantics builds on the multiple inheritance calculus developed by Rossie and Friedman [17], but goes well beyond that work by providing an executable semantics and a type-safety proof. Roughly speaking, our semantics extends a formal model for a subset of Java called Jinja [9]. Jinja is a completely formal model of a Java-like language defined in higher-order logic (HOL) in the theorem prover Isabelle/HOL. C+, the language defined in this paper, is derived from Jinja by moving from single to (shared and repeated) multiple inheritance. In this paper we have refrained from presenting the formal definition of all of C+ but have concentrated on those aspects that are affected by multiple inheritance. The rest is practically identical to Jinja and can be found elsewhere [9].

In the course of designing the semantics, we also discovered several subtle ambiguities regarding member access and method calls in C++. Compilers resolve such ambiguities in an ad-hoc manner, or even produce code that leads to uncontrolled run-time errors. C+ either disallows such ambiguities, or generates a controlled exception instead of an uncontrolled run-time error. We believe that this approach should have been adopted for C++ as well.

Our interest in formalizing the semantics of multiple inheritance is motivated by previous work by two of the present authors on: (i) restructuring class hierarchies in order to reduce object size at run-time [30], (ii) composition of class hierarchies in the context of an approach for aspect-orientation [21], and (iii) refactoring class hierarchies in order to improve their design [22, 20], 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 use the formal semantics for C+ to demonstrate such program transformations for eliminating multiple inheritance to be semantics-preserving.

2. Multiple inheritance

2.1 An intuitive introduction to subobjects

C+ features both *repeated* and *shared* multiple inheritance (corresponding to *nonvirtual* and *virtual* inheritance in C++, respectively). 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 Jinja class hierarchies.

```
class Top { int x, y; ... }
class Left :  Top { ... }
class Right : Top { int y; ... }
class Bottom : Left, Right { int x; ... }
```

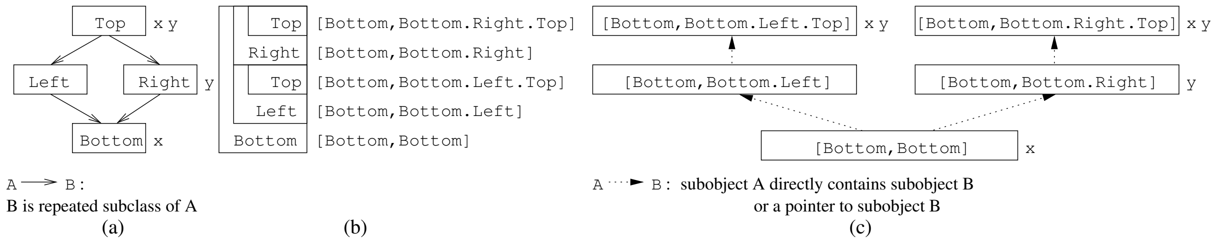


Figure 1. The repeated diamond

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

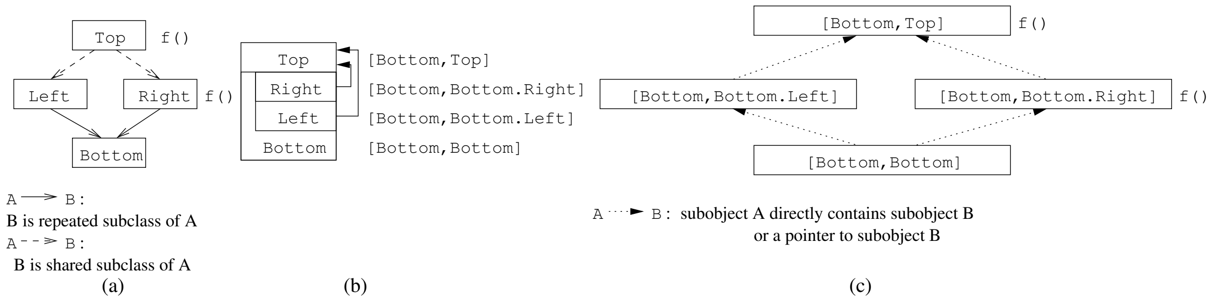


Figure 2. The shared diamond

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 (in the form of *virtual-base pointers*) [24, p.266], although indirections can be avoided in certain special cases [34, 26, 27]. 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 [17] 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*[1] 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

[1] In this paper, we follow the terminology of [17] 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.

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:

1. $[C, C]$ is the subobject that represents the "full" C-object.
2. 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 directly contained within subobject S_1.
3. 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* [25]. 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[2]. 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 $[\texttt{Bottom}, \texttt{Bottom.Right}]$ dominates the one in $[\texttt{Bottom}, \texttt{Top}]$. On the other hand, the subobject graph in Fig. 1(c) contains 3 definitions of `y` in $[\texttt{Bottom}, \texttt{Bottom.Right}]$, $[\texttt{Bottom}, \texttt{Bottom.Right.Top}]$, and $[\texttt{Bottom}, \texttt{Bottom.Left.Top}]$. As there is no unique dominant definition of `y` here, a field access `(new Bottom())->y` is ambiguous.

2.3 Casts in C+

In the discussion that follows, we will assume e to be an expression declared to be of type D. A cast $(C)e$ is an *up-cast* if D is a direct or indirect subclass of C, and such an up-cast is statically type-correct if $[D, D]$ contains exactly one subobject whose static class is C. At run-time, when e may point to a subobject $S = [C', \cdots D]$, for some subclass C' of D, the up-cast will always succeed and result in the selection of the unique subobject contained in S whose static class is C. A cast $(C)e$ is a *down-cast* if D is a supertype of C. At run-time, when e may point to a subobject $S = [C', \cdots D]$, for some subclass C' of D, this down-cast will succeed if there is a unique subobject S' that contains S, and whose static class is C.

C++ has three cast operators for traversing class hierarchies, each of which has significant limitations[3]. Most commonly used (and used in the examples presented later in this section) 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 type does not contain a unique subobject whose static class is the type being casted to). C-style casts cannot be used to down-cast along a shared inheritance relation. 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 has the desirable property that failing casts result in controlled exceptions (when the operand is of a reference type) or the special value `NULL` (when the operand 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. Clearly, none of these three C++ cast operators always provides the desired, type-safe behavior.

In C+, we have defined a type-safe variation on `static_cast` that throws controlled exceptions in cases where down-casting fails. An alternative would be to define a variation on `dynamic_cast` that does not suffer from the problems discussed above.

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 for 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):

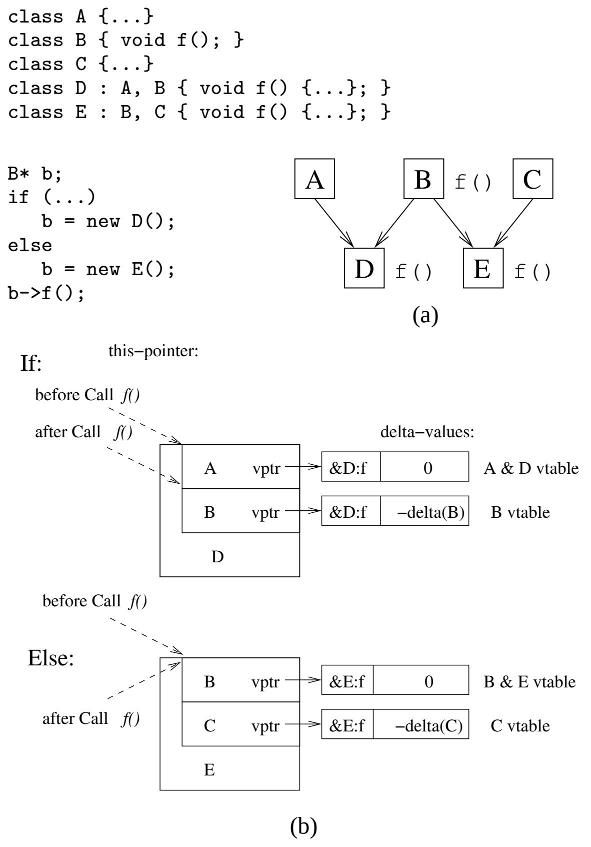
```
Left* b = new Bottom(); b->f();
```

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` 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).

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

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


```
class A {...}
class B { void f(); }
class C {...}
class D : A, B { void f() {...}; }
class E : B, C { void f() {...}; }

B* b;
if (...)
   b = new D();
else
   b = new E();
b->f();
```

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

But for the area computation, the hidden border must be taken into account, thus f() from "Window with border" must be called.

Example 2. Consider the "repeated diamond" of Fig. 1, and assume that a method f() is defined in Right. Now, consider the following sequence of (implicit and explicit) casts, followed by a method call:

```
Top* t; Left* l; Right* r; Bottom* b;
b = new Bottom();
l = b; t = l; r = (Right*) t;
r->f();
```

The down-cast to (Right∗) is statically allowed in C++, but the method call leads to a run-time error (in our semantics, an exception is thrown). The same example for the "shared diamond" is statically incorrect in C++ if a C-style cast or static_cast is used, because down-casts from shared superclasses are disallowed. However, C++ would allow the use of dynamic_cast in that case.

Example 3. 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() involve 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, and 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 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 $-delta(B)$ 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) [24] 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).

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

Example 4. The following example shows how C++ sometimes resolves ambiguities in an ad-hoc manner. 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() is 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. We can model this solution, but it makes the semantics of the method call more complex. Therefore, we decided not to follow C++ in our C+ semantics, but to make the ambiguity visible by throwing an exception instead. We believe this is what C++ should also do, instead of hiding the ambiguity and "polluting" the mechanism of dynamic dispatch.

Example 5. C++ allows method overriding with *covariant* (i.e., more specific) return types, and so does C+[4]. Unrestricted covariance can lead to ambiguities. In the context of the repeated diamond of Fig. 1 with a field x declared in class Top, consider:

[4] We also allow *contravariant* (i.e. more general) parameter types in method overriding. In C++ parameter types must be invariant (otherwise it is overloading, which we do not support).

```
class A { Top f(); }
class B : A { Bottom f(); }

A* a = new B();
(a->f())->x = 42;
```

Statically, everything seems fine: because the type of `a` is `A`, the type of `a->f()` is `Top`, which has a (unique) field `x`. However, if we allowed the redefinition of `f()`, at run-time `a->f()` evaluates to a `Bottom` object and the field access will be ambiguous. Hence, C++ and C+ require *unique covariance*: if the old return type is C and the new return type is D, then there must exist a unique path from D back to C.

3. Formalization

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 The basic types of truth values, natural numbers and integers are called *bool*, *nat*, and *int*. 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 $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$.

$$\textbf{datatype}\ 'a\ option = None \mid Some\ 'a$$

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:

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

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:

- $\boldsymbol{P \vdash C \prec_R D}$ means D is a repeated base class of C.
- $\boldsymbol{P \vdash C \prec_S D}$ means D is a 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 both this definition and the one by Rossie and Friedman yield pairs $(C, [C_1, \ldots, C_n])$ such that $C_1 \prec_R \cdots \prec_R C_n$ and either $C = C_1$ or $C \preceq^* C' \prec_S C_1$ for some C'.

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. It is similar to $@$ but has to take into account that the second path may begin with a shared class, in which case the first path just disappears:

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

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.

Ordering on paths:

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

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

The reflexive and transitive closure of $\sqsubset^1$ is written $\sqsubseteq$.

4. Abstract syntax of C+

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

In the sequel we use the following (HOL) variable conventions: V is a (C+) 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.

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 $[E, B]$ represents the fact that this object has been upcast to B and `b` in fact points to the B subobject.

4.2 Values and Expressions

A C+ **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*.

C+ 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 C+:

- creation of new object: `new` C
- casting: `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:

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

prog	=	*cdecl list*
cdecl	=	*cname* × *class*
class	=	*base list* × *fdecl list* × *mdecl list*
fdecl	=	*vname* × *ty*
mdecl	=	*mname* × *method*
method	=	*ty list* × *ty* × *vname list* × *expr*
datatype *base*	=	*Repeats cname* | *Shares cname*

Figure 5. Abstract program syntax

4.3 Programs

The abstract syntax of programs is given by the type definitions in Fig. 5, where *ty* is the HOL type of C+ types.

A 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 C+ (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 [9]) 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*** is the class (more precisely: *class option*) associated with C in P.

5. Type system

C+ types are either primitive (*Boolean* and *Integer*), class types *Class C*, *NT* (the type of *Null*), or *Void* (the type of *Unit*). The corresponding HOL type is called *ty*. The subclass relation $\preceq^*$ induces a subtype relation $\leq$ on *ty* as follows:

$$\frac{P \vdash C \preceq^* D}{P \vdash Class\ C \leq Class\ D} \qquad P \vdash NT \leq Class\ C \qquad P \vdash T \leq T$$

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. 4) construct by construct, concentrating on object-orientation. The remaining rules can be found elsewhere [9]. 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*. C+, like Java or ML, does not allow explicit references for well known reasons.

5.1.1 Cast

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

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

$$\frac{P,E \vdash e_1 :: T_1 \qquad P,E \vdash e_2 :: T_2 \qquad \mathit{case\ bop\ of} \texttt{=} \Rightarrow T_1 = T_2 \land T = \mathit{Boolean} \mid \texttt{+} \Rightarrow T_1 = \mathit{Integer} \land T_2 = \mathit{Integer} \land T = \mathit{Integer}}{P,E \vdash e_1 \ll bop \gg e_2 :: T}\ \text{WT3}$$

$$\frac{E\ V = \lfloor T \rfloor \qquad P,E \vdash e :: T}{P,E \vdash V \texttt{:=} e :: T}\ \text{WT2} \qquad \frac{P,E \vdash e :: \mathit{Class}\ C \qquad P \vdash C\ \mathsf{has\ least}\ F : T\ \mathsf{via}\ Cs}{P,E \vdash e.F\{Cs\} :: T}\ \text{WT4}$$

$$\frac{P,E \vdash e_1 :: \mathit{Class}\ C \qquad P \vdash C\ \mathsf{has\ least}\ F : T\ \mathsf{via}\ Cs \qquad P,E \vdash e_2 :: T}{P,E \vdash e_1.F\{Cs\} \texttt{:=} e_2 :: T}\ \text{WT5}$$

$$\frac{P,E \vdash e :: \mathit{Class}\ C \qquad P \vdash C\ \mathsf{has\ least}\ M = (Ts, T, m)\ \mathsf{via}\ Cs \qquad P,E \vdash es\ [::]\ Ts}{P,E \vdash e.M(es) :: T}\ \text{WT6}$$

Figure 4. The typing rules

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

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`, for which C++ has fixed the rules: down-casts may only involve repeated inheritance. To enforce this restriction we introduce the predicate

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

Combining the checks for up- and down-cast in one rule we obtain WT1 (see Fig. 4). 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 static class `C` to `Top` is ambiguous because there are two `Top` subobjects.

Note that we could have chosen to model `dynamic_cast` just as well. We come back to this point in connection with the semantics in §6.3.2.

5.1.2 Variable assignment

The assignment rule WT2 looks puzzling because it requires e to have the same type as V, not a subtype, as in C++. That is, we expect the programmer (or some preprocessor) to insert an up-cast if necessary. This is just a simplification of the rule and does not affect the language: these up-casts need to be performed at run-time and the question is merely if they are inserted explicitly by the programmer of implicitly by the compiler.

5.1.3 Binary operators

As explained above, we assume that all necessary casts are performed explicitly and hence we expect that both arguments of an equality test have the same type. So in the typing rule WT3 for binary operations we need not perform any implicit casting.

5.1.4 Field access and assignment

The typing rule for field access WT4 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 $P \vdash C\ \mathsf{has\ least}\ F : T\ \mathsf{via}\ Cs$ can compute T and Cs from P, C and F. So it remains to explain $P \vdash C\ \mathsf{has\ least}\ F : T\ \mathsf{via}\ Cs$: 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 $\mathit{FieldDecls}\ P\ C\ F$ of all (Cs, T) such that Cs is a valid path leading to a class with an F of type T:

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

Then we select a least element from that set:

$P \vdash C\ \mathsf{has\ least}\ F : T\ \mathsf{via}\ Cs \equiv$
$(Cs, T) \in \mathit{FieldDecls}\ P\ C\ F \land$
$(\forall\, (Cs', T') {\in} \mathit{FieldDecls}\ P\ C\ F.\ P,C \vdash Cs \sqsubseteq Cs')$

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 $\mathit{FieldDecls}\ P\ \mathit{Bottom}\ x$ are [`Bottom`], [`Bottom`,`Left`,`Top`], [`Bottom`,`Right`,`Top`].
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 WT5. Again we expect that the necessary casts are already present.

5.1.5 Method call

In the call typing rule WT6 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

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

This set pairs the method (of type *method*, see Fig. 5) with the path Cs leading to the defining class. Among all definitions the least one (w.r.t. the ordering on paths) is selected:

$P \vdash C\ \mathsf{has\ least}\ M = \mathit{mthd}\ \mathsf{via}\ Cs \equiv$
$(Cs, \mathit{mthd}) \in \mathit{MethodDefs}\ P\ C\ M \land$
$(\forall\, (Cs', \mathit{mthd}') {\in} \mathit{MethodDefs}\ P\ C\ M.\ P,C \vdash Cs \sqsubseteq Cs')$

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 showed this problem in Example 4 and will discuss it in detail in §6.3.5.

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

Figure 6. The type of C+ program states

The relation [::] is the pointwise extension of :: to lists. We expect the actual and formal parameters to have exactly the same type. This may require explicit casts, as for assignments.

5.2 Method overriding

In Example 5 we already motivated and stated the *unique covariance* rule for return types in method overriding enforced by C+ and C++. Let us now look ahead to the semantics for a moment. It will be such that if e is of type *Class C* and evaluates to a reference $(\_, Cs)$ then $\mathit{last}\ Cs = C$ — except in the presence of covariance where we only have the weaker $\mathit{last}\ Cs \preceq^* C$. See Example 5: the C+ analogue of `b->foo()` has type *Class B* because `b` is of type *Class B* and hence type checking is based on the definition of `foo` in class `B`, where its return type is *Class B*. At run-time, however, `b` may refer to a `C` object, in which case the definition of `foo` in `C` applies and it may return a reference $(\_, [C])$ to a `C` object.

5.3 Well-formed programs

A well-formed C+ program (*wf-C-prog P*) must obey all the usual requirements (every method body is well-typed and of the declared result type (as in assignments, upcasts must be explicit), the class hierarchy is acyclic, etc — for details see [9]). The only C+-specific condition is 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.

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 $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ 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 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 C+, in contrast to C++, does not allow stack-allocated objects: variable values can only be 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

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

6.3 Evaluation

Remember that $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ is the evaluation judgment. 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:
  $\{Ds \mid \exists fs.\ (Ds, fs) \in S\} = \{Ds \mid (C, Ds) \in \mathit{Subobjs}\ P\}$,
- S is a (finite) function:
  $\forall\, (Cs,fs), (Cs',fs') \in S.\ Cs = Cs' \longrightarrow fs = fs'$

Furthermore, if an expression e evaluates to $\mathit{ref}\ (a, Cs)$ 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.
- $\mathit{last}\ Cs$ is equal to or a subclass of 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 [9].

6.3.1 Object creation

Rule BS1 shows the big step rule for object creation. The result of evaluating `new` C is a reference $\mathit{Ref}\ (a, [C])$ 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 $S = \mathit{init\text{-}obj}\ P\ C$ is the set of all subobjects (Cs, fs) such that $(C, Cs) \in \mathit{Subobjs}\ P$ and $fs :: \mathit{vname} \rightharpoonup \mathit{val}$ is the field table that contains every field declared in class $\mathit{last}\ Cs$ 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, either by lengthening it (in case of an upcast), or shortening it (in case of a down-cast).

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 upcast rule BS2: After evaluating e to a reference with path Cs, that path is extended (upwards) by a unique 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_cast` of C++ (§2.3) 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". The rule is deterministic because any class can occur at most once in a path.

If neither BS2 nor BS3 can be applied, we throw a *ClassCast* exception.

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

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

$$\frac{P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a,\ Cs\ @\ [C]\ @\ Cs'),s_1\rangle}{P \vdash \langle \texttt{Cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a,\ Cs\ @\ [C]),s_1\rangle}\ \text{BS3} \qquad \frac{P \vdash \langle e,s_0\rangle \Rightarrow \langle \texttt{Val}\ v,(h,\ l)\rangle \qquad l' = l(V \mapsto v)}{P \vdash \langle V\ \texttt{:=}\ e,s_0\rangle \Rightarrow \langle \texttt{Val}\ v,(h,\ l')\rangle}\ \text{BS4}$$

$$\frac{P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \texttt{Val}\ v_1,s_1\rangle \qquad P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \texttt{Val}\ v_2,s_2\rangle \qquad \textit{binop}\ (\textit{bop},\ v_1,\ v_2) = \lfloor v \rfloor}{P \vdash \langle e_1 \ll\!\textit{bop}\!\gg e_2,s_0\rangle \Rightarrow \langle \texttt{Val}\ v,s_2\rangle}\ \text{BS5}$$

$$\frac{\begin{array}{c} P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a,\ Cs'),(h,\ l)\rangle \qquad h\ a = \lfloor (D,\ S) \rfloor \\ P \vdash \textit{path last Cs}' \textit{ to hd Cs unique} \qquad P \vdash \textit{path last Cs}' \textit{ to hd Cs via Cs}'' \qquad Ds = Cs'\ @_p\ Cs''\ @_p\ Cs \qquad (Ds,\ fs) \in S \qquad fs\ F = \lfloor v \rfloor \end{array}}{P \vdash \langle e.F\{Cs\},s_0\rangle \Rightarrow \langle \texttt{Val}\ v,(h,\ l)\rangle}\ \text{BS6}$$

$$\frac{\begin{array}{c} P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \textit{ref}\ (a,\ Cs'),s_1\rangle \qquad P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \texttt{Val}\ v,(h_2,\ l_2)\rangle \qquad h_2\ a = \lfloor (D,\ S) \rfloor \\ P \vdash \textit{path last Cs}' \textit{ to hd Cs unique} \qquad P \vdash \textit{path last Cs}' \textit{ to hd Cs via Cs}'' \qquad Ds = Cs'\ @_p\ Cs''\ @_p\ Cs \\ (Ds,\ fs) \in S \qquad fs' = fs(F \mapsto v) \qquad S' = S - \{(Ds,\ fs)\} \cup \{(Ds,\ fs')\} \qquad h_2{}' = h_2(a \mapsto (D,\ S')) \end{array}}{P \vdash \langle e_1.F\{Cs\}\ \texttt{:=}\ e_2,s_0\rangle \Rightarrow \langle \texttt{Val}\ v,(h_2{}',\ l_2)\rangle}\ \text{BS7}$$

$$\frac{\begin{array}{c} P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{ref}\ (a,\ Cs),s_1\rangle \qquad P \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle \textit{map}\ \texttt{Val}\ vs,(h_2,\ l_2)\rangle \\ h_2\ a = \lfloor (C,\ S) \rfloor \qquad P \vdash C \textit{ has least } M = (Ts,\ T,\ pns,\ body) \textit{ via } Cs' \\ |vs| = |pns| \qquad l_2{}' = [\textit{this} \mapsto \textit{Ref}\ (a,\ Cs'),\ pns\ [\mapsto]\ vs] \qquad P \vdash \langle \textit{body},(h_2,\ l_2{}')\rangle \Rightarrow \langle e',(h_3,\ l_3)\rangle \end{array}}{P \vdash \langle e.M(ps),s_0\rangle \Rightarrow \langle e',(h_3,\ l_2)\rangle}\ \text{BS8}$$

Figure 7. The Big Step rules

Although down-casts along shared inheritance can lead to ambiguities (§5.1.1), this is not necessarily so. For example, a down-cast in the shared diamond in Fig. 2 from a *Bottom* object of static class *Top* to class *Bottom* is unambiguous, but `static_cast`, i.e. rule BS3, will not work: the reference to the object is of the form $(\_,\ [\textit{Top}])$ but *Bottom* does not appear in path $[\textit{Top}]$. Nevertheless this cast would be unique, as there exists only one path with static class *Bottom*, namely $[\textit{Bottom}]$. In C++, the operator `dynamic_cast` can be used in such cases. We could easily model it as follows: if e evaluates to $\textit{ref}\ (a,\ \_)$ and $h\ a = \lfloor (B,\ \_) \rfloor$ then `Cast` $C\ e$ evaluates to $\textit{ref}\ (a,\ Cs)$ if Cs is the unique path from B to C. This performs all possible type safe casts and captures `dynamic_cast` (modulo the weird restriction that at least one virtual method is declared, see §2.3).

6.3.3 Variable assignment and binary operators

The assignment rule BS4 is straightforward because of our assumption that upcasts from the type of e to that of V are performed explicitly. Thus both sides have the same type.

The evaluation rule for binary operators BS5 is based on a function *binop* taking the operator and its two argument values and returning an optional result. Optional in order to deal with type mismatches. The definition of *binop* for our two binary operators `=` and `+` is straightforward:

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

Equality on the lhs is the C+ equality operator, equality in the middle is definitional equality, and equality on the rhs is the test for equality.

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. Just as for assignment, `=` does not perform any implicit casts.

6.3.4 Field access and assignment

Let us first look at field access in rule BS6. There are three paths involved. Cs is (if the expression is well-typed, §5.1.4) 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 §5.2, *last* Cs' is in general a subclass of the static class of e. Hence we need to fill the gap between Cs' and Cs. To obtain the complete path leading to the subobject in which F lives, we have to find the missing link Cs'' from *last* Cs' to *hd* Cs. And it must be unique. If it is, $Ds = Cs'\ @_p\ Cs''\ @_p\ Cs$ is the path to the subobject we are looking for. If it is not, we throw a *MemberAmbiguous* exception.

Field assignment (rule BS7) is similar, except that we now have to update the heap at a with a new set of subobjects. Note that the functional nature of this set is preserved. Again, no implicit casts are applied to v.

6.3.5 Method call

Rule BS8, describing method calls, is lengthy, but easy: evaluate e to a reference $(a,\ Cs)$ and the parameter list ps to a list of values vs[5], look up the class C of the object in the heap at a, look up the parameter names *pns* and body *body* of the least method M visible from C, and evaluate the body in a store that contains *this* and the parameters (having made sure that *vs* and *pns* have the same length). 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 C+ does not have global variables.

Method selection follows [17]. We use the same predicate as in §5.1.5, but here we need the least method definition to determine the method body uniquely, whereas in the typing rule only the parameter and return types were of importance. The least path Cs' is used to cast the *this* reference to the subobject expected by the selected method.

[5] $[\Rightarrow]$ is the obvious left-to-right evaluation of expression lists. Saying that the result is of the form *map* `Val` *vs* is a declarative way of ensuring that it is a list of values and of obtaining the actual value list *vs* (as opposed to an expression list).

$$\frac{P \vdash \mathit{typeof}_h\ v = \lfloor T \rfloor}{P,E,h \vdash \mathtt{Val}\ v : T}\ \text{RT1} \qquad \frac{P,E,h \vdash e : \mathit{Class}\ C \qquad Cs \neq [] \qquad P \vdash C\ \mathsf{has}\ F : T\ \mathit{via}\ Cs' @_p\ Cs \qquad Cs' \neq []}{P,E,h \vdash e.F\{Cs\} : T}\ \text{RT2}$$

$$\frac{P,E,h \vdash e : NT}{P,E,h \vdash e.F\{Cs\} : T}\ \text{RT3} \qquad \frac{P,E,h \vdash e : \mathit{Class}\ C \qquad P \vdash C\ \mathit{has}\ M = (Ts,\ T,\ m)\ \mathit{via}\ Cs \qquad P,E,h \vdash es\ [:]\ Ts' \qquad P \vdash Ts'\ [\leq]\ Ts}{P,E,h \vdash e.M(es) : T}\ \text{RT4}$$

Figure 8. Run-time type system

The type system ensures that class C provides or inherits some definition of M. But there might be no least definition, i.e. the call is ambiguous. Such a situation can unfortunately not be ruled out completely statically, as discussed in §5.1.5. In such cases we throw a *MemberAmbiguous* exception.

There are some interesting tradeoffs between our formalization and the approach taken in C++. The approach taken in this paper has the property that the *resolution of a dynamically dispatched method call only depends on the dynamic class of the receiver expression*. This is not the case in C++, where the dynamic class is used if there exists a unique least path on which the method definition occurs, but where the static class of the receiver expression is used to resolve the call otherwise. Our approach is more uniform in the sense that it always consistently uses just the dynamic class, and formalizing the C++ approach would complicate the semantics.

6.4 Small Step Semantics

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

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):

$$P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle = (P \vdash \langle e,s\rangle \to^* \langle e',s'\rangle \wedge \mathit{final}\ e').$$

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 [33] 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 C+ 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 [9]. For a tutorial introduction to type safety see, for example, [16].

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* [6] 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. The key rules are shown in Fig. 8.

Rule RT1 takes care of the fact that small step reduction may introduce references into an expression (although the static type system forbids them, see §5.1). The premise $P \vdash \mathit{typeof}_h\ v = \lfloor T \rfloor$ expresses that the value is of the right type; if $v = \mathit{Ref}\ (a, Cs)$, its type is $\mathit{Class}\ (\mathit{last}\ Cs)$ provided $h\ a = \lfloor(C,\ \_)\rfloor$ and $(C, Cs) \in \mathit{Subobjs}\ P$.

The main reason why static typing is not preserved by reduction is that the type of subexpressions may decrease with reduction. Rule RT2 takes care of this for field access. It no longer insists that Cs leads to the least declaration of F (as in WT4) but to some declaration of F. In the worst case this can lead to a *MemberAmbiguous* exception. Additionally the rule allows a prefix Cs' to compensate for the potentially lower class C of e. The same phenomenon occurred in rule BS6. Rule RT3 takes care of $e.F\{Cs\}$ where the type of e has reduced to NT, the null type. Since this is going to throw an exception, and exceptions can have any type, this expression can have any type, too.

Rule RT4 again compensates for the fact that the types involved may decrease: $\mathsf{has\ least}$ (in rule WT4) is replaced by has just as for field access. It is also allowed that the actual argument types are more specific than the declared argument types. There is also a rule for the case e::NT in which case $e.M(es)$ can have any type.

For well-formed programs P 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 (in the sense of rule RT1) is a subtype of T. A heap conforms to a program if for every object (C, S) on the heap

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

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: $P,h \vdash l\ (:\leq)_w\ E$. If $P \vdash h\ \surd$ and $P,h \vdash l\ (:\leq)_w\ E$ then we write $P,E \vdash (l, h)\ \surd$.

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 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 $\mathcal{D}\ e\ \lfloor \mathit{dom}\ l \rfloor$. Since $\mathcal{D}$ is completely orthogonal to multiple inheritance we have omitted all details and refer to [9] 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 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 $\mathcal{D}\ e\ \lfloor dom\ l \rfloor$ and $\neg\ \textit{final}\ e$ then $\exists e'\ s'.\ P \vdash \langle e,(h,\, l)\rangle \rightarrow \langle e',s'\rangle$.

This theorem is proved by rule induction on the (run-time) typing rules.

7.4 Preservation

We have shown that the semantics preserves the assumptions in the Progress theorem above: conformance, definite assignment and well-typedness. Preservation of well-typedness means that the type of the reduced expression is $\leq$ that of the original expression:

If $\textit{wf-C-prog}\ P$ and $P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle$ and $P,E \vdash s \surd$ and $P,E,hp\ s \vdash e : T$ then $\exists T'.\ P,E,hp\ s' \vdash e' : T' \wedge P \vdash T' \leq T$.

where *hp s* is the heap component of *s*. All preservation lemmas are shown by induction on the small step rules.

Extending the preservation lemmas from $\rightarrow$ to $\rightarrow^*$ (by induction) and combining type preservation with progress yields the main theorem:

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

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 a type $\leq T$ or an exception `Throw` *r* such that the address part of *r* is a valid address in the heap.

8. 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.

8.1 Semantics of Multiple Inheritance

Cardelli [4] 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 record types is [3]. It is difficult to relate the language constructs used in each of these works to the multiple inheritance model of C++.

8.2 C++ Multiple Inheritance

Wallace [32] 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 [19] does not include multiple inheritance or covariant return types. Most closely related to our work is [7], 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 [25], the semantics of operations such as method calls and casts that involve class hierarchies are defined informally, while several other works (see, e.g., [23]) discuss the implementation of these operations in terms of compiler data structures such as virtual function pointer tables ("vtables").

Rossie and Friedman [17] 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 [30, 20, 21, 22].

It has long been known that inheritance can be modeled using a combination of additional fields and methods (a mechanism commonly called "delegation") [10]. Several authors have suggested independently that multiple inheritance can be simulated using a combination of interfaces and delegation [29, 28, 31]. 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++ [27, 26, 34].

8.3 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.* [1] 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 [13] and Concord [8], multiple inheritance arises as a result of allowing classes to override other classes, in the spirit of BETA's virtual classes [11]. In Jx [13], 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 [8] 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 similar 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 [14] provides a mechanism for symmetrical mixin inheritance [2] 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 [15].

9. Conclusion

The full C+ semantics is 10403 LOC of Isabelle code consisting of 133 definitions and 407 theorems and their accompanying proofs. These proofs are handcrafted texts combining high-level proof structures (e.g. inductions and case distinctions) with appeals to automation of low-level inferences (e.g. simplification or predicate calculus proof search). Processing the complete semantics, which entails checking all proofs, takes slightly less than 5 minutes and 400MB of RAM space on an Athlon 3200+ with 2GB of RAM.

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 convinced us that its mixture of shared and repeated multiple inheritance is a very problematic design. We identified a number of ambiguities that C++ resolves in ad-hoc ways, and suggested minor semantic variations that enabled us to prove C+ type-safe. Our semantics, for the first time, allows to explain C++ behavior in terms of a well-defined model (as opposed to in terms of run-time data structures such as vtables), provides a type safety proof which has been an open problem for many years, and opens the door to machine-checked correctness proofs of transformations such as the automated elimination of multiple inheritance from C++ programs. Hence, our semantics is not just a theoretical exercise but of practical relevance.

References

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

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

[3] 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.

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

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

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] 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/.

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

[14] 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`.

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

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

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

[18] 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.

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

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

[21] 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.

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

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

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

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

[26] 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.

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

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

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

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

[31] 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.

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

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

[34] 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.