A Typed Intermediate Language for Compiling Multiple Inheritance

Juan Chen
Microsoft Research
juanchen@microsoft.com

Abstract

Type-preserving compilation can improve software reliability by generating code that can be verified independently of the compiler. Practical type-preserving compilation does not exist for languages with multiple inheritance. This paper presents E_{MI}, the first typed intermediate language to support practical compilation of a programming language with fully general multiple inheritance. The paper demonstrates the practicality of E_{MI} by showing that E_{MI} can be used to faithfully model standard implementation strategies of multiple inheritance for C++, the most widely-used programming language with general multiple inheritance.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and Objects; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages

Keywords Typed intermediate language, class and object encoding, multiple inheritance

1. Introduction

Type-preserving compilation can improve software reliability by generating code that can be verified independently of the compiler [15, 12, 9]. Techniques for practical type-preserving compilation, however, do not exist for programming languages with multiple inheritance, such as C++, Eiffel, SELF, Cecil, and CLOS.

This paper describes E_{MI} (Encoding for Multiple Inheritance), the first typed intermediate language to support practical compilation of a programming language with fully general multiple inheritance. It demonstrates the practicality of E_{MI} by showing that E_{MI} can be used to faithfully model standard implementation strategies of multiple inheritance for C++, the most widely-used programming language with general multiple inheritance.

In a language with multiple inheritance, a class may have multiple superclasses. A subclass object contains embedded objects for superclasses. Standard single-inheritance techniques assume that an object and its embedded superclass object can share the same base location within the object. With multiple inheritance, this is impossible when there is more than one embedded superclass object. Furthermore, there are different ways of handling multiple embedded objects of the same superclass. Repeated inheritance allows multiple copies of the same superclass in a subclass object. Shared inheritance (virtual inheritance in C++) has only one shared copy of a superclass, even if the subclass inherits the superclass more than once. Languages may allow both types of inheritance in the same subclass. With repeated inheritance, a single class name is not sufficient to distinguish embedded objects because a subclass object may have several embedded objects for the same superclass, inherited via different ancestors.

To handle these difficulties, E_{MI} introduces paths—sequences of class names—to describe the location of an embedded object. At compile time, most objects have unknown runtime types and paths. E_{MI} uses path abstractions to represent those objects. E_{MI} introduces special address arithmetic expressions to model pointer adjustment that may occur during casting and dynamic dispatch (so that we can adjust a pointer to a subclass object to point to an embedded superclass object, or vice versa).

E_{MI} borrows some concepts from LIL_C, a typed intermediate language for compiling object-oriented languages with single inheritance of classes (presented in our previous POPL paper [4]). Specifically, it preserves name-based notions such as classes and subclassing, as LIL_C does. E_{MI} also borrows notions such as exact classes and subclassing-bounded quantification. However, it differs from LIL_C in representing objects and inheritance.

Prior work on typed compilation of languages with multiple inheritance is not suitable for practical compilers. It does not address implementation details such as object layout and pointer adjustment. Some work requires non-standard semantics or implementation strategies. More discussion is in Section 5.

In the rest of the paper, Section 2 gives an informal overview of E_{MI}. The next two sections explain the syntax and semantics. Section 5 discusses related work. Section 6 concludes.

A type-preserving translation from a source language to E_{MI}, the complete semantics of E_{MI}, and the proofs of properties of E_{MI} are presented in a companion technical report [2].

2. Overview

This section describes E_{MI} informally. For simplicity, E_{MI} focuses on only core features and omits non-virtual methods, static members, constructors, access control, arrays, local variable assignment, etc. We use capital letters A-E to range over class names.

We first describe a typical object layout for multiple inheritance. In an object of class E, the embedded objects for E's direct superclasses are listed in declaration order and followed by fields introduced in E. Each embedded object has a pointer to a corresponding vtable. A subclass object can share the vtable with the embedded object if the two objects share addresses. Figure 1 shows the layout of a class E that inherits both C and D. The notations f and m with a class name subscript mean fields and virtual methods introduced

POPL'07 January 17–19, 2007, Nice, France.

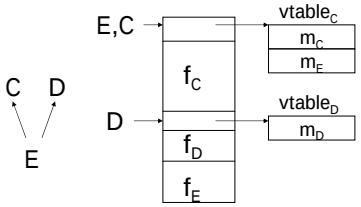

Figure 1. Multiple Inheritance

by the class respectively. The embedded C object is followed by the D object and then by f_E. The embedded C object and the E object share a vtable that contains virtual methods introduced by C and E.

Typically, casting a subclass object to a superclass object requires changing the value of the object pointer. For example, to cast an E object to the superclass D, we need to add an offset to the value of the object pointer. The offset is the difference between E and D. A virtual method call is similar, in that an adjustment may be needed before passing the object to the virtual method. Note that for an embedded object that shares the address with the subclass object, no adjustment is necessary. For example, casting the E object to superclass C does not need to change the object pointer.

With repeated inheritance, a subclass object may have multiple embedded objects for the same superclass. Figure 2 shows that if both C and D inherit a class A, then an E object has two embedded A objects, one for C and the other for D. Class name A is not sufficient to identify the two A objects.

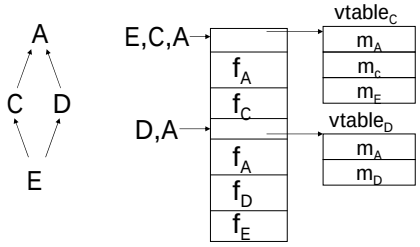

Figure 2. Repeated Inheritance

2.1 Basic Ideas of E_{MI}

E_{MI}, like LIL_C, carries class types all the way through compilation. It uses record types to represent the actual layout of an object. Objects and records of the corresponding record type may be coerced to each other without any runtime effect. Objects are lightweight and are preserved only to simplify the type system.

For each class C, a record type $R(C)$ describes the layout of C. R is not a type constructor in E_{MI}, but a "macro" used by the type checker. The layout in Figure 1 is represented as follows.

$$R(E) = \{C : \{vtable : \mathrm{Ptr}\{m_C, m_E\}, f_C\},\\ D : \{vtable : \mathrm{Ptr}\{m_D\}, f_D\},\ f_E\}$$

The top-level fields C and D in $R(E)$ describe the layouts of the corresponding embedded objects. These fields are themselves records, which are "inlined" into their enclosing record. Thus, the fields in C and D (f_C and f_D) are at fixed offsets from the beginning of the E object.

To represent indirection, E_{MI} has pointer types: pointer type "Ptr τ" represents pointers to heap values of type τ. These are used in this example to represent the indirections to the vtables.

E_{MI} uses **paths**—sequences of class names— to uniquely identify embedded objects. P and Q range over paths. A path P lists *all* intermediate classes between two classes τ_s and τ_e (τ_s must be a subclass of τ_e), represented as $P : (\tau_s, \tau_e)$. We say that P is from τ_s to τ_e. For example, in Figure 2 the two A objects have paths $E :: C :: A$ and $E :: D :: A$ respectively. Both paths are from E to A and they differ only in intermediate classes.

A path with a single class name C means a *complete* C object, which is not embedded in another object. E_{MI} uses class names as labels in record type $R(C)$, so that the path of an embedded object corresponds to the label sequence to fetch the object in $R(C)$. The layout of class E in Figure 2 is:

$$R(E) = \{C : \{A : \{vtable : \mathrm{Ptr}\{m_A, m_C, m_E\}, f_A\}, f_C\},\\ D : \{A : \{vtable : \mathrm{Ptr}\{m_A, m_D\}, f_A\}, f_D\},\ f_E\}$$

Following the label sequence "C, A" in $R(E)$ we can get the layout of the embedded A object with path $E :: C :: A$.

Note that vtable access may require the use of a path. This occurs when an object shares its vtable with a superclass. In general, each object shares its vtable with its **first** non-virtual superclass in declaration order. One can find the vtable by following the chain of first non-virtual superclasses until one reaches a class with no non-virtual superclasses. For example, the vtable of the E object can be accessed with path $E :: C :: A$.

In E_{MI}, paths are used as types of objects. The path of an object starts from the runtime type of the enclosing object and leads to the object. We need this generality because an object with source type C may be a complete C object, or an embedded C object in a subclass object. In turn, using paths as object types allows us to ensure the safety of dynamic dispatch.

E_{MI} uses existential types with path abstraction to describe objects with statically unknown paths. An object with source type C has type $\exists\alpha \ll C.\ \exists\rho : (\alpha, C).\ \rho$ in E_{MI}, read as "there exists a type α which is a subclass of C and a path ρ which is from α to C, and the object has path ρ". The notation "$\ll$" means subclassing. The type variable α identifies the object's runtime type, which must be a subclass of C. The path variable ρ abstracts the path of the object, which is from α to C.

If an object has a statically unknown runtime type and path, its layout is only partially known at compile time. For example, the object might share the vtable with an enclosing subclass object and thus the vtable might contain methods for the unknown subclass.

The record type $\mathrm{Approx}R(P)$ describes the approximate layout of objects with path $P : (\tau, C)$ where P can be concrete or abstract. $\mathrm{Approx}R(P)$ is very similar to $R(C)$. The two types differ in "this" pointer types (Section 2.2) and embedded virtual superclass objects (Section 2.3).

E_{MI} has two expressions for pointer adjustment between superclass and subclass objects. Suppose $P : (\tau, E)$ and D is a "direct" non-virtual superclass of E, that is, E's declaration lists D as one of its superclasses. If a pointer o points to an E object with path P, expression "$o \oplus D$" adjusts o to a pointer to the embedded D object with path $P :: D$. At run time, the expression adds to o the offset between E and D. Adjusting to superclasses higher in the class hierarchy is done by chaining such expressions. Expression "$o' \ominus D$" adjusts back: if o' points to an embedded D object with path $P :: D$, the expression adjusts o' to an E pointer with path P.

2.2 "This" Pointer Types

Each virtual method has a hidden parameter "this". Calling method m on an object o is translated to calling m with the pointer to o as "this". If a class C introduces a method m, the vtable of a C object (complete or embedded) has an entry for m. The entry expects a C pointer as "this". To call m on a subclass object, the compiler inserts code to adjust the subclass object to C. A challenge of typed intermediate languages for OO languages is to give "this" pointers appropriate types to guarantee the safety of dynamic dispatch.

Both runtime types and paths are important in "this" pointer types. Suppose in Figure 2 the class A introduces a method foo

with implementation foo_A and only class C overrides foo with foo_C. The foo method fetched from the embedded object with path $E :: C :: A$ calls foo_C, which might access fields introduced in C. Therefore, the "this" pointer in the method cannot point to an arbitrary A object: it is unsafe to use a pointer to a complete A object or to the embedded A object with path $E :: D :: A$.

Multiple inheritance may require "this" pointer adjustment when subclasses override virtual methods. In the above example, the implementation foo_C in class C expects a C pointer. But the foo method in the vtable of the embedded object with path $E :: C :: A$ expects an A pointer. One standard strategy is to put an "adjuster thunk" for foo in the vtable of the embedded A object. The thunk converts a pointer to an A object that is embedded in a C object to a C pointer, and calls foo_C on the C pointer.

The adjustment involves two kinds of "this" pointers: for method implementations and for thunks (methods in vtables). Subclasses can inherit method implementations in superclasses. Therefore, the "this" pointer of a method implementation in class C (e.g. foo_C) is given type $\exists\alpha \ll C.\ \exists\rho : (\alpha, C).$ Ptr ρ, representing pointers to any C objects, complete or embedded.

For a thunk in the vtable of an object with path P, the "this" pointer has type Ptr P, representing only pointers to objects with the same path P. In the vtable of the object with path $E :: C :: A$, the thunk for foo has "this" pointer type Ptr $(E :: C :: A)$. No subclass objects are allowed because of virtual inheritance (see Section 2.3).

Thunks adjust "this" and then call method implementations. We show the three cases of thunks with the above foo example:

Overriding Class C overrides foo with implementation foo_C. The "this" pointer in foo_C has type $\exists\alpha \ll C.\ \exists\rho : (\alpha, C).$ Ptr ρ. The thunk for the embedded A object in C adjusts "this" (with path $E :: C :: A$) to a C pointer $this_C$ and then passes it (after packing to the existential type required by foo_C) to foo_C:

$$\begin{array}{l} foo : (this : \text{Ptr}\ (E :: C :: A), \ldots)\{ \\ \quad this_C = this \ominus A; \\ \quad foo_C(\text{pack}\ this_C\ \text{to}\ \exists\alpha \ll C.\ \exists\rho : (\alpha, C).\ \text{Ptr}\ \rho,\ \ldots); \} \end{array}$$

Inheriting Class D inherits foo_A from A. The "this" pointer in foo_A has type $\exists\alpha \ll A.\ \exists\rho : (\alpha, A).$ Ptr ρ. The embedded A object in D has path $E :: D :: A$ and its vtable has a thunk for foo with "this" pointer type Ptr $(E :: D :: A)$. The thunk packs "this" and calls foo_A:

$$\begin{array}{l} foo : (this : \text{Ptr}\ (E :: D :: A), \ldots)\{ \\ \quad foo_A(\text{pack}\ this\ \text{to}\ \exists\alpha \ll A.\ \exists\rho : (\alpha, A).\ \text{Ptr}\ \rho,\ \ldots); \} \end{array}$$

Same class Class A introduces method foo with implementation foo_A. The thunk for foo in the vtable of a complete A object has "this" pointer type Ptr A. The thunk simply packs the "this" pointer and calls foo_A:

$$\begin{array}{l} foo : (this : \text{Ptr}\ A, \ldots)\{ \\ \quad foo_A(\text{pack}\ this\ \text{to}\ \exists\alpha \ll A.\ \exists\rho : (\alpha, A).\ \text{Ptr}\ \rho,\ \ldots); \} \end{array}$$

The thunks in the last two cases only change "this" pointer types. They can share the same address with foo_A to avoid calls to foo_A because "pack" is a runtime no-op.

2.3 Virtual Inheritance

With virtual inheritance, a subclass object has only one embedded object of a superclass if the subclass inherits the superclass multiple times. Figure 3 shows a class hierarchy with virtual inheritance. Dotted lines mean virtual inheritance. Both C and D virtually inherit B. Any E object has only one embedded B object, although E inherits B from both C and D. The challenge is to let the embedded C and D objects share the same B object in E. A standard strategy is to put embedded objects for virtual superclasses at the end of subclass objects. As a result, the offset between a C object and its embedded B object depends on the runtime type. The offset in a complete C object is different from the one in a complete E object. Each C object stores the offset and computes at run time the location of its embedded B object.

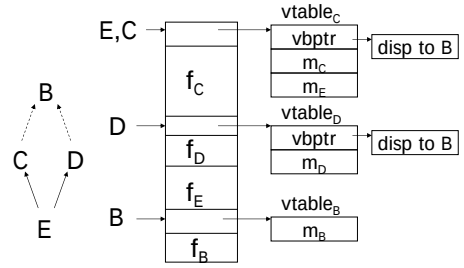

Figure 3. Virtual Inheritance

In E_{MI}, each vtable has a virtual base pointer (**vbptr**), which points to a table of offsets between the object where the vtable is fetched and embedded virtual superclass objects.[1] In the layout of class E in Figure 3, the vtable for C (or D) contains a vbptr that points to a one-entry table and the entry is the offset between C (or D) and B.

Virtual superclasses are "flattened" in the sense that, if class E inherits C (virtually or not), and C has a virtual superclass B, then E has a virtual superclass B. In Figure 3, the B object is not embedded in C, but at the same level as the C object.

Paths that involve virtual superclasses use a pseudo class name VB to separate virtual superclasses from non-virtual ones. In Figure 3, the B object in E has path $E :: \text{VB} :: B$. If E also has a non-virtual superclass B, the B object with the non-virtual inheritance would have a different path $E :: B$. Because virtual superclasses are flattened, VB must follow runtime types of objects. "$E :: C :: \text{VB} :: B$" is invalid.

Casting a subclass object to a virtual superclass includes several steps: (1) get the vtable from the object; (2) get the vbptr from the vtable; (3) get the offset that corresponds to the target in the vbptr; (4) add the offset to the object.

E_{MI} has special expressions for address arithmetic related to virtual inheritance. Expression "$\text{disp}(P_1, P_2)$" is a constant representing the offset between two objects with concrete paths (and statically known runtime types). Path P_1 represents a subclass object. Path P_2 represents a virtual superclass object and must be of format $E :: \text{VB} :: B$. The expression has type "$\text{Disp}(P_1, P_2)$". Suppose e_2 has type $\text{Disp}(P_1, P_2)$. Expression "$e_1 \uplus e_2$" adjusts e_1 with path P_1 to a virtual superclass object with path P_2, by adding e_2 to e_1. Expression "$e_1' \uplus e_2$" is the dual operation and it subtracts e_2 from e_1' if e_1' points to the virtual superclass object.

$R(E)$ and $\text{Approx}R(P)$ for the class E in Figure 3 and $P : (\tau, E)$ are defined as follows.

$$\begin{array}{l} R(E) = \\ \quad \{C : \{vtable : \text{Ptr}\{vbptr : \text{Ptr}\{B : \text{Disp}(E :: C, E :: \text{VB} :: B)\}, \\ \qquad\qquad m_C, m_E\}, f_C\}, \\ \quad\ D : \{vtable : \text{Ptr}\{vbptr : \text{Ptr}\{B : \text{Disp}(E :: D, E :: \text{VB} :: B)\}, \\ \qquad\qquad m_D\}, f_D\}, f_E, \\ \quad\ VB : \{B : \{vtable : \text{Ptr}\{m_B\}, f_B\}\}\} \\ \text{Approx}R(P) = \\ \quad \{C : \{vtable : \text{Ptr}\{vbptr : \text{Ptr}\{B : \text{Disp}(P :: C, \tau :: \text{VB} :: B)\}, \\ \qquad\qquad m_C, m_E\}, f_C\}, \\ \quad\ D : \{vtable : \text{Ptr}\{vbptr : \text{Ptr}\{B : \text{Disp}(P :: D, \tau :: \text{VB} :: B)\}, \\ \qquad\qquad m_D\}, f_D\}, f_E\} \end{array}$$

[1] Microsoft Visual C++ puts the vbptr in objects (instead of in vtables) and the offsets are from the address of the "vbptr" to the virtual superclass objects [7]. Another approach stores offsets in method entries of vtables [8, 14]. E_{MI} can model both approaches easily.

The two record types differ in "this" pointer types and in types for the offsets in vbptr. Also Approx$R(P)$ does not contain the embedded B object because the location of the B object is statically unknown. The B object is accessed through offsets in vbptr.

"This" pointer With virtual inheritance, "this" pointers are typed the same way as with repeated inheritance. Note that if a class E overrides a method foo introduced in a virtual superclass B with implementation foo_E, the thunk in the vtable of the embedded B object needs to adjust a B pointer to an E pointer to call foo_E. This adjustment is possible only when the runtime type of the B object is statically known. The B object does not contain the offset to adjust itself to E. Therefore, "this" pointers in thunks have concrete paths, not existential types that hide runtime types and paths.

The thunk in the embedded B object has a "this" pointer type Ptr $(E :: \text{VB} :: B)$. The thunk subtracts the offset disp$(E, E :: \text{VB} :: B)$ from "this" and gets an E pointer $this_E$. Then it calls foo_E after packing $this_E$:

$$
\begin{array}{l}
foo : (this : \text{Ptr } (E :: \text{VB} :: B), \ldots)\{ \\
\quad this_E = this \uplus \text{disp}(E, E :: \text{VB} :: B); \\
\quad foo_E(\text{pack } this_E \text{ to } \exists\alpha \ll E.\ \exists\rho : (\alpha, E).\ \text{Ptr } \rho,\ \ldots); \}
\end{array}
$$

3. Syntax

This section describes the formal syntax of E_{MI}. Kinds and types are as follows.

$$
\begin{array}{lcl}
\kappa & ::= & \Omega \mid \Omega_c \mid \Omega_1 \\
P & ::= & \rho \mid C \mid C :: \text{VB} :: B \mid \alpha :: \text{VB} :: B \mid P :: C \\
\tau & ::= & \text{int} \mid P \mid \text{Ptr}^\phi\ \tau \mid \{l_i^{\phi_i} : \tau_i\}_{i=1}^n \mid (\tau_1, \ldots, \tau_n) \to \tau \\
 & & \mid \alpha \mid \exists\rho : (\tau_s, \tau_e).\ \tau \mid \exists\alpha \ll \tau.\ \tau' \mid \text{Disp}(P_1, P_2) \\
\phi & ::= & I \mid M
\end{array}
$$

A special kind Ω_c classifies class names and type variables that will be instantiated with class names. Well-formedness of types requires that certain types have kind Ω_c, for example, the bounds of type variables and path variables. Kind Ω_1 classifies word-sized types to guarantee that the heap is updated one word at a time. Ω_c and Ω_1 are subkinds of Ω.

Paths are used to type objects. A path is a path variable ρ, a class name C, a path from a class to a virtual superclass $C :: \text{VB} :: B$, a path from a type variable to a virtual superclass $\alpha :: \text{VB} :: B$, or appending a class name to a path $P :: C$. A path can only be a sequence of class names, or a path variable or a type variable followed by a sequence of class names. VB must appear between the runtime type of an object (a class name or a type variable) and a virtual superclass (another class name).

E_{MI} has standard types such as pointer type "Ptr$^\phi$ τ", record type "$\{l_i^{\phi_i} : \tau_i\}_{i=1}^n$", and function type "$(\tau_1, \ldots, \tau_n) \to \tau$" to represent object layout, vtable, and virtual methods.

Pointers and fields in record types have mutability annotations. Ptr^M τ means the pointed-to value can be modified to a new value of type τ. The annotation I means immutable. Annotations on field labels mean mutability of the corresponding fields. In E_{MI}, pointers to (embedded) objects, the vtable pointer, fields in the vtable, and the "this" pointers are immutable. All other fields and pointers are mutable. We often omit the annotation I.

E_{MI} uses existential types that abstract unknown types and paths to represent objects. A source type C is translated to an existential type $\exists\alpha \ll C.\ \exists\rho : (\alpha, C).\ \rho$ in E_{MI}. Type "$\exists\alpha \ll \tau.\ \tau'$" introduces a type variable α with a subclassing bound τ. Type "$\exists\rho : (\tau_s, \tau_e).\ \tau$" introduces a path variable ρ from τ_s to τ_e. Universal types are not needed in E_{MI} and can be added easily. Type "Disp(P_1, P_2)" represents the offset between two embedded objects with paths P_1 and P_2 respectively. It is used to type offsets in the virtual base table.

Expressions and values are as follows.

$$
\begin{array}{lcl}
e & ::= & x \mid n \mid null^\tau \mid \ell \mid \text{new}[\tau] \mid e.l \mid *e \mid *e_1 := e_2 \text{ in } e_3 \\
 & & \mid x : \tau = e_1 \text{ in } e_2 \mid e(e_1, \ldots, e_n) \mid C(e) \mid \text{c2r}(e) \\
 & & \mid (\alpha, x) = \text{open}(e_1) \text{ in } e_2 \mid (\rho, x) = \text{open}(e_1) \text{ in } e_2 \\
 & & \mid \text{pack } \tau \text{ as } \alpha \ll \tau_u \text{ in } (e : \tau') \\
 & & \mid \text{pack } P \text{ as } \rho : (\tau_s, \tau_e) \text{ in } (e : \tau) \mid e \bullet P \\
 & & \mid e \oplus C \mid e \ominus C \mid e_1 \uplus e_2 \mid e_1 \cup\!\!\!\!-\, e_2 \mid \text{disp}(P_1, P_2) \\
v & ::= & n \mid \ell \mid C(v) \bullet P \mid v.l \mid \text{pack } \tau \text{ as } \alpha \ll \tau_u \text{ in } (v : \tau') \\
 & & \mid null^\tau \mid \text{pack } P \text{ as } \rho : (\tau_s, \tau_e) \text{ in } (v : \tau) \mid \text{disp}(P_1, P_2) \\
hv & ::= & recd \mid \text{fix } g(x_i : \tau_i)_{i=1}^n : \tau = e_m \\
recd & ::= & \{l_i = v_i \mid l_i = recd_i\}_{i=1}^n
\end{array}
$$

E_{MI} has standard expressions such as variable "x", integer "n", null pointer "$null^\tau$", label "ℓ", record allocation "new$[\tau]$", field fetch "$e.l$", pointer dereference "$*e$", assignment "$*e_1 := e_2 \text{ in } e_3$", let binding "$x : \tau = e_1 \text{ in } e_2$", and function call "$e(e_1, \ldots, e_n)$". Record allocation "new$[\tau]$" allocates a record of type τ on the heap and initializes each field with the default value of the field type. Expression "$e.l$" returns an interior pointer to the field l of a record pointed to by e. Expression "$*e_1 := e_2 \text{ in } e_3$" assigns e_2 as the new content of pointer e_1. E_{MI} uses "$*(e.l)$" to return the value of the field l of e and "$*(e_1.l) := e_2 \text{ in } e_3$" to assign the field.

Expressions "$C(e)$" and "c2r(e)" are coercions between object pointers and record pointers. The former coerces a record pointer e to a C pointer and the latter coerces an object pointer to a record pointer. To create a C object, we allocate a record of type $R(C)$ using "new$[R(C)]$", assign values to fields, and coerce the record pointer to an object pointer using "$C(e)$". To fetch a field or to call a method, we first use "c2r(e)" to coerce the object pointer e to a record pointer. These coercions have no runtime cost.

The pack and open expressions introduce and eliminate existential types respectively. Expressions "pack τ as $\alpha \ll \tau_u$ in $(e : \tau')$" hides a type τ bounded by τ_u with a type variable α in e. Expression "$(\alpha, x) = \text{open}(e_1) \text{ in } e_2$" opens a package e_1 and introduces α for the hidden type. Both α and x are in scope in e_2. Expression "pack P as $\rho : (\tau_s, \tau_e)$ in $(e : \tau)$" hides path P from τ_s to τ_e with path variable ρ in expression e. Expression "$(\rho, x) = \text{open}(e_1) \text{ in } e_2$" opens e_1 and introduces a path variable ρ for the hidden path and a value variable x for e_1.

Address arithmetic expressions "$e \oplus C$", "$e \ominus C$", "$e_1 \uplus e_2$", "$e_1 \cup\!\!\!\!-\, e_2$", and "disp(P_1, P_2)" are used for adjusting between superclass and subclass objects. The former two are for non-virtual superclasses. The rest are for virtual superclasses. Expression "$e \bullet P$" represents an embedded object in e following path P.

Records are allocated on the heap. Each field in a record "$\{l_i = v_i \mid l_i = recd_i\}_{i=1}^n$" is a word-sized value v_i or an inlined record $recd_i$. For simplicity E_{MI} does not support stack-allocated objects.

A function "fix $g(x_i : \tau_i)_{i=1}^n : \tau = e_m$" defines g with formals $x_1, \ldots, x_n$ (of type $\tau_1, \ldots, \tau_n$ respectively), return type τ and function body e_m. The body e_m may call g recursively.

Class and program declarations are as follows. The notation $\overrightarrow{\Psi}$ means a sequence of items in Ψ.

$$
\begin{array}{lcl}
field & ::= & f : \tau \qquad method ::= m : (\tau_1, \ldots, \tau_n) \to \tau \\
H & ::= & \overrightarrow{\ell \rightsquigarrow hv} \qquad class ::= C : \overrightarrow{A}, \text{virtual } \overrightarrow{B}\{\overrightarrow{field}, \overrightarrow{method}\} \\
Prog & ::= & (\overrightarrow{class}; H; e)
\end{array}
$$

A class declaration "$C : \overrightarrow{A}$, virtual $\overrightarrow{B}\{\overrightarrow{field}, \overrightarrow{method}\}$" declares a class C with *direct* non-virtual superclasses $\overrightarrow{A}$, *direct* virtual superclasses $\overrightarrow{B}$, fields $\overrightarrow{field}$, and methods $\overrightarrow{method}$. A field declaration "$f : \tau$" declares a field f with type τ. A method declaration "$m : (\tau_1, \ldots, \tau_n) \to \tau$" declares a method m with formal types $\tau_1, \ldots, \tau_n$ and return type τ. Method declarations do not include explicit "this" pointer types. Method bodies are represented as functions on the heap, not in class declarations.

A program declaration "$(\overrightarrow{class}; H; e)$" declares a program with class declarations $\overrightarrow{class}$, a heap H, and the "main" expression e. The heap H maps labels to heap values.

4. Semantics

This section formalizes the semantics and properties of E_{MI}.

4.1 Dynamic Semantics

Figure 4 shows selected evaluation rules. The programs in the first column evaluate to the ones in the second column, if the side conditions in the third column hold. Class declarations in programs are omitted because they do not change during evaluation. The notation "$[\delta_1/\delta_2]$" means replacing δ_2 with δ_1.

The notation "$C(v) \bullet P$" represents a pointer to the object with path P in a complete C object. Given $P = C :: C_1 :: \ldots :: C_n$, the object pointer models the same address as an interior pointer $v.C_1 \ldots C_n$. As explained in Section 2, the label sequence $C_1, \ldots, C_n$ leads to the part of the record that represents the embedded object with path P. The two pointers have different types.

Expression "new$[\tau]$" allocates a record of type τ (with value $default_\tau$) on the heap and returns a label to the record. The default value for the vtable field in $R(C)$ is a pointer to the vtable of C. Expression "$C(v)$" coerces the record label to a C pointer "$C(v).C$". Expression "c2r" coerces an object pointer $C(v) \bullet P$ to an interior pointer as described above.

Expression "$*(v.C_1 \ldots C_n.f)$" fetches the value of a field f in the embedded object and "$*(v.C_1 \ldots C_n.f) := v_2$ in e_3" assigns v_2 to the field and evaluates e_3.

Expression "$(C(v) \bullet P) \oplus A$" adjusts a pointer $C(v) \bullet P$ to $C(v) \bullet (P :: A)$, which points to the embedded object for a *non-virtual* superclass A. Expression "$(C(v) \bullet (P :: A)) \ominus A$" adjusts the A pointer back. Similarly, expression "$(C(v) \bullet P_1) \uplus \mathrm{disp}(P_1, P_2)$" adjusts $C(v) \bullet P_1$ to a *virtual* superclass pointer $C(v) \bullet P_2$. Expression "$(C(v) \bullet P_2) \uplus \mathrm{disp}(P_1, P_2)$" adjusts back to $C(v) \bullet P_1$.

Pointer-related expressions such as dereference, assignment, and address arithmetic infinitely loop if the pointers are null. Future versions of E_{MI} will use exceptions.

4.2 Static Semantics

The type checker maintains several environments. A class declaration table Θ maps class names to declarations. A kind environment Δ tracks type and path variables. Each type variable has a subclassing upper bound (a class name or a type variable introduced previously in Δ). Each path variable has a starting type and an ending type. A heap environment Σ maps labels to types. A type environment Γ maps variables to types.

Well-formedness rules of paths are as follows. The judgment $\Theta; \Delta \vdash P : (\tau_s, \tau_e)$ means that, under environments Θ and Δ, P is a well-formed path from τ_s to τ_e. If VB appears in the path, a virtual superclass must follow immediately. Paths are continuous: a path $P : (\tau, C)$ can be concatenated only with a direct non-virtual superclass of C.

$$\frac{\rho : (\tau_1, \tau_2) \in \Delta}{\Theta; \Delta \vdash \rho : (\tau_1, \tau_2)} \qquad \frac{B \text{ is a virtual superclass of } \tau}{\Theta; \Delta \vdash \tau :: \mathrm{VB} :: B : (\tau, B)}$$

$$\frac{}{\Theta; \Delta \vdash C : (C, C)} \qquad \frac{\Theta; \Delta \vdash P : (\tau, C) \quad \Theta(C) = \overrightarrow{A}, \_\{\_\} \quad A_i \in \overrightarrow{A}}{\Theta; \Delta \vdash P :: A_i : (\tau, A_i)}$$

Well-formedness of types requires that the bounds for type variables and the starting and the ending types of paths be of kind Ω_c. Type $\mathrm{Disp}(P_1, P_2)$ requires that P_1 and P_2 start from the same type and that P_2 lead to a virtual superclass (of the format $\tau :: VB :: C$). The judgment $\Theta; \Delta \vdash \tau : \kappa$ means that, under environments Θ and Δ, type τ has kind κ.

Subclassing rules are straightforward. The subclassing judgment $\Theta; \Delta \vdash \tau_1 \ll \tau_2$ means that, under environments Θ and Δ, τ_1 is a subclass of τ_2. Subclassing between class names preserves the class hierarchy in the source programs. Subclassing is reflexive and transitive.

Subtyping in E_{MI} includes standard record breadth subtyping and depth subtyping (on immutable fields), function subtyping, and pointer subtyping (on pointers to immutable values). Subtyping is reflexive and transitive. The subtyping judgment $\Theta; \Delta \vdash \tau_1 \leq \tau_2$ means that, under environments Θ and Δ, τ_1 is a subtype of τ_2.

The structural subtyping guarantees validity of layout approximation. Subtyping between quantified types is unnecessary because inheritance is represented by explicit pointer adjustments.

Figure 5 shows selected expression typing rules. The typing judgment $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ means that, under environments Θ, Δ, Σ, and Γ, expression e has type τ.

A pointer to a complete C object has type Ptr C. A pointer to an object with path P has type Ptr P.

Coercions A record pointer of type Ptr $(R(C))$ can be coerced to and from an object pointer of type Ptr C by expressions $C(e)$ and $c2r(e)$ respectively. If e has type Ptr P, that is, e pointers to an embedded object with path P, $c2r(e)$ returns a pointer to a record of type Approx$R(P)$. The definitions of R and Approx$R$ for the current layout strategy are in [2].

The compiler has the freedom to choose layout strategies and to use other definitions of R and Approx$R$, although the layout information is part of the type system. The soundness of the type system requires only that for any path $P : (C, A)$, the real layout of objects with P be a subtype of Approx$R(P)$ (P may contain VB).

Address arithmetic Expressions $e \oplus A$ and $e \ominus A$ adjust between an object pointer with type Ptr P and a pointer with type Ptr $(P :: A)$. Expressions $e_1 \uplus e_2$ and $e_1 \uplus e_2$ adjust between an object pointer with type Ptr P_1 and a pointer with type Ptr P_2, if e_2 has type $\mathrm{Disp}(P_1, P_2)$. When both paths are concrete, expression "$\mathrm{disp}(P_1, P_2)$" has type $\mathrm{Disp}(P_1, P_2)$ and requires that P_2 lead to a virtual superclass.

A program $(\Theta; H; e)$ is well-formed if its class declaration Θ is well-formed, each heap value has the corresponding type in the heap environment, and the main expression e is well-typed.

4.3 Properties of $\mathbf{E}_{MI}$

E_{MI} has the following properties.

Theorem 1 (**Soundness**) If a program is well-formed, then it will not get stuck.

Theorem 2 (**Decidable type checking**) It is decidable whether $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ holds.

5. Related Work

Prior work on supporting typed compilation of multiple inheritance cannot describe standard implementation techniques. None supports virtual inheritance. The model proposed by Chen *et al.* preserves class names and uses paths to identify embedded objects [1]. It encodes objects with functions and cannot address object layout. ML-ART uses row variables to abstract class extensions and serializes multiple inheritance to a sequence of single inheritance [10]. Fisher *et al.* proposed an *untyped* calculus to support inheritance from unknown base classes [6]. They use dictionary lookups for member access. Stone used indices (offsets) as first-class values to fetch members from objects so that classes can be extended without full knowledge of the base classes [13]. Compagnoni and Pierce used intersection types to model multiple inheritance [5]. The language does not preserve classes and subclassing. "This" pointer adjustment is hidden by intersection types and cannot be expressed.

Original Program	New Program	Side Conditions
$(H; \text{new}[\tau])$	$(H, \ell \rightsquigarrow default_\tau; \ell)$	$\ell \notin H$
$(H; C(v))$	$(H; C(v) \bullet C)$	
$(H; \text{c2r}(C(v) \bullet P))$	$(H; v.C_1.\ldots.C_n)$	$P = C :: C_1 :: \ldots :: C_n$
$(H; *(\ell.l_1.\ldots.l_n))$	$(H; v)$	$H(\ell) = \{\ldots, l_1 = \{\ldots, l_n = v \ldots\}, \ldots\}$
$(H; *(\ell.l_1.\ldots.l_n) := v' \text{ in } e_3)$	$(H'; e_3)$	$H' = H$ except that $H'(\ell).l_1.\ldots.l_n = v'$
$(H; (C(v) \bullet P) \oplus A)$	$(H; C(v) \bullet (P :: A))$	
$(H; (C(v) \bullet P) \ominus A)$	$(H; C(v) \bullet P')$	$P = P' :: A$
$(H; (C(v) \bullet P_1) \uplus v_2)$	$(H; C(v) \bullet P_2)$	$v_2 = \text{disp}(P_1, P_2)$
$(H; (C(v) \bullet P_2) \uplus v_2)$	$(H; C(v) \bullet P_1)$	$v_2 = \text{disp}(P_1, P_2)$
$(H; (\alpha, x) = \text{open}(v) \text{ in } e)$	$(H; e[\tau/\alpha][v'/x])$	$v = \text{pack } \tau \text{ as } \beta \ll \_ \text{ in } (v' : \_)$
$(H; (\rho, x) = \text{open}(v) \text{ in } e)$	$(H; e[P/\rho][v'/x])$	$v = \text{pack } P \text{ as } \rho : \_ \text{ in } (v' : \_)$

Figure 4. Selected Evaluation Rules

$$\frac{}{\Theta;\Delta;\Sigma;\Gamma \vdash \ell : \text{Ptr}(\Sigma(\ell))} \quad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \text{Ptr}(R(C))}{\Theta;\Delta;\Sigma;\Gamma \vdash C(e) : \text{Ptr } C} \quad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \text{Ptr } C}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{c2r}(e) : \text{Ptr}(R(C))} \quad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \text{Ptr } P \quad \Theta;\Delta \vdash P : (\tau, C)}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{c2r}(e) : \text{Ptr}(\text{Approx}R(P))}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \exists\rho' : (\tau_s, \tau_e).\ \tau \quad \rho \notin \text{domain}(\Delta) \quad \rho \notin free(\tau') \quad \Theta;\Delta,\rho : (\tau_s,\tau_e);\Sigma;\Gamma, x : \tau[\rho/\rho'] \vdash e_2 : \tau'}{\Theta;\Delta;\Sigma;\Gamma \vdash (\rho, x) = \text{open}(e_1) \text{ in } e_2 : \tau'} \quad \frac{\Theta;\Delta \vdash P : (\tau_s,\tau_e) \quad \rho \notin \text{domain}(\Delta) \quad \Theta;\Delta;\Sigma;\Gamma \vdash e : \tau'[P/\rho]}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{pack } P \text{ as } \rho : (\tau_s,\tau_e) \text{ in } (e : \tau') : \exists\rho : (\tau_s,\tau_e).\ \tau'}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \text{Ptr } P_1 \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \text{Disp}(P_1, P_2)}{\Theta;\Delta;\Sigma;\Gamma \vdash e_1 \uplus e_2 : \text{Ptr } P_2} \quad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \text{Ptr } P_2 \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \text{Disp}(P_1, P_2)}{\Theta;\Delta;\Sigma;\Gamma \vdash e_1 \uplus e_2 : \text{Ptr } P_1} \quad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \text{Ptr } C \quad \Theta;\bullet \vdash P : (C, \tau)}{\Theta;\Delta;\Sigma;\Gamma \vdash e \bullet P : \text{Ptr } P} \quad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \text{Ptr } P \quad \Theta;\Delta \vdash P : (\tau, C) \quad A \text{ is a direct non-virtual superclass of } C}{\Theta;\Delta;\Sigma;\Gamma \vdash e \oplus A : \text{Ptr}(P :: A)}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \text{Ptr}(P :: A)}{\Theta;\Delta;\Sigma;\Gamma \vdash e \ominus A : \text{Ptr } P} \quad \frac{\Theta;\bullet \vdash P_1 : (C, \tau_1) \quad \Theta;\bullet \vdash C :: \text{VB} :: B : (C, B)}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{disp}(P_1, C :: \text{VB} :: B) : \text{Disp}(P_1, C :: \text{VB} :: B)}$$

Figure 5. Selected Expression Typing Rules

LIL_{CI} extends LIL_C to support multiple inheritance of interfaces as in Java and C# [3]. Interfaces do not have object layout problems because they contain no fields.

Rossie *et al.* [11] and Wasserrab *et al.* [16] formalized multiple inheritance in C++ at the source language level. The formalizations refer to no implementation details such as object layout and vbptr.

6. Conclusion

E_{MI} is a typed intermediate language for compiling multiple inheritance, both repeated and shared inheritance. It uses paths to type objects, existential types to represent path abstractions, and special address arithmetic to model pointer adjustment. The type system is sound. The type checking is decidable. The translation from a source language to E_{MI} preserves types. E_{MI} can express standard implementation strategies of multiple inheritance in C++, including object layout, "this" pointer, pointer adjustment, and virtual base pointer.

References

[1] C. Chen, R. Shi, and H. Xi. A typeful approach to object-oriented programming with multiple inheritance. In *Proc. 6th PADL*, pages 23–38, 2004.

[2] J. Chen. A typed intermediate language for compiling multiple inheritance. Technical Report MSR-TR-2005-98, Microsoft Corporation.

[3] J. Chen and C. Chen. A typed intermediate language for supporting multiple inheritance via interfaces. Technical Report MSR-TR-2004-141, Microsoft Corporation.

[4] J. Chen and D. Tarditi. A simple typed intermediate language for object-oriented languages. In *Proc. 32nd POPL*, pages 38–49, 2005.

[5] A. B. Compagnoni and B. C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[6] K. Fisher, J. H. Reppy, and J. G. Riecke. A calculus for compiling and linking classes. In *Proc. 9th ESOP*, pages 135–149, 2000.

[7] J. Gray. C++: under the hood. http://msdn.microsoft.com/archive/default.asp? url=/archive/en-us/dnarvc/html/jangrayhood.asp.

[8] S. B. Lippman. *Inside the C++ object model*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[9] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Prog. Lang. Syst.*, 21(3):527–568, May 1999.

[10] D. Rémy. Programming objects with ML-ART, an extension to ML with abstract and record types. In *Proc. International Conference on Theoretical Aspects of Computer Software*, pages 321–346, 1994.

[11] J. G. Rossie and D. P. Friedman. An algebraic semantics of subobjects. In *Proc. OOPSLA*, pages 187–199, 1995.

[12] Z. Shao. An overview of the FLINT/ML compiler. In *ACM SIGPLAN Workshop on Types in Compilation*, 1997.

[13] C. A. Stone. Extensible objects without labels. *ACM Trans. Prog. Lang. Syst.*, 26(5):805–835, 2004.

[14] B. Stroustrup. Multiple inheritance for C++. In *Proc. of the European Unix Users Group Conference*, Helsinki, 1987.

[15] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[16] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *Proc. OOPSLA*, 2006.