A Typed Intermediate Language for Supporting Interfaces

Juan Chen
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
juanchen@microsoft.com

ABSTRACT

Object-oriented languages such as Java and C# provide interfaces to support a restricted form of multiple inheritance. Existing low-level typed intermediate languages for object-oriented languages, however, either do not support interfaces or require non-standard interface implementations. This paper describes a low-level typed intermediate language that can express the standard interface implementation strategies based on interface tables (itables). The language can faithfully model itables, the standard itable-based interface method invocation, and interface cast. The type system is sound and the type checking is decidable.

1. INTRODUCTION

Object-oriented languages that allow only single inheritance between classes, e.g. Java and C#, use interfaces to support a restricted form of multiple inheritance. Compilation from the source languages (Java and C#) to typed intermediate languages (Java bytecode [16] and .NET CIL [11]) preserves types, including interfaces, so that we can verify bytecode and CIL to guarantee type safety. Proof-Carrying Code (PCC) [18] and Typed Assembly Language (TAL) [17] push types further to lower-level intermediate languages, even assembly languages. This way, we can verify type safety and memory safety of assembly language programs and remove the compiler (which is often a large and complicate piece of software) from the trusted computing base. PCC and TAL require that the type systems of the low-level intermediate languages be able to express all features in those languages. Existing lower-level typed intermediate languages for object-oriented languages, however, either do not support interfaces or require non-standard and inefficient implementations.

This paper explains ECI (Encoding for Classes and Interfaces), a low-level typed intermediate language that supports both classes and interfaces. ECI is based on LIL_C, a language that addresses only classes and single inheritance between classes [6]. ECI faithfully models the standard implementation techniques based on interface tables (*itables*), itable-based interface method invocation, and interface cast. It is the first typed intermediate language that is able to express these details.

Interfaces are difficult to support because of multiple inheritance. An interface may have different offsets in itables of different classes. Interface method invocation has to look up the itable at run time for the desired interface. ECI uses arrays and subclassing-bounded quantification to represent itables. Interface method invocation and interface cast are expressed as polymorphic functions in ECI. They can be inlined into ECI code compiled from source programs, and the result program can be optimized and type-checked because interface method invocation and interface cast are expressed entirely in ECI.

ECI is sound and its type checking is decidable. The proofs are in a companion technical report [5]. The technical report also formalizes a source language and a type-preserving translation from the source language to ECI. The source language is roughly Featherweight Java [13] with extensions of interfaces and arrays. The technical report also discusses how ECI may be extended to express advanced interface implementation techniques [14, 12, 8, 9, 10, 1].

The rest of the paper is organized as follows. Section 2 gives an informal overview of ECI. Sections 3 and 4 explain the syntax and semantics of ECI respectively. Sections 5 discusses related work and Section 6 concludes.

2. OVERVIEW

We first give an informal overview of ECI. For clarity, this paper focuses on only core object-oriented features that are related to interfaces. Features such as generics, non-virtual methods, and null pointers are omitted.

Section 2.1 summarizes the key ideas of the base language LIL_C. We refer readers to the previous paper [6] for details of virtual method invocation, class cast, and arrays. Section 2.2 explains how ECI represents itables. Section 2.3 describes object layout. Section 2.4 addresses interface method invocation and interface cast.

2.1 Key Ideas of LIL_C

LIL_C is a low-level typed intermediate language for compiling object-oriented languages with classes. It is lower-level than bytecode and CIL because it describes implementation of virtual method invocation and type cast instead of treating them as primitives.

LIL_C differs from prior class and object encodings in that it preserves object-oriented notions such as class names and

FTfJP '09, July 6 2009, Genova, Italy

name-based subclassing, whereas prior encodings compiled these notions away.

LIL_C uses an "exact" notion of classes. A class name in LIL_C represents only objects of exactly that class, not including objects of subclasses. Each class C has a corresponding record type $\mathsf{R}(C)$ that describes the object layout of C, including the vtable and the fields.

Objects can be coerced to or from records with appropriate record types, without runtime overhead. To create an object, we create a record and coerce it to an object. To fetch a field or invoke a method, we coerce the object to a record and then fetch the field or the method pointer from the record.

To represent an object of C or C's subclasses, LIL_C uses an existential type $\exists\alpha \ll C.\ \alpha$, read as "there exists α where α is a subclass of C, and the object has type α". The type variable α represents the object's runtime type and the notation $\ll$ means subclassing. The type variable α has a subclassing bound C, which means that the runtime type of the object is a subclass of C. The subclassing bounds can only be class names or type variables that will be instantiated with class names. Subclassing-bounded quantification and the separation between name-based subclassing and structure-based subtyping make the type system decidable and expressive.

A source class name C is then translated to an LIL_C type $\exists\alpha \ll C.\ \alpha$. If C is a subclass of class B, then $\exists\alpha \ll C.\ \alpha$ is a subtype of $\exists\alpha \ll B.\ \alpha$, which expresses inheritance—objects of C or C's subclasses can be used as objects of B or B's subclasses.

A class C has a unique identifier (called tag) represented as $tag(C)$. The tag has type $\text{Tag}(C)$. If two tags are equal, the corresponding classes are the same.

2.2 Itable Representation

ECI extends the key ideas of LIL_C naturally to interfaces. ECI preserves interface names (ranged over by I, J, and K). Subclassing applies to interfaces: $C \ll I$ represents that class C implements interface I, and $I \ll J$ represents that interface I is a subinterface of interface J. Type variables can be bounded by and be instantiated with interface names. A source interface name I is translated to an ECI type $\exists\alpha \ll I.\ \alpha$, meaning objects of classes that implement I. Interface I has a unique identifier $tag(I)$, which has type $\text{Tag}(I)$.

The key data structure of itable-based interface implementation is the itable. In a common itable representation, a class includes in its itable an entry for each interface the class implements. Each interface entry contains two words: a tag and a pointer to the method table for the corresponding interface. The method table contains a function pointer for each method declared in the interface. The order of function pointers in the method table is significant: the order is the same in all itable entries that correspond to the same interface. This allows the compiler to use the same offset for a method in the method table of an interface, no matter in which class.

Figure 1 illustrates the itable of a class C that implements two interfaces J and K, each declaring an interface method m_J and m_K respectively. The itable has two entries, one for J and the other for K. The method table for J contains only one function pointer, the one for m_J. The method table pointer is given an ECI type $\text{Imty}(J, C)$, representing

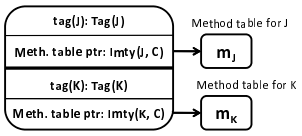

Figure 1: Itable of a Class C Implementing J and K

the method table for interface J in class C's itable. Method table type $\text{Imty}(J, C)$ can be coerced to/from a record type $\{m_J : \text{type-of-}m_J\}$, which contains a pointer to m_J's implementation in C. Again, the coercions are runtime no-ops. The method table type refers to class C, indicating that the implementation of m_J in the method table can be applied only to objects of C or C's subclasses. The class name C is used in the "this" pointer type of m_J ($\exists\alpha \ll C.\ \alpha$). The method table for K is similar.

For an object whose runtime type is statically unknown, we cannot determine statically what interfaces its itable contains. Furthermore, the offsets of itable entries cannot be pre-determined because of multiple inheritance of interfaces. To invoke an interface method, we need to look up the itable at run time for the entry of the target interface.

To represent an itable that holds statically unknown interfaces with unknown ordering, ECI uses an array type, where each element corresponds to an itable entry. Each entry has two words, represented by a record type with two fields. ECI uses an existential type to abstract the interface to which each entry corresponds. An itable entry for class C has type $\exists\alpha \gg C.\ \{tag : \text{Tag}(\alpha), mtable : \text{Imty}(\alpha, C)\}$. The type variable α indicates the interface to which the entry corresponds. C implements the interface, therefore $C \ll \alpha$ and α is given the lower bound C. The body of the existential type is a record type with two fields labeled tag and $mtable$, meaning the tag for the interface α and the method table respectively.

The itable entry array for class τ then has type $\text{array}(\exists\alpha \gg \tau.\{tag : \text{Tag}(\alpha), mtable : \text{Imty}(\alpha, \tau)\})$, often abbreviated as $ITY(\tau)$. The itable of class τ has a record type $\{length : \text{int}, table : ITY(\tau)\}$. The field "length" records the number of itable entries (used by itable search). The field "table" is the entry array.

Interface method invocation searches the itable for the target interface by comparing the tag in each entry with the tag of the target interface. If an entry for the target interface is found, the method table in the entry is coerced to the record type that contains methods of the target interface. The desired interface method can be fetched at a pre-determined offset from the record type.

Interface cast searches the itable in a similar way. Suppose the itable of an object with runtime type τ contains an entry for interface I. The subclassing bound in the itable entry type indicates that $I \gg \tau$. Thus the object can be cast to interface I.

2.3 Object Layout

Object layout—the organization of fields, methods, and runtime tags—in ECI is standard. Each object contains a pointer to the vtable and the fields. The vtable contains the tag, a pointer to the itable, and virtual method pointers. The rest of the paper uses records and pointers to records interchangeably. It should be clear from the context which one is being used.

Suppose a class C has fields $f_1, \ldots, f_n$ of types $s_1, \ldots, s_n$ respectively, and virtual methods $m_1, \ldots, m_k$ of types $t_1, \ldots, t_k$ (with explicit "this" pointer types) respectively. The layout of C is represented by the following record type:

$$\begin{aligned}\mathsf{R}(C) = \{&vtable : \{tag : \mathrm{Tag}(C),\\ &\quad itable : \{length : \mathrm{int}, table : ITY(C)\},\\ &\quad m_1 : t_1, \ldots, m_k : t_k\},\\ &f_1 : s_1, \ldots, f_n : s_n\}\end{aligned}$$

The tag of C has type $\mathrm{Tag}(C)$. The itable type is explained in Section 2.2. Each virtual method in the vtable has an explicit "this" pointer as the first parameter. The "this" pointer has type $\exists\alpha \ll C.\ \alpha$ to guarantee that only objects of C or C's subclasses can be passed to the methods.

2.3.1 Approximation of Object Layout

Layouts of objects whose runtime types are statically unknown can be approximated. Suppose an object has runtime type α where α is a subclass of the above class C. The object contains at least the fields and methods of C. The object layout can be represented as follows:

$$\begin{aligned}&\mathsf{ApproxR}(\alpha, C) =\\ &\quad\{vtable : \{tag : \mathrm{Tag}(\alpha),\\ &\qquad itable : \{length : \mathrm{int}, table : ITY(\alpha)\},\\ &\qquad m_1 : t_1, \ldots, m_k : t_k\},\\ &\quad f_1 : s_1, \ldots, f_n : s_n\}\end{aligned}$$

For objects with source type I, the approximation record contains only the tag and the itable, no fields or methods. We cannot represent the entry for I in the itable because we cannot statically determine where the entry is located or what other entries the itable has.

$$\begin{aligned}&\mathsf{ApproxR}(\alpha, I) =\\ &\quad\{vtable : \{tag : \mathrm{Tag}(\alpha),\\ &\qquad itable : \{length : \mathrm{int}, table : ITY(\alpha)\}\}\}\end{aligned}$$

R and ApproxR are not type constructors in ECI, but macros used by the type checker. ECI can adopt other layout strategies by changing the macros.

Objects of type C (or α where $\alpha \ll C$) are coerced to records of type $\mathsf{R}(C)$ (or $\mathsf{ApproxR}(\alpha, C)$) before field fetch or virtual method invocation.

2.4 Interface Method Invocation and Interface Cast

We can now describe the itable lookup process during interface method invocation as a polymorphic function in ECI (Figure 2). Note that without coercions (which are runtime no-ops anyway), it is exactly the standard implementation.

The function *ILookup* takes two type parameters—the target interface α and the runtime type β of the object, and two value parameters—the tag of the target interface and the object. *ILookup* returns the method table for the target interface α in class β if α is found. Otherwise an error occurs.

Step 1) coerces *obj* to a record c2r(*obj*) and fetches the vtable and then the itable from the record. Step 2) gets the entry array from the itable. Step 3) gets the length of the itable and step 4) calls *loop* to iterate over the entry array.

Function *loop* is a polymorphic function with the same type parameters as *ILookup*. It has three additional parameters: *arr* for the interface entry array, i for the index of the current entry, and *len* for the length of the itable. Step 5) tests if the search reaches the end of the itable. If so, an error is returned. Otherwise, step 6) fetches the ith entry from the entry array. Step 7) opens the entry and introduces a fresh type variable γ' for the interface corresponding to the entry. Step 8) compares the tag in the entry (*entryi'.tag* with type $\mathrm{Tag}(\gamma')$) with the target interface's tag (t_α with type $\mathrm{Tag}(\alpha)$). If the tags are the same, then $\gamma' = \alpha$. The method table in the entry (with type $\mathrm{Imty}(\gamma', \beta)$) is returned as a value of type $\mathrm{Imty}(\alpha, \beta)$. Otherwise, step 9) calls *loop* for the next iteration.

The following example demonstrates interface method invocation—how to invoke the method m_J on an object *obj* with source type J in Figure 1:

```
    //open object. obj : ∃α ≪ J. α,  obj' : β
1)  (β, obj') = open(obj)
    //lookup itable. mtable : Imty(J, β)
2)  mtable = ILookup[J, β](tag(J), obj')
    //coerce mtable. r : {m_J : type_of_m_J}
3)  r = im2r(mtable)
    //fetch method. m : type_of_m_J
4)  m = r.m_J
    //invoke the method.
5)  m(obj')
```

The source type J is translated to ECI type $\exists\alpha \ll J.\ \alpha$, the type of *obj*. Step 1) opens the existential type and introduces a fresh type variable β for the runtime type of *obj*. The opened object *obj'* is an alias of *obj* and has type β. Step 2) calls *ILookup* to search for the interface J. The return value is the method table for J in class β, which has type $\mathrm{Imty}(J, \beta)$. Step 3) then coerces the method table to a record r that lists all J's methods. Step 4) fetches the method m_J from r and step 5) calls the method with *obj'* as the "this" pointer.

ECI can express without difficulties optimizations of itable lookup such as caching and move-to-front. The caching strategy caches the last entry looked up successfully in the itable. The cached entry can have the same type as the itable entries. The move-to-front strategy moves the last found entry to the front of the itable. It can also be expressed because ECI's array representation of itables does not care about the orders of itable entries.

2.4.1 Interface Cast

Interface downward cast may cast an object of an arbitrary (class or interface) type to an interface. A typical implementation of interface downward cast searches at run time for an itable entry that corresponds to the target interface. The downward cast is represented by a polymorphic function similar to *ILookup* (see the companion technical report).

3. SYNTAX OF ECI

This section explains the syntax of types and expressions in ECI. Class, interface, and program declarations are explained in the technical report. We underline new interface-related constructs to distinguish them from those introduced in the base language LIL_C.

$$fix\ ILookup\langle\alpha,\beta\rangle \left(\begin{array}{ll} t_\alpha : \mathrm{Tag}(\alpha), & //\text{tag of target interface } \alpha \\ obj : \beta & //object \end{array}\right) : \mathrm{Imty}(\alpha,\beta)$$

1) $itable : \{length : \mathrm{int}, table : ITY(\beta)\} = \mathrm{c2r}(obj).vtable.itable$ in //get itable
2) $arr : ITY(\beta) = itable.table$ in //get entry array
3) $len : \mathrm{int} = itable.length$ in //get itable length
4) $loop[\alpha,\beta](t_\alpha, obj, arr, 0, len)$ //start search

$$fix\ loop\langle\alpha,\beta\rangle \left(\begin{array}{ll} t_\alpha : \mathrm{Tag}(\alpha), & //\text{tag of target interface } \alpha \\ obj : \beta, & //\text{object of type } \beta \\ arr : ITY(\beta), & //\text{interface entry array} \\ i : \mathrm{int}, & //\text{index of the current entry} \\ len : \mathrm{int} & //\text{the array length} \end{array}\right) : \mathrm{Imty}(\alpha,\beta)$$

5) if$(i \geq len)$ then error$[\mathrm{Imty}(\alpha,\beta)]$ else //not found
6) $entryi : \exists\gamma \gg \beta.\ \{tag : \mathrm{Tag}(\gamma), mtable : \mathrm{Imty}(\gamma,\beta)\} = arr[i]$ in //get the ith entry
7) $(\gamma', entryi') = \mathrm{open}(entryi)$ in $//\gamma' \gg \beta, entryi' : \{tag : \mathrm{Tag}(\gamma'), mtable : \mathrm{Imty}(\gamma',\beta)\}$
8) ifEqTag$^{\mathrm{Imty}(\alpha,\beta)}(entryi'.tag, t_\alpha)$ then $entryi'.mtable$ else $//\gamma' = \alpha$
9) $loop[\alpha,\beta](t_\alpha, obj, arr, i+1, len)$ //next iteration

Figure 2: Itable Lookup

3.1 Kinds and Types

$$\begin{array}{llll} (kind) & \kappa & ::= & \Omega_\mathrm{c} \mid \Omega \\ (type) & \tau & ::= & \mathrm{int} \mid \alpha \mid \mathrm{array}(\tau) \mid (\tau_1,\ldots,\tau_n) \to \tau \\ & & & \mid \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\} \\ & & & \mid C \mid \mathrm{Tag}(\tau) \mid \forall\alpha \ll \tau.\ \tau' \mid \exists\alpha \ll \tau.\ \tau' \\ & & & \mid \{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}\!\} \\ & & & \mid \underline{I} \mid \underline{\mathrm{Imty}(\tau_1,\tau_2)} \mid \underline{\exists\alpha \gg \tau.\ \tau'} \\ (lbl\ ann) & \phi & ::= & \mathrm{I} \mid \mathrm{M} \end{array}$$

A special kind Ω_c classifies class and interface names and type variables that will be instantiated with class or interface names. The well-formedness rules of types use this kind to guarantee that certain type constructors (e.g. Tag) are applied to only class and interface names and type variables. Kind Ω classifies all types. Ω_c is a subkind of Ω, that is, a type that has kind Ω_c also has kind Ω.

Standard types include the integer type, type variable "α", array type "array(τ)", function type "$(\tau_1,\ldots,\tau_n) \to \tau$", and record type "$\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}$". In a record type, each field label is annotated with either "I" or "M", representing immutable and mutable respectively. "I" is often omitted. Vtables, itables, and virtual methods are all immutable fields in their enclosing record types. LIL$_C$ introduced class name "C", tag type "Tag(τ)", subclassing-bounded quantified types "$\forall\alpha \ll \tau.\ \tau'$" and "$\exists\alpha \ll \tau.\ \tau'$", and exact record type "$\{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}\!\}$" (which represents records that have and only have the specified fields). $\mathsf{R}(C)$ is an exact record type to rule out extra fields.

New types for interfaces include interface names "I", interface method table type "Imty(τ_1,τ_2)", and existential type "$\exists\alpha \gg \tau.\ \tau'$" which specifies a lower subclassing bound τ for its type variable α. The new existential type is used in itable types, as shown in Section 2.2.

3.2 Values and Expressions

Word-sized values include integer "n", heap label "ℓ", object of class C "$C(v)$" (coerced from record v), tag of class C "$tag(C)$", and packed word-sized value "pack τ as $\alpha \ll \tau_u$ in $(v : \tau')$". The packed value has type $\exists\alpha \ll \tau_u.\ \tau'$.

$$\begin{array}{llll} (val) & v & ::= & n \mid \ell \mid C(v) \mid tag(C) \\ & & & \mid \underline{tag(I)} \mid \underline{r2im[I,C](v)} \\ & & & \mid \underline{\text{pack } \tau \text{ as } \alpha \ll \tau_u \text{ in } (v : \tau')} \\ & & & \mid \underline{\text{pack } \tau \text{ as } \alpha \gg \tau_u \text{ in } (v : \tau')} \\ (expr) & e & ::= & \underline{x \mid n \mid \ell \mid tag(C) \mid \underline{tag(I)} \mid C(e)} \mid \mathrm{c2r}(e) \\ & & & \mid \underline{r2im[\tau_1,\tau_2](e)} \mid \underline{im2r(e)} \mid \mathrm{error}[\tau] \\ & & & \mid \underline{\mathrm{new}[\tau]\{l_i = e_i\}_{i=1}^n} \mid e.l \mid e_1.l_i := e_2 \text{ in } e_3 \\ & & & \mid \mathrm{new}[e_0,\ldots,e_{n-1}]^\tau \mid e_1[e_2] \\ & & & \mid e_1[e_2] := e_3 \text{ in } e_4 \\ & & & \mid x : \tau = e_1 \text{ in } e_2 \mid x := e_1 \text{ in } e_2 \\ & & & \mid e[\tau_1,\ldots,\tau_m](e_1,\ldots,e_n) \\ & & & \mid (\alpha, x) = \mathrm{open}(e_1) \text{ in } e_2 \\ & & & \mid \text{pack } \tau \text{ as } \alpha \ll \tau_u \text{ in } (e : \tau') \\ & & & \mid \underline{\text{pack } \tau \text{ as } \alpha \gg \tau_u \text{ in } (e : \tau')} \\ & & & \mid \mathrm{ifParent}^\tau(e) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2 \\ & & & \mid \mathrm{ifEqTag}^\tau(e_{t1}, e_{t2}) \text{ then } e_1 \text{ else } e_2 \end{array}$$

New word-sized values include tag of interface I "$tag(I)$", method table for interface I in class C "$r2im[I,C](v)$" (coerced from a record v containing C's implementations of methods in I), and packed values with lower subclassing bounds "pack τ as $\alpha \gg \tau_u$ in $(v : \tau')$". The value $r2im[I,C](v)$ has type $\mathrm{Imty}(I,C)$.

Besides values, ECI expressions include variable "x", coercing a record e (with type $\mathsf{R}(C)$) to a C object "$C(e)$", coercing an object e to a record "c2r(e)", runtime error expression "error[τ]" (an expression of type τ is expected in normal execution), record creation "new[τ]$\{l_i = e_i\}_{i=1}^n$", field access "$e.l$", field assignment "$e_1.l_i := e_2$ in e_3", array creation "new$[e_0,\ldots,e_{n-1}]^\tau$", array subscript "$e_1[e_2]$", array element assignment "$e_1[e_2] := e_3$ in e_4", let binding "$x : \tau = e_1$ in e_2" (τ specifies x's type), variable assignment "$x := e_1$ in e_2", function call "$e[\tau_1,\ldots,\tau_m](e_1,\ldots,e_n)$", open expression "$(\alpha, x) = \mathrm{open}(e_1)$ in e_2", pack expression "pack τ as $\alpha \ll \tau_u$ in $(e : \tau')$", fetching-parent-tag expression "ifParent$^\tau(e)$ then bind (α, x) in e_1 else e_2", and tag comparison expression "ifEqTag$^\tau(e_{t1}, e_{t2})$ then e_1 else e_2". The "ifParent" expression is used in class downward casts [6]. The "ifEqTag" expression compares two tags e_{t1} and e_{t2}. If they are equal, the types identified by the tags are the same.

The true branch e_1 is type-checked assuming that the two types are the same.

ECI introduces two new expressions to describe coercions between interface method tables and records. Expression "$r2im[\tau_1, \tau_2](e)$" coerces a record e containing methods for interface τ_1 in class τ_2 to a method table (of type $\text{Imty}(\tau_1, \tau_2)$). Expression "$im2r(e)$" coerces a method table e to a record.

To introduce values of existential types with lower subclassing bounds, a second pack expression "pack τ as $\alpha \gg \tau_u$ in $(e : \tau')$" coerces an expression e to type $\exists\alpha \gg \tau_u.\ \tau'$, by hiding type τ (where $\tau \gg \tau_u$) with type variable α. The packed values can be unpacked by the existing "open" expression.

4. ECI SEMANTICS

This section focuses on the semantics related to interfaces. The rest of the rules can be found in the technical report.

ECI maintains a declaration table Θ that maps class and interface names to their declarations. A kind environment Δ tracks type variables in scope and their bounds. Each entry in Δ introduces a new type variable and an upper or lower bound of the type variable. The bound is a class name, an interface name, or another type variable introduced previously in Δ. A heap environment Σ maps heap labels to types. A type environment Γ maps variables to types. Substitution τ/α means replacing α with τ.

4.1 Types

The kinding judgment $\Theta; \Delta \vdash \tau : \kappa$ means that, under environments Θ and Δ, type τ has kind κ.

Non-standard kinding rules are those related to Ω_c, the kind that classifies class and interface names. All type variables have kind Ω_c. The tag constructor can be applied only to types with kind Ω_c. Bounds of type variables in quantified types must have kind Ω_c.

Interface names have kind Ω_c. Type $\text{Imty}(\tau_1, \tau_2)$ requires that both τ_1 and τ_2 have kind Ω_c. The new kinding rules are as follows:

$$\frac{I \in \text{domain}(\Theta)}{\Theta; \bullet \vdash I : \Omega_c} \quad \frac{\Theta; \Delta \vdash \tau_1 : \Omega_c \quad \Theta; \Delta \vdash \tau_2 : \Omega_c}{\Theta; \Delta \vdash \text{Imty}(\tau_1, \tau_2) : \Omega}$$

$$\frac{\Theta; \Delta \vdash \tau : \Omega_c \quad \Theta; \Delta, \alpha \gg \tau \vdash \tau' : \Omega}{\Theta; \Delta \vdash \exists\alpha \gg \tau.\ \tau' : \Omega}$$

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 tracks the source-level inheritance hierarchy. Subclassing is reflexive and transitive. Subclassing between interfaces or between classes and interfaces are as follows:

$$\frac{J \text{ is a superinterface of } I}{\Theta; \Delta \vdash I \ll J} \quad \frac{C \text{ implements } I}{\Theta; \Delta \vdash C \ll I}$$

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. As expected, there are standard structural record prefix and depth subtyping, function subtyping, reflexivity, and transitivity. Exact record types are subtypes of normal record types with same fields. Structural subtyping between quantified types is similar to Castagna and Pierce's $\forall -$ top rule [3], where the checker ignores the bounds of type variables (relax it to Topc, the superclass of any other class) when checking subtyping between the body types. Contrary to ECI, Castagna and Pierce's bounded quantification was based on subtyping, and did not have the minimal type property [2].

The subtyping rule for existential types with lower subclassing bounds is shown as follows:

$$\frac{\Theta; \Delta \vdash u_2 \ll u_1 \quad \Theta; \Delta, \alpha \ll \text{Topc} \vdash \tau_1 \leq \tau_2}{\Theta; \Delta \vdash (\exists\alpha \gg u_1.\ \tau_1) \leq (\exists\alpha \gg u_2.\ \tau_2)}$$

4.2 Expressions

The typing judgment $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ means that, under environments Θ, Δ, Σ and Γ, expression e has type τ. Rules related to interfaces are as follows.

$$\frac{\Theta; \Delta; \Sigma; \Gamma \vdash e : \mathsf{R_I}(I, C)}{\Theta; \Delta; \Sigma; \Gamma \vdash r2im[I, C](e) : \text{Imty}(I, C)}$$

$$\frac{\Theta; \Delta; \Sigma; \Gamma \vdash e : \text{Imty}(I, \tau)}{\Theta; \Delta; \Sigma; \Gamma \vdash im2r(e) : \mathsf{R_I}(I, \tau)}$$

$$\frac{I \in \text{domain}(\Theta)}{\Theta; \Delta; \Sigma; \Gamma \vdash tag(I) : \text{Tag}(I)}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \exists\beta \gg \tau_u.\ \tau \\ \alpha \notin \text{domain}(\Delta) \qquad \alpha \notin free(\tau') \\ \Theta; \Delta, \alpha \gg \tau_u; \Sigma; \Gamma, x : \tau[\alpha/\beta] \vdash e_2 : \tau'\end{array}}{\Theta; \Delta; \Sigma; \Gamma \vdash (\alpha, x) = \text{open}(e_1) \text{ in } e_2 : \tau'}$$

$$\frac{\Theta; \Delta \vdash \tau_u \ll \tau \quad \alpha \notin \text{domain}(\Delta) \quad \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau'[\tau/\alpha]}{\Theta; \Delta; \Sigma; \Gamma \vdash \text{pack } \tau \text{ as } \alpha \gg \tau_u \text{ in } (e : \tau') : \exists\alpha \gg \tau_u.\ \tau'}$$

$\mathsf{R_I}$ is a macro used by the type checker to represent record types that correspond to method tables, similar to R and $\mathsf{ApproxR}$. Suppose interface I declares methods $m_1, \ldots, m_k$ of type $\tau_1, \ldots, \tau_k$ respectively. $\mathsf{R_I}(I, \tau)$ is defined as $\mathsf{R_I}(I, \tau) = \{m_1 : \tau_1, \ldots, m_k : \tau_k\}$ where τ_i refers to τ in the "this" pointer types.

Expression $r2im[I, C](e)$ coerces a record e of type $\mathsf{R_I}(I, C)$ to method table type $\text{Imty}(I, C)$. Expression $im2r(e)$ coerces a method table e with type $\text{Imty}(I, \tau)$ to record type $\mathsf{R_I}(I, \tau)$ where τ can be a type variable or a class name.

ECI has additional typing rules for the open and pack expressions that eliminate and introduce existential types with lower subclassing bounds respectively.

4.3 Dynamic Semantics

The evaluation rules for the new expressions are as follows. The notation $e \rightsquigarrow e'$ means e steps to e'. Expressions "$r2im[\tau_1, \tau_2](v)$" and "$im2r(v)$" coerce between records and interface method tables. The open expression can eliminate new packed values of existential types with lower subclassing bounds.

$im2r(r2im[I, C](v)) \rightsquigarrow v$
$(\alpha, x) = \text{open}(\text{pack } \tau \text{ as } \beta \gg \tau_u \text{ in } (v' : \tau')) \text{ in } e_0 \rightsquigarrow$ $e_0[\tau/\alpha] \text{ with } x \mapsto v'$

4.4 Properties of ECI

We have proved that ECI has the following properties:

THEOREM 1 (SOUNDNESS). *ECI is sound.*

Proof sketch: by proving the standard progress and preservation lemmas.

THEOREM 2 (DECIDABILITY). *Type checking ECI is decidable.*

Proof sketch: by proving decidability of the algorithmic typing rules and decidability of subtyping.

THEOREM 3 (TYPE-PRESERVING TRANSLATION). *Well-typed source language programs are translated to well-typed ECI programs.*

5. RELATED WORK

League *et al.* proposed a typed intermediate language JFlint for compiling Java-like languages [15]. JFlint uses existential quantification over row variables for object encodings. There are two approaches to support interfaces. The first approach uses unordered records for itable entries, which does not model itable search. The second approach pairs a specialized itable with an object when casting the object to an interface. The specialized itable contains only information of the target interface. The second approach requires non-standard implementation: upcasting is no longer free.

Chen *et al.* proposed a model based on guarded recursive datatypes that supports multiple inheritance between classes [4]. The model preserves class names. Each method invocation is associated with a sequence of class names, for identifying methods. This approach encodes objects with functions, which differs from the standard implementation of objects as records. As a result, object layouts, including vtables and itables, cannot be addressed.

SpecialJ is a certifying compiler for Java [7]. It compiles Java to assembly code with type annotations. The target language preserves class and interface names, but the paper [7] didn't discuss how the target type system addresses itable or interface implementation.

6. CONCLUSION

This paper describes a typed intermediate language that supports interfaces, a restricted form of multiple inheritance. The language models itables with array types and subclassing-bounded quantification. It faithfully models the standard itable-based implementation of interface method invocation and interface cast.

7. REFERENCES

[1] B. Alpern, A. Cocchi, S. J. Fink, D. Grove, and D. Lieber. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *OOPSLA '01: ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 108–124, 2001.

[2] G. Castagna and B. C. Pierce. *Corrigendum: Decidable Bounded Quantification.* http://www.cis.upenn.edu/∼bcpierce/papers/fsubnew-corrigendum.ps.

[3] G. Castagna and B. C. Pierce. Decidable bounded quantification. In *POPL '94: ACM Symposium on Principles of Programming Languages*, pages 151–162, 1994.

[4] C. Chen, R. Shi, and H. Xi. A typeful approach to object-oriented programming with multiple inheritance. In *PADL '04: International Symposium on Practical Aspects of Declarative Languages*, pages 23–38, June 2004.

[5] J. Chen. A typed intermediate language for supporting interfaces. Technical report, Microsoft Corporation, March 2009. http://research.microsoft.com/apps/pubs/?id=80117.

[6] J. Chen and D. Tarditi. A simple typed intermediate language for object-oriented languages. In *POPL '05: ACM SIGPLAN Conference on Principles of Programming Languages*, pages 38–49, January 2005.

[7] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *PLDI '00: ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[8] B. J. Cox and A. Novobilski. *Object-Oriented Programming; An Evolutionary Approach.* Addison-Wesley Longman Publishing Co., Inc., 1991.

[9] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA '89: Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–214, 1989.

[10] K. Driesen. Selector table indexing sparse arrays. In *OOPSLA '93: Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 259–270, 1993.

[11] Microsoft Corp. et al. *Common Language Infrastructure.* 2002. http://msdn.microsoft.com/net/ecma/.

[12] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91: European Conference on Object-Oriented Programming*, pages 21–38, 1991.

[13] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. on Programming Languages and Systems*, 23(3):396–450, 2001.

[14] A. Krall and R. Grafl. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[15] C. League, Z. Shao, and V. Trifonov. Type-preserving compilation of Featherweight Java. *ACM Trans. on Programming Languages and Systems*, 24(2), March 2002.

[16] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1999.

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

[18] G. Necula. Proof-Carrying Code. In *POPL '97: ACM Symposium on Principles of Programming Languages*, pages 106–119, 1997.