A Typed Intermediate Language for Supporting Interfaces

Juan Chen
juanchen@microsoft.com

May 2009

Technical Report
MSR-TR-2009-35

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.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

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 [10]) 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 [5]. 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 Appendix A. Appendix B 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. Appendix C discusses how ECI may be extended to express advanced interface implementation techniques [14, 12, 7, 8, 9, 1].

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 [5] 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 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

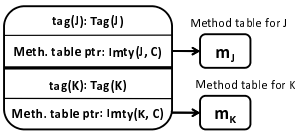

Figure 1: Itable of a Class C Implementing J and K

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 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:

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

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:

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

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.

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

R and $\mathsf{ApproxR}$ 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.

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

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[Imty(α,β)] 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

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 Tag(γ')) with the target interface's tag (t_α with type Tag(α)). If the tags are the same, then $\gamma' = \alpha$. The method table in the entry (with type Imty(γ',β)) is returned as a value of type Imty(α,β). 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 $\text{Imty}(J, \beta)$. 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*.

$$fix\ ICast\langle\alpha,\beta\rangle \left(\begin{array}{ll} t_\alpha : \text{Tag}(\alpha), & \text{//tag of target interface } \alpha \\ obj : \beta & \text{//object to cast} \end{array}\right) : \exists\delta \ll \alpha.\ \delta$$

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

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

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

The *ICast* function returns an object of the target interface α, represented by the return type $\exists\delta \ll \alpha.\ \delta$. It calls *castLoop* to search the itable for the entry for α. If found, step 8) coerces the object (with type β) to the return type $\exists\delta \ll \alpha.\ \delta$ because β has a superinterface γ' that equals α (thus $\beta \ll \alpha$).

3 Syntax of ECI

This section explains the syntax of ECI. We underline new interface-related constructs to distinguish them from those introduced in the base language LIL$_C$.

3.1 Kinds and Types

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

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

Figure 3: Kinds and Types

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 "$\mathrm{Imty}(\tau_1, \tau_2)$", 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.

Type variable sequence tvs is a sequence of type variables, each with a superclass bound. The sequence is used in declarations of polymorphic functions.

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 $\exists \alpha \ll \tau_u.\ \tau'$.

$$
\begin{array}{llll}
(val) & v & ::= & n \mid \ell \mid C(v) \mid tag(C) \\
 & & & \mid \underline{tag(I)} \mid \underline{r2im[I, C](v)} \\
 & & & \mid \underline{\mathrm{pack}\ \tau\ \mathrm{as}\ \alpha \ll \tau_u\ \mathrm{in}\ (v : \tau')} \\
 & & & \mid \underline{\mathrm{pack}\ \tau\ \mathrm{as}\ \alpha \gg \tau_u\ \mathrm{in}\ (v : \tau')} \\
(hvalue) & hv & ::= & \overline{\{l_i = v_i\}_{i=1}^n} \mid \overline{[v_0, \ldots, v_{n-1}]^\tau} \\
 & & & \mid \mathrm{fix}\ g\langle tvs \rangle (x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e_m \\
(expr) & e & ::= & x \mid n \mid \ell \mid tag(C) \mid \underline{tag(I)} \mid C(e) \mid \mathrm{c2r}(e) \\
 & & & \mid \underline{r2im[\tau_1, \tau_2](e)} \mid \underline{im2r(e)} \mid \mathrm{error}[\tau] \\
 & & & \mid \overline{\mathrm{new}[\tau]\{l_i = e_i\}_{i=1}^n} \mid e.l \mid e_1.l_i := e_2\ \mathrm{in}\ e_3 \\
 & & & \mid \overline{\mathrm{new}[e_0, \ldots, e_{n-1}]^\tau} \mid e_1[e_2] \\
 & & & \mid e_1[e_2] := e_3\ \mathrm{in}\ e_4 \\
 & & & \mid x : \tau = e_1\ \mathrm{in}\ e_2 \mid x := e_1\ \mathrm{in}\ e_2 \\
 & & & \mid e[\tau_1, \ldots, \tau_m](e_1, \ldots, e_n) \\
 & & & \mid (\alpha, x) = \mathrm{open}(e_1)\ \mathrm{in}\ e_2 \\
 & & & \mid \mathrm{pack}\ \tau\ \mathrm{as}\ \alpha \ll \tau_u\ \mathrm{in}\ (e : \tau') \\
 & & & \mid \underline{\mathrm{pack}\ \tau\ \mathrm{as}\ \alpha \gg \tau_u\ \mathrm{in}\ (e : \tau')} \\
 & & & \mid \overline{\mathrm{ifParent}^\tau(e)\ \mathrm{then\ bind}\ (\alpha, x)}\ \mathrm{in}\ e_1\ \mathrm{else}\ e_2 \\
 & & & \mid \mathrm{ifEqTag}^\tau(e_{t1}, e_{t2})\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2
\end{array}
$$

Figure 4: Values and Expressions

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

$\text{Imty}(I, C)$.

Heap values include record "$\{l_i = v_i\}_{i=1}^{n}$", array "$[v_0, \ldots, v_{n-1}]^{\tau}$", and function "$\text{fix}\, g\langle tvs\rangle(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e_m$". Arrays are annotated with their element types. Functions may be polymorphic. The return type is also specified in each function.

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 "$(\alpha, x) = \text{open}(e_1)$ 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 [5]. If tag e has a parent tag, then the "ifParent" expression binds the superclass the parent tag indicates to a fresh type variable α and the parent tag to x and evaluates e_1. Otherwise the control transfers to e_2. The desired result type τ is annotated to simplify type checking. 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.

3.2.1 Declarations and Programs

$$
\begin{array}{llll}
\mathit{field} & ::= & f : \tau & \\
\mathit{method} & ::= & m : \forall\ tvs\ (\tau_1, \ldots, \tau_n) \to \tau & \\
\mathit{class} & ::= & C : B, \overrightarrow{I}\{\overrightarrow{\mathit{field}}, \overrightarrow{\mathit{method}}\} & \\
\underline{\mathit{interface}} & ::= & \underline{I : \overrightarrow{J}\{\overrightarrow{\mathit{method}}\}} & \\
H & ::= & \bullet \mid H, \ell \rightsquigarrow hv & \\
V & ::= & \bullet \mid V, x = v & \\
\mathit{decl} & ::= & \mathit{interface} \mid \mathit{class} & \\
\mathit{prog} & ::= & (\overrightarrow{\mathit{decl}}; H; V; e) &
\end{array}
$$

Figure 5: Classes and Programs

Field declaration "$f : \tau$" introduces a field named f with type τ. Method declaration "$m : \forall\, tvs\ (\tau_1, \ldots, \tau_n) \to \tau$" specifies a method named m with type $\forall\ tvs\ (\tau_1, \ldots, \tau_n) \to \tau$. Only the method signature is declared here. Method implementations are represented as functions on the heap. Heap "H" maps labels to heap values. Variable-value mapping "V" maps variables to word-sized values.

A class declaration $C : B, \overrightarrow{I}\{\overrightarrow{\mathit{field}}, \overrightarrow{\mathit{method}}\}$ declares a class name C with superclass B, all its super interfaces $\overrightarrow{I}$, and all its fields and methods including those from the superclasses. The notation $\overrightarrow{\rho}$ means a sequence of items in ρ.

An interface declaration "$I : \overrightarrow{J}\{\overrightarrow{\mathit{method}}\}$" specifies an interface name I, all its super interfaces $\overrightarrow{J}$, and declarations of all the methods in the interface and its superinterfaces $\overrightarrow{\mathit{method}}$.

A program has a set of class declarations, a heap H, a variable-value mapping V, and a main expression.

4 ECI Semantics

This section explains the semantics of ECI.

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,

$$\frac{}{\Theta;\bullet \vdash \text{int} : \Omega} \qquad \frac{C \in \text{domain}(\Theta)}{\Theta;\bullet \vdash C : \Omega_c} \qquad \frac{I \in \text{domain}(\Theta)}{\Theta;\bullet \vdash I : \Omega_c} \qquad \frac{}{\Theta;\bullet \vdash \text{Topc} : \Omega_c}$$

$$\frac{\Theta;\Delta \vdash \tau : \Omega_c}{\Theta;\Delta \vdash \text{Tag}(\tau) : \Omega} \qquad \frac{\Theta;\Delta \vdash \tau : \Omega}{\Theta;\Delta \vdash \text{array}(\tau) : \Omega} \qquad \frac{\Theta;\Delta \vdash \tau_1 : \Omega_c \quad \Theta;\Delta \vdash \tau_2 : \Omega_c}{\Theta;\Delta \vdash \text{Imty}(\tau_1, \tau_2) : \Omega}$$

$$\frac{\alpha \ll \tau \in \Delta \text{ or } \alpha \gg \tau \in \Delta}{\Theta;\Delta \vdash \alpha : \Omega_c} \qquad \frac{\Theta;\Delta, \alpha \ll \tau \vdash \tau' : \Omega}{\Theta;\Delta \vdash \forall \alpha \ll \tau.\ \tau' : \Omega} \qquad \frac{\Theta;\Delta, \alpha \ll \tau \vdash \tau' : \Omega}{\Theta;\Delta \vdash \exists \alpha \ll \tau.\ \tau' : \Omega}$$

$$\frac{\Theta;\Delta, \alpha \gg \tau \vdash \tau' : \Omega}{\Theta;\Delta \vdash \exists \alpha \gg \tau.\ \tau' : \Omega} \qquad \frac{\Theta;\Delta \vdash \tau_i : \Omega\ \forall 1 \le i \le n \quad \Theta;\Delta \vdash \tau : \Omega}{\Theta;\Delta \vdash (\tau_1, \ldots, \tau_n) \to \tau : \Omega} \qquad \frac{\Theta;\Delta \vdash \tau : \Omega_c}{\Theta;\Delta \vdash \tau : \Omega}$$

$$\frac{\Theta;\Delta \vdash \tau_i : \Omega\ \forall 1 \le i \le n}{\Theta;\Delta \vdash \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\} : \Omega} \qquad \frac{\Theta;\Delta \vdash \tau_i : \Omega\ \forall 1 \le i \le n}{\Theta;\Delta \vdash \{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}\!\} : \Omega}$$

Figure 6: Kinding Rules

$$\frac{\Theta(I) = I : I_1, \ldots, I_n\{\_\} \quad 1 \le i \le n}{\Theta;\Delta \vdash I \ll I_i} \qquad \frac{\begin{array}{c}\Theta(C) = C : B, I_1, \ldots, I_n\{\ldots\} \\ \tau = B \text{ or } I_i (1 \le i \le n)\end{array}}{\Theta;\Delta \vdash C \ll \tau} \qquad \frac{\Theta;\Delta \vdash \tau : \Omega_c}{\Theta;\Delta \vdash \tau \ll \text{Topc}}$$

$$\frac{\alpha \ll \tau \in \Delta}{\Theta;\Delta \vdash \alpha \ll \tau} \qquad \frac{\alpha \gg \tau \in \Delta}{\Theta;\Delta \vdash \tau \ll \alpha} \qquad \frac{\Theta;\Delta \vdash \tau : \Omega_c}{\Theta;\Delta \vdash \tau \ll \tau} \qquad \frac{\Theta;\Delta \vdash \tau_1 \ll \tau_2 \quad \Theta;\Delta \vdash \tau_2 \ll \tau_3}{\Theta;\Delta \vdash \tau_1 \ll \tau_3}$$

Figure 7: Subclassing Rules

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. Figure 6 shows the kinding rules. Topc is a special class name similar to the Object class. Any other class name is a subclass of Topc.

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 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. Figure 7 shows the subclassing rules.

The subtyping judgment $\Theta;\Delta \vdash \tau_1 \le \tau_2$ means that, under environments Θ and Δ, τ_1 is a subtype of τ_2. Figure 8 shows the subtyping rules. 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].

$$\frac{m \geq n}{\Theta;\Delta \vdash \{l_i^{\phi_i} : \tau_i\}_{i=1}^{m} \leq \{l_i^{\phi_i} : \tau_i\}_{i=1}^{n}}\ \textbf{st\_breadth} \qquad \frac{\forall 1 \leq i \leq m, \begin{cases} \Theta;\Delta \vdash \tau_i \leq \tau_i' & if\ \phi_i = \mathrm{I} \\ \tau_i = \tau_i' & if\ \phi_i = \mathrm{M} \end{cases}}{\Theta;\Delta \vdash \{l_i^{\phi_i} : \tau_i\}_{i=1}^{m} \leq \{l_i^{\phi_i} : \tau_i'\}_{i=1}^{m}}\ \textbf{st\_depth}$$

$$\frac{}{\Theta;\Delta \vdash \{\!\{l_i^{\phi_i} : \tau_i\}\!\}_{i=1}^{m} \leq \{l_i^{\phi_i} : \tau_i\}_{i=1}^{m}}\ \textbf{st\_exact} \qquad \frac{\Theta;\Delta \vdash t_i \leq s_i\ \forall 1 \leq i \leq n \quad \Theta;\Delta \vdash s \leq t}{\Theta;\Delta \vdash (s_1,\ldots,s_n) \to s \leq (t_1,\ldots,t_n) \to t}\ \textbf{st\_fun}$$

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

$$\frac{\Theta;\Delta \vdash u_2 \ll u_1 \quad \Theta;\Delta,\alpha \ll \mathrm{Topc} \vdash \tau_1 \leq \tau_2}{\Theta;\Delta \vdash (\forall \alpha \ll u_1.\ \tau_1) \leq (\forall \alpha \ll u_2.\ \tau_2)}\ \textbf{st\_}\forall \qquad \frac{\Theta;\Delta \vdash \tau_1 \leq \tau_2 \quad \Theta;\Delta \vdash \tau_2 \leq \tau_3}{\Theta;\Delta \vdash \tau_1 \leq \tau_3}\ \textbf{st\_trans} \qquad \frac{}{\Theta;\Delta \vdash \tau \leq \tau}\ \textbf{st\_ref}$$

Figure 8: Subtyping Rules

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. Figure 9 lists the expression typing rules.

Only records of type $\mathsf{R}(C)$ can be coerced to objects of C (rule **object**). Coercing an object of C to a record results in a record of type $\mathsf{R}(C)$ (rule **c2r_c**). Coercing an object of α to a record results in a record of type $\mathsf{ApproxR}(\alpha, C)$ if $\alpha \ll C$ (rule **c2r_tv**).

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

Rule **call** requires that in a call expression "$e[\tau_1,\ldots,\tau_m](e_1,\ldots,e_n)$", the (polymorphic) function e have a function type $\forall tvs(s_1,\ldots,s_n) \to s$. The type arguments $\tau_1,\ldots,\tau_m$ must satisfy the constraints tvs specifies. The value arguments $e_1,\ldots,e_n$ have types $s_1,\ldots s_n$ respectively, after the type variables in tvs are replaced with $\tau_1,\ldots,\tau_m$.

Rule **open** checks elimination of existential types with upper bounds. It is standard except for the subclassing bounds. Rule **pack** introduces existential types with upper bounds. It checks that the hidden type τ is a subclass of the specified bound τ_u.

ECI has additional typing rules for the open and pack expressions that eliminate and introduce existential types with lower subclassing bounds respectively (rules **open_**$\gg$ and **pack_**$\gg$).

Rule **ifParent** says that in "ifParent$^{\tau}(e)$ then bind (α, x) in e_1 else e_2", tag e has type $\mathrm{Tag}(\tau')$ for some type τ'. If e has a parent tag, a new type variable α is introduced for the superclass of τ' and a new value variable x (of type $\mathrm{Tag}(\alpha)$) is introduced for the parent tag. Both branches e_1 and e_2 need to have the annotated type τ.

The last three rules check tag comparison expressions. In "ifEqTag$^{\tau}(e_{t1}, e_{t2})$ then e_1 else e_2", if both e_{t1} and e_{t2} are tags for concrete class names, the type checker only checks the branch to be taken (rule **ifTag_eq, ifTag_neq**). Otherwise (rule **ifTag_tv**), the first tag e_{t1} must have type $\mathrm{Tag}(\gamma)$ for some type variable γ. The second tag e_{t2} can be a tag for type τ_γ, either a class name or a type variable introduced before γ. The checker uses $\gamma = \tau_\gamma$ to refine the type of the true branch e_1, because $\gamma = \tau_\gamma$ must hold if the true branch is taken. The false branch has no type refinement.

Figure 10 lists the typing rules for heap values. The judgment $\Theta;\Sigma \vdash hv : \tau$ means that under the environments Θ and Σ, heap value hv has type τ.

$$\frac{}{\Theta;\Delta;\Sigma;\Gamma \vdash x : \Gamma(x)}\ \textbf{var} \qquad \frac{}{\Theta;\Delta;\Sigma;\Gamma \vdash n : \text{int}}\ \textbf{int} \qquad \frac{\Theta;\Delta \vdash \tau : \Omega}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{error}[\tau] : \tau}\ \textbf{error}$$

$$\frac{}{\Theta;\Delta;\Sigma;\Gamma \vdash \ell : \Sigma(\ell)}\ \textbf{label} \qquad \frac{\tau = C \text{ or } I \quad \tau \in \text{domain}(\Theta)}{\Theta;\Delta;\Sigma;\Gamma \vdash tag(\tau) : \text{Tag}(\tau)}\ \textbf{tag} \qquad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \mathsf{R}(C)}{\Theta;\Delta;\Sigma;\Gamma \vdash C(e) : C}\ \textbf{object}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : C}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{c2r}(e) : \mathsf{R}(C)}\ \textbf{c2r\_c} \qquad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \alpha \quad C \text{ is a concrete class name} \quad \Theta;\Delta \vdash \alpha \ll C}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{c2r}(e) : \mathsf{ApproxR}(\alpha, C)}\ \textbf{c2r\_tv}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \mathsf{R}_{\mathsf{I}}(I, C)}{\Theta;\Delta;\Sigma;\Gamma \vdash r2im[I,C](e) : \text{Imty}(I, C)}\ \textbf{r2im} \qquad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \text{Imty}(I, \tau)}{\Theta;\Delta;\Sigma;\Gamma \vdash im2r(e) : \mathsf{R}_{\mathsf{I}}(I, \tau)}\ \textbf{im2r}$$

$$\frac{\tau = \{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}\!\} \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_i : \tau_i\ \forall 1 \le i \le n}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{new}[\tau]\{l_i = e_i\}_{i=1}^n : \tau}\ \textbf{record} \qquad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \{l_1^{\phi_1} : \tau_1, \ldots, l_i^{\text{M}} : \tau_i, \ldots, l_n^{\phi_n} : \tau_n\} \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau_i \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_3 : \tau}{\Theta;\Delta;\Sigma;\Gamma \vdash e_1.l_i := e_2 \text{ in } e_3 : \tau}\ \textbf{assignR}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \{l_i^{\phi_i} : \tau_i\}_{i=1}^n \quad 1 \le i \le n}{\Theta;\Delta;\Sigma;\Gamma \vdash e.l_i : \tau_i}\ \textbf{field} \qquad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_i : \tau\ \forall 0 \le i \le n-1}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{new}[e_0, \ldots, e_{n-1}]^\tau : \text{array}(\tau)}\ \textbf{array}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \text{array}(\tau) \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \text{int}}{\Theta;\Delta;\Sigma;\Gamma \vdash e_1[e_2] : \tau}\ \textbf{subscript} \qquad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \text{array}(\tau) \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \text{int} \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_3 : \tau \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_4 : \tau'}{\Theta;\Delta;\Sigma;\Gamma \vdash e_1[e_2] := e_3 \text{ in } e_4 : \tau'}\ \textbf{assignA}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau \quad \Theta;\Delta;\Sigma;\Gamma, x :_{\text{M}} \tau \vdash e_2 : \tau'}{\Theta;\Delta;\Sigma;\Gamma \vdash x : \tau = e_1 \text{ in } e_2 : \tau'}\ \textbf{let} \qquad \frac{x :_{\text{M}} \tau \in \Gamma \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau'}{\Theta;\Delta;\Sigma;\Gamma \vdash x := e_1 \text{ in } e_2 : \tau'}\ \textbf{assign}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \forall tvs(\tau_1, \ldots, \tau_n) \to \tau \quad tvs = \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m \quad \sigma = t_1, \ldots, t_m/tvs \quad \Theta;\Delta \vdash t_i \ll u_i[\sigma]\ \ \forall 1 \le i \le m \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_i : \tau_i[\sigma]\ \forall 1 \le i \le n}{\Theta;\Delta;\Sigma;\Gamma \vdash e[t_1, \ldots, t_m](e_1, \ldots, e_n) : \tau[\sigma]}\ \textbf{call}$$

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

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \tau_1 \quad \Theta;\Delta \vdash \tau_1 \le \tau_2}{\Theta;\Delta;\Sigma;\Gamma \vdash e : \tau_2}\ \textbf{sub} \qquad \frac{\Theta;\Delta \vdash \tau \ll \tau_u \quad \alpha \notin \text{domain}(\Delta) \quad \Theta;\Delta;\Sigma;\Gamma \vdash e : \tau'[\tau/\alpha]}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{pack } \tau \text{ as } \alpha \ll \tau_u \text{ in } (e : \tau') : \exists \alpha \ll \tau_u.\ \tau'}\ \textbf{pack}$$

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

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \text{Tag}(\tau') \quad \alpha \notin \text{domain}(\Delta) \quad \Theta;\Delta, \alpha \gg \tau';\Sigma;\Gamma, x : \text{Tag}(\alpha) \vdash e_1 : \tau \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{ifParent}^\tau(e) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2 : \tau}\ \textbf{ifParent}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_{t1} : \text{Tag}(\tau_1) \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_{t2} : \text{Tag}(\tau_2) \quad \tau_1 = \tau_2 \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{ifEqTag}^\tau(e_{t1}, e_{t2}) \text{ then } e_1 \text{ else } e_2 : \tau}\ \textbf{ifTag\_eq}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_{t1} : \text{Tag}(\tau_1) \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_{t2} : \text{Tag}(\tau_2) \quad \tau_1 \text{ and } \tau_2 \text{ are class or interface names} \quad \tau_1 \neq \tau_2 \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{ifEqTag}^\tau(e_{t1}, e_{t2}) \text{ then } e_1 \text{ else } e_2 : \tau}\ \textbf{ifTag\_neq}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_{t1} : \text{Tag}(\gamma) \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_{t2} : \text{Tag}(\tau_\gamma) \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau[\gamma/\tau_\gamma] \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau \quad \Delta = \Delta_1, P(\gamma), \Delta_2 \quad P(\gamma) = \gamma \ll u \text{ or } P(\gamma) = \gamma \gg u \quad \Theta;\Delta_1 \vdash \tau_\gamma : \Omega_{\text{c}}}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{ifEqTag}^\tau(e_{t1}, e_{t2}) \text{ then } e_1 \text{ else } e_2 : \tau}\ \textbf{ifTag\_tv}$$

Figure 9: Expression Typing Rules

$$\frac{\Theta;\bullet;\Sigma;\bullet \vdash v_i : \tau \ \forall 0 \le i \le n-1}{\Theta;\Sigma \vdash [v_0,\ldots,v_{n-1}]^\tau : \text{array}(\tau)}\ \textbf{hv\_array} \qquad \frac{\Theta;\bullet;\Sigma;\bullet \vdash v_i : \tau_i \ \forall 1 \le i \le n}{\Theta;\Sigma \vdash \{l_i = v_i\}_{i=1}^n : \{\!\{l_i^{\phi_i} : \tau_i\}\!\}_{i=1}^n}\ \textbf{hv\_record}$$

$$\frac{\begin{array}{c}\tau_f = \forall\ tvs\ (\tau_1,\ldots,\tau_n) \to \tau \\ \Theta; tvs; \Sigma; g : \tau_f, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_m : \tau\end{array}}{\Theta;\Sigma \vdash \text{fix } g\langle tvs\rangle(x_1 : \tau_1,\ldots,x_n : \tau_n) : \tau = e_m : \tau_f}\ \textbf{hv\_fun}$$

Figure 10: Heap Value Typing Rules

4.3 Interface Declaration

A well-formed interface contains all the methods of its super interfaces, with same types. The superinterfaces must be declared before the subinterface.

$$\frac{\begin{array}{c}\forall 1 \le i \le n, I_i \text{ declared before } I \quad \Theta \vdash I_i : \_\{\_\} \\ \forall \text{ methods } m \text{ in } I_i, m \in m_1,\ldots,m_j\end{array}}{\Theta \vdash I : I_1,\ldots,I_n\{m_1 : t_1,\ldots,m_j : t_j\}}$$

4.4 Class Declarations and Programs

$$\frac{}{\Theta \vdash \text{Topc}\{\}}$$

$$\frac{\begin{array}{c}\Theta \vdash B : \_\left\{\ f_1 : s_1,\ldots,f_p : s_p, m_1 : t_1,\ldots,m_q : t_q\ \right\} \qquad B \text{ declared before } C \\ p \le n \quad q \le j \quad \Theta;\bullet \vdash s_i : \Omega\ \forall p+1 \le i \le n \qquad \Theta;\bullet \vdash t_i : \Omega\ \forall q+1 \le i \le j \\ \forall 1 \le i \le k, \Theta \vdash I_i : \_\{\_\},\ \forall \text{ method } m \text{ in } I_i, m \in m_1,\ldots,m_j\end{array}}{\Theta \vdash C : B, I_1,\ldots,I_k\left\{\ f_1 : s_1,\ldots,f_n : s_n, m_1 : t_1,\ldots,m_j : t_j\ \right\}}$$

Figure 11: Well-formedness of Class Declarations

Figure 11 shows the well-formedness rules for class declarations. A well-formed class contains all the members inherited from its parent class and superinterfaces, which is solely an implementation strategy to simplify looking up members. The parent class and the superinterfaces must be declared before the child class.

$$\frac{\begin{array}{c}\text{domain}(V) = \text{domain}(\Gamma) \\ \Theta;\bullet;\Sigma;\bullet \vdash V(x) : \Gamma(x)\ \forall x \in \text{domain}(V)\end{array}}{\Theta;\Sigma \vdash V : \Gamma} \qquad \frac{\begin{array}{c}\text{domain}(H) = \text{domain}(\Sigma) \\ \Theta;\Sigma \vdash H(\ell) : \Sigma(\ell)\ \forall \ell \in \text{domain}(H)\end{array}}{\Theta \vdash H : \Sigma}$$

$$\frac{\begin{array}{c}\Theta = decl_1,\ldots,decl_n \qquad \Theta \vdash decl_i\ \forall 1 \le i \le n \\ \Theta \vdash H : \Sigma \qquad \Theta;\Sigma \vdash V : \Gamma \qquad \Theta;\bullet;\Sigma;\Gamma \vdash e : \tau\end{array}}{\Sigma \vdash (\Theta; H; V; e) : \tau}$$

Figure 12: Well-formedness of Programs

Figure 12 lists the well-formedness rule for programs. A program $(\Theta; H; V; e)$ is well-formed with respect to a heap environment if all its declarations in Θ are well-formed (including interface declarations), the heap H respects the heap environment, the variable-value binding V is well-formed and the main expression e is well-typed (see Figure 12). The main expression e has no free type variables, and refers only to labels in H and variables in V.

Original Program	New Program	Side Conditions
$(H; V; x)$	$(H; V; V(x))$	
$(H; V; \text{c2r}(C(v)))$	$(H; V; v)$	
$(H; V; im2r(r2im[I, C](v)))$	$(H; V; v)$	
$(H; V; \text{new}[\tau]\{l_1 = v_1, \ldots, l_n = v_n\})$	$(H'; V; \ell)$	$H' = H, \ell \rightsquigarrow \{l_i = v_i\}_{i=1}^n$ ℓ is new
$(H; V; \ell.l_i)$	$(H; V; v_i)$	$H(\ell) = \{l_i = v_i\}_{i=1}^n$ $1 \le i \le n$
$(H; V; \ell.l_i := v_i' \text{ in } e)$	$(H'; V; e)$	$\begin{cases} \forall \ell' \neq \ell, H'(\ell') = H(\ell') \\ H(\ell) = \{l_1 = v_1, \ldots, l_n = v_n\} \\ H'(\ell) = \{l_1 = v_1, \ldots, l_i = v_i', \ldots, l_n = v_n\} \end{cases}$
$(H; V; \text{new}[v_0, \ldots, v_{n-1}]^\tau)$	$(H'; V; \ell)$	$H' = H, \ell \rightsquigarrow [v_0, \ldots, v_{n-1}]^\tau$ $\ell \notin \text{domain}(H)$
$(H; V; \ell[i])$	$(H; V; v_i)$	$H(\ell) = [v_0, \ldots, v_{n-1}]^\tau$ and $0 \le i \le n-1$
$(H; V; \ell[i] := v_i' \text{ in } e)$	$(H'; V; e)$	$\begin{cases} 0 \le i < n \quad \forall \ell' \neq \ell, H'(\ell') = H(\ell') \\ H(\ell) = [v_0, \ldots, v_{n-1}]^\tau \\ H'(\ell) = [v_0, \ldots, v_i', \ldots, v_{n-1}]^\tau \end{cases}$
$(H; V; x : \tau = v \text{ in } e)$	$(H; V'; e)$	$V' = V, x = v$
$(H; V; x := v' \text{ in } e)$	$(H; V'; e)$	$V = V_1, x = v, V_2 \quad V' = V_1, x = v', V_2$
$(H; V; \ell[\tau_1, \ldots, \tau_m](v_1, \ldots, v_n))$	$(H; V'; e_m[\vec{\tau}/tvs])$	$\begin{cases} H(\ell) = \text{fix } g\langle tvs\rangle(x_i : \tau_i)_{i=1}^n : \tau = e_m \\ V' = V, x_1 = v_1, \ldots, x_n = v_n, g = \ell \end{cases}$
$(H; V; (\alpha, x) = \text{open}(v) \text{ in } e_0)$	$(H; V'; e_0[\tau/\alpha])$	$v = \text{pack } \tau \text{ as } \beta \ll \tau_u \text{ in } (v' : \tau')$ $V' = V, x = v'$
$(H; V; (\alpha, x) = \text{open}(v) \text{ in } e_0)$	$(H; V'; e_0[\tau/\alpha])$	$v = \text{pack } \tau \text{ as } \beta \gg \tau_u \text{ in } (v' : \tau')$ $V' = V, x = v'$
$(H; V; \text{ifParent}^\tau(v) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2)$	$(H; V'; e_1[B/\alpha])$	$v = tag(C)$ C extends B $V' = V, x = tag(B)$
$(H; V; \text{ifParent}^\tau(v) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2)$	$(H; V; e_2)$	$v = tag(\text{Topc})$
$(H; V; \text{ifEqTag}^\tau(v_1, v_2) \text{ then } e_1 \text{ else } e_2)$	$(H; V; e_1)$	$v_1 = tag(C_1), v_2 = tag(C_2), C_1 = C_2$
$(H; V; \text{ifEqTag}^\tau(v_1, v_2) \text{ then } e_1 \text{ else } e_2)$	$(H; V; e_2)$	$v_1 = tag(C_1), v_2 = tag(C_2), C_1 \neq C_2$
$(H; V; \text{error}[\tau])$	$(H; V; \text{error}[\tau])$	

Figure 13: Evaluation Rules

4.5 Dynamic Semantics

Figure 13 shows the evaluation rules for the ECI expressions. The expressions in the first column evaluate one step to the corresponding ones in the second column, if the side conditions in the third column hold. Judgment $P \mapsto P'$ means that P steps to P'.

Expressions $C(v)$ and $\text{c2r}(v)$ coerce between objects and records. Expressions $r2im[\tau_1, \tau_2](v)$ and $im2r(v)$ coerce between concrete and abstract interface method tables. "$\text{ifParent}^\tau(tag(C))$ then bind (α, x) in e_1 else e_2" steps to e_1 with α replaced with B and x assigned $tag(B)$, if C extends some class B. Otherwise it steps to e_2. Expression "$\text{ifEqTag}^\tau(tag(C_1), tag(C_2))$ then e_1 else e_2" steps to e_1 if $C_1 = C_2$, and to e_2 otherwise. The evaluation of an error expression "$\text{error}[\tau]$" infinitely loops.

4.6 Properties of ECI

We have proved that ECI has the following properties:

Theorem 1 (Preservation) *If $\Sigma \vdash P : \tau$ and $P \mapsto P'$, then $\exists \Sigma'$ such that $\Sigma' \vdash P' : \tau$.*

Theorem 2 (Progress) *If $\Sigma \vdash P : \tau$, then the main expression in P is a value, or $\exists P'$ such that $P \mapsto P'$.*

Theorem 3 (Decidability) *It is decidable whether $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ holds.*

Proofs can be found in Appendix A.

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 [6]. It compiles Java to assembly code with type annotations. The target language preserves class and interface names, but the paper [6] 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.

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 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.

[6] 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.

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

[8] 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.

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

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

[11] E. M. Gagnon and L. J. Hendren. Sablevm: A research framework for the efficient execution of java bytecode. In *In Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 27–40, 2001.

[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.

A Decidability and Soundness Proofs of ECI

This section presents the proofs for decidability and soundness of ECI. We describe only important lemmas and proof sketches.

A.1 Supporting Lemmas

Lemma 4 *Weakening of kind environment:*
Suppose $\alpha \notin domain(\Delta)$*, and* $\Delta' = \Delta, \alpha \ll u$ *or* $\Delta' = \Delta, \alpha \gg u$*:*

1. *If* $\Theta; \Delta \vdash \tau : \kappa$*, then* $\Theta; \Delta' \vdash \tau : \kappa$.
2. *If* $\Theta; \Delta \vdash \tau_1 \ll \tau_2$*, then* $\Theta; \Delta' \vdash \tau_1 \ll \tau_2$.
3. *If* $\Theta; \Delta \vdash \tau_1 \leq \tau_2$*, then* $\Theta; \Delta' \vdash \tau_1 \leq \tau_2$

Proof: by induction on kinding, subclassing and subtyping rules.

Lemma 5 *Weakening of heap environment. If* $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ *and* $\Sigma' = \Sigma, \ell : \tau$ *where* ℓ *is a fresh label, then* $\Theta; \Delta; \Sigma'; \Gamma \vdash e : \tau$.

Proof: by induction on expression typing rules.

Lemma 6 *Weakening of type environment. If* $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ *and* $\Gamma' = \Gamma, x : \tau$ *or* $\Gamma' = \Gamma, x :_{\mathrm{M}} \tau$ *where* x *is a fresh variable, then* $\Theta; \Delta; \Sigma; \Gamma' \vdash e : \tau$.

Proof: by induction on expression typing rules.

Lemma 7 *Type substitution preserves kinding, subclassing and subtyping.*
Suppose $\Delta = \Delta_1, \eta \ll \tau, \Delta_2$ *and* $\Theta; \Delta_1 \vdash s \ll \tau$ *(or* $\Delta = \Delta_1, \eta \gg \tau, \Delta_2$ *and* $\Theta; \Delta_1 \vdash \tau \ll s$*), and* $\delta = s/\eta$ *and* $\Delta' = \Delta_1, \Delta_2[\delta]$.

- *If* $\Theta; \Delta \vdash \tau : \kappa$*, then* $\Theta; \Delta' \vdash \tau[\delta] : \kappa$.
- *If* $\Theta; \Delta \vdash \tau_1 \ll \tau_2$*, then* $\Theta; \Delta' \vdash \tau_1[\delta] \ll \tau_2[\delta]$.
- *If* $\Theta; \Delta \vdash \tau_1 \leq \tau_2$*, then* $\Theta; \Delta' \vdash \tau_1[\delta] \leq \tau_2[\delta]$.

Proof: by induction on kinding, subclassing and subtyping rules.

A.1.1 Subclassing

Definition 8 *Class/interface and type variable lifting. We define an operation $\tau \uparrow_\Delta$ to compute the least super class name for type τ in environment Δ. This is used in minimal typing.*

$$\begin{array}{lcl} C \uparrow_\Delta & = & C \\ I \uparrow_\Delta & = & \mathit{Topc} \\ \alpha \uparrow_\Delta & = & \begin{cases} \tau \uparrow_\Delta & \alpha \ll \tau \in \Delta \\ \mathit{Topc} & \alpha \gg \tau \in \Delta \end{cases} \end{array}$$

By definition, $\tau \uparrow_\Delta$ is a class name. The process of computing $\tau \uparrow_\Delta$ always terminates because there is no loop in Δ. Also it has the following properties:

Lemma 9 *Properties of class/interface and type variable lifting*

1. $\Theta; \Delta \vdash \gamma \ll \gamma \uparrow_\Delta$.
2. *If* $\Theta; \Delta \vdash \tau_1 \ll \tau_2$, *then* $\Theta; \Delta \vdash \tau_1 \uparrow_\Delta \ll \tau_2 \uparrow_\Delta$.

Proof: *1*: by induction on depth where depth is defined as: $\text{depth}(\alpha) = \text{depth}(\tau) + 1$ if $\alpha \ll \tau \in \Delta$, and for all other cases, $\text{depth}(\tau) = 0$.
2: by induction on subclassing rules.

A.1.2 Subtyping

We define type lifting and type lowering in Figure 14. They are also used in minimal typing.

$(\tau) \Uparrow^\alpha_\Delta$ is defined to be the least super type of τ under environment Δ that contains no free occurence of α. Similarly, $(\tau) \Downarrow^\alpha_\Delta$ is defined to be the greatest subtype of τ under environment Δ that contains no free occurence of α. Note that α appears as the last entry in Δ. The definitions of lifting and lowering are mutually recursive.

Type lifting and lowering have the following properties:

Lemma 11 *Properties of Type Lifting.*
Let $\Delta' = \Delta, \alpha \ll u_\alpha$ or $\Delta' = \Delta, \alpha \gg u_\alpha$.

1. *If $\exists s$ such that $\alpha \notin free(s)$ and $\Theta; \Delta' \vdash \tau \le s$, then $(\tau) \Uparrow^\alpha_{\Delta'}$ exists, and $\Theta; \Delta' \vdash \tau \le (\tau) \Uparrow^\alpha_{\Delta'}$, and $\forall t$ such that $\Theta; \Delta' \vdash \tau \le t$ and $\alpha \notin free(t)$, then $\Theta; \Delta \vdash (\tau) \Uparrow^\alpha_{\Delta'} \le t$.*
2. *If $\Theta; \Delta' \vdash s \le \tau$ and $\alpha \notin free(s)$, then $(\tau) \Downarrow^\alpha_{\Delta'}$ exists and $\Theta; \Delta' \vdash (\tau) \Downarrow^\alpha_{\Delta'} \le \tau$, and $\forall t$ such that $\Theta; \Delta' \vdash t \le \tau$ and $\alpha \notin free(t)$, $\Theta; \Delta \vdash t \le (\tau) \Downarrow^\alpha_{\Delta'}$.*
3. *If $\Theta; \Delta' \vdash \tau_1 \le \tau_2$, then $\Theta; \Delta \vdash (\tau_1) \Uparrow^\alpha_{\Delta'} \le (\tau_2) \Uparrow^\alpha_{\Delta'}$.*

Proof:
1, 2: by mutal induction on the definition of lifting and lowering.
3: let $T_1 = (\tau_1) \Uparrow^\alpha_{\Delta'}$ and $T_2 = (\tau_2) \Uparrow^\alpha_{\Delta'}$. By *1* $\Theta; \Delta' \vdash \tau_2 \le T_2$. By transitivity of subtyping $\Theta; \Delta' \vdash \tau_1 \le T_2$. By *1* and $\alpha \notin free(T_2)$, $\Theta; \Delta \vdash T_1 \le T_2$.

Lemma 12 *Properties of Approximation*

- $\mathsf{R}(C)$ *contains no free type variables.* $\mathsf{ApproxR}(\alpha, C)$ *contains no free type variables other than* α.
- *If* $\Theta; \Delta \vdash C_1 \ll C_2$ *and* $\Theta; \Delta \vdash \alpha : \Omega_c$, *then* $\Theta; \Delta \vdash \mathsf{ApproxR}(\alpha, C_1) \le \mathsf{ApproxR}(\alpha, C_2)$.
- *If* $\Theta; \Delta \vdash C_1 \ll C_2$, *then* $\Theta; \Delta \vdash \mathsf{R}(C_1) \le \mathsf{ApproxR}(\alpha, C_2)[C_1/\alpha]$.

Proof: by the definitions of R, $\mathsf{ApproxR}$ and record subtyping rules.

Definition 10 *Type lifting and lowering. Suppose* $\Delta = \Delta_1, P(\alpha)$ *where* $P(\alpha) = \alpha \ll u_\alpha$ *or* $P(\alpha) = \alpha \gg u_\alpha$*:*

$$
\begin{array}{lcll}
(\tau) \Uparrow^\alpha_\Delta & = & \tau & \alpha \notin free(\tau) \\
(\exists \beta \ll u.\ \tau) \Uparrow^\alpha_\Delta & = & \exists \beta \ll u.\ \tau' & u \neq \alpha \text{ and } \tau' = (\tau) \Uparrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\exists \beta \ll \alpha.\ \tau) \Uparrow^\alpha_\Delta & = & \exists \beta \ll u_\alpha.\ \tau' & P(\alpha) = \alpha \ll u_\alpha \text{ and } \tau' = (\tau) \Uparrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\exists \beta \ll \alpha.\ \tau) \Uparrow^\alpha_\Delta & = & \exists \beta \ll Topc.\ \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \Uparrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\exists \beta \gg u.\ \tau) \Uparrow^\alpha_\Delta & = & \exists \beta \gg u.\ \tau' & u \neq \alpha \text{ and } \tau' = (\tau) \Uparrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\exists \beta \gg \alpha.\ \tau) \Uparrow^\alpha_\Delta & = & \exists \beta \gg u_\alpha.\ \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \Uparrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\forall \beta \ll u.\ \tau) \Uparrow^\alpha_\Delta & = & \forall \beta \ll u.\ \tau' & u \neq \alpha \text{ and } \tau' = (\tau) \Uparrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\forall \beta \ll \alpha.\ \tau) \Uparrow^\alpha_\Delta & = & \forall \beta \ll u_\alpha.\ \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \Uparrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
((\tau_1, \ldots, \tau_n) \to \tau) \Uparrow^\alpha_\Delta & = & (\tau'_1, \ldots, \tau'_n) \to \tau' & \tau'_i = (\tau_i) \Downarrow^\alpha_\Delta \text{ and } \tau' = (\tau) \Uparrow^\alpha_\Delta \\
(\{l_i^{\phi_i} : \tau_i\}_{i=1}^n) \Uparrow^\alpha_\Delta & = & \{l_i^{\phi_i} : \tau'_i\}_{i=1}^m & \left\{ \begin{array}{l} m = max \left\{ j \;\middle|\; \begin{array}{l} \forall 1 \leq i \leq j \text{ if } \phi_i = \mathrm{M} \text{ then } \alpha \notin free(\tau_i) \\ \text{otherwise } (\tau_i) \Uparrow^\alpha_\Delta \text{ exists} \end{array} \right\} \\ \tau'_i = (\tau_i) \Uparrow^\alpha_\Delta \ \forall \phi_i = \mathrm{I} \text{ and } \tau_i \ \forall \phi_i = \mathrm{M} \ \forall 1 \leq i \leq m \end{array} \right. \\
(\{\!\{l_i^{\phi_i} : \tau_i\}\!\}_{i=1}^n) \Uparrow^\alpha_\Delta & = & (\{l_i^{\phi_i} : \tau_i\}_{i=1}^n) \Uparrow^\alpha_\Delta & \\
 & & & \\
(\tau) \Downarrow^\alpha_\Delta & = & \tau & \alpha \notin free(\tau) \\
(\exists \beta \ll u.\ \tau) \Downarrow^\alpha_\Delta & = & \exists \beta \ll u.\ \tau' & u \neq \alpha \text{ and } \tau' = (\tau) \Downarrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\exists \beta \ll \alpha.\ \tau) \Downarrow^\alpha_\Delta & = & \exists \beta \ll u_\alpha.\ \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \Downarrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\exists \beta \gg u.\ \tau) \Downarrow^\alpha_\Delta & = & \exists \beta \gg u.\ \tau' & u \neq \alpha \text{ and } \tau' = (\tau) \Downarrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\exists \beta \gg \alpha.\ \tau) \Downarrow^\alpha_\Delta & = & \exists \beta \gg Topc.\ \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \Downarrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\exists \beta \gg \alpha.\ \tau) \Downarrow^\alpha_\Delta & = & \exists \beta \gg u_\alpha.\ \tau' & P(\alpha) = \alpha \ll u_\alpha \text{ and } \tau' = (\tau) \Downarrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\forall \beta \ll u.\ \tau) \Downarrow^\alpha_\Delta & = & \forall \beta \ll u.\ \tau' & u \neq \alpha \text{ and } \tau' = (\tau) \Downarrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\forall \beta \ll \alpha.\ \tau) \Downarrow^\alpha_\Delta & = & \forall \beta \ll Topc.\ \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \Downarrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
(\forall \beta \ll \alpha.\ \tau) \Downarrow^\alpha_\Delta & = & \forall \beta \ll u_\alpha.\ \tau' & P(\alpha) = \alpha \ll u_\alpha \text{ and } \tau' = (\tau) \Downarrow^\alpha_{\Delta_1, \beta \ll Topc, P(\alpha)} \\
((\tau_1, \ldots, \tau_n) \to \tau) \Downarrow^\alpha_\Delta & = & (\tau'_1, \ldots, \tau'_n) \to \tau' & \tau'_i = (\tau_i) \Uparrow^\alpha_\Delta \text{ and } \tau' = (\tau) \Downarrow^\alpha_\Delta \\
(\{l_i^{\phi_i} : \tau_i\}_{i=1}^n) \Downarrow^\alpha_\Delta & = & \{l_i^{\phi_i} : \tau'_i\}_{i=1}^n & \left\{ \begin{array}{l} \forall 1 \leq i \leq n \text{ if } \phi_i = \mathrm{M} \text{ then } \alpha \notin free(\tau_i) \\ \qquad\qquad \text{otherwise } (\tau_i) \Downarrow^\alpha_\Delta \text{ exists} \\ \tau'_i = (\tau_i) \Downarrow^\alpha_\Delta \ \forall \phi_i = \mathrm{I} \text{ and } \tau_i \ \forall \phi_i = \mathrm{M} \end{array} \right.
\end{array}
$$

Figure 14: Type Lifting and Lowering

$$\frac{\forall 1 \le i \le n \ \begin{cases} \Theta; \Delta \vDash \tau_i \le \tau_i' & \text{if } \phi_i = \mathrm{I} \\ \tau_i = \tau_i' & \text{if } \phi_i = \mathrm{M} \end{cases}}{\Theta; \Delta \vDash \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n, \ldots\} \le \{l_1^{\phi_1} : \tau_1', \ldots, l_n^{\phi_n} : \tau_n'\}} \ \textbf{ast\_record}$$

$$\frac{\forall 1 \le i \le n \ \begin{cases} \Theta; \Delta \vDash \tau_i \le \tau_i' & \text{if } \phi_i = \mathrm{I} \\ \tau_i = \tau_i' & \text{if } \phi_i = \mathrm{M} \end{cases}}{\Theta; \Delta \vDash \{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_1} : \tau_n, \ldots\}\!\} \le \{l_1^{\phi_1} : \tau_1', \ldots, l_n^{\phi_n} : \tau_n'\}} \ \textbf{ast\_exact}$$

$$\frac{\Theta; \Delta \vdash u_1 \ll u_2 \quad \Theta; \Delta, \alpha \ll \mathrm{Topc} \vDash \tau_1 \le \tau_2}{\Theta; \Delta \vDash (\exists \alpha \ll u_1.\ \tau_1) \le (\exists \alpha \ll u_2.\ \tau_2)} \ \textbf{ast\_}\exists$$

$$\frac{\Theta; \Delta \vdash u_2 \ll u_1 \quad \Theta; \Delta, \alpha \ll \mathrm{Topc} \vDash \tau_1 \le \tau_2}{\Theta; \Delta \vDash (\exists \alpha \gg u_1.\ \tau_1) \le (\exists \alpha \gg u_2.\ \tau_2)} \ \textbf{ast\_}\exists \gg$$

$$\frac{\Theta; \Delta \vdash u_2 \ll u_1 \quad \Theta; \Delta, \alpha \ll \mathrm{Topc} \vDash \tau_1 \le \tau_2}{\Theta; \Delta \vDash (\forall \alpha \ll u_1.\ \tau_1) \le (\forall \alpha \ll u_2.\ \tau_2)} \ \textbf{ast\_}\forall$$

$$\frac{\Theta; \Delta \vDash t_i \le s_i \ \forall 1 \le i \le n \quad \Theta; \Delta \vDash s \le t}{\Theta; \Delta \vDash (s_1, \ldots, s_n) \to s \le (t_1, \ldots, t_n) \to t} \ \textbf{ast\_fun} \qquad \frac{}{\Theta; \Delta \vDash \tau \le \tau} \ \textbf{ast\_ref}$$

Figure 15: Algorithmic Subtyping Rules

A.2 Decidability of Type Checking

Lemma 13 *Subclassing is decidable.*

Proof: subclassing is a partial order. Class names, interface names, and type variables constitute a finite partial order set. One simple way to decide whether a type τ_1 is a subclass of type τ_2 is by brute-forth search: first get all the types τ such that $\tau_1 \ll \tau$ and then test whether τ_2 is one of the super types.

A.2.1 Decidable Subtyping

We develop a new set of subtyping rules that is equivalent to the existing set but does not have the transitivity rule, called algorithmic subtyping rules (shown in Figure 15). This new set is syntax-directed except for the reflexivity rule, thus can be used as an algorithm for checking subtyping.

Lemma 14 *The algorithmic subtyping rules are sound. If* $\Theta; \Delta \vDash T_1 \le T_2$*, then* $\Theta; \Delta \vdash T_1 \le T_2$*.*

Proof: by induction on the algorithmic subtyping rules.

Lemma 15 *The algorithmic subtyping is transitive. If* $\Theta; \Delta \vDash T_1 \le T_2$ *and* $\Theta; \Delta \vDash T_2 \le T_3$*, then* $\Theta; \Delta \vDash T_1 \le T_3$*.*

Proof: by induction on the sum of the sizes of the left derivation $\Theta; \Delta \vDash T_1 \le T_2$ and the right derivation $\Theta; \Delta \vDash T_2 \le T_3$.

Lemma 16 *The algorithmic subtyping is complete. If* $\Theta; \Delta \vdash T_1 \le T_2$*, then* $\Theta; \Delta \vDash T_1 \le T_2$*.*

Proof: by induction on the subtyping rules. The transitivity case is proven by Lemma 15.

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vDash e : \alpha}{\Theta;\Delta;\Sigma;\Gamma \vDash \text{c2r}(e) : \mathsf{ApproxR}(\alpha, \alpha\uparrow_\Delta)}\ \textbf{a\_c2r\_tv}$$

$$\frac{\begin{array}{c}\Theta;\Delta;\Sigma;\Gamma \vDash e_1 : \exists\beta \ll \tau_u.\ \tau \qquad \alpha \notin \text{domain}(\Delta) \\ \Theta;\Delta,\alpha \ll \tau_u;\Sigma;\Gamma, x : \tau[\alpha/\beta] \vDash e_2 : \tau'\end{array}}{\Theta;\Delta;\Sigma;\Gamma \vDash (\alpha, x) = \text{open}(e_1)\ \text{in}\ e_2 : (\tau') \Uparrow^\alpha_{\Delta,\alpha\ll\tau_u}}\ \textbf{a\_open}$$

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

Figure 16: Selected Algorithmic Expression Typing Rules

Definition 17 *size of types.*

$$\begin{array}{lcl}
size(\forall\alpha \ll \tau.\ \tau') & = & size(\tau') + 1 \\
size(\exists\alpha \ll \tau.\ \tau') & = & size(\tau') + 1 \\
size(\exists\alpha \gg \tau.\ \tau') & = & size(\tau') + 1 \\
size((\tau_1, \ldots, \tau_n) \to \tau) & = & \sum_{i=1}^n size(\tau_i) + size(\tau) + 1 \\
size(\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}) & = & \sum_{i=1}^n size(\tau_i) + 1 \\
size(\{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}\!\}) & = & \sum_{i=1}^n size(\tau_i) + 1 \\
size(\tau) & = & 1 \quad otherwise
\end{array}$$

Lemma 18 *The algorithmic subtyping terminates.*

Proof: in all algorithmic subtyping rules, the sum of the sizes of the types in the subtyping judgments of the premises is strictly smaller than that of the conclusion. Also, the side conditions such as subclassing are decidable too.

Corollary 19 *Subtyping is decidable. It is decidable to check whether* $\Theta;\Delta \vdash \tau_1 \le \tau_2$ *holds.*

Proof: by Lemmas 14, 16 and 18.

We will use the subtyping rules in Figure 8 and the algorithmic subtyping rules in Figure 15 interchangeably in the rest of the proof.

A.2.2 Minimal Typing

We design a set of algorithmic expression typing rules. Most rules are standard. Figure 16 lists non-standard rules. In **a_c2r_tv**, to get the minimal typing of a record coerced from an object with type α, we need to find the least upper subclassing bound of α that is a concrete class name, which is $\alpha\uparrow_\Delta$ (defined in Section A.1.1). In **a_open** and **a_open**$\gg$, the minimal type of the expression e_2 might contain type variable α, whereas the minimal type of the open expression should not contain α. Therefore, we need to find the least subtyping bound of the minimal type of e_2, which is $(\tau)\Uparrow^\alpha_\Delta$ (defined in Section A.1.2).

This new set of rules is syntax-directed. We prove that it is sound and complete with respect to the original expression typing rules.

Lemma 20 *Minimal typing is sound. If* $\Theta;\Delta;\Sigma;\Gamma \vDash E : T$*, then* $\Theta;\Delta;\Sigma;\Gamma \vdash E : T$*.*

Proof: by induction on algorithmic expression typing rules. **Case a_c2r_tv** $E = \text{c2r}(e), T = \mathsf{ApproxR}(\alpha, \alpha\uparrow_\Delta)$ with subderivation $\Theta;\Delta;\Sigma;\Gamma \vDash e : \alpha$.

By induction hypotheis, $\Theta;\Delta;\Sigma;\Gamma \vdash e : \alpha$. By Lemma 9, $\Theta;\Delta \vdash \alpha \ll \alpha\uparrow_\Delta$ and $\alpha\uparrow_\Delta$ is a concrete class name. By **c2r_tv** $\Theta;\Delta;\Sigma;\Gamma \vdash E : T$.

Case a_open $E = (\alpha, x) = \text{open}(e_1)$ in e_2, $T = (\tau') \Uparrow^{\alpha}_{\Delta, \alpha \ll \tau_u}$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vDash e_1 : \exists \beta \ll \tau_u.\ \tau$, $\Theta; \Delta, \alpha \ll \tau_u; \Sigma; \Gamma, x : \tau[\alpha/\beta] \vDash e_2 : \tau'$ ($\alpha \notin \text{domain}(\Delta)$).

By induction hypothesis, $\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \exists \beta \ll \tau_u.\ \tau$ and $\Theta; \Delta, \alpha \ll \tau_u; \Sigma; \Gamma, x : \tau[\alpha/\beta] \vdash e_2 : \tau'$. By properties of type lifting Lemma 11, $\Theta; \Delta, \alpha \ll \tau_u \vdash \tau' \leq T$. By **sub** we have $\Theta; \Delta, \alpha \ll \tau_u; \Sigma; \Gamma, x : \tau[\alpha/\beta] \vdash e_2 : T$. By the definition of type lifting, $\alpha \notin free(T)$. Therefore, $\Theta; \Delta; \Sigma; \Gamma \vdash E : T$ by **open**.

Lemma 21 *Inversion of Subtyping*

- *If* $\Theta; \Delta \vdash s \leq \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}$, *then either* $s = \{l_1^{\phi_1} : \tau_1', \ldots, l_n^{\phi_n} : \tau_n', \ldots\}$ *or* $s = \{\!\{l_1^{\phi_1} : \tau_1', \ldots, l_n^{\phi_n} : \tau_n', \ldots\}\!\}$. *In either case* $\Theta; \Delta \vdash \tau_i' \leq \tau_i\ \forall \phi_i = \text{I}$, *and* $\tau_i' = \tau_i\ \forall \phi_i = \text{M}$.
- *If* $\Theta; \Delta \vdash s \leq \{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}\!\}$, *then* $s = \{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}\!\}$.
- *If* $\Theta; \Delta \vdash s \leq array(\tau)$, *then* $s = array(\tau)$.
- *If* $\Theta; \Delta \vdash s \leq int$, *then* $s = int$.
- *If* $\Theta; \Delta \vdash S \leq \forall \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m.(\tau_1, \ldots, \tau_n) \to \tau$, *then (1)* $S = \forall \alpha_1 \ll u_1', \ldots, \alpha_m \ll u_m'.(s_1, \ldots, s_n) \to s$, *(2)* $\Theta; \Delta, \alpha_1 \ll Topc, \ldots, \alpha_m \ll Topc \vdash \tau_i \leq s_i\ \forall 1 \leq i \leq n$, *(3)* $\Theta; \Delta, \alpha_1 \ll Topc, \ldots, \alpha_m \ll Topc \vdash s \leq \tau$, *and (4) if* $\exists t_1, \ldots, t_m$ *such that* $\forall 1 \leq i \leq m$, $\Theta; \Delta \vdash t_i \ll u_i[t_1, \ldots, t_m/\alpha_1, \ldots, \alpha_m]$, *then* $\forall 1 \leq i \leq m$, $\Theta; \Delta \vdash t_i \ll u_i'[t_1, \ldots, t_m/\alpha_1, \ldots, \alpha_m]$.
- *If* $\Theta; \Delta \vdash s \leq \exists \alpha \ll \tau_1.\ \tau_2$, *then* $s = \exists \alpha \ll s_1.\ s_2$, $\Theta; \Delta \vdash s_1 \ll \tau_1$ *and* $\Theta; \Delta, \alpha \ll Topc \vdash s_2 \leq \tau_2$.
- *If* $\Theta; \Delta \vdash s \leq \exists \alpha \gg \tau_1.\ \tau_2$, *then* $s = \exists \alpha \gg s_1.\ s_2$, $\Theta; \Delta \vdash \tau_1 \ll s_1$ *and* $\Theta; \Delta, \alpha \ll Topc \vdash s_2 \leq \tau_2$.
- *If* $\Theta; \Delta \vdash s \leq Tag(\tau)$, *then* $s = Tag(\tau)$.
- *If* $\Theta; \Delta \vdash s \leq \tau$ *and* $\Theta; \Delta \vdash \tau : \Omega_c$, *then* $s = \tau$.
- *If* $\Theta; \Delta \vdash s \leq Imty(\tau_1, \tau_2)$, *then* $s = Imty(\tau_1, \tau_2)$.

Proof: by inspection of algorithmic subtyping rules.

Lemma 22 *Minimal typing is complete. If* $\Theta; \Delta; \Sigma; \Gamma \vdash E : T$, *then* $\exists T_m$ *such that* $\Theta; \Delta; \Sigma; \Gamma \vDash E : T_m$ *and* $\Theta; \Delta \vdash T_m \leq T$.

Proof: by induction on the expression typing rules.
We define the size of expressions in Figure 17.

Lemma 24 *Minimal Typing terminates.*

Proof: in each inference rule, the premises refer to only subexpressions of the expression in the conclusion. The size of the expression in the conclusion is always larger than the sum of the subexpression sizes in the premises. Also the side conditions are decidable.

Theorem 25 *Type checking of ECI is decidable.*

Proof: to decide whether $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ holds, we can first get the minimal type τ_m of e such that $\Theta; \Delta; \Sigma; \Gamma \vDash e : \tau_m$, then test whether $\Theta; \Delta \vdash \tau_m \leq \tau$. Because both minimal typing and subtyping are decidable, type checking is decidable.

A.3 Soundness

A.3.1 Substitution

Lemma 26 *It preserves typing to substitute an appropriate type for the* first *type variable in the kind environment. If* $\Theta; \Delta; \Sigma; \Gamma \vdash E : T$, $\Delta = \eta \ll \tau, \Delta'$ *and* $\Theta; \bullet \vdash s \ll \tau$ *(or* $\Delta = \eta \gg \tau, \Delta'$ *and* $\Theta; \bullet \vdash \tau \ll s$*), and* $\delta = s/\eta$, *then* $\Theta; \Delta'[\delta]; \Sigma[\delta]; \Gamma[\delta] \vdash E[\delta] : T[\delta]$.

Proof: by induction on expression typing rules.

Definition 23 *Expression Sizes.*

$$
\begin{array}{lcl}
Size(C(e)) & = & Size(e)+1 \\
Size(c2r(e)) & = & Size(e)+1 \\
Size(r2im[I,C](e)) & = & Size(e)+1 \\
Size(im2r(e)) & = & Size(e)+1 \\
Size(new[\tau]\{l_1 = e_1, \ldots, l_n = e_n\}) & = & Size(e_1)+\ldots+Size(e_n)+1 \\
Size(e.l) & = & Size(e)+1 \\
Size(e_1.l_i := e_2 \ in \ e_3) & = & Size(e_1)+Size(e_2)+Size(e_3)+1 \\
Size(new[e_0, \ldots, e_{n-1}]^\tau) & = & Size(e_0)+\ldots+Size(e_{n-1})+1 \\
Size(e_1[e_2]) & = & Size(e_1)+Size(e_2)+1 \\
Size(e_1[e_2] := e_3 \ in \ e_4) & = & Size(e_1)+Size(e_2)+Size(e_3)+Size(e_4)+1 \\
Size(x : \tau = e \ in \ e') & = & Size(e)+Size(e')+1 \\
Size(x := e_1 \ in \ e_2) & = & Size(e_1)+Size(e_2)+1 \\
Size(len(e)) & = & Size(e)+1 \\
Size(e[\tau_1, \ldots, \tau_m](e_1, \ldots, e_n)) & = & Size(e)+Size(e_1)+\ldots+Size(e_n)+1 \\
Size(pack \ \tau \ as \ \alpha \ll \tau_u \ in \ (e : \tau')) & = & Size(e)+1 \\
Size((\alpha, x) = open(e) \ in \ e') & = & Size(e)+Size(e')+1 \\
Size(pack \ \tau \ as \ \alpha \gg \tau_u \ in \ (e : \tau')) & = & Size(e)+1 \\
Size(ifParent^\tau(e) \ then \ bind \ (\alpha, x) \ in \ e_1 \ else \ e_2) & = & Size(e)+Size(e_1)+Size(e_2)+1 \\
Size(ifEqTag^\tau(e_{t1}, e_{t2}) \ then \ e_1 \ else \ e_2) & = & Size(e_{t1})+Size(e_{t2})+Size(e_1)+Size(e_2)+1 \\
Size(e) & = & 1 \ \ otherwise
\end{array}
$$

Figure 17: Expression Sizes

A.3.2 Preservation

Lemma 27 *Inversion of Values*

- *If* $H(\ell) = \{k_1 = v_1, \ldots, k_m = v_m\}$ *and* $\Theta; \bullet; \Sigma; \Gamma \vdash \ell : \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}$, *then* $m \geq n$, $k_i = l_i$ *and* $\Theta; \bullet; \Sigma; \Gamma \vdash v_i : \tau_i \ \forall 1 \leq i \leq n$.
- *If* $H(\ell) = [v_0, \ldots, v_{n-1}]^\tau$ *and* $\Theta; \bullet; \Sigma; \Gamma \vdash \ell : array(\tau')$, *then* $\tau' = \tau$ *and* $\Theta; \bullet; \Sigma; \Gamma \vdash v_i : \tau \ \forall 0 \leq i \leq n-1$.
- *If* $\Theta; \bullet; \Sigma; \Gamma \vdash \ell : \forall \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m.(\tau_1, \ldots, \tau_n) \to \tau$, *and* $\Sigma(\ell) = fun \ < \alpha_1 \ll u_1', \ldots, \alpha_m \ll u_m' > (x_1 : s_1, \ldots, x_n : s_n) : s = e_0$, *then (1) let* $\tau_f = \forall \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m.(\tau_1, \ldots, \tau_n) \to \tau$ *and* $\tau_f' = \forall \alpha_1 \ll u_1', \ldots, \alpha_m \ll u_m'.(s_1, \ldots, s_n) \to s$, *then* $\Sigma(\ell) = \tau_f'$ *and* $\Theta; \bullet \vdash \tau_f' \leq \tau_f$. *(2)* $\Theta; tvs'; \Sigma; g : \tau_f', x_1 : s_1, \ldots, x_n : s_n \vdash e_0 : s$.
- *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : \exists \beta \ll \tau_u.\ \tau$ *and* $v = pack \ s_0 \ as \ \beta \ll s_u \ in \ (v' : s)$, *then* $\Theta; \bullet \vdash s_0 \ll s_u$, $\Theta; \bullet \vdash s_u \ll \tau_u$, $\Theta; \beta \ll Topc \vdash s \leq \tau$, *and* $\Theta; \bullet; \Sigma; \Gamma \vdash v' : s[s_0/\beta]$.
- *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : \exists \beta \gg \tau_u.\ \tau$ *and* $v = pack \ s_0 \ as \ \beta \gg s_u \ in \ (v' : s)$, *then* $\Theta; \bullet \vdash s_u \ll s_0$, $\Theta; \bullet \vdash \tau_u \ll s_u$, $\Theta; \beta \ll Topc \vdash s \leq \tau$, *and* $\Theta; \bullet; \Sigma; \Gamma \vdash v' : s[s_0/\beta]$.
- *If* $v = tag(C)$ *and* $\Theta; \bullet; \Sigma; \Gamma \vdash v : Tag(\tau)$, *then* $\tau = C$.
- *If* $\Theta; \bullet; \Sigma; \Gamma \vdash C(v) : C'$ *then* $C = C'$ *and* $\Theta; \bullet; \Sigma; \Gamma \vdash v : \mathsf{R}(C)$.
- *If* $\Theta; \bullet; \Sigma; \Gamma \vdash r2im[I,C](v) : Imty(I', C')$ *then* $I = I'$, $C = C'$ *and* $\Theta; \bullet; \Sigma; \Gamma \vdash v : \mathsf{R_I}(I, C)$.

Definition 28 *Extension of Environments.* Σ' *extends* Σ *means that* $\Sigma' = \Sigma, \Sigma_{new}$ *for some* Σ_{new} *such that* $domain(\Sigma) \cap domain(\Sigma_{new}) = \emptyset$. *Similarly,* Γ' *extends* Γ *means that* $\Gamma' = \Gamma, \Gamma_{new}$ *for some* Γ_{new} *such that* $domain(\Gamma) \cap domain(\Gamma_{new}) = \emptyset$.

Theorem 29 *Evaluation of ECI expressions preserves types. If*

- $freetvs(\Sigma) = \emptyset \quad \Theta \vdash H : \Sigma \quad \Theta; \Sigma \vdash V : \Gamma$

- $\Theta; \bullet; \Sigma; \Gamma \vdash E : T$
- $(H; V; E) \mapsto (H'; V'; E')$,

then $\exists \Sigma'$ *and* Γ' *such that*

1. $freetvs(\Sigma') = \emptyset \quad \Theta \vdash H' : \Sigma' \quad \Theta; \Sigma' \vdash V' : \Gamma'$
2. $\Theta; \bullet; \Sigma'; \Gamma' \vdash E' : T$
3. Σ' *extends* Σ *and* Γ' *extends* Γ.

Proof: by induction on expression typing rules.

Corollary 30 (Preservation) *If* $\Sigma \vdash P : \tau$ *and* $P \mapsto P'$, *then* $\exists \Sigma'$ *such that* $\Sigma' \vdash P' : \tau$.

A.3.3 Progress

Lemma 31 *Canonical forms*

1. *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}$ *and* $\Theta \vdash H : \Sigma$, *then* v *is a label and* $H(v) = \{l_1 = v_1, \ldots, l_n = v_n, \ldots\}$.
2. *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : array(\tau)$ *and* $\Theta \vdash H : \Sigma$, *then* v *is a label and* $H(v) = [v_0, \ldots, v_{n-1}]^\tau$.
3. *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : \forall tvs(\tau_1, \ldots, \tau_n) \to \tau$ $(tvs = \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m)$ *and* $\Theta \vdash H : \Sigma$, *then* v *is a label and* $H(v) = fix\ g\langle tvs' \rangle(x_1 : \tau_1', \ldots, x_n : \tau_n') : \tau' = e_m$ *and* $tvs' = \alpha_1 \ll u_1', \ldots, \alpha_m \ll u_m'$.
4. *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : \exists \alpha \ll \tau_u.\ \tau$, *then* $v = pack\ \tau_0\ as\ \alpha \ll \tau_u'\ in\ (v' : \tau')$.
5. *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : \exists \alpha \gg \tau_u.\ \tau$, *then* $v = pack\ \tau_0\ as\ \alpha \gg \tau_u'\ in\ (v' : \tau')$.
6. *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : Tag(\tau)$, *then* $v = tag(C)$ *for some* C.
7. *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : Tag(C)$, *then* $v = tag(C)$.
8. *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : C$, *then* $v = C(v')$ *for some value* v'.
9. *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : Imty(I, \tau)$, *then* $v = r2im[I, C](v')$ *for some value* v' *and class* C.
10. *If* $\Theta; \bullet; \Sigma; \Gamma \vdash v : int$, *then* v *is an integer.*

Proof: by inspection on expression typing rules, heap value rules and subtyping inversion Lemma 21.

Theorem 32 *If* $freetvs(\Sigma) = \emptyset$, *and* $\Theta \vdash H : \Sigma$, *and* $\Theta; \Sigma \vdash V : \Gamma$ *and* $\Theta; \bullet; \Sigma; \Gamma \vdash E : T$, *then either* E *is a value, or* E *can evaluate one step, that is,* $\exists H'$, V' *and* E' *such that* $(H; V; E) \mapsto (H'; V'; E')$.

Proof: by induction on expression typing rules.

Corollary 33 (progress) *If* $\Sigma \vdash P : \tau$, *then either the main expression in* P *is a value, or* $\exists P'$ *such that* $P \mapsto P'$.

Theorem 34 *ECI is Sound. Well-typed ECI programs do not get stuck.*

Proof: by progress and preservation.

$$
\begin{array}{lcl}
\tau & := & \text{int} \mid C \mid I \mid \text{array}(C) \\
e & := & n \mid \ell \mid x \mid x : \tau = e_1 \text{ in } e_2 \mid x := e_1 \text{ in } e_2 \mid new\ C\{f_1 = e_1, \ldots, f_n = e_n\} \mid e.f_i \\
 & & \mid e_1.f_i := e_2 \text{ in } e_3 \mid e.m(e_1, \ldots, e_n) \mid imthd[I.m]e(e_1, \ldots, e_n) \\
 & & \mid new[e_0, \ldots, e_{n-1}]^\tau \mid e_1[e_2] := e_3 \text{ in } e_4 \mid e_1[e_2] \mid \text{len}(e) \mid cast[C](e) \mid cast[I](e) \\
v & := & n \mid \ell \\
hv & := & C\{f_1 = v_1, \ldots, f_n = v_n\} \mid [v_0, \ldots, v_{n-1}]^\tau \\
mdecl & := & \tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e \\
msig & := & \tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n) \\
idecl & := & I : J_1, \ldots, J_n\{msig_1, \ldots, msig_k\} \\
cdecl & := & C : B, I_1, \ldots, I_p\{f_1 : \tau_1, \ldots, f_n : \tau_n, mdecl_1, \ldots, mdecl_k\} \\
decl & := & cdecl \mid idecl \\
prog & := & decl_1, \ldots, decl_n \text{ in } e
\end{array}
$$

Figure 18: Syntax of the Source Language

B The Source Language

This section describes the source language and formalizes the translation from the source language to ECI. Figure 18 shows the syntax of the source language.

The source language is roughly Featherweight Java (FJ) [13] enhanced with interfaces, assignments, and arrays. To simplify the representation, we use only one dimensional arrays of objects. The source language can support more general arrays with simple extensions. All local variables and record fields are mutable. Variables are renamed such that no two variables have the same name. A "new" expression is used to create objects instead of constructors as in FJ. Expression "$imthd[I.m]e(e_1, \ldots, e_n)$" invokes an interface method m (in interface I) on e with arguments $e_1, \ldots, e_n$. Interface cast expression $cast[I](e)$ casts an expression e to interface I. An interface declaration contains a sequence of superinterfaces and a sequence of method signatures. For simplicity, the source language requires that methods declared in each interface be different from those in the superinterfaces. The semantics of the source language is straightforward.

B.1 Semantics of the Source Language

Figures 19–24 show the semantics of the source language.

As shown in Figure 19, the source language has covariant arrays and does not separate subclassing from subtyping.

$$
\frac{\Theta(I) = I : J_1, \ldots, J_n\{\_\} \quad 1 \le i \le n}{\Theta \vdash I \le J_i}
\qquad
\frac{\Theta(C) = C : B, I_1, \ldots, I_p\{\ldots\} \quad \tau = B \text{ or } I_j \ \ \forall 1 \le j \le p}{\Theta \vdash C \le \tau}
$$

$$
\frac{\Theta \vdash \tau_1 \le \tau_2}{\Theta \vdash \text{array}(\tau_1) \le \text{array}(\tau_2)}
\qquad
\frac{}{\Theta \vdash \tau \le \tau}
\qquad
\frac{\Theta \vdash \tau_1 \le \tau_2 \quad \Theta \vdash \tau_2 \le \tau_3}{\Theta \vdash \tau_1 \le \tau_3}
$$

Figure 19: Subtyping of the Source Language

Figure 20 defines helper functions used for type checking source language programs. Suppose $\overrightarrow{F}$ is a set of field declarations, $\overrightarrow{M}$ is a set of method declarations, and $\overrightarrow{MS}$ is a set of method signatures. The function $fields(\Theta, C)$ returns all C's fields, including those from superclasses. The function $mtype(\Theta, m, C)$ returns the type of method m in class C. The function $mbody(\Theta, m, C)$ returns the implementations of method m in class C. If C has a declaration for m, $mtype$ and $mbody$ return the type and implementation for m respectively. Otherwise, they look in C's superclass. When applied to an interface I, $mtype(\Theta, m, I)$ returns the types of m in I.

$$\frac{\Theta(C) = C : B, I_1, \ldots, I_p\{\overrightarrow{F}, \overrightarrow{M}\}}{fields(\Theta, C) = fields(\Theta, B), \overrightarrow{F}}$$

$$\frac{\Theta(C) = C : B, I_1, \ldots, I_p\{\overrightarrow{F}, \overrightarrow{M}\} \quad \tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e \in \overrightarrow{M}}{\begin{array}{c} mtype(\Theta, m, C) = (\tau_1, \ldots, \tau_n) \to \tau \\ mbody(\Theta, m, C) = \tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e \end{array}}$$

$$\frac{\Theta(C) = C : B, I_1, \ldots, I_p\{\overrightarrow{F}, \overrightarrow{M}\} \quad m \text{ is not declared in any of } \overrightarrow{M}}{\begin{array}{c} mtype(\Theta, m, C) = mtype(\Theta, m, B) \\ mbody(\Theta, m, C) = mbody(\Theta, m, B) \end{array}}$$

$$\frac{\Theta(I) = I : \_\{\overrightarrow{MS}\} \quad \tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n) \in \overrightarrow{MS}}{mtype(\Theta, m, I) = (\tau_1, \ldots, \tau_n) \to \tau}$$

$$\frac{\Theta(I) = I : I_1, \ldots, I_n\{\_\} \quad \exists 1 \le j \le n\ mtype(\Theta, m, I_j) = (\tau_1, \ldots, \tau_n) \to \tau}{mtype(\Theta, m, I) = (\tau_1, \ldots, \tau_n) \to \tau}$$

Figure 20: Helper Functions for the Source Language Typing

Figure 21 defines well-formedness of methods, interface declarations, class declarations, and programs. Note that when type checking a method, we need to know the "this" pointer type. The methods declared in an interface should be distinct from those in the superinterfaces. Similarly, fields declared in a class should be distinct from those in the superclass. Method overriding requires method types be the same.

Figure 22 shows typing rules for heap values. There are only two kinds of heap values: objects and arrays. An object of C is required to specify all fields of C including those in the superclasses to simplify field fetch.

Figure 23 shows typing rules for expressions. All rules are straightforward.

Figure 24 shows the evaluation rules.

B.2 Translation from the Source to ECI

The source-ECI translation performs several tasks, besides lowering types and expressions. First, it collects members for each class and each interface, including those from superclasses or superinterfaces. Second, it lifts method definitions to global functions, after adding the "this" parameter. Therefore after translation, class declarations have only method signatures, not method bodies. Third, it creates itables and vtables for classes.

Type Translation Class and interface names are translated to existential types. Covariant array types are translated to existential types that hide the actual element types. The type translation is shown as follows:

$$\begin{array}{lcll} |\text{int}| & = & \text{int} & \\ |\tau| & = & \exists\alpha \ll \tau.\ \alpha & \tau = C \text{ or } I \\ |\text{array}(C)| & = & \exists\alpha \ll C.\{tag : \text{Tag}(\alpha), table : \text{array}(\exists\beta \ll \alpha.\ \beta)\} & \end{array}$$

Method Translation A method declaration is translated to a pair of a function label and a global function definition with an explicit "this". The translation of a method needs the enclosing class name to specify the "this" pointer's type. Suppose class C implements a method m. The translation generates a new label l_m and translates the method as follows:

$$\begin{array}{l} |\tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e, C| = \\ \qquad (l_m,\ \text{fix}\ m\langle\ \rangle(this : \exists\alpha \ll C.\ \alpha, x_1 : |\tau_1|, \ldots, x_n : |\tau_n|) : |\tau| = |e|) \end{array}$$

A method signature in the source language is translated to an ECI method declaration:

$$\frac{mbody(\Theta, m, C) = (\tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e) \quad \Theta; \bullet; this : C, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau}{\Theta \vdash (C, \tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e)}$$

$$\frac{\forall 1 \leq i \leq p, I_i \text{ declared before } I, \quad \Theta \vdash I_i, \text{ and } \forall 1 \leq q \leq k, msig_q \notin mclosure(I_i)}{\Theta \vdash I : I_1, \ldots, I_p\{msig_1, \ldots, msig_k\}}$$

$$\frac{\begin{array}{c}\Theta \vdash B : \_\{\_\} \qquad f_i \notin fields(\Theta, B)\ \forall 1 \leq i \leq n \qquad \Theta \vdash (C, mdecl_i)\ \forall 1 \leq i \leq k \\ \text{If } mdecl_i \text{ overrides a method } m \text{ in } B, \text{ then } mtype(\Theta, m, C) = mtype(\Theta, m, B) \\ \forall 1 \leq i \leq p, \Theta \vdash I_i : \_\{\_\}, \forall m \in mclosure(I_i), mtype(\Theta, m, C) = mtype(\Theta, m, I_i)\end{array}}{\Theta \vdash C : B, I_1, \ldots, I_p\{f_1 : \tau_1, \ldots, f_n : \tau_n, mdecl_1, \ldots, mdecl_k\}}$$

$$\frac{decl_1, \ldots, decl_n \vdash decl_i \ \ \forall 1 \leq i \leq n \qquad decl_1, \ldots, decl_n; \bullet; \bullet \vdash e : \tau}{\vdash decl_1, \ldots, decl_n \text{ in } e : \tau}$$

Figure 21: Well-formedness of Source Programs

$$\frac{fields(\Theta; C) = f_1 : \tau_1, \ldots, f_n : \tau_n \quad \Theta; \Sigma; \Gamma \vdash v_i : \tau_i\ \forall 1 \leq i \leq n}{\Theta; \Sigma \vdash C\{f_i = v_i\}_{i=1}^n : C} \qquad \frac{\Theta; \Sigma; \Gamma \vdash v_i : \tau\ \forall 0 \leq i \leq n-1}{\Theta; \Sigma \vdash [v_0, \ldots, v_{n-1}]^\tau : \text{array}(\tau)}$$

Figure 22: Heap Value Typing of the Source Language

$$|\tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n)| = m : (|\tau_1|, \ldots, |\tau_n|) \to |\tau|$$

Expression Translation Figure 25 shows the translation of expressions. Each new variable introduced during translation is unique. The translation of expression $new\ C\{f_i = e_i\}_{i=1}^n$ creates a record with $vtable_C$ (vtable of C) and fields $f_1, \ldots, f_n$ and then coerces the record to an object. Field fetch and method invocation first open the object and then coerce the opened object to a record.

The translation of interface method invocation "$imthd[I.m]e(e_1, \ldots e_n)$" uses the "$ILookup$" function defined in Section 2 to find the method table for I, and then coerces the abstract method table to a concrete one and fetches the method m.

The translation of array creation packs the tag of the element type along with the element array to the desired existential type. The translation of array store instructions inserts store checks: the object to be stored is first cast to the target element type.

The translation of the two cast expressions "$cast[C](e)$" and "$cast[I](e)$" calls the polymorphic functions "$Downcast$" and "$InterfaceCast$" with appropriate type arguments.

Interface Translation A source interface declaration is translated to an ECI interface declaration. The translation collects all super interfaces and all methods including those in the super interfaces. Function $iclosure$ computes all the superinterfaces of a class or an interface. Definitions of $iclosure$ on classes are in the explanation of class declaration translation. Function $mclosure$ computes all the methods in an interface. Duplicated elements are removed after set union. Interface declarations are translated as follows:

$$\begin{array}{l} iclosure(I : I_1, \ldots, I_n\{\_\}) = iclosure(I_1) \cup \ldots \cup iclosure(I_n) \cup \{I_1, \ldots, I_n\} \\ mclosure(I : I_1, \ldots, I_n\{msig_1, \ldots, msig_k\}) = \\ \qquad mclosure(I_1) \cup \ldots \cup mclosure(I_n) \cup \{msig_1, \ldots, msig_k\} \end{array}$$

$$\frac{}{\Theta;\Sigma;\Gamma \vdash n : \text{int}} \qquad \frac{}{\Theta;\Sigma;\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Theta;\Sigma;\Gamma \vdash \ell : \Sigma(\ell)} \qquad \frac{\Theta;\Sigma;\Gamma \vdash e : \tau \quad \Theta \vdash \tau \leq \tau'}{\Theta;\Sigma;\Gamma \vdash e : \tau'}$$

$$\frac{\Theta;\Sigma;\Gamma \vdash e_1 : \tau \quad \Theta;\Sigma;\Gamma, x : \tau \vdash e_2 : \tau'}{\Theta;\Sigma;\Gamma \vdash x : \tau = e_1 \text{ in } e_2 : \tau'} \qquad \frac{\Theta;\Sigma;\Gamma \vdash e_1 : \Gamma(x) \quad \Theta;\Sigma;\Gamma \vdash e_2 : \tau}{\Theta;\Sigma;\Gamma \vdash x := e_1 \text{ in } e_2 : \tau}$$

$$\frac{\begin{array}{c} fields(\Theta, C) = f_1 : \tau_1, \ldots, f_n : \tau_n \\ \Theta;\Sigma;\Gamma \vdash e_i : \tau_i \ \forall 1 \leq i \leq n \end{array}}{\Theta;\Sigma;\Gamma \vdash new\ C\{f_1 = e_1, \ldots, f_n = e_n\} : C} \qquad \frac{\begin{array}{c} \Theta;\Sigma;\Gamma \vdash e : C \\ fields(\Theta, C) = f_1 : \tau_1, \ldots, f_n : \tau_n \qquad 1 \leq i \leq n \end{array}}{\Theta;\Sigma;\Gamma \vdash e.f_i : \tau_i}$$

$$\frac{\Theta;\Sigma;\Gamma \vdash e_1 : C \quad fields(\Theta, C) = f_1 : \tau_1, \ldots, f_n : \tau_n \quad 1 \leq i \leq n \quad \Theta;\Sigma;\Gamma \vdash e_2 : \tau_i \quad \Theta;\Sigma;\Gamma \vdash e_3 : \tau}{\Theta;\Sigma;\Gamma \vdash e_1.f_i := e_2 \text{ in } e_3 : \tau}$$

$$\frac{\Theta;\Sigma;\Gamma \vdash e : C \quad mtype(\Theta, m, C) = (\tau_1, \ldots, \tau_n) \to \tau \quad \Theta;\Sigma;\Gamma \vdash e_i : \tau_i \ \forall 1 \leq i \leq n}{\Theta;\Sigma;\Gamma \vdash e.m(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Theta;\Sigma;\Gamma \vdash e_i : \tau \ \forall 0 \leq i \leq n-1}{\Theta;\Sigma;\Gamma \vdash new[e_0, \ldots, e_{n-1}]^{\tau} : \text{array}(\tau)} \qquad \frac{\Theta;\Sigma;\Gamma \vdash e_1 : \text{array}(\tau) \quad \Theta;\Sigma;\Gamma \vdash e_2 : \text{int}}{\Theta;\Sigma;\Gamma \vdash e_1[e_2] : \tau}$$

$$\frac{\begin{array}{c} \Theta;\Sigma;\Gamma \vdash e_1 : \text{array}(\tau) \quad \Theta;\Sigma;\Gamma \vdash e_2 : \text{int} \\ \Theta;\Sigma;\Gamma \vdash e_3 : \tau \quad \Theta;\Sigma;\Gamma \vdash e_4 : \tau' \end{array}}{\Theta;\Sigma;\Gamma \vdash e_1[e_2] := e_3 \text{ in } e_4 : \tau'} \qquad \frac{\begin{array}{c} \Theta;\Sigma;\Gamma \vdash e : I \\ mtype(\Theta, m, I) = (\tau_1, \ldots, \tau_n) \to \tau \\ \Theta;\Sigma;\Gamma \vdash e_i : \tau_i \ \forall 1 \leq i \leq n \end{array}}{\Theta;\Sigma;\Gamma \vdash imthd[I.m]e(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Theta;\Sigma;\Gamma \vdash e : \text{array}(\tau)}{\Theta;\Sigma;\Gamma \vdash \text{len}(e) : \text{int}} \qquad \frac{\Theta;\Sigma;\Gamma \vdash e : C \text{ or } \Theta;\Sigma;\Gamma \vdash e : I}{\Theta;\Sigma;\Gamma \vdash cast[\tau](e) : \tau} \qquad \frac{\Theta;\Sigma;\Gamma \vdash e : \tau_1 \quad \Theta \vdash \tau_1 \leq \tau_2}{\Theta;\Sigma;\Gamma \vdash e : \tau_2}$$

Figure 23: Expression Typing of the Source Language

Original State	New State	Side Conditions
$(H; V; x)$	$(H; V; V(x))$	
$(H; V; x : \tau = v \text{ in } e)$	$(H; V'; e)$	$V' = V, x = v$
$(H; V; x := v' \text{ in } e)$	$(H; V'; e)$	$V = V_1, x = v, V_2$ *and* $V = V_1, x = v', V_2$
$(H; V; new\ C\{f_i = v_i\}_{i=1}^n)$	$(H'; V; \ell)$	$H' = H, \ell \rightsquigarrow C\{f_i = v_i\}_{i=1}^n \quad \ell \notin \text{domain}(H)$
$(H; V; \ell.f_i)$	$(H; V; v_i)$	$H(\ell) = C\{f_1 = v_1, \ldots, f_n = v_n\} \quad 1 \le i \le n$
$(H; V; \ell.f_i := v_i' \text{ in } e)$	$(H'; V; e)$	$\forall \ell' \neq \ell \quad H'(\ell') = H(\ell')$ $H(\ell) = C\{f_j = v_j\}_{j=1}^n \quad 1 \le i \le n$ $H'(\ell) = C\{f_1 = v_1, \ldots, f_i = v_i', \ldots, f_n = v_n\}$
$(H; V; \ell.m(v_1, \ldots, v_p))$	$(H; V'; e)$	$H(\ell) = C\{f_i = \_\}_{i=1}^n$ $mbody(\Theta, m, C) = \tau\ m(x_1 : \tau_1, \ldots, x_p : \tau_p) = e$ $V' = V, x_1 = v_1, \ldots, x_p = v_p, this = \ell$
$(H; V; imthd[I.m]e(v_1, \ldots, v_p))$	$(H; V'; e)$	$H(\ell) = C\{f_i = \_\}_{i=1}^n$ $mbody(\Theta, m, C) = \tau\ m(x_1 : \tau_1, \ldots, x_p : \tau_p) = e$ $V' = V, x_1 = v_1, \ldots, x_p = v_p, this = \ell$
$(H; V; new[v_0, \ldots, v_{n-1}]^\tau)$	$(H'; V; \ell)$	$H' = H, \ell \rightsquigarrow [v_0, \ldots, v_{n-1}]^\tau \quad \ell \notin \text{domain}(H)$
$(H; V; \ell[i])$	$(H; V; v_i)$	$H(\ell) = [v_0, \ldots, v_{n-1}]^\tau \quad 0 \le i \le n-1$
$(H; V; \ell[i] := v_i' \text{ in } e)$	$(H'; V; e)$	$\forall \ell' \neq \ell \quad H'(\ell') = H(\ell')$ $H(\ell) = [v_0, \ldots, v_{n-1}]^\tau \quad 0 \le i \le n-1$ $H'(\ell) = [v_0, \ldots, v_{i-1}, v_i', v_{i+1}, \ldots, v_{n-1}]^\tau$
$(H; V; \text{len}(\ell))$	$(H; V; n)$	$H(\ell) = [v_0, \ldots, v_{n-1}]^\tau$
$(H; V; cast[\tau](\ell))$	$(H; V; \ell)$	$H(\ell) = C\{f_i = v_i\}_{i=1}^n$ *and* $\Theta \vdash C \le \tau$

Figure 24: Dynamic Semantics of the Source Language

$$
\begin{array}{lcl}
|n| & = & n \\
|x| & = & x \\
|x : \tau = e_1 \text{ in } e_2| & = & x : |\tau| = |e_1| \text{ in } |e_2| \\
|x := e_1 \text{ in } e_2| & = & x := |e_1| \text{ in } |e_2| \\
|new\ C\{f_i = e_i\}_{i=1}^n| & = & \text{pack } C \text{ as } \alpha \ll C \text{ in} \\
 & & (C(\text{new}[\mathsf{R}(C)]\{vtable = vtable_C, (f_i = |e_i|)_{i=1}^n\}) : \alpha) \\
|e.f_i| & = & (\alpha, x) = \text{open}(|e|) \text{ in } \text{c2r}(x).f_i \\
|e_1.f_i := e_2 \text{ in } e_3| & = & (\alpha, x) = \text{open}(|e_1|) \text{ in } \text{c2r}(x).f_i := |e_2| \text{ in } |e_3| \\
|e.m(e_1, \ldots, e_n)| & = & (\alpha, x) = \text{open}(|e|) \text{ in} \\
 & & \text{c2r}(x).vtable.m(\text{pack } \alpha \text{ as } \beta \ll \alpha \text{ in } (x : \beta), |e_1|, \ldots |e_n|) \\
|imthd[I.m]e(e_1, \ldots, e_n)| & = & (\alpha, x) = \text{open}(|e|) \text{ in} \\
 & & x' : \exists \beta \ll \alpha.\ \beta = \text{pack } \alpha \text{ as } \beta \ll \alpha \text{ in } (x : \beta) \text{ in} \\
 & & im2r(ILookup[I, \alpha](tag(I), x)).m(x', |e_1|, \ldots |e_n|) \\
|new[e_0, \ldots, e_{n-1}]^\tau| & = & \text{pack } \tau \text{ as } \alpha \ll \tau \text{ in} \\
 & & (\{tag = tag(\tau), table = \text{new}[|e_0|, \ldots, |e_{n-1}|]^{|\tau|}\} : \\
 & & \{tag : \text{Tag}(\alpha), table : \text{array}(\exists \beta \ll \alpha.\ \beta)\}) \\
|e_1[e_2]| & = & (\alpha, x) = \text{open}(|e_1|) \text{ in } x.table[|e_2|] \\
|e_1[e_2] := e_3 \text{ in } e_4| & = & (\alpha, x) = \text{open}(|e_1|) \text{ in } (\beta, y) = \text{open}(|e_3|) \text{ in} \\
 & & x.table[|e_2|] := Downcast[\alpha, \beta](x.tag, y) \text{ in } |e_4| \\
|\text{len}(e)| & = & (\alpha, x) = \text{open}(|e|) \text{ in } \text{len}(x) \\
|cast[C](e)| & = & (\alpha, x) = \text{open}(|e|) \text{ in } DownCast[C, \alpha](tag(C), x) \\
|cast[I](e)| & = & (\alpha, x) = \text{open}(|e|) \text{ in } InterfaceCast[I, \alpha](tag(I), x)
\end{array}
$$

Figure 25: Translation of Expressions

$$\frac{idecl = I : J_1, \ldots, J_n\{msig_1, \ldots, msig_k\} \quad mclosure(idecl) = msig'_1, \ldots, msig'_q \quad iclosure(idecl) = J_1, \ldots, J'_p \quad |msig'_i| = mdecl'_i \ \forall 1 \le i \le q}{|idecl| = I : J'_1, \ldots, J'_p\{mdecl'_1, \ldots, mdecl'_q\}}$$

Itable Entry Creation When the translation creates the itable for class C, it first creates an entry for each interface that C implements. Suppose C implements interface I, the translation creates two heap values for I: the first is a record labeled by l_{mtable} for the method table, and the second is a record labeled by l_{entry} consisting of the interface tag and the method table. The itable entry for the interface is coerced from the second record. We use $H_{(I,C)}$ to represent the two heap values, and $entry_{(I,C)}$ for the itable entry. They are defined as follows:

$$\frac{mclosure(I) = msig'_1, \ldots, msig'_q \quad msig'_i \text{ declares } m_i \ \forall 1 \le i \le q \quad mdecl'_i \text{ defines } m_i \text{ in } C \quad |mdecl'_i, C| = (l_{mi}, \_) \ \forall 1 \le i \le q}{\left(\begin{array}{l} H_{(I,C)} = \left(\begin{array}{l} l_{mtable} \rightsquigarrow \{m_1 = l_{m1}, \ldots, m_q = l_{mq}\}, \\ l_{entry} \rightsquigarrow \{tag = tag(I), mtable = r2im[I, C](l_{mtable})\} \end{array} \right) \\ entry_{(I,C)} = \text{pack } I \text{ as } \alpha \gg C \text{ in } (l_{entry} : \{tag : \text{Tag}(\alpha), mtable : \text{Imty}(\alpha, C)\}) \end{array} \right)}$$

Class Translation A class declaration in the source language is translated to a class declaration in ECI and a set of heap values. Suppose class C has a superclass B and B has been translated to a declaration in ECI with fields $fields_B$ and methods $methods_B$ and method implementations vms_B in the vtable. The method definitions in C $mdecl_1, \ldots, mdecl_k$ are translated to functions $mbody_1, \ldots, mbody_k$.

Methods for C ($methods_C$) consist of those in B ($methods_B$) and C's methods $m_1, \ldots, m_k$. Because of overriding, $m_1, \ldots, m_k$ may override methods in $methods_B$. We use operator $\oplus$ for concatenation with masking: if S is a sequence of bindings, then $S \oplus b$ appends b to S if S does not have the item bound in b. Otherwise, $S \oplus b$ replaces with b the corresponding entry in S. Auxiliary function $mtype(\Theta, m, C)$ returns the type of method m in class C.

To create vtables of C, we need to find all method implementations in C (vms_C). Again, we use $\oplus$ to concatenate implementations in B vms_B with method definitions in C.

The class declaration for C ($class$) includes all C's super interfaces ($iclosure(C)$), fields in B ($fields_B$) and its new fields $f_1, \ldots, f_n$, and $method_C$.

The translation of C's declaration builds a set of heap values H_C, which includes functions $mbody_1, \ldots, mbody_k$, heap values for the superinterfaces $I'_1, \ldots, I'_p$, the itable $itable_C$, and the vtable $vtable_C$. The itable has an entry for each of the superinterfaces. The vtable is a record of C's tag, the itable, and method implementations vms_C.

The following rule shows the class translation:

$$\frac{\begin{array}{c} iclosure(C : B, I_1, \ldots, I_n\{\ldots\}) = \\ iclosure(B) \cup iclosure(I_1) \cup \ldots \cup iclosure(I_n) \cup \{I_1, \ldots, I_n\} \end{array} \quad \begin{array}{c} |B| = (B : \_\{fields_B, methods_B\}, H_B) \\ H_B(vtable_B) = \{tag = \_, itable = \_, vms_B\} \\ mdecl_i \text{ defines } m_i \quad |mdecl_i, C| = (l_{mi}, mbody_i) \ \forall 1 \le i \le k \\ methods_C = methods_B \oplus m_1 : |mtype(\Theta, m_1, C)| \oplus \ldots \oplus m_k : |mtype(\Theta, m_k, C)| \\ vms_C = vms_B \oplus m_1 = l_{m1} \oplus \ldots \oplus m_k = l_{mk} \qquad iclosure(C) = I'_1, \ldots, I'_p \\ class = C : B, I'_1, \ldots, I'_p\{fields_B, f_1 : |\tau_1|, \ldots, f_n : |\tau_n|, methods_C\} \\ H_C = \left(\begin{array}{l} l_{m1} \rightsquigarrow mbody_1, \ldots, l_{mk} \rightsquigarrow mbody_k, H_{(I'_1,C)}, \ldots, H_{(I'_p,C)}, \\ itable_C \rightsquigarrow [entry_{(I'_1,C)}, \ldots, entry_{(I'_p,C)}]^{\exists \alpha \gg C. \ \{tag:\text{Tag}(\alpha), mtable:\text{Imty}(\alpha,C)\}}, \\ vtable_C \rightsquigarrow \{tag = tag(C), itable = itable_C, vms_C\} \end{array} \right) \end{array}}{|C : B, \_\{f_1 : \tau_1, \ldots, f_n : \tau_n, mdecl_1, \ldots, mdecl_k\}| = (class, H_C)}$$

Program Translation A program is translated to a set of new declarations, an initial heap (which contains function translated from method definitions, itables, and vtables), and a new main expression.

$$\frac{\Theta = cdecl_1, \ldots, cdecl_m, idecl_1, \ldots, idecl_n \quad |cdecl_i| = (newcdecl_i, H_i)\ \forall 1 \le i \le m \quad |idecl_i| = newidecl_i\ \forall 1 \le i \le n \quad H_0 = H_1, \ldots, H_m}{|\Theta \text{ in } e| = (newcdecl_1, \ldots, newcdecl_m, newidecl_1, \ldots, newidecl_n; H_0; \bullet; |e|)}$$

Next we prove that the translation from the source language to ECI preserves types.

Lemma 35 *If τ_1 and τ_2 are class or interface names and $\Theta \vdash \tau_1 \le \tau_2$ in the source language, then $\Theta; \bullet \vdash \tau_1 \ll \tau_2$ and $\Theta; \bullet \vdash |\tau_1| \le |\tau_2|$ in ECI.*

Proof: by induction on source language subtyping rules.

In the rest of the section, we assume a source language program $P = \Theta$ in e, and $|P| = (|\Theta|; H_0; \bullet; |e|)$, and $|\Theta| \vdash H_0 : \Sigma_0$ in ECI.

Lemma 36

- *If $fields(\Theta, C) = f_1 : \tau_1, \ldots, f_n : \tau_n$ in the source language, then $|\Theta|(C) = \_\{f_1 : |\tau_1|, \ldots, f_n : |\tau_n|, mdecls\}$.*
- *If $mtype(\Theta, m, C) = (\tau_1, \ldots, \tau_n) \to \tau$ in the source language, then $|\Theta|(C) = \_\{\ldots, m : (|\tau_1|, \ldots, |\tau_n|) \to |\tau|, \ldots\}$.*
- *If $mtype(\Theta, m, I) = (\tau_1, \ldots, \tau_n) \to \tau$ in the source language, then $|\Theta|(I) = \{\ldots, m : (|\tau_1|, \ldots, |\tau_n|) \to |\tau|, \ldots\}$.*
- *$\forall$ class C in Θ, if $\mathsf{R}(C) = \{\!\{vtable : vttype_C, \ldots\}\!\}$, then $\Sigma_0(vtable_C) = vttype_C$.*

Proof: the first three are proven by induction on the definition of $fields(\Theta, C)$, $mtype(\Theta, m, C)$ and $mtype(\Theta, m, I)$ respectively. The last one is proven by induction on class declaration translation rules and by that method translation preserves types.

Lemma 37 *The expression translation from the source language to ECI preserves types. If $\Theta; \Sigma; \Gamma \vdash E : T$ in the source language, then $|\Theta|; \bullet; \Sigma_0, |\Sigma|; |\Gamma| \vdash |E| : |T|$ in ECI, where $|\ell_1 : \tau_1, \ldots, \ell_n : \tau_n| = \ell_1 : |\tau_1|, \ldots, \ell_n : |\tau_n|$, and $|x_1 : \tau_1, \ldots, x_m : \tau_m| = x_1 :_{\mathrm{M}} |\tau_1|, \ldots, x_m :_{\mathrm{M}} |\tau_m|$.*

Proof: by induction over the expression typing rules in the source language.

C Extensions

ECI may be extended to support other interface implementation techniques.

CACAO used a table of itable pointers in the vtable [14]. Each interface has been assigned a unique offset in the table and thus can be accessed directly without runtime lookup. The table for type τ can be represented as a record type in ECI $\{I_1 : \mathrm{Imty}(I_1, \tau), \ldots, I_n : \mathrm{Imty}(I_n, \tau)\}$.

The inline cache-based strategies cache one or more most recently invoked methods [12] and can be applied to interface method invocation. Method invocation is replaced with a stub that first compares the target method with the cached ones. If there is a match, the stub invokes the cached method. Otherwise, it resorts to the slow search. ECI may be extended with method tags, similar to class or interface tags, to model the method identities, and method tag comparison to refine types of methods.

Selector-based strategies assign unique ids (selectors) to identify method signatures [7, 11]. Each class contains a table of method pointers indexed by selectors. The table could be modeled by a record type in ECI, each field representing an interface method. Driesen used sparse arrays to reduce the space overhead for selector-indexed tables [9]. Unused space in the table of one class may contain method pointers for other classes. The type system of ECI could ignore the method pointers for other classes by using record subtyping to view the table as containing only methods for the current class. Graph-coloring [8] can assign the same selector to two different method signatures if the two methods are never implemented by the same class. To model this, ECI need to assign different types to the selector, depending on the class the selector is for.

Alpern *et al.* improved selector coloring by allowing color collisions [1]. If a collision occurs, a stub compares the target interface signature with those that share the same color to find out the right method pointer. This strategy could be modeled in ECI similarly to the inline cache-based one, by selector comparison.