A Simple Typed Intermediate Language for Object-Oriented Languages

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

David Tarditi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
dtarditi@microsoft.com

ABSTRACT

Traditional class and object encodings are difficult to use in practical type-preserving compilers because of the complexity of the encodings. We propose a simple typed intermediate language for compiling object-oriented languages and prove its soundness. The key ideas are to preserve lightweight notions of classes and objects instead of compiling them away and to separate name-based subclassing from structure-based subtyping. The language can express standard implementation techniques for both dynamic dispatch and runtime type tests. It has decidable type checking even with subtyping between quantified types with different bounds. Because of its simplicity, the language is a more suitable starting point for a practical type-preserving compiler than traditional encoding techniques.

Categories and Subject Descriptors

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

General Terms

Languages

Keywords

Typed intermediate language, class and object encoding

1. INTRODUCTION

Preserving types during compilation has significant benefits [33, 32, 25, 14]. Types can be used to debug compilers, to guide optimizations, and to generate safety proofs for programs [26]. In practice, however, compilers for object-oriented languages do not use statically typed low-level intermediate languages, even though their input is statically typed.

POPL'05, January 12–14, 2005, Long Beach, California, USA.

One reason compilers for object-oriented languages have not adopted type-preserving compilation is the complexity of traditional class and object encodings. A practical compiler requires simple, general and efficient type systems. First, compiler writers who are not type theorists should be able to understand the type system. Second, the type system needs to cover a large set of realistic language features and compiler transformations. Third, the type system needs to express standard implementation techniques without introducing extra runtime overhead. Traditional encodings are not a good match for these goals. We discuss these encodings more in Section 7.

This paper describes a simple typed intermediate language LIL_C (Low-level Intermediate Language with Classes) for compiling class-based object-oriented languages. LIL_C is lower level than JVML [25] or CIL [14]. The key ideas are to preserve lightweight notions of classes and objects instead of compiling them away and to separate name-based subclassing from structure-based subtyping.

LIL_C divides types into two parts: one part uses class names to type objects and keeps the name-based class hierarchy; the other part uses record types and has structural subtyping. Each class has a corresponding record type that represents its object layout. Objects and records can be coerced to each other with no runtime overhead. Keeping classes and objects has a low cost because most interesting work, such as field fetching, method invocation and cast, is done on records.

Our approach simplifies the type system. First, structural recursive types are not necessary because each record type can refer to any class name, including the class to which the record type corresponds. Second, it simplifies the bounded quantification that is needed to express inheritance. The bounds for type variables are in terms of subclassing, not subtyping as in traditional bounded quantification. Thus, the bounds must be classes or type variables, not arbitrary types. As a result, LIL_C has decidable type checking.

The contributions of this work include:

- LIL_C is simpler and more natural than traditional encodings. It is also sound and efficient.
- LIL_C can express standard implementation strategies for self application, dynamic type dispatch, and runtime type tests.

  LIL_C uses existential types and invariant array types to describe covariant source-level array types. It can also express runtime "store checks".

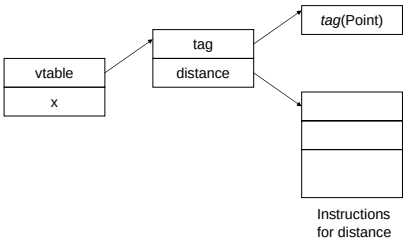

Figure 1: Object Layout for Point

> By supporting the decomposition of runtime primitives such as type cast and runtime store checks, LIL_C makes the interface between the runtime system and compiled code more explicit and checkable.

The rest of the paper is organized as follows. Section 2 gives an informal overview of LIL_C. The next two sections explain the syntax and semantics of LIL_C. Section 5 describes a source language and the translation from the source language to LIL_C. Section 6 presents some extensions. Section 7 discusses related work and Section 8 concludes.

2. OVERVIEW

This section describes informally how LIL_C addresses the implementation of object-oriented features such as object layout, inheritance, dynamic dispatch and runtime type tests. Roughly speaking, LIL_C supports features in Featherweight Java [21], assignments and arrays. For clarity, we make some simplifications. First, we omit non-virtual methods. The compiler can transform those methods to global functions (with explicit "this" pointers) because it knows all call sites of those methods. Second, we make each class contain declarations of all its members, including the inherited ones from its super classes. Third, we rename variables so that they are all unique. Finally, we omit null references and consider only proper objects.

2.1 Object Layout

LIL_C has both class names and record types. We use B, C and D to range over class names. Each class name C has a corresponding record type $R(C)$ that describes its object layout—the organization of fields, methods and runtime tags. R is not a constructor in LIL_C, but a "macro" used by the type checker (Section 4.1.1).

We use a standard object layout. An object of C contains a vtable of C and fields. Therefore, $R(C)$ is a record type containing entries for C's vtable and fields. The vtable is a record that includes a runtime tag for the class and function pointers for the virtual methods. The runtime tag contains class-specific information and is used to identify the class.

As shown in Section 4.1.1, it is easy to substitute other layout strategies for the current one.

The type of a field or a method in $R(C)$ can refer to any class name declared in the program, including C. This allows a straightforward typing of self-application semantics [22], where each method takes the "this" pointer as its first parameter. LIL_C assigns the type $\exists\alpha \ll C.\ \alpha$, which represents objects of class C or C's subclasses, to the "this" pointer of methods in C. We explain the type in Section 2.2.

We use the following example throughout the paper:

$$\begin{array}{l}\text{class Point } \{\\ \quad \text{int } x;\\ \quad \text{int } \mathit{distance}()\{\ldots\}\}\end{array}$$

The object layout of Point is shown in Figure 1. Type $R(\text{Point})$ represents this layout naturally:

$$\begin{array}{l}R(\text{Point}) =\\ \quad \{\mathit{vtable} : \{\mathit{tag} : \text{Tag(Point)},\\ \qquad\qquad \mathit{distance} : (\exists\alpha \ll \text{Point. } \alpha) \to \text{int}\},\\ \quad x : \text{int}\}\end{array}$$

The tag of a class C identifies C at run time. Its type is represented as $\text{Tag}(C)$, where Tag is an abstract type constructor. For simplicity, we treat tags as abstract.

An object of C can be coerced to a record of type $R(C)$ and *vice versa*. The coercions are no-ops at run time and introduce no overhead.

Object Creation To create an object of class C, we create a record of $R(C)$ and then coerce the record to an object.

Field Fetch To fetch a field from an object, we coerce the object to a record and fetch the field from the record.

Method Invocation To call a method on an object o of class C, we coerce o to a record, fetch the vtable from the record, then fetch the method from the vtable, and pass o (after packing it to have type $\exists\alpha \ll C.\ \alpha$) to the method. This is exactly how most compilers implement virtual method invocation, if we ignore types and coercions, which turn into no-ops at run time anyway.

$R(C)$ specifies the full shape of an object of C, including the fields and methods from C's super classes. The inherited members must have the same order in C as in C's super classes, and appear before the members specific to C. Suppose class Point2D extends class Point as follows:

$$\begin{array}{l}\text{class Point2D : Point}\{\\ \quad \text{int } y;\\ \quad \text{int } \mathit{distance}()\{\ldots y \ldots\}\}\end{array}$$

$R(\text{Point2D})$ will be the following type :

$$\begin{array}{l}R(\text{Point2D}) =\\ \quad \{\mathit{vtable} : \{\mathit{tag} : Tag(\text{Point2D}),\\ \qquad\qquad \mathit{distance} : (\exists\alpha \ll \text{Point2D. } \alpha) \to \text{int}\},\\ \quad x : \text{int}, y : \text{int}\}\end{array}$$

$R(\text{Point2D})$ includes members in Point, but it has its own tag and its own type for the "this" pointer.

2.2 Subclassing and Bounded Quantification

LIL_C defines a subclassing relation "$\ll$" between class names to preserve the name-based class hierarchy in the source program. If a class C extends class B, then $C \ll B$. For example, Point2D $\ll$ Point. The subclassing relation is reflexive and transitive.

In most source languages, a class name C describes objects of C and C's subclasses. LIL_C translates a source-level class name C to an existential type with subclassing-based quantification: $\exists\alpha \ll C.\ \alpha$, where α indicates the "actual" class. Objects that have type C or one of C's subclasses can be packed to have type $\exists\alpha \ll C.\ \alpha$. A class name in LIL_C represents only objects of itself.

Similarly, type variables introduced by universal types can have bounds too. Type $\forall\alpha \ll C.\ \tau$ describes terms (polymorphic functions) that work on C and any subclass of C.

The bound of a type variable must be a class name or another type variable, not an arbitrary type. The bound never refers to the type variable it constrains[1].

2.3 Dynamic Dispatch

Dynamic dispatch requires a distinction between the static type and the dynamic type of an object. LIL$_C$ uses class names for "exact" types (dynamic types) and existential types for "imprecise" types (static types).

By having "exact" notions, LIL$_C$ excludes unsafe dynamic dispatch. Consider the following function:

$$\begin{array}{l}\text{void Test(Point } p_1\text{, Point } p_2)\{\\ \quad vt = p_1.vtable;\\ \quad dist = vt.distance;\\ \quad dist(p_2);\}\end{array}$$

This function is unsafe, even though the distance method fetched from p_1 requires an object of Point and p_2 is indeed an object of Point. The function Test can be called in an unsafe way:

$$\begin{array}{l}\text{Point } point_1 = new \text{ Point2D}(\ldots);\\ \text{Point } point_2 = new \text{ Point}(\ldots);\\ \text{Test}(point_1, point_2);\end{array}$$

The argument $point_1$ is an object of Point2D. The distance method fetched from $point_1$'s vtable accesses the field y, which $point_2$ does not have. A language *without* a way to express "exact" dynamic types cannot catch such errors.

In LIL$_C$, if an object has type τ and its vtable contains a method m, then only objects of type $\exists\alpha \ll \tau.\ \alpha$ can be passed to the "this" pointer of m. In this representation, Test is translated to an ill-typed function:

$$\begin{array}{l}\text{void Test}(p_1 : \exists\alpha \ll \text{Point. } \alpha,\ p_2 : \exists\beta \ll \text{Point. } \beta)\{\\ \quad (\alpha, p_1') = \text{open}(p_1);\\ \quad p_1'' = \text{toRecord}(p_1');\\ \quad vt = p_1''.vtable;\\ \quad dist = vt.distance;\\ \quad dist(p_2); \text{//}\textbf{ill-typed!}\ \}\end{array}$$

p_1' has type α and $dist$ requires an object of type $\exists\delta \ll \alpha.\ \delta$. Object p_2 does not have this type.

2.4 Subtyping

LIL$_C$ also has structural subtyping, represented by "$\leq$". The subtyping relation is reflexive and transitive.

Record types have standard prefix subtyping and depth subtyping on immutable fields. By prefix subtyping, appending more fields to a record type creates a subtype. The super type is a prefix of the subtype. Specializing immutable field types leads to depth subtyping. LIL$_C$ excludes depth subtyping on mutable fields to preserve soundness. The label order in a record type is significant because the fields represent the physical layout of data.

LIL$_C$ uses record subtyping to approximate the layout of an object whose "exact" type is unknown at compile time. A detailed explanation is in Section 4.1.1.

Bounded quantified types have subtyping. The most frequently used rule is $(\exists\alpha \ll C.\ \alpha) \leq (\exists\alpha \ll B.\ \alpha)$ if $C \ll B$. It is used for:

[1]LIL$_C$ has a much simpler solution to binary methods than F-bounded quantification [5]. See Section 6.2.

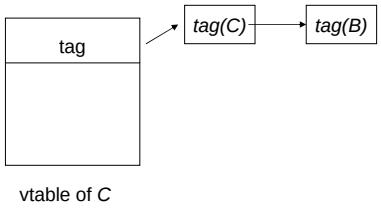

Figure 2: Tag Chains

Inheritance Subsumption If $C \ll B$ and an object o has type $\exists\alpha \ll C.\ \alpha$, then o can be used wherever an object of class B or B's subclasses (type $\exists\alpha \ll B.\ \alpha$) is expected.

Inherited Method Implementation A subclass can inherit a method implementation from its super classes. Suppose class C is a subclass of B. The "this" pointer of methods in C has type $\exists\alpha \ll C.\ \alpha$. The "this" pointer of methods in B has type $\exists\alpha \ll B.\ \alpha$. Because $(\exists\alpha \ll C.\ \alpha) \leq (\exists\alpha \ll B.\ \alpha)$, a function that takes a parameter of type $\exists\alpha \ll B.\ \alpha$ can be used as one with a parameter of type $\exists\alpha \ll C.\ \alpha$, that is, C can use B's method implementation.

Subclassing is distinct from subtyping. If $C \ll B$ and C and B are different classes, then C is *not* a subtype of B, and neither is $R(C)$ a subtype of $R(B)$, because C represents objects of exact C. An object of exact C cannot be used where an object of exact B is needed.

2.5 Type Cast

It is difficult to model the implementation of downward type casts in a typed language. Most typed intermediate languages treat downcast as a primitive.

Downward type casts check at run time whether an arbitrary object belongs to some class or its subclasses. In a typical implementation, each class stores a tag in its vtable. If C extends B, then the tag of C has a pointer pointing to the tag of B (Figure 2). The pointers form a tag chain. Downward type casts fetch tags in the chain and compare them with the tag of the target class.

This implementation can be expressed as a well-typed function in LIL$_C$ that can be used for arbitrary objects and arbitrary classes. The key ideas are to use types to connect an object with the tag it contains, and to refine types according to the tag comparison result.

The tag of class C is represented as a constant $tag(C)$, which has type Tag(C). If an object has type τ, then the tag it contains has type Tag(τ). The type checker checks the connection at object creation sites.

Tag comparison is based on that two classes are the same iff their tags are equal. If an object o has type τ and the tag in o is equal to $tag(C)$, then $\tau = C$. If one of the parent tag, which identifies a parent class of τ, is equal to $tag(C)$, then $\tau \ll C$ and o can be packed to have type $\exists\delta \ll C.\ \delta$.

Figure 3 shows a global polymorphic function that implements downcast. More details of the function are in Section 4.1.3. Primitive "ifEqTag" compares tags and "ifParent" tests whether there exists a parent tag. Each of them corresponds to only two x86 instructions—compare and branch. We could further break down the primitives by exposing tag details. The details are irrelevant to our basic ideas, so we keep tags abstract for simplicity.

A companion technical report explains how to express optimizations of the above linear search, such as caching the

$$
\begin{array}{l}
fix\ Downcast\langle\alpha,\beta\rangle(t_\alpha : \mathrm{Tag}(\alpha), obj : \beta) : \exists\delta \ll \alpha.\ \delta = \quad //t_\alpha : \text{tag of target class } \alpha, obj : \text{object to cast} \\
\quad loop[\alpha,\beta,\beta](t_\alpha, obj, (\mathrm{c2r}(obj).vtable).tag) \\
fix\ loop\langle\alpha,\gamma,\beta \ll \gamma\rangle(t_\alpha : \mathrm{Tag}(\alpha), obj : \beta, t_\gamma : \mathrm{Tag}(\gamma)) : \exists\delta \ll \alpha.\ \delta = \quad //t_\gamma : \text{a tag in the tag chain} \\
\quad \mathrm{ifEqTag}^{\exists\delta \ll \alpha.\ \delta}(t_\gamma, t_\alpha) \text{ then pack } \beta \text{ as } \delta \ll \gamma \text{ in } (obj : \delta) \text{ else} \quad //\gamma = \alpha \\
\quad\quad \mathrm{ifParent}(t_\gamma) \text{ then bind } (\gamma', t'_\gamma) \text{ in } loop[\alpha,\gamma',\beta](t_\alpha, obj, t'_\gamma) \text{ else error}[\exists\delta \ll \alpha.\ \delta] \quad //\gamma \ll \gamma'
\end{array}
$$

Figure 3: Cast Method

most recently successful parent tag, or storing a display of super classes [10]. It also explains array casting.

2.6 Arrays

Covariant array types require runtime "store checks" each time an object is stored into an array. If array a has type array(B), to store an object of type B in a, we have to check whether the object has the "actual" element type of a because a might be an array of B's subclasses.

LIL$_C$ uses existential types and invariant array types to express source-level array types. LIL$_C$ type $ARRAY(B) = \exists\alpha \ll B.\ \{tag : \mathrm{Tag}(\alpha), table : \mathrm{array}(\exists\beta \ll \alpha.\ \beta)\}$ represents source type array(B). The outer existential type binds the "actual" element type of the array, and the inner one binds the "actual" type of each element. The tag indicates the array element type. The source-level array subtyping is transferred to subtyping on existential types in LIL$_C$. If C is a subclass of B, then $ARRAY(C) \leq ARRAY(B)$.

To store an object in an array, LIL$_C$ programs must explicitly check whether the object matches the element type of the array. The cast routine in Figure 3 will suffice. The checks are inserted automatically by the translation from source programs, and can be eliminated if the type system can prove that the elimination is safe.

LIL$_C$ relies on runtime checks for array-bounds checking. For simplicity, it does not reason about integer constraints.

3. SYNTAX OF LIL$_C$

LIL$_C$ extends the polymorphic lambda calculus with class names, subclassing, subclassing-based quantification and imperative features. Figures 4, 5 and 6 show the syntax. The notation $\bullet$ represents empty lists.

3.1 Kinds and Types

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

Figure 4: Kinds and Types

LIL$_C$ uses a special kind Ω_c to classify class names: only class names and type variables that will be instantiated with class names have kind Ω_c. Topc is a special class name that is a super class of all classes, like *System.Object* in Microsoft CLI. Kind Ω_c guarantees that certain operations, such as "$\ll$" and "Tag", are applied properly. Kind Ω is used for other types. Kind Ω_c is a subkind of Ω.

Standard types in LIL$_C$ include int, type variables, array types, function types and record types $\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}$. The superscript annotations I or M on record fields indicate immutable or mutable fields respectively. For convenience, I is often omitted.

Non-standard types are class names, tag types, bounded quantified types and exact record types. Section 2 explained the first three types already.

Exact record types are denoted by "$\{\!\{$" and "$\}\!\}$". Sometimes the type checker needs to know the full shape of a record—all fields in the record. For example, when a record is coerced to an object, the record must have and only have the elements specified in the class declaration. Extra fields are undesirable. The normal record type is imprecise because of prefix subtyping. A record of an exact record type $\{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}\!\}$ has and only has fields labeled as $l_1, \ldots, l_n$. These fields have types $\tau_1, \ldots, \tau_n$ respectively.

Type $R(C)$, which describes the object layout of C, is an exact record type. The vtable in $R(C)$ also has an exact record type to exclude extra fields (extra methods).

To simplify the type system, LIL$_C$ has the subtyping rule $\{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}\!\} \leq \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}$.

Type $\forall\alpha_1 \ll \tau_1 \ldots . \forall\alpha_n \ll \tau_n.\tau$ is often abbreviated as $\forall\alpha_1 \ll \tau_1, \ldots, \alpha_n \ll \tau_n.\tau$. Existential types have similar abbreviation. The upper bound Topc is often omitted.

3.2 Values and Expressions

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

Figure 5: Values and Expressions

LIL$_C$ has the following word-size values: integer literal n, label ℓ (a pointer to a value on the heap), $C(v)$ representing an object of C coerced from a record labeled by v, runtime tag $tag(C)$ for class C and packed word-size values.

Large values—records, arrays and functions—are allocated on the heap. Function "fix $g\langle tvs\rangle(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e_m$" declares a function g with type variables tvs, formals $x_1 : \tau_1, \ldots, x_n : \tau_n$, return type τ and function body e_m. The body e_m can call g recursively.

Most expressions in LIL$_C$ are standard. The notation ":=" stands for assignment and "$e[e']$" stands for array element access. Expression "new$[\tau]\{l_1 = e_1, \ldots, l_n = e_n\}$" creates a record of type τ. Expression "new$[e_0, \ldots, e_{n-1}]^\tau$" creates an array of τ. The let expression "$x : \tau = e$ in e'" specifies the type of the new variable x. The pack expression "pack τ as $\alpha \ll \tau_u$ in $(e : \tau')$" hides type τ with a type variable α bounded by τ_u.

Expression "$C(e)$" coerces a record e to an object of C, and "c2r(e)" coerces an object e to a record.

Expression "ifParent(e) then bind (α, x) in e_1 else e_2" tests whether tag e has a parent tag. If so, a new value variable x is introduced to represent the parent tag and a new type variable α is bound to the type x indicates, and the control transfers to e_1. Otherwise, the control transfers to e_2.

Expression "ifEqTag$^\tau(e_{t1}, e_{t2})$ then e_1 else e_2" is a conditional branch for tag comparison. If two tags e_{t1} and e_{t2} are the same, e_1 is evaluated. Otherwise e_2 is evaluated. The expression specifies the result type τ for the whole expression to simplify type checking.

Expression "error$[\tau]$" represents runtime errors, such as cast failures. Type τ means that a value of type τ is expected if no errors happen. Future versions of LIL$_C$ will replace error expressions with exception handling.

3.3 Classes and Programs

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

Figure 6: Classes and Programs

A field declaration "$f : \tau$" declares a field f of type τ. Type τ contains no free type variables. A method declaration "$m : \forall\ \mathit{tvs}\ (\tau_1, \ldots, \tau_n) \to \tau$" declares a method m with type variables tvs, formal types $\tau_1, \ldots, \tau_n$ and return type τ. Types $\tau_1, \ldots, \tau_n$ and τ can contain only type variables in tvs. Method declarations do not have explicit "this" types.

A class declaration has a class name, a parent class, a collection of field declarations and method declarations ($\overrightarrow{\rho}$ means a list of items in ρ). Class declarations contain only the signatures of their methods, not any implementations. Method bodies are translated to functions on the heap, and they are called through function pointers inside vtables.

Finally a program has a set of class declarations, a heap H that maps labels to heap values, a set of variable-value bindings V, and a main expression. Method implementations and vtables are static data on the heap and are generated during the translation from the source language.

4. LIL$_C$ SEMANTICS

4.1 Static Semantics

LIL$_C$ maintains a class declaration table Θ that maps class names to declarations. The class declaration part of a program can serve as such a table.

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 or another type variable introduced previously in Δ. The typing rule for expression "ifParent" introduces type variables with lower bounds (Figure 10).

A heap environment Σ maps labels to types. A type environment Γ maps variables to types. An entry $x :_M \tau$ in Γ means that variable x is mutable. Substitution τ/α means replacing α with τ.

4.1.1 Helper Functions

We first explain three important helper functions that the type checker uses.

The first function $addThis(\tau_{this}, \tau)$ adds τ_{this}, the type for the "this" pointer, to a function type τ.

$$
\begin{array}{l}
addThis(\tau_{this}, \forall\ tvs\ (\tau_1, \ldots, \tau_n) \to \tau) \\
= \forall\ tvs(\tau_{this}, \tau_1, \ldots, \tau_n) \to \tau
\end{array}
$$

Suppose a class C has fields $f_1 : s_1, \ldots, f_n : s_n$ and methods $m_1 : t_1, \ldots, m_k : t_k$. Function $R(C)$ returns an exact record type that represents the object layout of C. Function Approx$R(\alpha, C)$ returns a normal record type that approximates $R(\alpha)$ for some type variable $\alpha \ll C$.

$$
\begin{array}{l}
R(C) = \{\!\{vtable : \{\!\{tag : \mathrm{Tag}(C), \\
\qquad\qquad m_1 : addThis(\exists\gamma \ll C.\ \gamma, t_1), \ldots, \\
\qquad\qquad m_k : addThis(\exists\gamma \ll C.\ \gamma, t_k)\}\!\}, \\
\qquad f_1^M : s_1, \ldots, f_n^M : s_n\}\!\}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{Approx}R(\alpha, C) = \{vtable : \{tag : \mathrm{Tag}(\alpha), \\
\qquad\qquad\qquad\qquad m_1 : addThis(\exists\gamma \ll \alpha.\ \gamma, t_1), \\
\qquad\qquad\qquad\qquad \ldots, \\
\qquad\qquad\qquad\qquad m_k : addThis(\exists\gamma \ll \alpha.\ \gamma, t_k)\}, \\
\qquad\qquad\qquad f_1^M : s_1, \ldots, f_n^M : s_n\} \\
\mathrm{Approx}R(\alpha, \mathrm{Topc}) = \{vtable : \{tag : \mathrm{Tag}(\alpha)\}\}
\end{array}
$$

In Approx$R(\alpha, C)$, the tag has type Tag(α) and the "this" pointer has type $\exists\gamma \ll \alpha.\ \gamma$. Approx$R(\alpha, \mathrm{Topc})$ contains only a vtable and the vtable contains only the tag of α.

Structural record subtyping guarantees that the approximation of Approx$R$ is valid. Suppose an object o has type $\exists\alpha \ll \mathrm{Point}.\ \alpha$. It might be an object of Point or any subclass of Point. The layout of o can be approximated by:

$$
\begin{array}{l}
\mathrm{Approx}R(\alpha, \mathrm{Point}) = \\
\quad \{vtable : \{tag : \mathrm{Tag}(\alpha), \\
\qquad\qquad distance : (\exists\gamma \ll \alpha.\ \gamma) \to \mathrm{int}\}, \\
\quad x^M : \mathrm{int}\ \}
\end{array}
$$

If o is actually an object of Point2D, its layout is:

$$
\begin{array}{l}
R(\mathrm{Point2D}) = \\
\quad \{\!\{vtable : \{\!\{tag : Tag(\mathrm{Point2D}), \\
\qquad\qquad distance : (\exists\gamma \ll \mathrm{Point2D}.\ \gamma) \to \mathrm{int}\}\!\}, \\
\quad x^M : \mathrm{int}, y^M : \mathrm{int}\}\!\}
\end{array}
$$

Structural record subtyping in LIL$_C$ guarantees that $R(\mathrm{Point2D}) \le \mathrm{Approx}R(\alpha, \mathrm{Point})[\mathrm{Point2D}/\alpha]$ holds.

The two helper functions $R(C)$ and Approx$R(\alpha, C)$ need full knowledge of what layout the compiler chooses for objects. Therefore, the layout information is part of the type system. However, only three typing rules use the helper functions. The rest of the type system is independent of the layout strategy. The soundness of the type system only requires that: (1) Approx$R(\alpha, C) \le$ Approx$R(\alpha, B)$ if $C \ll B$; (2) $R(C) \le$ Approx$R(\alpha, B)[C/\alpha]$ if $C \ll B$ (Section C.3 in

$$\frac{C \in \text{domain}(\Theta)}{\Theta; \bullet \vdash C : \Omega_c} \qquad \frac{\Theta; \Delta \vdash \tau : \Omega_c}{\Theta; \Delta \vdash \text{Tag}(\tau) : \Omega}$$

$$\frac{}{\Theta; \bullet \vdash \text{Topc} : \Omega_c} \qquad \frac{\alpha \ll \tau \in \Delta \text{ or } \alpha \gg \tau \in \Delta}{\Theta; \Delta \vdash \alpha : \Omega_c}$$

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

Figure 7: Selected Kinding Rules

$$\frac{\Theta(C) = C : B\{\ldots\}}{\Theta; \Delta \vdash C \ll B} \qquad \frac{\Theta; \Delta \vdash \tau : \Omega_c}{\Theta; \Delta \vdash \tau \ll \text{Topc}} \qquad \frac{\Theta; \Delta \vdash \tau : \Omega_c}{\Theta; \Delta \vdash \tau \ll \tau}$$

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

Figure 8: Subclassing Rules

$$\frac{m \geq n}{\Theta; \Delta \vdash \{l_i^{\phi_i} : \tau_i\}_{i=1}^m \leq \{l_i^{\phi_i} : \tau_i\}_{i=1}^n}$$

$$\frac{\forall 1 \leq i \leq m, \begin{cases} \Theta; \Delta \vdash \tau_i \leq \tau_i' & \text{if } \phi_i = I \\ \tau_i = \tau_i' & \text{if } \phi_i = M \end{cases}}{\Theta; \Delta \vdash \{l_i^{\phi_i} : \tau_i\}_{i=1}^m \leq \{l_i^{\phi_i} : \tau_i'\}_{i=1}^m}$$

$$\frac{}{\Theta; \Delta \vdash \{\!\{l_i^{\phi_i} : \tau_i\}\!\}_{i=1}^m \leq \{l_i^{\phi_i} : \tau_i\}_{i=1}^m}$$

$$\frac{\Theta; \Delta \vdash t_i \leq s_i\ \forall 1 \leq i \leq n \quad \Theta; \Delta \vdash s \leq t}{\Theta; \Delta \vdash (s_1, \ldots, s_n) \to s \leq (t_1, \ldots, t_n) \to t}$$

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

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

$$\frac{}{\Theta; \Delta \vdash \tau \leq \tau} \qquad \frac{\Theta; \Delta \vdash \tau_1 \leq \tau_2 \quad \Theta; \Delta \vdash \tau_2 \leq \tau_3}{\Theta; \Delta \vdash \tau_1 \leq \tau_3}$$

Figure 9: Subtyping Rules

the appendix of the technical report). The type system is parameterized by the compiler layout strategies.

4.1.2 Types

The kinding judgment $\Theta; \Delta \vdash \tau : \kappa$ means that under environments Θ and Δ, type τ has kind κ. All rules are standard except for the kind Ω_c. Figure 7 shows some kinding rules. The rest are in the technical report.

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 (Figure 8).

The subtyping judgment $\Theta; \Delta \vdash \tau_1 \leq \tau_2$ means that under environments Θ and Δ, τ_1 is a subtype of τ_2 (Figure 9).

Subtyping between quantified types is similar to Castagna and Pierce's $\forall -$ top rule [9], where the checker ignores the bounds of type variables (relax it to Topc) when checking subtyping between the body types. Contrary to LIL$_C$, Castagna and Pierce's bounded quantification was based on subtyping, and did not have the minimal type property [8].

4.1.3 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 10 shows all expression typing rules.

Only records of type $R(C)$ can be coerced to objects of C. Coercing an object of C to a record results in a record of type $R(C)$. Coercing an object of α to a record results in a record of type Approx$R(\alpha, C)$ if $\alpha \ll C$.

In a call expression "$e[\tau_1, \ldots, \tau_m](e_1, \ldots, e_n)$", e must 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 $\tau_1, \ldots, \tau_m$.

In "ifParent(e) then bind (α, x) in e_1 else e_2", tag e has type Tag(τ) for some type τ. If e has a parent tag, a new type variable α is introduced for the super class of τ and a new value variable x (of type Tag(α)) is introduced for the parent tag.

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. Otherwise, the first tag e_{t1} must have type Tag(γ) 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.

As usual, a subsumption rule says: if $\tau_1 \leq \tau_2$ and e has type τ_1, then e has type τ_2. No similar rules exist for subclassing because of the exact class notions.

We elaborate the downcast function in Figure 3 a bit here to explain the usage of "ifEqTag". To cast an object *obj* of type β to a class α, the *loop* function compares a tag t_γ in the tag chain of *obj* with the tag of the target class. t_γ identifies γ and $\beta \ll \gamma$. If the two tags are equal, then the cast succeeds: *obj* is packed to have type $\exists \delta \ll \gamma.\ \delta$, which is $\exists \delta \ll \alpha.\ \delta$ because $\gamma = \alpha$. Otherwise, the parent tag of t_γ is fetched and *loop* is called recursively. The cast fails if no such parent tag exists. The downcast function *Downcast* calls *loop* with the target tag t_α, the object *obj* and the tag in *obj*'s vtable (of type Tag(β)).

The typing judgment $\Theta; \Sigma \vdash hv : \tau$ means under environments Θ and Σ, heap value hv has type τ (Figure 11).

$$\frac{\Theta; \bullet; \Sigma; \bullet \vdash v_i : \tau\ \forall 0 \leq i \leq n-1}{\Theta; \Sigma \vdash [v_0, \ldots, v_{n-1}]^\tau : \text{array}(\tau)}\ \textbf{hv\_array}$$

$$\frac{\Theta; \bullet; \Sigma; \bullet \vdash v_i : \tau_i\ \forall 1 \leq i \leq n}{\Theta; \Sigma \vdash \{l_i = v_i\}_{i=1}^n : \{\!\{l_i^{\phi_i} : \tau_i\}\!\}_{i=1}^n}\ \textbf{hv\_record}$$

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

Figure 11: Heap Value Typing Rules

4.1.4 Class Declarations and Programs

A well-formed class contains all the members inherited from its parent, which is solely an implementation strategy to simplify looking up members. LIL$_C$ requires that the

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

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

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : C}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{c2r}(e) : R(C)}\ \textbf{c2r\_c} \qquad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \alpha \quad \Theta;\Delta \vdash \alpha \ll C}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{c2r}(e) : \text{Approx}R(\alpha, C)}\ \textbf{c2r\_tv}$$

$$\frac{\begin{array}{c}\tau = \{\!\{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\}\!\} \\ \Theta;\Delta;\Sigma;\Gamma \vdash e_i : \tau_i\ \forall 1 \le i \le n\end{array}}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{new}[\tau]\{l_1 = e_1, \ldots, l_n = e_n\} : \tau}\ \textbf{record} \qquad \frac{\Theta;\Delta;\Sigma;\Gamma \vdash e : \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\} \quad 1 \le i \le n}{\Theta;\Delta;\Sigma;\Gamma \vdash e.l_i : \tau_i}\ \textbf{field}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \{l_1^{\phi_1} : \tau_1, \ldots, l_i^{M} : \tau_i, \ldots, l_n^{\phi_n} : \tau_n\} \qquad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau_i \qquad \Theta;\Delta;\Sigma;\Gamma \vdash e_3 : \tau}{\Theta;\Delta;\Sigma;\Gamma \vdash e_1.l_i := e_2 \text{ in } e_3 : \tau}\ \textbf{assignR}$$

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

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

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

$$\frac{\begin{array}{c}\Theta;\Delta;\Sigma;\Gamma \vdash e : \forall tvs(\tau_1, \ldots, \tau_n) \to \tau \\ tvs = \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m \quad \sigma = t_1, \ldots, t_m/tvs \\ \Theta;\Delta \vdash t_i \ll u_i[\sigma]\ \ \forall 1 \le i \le m \\ \Theta;\Delta;\Sigma;\Gamma \vdash e_i : \tau_i[\sigma]\ \forall 1 \le i \le n\end{array}}{\Theta;\Delta;\Sigma;\Gamma \vdash e[t_1, \ldots, t_m](e_1, \ldots, e_n) : \tau[\sigma]}\ \textbf{call} \qquad \frac{\begin{array}{c}\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \exists \beta \ll \tau_u.\ \tau \\ \alpha \notin \text{domain}(\Delta) \qquad \alpha \notin free(\tau') \\ \Theta;\Delta, \alpha \ll \tau_u;\Sigma;\Gamma, x : \tau[\alpha/\beta] \vdash e_2 : \tau'\end{array}}{\Theta;\Delta;\Sigma;\Gamma \vdash (\alpha, x) = \text{open}(e_1) \text{ in } e_2 : \tau'}\ \textbf{open}$$

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

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

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_{t1} : \text{Tag}(C_1) \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_{t2} : \text{Tag}(C_2) \quad C_1 = C_2 \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{ifEqTag}^{\tau}(e_{t1}, e_{t2}) \text{ then } e_1 \text{ else } e_2 : \tau}\ \textbf{ifTag\_eq}$$

$$\frac{\Theta;\Delta;\Sigma;\Gamma \vdash e_{t1} : \text{Tag}(C_1) \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_{t2} : \text{Tag}(C_2) \quad C_1 \neq C_2 \quad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{ifEqTag}^{\tau}(e_{t1}, e_{t2}) \text{ then } e_1 \text{ else } e_2 : \tau}\ \textbf{ifTag\_neq}$$

$$\frac{\begin{array}{c}\Theta;\Delta;\Sigma;\Gamma \vdash e_{t1} : \text{Tag}(\gamma) \qquad \Theta;\Delta;\Sigma;\Gamma \vdash e_{t2} : \text{Tag}(\tau_\gamma) \\ \Delta = \Delta_1, P(\gamma), \Delta_2 \quad P(\gamma) = \gamma \ll u \text{ or } P(\gamma) = \gamma \gg u \quad \Theta;\Delta_1 \vdash \tau_\gamma : \Omega_c \\ \Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau[\gamma/\tau_\gamma] \qquad \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau\end{array}}{\Theta;\Delta;\Sigma;\Gamma \vdash \text{ifEqTag}^{\tau}(e_{t1}, e_{t2}) \text{ then } e_1 \text{ else } e_2 : \tau}\ \textbf{ifTag\_tv} \qquad \frac{\begin{array}{c}\Theta;\Delta;\Sigma;\Gamma \vdash e : \tau_1 \\ \Theta;\Delta \vdash \tau_1 \le \tau_2\end{array}}{\Theta;\Delta;\Sigma;\Gamma \vdash e : \tau_2}\ \textbf{sub}$$

Figure 10: Expression Typing Rules

parent class be declared before the child class. Figure 12 shows the rule for class declarations.

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

$$\frac{\begin{array}{c}\Theta \vdash B : D\{f_1 : s_1, \ldots, f_p : s_p,\, m_1 : t_1, \ldots, m_q : t_q\} \\ B \text{ declared before } C \text{ in } \Theta \qquad D \text{ is optional} \qquad p \leq n \quad q \leq k \\ \Theta; \bullet \vdash s_i : \Omega\ \forall p+1 \leq i \leq n \qquad \Theta; \bullet \vdash t_i : \Omega\ \forall q+1 \leq i \leq k\end{array}}{\Theta \vdash C : B\{f_1 : s_1, \ldots, f_n : s_n,\, m_1 : t_1, \ldots, m_k : t_k\}}$$

Figure 12: Well-formedness of Class Declarations

A program $(\Theta; H; V; e)$ is well-formed with respect to a heap environment if its class declaration Θ is well-formed, 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 13). The main expression e has no free type variables, and refers only to labels in H and variables in V.

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

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

$$\frac{\begin{array}{c}\Theta = class_1, \ldots, class_m \qquad \Theta \vdash class_i\ \forall 1 \leq i \leq m \\ \Theta \vdash H : \Sigma \qquad \Theta; \Sigma \vdash V : \Gamma \qquad \Theta; \bullet; \Sigma; \Gamma \vdash e : \tau\end{array}}{\Sigma \vdash (\Theta; H; V; e) : \tau}$$

Figure 13: Well-formedness of Programs

4.2 Dynamic Semantics

Figure 14 shows all the evaluation rules except for congruence rules. The programs 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 program P steps to P'.

Expressions $C(v)$ and $\text{c2r}(v)$ coerce between objects and records. "ifParent($tag(C)$) 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 "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 "error[τ]" infinitely loops.

4.3 Properties of LIL$_C$

We have proved the soundness of LIL$_C$, in the style of [34], and the decidability of type checking. Full proofs are in the technical report.

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 either the main expression in P is a value, or $\exists P'$ such that $P \mapsto P'$.*

Proof sketch: by standard induction over the typing rules.

Combining these two theorems, we can conclude that a well-formed LIL$_C$ program never gets stuck: if the main expression in a program is not a value, the program can always go one more step to another well-formed program.

THEOREM 3 (DECIDABILITY OF TYPE CHECKING). *It is decidable whether $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ holds.*

Proof sketch: by proving the minimal type property and decidability of subtyping. We designed a set of syntax-directed subtyping rules (without the transitivity rule) of the form $\Theta; \Delta \vDash \tau_1 \leq \tau_2$. We proved that the new rules terminate and that $\Theta; \Delta \vdash \tau_1 \leq \tau_2$ iff $\Theta; \Delta \vDash \tau_1 \leq \tau_2$.

Similarly, we designed a set of syntax-directed expression typing rules (without the subsumption rule) of the form $\Theta; \Delta; \Sigma; \Gamma \vDash e : \tau$ and proved that: (1) the new rules terminate; (2) if $\Theta; \Delta; \Sigma; \Gamma \vDash e : \tau$, then $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$; (3) if $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$, then $\exists \tau_m$ such that $\Theta; \Delta; \Sigma; \Gamma \vDash e : \tau_m$ and $\Theta; \Delta \vdash \tau_m \leq \tau$.

LEMMA 4 (BOUNDED JOINS AND MEETS). *Two types have a join (least upper bound) if they have a common upper bound. Similarly, two types have a meet (greatest lower bound) if they have a common lower bound. Both joins and meets are in terms of subtyping.*

5. SOURCE LANGUAGE AND TRANSLATION TO LIL$_C$

$$\begin{array}{lcl}
\tau & := & \text{int} \mid C \mid \text{array}(C) \\
e & := & n \mid \ell \mid x \mid x : \tau = e_1 \text{ in } e_2 \mid x := e_1 \text{ in } e_2 \\
 & & \mid new\ C\{f_1 = e_1, \ldots, f_n = e_n\} \mid e.f_i \\
 & & \mid e_1.f_i := e_2 \text{ in } e_3 \mid e.m(e_1, \ldots, e_n) \\
 & & \mid new[e_0, \ldots, e_{n-1}]^\tau \mid e_1[e_2] := e_3 \text{ in } e_4 \\
 & & \mid e_1[e_2] \mid cast[C](e) \mid cast[C[\,]](e) \\
v & := & n \mid \ell \\
hv & := & C\{f_1 = v_1, \ldots, f_n = v_n\} \mid [v_0, \ldots, v_{n-1}]^\tau \\
mdecl & := & \tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e \\
cdecl & := & C\ extends\ B\{f_1 : \tau_1, \ldots, f_n : \tau_n, \\
 & & \qquad mdecl_1, \ldots, mdecl_k\} \\
prog & := & cdecl_1, \ldots, cdecl_n \text{ in } e
\end{array}$$

Figure 15: Syntax of the Source Language

The source language is roughly Featherweight Java (FJ) [21] enhanced with assignments and one dimensional arrays of objects. Figure 15 shows the syntax. 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 $cast[C[\,]](e)$ represents casting expression e to an array of C. The semantics of the source language is standard and is shown in the technical report.

The source-target translation performs several tasks, besides lowering types and expressions: (1) It collects members for each class, including those from super classes. (2) It lifts method definitions to global functions, after adding the "this" parameter. Therefore after translation, class declarations have only method signatures, not method bodies. (3) It creates vtables for classes.

The translation can be optimized, for example, to eliminate pairs of pack and open. We do not focus on those optimizations at present.

Original Program	New Program	Side Conditions
$(H; V; x)$	$(H; V; V(x))$	
$(H; V; \text{c2r}(C(v)))$	$(H; V; v)$	
$(H; V; \text{new}[\tau]\{l_1 = v_1, \ldots, l_n = v_n\})$	$(H'; V; \ell)$	$H' = H, \ell \rightsquigarrow \{l_i = v_i\}_{i=1}^n \quad \ell \notin \text{domain}(H)$
$(H; V; \ell.l_i)$	$(H; V; v_i)$	$H(\ell) = \{l_i = v_i\}_{i=1}^n \; 1 \le i \le n$
$(H; V; \ell.l_i := v_i' \text{ in } e)$	$(H'; V; e)$	$\begin{cases} \forall \ell' \neq \ell, H'(\ell') = H(\ell') \\ H(\ell) = \{l_1 = v_1, \ldots, l_n = v_n\} \\ H'(\ell) = \{l_1 = v_1, \ldots, l_i = v_i', \ldots, l_n = v_n\} \end{cases}$
$(H; V; \text{new}[v_0, \ldots, v_{n-1}]^\tau)$	$(H'; V; \ell)$	$H' = H, \ell \rightsquigarrow [v_0, \ldots, v_{n-1}]^\tau \quad \ell \notin \text{domain}(H)$
$(H; V; \ell[i])$	$(H; V; v_i)$	$H(\ell) = [v_0, \ldots, v_{n-1}]^\tau$ and $0 \le i \le n-1$
$(H; V; \ell[i] := v_i' \text{ in } e)$	$(H'; V; e)$	$\begin{cases} 0 \le i < n \quad \forall \ell' \neq \ell, H'(\ell') = H(\ell') \\ H(\ell) = [v_0, \ldots, v_{n-1}]^\tau \\ H'(\ell) = [v_0, \ldots, v_i', \ldots, v_{n-1}]^\tau \end{cases}$
$(H; V; x : \tau = v \text{ in } e)$	$(H; V'; e)$	$V' = V, x = v$
$(H; V; x := v' \text{ in } e)$	$(H; V'; e)$	$V = V_1, x = v, V_2 \quad V' = V_1, x = v', V_2$
$(H; V; \ell[\tau_1, \ldots, \tau_m](v_1, \ldots, v_n))$	$(H; V'; e_m[\vec{\tau}/tvs])$	$\begin{cases} H(\ell) = \text{fix } g\langle tvs\rangle(x_i : \tau_i)_{i=1}^n : \tau = e_m \\ V' = V, x_1 = v_1, \ldots, x_n = v_n, g = \ell \end{cases}$
$(H; V; (\alpha, x) = \text{open}(v) \text{ in } e_0)$	$(H; V'; e_0[\tau/\alpha])$	$v = \text{pack } \tau \text{ as } \beta \ll \tau_u \text{ in } (v' : \tau') \quad V' = V, x = v'$
$(H; V; \text{ifParent}(v) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2)$	$(H; V'; e_1[B/\alpha])$	$v = tag(C) \quad C \text{ extends } B \quad V' = V, x = tag(B)$
$(H; V; \text{ifParent}(v) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2)$	$(H; V; e_2)$	$v = tag(C) \quad C \text{ does not extend any class.}$
$(H; V; \text{ifEqTag}^\tau(v_1, v_2) \text{ then } e_1 \text{ else } e_2)$	$(H; V; e_1)$	$v_1 = tag(C_1), v_2 = tag(C_2), C_1 = C_2$
$(H; V; \text{ifEqTag}^\tau(v_1, v_2) \text{ then } e_1 \text{ else } e_2)$	$(H; V; e_2)$	$v_1 = tag(C_1), v_2 = tag(C_2), C_1 \neq C_2$
$(H; V; \text{error}[\tau])$	$(H; V; \text{error}[\tau])$	

Figure 14: Evaluation Rules

Type Translation Class names are translated to existential types. Covariant array types are translated to existential types that hide the actual element types.

$$\begin{array}{lcl} |\text{int}| & = & \text{int} \\ |C| & = & \exists\alpha \ll C.\ \alpha \\ |\text{array}(C)| & = & \exists\alpha \ll C.\{\!\{tag : Tag(\alpha), \\ & & \qquad table : \text{array}(\exists\beta \ll \alpha.\ \beta)\}\!\} \\ |(\tau_1, \ldots, \tau_n) \to \tau| & = & (|\tau_1|, \ldots, |\tau_n|) \to |\tau| \end{array}$$

Class and Program Translation A method declaration is translated to a pair of a function label and a global function definition with an explicit "this". Suppose class C implements a method m. The translation generates a new label l_m and translates the method as follows:

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

A class declaration is translated to a triple: a class declaration in LIL_C, function definitions that correspond to method declarations and the vtable (see Figure 16). For convenience, we use $vtable_C$ to represent the label for C's vtable. If S is a sequence of bindings, then $S \oplus b$ means "S, b" if S does not have the item bound in b. Otherwise, $S \oplus b$ means replacing with b the corresponding entry in S. "$mtype(\Theta, m, C)$" means the type of method m in class C.

A program is translated to a set of new class declarations, an initial heap that contains function definitions and vtables, and a new main expression.

$$\frac{\begin{array}{c}\Theta = cdecl_1, \ldots, cdecl_n \\ |cdecl_i| = (newc_i, newms_i, vt_i)\ \forall 1 \le i \le n \\ H_0 = newms_1, \ldots, newms_n, vt_1, \ldots, vt_n\end{array}}{|\Theta \text{ in } e| = (newc_1, \ldots, newc_n; H_0; \bullet; |e|)}$$

Expression Translation Figure 17 shows the translation of expressions. Each new variable introduced during translation is unique. The translation of the new expression "$new\ C\{f_i = e_i\}_{i=1}^n$" creates a record of $vtable_C$ (vtable of C) and fields $f_1, \ldots, f_n$ and then coerces the record to an object. 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 array cast uses "Arraycast", which is similar to "Downcast" and is defined in the technical report.

THEOREM 5. *The translation from the source language to LIL_C preserves types. If $\vdash P : \tau$ holds in the source language, then $\Sigma_0 \vdash |P| : |\tau|$ holds in LIL_C, where Σ_0 is the environment for the initial heap (H_0 in the program translation rule).*

6. EXTENSIONS

6.1 Interfaces

The subclassing relation can extend to interfaces. If a class C implements an interface I, then $C \ll I$. An object of I has type $\exists\alpha \ll I.\ \alpha$.

A standard implementation uses an *itable* to store a tag and function pointers for each interface a class implements, and compares the tag of each itable entry with the tag of the target interface at interface method invocation time[2].

LIL_C can express the searching process with some extensions. The itable of class C can have an array type $\text{array}(\exists\gamma \gg C.\{\!\{tag : \text{Tag}(\gamma), methods : R_I(\gamma, C)\}\!\})$, because each entry in the itable represents a super interface of C. Each entry is a record of two fields: a tag and a method table. R_I is an abstract type constructor for representing method tables. $R_I(I, C)$ is equivalent to a record type that

[2]Some strategies create new objects with specialized itables when objects are cast to interfaces. We do not think those approaches are practical, because they have difficulties with mutable fields and with casting the new objects to other types.

$$|\text{Topc}| = (\text{Topc}\{\}, (), vtable_{\text{Topc}} \rightsquigarrow \{tag = tag(\text{Topc})\})$$

$$\frac{|B| = (B : D\{fields_B, mtypes_B\}, mdefs_B, vtable_B \rightsquigarrow \{tag = tag(B), fps_B\}) \qquad mdecl_i \text{ defines } m_i \qquad |mdecl_i, C| = (l_i, mbody_i)\ \forall 1 \le i \le k}{|C \text{ extends } B\{f_1 : \tau_1, \ldots, f_n : \tau_n, mdecl_1, \ldots, mdecl_k\}| = \left(\begin{array}{l} C : B\{fields_B, f_1 : |\tau_1|, \ldots, f_n : |\tau_n|, mtypes_B \oplus m_1 : |mtype(\Theta, m_1, C)| \oplus \ldots \oplus m_k : |mtype(\Theta, m_k, C)|\}, \\ (l_1 \rightsquigarrow mbody_1, \ldots, l_k \rightsquigarrow mbody_k), \\ vtable_C \rightsquigarrow \{tag = tag(C), fps_B \oplus m_1 = l_1 \oplus \ldots \oplus m_k = l_k\} \end{array} \right)$$

Figure 16: Class Declaration Translation

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

Figure 17: Translation of Expressions

specifies all methods in I. If the tag of interface γ is equal to the tag of target interface I, then $R_I(\gamma, C)$ can be refined to $R_I(I, C)$ and methods can be fetched from it.

6.2 Binary Methods

A binary method takes a parameter that has the same "exact" type as the "this" pointer. A typical example is the equality operation. For example, "eq" in class Point compares two objects of exact class Point. The subclasses must override the "eq" method in the same spirit. The "eq" method in Point2D compares two objects of exact Point2D.

With notions of "exact" types, LIL$_C$ can express binary methods easily. It can use a special annotation on parameters to indicate that those parameters have the same type as "this". The two parameters of a binary method in class C, "this" and the annotated parameter, will be given type C, not $\exists \alpha \ll C.\ \alpha$, because C's subclasses have to override the binary method. For example, the "eq" method in Point will have type Point → Point → bool, and the "eq" method in Point2D will have type Point2D → Point2D → bool.

7. RELATED WORK

The type-theoretic foundations for object-oriented languages have been studied extensively [6, 30, 4, 31, 2, 1, 15, 20, 13]. Most existing work focused on encoding classes and objects to some form of the typed lambda-calculus with records and functions. Many encodings use non-standard implementation techniques for object-oriented features. Some encodings introduce extra runtime overhead [2, 30, 4, 15].

Two complexities involved in class and object encodings are recursion and inheritance. The recursive reference to class C in the definition of C itself is often expressed by recursive types. Bounded quantification (based on subtyping) is introduced to express inheritance [7], but the full-scale version has undecidable type checking [29, 18][3]. The interaction with constructs such as recursive types, intersection types and binary methods further complicates the type system [17, 28, 5, 3].

The object calculi proposed by Abadi and Cardelli can be used to express class-based languages [1]. Object calculi use operations on objects as primitives, instead of those on records. They are not suitable for addressing low-level details used by the compiler, such as object layout and vtables.

Some recent work aimed at efficient encodings: Glew's F_{self} [20], Crary's OREI [13] and League *et al.*'s JFlint [24]. All three calculi and LIL$_C$ express abstract views of objects with type variables and express concrete views with record types, but they use different mechanisms. To aid comparison, we use the LIL$_C$ notation "Approx$R$" in the explanation. LIL$_C$ uses coercion "c2r" to coerce an object of type α to a record type Approx$R(\alpha, C)$, if $\alpha \ll C$. F_{self} combined F-bounds with equi-recursive types in a "self" constructor. An object of class C has type $self\alpha.\text{Approx}R(C, \alpha)$. When unfolded, the object has type α, and α is a subtype of Approx$R(\alpha, C)$. OREI used intersection types $\exists \alpha.\alpha \cap \text{Approx}R(\alpha, C)$ to express both views of objects of C. Either view can be projected by coercions. JFlint used existentially quantified row variables $\exists tail.\mu self.\text{Approx}R(self, C)+tail$[4] to type objects of C. The row variable $tail$ abstracts possible extensions in subclasses.

Among the four efficient calculi, only LIL$_C$ can specify (subclassing) bounds on type variables. As a result, LIL$_C$ can easily express covariant source array types with invariant array types, and with minor extensions it can model

[3] A restricted variant (Kernel F$_{<:}$) has decidable type checking, but it is not very expressive.

[4] "+" is not a notation in League's encoding. We use it to express concatenation.

displays of super classes (for casting) and itables. Also, LIL_C supports type cast implemented by walking through tag chains. JFlint used sum types and a one-armed case to represent cast. Each class must override a projection function that can cast an object to the class, which is non-standard. In another paper, Glew proposed a mechanism to model hierarchical dispatch [19]. He introduced subtyping between tag types, which we think does not agree with the convention that a tag uniquely identifies a class. Because of this subtyping, he had to use variance annotations on tag types because tag creation and tag comparisons need different subtyping behaviors.

Canning *et al.* separated inheritance and subtyping back in 1990 [12]. Their work gave a denotational semantics to inheritance. Objects were modeled as fixed points of recursive records, and classes were fixed points of F-bounded recursive functions, which made the encoding not as simple or as natural as LIL_C.

Bruce used *matching* and *match-bounded polymorphism* to express inheritance [4]. The denotational semantics, which can be seen as an encoding, was based on F-bounded quantification with fixed points at both the term level and the type level. Matching is purely structural. No nominal subclassing relation is preserved.

Xi *et al.* presented an object encoding as an application of guarded recursive datatype constructors [36]. Classes are tags on messages. Subclassing does not imply subtyping. Objects are interpreted as functions that can dispatch messages, which differs from the standard implementation of objects as records. As a result, fetching fields and invoking methods inherited from super classes have extra overhead.

Wright *et al.* compiled a subset of Java to a typed intermediate language [35]. They relied on runtime type checks to perform dynamic dispatch and type cast, and used unordered records to represent interfaces.

Fisher *et al.* proposed an *untyped* calculus for compiling class-based object-oriented languages [16]. They aimed to support inheritance from unknown base classes. Therefore, they used dictionary lookups for field and method access.

Necula *et al.* developed a certifying compiler called SpecialJ for Java [11] as part of the Proof-Carrying Code framework [27]. The compiler translates Java bytecode to x86 assembly code with type annotation, and a safety proof that certifies the code. The compiler handles a large subset of Java, including exceptions. The type system in SpecialJ uses only high-level abstractions such as classes and objects. Subclassing implies subtyping. SpecialJ does not have "exact" type notions. As a result, its type system is unsound—the unsafe dispatch shown in Section 2.3 would be well-typed in SpecialJ [23].

8. CONCLUSION

This paper presents a simple typed intermediate language LIL_C for type-preserving compilation of object-oriented languages. By preserving lightweight class names and restricting bounded quantification to subclassing, LIL_C reduces the encoding complexity significantly, and it is able to faithfully model standard implementation techniques for self application, dynamic dispatch and type cast. It is sound and has decidable type checking. We believe LIL_C is more suitable as a starting point for practical compilers than traditional encodings.

9. REFERENCES

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, New York, 1996.

[2] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *ACM Symposium on Principles of Programming Languages*, pages 396–409, St. Petersburg Beach, Florida, 1996.

[3] Paolo Baldan, Giorgio Ghelli, and Alessandra Raffaeta. Basic theory of f-bounded quantification. *Information and Computation*, 153(1):173–237, 1999.

[4] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[5] Peter Canning, William Cook, Walt Hill, Walter Olthoff, and John C. Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[6] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.

[7] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[8] Giuseppe Castagna and Benjamin C. Pierce. Corrigendum: Decidable bounded quantification. http://www.cis.upenn.edu/∼bcpierce/papers/fsubnew-corrigendum.ps.

[9] Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *21st ACM Symposium on Principles of Programming Languages*, pages 151–162. ACM Press, 1994.

[10] Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. Technical Report MSR-TR-2004-68, Microsoft Corporation. ftp://ftp.research.microsoft.com/pub/tr/TR-2004-68.pdf.

[11] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, June 2000.

[12] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *17th ACM Symposium on Principles of Programming Languages*, pages 125–135. ACM Press, 1990.

[13] Karl Crary. Simple, efficient object encoding using intersection types. Technical report. CMU Technical Report CMU-CS-99-100.

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

[15] Kathleen Fisher. *Type Systems for Object-oriented Programming Languages*. PhD thesis, Stanford University, 1996.

[16] Kathleen Fisher, John H. Reppy, and Jon G. Riecke. A calculus for compiling and linking classes. In *Proceedings of the 9th European Symposium on*

Programming Languages and Systems, pages 135–149. Springer-Verlag, 2000.

[17] Giorgio Ghelli. Recursive types are not conservative over $F_{\leq}$. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 146–162. Springer-Verlag, 1993.

[18] Giorgio Ghelli. Divergence of $F_{<}$ type checking. *Theoretical Computer Science*, 139(1–2):131–162, 1995.

[19] Neal Glew. Type dispatch for named hierarchical types. In *ACM SIGPLAN International Conf. on Functional Programming*, pages 172–182, 1999.

[20] Neal Glew. An efficient class and object encoding. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 311–324, 2000.

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

[22] S. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *15th ACM Symposium on Principles of Programming Languages*, pages 80–87. ACM Press, 1988.

[23] Christopher League. *A Type-preserving Compiler Infrastructure*. PhD thesis, Yale University, 2002.

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

[25] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[26] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527–568, May 1999.

[27] George Necula. Proof-Carrying Code. In *ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[28] Benjamin C. Pierce. Intersection types and bounded polymorphism. In *Typed Lambda Calculi and Applications*, volume 664, pages 346–360. Springer-Verlag, 1993.

[29] Benjamin C. Pierce. Bounded quantification is undecidable. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 427–459. The MIT Press, MA, 1994.

[30] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[31] Didier Rémy. Programming objects with ml-art, an extension to ml with abstract and record types. In *International Conference on Theoretical Aspects of Computer Software*, pages 321–346. Springer-Verlag, 1994.

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

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

[34] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[35] Andrew K. Wright, Suresh Jagannathan, Cristian Ungureanu, and Aaron Hertzmann. Compiling Java to a typed lambda-calculus: A preliminary report. In *Types in Compilation*, pages 9–27, 1998.

[36] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.