Type-Preserving Compilation of Featherweight Java

Christopher League, Valery Trifonov, and Zhong Shao*
league, trifonov, shao}@cs.yale.edu

Computer Science Department, Yale University
P. O. Box 208285, New Haven, CT 06520 USA

Abstract

We present an efficient encoding of core Java constructs in a simple, implementable typed intermediate language. The encoding, after type erasure, has the same operational behavior as a standard implementation using vtables and self-application for method invocation. Classes inherit super-class methods with no overhead. We support mutually recursive classes while preserving separate compilation. Our strategy extends naturally to a significant subset of Java, including interfaces and privacy. The formal translation using Featherweight Java allows comprehensible type-preservation proofs and serves as a starting point for extending the translation to new features.

1 Introduction

Many compilation techniques for functional languages focus on type-directed compilation [22, 25, 30]. Source-level types are transformed along with the program and then used to guide and justify advanced optimizations. More generally, types preserved throughout compilation can be used to reason about the safety and security of object code [21, 23, 24].

Recently, several researchers have attempted to bring these benefits to object-oriented languages [7, 12, 18, 32]. Last year’s FOOL workshop even featured a panel discussion on typed intermediate languages.

These intermediate languages are typically based on typed λ-calculi. There is significant precedent for encoding object-oriented languages in typed λ-calculi [2, 4, 5, 6, 9], but this domain—type-preserving compilation—imposes several new requirements and allows us to reject a few traditional assumptions. The intermediate language must provide extremely simple primitives (that correspond, e.g., to at most several machine instructions), so that our encodings are amenable to optimization. We must avoid introducing any dynamic overhead solely to achieve static typing. In addition, the type system should be as simple as possible, so that type checking is efficient in practice. Subsumption is not required—it can be replaced with explicit coercions, as long as their runtime cost is nil. In an intermediate language we are not concerned with syntactic niceties or resemblance to source-level constructs. Finally, a type-preserving compiler should preserve source-level abstractions. Link-time type checking will not prevent, e.g., one class from accessing the private fields of another—unless the abstractions are preserved in the object code.

The main contribution of this paper is an efficient encoding of key Java™ [13] constructs in a simple, implementable typed intermediate language. After type erasure, our code has the same operational behavior as a standard implementation using self-application for method invocation. Our strategy extends naturally to a significant subset of Java and an implementation is in progress.

This paper extends and improves our previous work [18] in four significant ways. First, it supports mutually recursive classes. Java allows classes to depend on one another’s types and components in ways that test the limitations of the SML module system. Our solution maintains separate compilation of classes. Second, we give a complete implementation of dynamic casts—another challenge for type theory—without using an imperative tag generator. Again, our solution is compatible with separate compilation. Third, the small source calculus we use allows comprehensible proofs of interesting formal properties of the translation, such as type preservation. Finally, the core translation presented here is an effective starting point for designing encodings of and proving properties about interesting source language extensions, such as privacy, genericity, and reflection.

We describe the syntax and semantics of our source and target languages in the next two sections. In section 4, we explain and formalize each aspect of our translation and prove that it preserves types. Section 5 discusses several extensions, focusing on a tricky but tractable interaction between mutual recursion and privacy. We contrast our technique with recent related work in section 6.

2 Source language

The source language for our translation is Featherweight Java (FJ), a “minimal core calculus for modeling Java’s type system” [16]. The syntax is given in figure 1; for reference, we reprint the semantics in appendix A.

Class declarations (CL) contain the names of the new class and its super class, a sequence of field declarations, a con-

*This research was sponsored in part by the Defense Advanced Research Projects Agency ISO under the title “Scaling Proof-Carrying Code to Production Compilers and Security Policies,” ARPA Order No. H529, issued under Contract No. F30602-99-1-0519, and in part by NSF Grant CCR-9901011. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.
We present an efficient encoding of core Java constructs in a simple, implementable typed intermediate language. The encoding, after type erasure, has the same operational behavior as a standard implementation using vtables and selfapplication for method invocation. Classes inherit super-class methods with no overhead. We support mutually recursive classes while preserving separate compilation. Our strategy extends naturally to a significant subset of Java, including interfaces and privacy. The formal translation using Featherweight Java allows comprehensible type-preservation proofs and serves as a starting point for extending the translation to new features.
of class \( C \) field selection, method invocation, object creation, and cast. Other variables. There are five forms of expressions: variables, polymorphic

\[ \Gamma \vdash \text{abs} \] to types and call constructors of \( C \) to range over field names, \( \text{ctor} \) (constructor \( \text{ctor} \)), returns the sequence of all the fields found in objects given in figure 3. The subtype relation \( \kappa \) is already bound. Boldface braces to constructors of \( C \) and have tuple kinds \( \{ \tau : \kappa \} \). Their components are se-

The subtype relation \( \kappa \) is reflexive, transitive closure \( \kappa \). The relation \( \text{fields}(C) \) returns the sequence of all the fields found in objects of class \( C \). The relation \( \text{type}(m, C) \) finds the type signature for method \( m \) in class \( C \) by searching up the hierarchy. Type signatures have the form \( D_1 \ldots D_n \rightarrow D_0 \).

The expression typing rules govern judgments of the form \( \Gamma \vdash e \in C \), meaning that \( FJ \) expression \( e \) is of type \( C \) in context \( \Gamma \). The operational semantics are given by three primitive reduction rules and the expected congruence rules. Since there are no side effects, evaluation order is unspecified. The \( FJ \) type system is sound and decidable. Please see the appendix for the rules, or [16] for further explanation.

3 Target language

The target language of our translation is the higher-order polymorphic \( \lambda \)-calculus \( F_\omega \) [11, 29] extended with type tuples, existential types [20], row polymorphism [27], ordered records, sum types, iso-recursive types, and a term-level fix-

Figure 1: Syntax of Featherweight Java: classes, constructors, methods, and expressions.

Figure 2: Syntax of the target language.

\{ \cdot \} denote the record constructor, which lifts a complete row type (of kind \( R^0 \)) to kind \( Type \). Permutations of rows are not considered equivalent, so record selection \( e.1 \) can be compiled using fixed offsets. We sometimes use commas and omit \( Abs^k \) when specifying complete rows (see the derived forms in figure 2). We let \( 1 \) (read ‘unit’) denote the empty record type.

Labeled sum types are constructed by enclosing a complete row within boldface brackets: \( [ \cdot ] \). Sum types are introduced by a term-level injection and eliminated by an ML-like case statement; see rules (8) and (9). Figure 2 defines a parameterized type \( \text{maybe} \) with constructors \( \text{some} \) and \( \text{none} \).

We use iso-recursive types at higher kinds. The rules for folding and unfolding them are unconventional, and deserve further explanation. Suppose we wish to encode the following mutually recursive type abbreviations:

\[
\begin{align*}
\text{type } \text{even} & = \text{maybe} \{ \text{hd} : \text{int}, \text{tl} : \text{odd} \} \\
\text{type } \text{odd} & = \{ \text{hd} : \text{int}, \text{tl} : \text{even} \}
\end{align*}
\]

The solution is expressed as the fixpoint over a tuple:

\[
t = \mu \alpha : \{ \cdot \} \text{:: Type, } \alpha : \text{odd :: Type}. \\
\{ \cdot \} = \text{maybe} \{ \text{hd} : \text{int}, \text{tl} : \alpha \text{-odd} \}, \\
\alpha : \text{odd :: Type}
\]

Now, the two recursive types are expressed as \( t \text{-even} \) and \( t \text{-odd} \). There are, however, no type equivalence rules for reducing \( t \text{-even} \); a term having this type must first be \( \text{unfolded} \). We allow unfolding of recursive types within a tuple by specifying a label after the \( \text{at} \) keyword. If \( e \) has type \( t \text{-odd} \), then the expression \( \text{unfold } e \text{ as } t \text{ at } \text{odd} \) has type \( \{ \text{hd} : \text{int}, \text{tl} : t \text{-even} \} \). For recursive types of kind \( Type \), we simply omit the \( \text{at} \) clause. To conserve space, we sometimes omit type annotations that can be readily inferred, writing, e.g., \( \text{unfold } e \) for \( \text{unfold } e \text{ as } t \) when \( e \) has type \( t \).
already in wide use in the SML/NJ compiler [31].

will be available in a companion technical report. The imple-

Row and record types:

\[ \vdash \Phi \text{ kind env} \]

\[ \vdash \Phi \vdash \tau \text{ :: Type} \]

\[ \Phi \vdash l_i: \tau_i; j_i:\tau_j \vdash R_i^{(l_i)} \]

\[ \Phi \vdash \tau \vdash R^0 \]

\[ \Phi \vdash \{\tau\} \text{ :: Type} \]

Recursive record term:

\[ \Phi; \Delta \vdash e: \{\tau\} \rightarrow \{\tau\} \]

\[ \Phi; \Delta \vdash \text{fix}[\tau] e: \{\tau\} \]

Figure 3: Selected typing rules for the target language. The judgments represented are type formation \( \Phi \vdash \tau :: \kappa \) and term formation \( \Phi; \Delta \vdash e : \tau \), where \( \Phi \) maps type variables to their kinds and \( \Delta \) maps term variables to their types.

In addition to the rules in figure 3, the static semantics includes formation rules for all other syntactic forms and judgments for environment formation and type equivalence. All static judgments are decidable. The type system is sound with respect to a structured operational semantics. The target language also enjoys a type erasure property: type manipulations (e.g., type abstractions, folds, pack/open) can be erased before runtime without affecting the result. Complete details will be available in a companion technical report. The implementation of the target language should be quite practical; it is but a minor extension of FLINT, the intermediate language already in wide use in the SML/NJ compiler [31].

4 Translation

Each FJ class is separately compiled into a closed \( F_c \) term which imports the types, method tables, and constructors of other classes and produces its own method table and constructor. The compilation units are then instantiated and linked together with a term-level fixpoint constructor.

We begin this section by describing and formalizing our basic object encoding. In section 4.2, we give a type-directed translation of FJ expressions. Inheritance, overriding, and constructors are examined as part of the class encoding in section 4.3. Finally, section 4.4 covers linking. Many aspects of the translation are mutually dependent, but we believe this ordering yields a reasonably coherent explanation.

4.1 Object encoding

The standard explanation of method invocation in terms of records and fields is called self application [17]. In a class-based language, the object record contains values for all the fields plus a pointer to a record of methods, called the vtable. The vtable is created once and shared among all objects of the same class. The methods in the vtable expect the object itself as an argument. Suppose class Point has one integer field \( x \) and one method \( \text{getx} \) to retrieve it. Ignoring types for the moment, the term

\[ p_0 = \{ \text{vtab} = \{ \text{getx} = \lambda \text{self}: x. \text{getx}(\text{self}, x) \}, x = 42 \} \]

could be an instance of class Point. The self-application term \( p_0.\text{vtab} \text{getx} p_0 \) invokes the method.

What type can we assign to the self argument? The typing derivation for the self application term forces it to match the type of the object record itself. That is, well-typed self application requires that \( p_0 \) have type \( \tau \) where \( \tau = \{ \text{vtab} = \{ \text{getx} :: \tau \rightarrow \text{int} \}, x :: \text{int} \} \). Because \( \tau \) appears in its own definition, the solution must involve a fixpoint. The recursive types in our target language will suffice if augmenting the code with fold and unfold annotations allows for a proper typing derivation. Let the type of self be

\[ \tau_{p_0} = \text{self}. \{ \text{vtab} :: \{ \text{getx} :: \tau_{p_0} \rightarrow \text{int} \}, x :: \text{int} \} \]

Happily, the folded object term

\[ p_1 = \text{fold} \{ \text{vtab} = \{ \text{getx} = \lambda \text{self} :: \tau_{p_0}. (\text{unfold} \text{self}, x) \}, x = 42 \} \]

as \( \tau_{p_0} \).
is well-typed, as is the augmented self-application term: 

\((\text{unfold } p_1), \text{vtab}. \text{getx } p_1)\).

Suppose class ColorPoint extends Point with an additional field and method: The type of an object of class ColorPoint would be:

\[ \tau_{fp} = \mu \text{self}. \{ \text{vtab}: \{ \text{getx}: \text{self} \rightarrow \text{int}, \text{getc}: \text{self} \rightarrow \text{color} \}, \text{x}: \text{int}, \text{c}: \text{color} \} \]

How do we relate these two types? That is, how does a function expecting a Point accept a ColorPoint? Traditional models employ subsumption—in \( F_\omega \) extended with recursive types and a 'top' subtyping rule, \( \tau_{fp} \leq \tau_{pt} \). We favor explicit (but erasable) type manipulations over subsumption. While it may be possible to implement the necessary subtyping relationships in a calculus of coercions \([8]\), we have meanwhile developed an effective, efficient encoding using more standard, conservative extensions to \( F_\omega \).

Java programmers distinguish the static and dynamic classes of an object—dedicated types indicate static classes; constructors provide dynamic classes. Static classes of a given object differ at different program points; dynamic classes are unchanging. Static classes are known at compile-time; dynamic classes are revealed at run-time only by reflection and dynamic casts.

We implement this distinction via a pair of existentially-quantified rows. Some prefix of the object record is known; the rest is hidden, abstract. Consider this static type of a Point object:

\[ \tau_{pt} = \exists \text{self}. \{ \text{vtab}: \{ \text{getx}: \text{self} \rightarrow \text{int}, \text{tail}: \text{m} \text{self} \}; \text{x}: \text{int}; \text{tail}\} \]

The \( f \) component of the tuple \( \text{tail} \) denotes a hidden row missing the labels \( \text{vtab} \) and \( x \). Subclasses of Point append new fields by packaging non-trivial rows into the witness type. Similarly, \( \text{tail} \) contains a component \( m \) for appending new methods onto the vtable. This component is a type operator expecting the recursive self type, so that it can be propagated to method types in the dynamic class. The Point object \( p_1 \) can be packaged into a term of \( \tau_{pt} \) using the trivial witness type \( \{ f = \text{Abs}[\text{vtab}], m = \lambda s:\text{Type}. \text{Abs}[\text{getc}] \} \). A ColorPoint object would include a non-trivial witness type to append the new field and method:

\[ \{ f = (c:\text{color}; \text{Abs}[\text{vtab},c]), m = \lambda s:\text{Type}. (\text{getc}s : \text{color} ; \text{Abs}[\text{getc},\text{getc}]) \} \]

Now, objects of different dynamic classes can be repackaged into the type of a common super class.

This is, in essence, the object encoding we use to compile Java. Before embarking on the formal translation, we must explore one more aspect: recursive references. Suppose the Point class has also a method \( \text{bump} \) which returns a new Point. The type of objects of class Point must then refer to the type of objects of class Point. This recursive reference calls for another fixpoint, outside the existential:

\[ \mu \text{twin} \exists \text{tail}. \mu \text{self}. \{ \text{vtab}: \{ \text{getx}: \text{self} \rightarrow \text{int}; \text{bump}: \text{self} \rightarrow \text{twin}; \text{tail}: \text{m} \text{self} \}; \text{x}: \text{int}; \text{tail}\} \]

Using self as the return type would overly constrain implementations of \( \text{bump} \), forcing them to return objects of the same dynamic class as the receiver. In Java, type signatures constrain static classes only. Because \( \text{twin} \) is outside the existential, its witness type is distinct from that of self.

We used this technique in \([18]\) to explain self-references, but Java supports mutually recursive references as well. Suppose class \( A \) defines a method returning an object of class \( B \), and vice-versa; ignoring fields entirely for a moment, define the type

\[ AB \equiv \mu w:\{ A::\text{Type}, B::\text{Type} \}, \{ A = \exists \text{tail}::\text{Type} \rightarrow R(\text{getc}) \}, \mu \text{self}::\text{Type}. \{ \text{getc}::\text{self} \rightarrow w::B; \text{tail}::\text{self} \}, \mu w::\text{Type}. \{ \text{getc}::\text{self} \rightarrow w::A; \text{tail}::\text{self} \} \]

Using the contextual fold/unfold described earlier, objects of class \( A \) can be folded into the type \( AB:A.B \). This is the natural generalization of the twin fixpoint. In the most general case, any class can refer to any other; thus, \( w \) must expand to include all classes. This is the technique we use in the formal translation. In a real compiler, we would analyze the reference graph and cluster the strongly-connected classes only. Note that this only addresses the typing aspect; mutual recursion also has term-level implications (any class can construct objects of or downcast to any other—see section 4.3) as well as interactions with privacy—see section 5.

This completes our informal account of the object encoding; we now turn to a formal translation of \( F \) types. Figure 4 defines several functions which govern the layout of fields and methods in object types. Square brackets \( [\cdot] \) denote sequences. The sequence \( s_1 + s_2 \) is the concatenation of sequences \( s_1 \) and \( s_2 \). \( |s| \) denotes the number of elements in \( s \). The domain of a sequence of pairs \( \text{dom}(s) \) is a set consisting of the first elements of each pair in \( s \).

The function \( \text{fieldvec} \) maps a class name \( C \) to a sequence of tuples of the form \( (\text{f,D}) \), indicating a field of type \( D \) named \( f \)—except for the first tuple in the sequence, which is always \( (\text{vtab},v) \), a placeholder for the vtable. Each class simply appends its own fields onto the sequence of fields from its super class. (In \( F \), the fields of a class are assumed to be distinct from those of its super classes.)

The layout of methods in an object type is somewhat trickier. Methods that appear in a class definition are either fresh or they override methods in the super class. Overriding methods do not deserve a new slot in the vtable. The function \( \text{methvec} \) maps a class name \( C \) to a sequence of tuples of the form \( (m,T) \), indicating a method named \( m \) with signature \( T \). Signatures have the form \( D_1 \ldots D_n \rightarrow T \). The helper function \( \text{addmeth} \) iterates through all the methods defined in the class \( C \), adding only those methods that are new. The first tuple in \( \text{methvec} \) is always \( (\text{dyncast},d) \), a pseudo-method used to implement dynamic casts.

Let \( \text{ct} \) denote the set of class names in the program of interest, including \( \text{Obj} \). We abbreviate the kind of a tuple of all object types as \( \text{kct} \). The tuple of row kinds for class \( C \) is abbreviated \( \text{ktaill}[C] \).

\[ \text{kct} \equiv \{ (\text{E::Type})^{\text{EC} \text{ct}} \} \]
\[ \text{ktaill}[C] \equiv \{ m::\text{Type} \rightarrow R(\text{dom}(\text{methvec}(C)), f::R(\text{dom}(\text{fieldvec}(C))) \} \]

For brevity, we sometimes omit kind annotations. By convention, certain named type variables are bound by particular
kinds—w has kind \( kcn \), self and u have kind Type, and tail has kind \( ktail[C] \), where C should be evident from the context.

In figure 5 we define \( \text{Row} \), a type operator that produces rows containing the fields and methods introduced \( between \) two classes in a subclass relationship. Intuitively, \( \text{Row}[C, A] \) includes fields and methods in class C but not in its ancestor class A. Earlier we described how to package dynamic classes into static classes; the witness type was a tuple of rows containing the fields and methods in the dynamic class but not in the static class. This is just one use of the \( \text{Row} \) operator.

The type operator \( \text{Row}[C, A] \) expects three arguments: w, the tuple containing object types for all classes; u, a universal type used to implement dynamic casts; and tail, a tuple of rows containing members of subclasses. The implementation of dynamic cast will be explained in section 4.3. For now, we only observe that the macros in figure 5 simply propagate u so that it can appear in the type of the \( ktail \) pseudo-method.

\( \text{Row}[C, A] \) is defined in three cases. First, if C and A are the same class, then the result is just the tail—those members in subclasses of C. Second, if C is \( \text{Obj} \) (the root of the class hierarchy) and A is the special symbol \( \top \) then the result is the members declared in \( \text{Obj} \). Treating \( \top \) as the trivial super class of \( \text{Obj} \) permits more uniform specifications (since \( \text{Obj} \) contains members of its own). Finally, in the inductive case (where C < A) we look to C’s super class—let’s call it B. \( \text{Row}[B, A] \) produces a type operator for the members between B and A; we need only append the new members of C. Conveniently, \( \text{Row}[B, A] \) has a tail parameter specifically for appending new members.

The new fields in \( C \) are precisely those listed in the declaration of \( C \); we fetch their types from \( \text{w} \) and append tail-f.

The new methods in \( C \) are found using \( \text{addmeth} \), and their type signatures \( D_1 \cdots D_n \rightarrow D \) are translated to arrow types \( \text{self} 
\rightarrow w \cdot D_1 \rightarrow \cdots \rightarrow w \cdot D_n \rightarrow w \cdot D \). We use curried arguments for convenience; an implementation would use multi-argument functions instead. As shown in the informal examples, the row for methods is parameterized by the type of self.

Also in figure 5, we use the \( \text{Row} \) operator to define macros for several variants of the object type for any given class. \( \text{Empty}[C] \) denotes the tuple of empty field and method rows of kind \( ktail[C] \). \( \text{ObjRcd}[C] \) assembles the rows into records, leaving the subclass rows and self type open. \( \text{SelfTy}[C] \) closes self with a fixpoint, and \( \text{ObjTy}[C] \) hides the subclass rows with an existential. Each of these variants is used in our term translation. All of them remain abstracted over both \( w \) (the types of other objects) and u (the universal type, which is simply propagated into the type of dynact). Finally, \( \text{World} \) constructs a package of the types of objects of all classes, given the universal type \( u \); as we will see later, the actual universal type is a labeled sum of object types, and is defined recursively using \( \text{World} \).

4.2 Expression translation

Equipped with an efficient object encoding and several type operators for describing it, we now examine the type-directed translation of \( F \) expressions. Figure 6 contains term macros
The `pack` macro packages and folds an open-self term into a closed, complete object type in mutual fixpoint with all others. Supposing that tail is some row tuple in `ktail[C]` and `e` has type `(SelfTy[C] w u) tail`, the term `pack[C; u; tail; e]` has type `w·C`. The `upcast` macro unfolds and repackages an object term to a term of some super class. When `C <: A` and `e` has type `w·C`, the term `upcast[C; A; u; e]` has type `w·A`. These macros simply and effectively formalize the encoding techniques demonstrated in the previous section. They employ erasable type manipulations only. Note the use of `Rows[C, A]` as the new witness type in `upcast`.

We now explain each of the translation rules in figure 6, beginning with (VAR). Variables in FJ are bound as method arguments. Methods are translated as curried abstractions binding the `same` variable names. Therefore, variable translation (VAR) is trivial. An upcast expression `(C)e` (where `Γ ⊢ e ∈ D` and `D <: C`) is also trivial; the rule (UPCAST) delegates its task to the macro of the same name.

The field selection expression `e.f` translates to an unfold-open-unfold-select idiom in the target language (FIELD). In this sequence, the select alone has runtime effect. Method invocation `e.m(e₁, ..., eₙ)` augments the idiom with applications to self and the other arguments, but there is one complication. The FJ typing rule permits the actual arguments to have type self and the other arguments, but there is one complication. Presently it will suffice to say that classes applied to the unit value `{}` returns a record which contains a function `new`—the constructor for class `C`. The translation (NEW) upcasts all the arguments, then fetches and applies the constructor.

The final case, dynamic casts, may appear quite magical until we reveal the implementation of the dynacast pseudo-method in the next section. For now it is enough to treat dynacast as a black box—a polymorphic function with type `∀α.(u→maybe α)→maybe α`. The argument of dynacast `[ObjTy[C] w u]` is a projection function, attempting to convert a value of type `u` to an object of type `ObjTy[C] w u`. In addition to the new function, the classes record contains a `proj` field for each class `C`, of type `u→maybe (ObjTy[C] w u)`. Thus if we select the dynacast method from an object, instantiate it with the object type for some class `C`, then pass it the...
projection for class \( C \), it will return some \( C \) object if the cast succeeds, or none if it fails. In case of failure, evaluation gets stuck—just as it does in FJ. In full Java, we would throw a ClassCastException.

The expression translation judgment \( \text{exp} \) preserves types. Informally, if \( e \) has type \( C \) then its translation has type \( (\text{World } u)\cdot C \) (for some type \( u \)). To state this property formally, we must first translate all the types in the FJ typing environment \( \Gamma \):

\[
\text{Env}[u; \Gamma; x : D] = \text{Env}[u; \Gamma], x : (\text{World } u) \cdot D
\]

By inspection, it is easy to show that \( \text{Env}[u; \Gamma] \) is a well-formed environment, assuming that the range of \( \Gamma \) is a subset of \( cn \). The type preservation theorem and a proof sketch follow; for more detail, please consult the companion technical report.

**Theorem 1 (type preservation)** If \( \Phi; \Delta \vdash \text{classes} : \{\text{Classes (World } u)\} \) and \( \Gamma \vdash e \in C \) then \( \Phi; \Delta, \text{Env}[u; \Gamma] \vdash \text{exp}[\Gamma; u; \text{classes}; e] : (\text{World } u)\cdot C \).

**Proof:** by induction on the structure of \( e \). All cases are straightforward if we factor out and prove several lemmas. First, we must establish a correspondence between the fields used in the FJ semantics and the \text{fieldvec} relation used for object layout (likewise between \text{methvec} and \text{methvec}). Second, we must establish the correspondence between pairs in \text{fieldvec} (or \text{methvec}) and elements in Rows. All these correspondences are proved by induction on the class hierarchy. Finally, we must show that the \text{PACK} and \text{UPCAST} macros return expressions of the expected type. These can be proved by inspection, but the latter argument requires a non-trivial coherence property for Rows. Specifically, the composition \( \text{Rows}[A, \top] \cdot u \cdot (\text{Rows}[C, A] \cdot w \cdot \text{tail}) \) must be equivalent to \( \text{Rows}[C, \top] \cdot w \cdot u \cdot \text{tail} \). This is proved by induction on the derivation of \( C \in < A \).

4.3 Class encoding

Apart from defining types, classes in FJ serve three other roles: they are extended, invoked to create new objects, and specified as targets of dynamic casts. In our translation, each class declaration is separately compiled into a module exporting a record with three elements—one to address each of these roles. We informally explain our techniques for implementing inheritance, constructors, and dynamic casts, then give the formal translation of class declarations.

In a class-based language, each vtable is constructed once and shared among all objects of the same class. In addition, methods inherited by subclasses should be shared. How might we implement the Point methods so that they can be packaged with a ColorPoint? We make the method record polymorphic over the tail of the self type:

\[
\text{dictPT} = \text{\Lambda \text{tail} : ktail}[\text{PT}]. \left\{ \begin{array}{l}
\text{getx} = \lambda \text{self} : s_p. (\text{unfold self}).x \\
\end{array} \right.
\]

where \( s_p = \mu \alpha. \left\{ \begin{array}{l}
\text{vtab} : \left\{ \begin{array}{l}
\text{getx} : \alpha \rightarrow \text{int} ; \\
\text{x} : \text{int} ; \text{tail} \cdot \alpha
\end{array} \right. ; \\
\end{array} \right.
\]

We call this polymorphic record a dictionary. By instantiating it with different tails, we can directly package its contents into objects of subclasses. Instantiated with empty tails (e.g., \text{Empty}[\text{PT}]), this dictionary becomes a vtable for class Point. Suppose the ColorPoint subclass inherits empty tails. For example, this dictionary becomes a vtable for class Point.

Figure 7: Macros for dictionary, constructor, and class types.

Again, this dictionary can be instantiated with empty tails to produce the ColorPoint vtable. With other instantiations, further subclasses can inherit either of these methods. The dictionary is labeled \text{dict} in the record exported by the class translation.

Constructors in FJ are quite simple; they take all the fields as arguments in the correct order. Fields declared in the super class are immediately passed to the super initializer. We translate the constructor as a function which takes the fields as curried arguments, places them directly into a record with the vtable, and then folds and packages the object. The constructor function is labeled \text{new} in the class record. In section 5, we describe how to implement more realistic constructors.

Implementing dynamic cast in a strongly-typed language is challenging. Somewhere we must determine whether an arbitrary, abstractly-typed object belongs to a particular class. If it does belong, we must somehow refine its type to reflect this new information. Exception matching in SML poses a similar problem. To address these issues, Harper and Stone [15] introduce tags—values which track type information at runtime. If a tag of abstract type \( \text{Tag } \alpha \) equals another tag of known type \( \text{Tag } \tau \), then we update the context to reflect that \( \alpha = \tau \). Note that this differs from intensional type analysis [14], which performs structural comparison and does not distinguish named types.

Tags work well with our encoding; in an implementation
Class declaration translation:

\[
c\text{dec}(C) =
\begin{align*}
&\text{Au::Type. } \lambda \text{inj}: \text{Inj}(C) (\text{World } u) \ u. \lambda \text{proj}: \text{Proj}(C) (\text{World } u) \ u. \\
&\text{let \ dict: } \text{Vtail}::\text{ktail}[C]. \ \text{Dict}(C) (\text{World } u) \ u \ \\
&= \text{DICT}(C; u; \text{inj}; \text{classes}) \\
&\text{in let } \text{vtab} = \text{dict} \ [\text{Empty}[C]] \\
&\text{in } \{ \text{dict} = \text{dict}, \ \text{proj} = \text{proj}, \ \text{new} = \text{NEW}(C; u; \text{vtab}) \}
\end{align*}
\]

Dictionary construction:
\[
\text{DICT}(\text{Dbj}; u; \text{inj}; \text{classes}) = \\
\text{Atail}::\text{ktail}[\text{Obj}]. \ \{ \text{dyncast} = \\
\lambda \text{self}: \text{SelfTy}[C] (\text{World } u) \ u \ \text{tail}. \\
\lambda \text{obj}::\text{Type. } \lambda \text{proj}: u \mapsto \text{maybe } \alpha. \\
\text{proj} (\text{inj } \text{PACK}(\text{Dbj}; u; \text{tail}; \text{self})) \}
\]

\[
\text{CT}(C) = \text{class } C \triangleq B \{ \ldots \} \\
\text{dom}(\text{methvec}(C)) = [l_1 \ldots l_n]
\]

\[
\text{DICT}(C; u; \text{inj}; \text{classes}) = \\
\text{Atail}::\text{ktail}[C]. \\
\text{let } \text{super}: \text{Dict}(B) (\text{World } u) \ u \\
= (\text{classes} B \{ \}). \ \text{dict} \\
\text{of } \text{Row}(C, B) (\text{World } u) \ u \ \\
\text{in } \{ l_1 = \text{METH}(C; l_1; u; \text{tail}; \text{inj}; \text{classes}; \text{super}); \ldots, \\
\ldots, l_n = \text{METH}(C; l_n; u; \text{tail}; \text{inj}; \text{classes}; \text{super}) \}
\]

Constructor code:
\[
\begin{align*}
\text{fields}(C) = D_1: f_1 \ldots D_n: f_n \\
\text{NEW}(C; u; \text{vtab}) = \\
\lambda f_1: (\text{World } u) D_1. \ldots \lambda f_n: (\text{World } u) D_n. \\
\text{let } x = \text{fold} \{ \text{vtab} = \text{vtab}, f_1 = f_1, \ldots, f_n = f_n \} \\
\text{as } \text{SelfTy}[C] (\text{World } u) \ \text{Empty}[C] \\
\text{in } \text{PACK}(C; u; \text{Empty}[C]; x) \}
\end{align*}
\]

Method code:
\[
\begin{align*}
\text{METHOD}(C; \text{dyncast}; u; \text{tail}; \text{inj}; \text{classes}; \text{super}) = \\
\lambda \text{self}: \text{SelfTy}[C] (\text{World } u) \ u \ \text{tail}. \\
\lambda \text{obj}::\text{Type. } \lambda \text{proj}: u \mapsto \text{maybe } \alpha. \\
\text{case } \text{proj} (\text{inj } \text{PACK}(\text{Dbj}; u; \text{tail}; \text{self})) \ \\
\text{of } \text{some } x \Rightarrow \text{some } \{ \alpha \} x. \\
\text{else } \text{super}. \text{dyncast } \text{self } \{ \alpha \} \text{proj} \\
\end{align*}
\]

\[
\text{CT}(C) = \text{class } C \triangleq B \{ \ldots K_{M_1} \ldots M_n \} \\
\text{not } \text{defined in } M_1 \ldots M_n
\]

\[
\text{METHOD}(C; m; u; \text{tail}; \text{inj}; \text{classes}; \text{super}) = \text{super.m}
\]

\[
\text{CT}(C) = \text{class } C \triangleq B \{ \ldots K_{M_1} \ldots M_n \} \\
\exists j: M_j = A(m_1, \ldots, m_m, \text{thin:C}) \\
\Gamma = x_1: A_1, \ldots, x_m: A_m, \text{thin:C} \\
\Gamma = e \in D \Rightarrow D < A \\
\text{EXP}[\Gamma; u; \text{classes}; e] = e
\]

\[
\text{METHOD}(C; m; u; \text{tail}; \text{inj}; \text{classes}; \text{super}) = \\
\lambda \text{self}: \text{SelfTy}[C] (\text{World } u) \ u \ \text{tail}. \\
\lambda \text{obj}::\text{Type. } \lambda \text{proj}: u \mapsto \text{maybe } \alpha. \\
\text{let } \text{this}: (\text{World } u) C = \text{PACK}(C; u; \text{tail}; \text{self}) \\
\text{in } \text{UPCAST}(B; A; u; e)
\]

Figure 8: Translation of class declarations.

that supports assignment and an SML front-end, it may be a good choice. In this formal presentation, however, type refinement complicates the soundness proof and the imperative nature of make-tag constrains the operational semantics, which is otherwise free of side effects. make-tag implements a dynamically extensible sum, which is needed for SML exceptions, but is overkill for classes in Fj.

We propose a simpler approach, which co-opts the dynamic dispatch mechanism. The storable itself provides a kind of runtime class information. A designated method, if overridden in every class, could return the receiver at its dynamic class or any super class. We just need a runtime representation of the target class of the cast, and some way to connect that representation to the corresponding object type. For this, we can use the standard primitive type and a 'one-armed' case. Let \( u \) be a sum type with a variant for each class in the class table. The function
\[
\lambda x: u. \text{case } x \text{ of } c y \Rightarrow \text{some } \{ \text{ObjTy}[c] \ w u \} y \\
\text{else } \text{none } \{ \text{ObjTy}[c] \ w u \}
\]
could dynamically represent class \( C \). To connect it to the object type, we make the dyncast method polymorphic, with the type
\[
\text{self}\rightarrow \forall \alpha. (u\mapsto\text{maybe } \alpha)\mapsto\text{maybe } \alpha
\]
This method can check its own class against the target class by injecting self and applying the function argument. If the result is \textbf{none}, then it tries again by injecting as the super class, and so on up the hierarchy.

With this solution, we must be careful to preserve separate compilation—the universal type \( u \) includes a variant for every class in the program. Fortunately, in a particular class declaration we need only inject objects of that class. Class declarations can treat \( u \) as an abstract type and take the injection function as an argument. Then only the linker needs to know the concrete \( u \) type.

We now explore the formal translation of class declarations and construction of their method dictionaries. In figure 7 we define several macros for describing dictionary and class types. Figure 8 gives translations for each component of the class declaration.

Each class is separately compiled to code that resembles an SML functor—a set of definitions parameterized by both types and terms. Linking—the process of instantiating the separate functors and combining them into single coherent program—will be addressed in the next section.

\texttt{cdec[C]} produces the functor corresponding to class \( C \); see the definition in the top left of figure 8. The code has one type parameter, \( u \), the universal type used for dynamic
4.4 Linking

The final task: instantiate and link the separate class modules together into a single program. Figure 9 gives the translation for a complete FJ program. The LINK function creates a record of classes from a record of the class functors. The result is bound to $x_{\omega}$ and used as the classes parameter in translating the main program expression $e$.

LINK uses fix to create a fixpoint of the record of classes. Each class functor in $x$ has one type parameter and three value parameters. Tagged $u$ was defined in figure 7 as a parameterized sum type with a variant for the object type of each class in the class table. We instantiate each $x_{C}$ with the fixed point of Tagged. Next we pass the injection and projection functions, inj$_e$ and proj$_e$. The final argument to $x_{C}$ is the classes record itself.

Theorem 3 (Well-typed linkage)

$$\forall \Phi; \Delta \vdash \text{LINK}: \{ \{ C : \text{ClassF}[C] \} \in C^{\infty} \} \rightarrow \{ \text{Classes} \ (W u) \ u \}$$

where $u = \mu \omega : \text{Type}$. Tagged $u$

Proof: by inspection.

5 Extensions

Our encoding and translation strategy extend to support a significant subset of Java. Features which require little additional effort include null references (with maybe types), assignment (with mutable records), multiple parameterized constructors (by adding them to the class record), super calls (as used in dyncast), and exceptions (as in SML).

In [18] we ambitiously supported Java interfaces using views. To cast an object to an interface type, we fetch a precomputed view from the vtable and pair the object with it. Thereafter, interface method calls are no more expensive than virtual method calls. This technique works well with mutual recursion and dynamic casts (even dynamic casts to interface types), but we omit it because interfaces significantly complicate the formal presentation, including the source language semantics and type preservation proofs.

Another feature we supported in [18] is privacy—each class used an existential to hide the types of its own private fields. Thus privacy is preserved by the translation: link-time type checking will prevent any other module from accessing the private fields of a class—even if the module was translated from a different source language.

Unfortunately, privacy interacts badly with mutual recursion. Suppose that $A$ has a private field $b$ of class $B$ and that $B$ has a method $geta$ that returns an object of class $A$. From within class $A$, accessing this.$b$ is allowed, as is invoking this.$b$.geta(). It is more difficult to design an encoding that also allows this.$b$.geta()$\beta$. Using the existential interpretation of privacy from [18], each class has its own view of the types of all other objects. From within class $A$, private fields of other objects of class $A$ are visible. Private fields of objects of other classes are hidden, represented by type variables. In our example, this.$b$ would have a type something like “$B$ with private fields $\beta$” where $\beta$ is the abstract type. Likewise, from within class $B$, the type of method $geta$ might be self$\rightarrow$“$A$ with private fields $\alpha$”. The challenge is to allow class $A$ to see that the $\alpha$ in the type of $geta$ is actually the known type of its own private fields.

Propagating this information is especially tricky given the weaknesses of the iso-recursive types used in our target calculus. We have developed a solution which does not require extending the target calculus. Briefly, we need to parameterize everything (including the hidden type itself) by the types of objects of other classes. Then, each class can instantiate the types of the rest of the world using concrete types for its own private fields (wherever they may lurk in other classes) and abstract types for the rest. Unfortunately, the issues are
subtle and a detailed explanation would go out of the scope of the current paper. We are considering extending FJ itself with privacy in order to formalize our extensions.

We are also actively working on other extensions. In the original Featherweight Java paper, for example, Igarashi et al. formalize Generic Java (GJ) [3] and translate it back to FJ by erasing type parameters and adding dynamic casts. With at most a minor extension to our target language type system, we should be able to translate GJ without resorting to dynamic casts.

6 Related work

Fisher and Mitchell [10] use extensible objects to model Java-like class constructs. Our encoding does not rely on extensible objects as primitives, but it may be viewed as an implementation of some of their properties in terms of simpler constructs. Remy and Vouillon [28] use row polymorphism to formalize Objective ML for both class types and type inference on unordered records. Our calculus is explicitly typed, but we use ordered rows to represent the open type of self.

Our object representation is superficially similar to several of the classic encodings in F\(_\alpha\)-based languages [5, 26]. As in the Abadi, Cardelli, and Viswanathan encoding [2], method invocation uses self-application; however, we hide the actual class of the receiver using existential quantification over row variables instead of splitting the object into a known interface and a hidden implementation. This allows reuse of methods in subclasses without any overhead. We use an analog of the recursive-existential encoding due to Bruce [4] to give types to other arguments or results belonging to the same class or a subclass, as needed in Java, without over-restricting the type to be the same as the receiver's.

Several other researchers have described type-preserving compilation of object-oriented languages. Wright, et al. [32] compile a Java subset to a typed intermediate language, but they use unordered records and resort to dynamic type checks because their system is too weak to type self application. Crary [7] encodes the object calculus of Abadi and Cardelli [1] using existential and intersection types in a calculus of coercions. His object encoding has some of the same benefits as ours, though the coercion calculus is a significant departure from F\(_\alpha\). Glew [12] translates a simple class-based object calculus into an intermediate language with F-bounded polymorphism [6, 9] and a special 'self' quantifier: a more complex and ad-hoc target calculus. The present work is a significant extension and simplification of the preliminary results we reported in [18].

We present a more detailed comparison of Glew, Crary, and our own encoding in a forthcoming technical report [19]. Briefly, Glew's self quantifier self \(\alpha.I(\alpha)\) is equivalent to an encoding based on an F-bounded existential: \(\exists \alpha \leq I(\alpha), \alpha\), where \(I(\alpha)\) is the type of a record of methods, with \(\alpha\) as the type of each method's first argument. This connection was independently discovered by Glew and ourselves [personal communication, August 2000]. Self application is typable in this encoding because the object, via subsumption, enjoys two types: the interface type \(I(\alpha)\) and the abstract type \(\alpha\). Crary encodes precisely the same property as an intersection type: \(\exists \alpha, \alpha \land I(\alpha)\). Similarly, our encoding is derived by replacing the F-bound with a higher-order bound and a recursive type, implementing the bound as a coercion function, and then eliminating the coercion using row polymorphism. All three of these encodings are efficient and, we conjecture, fully abstract. (Crary’s informal argument [7] seems to apply to all three encodings, though no proof has been given for any of them.) The primary differences between these encodings are in the complexity required of the target calculi. In scaling them to realistic compilers and source languages, other differences may emerge.

7 Conclusion

We have developed an efficient encoding of key Java constructs in a simple, implementable typed intermediate language. The encoding, after type erasure, has the same operational behavior as a standard implementation of self-application. Our strategy extends naturally to a significant subset of Java. In comparison to our earlier work, we now support mutual recursion and dynamic cast while retaining separate compilation. The formal translation using Featherweight Java allows comprehensible type-preservation proofs and serves as a starting point for extending the translation to new features. We have already started implementing this translation as a new front-end to the SML/NJ compiler.

Acknowledgment

We wish to thank the anonymous referees for their many useful comments.

References


[16] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java—A minimal core calculus for Java and GJ. In Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA99), Denver, November 1999. ACM.


A Featherweight Java semantics

Syntax:

CL ::= class C < {C (f;)*} K M
K ::= C((f;)* {super(f;); (this.f;)*})
M ::= C m((C x;)* {·e;})
e ::= x | e.f | e.m(e;∗) | new C(e;∗) | (C)e

Field lookup:

fields(Obj) = •

CT(C) = class C < B {C1 f1;...Cn fn; K...}
fld(B) = B1 g1;...Bm gm

fields(C) = B1 g1;...Bm gm Ci f1;...Cn fn

Method lookup:

CT(C) = class C < B {...K Mi...Mj}
∃j: Mi = D m(D1 x1;...Dm xm} {·e;}

mtype(m, C) = D1,...Dm>D
mbody(m, C) = (x1...xm, e)

CT(C) = class C < B {...K Mi...Mj}
∀ not defined in Mi...Mj

mtype(m, C) = mtype(m, B)
mbody(m, C) = mbdoy(m, B)

Valid method overriding:

mtype(m, B) = C1...Cn>C0

override(m, B, C1...Cn>C0)

∃T such that mtype(a, B) = T

override(m, B, C1...Cn>C0)

Computation:

fields(C) = D1 f1;...Dn fn

(new C(e1...en)).f1 → e1

(R-Field)

mbody(m, C) = (x1...xn, e0)

(new C(e1...en)).m(d1...dn) →
[d1/x1,...,dn/xn, new C(e1...en)/this] e0

(R-Invk)

C <: D

(D)new C(e1...en) → new C(e1...en)

(R-Cast)

C <: C

CT(C) = class C < B {...} B <: A

Class typing:

K = C(B1 g1;...Bn gm, C1 f1;...Cm fm)

{super(g1;...gm); this.f1;...this.fm = fm;}
fld(B) = B1 g1;...Bm gm

C Mi ok in C ∀i ∈ {1...j}

class C < B {C1 f1;...Cm fm; K M1...Mj} ok

Method typing:

x1 : D1, ..., xn : Dn, this : C ⊢ e ∈ E

E <: D

CT(C) = class C < B {...}

D m(D1 x1;...Dn xn} {·e;} ok in C

Expression typing:

Γ ⊢ x ∈ Γ(x) (T-VAR)

Γ ⊢ e ∈ C

fields(C) = D1 f1;...Dn fn

Γ ⊢ e.fj ∈ Dj

(T-Field)

Γ ⊢ e ∈ C

mtype(m, C) = D1,...Dm>D

Γ ⊢ e.m(e1;...en) ∈ D

(T-Invk)

fields(C) = D1 f1;...Dn fn

Γ ⊢ e1 ∈ Ci, Cj <: Bj, (Vi ∈ {1...j})

Γ ⊢ new C(e1;...en) ∈ C

(T-New)

Γ ⊢ e ∈ D

D <: C

Γ ⊢ (C)e ∈ C

(T-Ucast)

Γ ⊢ e ∈ D

C <: D C ≠ D

Γ ⊢ (C)e ∈ C

(T-Dcast)

Γ ⊢ e ∈ D

C ≠ D D ≠ C

Γ ⊢ (C)e ∈ C

(T-Scast)

Figure 10: Semantics of Featherweight Java (reprinted from [16], with a few adaptations).