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. H559, 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.

```
CL ::= class C ◁ C {(C f;)* K M*}
K ::= C((C f)*) {super(f*); (this.f = f;)*}
M ::= C m((C x)*) {^e;}
e ::= x | e.f | e.m(e*) | new C(e*) | (C)e
```

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

structor (`K`), and a sequence of method declarations (`M`). We use letters `A` through `E` to range over class names, `f` and `g` to range over field names, `m` over method names, and `x` over other variables. There are five forms of expressions: variables, field selection, method invocation, object creation, and cast. A program (*CT*, `e`) consists of a fixed *class table, CT*, mapping class names to declarations, and a *main program expression* `e`.

There are no assignments, interfaces, `super` calls, exceptions, or access control. Constructors always take *all* the fields as arguments, in the correct order. FJ permits recursive class dependencies with the full generality of Java. A class can refer to types and call constructors of *any* other class, including its sub-classes. While this does not complicate the FJ semantics, it is one of the major challenges of our translation.

The subtype relation `<:` is the reflexive, transitive closure of the super class declarations (`class C ◁ B`). The relation *fields*(`C`) returns the sequence of all the fields found in objects of class `C`. The relation *mtype*(`m`, `C`) finds the type signature for method `m` in class `C` by searching up the hierarchy. Type signatures have the form $\texttt{D}_1 \ldots \texttt{D}_n\texttt{->D}_0$.

The expression typing rules govern judgments of the form $\Gamma \vdash \texttt{e} \in \texttt{C}$, meaning that FJ expression `e` is of type `C` in context Γ. 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 λ-calculus F_ω [11, 29] extended with type tuples, existential types [20], row polymorphism [27], ordered records, sum types, iso-recursive types, and a term-level fixpoint for constructing recursive records. The syntax appears in figure 2; typing rules for the non-standard features are given in figure 3.

Labeled tuples of types are enclosed in braces $\{l = \tau \ldots\}$ and have tuple kinds $\{\tau :: \kappa \ldots\}$. Their components are selected using a mid-dot: $\tau{\cdot}l$. The existential types are standard: introduced by the package construct $\langle \alpha :: \kappa = \tau,\ e : \tau' \rangle$ and eliminated (within some restricted scope) by **open**; see rules (1) and (2).

Following Rémy [27] we introduce a kind of rows R^L, where L is the set of labels banned from the row. Abs^L is an empty row of kind R^L, and $l : \tau\,; \tau'$ prepends field l of type τ onto the row τ'. The row formation rules (3) and (4) prohibit duplicate labels: $\forall\alpha :: \mathsf{R}^{\{x\}}.\,\tau$ cannot be instantiated with a row in which x is already bound. Boldface braces

$$
\begin{array}{lll}
\text{Kinds} & \kappa ::= & \mathsf{Type} \mid \mathsf{R}^L \mid \kappa \to \kappa' \mid \{(l :: \kappa)^*\} \\
\text{Types} & \tau ::= & \alpha \mid \lambda\alpha :: \kappa.\,\tau \mid \tau\ \tau' \mid \{(l = \tau)^*\} \mid \tau{\cdot}l \\
 & & \mid \tau \to \tau' \mid \mathsf{Abs}^L \mid l : \tau\,; \tau' \mid \mathbf{\{}\tau\mathbf{\}} \mid \mathbf{[}\tau\mathbf{]} \\
 & & \mid \mu\alpha :: \kappa.\,\tau \mid \forall\alpha :: \kappa.\,\tau \mid \exists\alpha :: \kappa.\,\tau \\
\text{Terms} & e ::= & x \mid \boldsymbol{\lambda} x : \tau.\,e \mid e\ e' \mid \mathbf{\{}(l = e)^*\mathbf{\}} \mid e.l \mid \mathbf{fix}\,\mathbf{[}\tau\mathbf{]}\ e \\
 & & \mid \mathbf{inj}_l^\tau \mid \mathbf{case}\ e\ \mathbf{of}\ (l\ x \Rightarrow e)^*\ \mathbf{else}\ e \\
 & & \mid \mathbf{fold}\ e\ \mathbf{as}\ \tau\ \mathbf{at}\ l \mid \mathbf{unfold}\ e\ \mathbf{as}\ \tau\ \mathbf{at}\ l \\
 & & \mid \boldsymbol{\Lambda}\alpha :: \kappa.\,e \mid e\ [\tau] \mid \langle \alpha :: \kappa = \tau,\ e : \tau' \rangle \\
 & & \mid \mathbf{open}\ e\ \mathbf{as}\ \langle \alpha :: \kappa,\ x : \tau \rangle\ \mathbf{in}\ e'
\end{array}
$$

Derived forms:

$$
\begin{aligned}
l_1 : \tau_1, \ldots, l_n : \tau_n &\equiv l_1 : \tau_1\,; \ldots l_n : \tau_n\,; \mathsf{Abs}^{\{l_1 \ldots l_n\}} \\
\mathbf{1} &\equiv \mathbf{\{}\mathsf{Abs}^{\emptyset}\mathbf{\}} \\
\mathbf{maybe} &\equiv \lambda\alpha :: \mathsf{Type}.\ \mathbf{[}\mathsf{some} : \alpha,\ \mathsf{none} : \mathbf{1}\mathbf{]} \\
\mathbf{some} &\equiv \boldsymbol{\Lambda}\alpha :: \mathsf{Type}.\ \mathbf{inj}_{\mathsf{some}}^{\mathbf{maybe}\ \alpha} \\
\mathbf{none} &\equiv \boldsymbol{\Lambda}\alpha :: \mathsf{Type}.\ \mathbf{inj}_{\mathsf{none}}^{\mathbf{maybe}\ \alpha}\ \mathbf{\{\}} \\
\mathbf{let}\ x : \tau = e\ \mathbf{in}\ e' &\equiv (\boldsymbol{\lambda} x : \tau.\,e')\ e
\end{aligned}
$$

Figure 2: Syntax of the target language.

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

Labeled sum types are constructed by enclosing a complete row within boldface brackets: $\mathbf{[}\cdot\mathbf{]}$. 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 **maybe** with constructors **some** and **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:

$$
\begin{aligned}
\mathbf{type}\ \mathsf{even} &= \mathbf{maybe}\ \mathbf{\{}\mathsf{hd} : \mathbf{int},\ \mathsf{tl} : \mathsf{odd}\mathbf{\}} \\
\mathbf{type}\ \mathsf{odd} &= \mathbf{\{}\mathsf{hd} : \mathbf{int},\ \mathsf{tl} : \mathsf{even}\mathbf{\}}
\end{aligned}
$$

The solution is expressed as the fixpoint over a tuple:

$$
\begin{array}{l}
t = \mu\alpha :: \{\mathsf{even} :: \mathsf{Type},\ \mathsf{odd} :: \mathsf{Type}\}. \\
\quad \{\mathsf{even} = \mathbf{maybe}\ \mathbf{\{}\mathsf{hd} : \mathbf{int},\ \mathsf{tl} : \alpha{\cdot}\mathsf{odd}\mathbf{\}}, \\
\quad\ \ \mathsf{odd} = \mathbf{\{}\mathsf{hd} : \mathbf{int},\ \mathsf{tl} : \alpha{\cdot}\mathsf{even}\mathbf{\}}\}
\end{array}
$$

Now, the two recursive types are expressed as $t{\cdot}\mathsf{even}$ and $t{\cdot}\mathsf{odd}$. There are, however, no type equivalence rules for reducing $t{\cdot}\mathsf{even}$; a term having this type must first be *unfolded*. We allow unfolding of recursive types within a tuple by specifying a label after the **at** keyword. If e has type $t{\cdot}\mathsf{odd}$, then the expression **unfold** e **as** t **at** odd has type $\mathbf{\{}\mathsf{hd} : \mathbf{int},\ \mathsf{tl} : t{\cdot}\mathsf{even}\mathbf{\}}$. For recursive types of kind Type, we simply omit the **at** clause. To conserve space, we sometimes omit type annotations that can be readily inferred, writing, *e.g.*, **unfold** e for **unfold** e **as** τ when e has type τ.

Pack and open for existential types:

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \mathsf{Type} \qquad \Phi \vdash \tau' :: \kappa \quad \Phi; \Delta \vdash e : \tau[\alpha := \tau']}{\Phi; \Delta \vdash \langle \alpha :: \kappa = \tau', e : \tau \rangle : \exists \alpha :: \kappa. \tau} \tag{1}$$

$$\frac{\Phi; \Delta \vdash e : \exists \alpha :: \kappa. \tau \qquad \Phi \vdash \tau' :: \mathsf{Type} \quad \Phi, \alpha :: \kappa; \Delta, x : \tau \vdash e' : \tau'}{\Phi; \Delta \vdash \mathbf{open}\ e\ \mathbf{as}\ \langle \alpha :: \kappa, x : \tau \rangle\ \mathbf{in}\ e' : \tau'} \tag{2}$$

Row and record types:

$$\frac{\vdash \Phi\ \mathit{kind\ env}}{\Phi \vdash \mathsf{Abs}^L :: \mathsf{R}^L} \tag{3}$$

$$\frac{\Phi \vdash \tau :: \mathsf{Type} \qquad \Phi \vdash \tau' :: \mathsf{R}^{L \cup \{l\}}}{\Phi \vdash l : \tau ; \tau' :: \mathsf{R}^{L - \{l\}}} \tag{4}$$

$$\frac{\Phi \vdash \tau :: \mathsf{R}^{\emptyset}}{\Phi \vdash \{\tau\} :: \mathsf{Type}} \tag{5}$$

Recursive record term:

$$\frac{\Phi; \Delta \vdash e : \{\tau\} \rightarrow \{\tau\}}{\Phi; \Delta \vdash \mathbf{fix}\,[\tau]\ e : \{\tau\}} \tag{6}$$

Sum type, its introduction and elimination:

$$\frac{\Phi \vdash \tau :: \mathsf{R}^{\emptyset}}{\Phi \vdash [\tau] :: \mathsf{Type}} \tag{7}$$

$$\frac{\Phi \vdash [l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau] :: \mathsf{Type}}{\Phi; \Delta \vdash \mathbf{inj}_{l_n}^{[l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau]} : \tau_n \rightarrow [l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau]} \tag{8}$$

$$\frac{\begin{array}{l} l'_j = l'_{j'} \Rightarrow j = j' \qquad (\forall j, j' \in \{1 \ldots m\}) \\ \Phi; \Delta \vdash e : [l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau] \qquad \Phi; \Delta \vdash e' : \tau' \\ \exists i \in \{1 \ldots n\} : l_i = l'_j \\ \quad \text{and } \Phi; \Delta, x_j : \tau_i \vdash e_j : \tau' \qquad (\forall j \in \{1 \ldots m\}) \end{array}}{\Phi; \Delta \vdash \mathbf{case}\ e\ \mathbf{of}\ (l'_j\ x_j \Rightarrow e_j)^{j \in \{1 \ldots m\}}\ \mathbf{else}\ e' : \tau'} \tag{9}$$

Fold and unfold for recursive types:

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \kappa \qquad \kappa \equiv \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\} \quad \Phi; \Delta \vdash e : \tau \cdot l_i[\alpha := \mu\alpha :: \kappa. \tau]}{\Phi; \Delta \vdash \mathbf{fold}\ e\ \mathbf{as}\ \mu\alpha :: \kappa. \tau\ \mathbf{at}\ l_i : (\mu\alpha :: \kappa. \tau) \cdot l_i} \tag{10}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \kappa \qquad \kappa \equiv \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\} \quad \Phi; \Delta \vdash e : (\mu\alpha :: \kappa. \tau) \cdot l_i}{\Phi; \Delta \vdash \mathbf{unfold}\ e\ \mathbf{as}\ \mu\alpha :: \kappa. \tau\ \mathbf{at}\ l_i : \tau \cdot l_i[\alpha := \mu\alpha :: \kappa. \tau]} \tag{11}$$

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 Φ maps type variables to their kinds and Δ 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_ω 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 getx to retrieve it. Ignoring types for the moment, the term $p_0 = \{\mathsf{vtab} = \{\mathsf{getx} = \lambda \mathsf{self}.\ (\mathsf{self}.\mathsf{x})\},\ \mathsf{x} = 42\}$ could be an instance of class Point. The self-application term $p_0.\mathsf{vtab}.\mathsf{getx}\ p_0$ 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 τ where $\tau = \{\mathsf{vtab} : \{\mathsf{getx} : \tau \rightarrow \mathbf{int}\},\ \mathsf{x} : \mathbf{int}\}$. Because τ 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

$$\tau_{pt} = \mu\mathsf{self}.\ \{\mathsf{vtab} : \{\mathsf{getx} : \mathsf{self} \rightarrow \mathbf{int}\},\ \mathsf{x} : \mathbf{int}\}$$

Happily, the folded object term

$$p_1 = \mathbf{fold}\ \{\mathsf{vtab} = \{\mathsf{getx} = \lambda \mathsf{self} : \tau_{pt}.\ (\mathbf{unfold}\ \mathsf{self}).\mathsf{x}\},\ \mathsf{x} = 42\}\ \mathbf{as}\ \tau_{pt}$$

is well-typed, as is the augmented self-application term: (**unfold** p_1).vtab.getx p_1.

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

$$\tau_{cp} = \mu\mathsf{self}.\ \{\mathsf{vtab}:\{\mathsf{getx}:\mathsf{self}\to\mathbf{int},\ \mathsf{getc}:\mathsf{self}\to\mathbf{color}\},\ \mathsf{x}:\mathbf{int},\ \mathsf{c}:\mathbf{color}\}$$

How do we relate these two types? That is, how does a function expecting a Point accept a ColorPoint? Traditional models employ subsumption—in $F^{\leq}_{\omega}$ extended with recursive types and a 'top' subtyping rule, $\tau_{cp} \leq \tau_{pt}$. 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_{ω}.

Java programmers distinguish the *static* and *dynamic* classes of an object—declared 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:

$$\begin{aligned}\tau'_{pt} = {}& \exists\mathsf{tail}::\{\mathsf{f}::\mathsf{R}^{\{\mathsf{vtab},\mathsf{x}\}},\ \mathsf{m}::\mathsf{Type}\to\mathsf{R}^{\{\mathsf{getx}\}}\}.\\ & \mu\mathsf{self}.\ \{\mathsf{vtab}:\{\mathsf{getx}:\mathsf{self}\to\mathbf{int};\mathsf{tail}\cdot\mathsf{m}\ \mathsf{self}\};\\ & \qquad \mathsf{x}:\mathbf{int};\\ & \qquad \mathsf{tail}\cdot\mathsf{f}\}\end{aligned}$$

The f component of the tuple tail denotes a hidden row missing the labels vtab and x. Subclasses of Point append new fields by packaging non-trivial rows into the witness type. Similarly, 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 τ'_{pt} using the trivial witness type $\{\mathsf{f}=\mathsf{Abs}^{\{\mathsf{vtab},\mathsf{x}\}},\ \mathsf{m}=\lambda\mathsf{s}::\mathsf{Type}.\ \mathsf{Abs}^{\{\mathsf{getx}\}}\}$. A ColorPoint object would include a non-trivial witness type to append the new field and method:

$$\begin{aligned}&\{\mathsf{f}=(\mathsf{c}:\mathbf{color};\mathsf{Abs}^{\{\mathsf{vtab},\mathsf{x},\mathsf{c}\}}),\\ &\ \mathsf{m}=\lambda\mathsf{s}::\mathsf{Type}.\ (\mathsf{getc}:\mathsf{s}\to\mathbf{color};\mathsf{Abs}^{\{\mathsf{getx},\mathsf{getc}\}})\}\end{aligned}$$

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 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:

$$\begin{aligned}\mu\mathsf{twin}.\ \exists\mathsf{tail}.\ \mu\mathsf{self}.\ \{&\mathsf{vtab}:\{\mathsf{getx}:\mathsf{self}\to\mathbf{int};\\ &\qquad \mathsf{bump}:\mathsf{self}\to\mathsf{twin};\mathsf{tail}\cdot\mathsf{m}\ \mathsf{self}\};\\ &\ \mathsf{x}:\mathbf{int};\ \mathsf{tail}\cdot\mathsf{f}\}\end{aligned}$$

Using self as the return type would overly constrain implementations of bump, forcing them to return objects of the same dynamic class as the receiver. In Java, type signatures constrain static classes only. Because 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

$$\begin{aligned}\mathsf{AB} \equiv \mu\mathsf{w}::\{&\mathsf{A}::\mathsf{Type},\ \mathsf{B}::\mathsf{Type}\}.\\ &\{\mathsf{A}=\exists\mathsf{tail}::\mathsf{Type}\to\mathsf{R}^{\{\mathsf{getb}\}}.\\ &\qquad \mu\mathsf{self}::\mathsf{Type}.\ \{\mathsf{getb}:\mathsf{self}\to\mathsf{w}\cdot\mathsf{B};\mathsf{tail}\ \mathsf{self}\},\\ &\ \mathsf{B}=\exists\mathsf{tail}::\mathsf{Type}\to\mathsf{R}^{\{\mathsf{geta}\}}.\\ &\qquad \mu\mathsf{self}::\mathsf{Type}.\ \{\mathsf{geta}:\mathsf{self}\to\mathsf{w}\cdot\mathsf{A};\mathsf{tail}\ \mathsf{self}\}\}\end{aligned}$$

Using the contextual fold/unfold described earlier, objects of class A can be folded into the type AB·A. 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 FJ 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 dom(s) is a set consisting of the first elements of each pair in s.

The function *fieldvec* maps a class name C to a sequence of tuples of the form (f, D), indicating a field of type D named f—except for the first tuple in the sequence, which is always (vtab, *vt*), a placeholder for the vtable. Each class simply appends its own fields onto the sequence of fields from its super class. (In FJ, 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 *new* or they *override* methods in the super class. Overriding methods do not deserve a new slot in the vtable. The function *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 $\mathtt{D}_1 \ldots \mathtt{D}_n\mathtt{->D}$. The helper function *addmeth* iterates through all the methods defined in the class C, adding only those methods that are new. The first tuple in *methvec* is always (dyncast, *dc*), a pseudo-method used to implement dynamic casts.

Let *cn* denote the set of class names in the program of interest, including Obj. We abbreviate the kind of a tuple of all object types as *kcn*. The tuple of row kinds for class C is abbreviated *ktail*[C].

$$\begin{aligned}&\mathit{kcn} \equiv \{(\mathtt{E}::\mathsf{Type})^{\,\mathtt{E}\in\mathit{cn}}\}\\ &\mathit{ktail}[\mathtt{C}] \equiv \{\mathsf{m}::\mathsf{Type}\to\mathsf{R}^{\mathrm{dom}(\mathit{methvec}(\mathtt{C}))},\ \mathsf{f}::\mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathtt{C}))}\}\end{aligned}$$

For brevity, we sometimes omit kind annotations. By convention, certain named type variables are bound by particular

$$\mathit{fieldvec}(\texttt{Obj}) = [(\texttt{vtab}, \mathit{vt})]$$

$$\frac{CT(\texttt{C}) = \texttt{class C} \lhd \texttt{B \{D}_1\ \texttt{f}_1\texttt{; \ldots D}_m\ \texttt{f}_m\texttt{; K \ldots\}}}{\mathit{fieldvec}(\texttt{C}) = \mathit{fieldvec}(\texttt{B}) +\!\!+ [(\texttt{f}_1, \texttt{D}_1) \ldots (\texttt{f}_m, \texttt{D}_m)]}$$

$$\mathit{methvec}(\texttt{Obj}) = [(\mathsf{dyncast}, \mathit{dc})]$$

$$\frac{CT(\texttt{C}) = \texttt{class C} \lhd \texttt{B \{\ldots K M}_1 \ldots \texttt{M}_m\texttt{\}}}{\mathit{methvec}(\texttt{C}) = \mathit{methvec}(\texttt{B}) +\!\!+ \mathit{addmeth}(\texttt{B}, [\texttt{M}_1 \ldots \texttt{M}_m])}$$

$$\frac{(\texttt{m}, \_) \in \mathit{methvec}(\texttt{B})}{\mathit{addmeth}(\texttt{B}, [\texttt{D m(D}_1\ \texttt{x}_1 \ldots \texttt{D}_k\ \texttt{x}_k\texttt{) \{\ldots\} M}_2 \ldots \texttt{M}_m]) = \mathit{addmeth}(\texttt{B}, [\texttt{M}_2 \ldots \texttt{M}_m])}$$

$$\frac{(\texttt{m}, \_) \notin \mathit{methvec}(\texttt{B})}{\mathit{addmeth}(\texttt{B}, [\texttt{D m(D}_1\ \texttt{x}_1 \ldots \texttt{D}_k\ \texttt{x}_k\texttt{) \{\ldots\} M}_2 \ldots \texttt{M}_m]) = [(\texttt{m}, \texttt{D}_1 \ldots \texttt{D}_k\texttt{->D})] +\!\!+ \mathit{addmeth}(\texttt{B}, [\texttt{M}_2 \ldots \texttt{M}_m])}$$

$$\mathit{addmeth}(\texttt{B}, []) = []$$

Figure 4: Field and method layouts for object types.

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 *Rows*, a type operator that produces rows containing the fields and methods introduced *between* two classes in a subclass relationship. Intuitively, *Rows*[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 *Rows* operator.

The type operator *Rows*[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 dyncast pseudo-method.

Rows[C, A] is defined by 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 Obj (the root of the class hierarchy) and A is the special symbol $\top$ then the result is the members declared in Obj. Treating $\top$ as the trivial super class of Obj permits more uniform specifications (since 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. *Rows*[B, A] produces a type operator for the members between B and A; we need only append the *new* members of C. Conveniently, *Rows*[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 w and append tail·f.

$$\mathit{Rows}[\texttt{C}, \texttt{C}] = \lambda\mathsf{w}.\ \lambda\mathsf{u}.\ \lambda\mathsf{tail}{::}\mathit{ktail}[\texttt{C}].\ \mathsf{tail}$$

$$\begin{aligned}\mathit{Rows}[\texttt{Obj}, \top] = {} & \lambda\mathsf{w}.\ \lambda\mathsf{u}.\ \lambda\mathsf{tail}{::}\mathit{ktail}[\texttt{Obj}]. \\ & \{\mathsf{m} = \lambda\mathsf{self}.\ (\mathsf{dyncast} : \mathsf{self} \longrightarrow \\ & \qquad \forall\alpha.\ (\mathsf{u} \longrightarrow \mathbf{maybe}\ \alpha) \longrightarrow \mathbf{maybe}\ \alpha\,; \\ & \qquad \mathsf{tail{\cdot}m\ self}) \\ & \ \mathsf{f} = \mathsf{tail{\cdot}f}\}\end{aligned}$$

$$\frac{\begin{array}{l}CT(\texttt{C}) = \texttt{class C} \lhd \texttt{B \{D}_1\ \texttt{f}_1 \ldots \texttt{D}_n\ \texttt{f}_n\ \texttt{K M}_1 \ldots \texttt{M}_m\texttt{\}} \\ \mathit{addmeth}(\texttt{B}, [\texttt{M}_1 \ldots \texttt{M}_m]) = [(l_1, T_1) \ldots (l_m, T_m)] \\ \mathit{Rows}[\texttt{B}, \texttt{A}] = \tau\end{array}}{\begin{array}{l}\mathit{Rows}[\texttt{C}, \texttt{A}] = \lambda\mathsf{w}.\ \lambda\mathsf{u}.\ \lambda\mathsf{tail}{::}\mathit{ktail}[\texttt{C}]. \\ \quad \tau\ \mathsf{w}\ \mathsf{u}\ \{\mathsf{m} = \lambda\mathsf{self}.\ (l_1 : \mathit{Ty}[\mathsf{self}; \mathsf{w}; T_1]\,; \ldots \\ \qquad\qquad l_m : \mathit{Ty}[\mathsf{self}; \mathsf{w}; T_m]\,; \mathsf{tail{\cdot}m\ self}), \\ \qquad \mathsf{f} = (\mathsf{f}_1 : \mathsf{w{\cdot}D}_1\,; \ldots \mathsf{f}_n : \mathsf{w{\cdot}D}_n\,; \mathsf{tail{\cdot}f})\}\end{array}}$$

$$\mathit{Ty}[\mathsf{self}; \mathsf{w}; \texttt{D}_1 \ldots \texttt{D}_n\texttt{->D}] = \mathsf{self} \longrightarrow \mathsf{w{\cdot}D}_1 \longrightarrow \ldots\ \mathsf{w{\cdot}D}_n \longrightarrow \mathsf{w{\cdot}D}$$

$$\begin{aligned}\mathit{Empty}[\texttt{C}] &\equiv \{\mathsf{m} = \lambda\mathsf{self}.\ \mathsf{Abs}^{\mathrm{dom}(\mathit{methvec}(\texttt{C}))}, \\ &\qquad \mathsf{f} = \mathsf{Abs}^{\mathrm{dom}(\mathit{fieldvec}(\texttt{C}))}\} \\ \mathit{ObjRcd}[\texttt{C}] &\equiv \lambda\mathsf{w}.\ \lambda\mathsf{u}.\ \lambda\mathsf{tail}.\ \lambda\mathsf{self}. \\ &\qquad \{\mathsf{vtab} : \{(\mathit{Rows}[\texttt{C}, \top]\ \mathsf{w\ u\ tail}){\cdot}\mathsf{m\ self}\}\,; \\ &\qquad\ (\mathit{Rows}[\texttt{C}, \top]\ \mathsf{w\ u\ tail}){\cdot}\mathsf{f}\} \\ \mathit{SelfTy}[\texttt{C}] &\equiv \lambda\mathsf{w}.\ \lambda\mathsf{u}.\ \lambda\mathsf{tail}.\ \mu\mathsf{self}.\ \mathit{ObjRcd}[\texttt{C}]\ \mathsf{w\ u\ tail\ self} \\ \mathit{ObjTy}[\texttt{C}] &\equiv \lambda\mathsf{w}.\ \lambda\mathsf{u}.\ \exists\mathsf{tail}.\ \mathit{SelfTy}[\texttt{C}]\ \mathsf{w\ u\ tail} \\ \mathit{World} &\equiv \lambda\mathsf{u}.\ \mu\mathsf{w}.\ \{(\texttt{E} = \mathit{ObjTy}[\texttt{E}]\ \mathsf{w\ u})^{\,\texttt{E} \in cn}\}\end{aligned}$$

Figure 5: Macros for object types.

The new methods in C are found using *addmeth*, and their type signatures $\texttt{D}_1 \ldots \texttt{D}_n\texttt{->D}$ are translated to arrow types $\mathsf{self} \longrightarrow \mathsf{w{\cdot}D}_1 \longrightarrow \ldots\ \mathsf{w{\cdot}D}_n \longrightarrow \mathsf{w{\cdot}D}$. 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 *Rows* operator to define macros for several variants of the object type for any given class. *Empty*[C] denotes the tuple of empty field and method rows of kind *ktail*[C]. *ObjRcd*[C] assembles the rows into records, leaving the subclass rows and self type open. *SelfTy*[C] closes self with a fixpoint, and *ObjTy*[C] hides the sublass 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 dyncast). Finally, *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 *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 FJ expressions. Figure 6 contains term macros

$$\text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{x}] = \mathtt{x} \qquad \text{(VAR)}$$

$$\frac{(\mathtt{f}, \_) \in \mathit{fieldvec}(\mathtt{C}) \quad \Gamma \vdash \mathtt{e} \in \mathtt{C} \quad \text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e}] = e}{\begin{array}{l}\text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e.f}] = \\ \quad \textbf{open unfold } e \textbf{ as } \mathit{World}\ \mathsf{u} \textbf{ at } \mathtt{C} \\ \quad\quad \textbf{as } \langle \mathsf{tail}, x : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}\rangle \\ \quad\quad \textbf{in } (\textbf{unfold } x).\mathtt{f}\end{array}} \qquad \text{(FIELD)}$$

$$\frac{\begin{array}{l}(\mathtt{m}, \mathtt{B}_1 \ldots \mathtt{B}_n\mathtt{->B}) \in \mathit{methvec}(\mathtt{C}) \\ \Gamma \vdash \mathtt{e} \in \mathtt{C} \quad \text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e}] = e \\ \left.\begin{array}{ll}\Gamma \vdash \mathtt{e}_i \in \mathtt{D}_i & \text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e}_i] = e_i \\ \mathtt{D}_i \mathtt{<:} \mathtt{B}_i & \text{UPCAST}[\mathtt{D}_i; \mathtt{B}_i; \mathsf{u}; e_i] = e_i'\end{array}\right\}_{i \in \{1 \ldots n\}}\end{array}}{\begin{array}{l}\text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e.m(e}_1 \ldots \mathtt{e}_n\mathtt{)}] = \\ \quad \textbf{open unfold } e \textbf{ as } \mathit{World}\ \mathsf{u} \textbf{ at } \mathtt{C} \\ \quad\quad \textbf{as } \langle \mathsf{tail}, x : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}\rangle \\ \quad\quad \textbf{in } (\textbf{unfold } x).\mathtt{vtab.m}\ x\ e_1'\ \ldots\ e_n'\end{array}} \qquad \text{(INVOKE)}$$

$$\frac{\begin{array}{l}\mathit{fields}(\mathtt{C}) = \mathtt{B}_1\ \mathtt{f}_1 \ldots \mathtt{B}_n\ \mathtt{f}_n \\ \left.\begin{array}{ll}\Gamma \vdash \mathtt{e}_i \in \mathtt{D}_i & \text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e}_i] = e_i \\ \mathtt{D}_i \mathtt{<:} \mathtt{B}_i & \text{UPCAST}[\mathtt{D}_i; \mathtt{B}_i; \mathsf{u}; e_i] = e_i'\end{array}\right\}_{i \in \{1 \ldots n\}}\end{array}}{\begin{array}{l}\text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{new\ C(e}_1 \ldots \mathtt{e}_n\mathtt{)}] = \\ \quad (\mathsf{classes.C}\ \{\})\mathtt{.new}\ e_1'\ \ldots\ e_n'\end{array}} \qquad \text{(NEW)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{D} \quad \text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e}] = e \quad \mathtt{D} \mathtt{<:} \mathtt{C} \quad \text{UPCAST}[\mathtt{D}; \mathtt{C}; \mathsf{u}; e] = e'}{\text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{(C)e}] = e'} \qquad \text{(UPCAST)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{D} \quad \mathtt{C} \mathtt{<:} \mathtt{D} \quad \text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e}] = e}{\begin{array}{l}\text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{(C)e}] = \\ \quad \textbf{open unfold } e \textbf{ as } \mathit{World}\ \mathsf{u} \textbf{ at } \mathtt{C} \\ \quad\quad \textbf{as } \langle \mathsf{tail}, x : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}\rangle \\ \quad\quad \textbf{in case } (\textbf{unfold } x).\mathtt{vtab.dyncast}\ x \\ \quad\quad\quad\quad [(\mathit{World}\ \mathsf{u}) \cdot \mathtt{C}] \\ \quad\quad\quad\quad (\mathsf{classes.C}\ \{\})\mathtt{.proj} \\ \quad\quad\quad \textbf{of some } y \Rightarrow y \\ \quad\quad\quad\quad \textbf{else } \text{¡ClassCast error!}\end{array}} \qquad \text{(DNCAST)}$$

Macros for pack and upcast transformations:

$$\begin{array}{l}\text{PACK}[\mathtt{C}; \mathsf{u}; \mathsf{tail}; e] = \\ \quad \textbf{fold } \langle \mathsf{tail}' :: \mathit{ktail}[\mathtt{C}] = \mathsf{tail}, \\ \quad\quad\quad e : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}'\rangle \\ \quad\quad \textbf{as } \mathit{World}\ \mathsf{u} \textbf{ at } \mathtt{C}\end{array}$$

$$\begin{array}{l}\text{UPCAST}[\mathtt{C}; \mathtt{A}; \mathsf{u}; e] = \\ \quad \textbf{open unfold } e \textbf{ as } \mathit{World}\ \mathsf{u} \textbf{ at } \mathtt{C} \\ \quad\quad \textbf{as } \langle \mathsf{tail}, x : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}\rangle \\ \quad\quad \textbf{in } \text{PACK}[\mathtt{A}; \mathsf{u}; \mathit{Rows}[\mathtt{C}, \mathtt{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}; x]\end{array}$$

Figure 6: Type-directed translation of FJ expressions.

PACK and UPCAST and six rules governing the judgment $\text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e}] = e$ for term translation. Γ is the FJ type environment, u is the universal sum type, classes is a record containing the runtime representations of each class, e is an FJ expression, and e is its corresponding term in the target language. If e has type C, then its translation e should have type $(\mathit{World}\ \mathsf{u}) \cdot \mathtt{C}$.

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 $\mathit{ktail}[\mathtt{C}]$ and e has type $(\mathit{SelfTy}[\mathtt{C}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})$, the term $\text{PACK}[\mathtt{C}; \mathsf{u}; \mathsf{tail}; e]$ has type $\mathsf{w} \cdot \mathtt{C}$. The UPCAST macro unfolds and repackages an object term to a term of some super class. When $\mathtt{C} \mathtt{<:} \mathtt{A}$ and e has type $\mathsf{w} \cdot \mathtt{C}$, the term $\text{UPCAST}[\mathtt{C}; \mathtt{A}; \mathsf{u}; e]$ has type $\mathsf{w} \cdot \mathtt{A}$. These macros simply and effectively formalize the encoding techniques demonstrated in the previous section. They employ erasable type manipulations only. Note the use of $\mathit{Rows}[\mathtt{C}, \mathtt{A}]$ 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 $\Gamma \vdash \mathtt{e} \in \mathtt{D}$ and $\mathtt{D} \mathtt{<:} \mathtt{C}$) 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 $\mathtt{e.m(e}_1 \ldots \mathtt{e}_n\mathtt{)}$ 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 types that are subclasses of the types in the method signature. Since our encoding does not utilize subtyping, the function selected from the vtable expects arguments of precisely the types in the method signature. Therefore, we must explicitly upcast all arguments. Rule (INVOKE) formalizes the self application technique demonstrated earlier.

The code to create a new object of class C essentially selects and applies C's constructor from the classes record. Until we explain class encoding and linking, the type of classes will be difficult to justify. Presently it will suffice to say that classes.C 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 dyncast pseudo-method in the next section. For now it is enough to treat dyncast as a black box—a polymorphic function with type $\forall\alpha.\,(\mathsf{u} \rightarrow \textbf{maybe}\ \alpha) \rightarrow \textbf{maybe}\ \alpha$. The argument of dyncast $[\mathit{ObjTy}[\mathtt{C}]\ \mathsf{w}\ \mathsf{u}]$ is a *projection* function, attempting to convert a value of type u to an object of type $\mathit{ObjTy}[\mathtt{C}]\ \mathsf{w}\ \mathsf{u}$. In addition to the new function, the classes record contains a proj field for each class C, of type $\mathsf{u} \rightarrow \textbf{maybe}\ (\mathit{ObjTy}[\mathtt{C}]\ \mathsf{w}\ \mathsf{u})$. Thus if we select the dyncast 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 ClassCast exception.

The expression translation judgment EXP *preserves types*. Informally, if e has type C then its translation has type (*World* u)·C (for some type u). To state this property formally, we must first translate all the types in the FJ typing environment Γ:

$$\mathcal{E}nv[\mathtt{u};\ \Gamma, \mathtt{x} : \mathtt{D}] = \mathcal{E}nv[\mathtt{u};\ \Gamma], \mathtt{x} : (\mathit{World}\ \mathtt{u})\cdot\mathtt{D}$$
$$\mathcal{E}nv[\mathtt{u};\ \circ] = \circ$$

By inspection, it is easy to show that $\mathcal{E}nv[\mathtt{u};\ \Gamma]$ is a well-formed environment, assuming that the range of Γ 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 $\Phi \vdash \mathtt{u} :: \mathsf{Type}$, $\Phi; \Delta \vdash \mathsf{classes} : \{\mathit{Classes}\ (\mathit{World}\ \mathtt{u})\}$ and $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ then $\Phi; \Delta, \mathcal{E}nv[\mathtt{u};\ \Gamma] \vdash \mathrm{EXP}[\Gamma; \mathtt{u}; \mathsf{classes}; \mathtt{e}] : (\mathit{World}\ \mathtt{u})\cdot\mathtt{C}$.

Proof: by induction on the structure of e. All cases are straightforward if we factor out and prove several properties as lemmas. First, we must establish a correspondence between the *fields* used in the FJ semantics and the *fieldvec* relation used for object layout (likewise between *mtype* and *methvec*). Second, we must establish the correspondence between pairs in *fieldvec* (or *methvec*) and elements in *Rows*. All these correspondences are proved by induction on the class hierarchy. Finally, we must show that the PACK and 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 *Rows*[A, ⊤] w u (*Rows*[C, A] w u tail) must be equivalent to *Rows*[C, ⊤] w u tail. This is proved by induction on the derivation of C <: 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:

$$\mathsf{dictPT} = \mathbf{\Lambda}\mathsf{tail}::\mathit{ktail}[\mathtt{PT}].\ \{\mathsf{getx} = \boldsymbol{\lambda}\mathsf{self} : s_{pt}.\ (\mathbf{unfold}\ \mathsf{self}).\mathsf{x}\}$$

$$\text{where } s_{pt} = \mu\alpha.\ \{\mathsf{vtab} : \{\mathsf{getx} : \alpha \to \mathbf{int};\ \mathsf{tail}\cdot\mathsf{m}\ \alpha\};\ \mathsf{x} : \mathbf{int};\ \mathsf{tail}\cdot\mathsf{f}\}$$

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.*, *Empty*[PT]), this dictionary becomes a vtable for class Point. Suppose the ColorPoint subclass inherits getx and adds a method of its own. Its dictionary would be:

$$\mathsf{dictCP} = \mathbf{\Lambda}\mathsf{tail}::\mathit{ktail}[\mathtt{CP}].\ \{\mathsf{getx} = (\mathsf{dictPT}\ [r_{cp}]).\mathsf{getx},\ \mathsf{getc} = \boldsymbol{\lambda}\mathsf{self} : s_{cp}.\ (\mathbf{unfold}\ \mathsf{self}).\mathsf{c}\}$$

$$\text{where } r_{cp} = \mathit{Rows}[\mathtt{CP}, \mathtt{PT}]\ (\mathit{World}\ \mathtt{u})\ \mathtt{u}\ \mathit{Empty}[\mathtt{CP}]$$
$$\text{and } s_{cp} = \mu\alpha.\ \{\mathsf{vtab} : \{\mathsf{getx} : \alpha \to \mathbf{int};\ \mathsf{getc} : \alpha \to \mathbf{color};\ \mathsf{tail}\cdot\mathsf{m}\ \alpha\};\ \mathsf{x} : \mathbf{int};\ \mathsf{c} : \mathbf{color};\ \mathsf{tail}\cdot\mathsf{f}\}$$

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 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 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. Somehow 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 Tag α equals another tag of known type Tag τ, 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

$$\begin{aligned}
\mathit{Dict}[\mathtt{C}] &\equiv \lambda\mathsf{w}.\ \lambda\mathsf{u}.\ \lambda\mathsf{self}.\ \{(\mathit{Rows}[\mathtt{C}, \top]\ \mathsf{w}\ \mathsf{u}\ \mathit{Empty}[\mathtt{C}])\cdot\mathsf{m}\ \mathsf{self}\}\\
\mathit{Ctor}[\mathtt{C}] &\equiv \lambda\mathsf{w}.\ \mathsf{w}\cdot\mathtt{D}_1 \to \ldots \mathsf{w}\cdot\mathtt{D}_n \to \mathsf{w}\cdot\mathtt{C}\\
&\quad \text{where } \mathit{fields}(\mathtt{C}) = \mathtt{D}_1\ \mathtt{f}_1 \ldots \mathtt{D}_n\ \mathtt{f}_n\\
\mathit{Proj}[\mathtt{C}] &\equiv \lambda\mathsf{w}.\ \lambda\mathsf{u}.\ \mathsf{u} \to \mathbf{maybe}\ \mathsf{w}\cdot\mathtt{C}\\
\mathit{Inj}[\mathtt{C}] &\equiv \lambda\mathsf{w}.\ \lambda\mathsf{u}.\ \mathsf{w}\cdot\mathtt{C} \to \mathsf{u}\\
\mathit{Class}[\mathtt{C}] &\equiv \lambda\mathsf{w}.\ \lambda\mathsf{u}.\\
&\quad \{\mathsf{dict} : \forall\mathsf{tail}.\ \mathit{Dict}[\mathtt{C}]\ \mathsf{w}\ \mathsf{u}\ (\mathit{SelfTy}[\mathtt{C}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}),\\
&\quad\ \mathsf{proj} : \mathit{Proj}[\mathtt{C}]\ \mathsf{w}\ \mathsf{u},\\
&\quad\ \mathsf{new} : \mathit{Ctor}[\mathtt{C}]\ \mathsf{w}\}\\
\\
\mathit{Classes} &\equiv \lambda\mathsf{w}.\ \lambda\mathsf{u}.\ ((\mathtt{E} : \mathbf{1} \to \mathit{Class}[\mathtt{E}]\ \mathsf{w}\ \mathsf{u};)^{\mathtt{E} \in cn}\ \mathsf{Abs}^{cn})\\
\mathit{ClassF}[\mathtt{C}] &\equiv \forall\mathsf{u}.\ \mathit{Inj}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u} \to \mathit{Proj}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u} \to\\
&\quad \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\} \to\\
&\quad \mathbf{1} \to \mathit{Class}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\\
\\
\mathit{Tagged} &\equiv \lambda\mathsf{u}.\ [(\mathtt{C} : \mathit{ObjTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u})^{\mathtt{C} \in cn}]
\end{aligned}$$

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

Class declaration translation:

$$\begin{array}{l}
\textsc{cdec}[\texttt{C}] = \\
\quad \boldsymbol{\Lambda}\texttt{u}{::}\textsf{Type}.\ \boldsymbol{\lambda}\texttt{inj} : \mathit{Inj}[\texttt{C}]\ (\mathit{World}\ \texttt{u})\ \texttt{u}.\ \boldsymbol{\lambda}\texttt{proj} : \mathit{Proj}[\texttt{C}]\ (\mathit{World}\ \texttt{u})\ \texttt{u}. \\
\quad \boldsymbol{\lambda}\texttt{classes} : \{\mathit{Classes}\ (\mathit{World}\ \texttt{u})\ \texttt{u}\}.\ \boldsymbol{\lambda}_ : \mathbf{1}. \\
\quad \textbf{let}\ \texttt{dict} : \forall \texttt{tail}{::}\mathit{ktail}[\texttt{C}].\ \mathit{Dict}[\texttt{C}]\ (\mathit{World}\ \texttt{u})\ \texttt{u} \\
\qquad\qquad\qquad\qquad (\mathit{SelfTy}[\texttt{C}]\ (\mathit{World}\ \texttt{u})\ \texttt{u}\ \texttt{tail}) \\
\qquad = \textsc{dict}[\texttt{C}; \texttt{u}; \texttt{inj}; \texttt{classes}] \\
\quad \textbf{in let}\ \texttt{vtab} = \texttt{dict}\ [\mathit{Empty}[\texttt{C}]] \\
\qquad \textbf{in}\ \{\texttt{dict} = \texttt{dict}, \texttt{proj} = \texttt{proj}, \texttt{new} = \textsc{new}[\texttt{C}; \texttt{u}; \texttt{vtab}]\}
\end{array}$$

Dictionary construction:

$$\begin{array}{l}
\textsc{dict}[\texttt{Obj}; \texttt{u}; \texttt{inj}; \texttt{classes}] = \\
\quad \boldsymbol{\Lambda}\texttt{tail}{::}\mathit{ktail}[\texttt{Obj}].\ \{\texttt{dyncast} = \\
\qquad\qquad \boldsymbol{\lambda}\texttt{self} : \mathit{SelfTy}[\texttt{C}]\ (\mathit{World}\ \texttt{u})\ \texttt{u}\ \texttt{tail}. \\
\qquad\qquad \boldsymbol{\Lambda}\alpha{::}\textsf{Type}.\ \boldsymbol{\lambda}\texttt{proj} : \texttt{u} \rightarrow \textbf{maybe}\ \alpha. \\
\qquad\qquad \texttt{proj}\ (\texttt{inj}\ \textsc{pack}[\texttt{Obj}; \texttt{u}; \texttt{tail}; \texttt{self}])\}
\end{array}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C} \lhd \texttt{B}\ \{\ldots\} \\ \mathrm{dom}(\mathit{methvec}(\texttt{C})) = [l_1 \ldots l_n] \end{array}}{\begin{array}{l}
\textsc{dict}[\texttt{C}; \texttt{u}; \texttt{inj}; \texttt{classes}] = \\
\quad \boldsymbol{\Lambda}\texttt{tail}{::}\mathit{ktail}[\texttt{C}]. \\
\quad \textbf{let}\ \texttt{super} : \mathit{Dict}[\texttt{B}]\ (\mathit{World}\ \texttt{u})\ \texttt{u} \\
\qquad\qquad (\mathit{SelfTy}[\texttt{C}]\ (\mathit{World}\ \texttt{u})\ \texttt{u}\ \texttt{tail}) \\
\qquad = (\texttt{classes.B}\ \{\})\texttt{.dict} \\
\qquad\quad [\mathit{Rows}[\texttt{C}, \texttt{B}]\ (\mathit{World}\ \texttt{u})\ \texttt{u}\ \texttt{tail}] \\
\quad \textbf{in}\ \{l_1 = \textsc{meth}[\texttt{C}; l_1; \texttt{u}; \texttt{tail}; \texttt{inj}; \texttt{classes}; \texttt{super}], \ldots, \\
\qquad l_n = \textsc{meth}[\texttt{C}; l_n; \texttt{u}; \texttt{tail}; \texttt{inj}; \texttt{classes}; \texttt{super}]\}
\end{array}}$$

Constructor code:

$$\frac{\mathit{fields}(\texttt{C}) = \texttt{D}_1\ \texttt{f}_1 \ldots \texttt{D}_n\ \texttt{f}_n}{\begin{array}{l}
\textsc{new}[\texttt{C}; \texttt{u}; \texttt{vtab}] = \\
\quad \boldsymbol{\lambda}\texttt{f}_1 : (\mathit{World}\ \texttt{u}){\cdot}\texttt{D}_1.\ \ldots\ \boldsymbol{\lambda}\texttt{f}_n : (\mathit{World}\ \texttt{u}){\cdot}\texttt{D}_n. \\
\quad\ \textbf{let}\ x = \textbf{fold}\ \{\texttt{vtab} = \texttt{vtab}, \texttt{f}_1 = \texttt{f}_1, \ldots, \texttt{f}_n = \texttt{f}_n\} \\
\qquad\qquad \textbf{as}\ \mathit{SelfTy}[\texttt{C}]\ (\mathit{World}\ \texttt{u})\ \texttt{u}\ \mathit{Empty}[\texttt{C}] \\
\quad\ \textbf{in}\ \textsc{pack}[\texttt{C}; \texttt{u}; \mathit{Empty}[\texttt{C}]; x]\}
\end{array}}$$

Method code:

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C} \lhd \texttt{B}\ \{\ldots \texttt{K}\ \texttt{M}_1 \ldots \texttt{M}_n\} \\ \texttt{m not defined in}\ \texttt{M}_1 \ldots \texttt{M}_n \end{array}}{\textsc{meth}[\texttt{C}; \texttt{m}; \texttt{u}; \texttt{tail}; \texttt{inj}; \texttt{classes}; \texttt{super}] = \texttt{super.m}}$$

$$\begin{array}{l}
\textsc{meth}[\texttt{C}; \texttt{m}; \texttt{u}; \texttt{tail}; \texttt{inj}; \texttt{classes}; \texttt{super}] = \\
\quad \textsc{meth}[\texttt{C}; \texttt{m}; \texttt{u}; \texttt{tail}; \texttt{inj}; \texttt{classes}; \texttt{super}] \\
\qquad \textbf{case}\ \texttt{proj}\ (\texttt{inj}\ \textsc{pack}[\texttt{C}; \texttt{u}; \texttt{tail}; \texttt{self}]) \\
\qquad\quad \textbf{of some}\ x \Rightarrow \textbf{some}\ [\alpha]\ x, \\
\qquad\qquad \textbf{else}\ \texttt{super.dyncast self}\ [\alpha]\ \texttt{proj}
\end{array}$$

$$\frac{\begin{array}{l} CT(\texttt{C}) = \texttt{class C} \lhd \texttt{B}\ \{\ldots \texttt{K}\ \texttt{M}_1 \ldots \texttt{M}_n\} \\ \exists j : \texttt{M}_j = \texttt{A m(A}_1\ \texttt{x}_1 \ldots \texttt{A}_m\ \texttt{x}_m)\ \{\hat{}\texttt{e;}\} \\ \Gamma = \texttt{x}_1{:}\texttt{A}_1, \ldots, \texttt{x}_m{:}\texttt{A}_m, \texttt{this:C} \\ \Gamma \vdash \texttt{e} \in \texttt{D} \qquad \texttt{D <: A} \\ \textsc{exp}[\Gamma; \texttt{u}; \texttt{classes}; \texttt{e}] = e \end{array}}{\begin{array}{l}
\textsc{meth}[\texttt{C}; \texttt{m}; \texttt{u}; \texttt{tail}; \texttt{inj}; \texttt{classes}; \texttt{super}] = \\
\quad \boldsymbol{\lambda}\texttt{self} : \mathit{SelfTy}[\texttt{C}]\ (\mathit{World}\ \texttt{u})\ \texttt{u}\ \texttt{tail}. \\
\quad \boldsymbol{\lambda}\texttt{x}_1 : (\mathit{World}\ \texttt{u}){\cdot}\texttt{A}_1.\ \ldots\ \boldsymbol{\lambda}\texttt{x}_m : (\mathit{World}\ \texttt{u}){\cdot}\texttt{A}_m. \\
\quad\ \textbf{let}\ \texttt{this} : (\mathit{World}\ \texttt{u}){\cdot}\texttt{C} = \textsc{pack}[\texttt{C}; \texttt{u}; \texttt{tail}; \texttt{self}] \\
\quad\ \textbf{in}\ \textsc{upcast}[\texttt{D}; \texttt{A}; \texttt{u}; e]
\end{array}}$$

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 maketag constrains the operational semantics, which is otherwise free of side effects. maketag 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 vtable 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 sum type and a 'one-armed' case. Let u be a sum type with a variant for each class in the class table. The function

$$\begin{array}{l}
\boldsymbol{\lambda}x : u.\ \textbf{case}\ x\ \textbf{of}\ \texttt{C}\ y \Rightarrow \textbf{some}\ [\mathit{ObjTy}[\texttt{C}]\ \texttt{w}\ \texttt{u}]\ y \\
\qquad\qquad \textbf{else none}\ [\mathit{ObjTy}[\texttt{C}]\ \texttt{w}\ \texttt{u}]
\end{array}$$

could dynamically represent class C. To connect it to the object type, we make the dyncast method polymorphic, with the type

$$\texttt{self} \rightarrow \forall\alpha.\ (u \rightarrow \textbf{maybe}\ \alpha) \rightarrow \textbf{maybe}\ \alpha$$

This method can check its own class against the target class by injecting self and applying the function argument. If the result is **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.

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

```
PROG[e] =
  let x_cn = LINK {(C = CDEC[C]) C∈cn}
   in EXP[∘; u; x_cn; e]
where u = μu::Type. Tagged u

LINK = λx:{(C:ClassF[C]) C∈cn}.
  fix [Classes (World u) u]
      (λclasses:{Classes (World u) u}.
         {(C = x.C [u] inj_C proj_C classes) C∈cn})
where u = μu::Type. Tagged u
    inj_C = λx:ObjTy[C] (World u) u. fold inj_C^{Tagged u} x as u
   proj_C = λx:u. case unfold x
                  of C y ⇒ some [ObjTy[C] (World u) u] y
                     else none [ObjTy[C] (World u) u]
```

Figure 9: Program translation and linking.

casts. Following it are two function parameters for injecting and projecting objects of class C. The next parameter is classes, a record containing definitions for other classes that are mutually recursive with C (for convenience, we assume that each class refers to all the others). The final parameter is of unit type; it simply delays references to classes so that linking terminates.

In the functor body, we define dict (using the macro DICT) and vtab (the trivial instantiation of dict). dict is placed in the class record (so subclasses can inherit its methods); vtab is passed to the NEW macro which creates the constructor code. The constructor is exported so that other classes can create C objects; and, finally, the projection function proj (a functor parameter) is exported so other classes can dynamically cast to C.

The dictionary for class Obj is hard-coded as DICT[Obj;...]. Its dyncast method injects self at class Obj, passes this to the proj argument and returns the result. If the class tags do not match, dyncast indicates failure by returning **none**; there is no super class to test. For all other classes, DICT fetches the super class dictionary from classes and instantiates it as super. It then uses METH to construct code for each method label in *methvec*.

METH supports three cases: it (1) produces the dyncast method (which must be overridden in every class), (2) inherits a method from the super class, or (3) constructs a new method body by translating FJ code.

Theorem 2 (Well-typed class declaration)
$\Phi; \Delta \vdash \text{CDEC}[\texttt{C}] : \mathit{ClassF}[\texttt{C}]$

Proof: by inspection. □

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_{cn} 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* 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_C and proj_C. The final argument to x.C is the classes record itself.

Theorem 3 (Well-typed linkage)
$\Phi; \Delta \vdash \text{LINK} : \{(\texttt{E} : \mathit{ClassF}[\texttt{E}])\ ^{\texttt{E}\in cn}\} \rightarrow \{\mathit{Classes}\ (\mathit{World}\ u)\ u\}$
where $u = \mu u::\text{Type}.\ \mathit{Tagged}\ u$

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().b. 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 β" where β is the abstract type. Likewise, from within class B, the type of method geta might be self⟶("A with private fields α"). The challenge is to allow class A to see that the α 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 argument.

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. Rémy and Vouillon [28] use row polymorphism in 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_ω-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_ω. 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 α 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 α. Crary encodes precisely the same property as an intersection type: $\exists\alpha.\,\alpha \wedge I(\alpha)$. 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

[1] Martín 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 *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 396–409, St. Petersburg, January 1996. ACM.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, October 1998. ACM.

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

[5] Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. In *Proc. Int'l Symp. on Theoretical Aspects of Computer Software (TACS)*, Sendai, Japan, September 1997. To appear in *Information and Computation*.

[6] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. Int'l*

Conf. on Functional Programming Languages and Computer Architecture, pages 273–280, London, September 1989. ACM.

[7] Karl Crary. Simple, efficient object encoding using intersection types. Technical Report CMU-CS-99-100, School of Computer Science, Carnegie Mellon University, Pittsburgh, January 1999.

[8] Karl Crary. Typed compilation of inclusive subtyping. In *Proc. Int'l Conf. on Functional Programming (ICFP)*, Montréal, September 2000. ACM.

[9] Jonathan Eifrig, Scott Smith, Valery Trifonov, and Amy Zwarico. An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation*, 8(4):357–397, 1995.

[10] Kathleen Fisher and John Mitchell. On the relationship between classes, objects and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–25, 1998.

[11] J. Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur*. PhD thesis, University of Paris VII, 1972.

[12] Neal Glew. *Low-Level Type Systems for Modularity and Object-Oriented Constructs*. PhD thesis, Cornell University, January 2000.

[13] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[14] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 130–141, San Francisco, January 1995. ACM.

[15] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[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.

[17] Samuel Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 80–87, San Diego, January 1988. ACM.

[18] Christopher League, Zhong Shao, and Valery Trifonov. Representing Java classes in a typed intermediate language. In *Proc. Int'l Conf. on Functional Programming (ICFP)*, pages 183–196, Paris, September 1999. ACM.

[19] Christopher League and Valery Trifonov. Comparing object encodings for typed intermediate languages. Technical report, Yale University, 2000. In preparation.

[20] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[21] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Proc. ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, May 1999. ACM.

[22] Greg Morrisett, David Tarditi, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. In *Proc. 1996 Workshop on Compiler Support for System Software (WCSSS)*, 1996.

[23] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[24] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. Second USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 229–243, Seattle, October 1996.

[25] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell Compiler: A technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT)*, Keele, December 1992.

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

[27] Didier Rémy. Syntactic theories and the algebra of record terms. Technical Report 1869, INRIA, 1993.

[28] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Paris, January 1997. ACM.

[29] John C. Reynolds. Towards a theory of type structure. In *Proc., Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

[30] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 116–129, La Jolla, June 1995. ACM.

[31] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Proc. Int'l Conf. on Functional Programming (ICFP)*, pages 313–323, Baltimore, September 1998. ACM.

[32] Andrew Wright, Suresh Jagannathan, Cristian Ungureanu, and Aaron Hertzmann. Compiling Java to a typed lambda-calculus: A preliminary report. In *Proc. Second Int'l Workshop on Types in Compilation (TIC98)*, volume 1473 of *Lecture Notes in Computer Science*, pages 1–14. Springer, March 1998.

A Featherweight Java semantics

Syntax:

$$\begin{array}{l}
\texttt{CL} ::= \texttt{class C} \lhd \texttt{C \{(C f;)}^* \texttt{ K M}^*\texttt{\}} \\
\texttt{K} ::= \texttt{C((C f)}^*\texttt{) \{super(f}^*\texttt{); (this.f = f;)}^*\texttt{\}} \\
\texttt{M} ::= \texttt{C m((C x)}^*\texttt{) \{\^{}e;\}} \\
\texttt{e} ::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m(e}^*\texttt{)} \mid \texttt{new C(e}^*\texttt{)} \mid \texttt{(C)e}
\end{array}$$

Field lookup:

$$\mathit{fields}(\texttt{Obj}) = \bullet$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C} \lhd \texttt{B \{C}_1\texttt{ f}_1\texttt{; ... C}_n\texttt{ f}_n\texttt{; K...\}} \quad \mathit{fields}(\texttt{B}) = \texttt{B}_1\texttt{ g}_1 \ldots \texttt{B}_m\texttt{ g}_m}{\mathit{fields}(\texttt{C}) = \texttt{B}_1\texttt{ g}_1 \ldots \texttt{B}_m\texttt{ g}_m\texttt{, C}_1\texttt{ f}_1 \ldots \texttt{C}_n\texttt{ f}_n}$$

Method lookup:

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C} \lhd \texttt{B \{...K M}_1 \ldots \texttt{M}_n\texttt{\}} \quad \exists j : \texttt{M}_j = \texttt{D m(D}_1\texttt{ x}_1 \ldots \texttt{D}_m\texttt{ x}_m\texttt{) \{\^{}e;\}}}{\begin{array}{l}\mathit{mtype}(\texttt{m}, \texttt{C}) = \texttt{D}_1 \ldots \texttt{D}_m\texttt{->D} \\ \mathit{mbody}(\texttt{m}, \texttt{C}) = (\texttt{x}_1 \ldots \texttt{x}_m, \texttt{e})\end{array}}$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C} \lhd \texttt{B \{...K M}_1 \ldots \texttt{M}_n\texttt{\}} \quad \texttt{m not defined in M}_1 \ldots \texttt{M}_n}{\begin{array}{l}\mathit{mtype}(\texttt{m}, \texttt{C}) = \mathit{mtype}(\texttt{m}, \texttt{B}) \\ \mathit{mbody}(\texttt{m}, \texttt{C}) = \mathit{mbody}(\texttt{m}, \texttt{B})\end{array}}$$

Valid method overriding:

$$\frac{\mathit{mtype}(\texttt{m}, \texttt{B}) = \texttt{C}_1 \ldots \texttt{C}_n\texttt{->C}_0}{\mathit{override}(\texttt{m}, \texttt{B}, \texttt{C}_1 \ldots \texttt{C}_n\texttt{->C}_0)}$$

$$\frac{\nexists T \text{ such that } \mathit{mtype}(\texttt{m}, \texttt{B}) = T}{\mathit{override}(\texttt{m}, \texttt{B}, \texttt{C}_1 \ldots \texttt{C}_n\texttt{->C}_0)}$$

Computation:

$$\frac{\mathit{fields}(\texttt{C}) = \texttt{D}_1\texttt{ f}_1 \ldots \texttt{D}_n\texttt{ f}_n}{\texttt{(new C(e}_1 \ldots \texttt{e}_n\texttt{)).f}_i \longrightarrow \texttt{e}_i} \quad \text{(R-Field)}$$

$$\frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\texttt{x}_1 \ldots \texttt{x}_n, \texttt{e}_0)}{\begin{array}{l}\texttt{(new C(e}_1 \ldots \texttt{e}_m\texttt{)).m(d}_1 \ldots \texttt{d}_n\texttt{)} \longrightarrow \\ \texttt{[d}_1\texttt{/x}_1, \ldots, \texttt{d}_n\texttt{/x}_n\texttt{, new C(e}_1 \ldots \texttt{e}_m\texttt{)/this] e}_0\end{array}} \quad \text{(R-Invk)}$$

$$\frac{\texttt{C <: D}}{\texttt{(D)new C(e}_1 \ldots \texttt{e}_n\texttt{)} \longrightarrow \texttt{new C(e}_1 \ldots \texttt{e}_n\texttt{)}} \quad \text{(R-Cast)}$$

Subtyping:

$$\texttt{C <: C}$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C} \lhd \texttt{B \{...\}} \qquad \texttt{B <: A}}{\texttt{C <: A}}$$

Class typing:

$$\frac{\begin{array}{l}\texttt{K = C(B}_1\texttt{ g}_1 \ldots \texttt{B}_n\texttt{ g}_n\texttt{, C}_1\texttt{ f}_1 \ldots \texttt{C}_m\texttt{ f}_m\texttt{)} \\ \qquad \texttt{\{super(g}_1 \ldots \texttt{g}_n\texttt{);} \\ \qquad\ \texttt{this.f}_1\texttt{ = f}_1\texttt{; ... this.f}_m\texttt{ = f}_m\texttt{;\}} \\ \mathit{fields}(\texttt{B}) = \texttt{B}_1\texttt{ g}_1 \ldots \texttt{B}_n\texttt{ g}_n \\ \texttt{M}_i \text{ ok in C} \quad \forall i \in \{1 \ldots k\}\end{array}}{\texttt{class C} \lhd \texttt{B \{C}_1\texttt{ f}_1\texttt{; ... C}_m\texttt{ f}_m\texttt{; K M}_1 \ldots \texttt{M}_k\texttt{\}} \text{ ok}}$$

Method typing:

$$\frac{\texttt{x}_1 : \texttt{D}_1, \ldots, \texttt{x}_n : \texttt{D}_n, \texttt{this} : \texttt{C} \vdash \texttt{e} \in \texttt{E} \qquad \texttt{E <: D} \quad \mathit{CT}(\texttt{C}) = \texttt{class C} \lhd \texttt{B \{...\}}}{\texttt{D m(D}_1\texttt{ x}_1 \ldots \texttt{D}_n\texttt{ x}_n\texttt{) \{\^{}e;\}} \text{ ok in C}}$$

Expression typing:

$$\Gamma \vdash \texttt{x} \in \Gamma(\texttt{x}) \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \texttt{e} \in \texttt{C} \qquad \mathit{fields}(\texttt{C}) = \texttt{D}_1\texttt{ f}_1 \ldots \texttt{D}_n\texttt{ f}_n}{\Gamma \vdash \texttt{e.f}_i \in \texttt{D}_i} \quad \text{(T-Field)}$$

$$\frac{\Gamma \vdash \texttt{e} \in \texttt{C} \qquad \mathit{mtype}(\texttt{m}, \texttt{C}) = \texttt{D}_1 \ldots \texttt{D}_n\texttt{->D} \quad \Gamma \vdash \texttt{e}_i \in \texttt{C}_i \quad \texttt{C}_i\texttt{ <: D}_i \quad (\forall i \in \{1 \ldots n\})}{\Gamma \vdash \texttt{e.m(e}_1 \ldots \texttt{e}_n\texttt{)} \in \texttt{D}} \quad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(\texttt{C}) = \texttt{D}_1\texttt{ f}_1 \ldots \texttt{D}_n\texttt{ f}_n \quad \Gamma \vdash \texttt{e}_i \in \texttt{C}_i \quad \texttt{C}_i\texttt{ <: D}_i \quad (\forall i \in \{1 \ldots n\})}{\Gamma \vdash \texttt{new C(e}_1 \ldots \texttt{e}_n\texttt{)} \in \texttt{C}} \quad \text{(T-New)}$$

$$\frac{\Gamma \vdash \texttt{e} \in \texttt{D} \qquad \texttt{D <: C}}{\Gamma \vdash \texttt{(C)e} \in \texttt{C}} \quad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \texttt{e} \in \texttt{D} \qquad \texttt{C <: D} \qquad \texttt{C} \neq \texttt{D}}{\Gamma \vdash \texttt{(C)e} \in \texttt{C}} \quad \text{(T-DCast)}$$

$$\frac{\Gamma \vdash \texttt{e} \in \texttt{D} \qquad \texttt{C} \not<: \texttt{D} \qquad \texttt{D} \not<: \texttt{C}}{\Gamma \vdash \texttt{(C)e} \in \texttt{C}} \quad \text{(T-SCast)}$$

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