Type-Preserving Compilation of Featherweight Java

CHRISTOPHER LEAGUE, ZHONG SHAO, and VALERY TRIFONOV
Yale University

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. Our work provides a foundation for supporting certifying compilation of Java-like class-based languages in a type-theoretic framework.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; F.3.3 [**Logic and Meanings of Programs**]: Studies of Program Constructs—*Object-Oriented Constructs*

General Terms: Languages, Verification

Additional Key Words and Phrases: Java, object encodings, type systems, typed intermediate languages

1. INTRODUCTION

Many compilation techniques for functional languages focus on type-directed compilation [Peyton Jones et al. 1992; Shao and Appel 1995; Morrisett et al. 1996]. 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 [Necula and Lee 1996; Necula 1997; Morrisett et al. 1999].

Type-preserving compilers typically use variants of the polymorphic typed λ-calculus F_ω [Girard 1972; Reynolds 1974] as their intermediate representations. Much is known about optimizing F_ω programs [Tarditi et al. 1996], about compiling them to machine code [Morrisett et al. 1999], and about implementing the F_ω type system efficiently in a production compiler [Shao et al. 1998].

Recently, several researchers have attempted to apply these techniques to object-oriented languages [Wright et al. 1998; Crary 1999; League et al. 1999; Vanderwaart 1999; Glew 2000a; League et al. 2001b]. While there is significant precedent for

This work was sponsored in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies. Authors' addresses: Department of Computer Science, Yale University, P.O. Box 208285, New Haven, CT 06511 USA; email: {league, shao, trifonov}@cs.yale.edu.

encoding object-oriented languages in typed λ-calculi [Canning et al. 1989; Bruce 1994; Eifrig et al. 1995; Abadi et al. 1996; Bruce et al. 1999], type-preserving compilation alters the requirements in some fundamental ways. The intermediate language must provide simple, orthogonal primitives that are amenable to optimization. If method invocation is an atomic primitive, for example, then we cannot safely optimize a sequence of calls on the same object. Furthermore, we must avoid introducing any dynamic overhead solely to achieve static typing. One can often, for example, simplify a type system by adding coercion functions or extra indirections, but these techniques have associated run-time penalties. The machinery used to achieve static typing should be *erasable,* meaning that it can be discarded after verification without affecting execution.

Given these constraints, the type system for the intermediate language should be as simple as possible. Type checking must be not only decidable, but efficient in practice. Typed compilation generally places greater demands on the implementation of a type system than does a simple type checker for a source language. On the other hand, at the level of the intermediate language, we can add more detailed and explicit type annotations than a source-level programmer might accept. With respect to object encodings, for example, *subsumption* is not necessarily required; it can be replaced with explicit coercions as long as their run-time cost is nil.

Finally, a type-preserving compiler should, where possible, maintain source-level abstractions. Source language type systems enforce certain abstractions (such as private fields and restricted interfaces) which could be eliminated in a translation without compromising type safety. This is dangerous if the translated code will be linked with other code, perhaps translated from a different source language. Link-time type checking will not prevent, for example, one module from accessing the private fields of another—unless the abstractions are preserved in the object code.

We have developed techniques for compiling a significant subset of Java into a simple and efficient typed intermediate language [League et al. 1999; 2001b]. Method invocation, after type erasure, has the same operational behavior as a standard implementation of self application using *vtables* (per-class tables of functions). Classes inherit or override methods from super classes with no overhead. By pairing an object with a particular *view* whenever it is cast to an interface type, interface calls are no more expensive than ordinary method calls. We support mutually recursive classes (at the type and term level) while still maintaining separate compilation. Dynamic casts and instance-of queries are implemented as polymorphic methods using tags generated at link-time. Private fields can be hidden from outsiders using existential types.

Ours is the first efficient encoding of a class-based language into F_ω without subtyping or bounded quantification. Glew [2000a] compiles a simple class-based calculus using F-bounded quantification. It is not known whether this feature is practical in a production compiler, since the type checker must *infer* derivations of the subtyping judgments. Fisher and Mitchell [1998] use extensible objects to model class constructs. For efficient implementation, though, these objects must be expressed using simpler primitives. Our intermediate representation uses simple, well-understood extensions: row polymorphism, existential, and recursive types. It is already implemented as part of the Standard ML of New Jersey compiler [Shao and Appel 1995; Shao 1997], and the new Java front end is in active develop-

```
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
```

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

```
class Point {
    int x;
    Point (int x)       { this.x = x; }
    int   getx ()       { ↑ this.x }
    Point move (int dx) { ↑ new Point (this.x + dx); }
    Point bump ()       { ↑ this.move (1); }
}

class ScaledPoint ◁ Point {
    int s;
    ScaledPoint (int x, int s)  { super(x); this.s = s; }
    int gets ()  { ↑ this.s }
    Point move (int dx)  { ↑ new ScaledPoint (this.x + this.s * dx, this.s); }
    ScaledPoint zoom (int s)  { ↑ new ScaledPoint (this.x, this.s * s); }
}
```

Fig. 2. Two classes in Featherweight Java, extended with integers and arithmetic.

ment [League et al. 2001a].

This paper focuses on a formal translation of programs in Featherweight Java (FJ) [Igarashi et al. 1999], a source calculus which models some of the salient features of Java (including classes, fields, methods, and dynamic cast). FJ is small enough to allow detailed proofs of interesting formal properties of the translation, such as type preservation. It also serves as an effective starting point for designing encodings of interesting extensions, such as genericity [Bracha et al. 1998], inner classes [Igarashi and Pierce 2001], and reflection.

We describe the syntax and semantics of the source and target languages in the next two sections. In section 4, we explain and formalize each aspect of our translation, ultimately proving that it is type-preserving. Section 5 discusses our strategies for implementing certain Java constructs which are not featured in FJ (such as interfaces and privacy). Finally, we contextualize our contribution with a survey of 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" [Igarashi et al. 1999]. FJ is small enough that perspicuous formal translation and detailed proofs are possible. Figure 1 contains the syntax of FJ; figure 2 illustrates some of the features of FJ with two sample classes.

Class declarations (CL) contain the names of the new class and its super class, a sequence of field declarations, a constructor (K), and a sequence of method dec-

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

Fig. 3. Syntax of the target language.

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

Fig. 4. Derived syntactic forms of the target language.

larations (`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 in FJ. Constructors always take *all* the fields as arguments, in the same order that they are declared in the class hierarchy. FJ permits recursive class dependencies with the full generality of Java. A class can refer to the name and constructor 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.

For reference, we reprint the semantics of FJ in appendix A. They begin by defining three relations. The subtype relation `<:` is the reflexive, transitive closure of the relation defined by the super class declarations (`class C` $\triangleleft$ `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 $\mathtt{D}_1 \ldots \mathtt{D}_n\texttt{->}\mathtt{D}_0$.

The expression typing rules govern judgments of the form $\Gamma \vdash \mathtt{e} \in \mathtt{C}$, 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 [Igarashi et al. 1999] for further explanation.

Pack and open for existential types:

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

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

Recursive record term:

$$\frac{\Phi; \Delta \vdash e : \{\tau\} \rightarrow \{\tau\}}{\Phi; \Delta \vdash \textsf{fix}\, [\tau]\ e : \{\tau\}} \tag{3}$$

Row and record types:

$$\frac{\vdash \Phi\ \mathit{kind\ env}}{\Phi \vdash \mathsf{Abs}^{L} :: \mathsf{R}^{L}} \tag{4}$$

$$\frac{\Phi \vdash \tau :: \mathsf{Type} \qquad \Phi \vdash \tau' :: \mathsf{R}^{L \cup \{l\}}}{\Phi \vdash l : \tau \,;\, \tau' :: \mathsf{R}^{L - \{l\}}} \tag{5}$$

$$\frac{\Phi \vdash \tau :: \mathsf{R}^{\emptyset}}{\Phi \vdash \{\tau\} :: \mathsf{Type}} \tag{6}$$

Sum type, its introduction and elimination:

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

$$\frac{\Phi \vdash [l_1 : \tau_1 \,;\, \ldots l_n : \tau_n \,;\, \tau] :: \mathsf{Type} \quad \Phi; \Delta \vdash e : \tau_i}{\Phi; \Delta \vdash \textsf{inj}_{l_i}^{[l_1 : \tau_1 \,;\, \ldots l_n : \tau_n \,;\, \tau]}\ e : [l_1 : \tau_1 \,;\, \ldots l_n : \tau_n \,;\, \tau]} \tag{8}$$

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

Fold and unfold for recursive types:

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \kappa \qquad \Phi \vdash \tau_s :: \kappa \Rightarrow \mathsf{Type} \qquad \Phi; \Delta \vdash e : \tau_s\ (\tau[\alpha := \mu\alpha{::}\kappa.\, \tau])}{\Phi; \Delta \vdash \textsf{fold}\ e\ \textsf{as}\ \mu\alpha{::}\kappa.\, \tau\ \textsf{at}\ \tau_s\ : \tau_s\ (\mu\alpha{::}\kappa.\, \tau)} \tag{10}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \kappa \qquad \Phi \vdash \tau_s :: \kappa \Rightarrow \mathsf{Type} \qquad \Phi; \Delta \vdash e : \tau_s\ (\mu\alpha{::}\kappa.\, \tau)}{\Phi; \Delta \vdash \textsf{unfold}\ e\ \textsf{as}\ \mu\alpha{::}\kappa.\, \tau\ \textsf{at}\ \tau_s : \tau_s\ (\tau[\alpha := \mu\alpha{::}\kappa.\, \tau])} \tag{11}$$

Fig. 5. 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.

3. TARGET LANGUAGE

The target language of our translation is the higher-order polymorphic λ-calculus F_ω [Girard 1972; Reynolds 1974] extended with existential types [Mitchell and Plotkin 1988], row polymorphism [Rémy 1993], ordered records, sum types, recursive types, and a term-level fixpoint for constructing recursive records. The syntax appears in figures 3 and 4. Typing rules for the non-standard features are given in figure 5; the remaining rules are in appendix B.

F_ω is an explicitly-typed calculus, with type annotations on function arguments and type applications for instantiating polymorphic functions. Type is the base kind of types which classify terms. The arrow kind $\kappa \Rightarrow \kappa'$ classifies type functions. A polymorphic array constructor, for example, would have kind Type⇒Type. The form $\lambda\alpha{::}\kappa.\, \tau$ introduces the arrow kind, and $\tau\ \tau'$ eliminates it. That is, $(\lambda\alpha{::}\kappa.\, \tau)\ \tau'$ is well-formed if τ' has kind κ. It is equivalent to $\tau[\alpha := \tau']$, which denotes the capture-avoiding substitution of τ' for α in τ. Labeled tuples of types are enclosed in braces $\{l = \tau \ldots\}$ and have tuple kinds $\{\tau :: \kappa \ldots\}$. The mid-dot syntax $\tau{\cdot}l$ denotes selection of a type from a tuple.

The single arrow $\tau \rightarrow \tau'$ is the type of a function expecting an argument of type τ and returning a result of type τ'. Our implementation supports multi-argument functions, but in this presentation, for simplicity, we simulate them using *curried* arguments ($\textsf{int} \rightarrow \textsf{int} \rightarrow \textsf{int}$). Anonymous functions are written using the boldface lambda, as in $\boldsymbol{\lambda} x \mathbin{\mathbf{:}} \tau.\, e$. Juxtaposition of terms ($e\ e'$) indicates applying function e to argument e'. Polymorphic functions are introduced by the capital lambda ($\boldsymbol{\Lambda}\alpha{::}\kappa.\, e$) which binds α in e. This term has type $\forall\alpha{::}\kappa.\, \tau$, where e has type τ and α may appear in τ. Thus, the polymorphic identity function is written as $\mathsf{id} = \boldsymbol{\Lambda}\alpha{::}\mathsf{Type}.\, \lambda x{::}\alpha.\, x$ and has type $\forall\alpha{::}\mathsf{Type}.\, \alpha \rightarrow \alpha$. An application of id to the integer 3 is written $\mathsf{id}\ [\textsf{int}]\ 3$.

The term **abort** $[\tau]$ represents a runtime error that otherwise would have produced a value of type τ. We use this to model a failed dynamic cast. In the operational semantics, evaluating **abort** $[\tau]$ produces an infinite loop. In a real system, **abort** would correspond to throwing an exception.

Following Rémy [1993] 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 $l \mathbin{\mathbf{:}} \tau \mathbin{\mathbf{;}} \tau'$ prepends a field with label l and type τ onto the row τ'. The row formation rules (4) and (5) prohibit duplicate labels: a type variable α of kind $\mathsf{R}^{\{\mathsf{m}\}}$ cannot be instantiated with a row in which the label m is already bound. Boldface braces $\boldsymbol{\{}\cdot\boldsymbol{\}}$ denote the type constructor for records; it lifts a complete row type (of kind $\mathsf{R}^{\emptyset}$) to kind Type. Record terms are written as a sequence of bindings in braces: $\boldsymbol{\{} l_1 = e\mathbf{,}\ l_2 = e\mathbf{,} \boldsymbol{\}}$. Permutations of rows are *not* considered equivalent—the labels are used only for readability. This means that record selection $e\mathbf{.}l$ can be compiled using offsets which are known at compile-time. We sometimes use commas and omit Abs^L when specifying complete rows (see the derived forms in figure 4). We let $\mathbf{1}$ (read 'unit') denote the empty record type. Row kinds can be used to encode functions which are polymorphic over the *tail* of a record argument. For example, the function $\boldsymbol{\Lambda}\rho{::}\mathsf{R}^{\{l\}}.\, \boldsymbol{\lambda} x \mathbin{\mathbf{:}} \boldsymbol{\{} l \mathbin{\mathbf{:}} \mathsf{string} \mathbin{\mathbf{;}} \rho \boldsymbol{\}}.\, \mathsf{print}\ x\mathbf{.}l$ can be instantiated and applied to any record which contains a string l as its first field.

Existential types ($\exists\alpha{::}\kappa.\, \tau$) support abstraction by *hiding* a witness type. They are introduced at the term level by a *package* $\langle \alpha{::}\kappa = \tau\mathbf{,}\ e \mathbin{\mathbf{:}} \tau' \rangle$, where τ is the witness type (of kind κ) and e has type $\tau'[\alpha := \tau]$. The existential is eliminated (within a restricted scope) by **open**; see rules (1) and (2).

Labeled sum types are constructed by enclosing a complete row within boldface brackets: $\boldsymbol{[}\cdot\boldsymbol{]}$. Sum types are introduced by a term-level injection and eliminated by an ML-like case statement; see rules (8) and (9). Figure 4 defines a parameterized type **maybe** with constructors **some** and **none**.

Recursive types are mediated by explicit *fold* and *unfold* terms. These so-called *iso-recursive* types (a term first used by Crary et al. [1999]) simplify type checking, but are less flexible than *equi-recursive* types unless the calculus is equipped with a definedness logic for coercions [Abadi and Fiore 1996]. Since we use recursive types at higher kinds, the syntax for folding and unfolding them deserves some explanation. Suppose we wish to encode the following mutually recursive type abbreviations:

$$\textbf{type}\ \mathsf{even} = \textbf{maybe}\ \boldsymbol{\{}\mathsf{hd}\,\boldsymbol{:}\,\textbf{int}\boldsymbol{,}\,\mathsf{tl}\,\boldsymbol{:}\,\mathsf{odd}\boldsymbol{\}}$$
$$\textbf{type}\ \mathsf{odd} = \boldsymbol{\{}\mathsf{hd}\,\boldsymbol{:}\,\textbf{int}\boldsymbol{,}\,\mathsf{tl}\,\boldsymbol{:}\,\mathsf{even}\boldsymbol{\}}$$

The solution is expressed as the fixpoint over a tuple:

$$\begin{aligned} t = \mu\alpha{::}\{&\mathsf{even} :: \mathsf{Type},\ \mathsf{odd} :: \mathsf{Type}\}.\\ &\{\mathsf{even} = \textbf{maybe}\ \boldsymbol{\{}\mathsf{hd}\,\boldsymbol{:}\,\textbf{int}\boldsymbol{,}\,\mathsf{tl}\,\boldsymbol{:}\,\alpha{\cdot}\mathsf{odd}\boldsymbol{\}},\\ &\ \ \mathsf{odd} = \boldsymbol{\{}\mathsf{hd}\,\boldsymbol{:}\,\textbf{int}\boldsymbol{,}\,\mathsf{tl}\,\boldsymbol{:}\,\alpha{\cdot}\mathsf{even}\boldsymbol{\}}\} \end{aligned}$$

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 coerced to a type in which t is *unfolded*. We allow unfolding of recursive types within a tuple by specifying a *selector* after the **at** keyword. Selectors are syntactically restricted to a (possibly empty) sequence of labeled selections from a tuple. The syntax $\lambda\gamma{::}\kappa.\, s[\gamma]$ allows identity $(\lambda\gamma{::}\kappa.\, \gamma)$, one selection $(\lambda\gamma{::}\kappa.\, \gamma{\cdot}l_1)$, two selections $(\lambda\gamma{::}\kappa.\, \gamma{\cdot}l_1{\cdot}l_2)$, and so on. The formation rules (10) and (11) further restrict the selectors to have a result of kind Type. Thus, if e has type $t{\cdot}\mathsf{odd}$, then the expression

$$\textbf{unfold}\ e\ \textbf{as}\ t\ \textbf{at}\ \lambda\gamma{::}\{\mathsf{even} :: \mathsf{Type},\ \mathsf{odd} :: \mathsf{Type}\}.\ \gamma{\cdot}\mathsf{odd}$$

has type $\boldsymbol{\{}\mathsf{hd}\,\boldsymbol{:}\,\textbf{int}\boldsymbol{,}\,\mathsf{tl}\,\boldsymbol{:}\,t{\cdot}\mathsf{even}\boldsymbol{\}}$. For recursive types of kind Type, the only allowed selector is identity, so we omit it. We sometimes also omit the **as** annotation where it can be readily inferred.

The typing judgments are decidable, and the type system is sound with respect to a structured operational semantics. We sketch the decidability proof in section B.2, and give a detailed soundness proof in section B.4. The target language also enjoys a type erasure property: type manipulations (*e.g.,* type abstractions, folds, pack, and open) can be erased before runtime without affecting the result.

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 by describing and formalizing our basic object encoding in sections 4.1 and 4.2. In section 4.3, we give a type-directed translation of FJ expressions. Inheritance, overriding, and constructors are examined as part of the class encoding in section 4.4, formalized in section 4.5. Finally, section 4.6 covers linking and section 4.7 discusses separate compilation. 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 uses *self-application* [Kamin 1988]. In a class-based language, the object record contains values for all the fields of the object plus a pointer to a record of methods, called the *vtable*. The vtable of a class is created once and shared among all objects of the 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

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

Happily, by unfolding the argument self and folding the object we obtain the term

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

which is well-typed, as is the augmented self-application term $(\mathbf{unfold}\ p_1).\mathsf{vtab}.\mathsf{getx}\ p_1$.

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

$$\tau_{sp} = \mu\mathsf{self}{::}\mathsf{Type}.\ \{\mathsf{vtab} : \{\mathsf{getx} : \mathsf{self} \rightarrow \mathbf{int},\ \mathsf{gets} : \mathsf{self} \rightarrow \mathbf{int}\},\ \mathsf{x} : \mathbf{int},\ \mathsf{s} : \mathbf{int}\}$$

How can we relate the types for objects of these two classes? More to the point, how can we make a function expecting a Point accept a ScaledPoint? Traditional models employ subsumption, but τ_{sp} is not a subtype of τ_{pt}, so other arrangements must be made. Can the subclass relationship be encoded using explicit (but erasable) type manipulations?

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 type of the object record is known; the rest is hidden, abstract. Consider this static type of a Point object:

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

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-empty 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 type τ'_{pt} using the trivial witness type $\{\mathsf{f} = \mathsf{Abs}^{\{\mathsf{vtab},\mathsf{x}\}},\ \mathsf{m} = \lambda\mathsf{s}{::}\mathsf{Type}.\,\mathsf{Abs}^{\{\mathsf{getx}\}}\}$. To package an object of dynamic class ScaledPoint into type τ'_{pt} we hide a non-trivial witness type, containing the new field and method:

$$\{\mathsf{f} = (\mathsf{s}\,\mathbf{:}\,\mathbf{int}\,\mathbf{;}\,\mathsf{Abs}^{\{\mathsf{vtab,x,s}\}}),\\ \mathsf{m} = \lambda\mathsf{self}{::}\mathsf{Type}.\,(\mathsf{gets}\,\mathbf{:}\,\mathsf{self}{\longrightarrow}\mathbf{int}\,\mathbf{;}\,\mathsf{Abs}^{\{\mathsf{getx,gets}\}})\}$$

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:

$$\mu\mathsf{twin}.\,\exists\mathsf{tail}.\,\mu\mathsf{self}.\,\{\mathsf{vtab}\,\mathbf{:}\,\{\mathsf{getx}\,\mathbf{:}\,\mathsf{self}{\longrightarrow}\mathbf{int}\,\mathbf{;}\,\mathsf{bump}\,\mathbf{:}\,\mathsf{self}{\longrightarrow}\mathsf{twin}\,\mathbf{;}\,\mathsf{tail}{\cdot}\mathsf{m}\ \mathsf{self}\}\mathbf{;}\\ \mathsf{x}\,\mathbf{:}\,\mathbf{int}\,\mathbf{;}\ \mathsf{tail}{\cdot}\mathsf{f}\}$$

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 can be distinct from that of self.

We used this technique in [League et al. 1999] 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

$$\mathsf{AB} \equiv \mu\mathsf{w}{::}\{\mathsf{A} :: \mathsf{Type},\ \mathsf{B} :: \mathsf{Type}\}.\\ \{\mathsf{A} = \exists\mathsf{tail}{::}\mathsf{Type}{\Rightarrow}\mathsf{R}^{\{\mathsf{getb}\}}.\,\mu\mathsf{self}{::}\mathsf{Type}.\,\{\mathsf{getb}\,\mathbf{:}\,\mathsf{self}{\longrightarrow}\mathsf{w}{\cdot}\mathsf{B}\,\mathbf{;}\,\mathsf{tail}\ \mathsf{self}\},\\ \mathsf{B} = \exists\mathsf{tail}{::}\mathsf{Type}{\Rightarrow}\mathsf{R}^{\{\mathsf{geta}\}}.\,\mu\mathsf{self}{::}\mathsf{Type}.\,\{\mathsf{geta}\,\mathbf{:}\,\mathsf{self}{\longrightarrow}\mathsf{w}{\cdot}\mathsf{A}\,\mathbf{;}\,\mathsf{tail}\ \mathsf{self}\}\}$$

Using the contextual fold and 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 4.4) as well as interactions with privacy—see section 5.

4.2 Type translation

This completes our informal account of the object encoding; we now turn to a formal translation of FJ types. Figure 6 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 $\mathrm{dom}(s)$ 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 $(\mathtt{f}, \mathtt{D})$, 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

$$\mathit{fieldvec}(\texttt{Obj}) = [(\mathsf{vtab}, \mathit{vt})] \tag{12}$$

$$\frac{CT(\texttt{C}) = \texttt{class C} \lhd \texttt{B \{}\texttt{D}_1\ \texttt{f}_1\texttt{;} \ldots \texttt{D}_m\ \texttt{f}_m\texttt{; K}\ldots\texttt{\}}}{\mathit{fieldvec}(\texttt{C}) = \mathit{fieldvec}(\texttt{B}) +\!\!+ [(\texttt{f}_1, \texttt{D}_1)\ldots(\texttt{f}_m, \texttt{D}_m)]} \tag{13}$$

$$\mathit{methvec}(\texttt{Obj}) = [(\mathsf{dyncast}, \mathit{dc})] \tag{14}$$

$$\frac{CT(\texttt{C}) = \texttt{class C} \lhd \texttt{B \{}\ldots\texttt{K M}_1\ldots\texttt{M}_m\texttt{\}}}{\mathit{methvec}(\texttt{C}) = \mathit{methvec}(\texttt{B}) +\!\!+ \mathit{addmeth}(\texttt{B}, [\texttt{M}_1\ldots\texttt{M}_m])} \tag{15}$$

$$\frac{(\texttt{m}, \_) \in \mathit{methvec}(\texttt{B})}{\mathit{addmeth}(\texttt{B}, [\texttt{D m(D}_1\ \texttt{x}_1\ldots\texttt{D}_k\ \texttt{x}_k\texttt{) \{}\ldots\texttt{\} M}_2\ldots\texttt{M}_m]) = \mathit{addmeth}(\texttt{B}, [\texttt{M}_2\ldots\texttt{M}_m])} \tag{16}$$

$$\frac{(\texttt{m}, \_) \notin \mathit{methvec}(\texttt{B})}{\begin{array}{l}\mathit{addmeth}(\texttt{B}, [\texttt{D m(D}_1\ \texttt{x}_1\ldots\texttt{D}_k\ \texttt{x}_k\texttt{) \{}\ldots\texttt{\} M}_2\ldots\texttt{M}_m]) = \\ \quad [(\texttt{m}, \texttt{D}_1\ldots\texttt{D}_k\texttt{->D})] +\!\!+ \mathit{addmeth}(\texttt{B}, [\texttt{M}_2\ldots\texttt{M}_m])\end{array}} \tag{17}$$

$$\mathit{addmeth}(\texttt{B}, []) = [] \tag{18}$$

Fig. 6. Field and method layouts for object types.

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 $(\texttt{m}, T)$, indicating a method named `m` with signature T. Signatures have the form $\texttt{D}_1\ldots\texttt{D}_n\texttt{->D}$. 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 $(\mathsf{dyncast}, \mathit{dc})$, a pseudo-method used to implement dynamic casts.

Let cn denote the set of class names in the program of interest, including `Obj`. For the purpose of presentation, we abbreviate the kind of a tuple of all object types as $k\,cn$. The tuple of row kinds for class `C` is abbreviated $\mathit{ktail}[\texttt{C}]$.

$$\begin{array}{ll} k\,cn & \equiv \{(\texttt{E} :: \mathsf{Type})^{\,\texttt{E}\in cn}\} \\ \mathit{ktail}[\texttt{C}] & \equiv \{\mathsf{m} :: \mathsf{Type}{\Rightarrow}\mathsf{R}^{\mathrm{dom}(\mathit{methvec}(\texttt{C}))}, \mathsf{f} :: \mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\texttt{C}))}\} \end{array}$$

For brevity, we sometimes omit kind annotations. By convention, certain named type variables are bound by particular kinds—w has kind $k\,cn$, self and u have kind Type, and tail has kind $\mathit{ktail}[\texttt{C}]$, where `C` should be evident from the context.

In figure 7 we define *Rows*, a type operator that produces rows containing the fields and methods introduced *between* two classes in a subclass relationship. Intuitively, $\mathit{Rows}[\texttt{C}, \texttt{A}]$ 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 $\mathit{Rows}[\texttt{C}, \texttt{A}]$ has kind $k\,cn{\Rightarrow}\mathsf{Type}{\Rightarrow}\mathit{ktail}[\texttt{C}]{\Rightarrow}\mathit{ktail}[\texttt{A}]$. Its first argument, $\mathsf{w}{::}k\,cn$, is a tuple containing object types for all classes in the compilation unit. The next argument, $\mathsf{u}{::}\mathsf{Type}$, is a *universal* type used to implement dynamic

$$Rows[\mathtt{C},\mathtt{C}] = \lambda\mathsf{w}::kcn.\ \lambda\mathsf{u}::\mathsf{Type}.\ \lambda\mathsf{tail}::ktail[\mathtt{C}].\ \mathsf{tail} \tag{19}$$

$$\begin{array}{l} Rows[\mathtt{Obj},\top] = \lambda\mathsf{w}::kcn.\ \lambda\mathsf{u}::\mathsf{Type}.\ \lambda\mathsf{tail}::ktail[\mathtt{Obj}].\\ \quad \{\mathsf{m} = \lambda\mathsf{self}::\mathsf{Type}.\ (\mathsf{dyncast}\,\mathbf{:}\,\mathsf{self}\longrightarrow\forall\alpha::\mathsf{Type}.\ (\mathsf{u}\longrightarrow\mathbf{maybe}\ \alpha)\longrightarrow\mathbf{maybe}\ \alpha\,\mathbf{;}\,\mathsf{tail}\cdot\mathsf{m}\ \mathsf{self})\\ \quad\ \mathsf{f}\ = \mathsf{tail}\cdot\mathsf{f}\} \end{array} \tag{20}$$

$$\frac{\begin{array}{l} CT(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{B}\ \{\mathtt{D}_1\ \mathtt{f}_1 \ldots \mathtt{D}_n\ \mathtt{f}_n\ \mathtt{K}\ \mathtt{M}_1 \ldots \mathtt{M}_m\}\\ Rows[\mathtt{B},\mathtt{A}] = \tau \qquad addmeth(\mathtt{B},[\mathtt{M}_1 \ldots \mathtt{M}_m]) = [(l_1,T_1)\ldots(l_m,T_m)] \end{array}}{\begin{array}{l} Rows[\mathtt{C},\mathtt{A}] = \lambda\mathsf{w}::kcn.\ \lambda\mathsf{u}::\mathsf{Type}.\ \lambda\mathsf{tail}::ktail[\mathtt{C}].\\ \quad \tau\ \mathsf{w}\ \mathsf{u}\ \{\mathsf{m} = \lambda\mathsf{self}::\mathsf{Type}.\ (l_1\,\mathbf{:}\,Ty[\mathsf{self};\mathsf{w};T_1]\,\mathbf{;}\ \ldots l_m\,\mathbf{:}\,Ty[\mathsf{self};\mathsf{w};T_m]\,\mathbf{;}\,\mathsf{tail}\cdot\mathsf{m}\ \mathsf{self}),\\ \qquad\quad \mathsf{f}\ = (\mathtt{f}_1\,\mathbf{:}\,\mathsf{w}\cdot\mathtt{D}_1\,\mathbf{;}\ \ldots \mathtt{f}_n\,\mathbf{:}\,\mathsf{w}\cdot\mathtt{D}_n\,\mathbf{;}\,\mathsf{tail}\cdot\mathsf{f})\} \end{array}} \tag{21}$$

$$Ty[\mathsf{self};\mathsf{w};\mathtt{D}_1\ldots\mathtt{D}_n\texttt{->}\mathtt{D}] = \mathsf{self}\longrightarrow\mathsf{w}\cdot\mathtt{D}_1\longrightarrow\ldots\mathsf{w}\cdot\mathtt{D}_n\longrightarrow\mathsf{w}\cdot\mathtt{D} \tag{22}$$

$$\begin{array}{lll} Empty[\mathtt{C}] & \equiv & \{\mathsf{m} = \lambda\mathsf{self}::\mathsf{Type}.\ \mathsf{Abs}^{\mathrm{dom}(methvec(\mathtt{C}))}, \mathsf{f} = \mathsf{Abs}^{\mathrm{dom}(fieldvec(\mathtt{C}))}\}\\ ObjRcd[\mathtt{C}] & \equiv & \lambda\mathsf{w}::kcn.\ \lambda\mathsf{u}::\mathsf{Type}.\ \lambda\mathsf{tail}::ktail[\mathtt{C}].\ \lambda\mathsf{self}::\mathsf{Type}.\\ & & \quad \{\mathsf{vtab}\,\mathbf{:}\,\{(Rows[\mathtt{C},\top]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{m}\ \mathsf{self}\}\,\mathbf{;}\ (Rows[\mathtt{C},\top]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{f}\}\\ SelfTy[\mathtt{C}] & \equiv & \lambda\mathsf{w}::kcn.\ \lambda\mathsf{u}::\mathsf{Type}.\ \lambda\mathsf{tail}::ktail[\mathtt{C}].\ \mu\mathsf{self}::\mathsf{Type}.\ ObjRcd[\mathtt{C}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}\ \mathsf{self}\\ ObjTy[\mathtt{C}] & \equiv & \lambda\mathsf{w}::kcn.\ \lambda\mathsf{u}::\mathsf{Type}.\ \exists\mathsf{tail}::ktail[\mathtt{C}].\ SelfTy[\mathtt{C}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}\\ World & \equiv & \lambda\mathsf{u}::\mathsf{Type}.\ \mu\mathsf{w}::kcn.\ \{(\mathtt{E} = ObjTy[\mathtt{E}]\ \mathsf{w}\ \mathsf{u})^{\mathtt{E}\in cn}\} \end{array}$$

Fig. 7. Macros for object types.

casts. This will be explained in section; for now, we only observe that the macros in figure 7 simply propagate u so that it can appear in the type of the dyncast pseudo-method. The final argument, $\mathsf{tail}::ktail[\mathtt{C}]$, contains the rows for some subclass of C.

$Rows[\mathtt{C},\mathtt{A}]$ 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 $\mathtt{C} \mathrel{\texttt{<:}} \mathtt{A}$) we look to C's super class—let's call it B. $Rows[\mathtt{B},\mathtt{A}]$ produces a type operator for the members between B and A; we need only append the *new* members of C. Conveniently, $Rows[\mathtt{B},\mathtt{A}]$ 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 $\mathsf{tail}\cdot\mathsf{f}$. The new methods in C are found using *addmeth*, and their type signatures $\mathtt{D}_1\ldots\mathtt{D}_n\texttt{->}\mathtt{D}$ are translated to arrow types $\mathsf{self}\longrightarrow\mathsf{w}\cdot\mathtt{D}_1\longrightarrow\ldots\mathsf{w}\cdot\mathtt{D}_n\longrightarrow\mathsf{w}\cdot\mathtt{D}$. 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 7, we use the *Rows* operator to define macros for several variants of the object type for any given class. $Empty[\mathtt{C}]$ denotes the tuple of empty field and method rows of kind $ktail[\mathtt{C}]$. $ObjRcd[\mathtt{C}]$ assembles the rows into records, leaving the subclass rows and self type open. $SelfTy[\mathtt{C}]$ closes self with a fixpoint, and $ObjTy[\mathtt{C}]$ 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

$$\begin{array}{l}\textsc{pack}[\mathtt{C};\mathsf{u};\mathsf{tail};e] = \\ \quad \textbf{fold}\ \langle \mathsf{tail}'{::}\mathit{ktail}[\mathtt{C}] = \mathsf{tail},\ e : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}'\rangle \\ \quad\quad \textbf{as}\ \mathit{World}\ \mathsf{u}\ \textbf{at}\ \lambda\gamma{::}kcn.\ \gamma{\cdot}\mathtt{C}\end{array}$$

$$\begin{array}{l}\textsc{upcast}[\mathtt{C};\mathtt{A};\mathsf{u};e] = \\ \quad \textbf{open}\ (\textbf{unfold}\ e\ \textbf{as}\ \mathit{World}\ \mathsf{u}\ \textbf{at}\ \lambda\gamma{::}kcn.\ \gamma{\cdot}\mathtt{C})\ \textbf{as}\ \langle \mathsf{tail}{::}\mathit{ktail}[\mathtt{C}],\ x : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}\rangle \\ \quad\quad \textbf{in}\ \textsc{pack}[\mathtt{A};\mathsf{u};\mathit{Rows}[\mathtt{C},\mathtt{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail};x]\end{array}$$

Fig. 8. Macros for pack and upcast transformations.

$$\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{x}] = \mathtt{x} \qquad (\textsc{var})$$

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

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e} \in \mathtt{C} \qquad (\mathtt{m},\mathtt{B}_1\ldots\mathtt{B}_n\mathtt{->B}) \in \mathit{methvec}(\mathtt{C}) \qquad \textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{e}] = e \\ \Gamma \vdash \mathtt{e}_i \in \mathtt{D}_i \quad \mathtt{D}_i \mathrel{\mathtt{<:}} \mathtt{B}_i \qquad \textsc{upcast}[\mathtt{D}_i;\mathtt{B}_i;\mathsf{u};\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{e}_i]] = e_i, \quad i \in \{1..n\}\end{array}}{\begin{array}{l}\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{e.m(e}_1\ldots\mathtt{e}_n\mathtt{)}] = \\ \quad \textbf{open}\ (\textbf{unfold}\ e\ \textbf{as}\ \mathit{World}\ \mathsf{u}\ \textbf{at}\ \lambda\gamma.\ \gamma{\cdot}\mathtt{C})\ \textbf{as}\ \langle \mathsf{tail},\ x : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}\rangle \\ \quad\quad \textbf{in}\ (\textbf{unfold}\ x)\mathsf{.vtab.}\mathtt{m}\ x\ e_1\ \ldots\ e_n\end{array}} \qquad (\textsc{invoke})$$

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{C}) = \mathtt{B}_1\ \mathtt{f}_1\ldots\mathtt{B}_n\ \mathtt{f}_n \\ \Gamma \vdash \mathtt{e}_i \in \mathtt{D}_i \qquad \mathtt{D}_i \mathrel{\mathtt{<:}} \mathtt{B}_i \qquad \textsc{upcast}[\mathtt{D}_i;\mathtt{B}_i;\mathsf{u};\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{e}_i]] = e_i, \quad i \in \{1..n\}\end{array}}{\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{new\ C(e}_1\ldots\mathtt{e}_n\mathtt{)}] = (\mathsf{classes.}\mathtt{C}\ \{\})\mathsf{.new}\ e_1\ \ldots\ e_n} \qquad (\textsc{new})$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{D} \qquad \mathtt{D} \mathrel{\mathtt{<:}} \mathtt{C}}{\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{(C)e}] = \textsc{upcast}[\mathtt{D};\mathtt{C};\mathsf{u};\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{e}]]} \qquad (\textsc{upcast})$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{C} \qquad \mathtt{D} \mathrel{\mathtt{<:}} \mathtt{C} \qquad \textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{e}] = e}{\begin{array}{l}\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{(D)e}] = \\ \quad \textbf{open}\ (\textbf{unfold}\ e\ \textbf{as}\ \mathit{World}\ \mathsf{u}\ \textbf{at}\ \lambda\gamma.\ \gamma{\cdot}\mathtt{C})\ \textbf{as}\ \langle \mathsf{tail},\ x : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}\rangle \\ \quad\quad \textbf{in case}\ (\textbf{unfold}\ x)\mathsf{.vtab.dyncast}\ x\ [(\mathit{World}\ \mathsf{u}){\cdot}\mathtt{D}]\ (\mathsf{classes.}\mathtt{D}\ \{\})\mathsf{.proj} \\ \quad\quad\quad \textbf{of}\ \mathsf{some}\ y \Rightarrow y\ \textbf{else abort}\ [(\mathit{World}\ \mathsf{u}){\cdot}\mathtt{D}]\end{array}} \qquad (\textsc{dncast})$$

Fig. 9. Type-directed translation of FJ expressions.

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*.

4.3 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. Figures 8 and 9 contain term macros PACK and UPCAST, and six rules governing the judgment $\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathtt{e}] = e$ 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 has type $(\mathit{World}\ \mathsf{u}){\cdot}\mathtt{C}$ (Theorem 5).

The PACK macro packages and folds a recursive record term into a closed, complete object whose type is selected from a mutual fixpoint of the types of objects of all classes. Supposing that tail is some row tuple in *ktail*[C] and e has type *SelfTy*[C] (*World* u) u tail, the term PACK[C; u; tail; e] has type (*World* u)·C. Since *unpacking* an object binds a type variable to the hidden witness type, it is not as convenient to define as a macro, and we perform it inline instead.

The UPCAST macro opens a term representing an object of class C and repackages it as a term representing an object of some super class A. The object term e has type (*World* u)·C where C <: A, but dynamically it might belong to some subclass D <: C. The **open** binds the type variable tail to the hidden row types that represent members in D but not in C. The UPCAST macro then uses *Rows* to prefix tail with the types of members in C but not in A. Finally, UPCAST calls on the PACK macro to hide the new tail, yielding an object term of type (*World* u)·A.

These macros simply and effectively formalize the encoding techniques demonstrated in the previous section. Importantly, they use type manipulations only (**fold**, **unfold**, **open**). Since these operations are erased before runtime, the PACK and UPCAST macros have no impact on performance.

We now explain each of the translation rules in figure 9, 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 $\Gamma \vdash$ e $\in$ 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($e_1 \ldots e_n$) 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 field 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 function of type self$\rightarrow\forall\alpha.$ (u$\rightarrow$**maybe** α)$\rightarrow$**maybe** α, where self is the type of the unfolded unpacked object bound to x. The argument of (**unfold** x).vtab.dyncast x [τ] is a *projection* function, attempting to convert a value of type u to an object of type τ. The record classes.C {} contains, in addition to the field new, a proj field of type u$\rightarrow$**maybe** ((*World* u)·C). 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 aborts. In full Java, we would throw a ClassCast exception.

Note that Featherweight Java's *stupid* casts [Igarashi et al. 1999] are not compiled at all. They arise in intermediate results during evaluation, but should not appear in valid source-level programs.

The expression translation judgment EXP *preserves types*. Informally, if $\mathtt{e}$ has type $\mathtt{C}$, then its translation has type $(\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$ (for some type u). To state this property formally, we must first translate a FJ typing environment Γ:

$$\begin{aligned}\mathrm{ENV}[\mathsf{u};\ \Gamma, \mathtt{x} : \mathtt{D}] &= \mathrm{ENV}[\mathsf{u};\ \Gamma], \mathtt{x}\,\mathbf{:}\,(\mathit{World}\ \mathsf{u})\cdot\mathtt{D}\\ \mathrm{ENV}[\mathsf{u};\ \circ] &= \circ\end{aligned}$$

It is easy to show by induction that $\mathrm{ENV}[\mathsf{u};\ \Gamma]$ is a well-formed environment, assuming that the range of Γ is a subset of *cn*. We are now prepared to state formally the type preservation theorem:

THEOREM 1 (TYPE PRESERVATION). *If* $\Phi \vdash \mathsf{u} :: \mathsf{Type}$, $\Phi; \Delta \vdash \mathsf{classes} : \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\}$ *and* $\Gamma \vdash \mathtt{e} \in \mathtt{C}$, *then* $\Phi; \Delta, \mathrm{ENV}[\mathsf{u};\ \Gamma] \vdash \mathrm{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e}] : (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$.

The detailed proof is in appendix C, but here is an overview. It proceeds by induction on the structure of $\mathtt{e}$. 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*.

4.4 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, the code of each inherited method should be shared by all inheritors. How might we implement the Point methods so that they can be packaged with a ScaledPoint? We make the method record polymorphic over the tail of the self type:

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

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

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 ScaledPoint subclass inherits getx and adds a method of its own. Its dictionary would be:

$$\begin{aligned}\mathsf{dictSP} = \mathbf{\Lambda}\mathsf{tail}{::}ktail[\mathtt{SP}].\, \{&\mathsf{getx} = (\mathsf{dictPT}\ [r_{sp}]).\mathsf{getx},\\ &\mathsf{gets} = \boldsymbol{\lambda}\mathsf{self} : s_{sp}.\, (\textbf{unfold}\ \mathsf{self}).\mathsf{s}\}\end{aligned}$$

$$\begin{aligned}\text{where } r_{sp} &= Rows[\mathtt{SP}, \mathtt{PT}]\ (World\ \mathsf{u})\ \mathsf{u}\ Empty[\mathtt{SP}]\\ \text{and } s_{sp} &= \mu\alpha.\, \{\mathsf{vtab} : \{\mathsf{getx} : \alpha \to \textbf{int}\,;\, \mathsf{gets} : \alpha \to \textbf{int}\,;\, \mathsf{tail}{\cdot}\mathsf{m}\ \alpha\}\,;\\ &\qquad \mathsf{x} : \textbf{int}\,;\, \mathsf{s} : \textbf{int}\,;\, \mathsf{tail}{\cdot}\mathsf{f}\}\end{aligned}$$

This dictionary can be instantiated with empty tails to produce the ScaledPoint 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 [1998] introduce *tags*—values which track type information at runtime. If a tag of abstract type $\mathsf{Tag}\ \alpha$ equals another tag of known type $\mathsf{Tag}\ \tau$, then we update the context to reflect that $\alpha = \tau$. Note that this differs from intensional type analysis [Harper and Morrisett 1995], which performs structural comparison and does not distinguish named types.

Tags work well with our encoding; in an implementation 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

$$\begin{aligned}\boldsymbol{\lambda}x : u.\, &\textbf{case}\ x\ \textbf{of}\ \mathtt{C}\ y \Rightarrow \textbf{some}\ [ObjTy[\mathtt{C}]\ (World\ u)\ u]\ y\\ &\textbf{else none}\ [ObjTy[\mathtt{C}]\ (World\ u)\ u]\end{aligned}$$

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

$$\mathsf{self} \to \forall\alpha.\, (u \to \textbf{maybe}\ \alpha) \to \textbf{maybe}\ \alpha$$

$$
\begin{array}{lll}
\mathit{Dict}[\mathtt{C}] & \equiv & \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{u}{::}\mathsf{Type}.\ \lambda \mathsf{self}{::}\mathsf{Type}.\ \pmb{\{}(\mathit{Rows}[\mathtt{C},\top]\ \mathsf{w}\ \mathsf{u}\ \mathit{Empty}[\mathtt{C}])\cdot\mathsf{m}\ \mathsf{self}\pmb{\}} \\
\mathit{Ctor}[\mathtt{C}] & \equiv & \lambda \mathsf{w}{::}kcn.\ \mathsf{w}\cdot\mathtt{D}_1 \longrightarrow \ldots\ \mathsf{w}\cdot\mathtt{D}_n \longrightarrow \mathsf{w}\cdot\mathtt{C} \\
 & & \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}{::}kcn.\ \lambda \mathsf{u}{::}\mathsf{Type}.\ \mathsf{u}\longrightarrow \textbf{maybe}\ \mathsf{w}\cdot\mathtt{C} \\
\mathit{Inj}[\mathtt{C}] & \equiv & \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{u}{::}\mathsf{Type}.\ \mathsf{w}\cdot\mathtt{C}\longrightarrow \mathsf{u} \\
\mathit{Class}[\mathtt{C}] & \equiv & \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{u}{::}\mathsf{Type}.\ \pmb{\{}\mathsf{dict}\,\pmb{:}\,\forall \mathsf{tail}{::}ktail[\mathtt{C}].\ \mathit{Dict}[\mathtt{C}]\ \mathsf{w}\ \mathsf{u}\ (\mathit{SelfTy}[\mathtt{C}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})\pmb{,} \\
 & & \qquad\qquad \mathsf{proj}\,\pmb{:}\,\mathit{Proj}[\mathtt{C}]\ \mathsf{w}\ \mathsf{u}\pmb{,}\ \ \mathsf{new}\,\pmb{:}\,\mathit{Ctor}[\mathtt{C}]\ \mathsf{w}\,\pmb{\}} \\
 & & \\
\mathit{Classes} & \equiv & \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{u}{::}\mathsf{Type}.\ ((\mathtt{E}\,\pmb{:}\,\mathbf{1}\longrightarrow \mathit{Class}[\mathtt{E}]\ \mathsf{w}\ \mathsf{u}\,\pmb{;})^{\mathtt{E}\in cn}\ \mathsf{Abs}^{cn}) \\
\mathit{ClassF}[\mathtt{C}] & \equiv & \forall \mathsf{u}{::}\mathsf{Type}.\ \mathit{Inj}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\longrightarrow \mathit{Proj}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\longrightarrow \\
 & & \qquad \pmb{\{}\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\pmb{\}}\longrightarrow \mathbf{1}\longrightarrow \mathit{Class}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}
\end{array}
$$

Fig. 10. Macros for dictionary, constructor, and class types.

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.

4.5 Class translation

We now explore the formal translation of class declarations and construction of their method dictionaries. In figure 10 we define several macros for describing dictionary and class types. Figure 11 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. Our compilation model is the subject of section 4.7.

CDEC[C] produces the functor corresponding to class C; see the definition in the top left of figure 11. The code has one type parameter: u, the universal type used for dynamic 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

Class declaration translation:

$$\begin{array}{l}\textsc{cdec}[\mathtt{C}] = \mathbf{\Lambda}\mathsf{u}{::}\mathsf{Type}.\ \boldsymbol{\lambda}\mathsf{inj} : \mathit{Inj}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}.\ \boldsymbol{\lambda}\mathsf{proj} : \mathit{Proj}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}.\\ \quad \boldsymbol{\lambda}\mathsf{classes} : \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\}.\ \boldsymbol{\lambda}\_ : \mathbf{1}.\\ \quad \mathbf{let}\ \mathsf{dict} : \forall \mathsf{tail}{::}\mathit{ktail}[\mathtt{C}].\ \mathit{Dict}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})\\ \quad\quad = \textsc{dict}[\mathtt{C}; \mathsf{u}; \mathsf{inj}; \mathsf{classes}]\\ \quad \mathbf{in\ let}\ \mathsf{vtab} = \mathsf{dict}\ [\mathit{Empty}[\mathtt{C}]]\\ \quad\quad \mathbf{in}\ \{\mathsf{dict} = \mathsf{dict}, \mathsf{proj} = \mathsf{proj}, \mathsf{new} = \textsc{new}[\mathtt{C}; \mathsf{u}; \mathsf{vtab}]\}\end{array} \tag{23}$$

Dictionary construction:

$$\begin{array}{l}\textsc{dict}[\mathtt{Obj}; \mathsf{u}; \mathsf{inj}; \mathsf{classes}] = \mathbf{\Lambda}\mathsf{tail}{::}\mathit{ktail}[\mathtt{Obj}].\\ \quad \{\mathsf{dyncast} = \boldsymbol{\lambda}\mathsf{self} : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}.\ \mathbf{\Lambda}\alpha{::}\mathsf{Type}.\ \boldsymbol{\lambda}\mathsf{proj} : \mathsf{u} \longrightarrow \mathbf{maybe}\ \alpha.\\ \quad\quad \mathsf{proj}\ (\mathsf{inj}\ \textsc{pack}[\mathtt{Obj}; \mathsf{u}; \mathsf{tail}; \mathsf{self}])\}\end{array} \tag{24}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{B}\,\{\ldots\} \qquad \mathrm{dom}(\mathit{methvec}(\mathtt{C})) = [l_1 \ldots l_n]}{\begin{array}{l}\textsc{dict}[\mathtt{C}; \mathsf{u}; \mathsf{inj}; \mathsf{classes}] = \mathbf{\Lambda}\mathsf{tail}{::}\mathit{ktail}[\mathtt{C}].\\ \quad \mathbf{let}\ \mathsf{super} : \mathit{Dict}[\mathtt{B}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})\\ \quad\quad = (\mathsf{classes}.\mathtt{B}\ \{\}).\mathsf{dict}\ [\mathit{Rows}[\mathtt{C},\mathtt{B}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}]\\ \quad \mathbf{in}\ \{l_1 = \textsc{meth}[\mathtt{C}; l_1; \mathsf{u}; \mathsf{tail}; \mathsf{inj}; \mathsf{classes}; \mathsf{super}], \ldots,\\ \quad\quad l_n = \textsc{meth}[\mathtt{C}; l_n; \mathsf{u}; \mathsf{tail}; \mathsf{inj}; \mathsf{classes}; \mathsf{super}]\}\end{array}} \tag{25}$$

Constructor code:

$$\frac{\mathit{fields}(\mathtt{C}) = \mathtt{D}_1\ \mathtt{f}_1 \ldots \mathtt{D}_n\ \mathtt{f}_n}{\begin{array}{l}\textsc{new}[\mathtt{C}; \mathsf{u}; \mathsf{vtab}] = \boldsymbol{\lambda}\mathtt{f}_1 : (\mathit{World}\ \mathsf{u}){\cdot}\mathtt{D}_1.\ \ldots \boldsymbol{\lambda}\mathtt{f}_n : (\mathit{World}\ \mathsf{u}){\cdot}\mathtt{D}_n.\\ \quad \mathbf{let}\ x = \mathbf{fold}\ \{\mathsf{vtab} = \mathsf{vtab}, \mathtt{f}_1 = \mathtt{f}_1, \ldots, \mathtt{f}_n = \mathtt{f}_n\}\\ \quad\quad \mathbf{as}\ \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathit{Empty}[\mathtt{C}]\\ \quad \mathbf{in}\ \textsc{pack}[\mathtt{C}; \mathsf{u}; \mathit{Empty}[\mathtt{C}]; x]\end{array}} \tag{26}$$

Method code:

$$\begin{array}{l}\textsc{meth}[\mathtt{C}; \mathsf{dyncast}; \mathsf{u}; \mathsf{tail}; \mathsf{inj}; \mathsf{classes}; \mathsf{super}] =\\ \quad \boldsymbol{\lambda}\mathsf{self} : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}.\ \mathbf{\Lambda}\alpha{::}\mathsf{Type}.\ \boldsymbol{\lambda}\mathsf{proj} : \mathsf{u} \longrightarrow \mathbf{maybe}\ \alpha.\\ \quad\quad \mathbf{case}\ \mathsf{proj}\ (\mathsf{inj}\ \textsc{pack}[\mathtt{C}; \mathsf{u}; \mathsf{tail}; \mathsf{self}])\\ \quad\quad\quad \mathbf{of}\ \mathsf{some}\ x \Rightarrow \mathbf{some}\ [\alpha]\ x\ \ \mathbf{else}\ \mathsf{super}.\mathsf{dyncast}\ \mathsf{self}\ [\alpha]\ \mathsf{proj}\end{array} \tag{27}$$

$$\frac{\begin{array}{l}\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{B}\,\{\ldots \mathtt{K}\ \mathtt{M}_1 \ldots \mathtt{M}_n\}\\ \mathtt{m}\ \text{not defined in}\ \mathtt{M}_1 \ldots \mathtt{M}_n\end{array}}{\textsc{meth}[\mathtt{C}; \mathtt{m}; \mathsf{u}; \mathsf{tail}; \mathsf{inj}; \mathsf{classes}; \mathsf{super}] = \mathsf{super}.\mathtt{m}} \tag{28}$$

$$\frac{\begin{array}{l}\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{B}\,\{\ldots \mathtt{K}\ \mathtt{M}_1 \ldots \mathtt{M}_n\} \qquad \exists j : \mathtt{M}_j = \mathtt{A}\ \mathtt{m}(\mathtt{A}_1\ \mathtt{x}_1 \ldots \mathtt{A}_m\ \mathtt{x}_m)\ \{\uparrow \mathtt{e};\}\\ \Gamma = \mathtt{x}_1{:}\mathtt{A}_1, \ldots, \mathtt{x}_m{:}\mathtt{A}_m, \mathtt{this}{:}\mathtt{C} \qquad \Gamma \vdash \mathtt{e} \in \mathtt{D} \qquad \mathtt{D}\ \mathtt{<:}\ \mathtt{A} \qquad \textsc{exp}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathtt{e}] = e\end{array}}{\begin{array}{l}\textsc{meth}[\mathtt{C}; \mathtt{m}; \mathsf{u}; \mathsf{tail}; \mathsf{inj}; \mathsf{classes}; \mathsf{super}] =\\ \quad \boldsymbol{\lambda}\mathsf{self} : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}.\ \boldsymbol{\lambda}\mathtt{x}_1 : (\mathit{World}\ \mathsf{u}){\cdot}\mathtt{A}_1.\ \ldots \boldsymbol{\lambda}\mathtt{x}_m : (\mathit{World}\ \mathsf{u}){\cdot}\mathtt{A}_m.\\ \quad\quad \mathbf{let}\ \mathtt{this} : (\mathit{World}\ \mathsf{u}){\cdot}\mathtt{C} = \textsc{pack}[\mathtt{C}; \mathsf{u}; \mathsf{tail}; \mathsf{self}]\\ \quad\quad \mathbf{in}\ \textsc{upcast}[\mathtt{D}; \mathtt{A}; \mathsf{u}; e]\end{array}} \tag{29}$$

Fig. 11. Translation of class declarations.

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.

$$\begin{aligned}
\mathit{Tagged} &= \lambda \mathsf{u}{::}\mathsf{Type}.\,\boldsymbol{[}(\mathtt{C} : (\mathit{World}\ u)\cdot\mathtt{C})^{\mathtt{C}\in cn}\boldsymbol{]}\\
u &= \mu \mathsf{u}{::}\mathsf{Type}.\ \mathit{Tagged}\ \mathsf{u}
\end{aligned}$$

$$\textsc{prog}[\mathsf{e}] = \textbf{let}\ x_{cn} = \textsc{link}\ \boldsymbol{\{}(\mathtt{C} = \textsc{cdec}[\mathtt{C}])^{\mathtt{C}\in cn}\boldsymbol{\}}\ \textbf{in}\ \textsc{exp}[\circ; u; x_{cn}; \mathsf{e}]$$

$$\begin{aligned}
\textsc{link} = {} & \boldsymbol{\lambda} x : \boldsymbol{\{}(\mathtt{C} : \mathit{ClassF}[\mathtt{C}])^{\mathtt{C}\in cn}\boldsymbol{\}}.\\
& \quad \textbf{fix}\,\boldsymbol{[}\mathit{Classes}\,(\mathit{World}\ u)\ u\boldsymbol{]}\\
& \quad\quad (\boldsymbol{\lambda}\mathsf{classes} : \boldsymbol{\{}\mathit{Classes}\,(\mathit{World}\ u)\ u\boldsymbol{\}}.\ \ \boldsymbol{\{}(\mathtt{C} = x.\mathtt{C}\ [u]\ \mathsf{inj}_{\mathtt{C}}\ \mathsf{proj}_{\mathtt{C}}\ \mathsf{classes})^{\mathtt{C}\in cn}\boldsymbol{\}})\\
& \text{where}\\
& \quad \mathsf{inj}_{\mathtt{C}} = \boldsymbol{\lambda} x : (\mathit{World}\ u)\cdot\mathtt{C}.\ \textbf{fold}\ \textbf{inj}_{\mathtt{C}}^{\mathit{Tagged}\ u}\ x\ \textbf{as}\ u\\
& \quad \mathsf{proj}_{\mathtt{C}} = \boldsymbol{\lambda} x : u.\ \textbf{case unfold}\ x\ \textbf{of}\ \ \mathtt{C}\ y \Rightarrow \textbf{some}\ [(\mathit{World}\ u)\cdot\mathtt{C}]\ y\\
& \qquad\qquad\qquad\qquad \textbf{else none}\ [(\mathit{World}\ u)\cdot\mathtt{C}]
\end{aligned}$$

Fig. 12. Program translation and linking.

4.6 Linking

Finally, we must instantiate and link the separate class modules together into a single program. Figure 12 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 10 as a parameterized sum type with a variant for the object type of each class in the class table. We instantiate each $x.\mathtt{C}$ with the fixed point of *Tagged*. Next we pass the injection and projection functions, $\mathsf{inj}_{\mathtt{C}}$ and $\mathsf{proj}_{\mathtt{C}}$. The final argument to $x.\mathtt{C}$ is the classes record itself.

4.7 Separate compilation

Our translation supports separate compilation, but the formal presentation does not make this clear. In this section, we describe our compilation model and justify that claim.

What must be known to compile a Java class `C` to native code? At a minimum, we must know the fields and methods of all super classes, to ensure that the layout of `C`'s vtable and objects are consistent. Next, it is helpful to know enough about classes referenced by `C` so that the offsets of their fields and methods can be embedded in the code. These principles do not mean that all referenced classes must be *compiled* together. Indeed, as long as the above information is known, classes can be compiled separately, in any order.

In our translation, we need not just offsets but the *full* type information for super classes and referenced classes. If `C` refers to field `x` from class `D`, we need to know all about the type of `x` (`E`, for example) as well. This clearly involves extracting type information from more classes, although not necessarily *every* class in the program. Even so, each class can still be compiled separately, in any order, assuming the requisite types are available.

A reasonable compilation strategy starts with some *root set* of classes and builds a dependence graph. For a given program, the root set contains just the class with the `main` method; for a library, it includes all exported classes. Next, traverse the graph bottom-up. Comple each class separately, but propagate the necessary information from `C` to all those classes that depend on it. Of course, there may be cyclic dependencies, represented by strongly-connected components (clusters) in the graph. In these cases, we extract type information from all members of the cluster before compiling any of them. Still, each class in the cluster is compiled separately.

A hallmark of whole-program compilation is that library code must be compiled along with application code. This is clearly not necessary in our model. Library classes would never depend on application classes, so they can be compiled in advance. The reason that our formal translation uses the macro *World* (containing object types for every class in the program) is that, in the most general case, every class in an FJ program refers to every other class. Thus, our translation assumes that the entire program falls within one strongly-connected cluster. In practice, *World* would include just the classes in the same cluster as the class being compiled.

5. EXTENSIONS

Our translation extends to support many features of Java which are excluded from FJ. Null references are encoded by lifting all external object types to sum types with a null alternate (just like the maybe type). Then, all object operations must verify that the object pointer is not null. Our target calculus, unlike JVML, can express that an object is non-null, so null pointer checks can be safely hoisted.

Static members, interface fields, and multiple parameterized constructors can be added to the class record, along with the dictionary and tag. Mutable fields are easily modeled using mutable records. As required by the JVM, the new function allocates the object record with a default 'zero' value for each field. Then any public constructor can be invoked to assign new values to the fields. Super invocations select the method statically from the super class dictionary (as is currently used in dyncast). Java exceptions work similarly to those of SML.

Private methods are defined along with the other methods. Since they can neither be called from subclasses nor overridden, we simply omit them from the vtable and dictionary. Protected and package scopes are difficult, however, because they transcend compilation unit boundaries. In Moby, Fisher and Reppy [1999] use two distinct views of classes, a class view and an object view. These correspond roughly to the dict and new fields of our class encoding. If we export a class outside its definitional package, all protected methods and fields should be hidden from the object view but not the class view while those of package scope should be hidden from both.

Private fields

Private fields can be hidden from outsiders using existential types [League et al. 1999]. For convenience, assume that the private fields of each class in the hierarchy are collected into separate records. Suppose that `Point` has private fields `x` and `y`, and public field `z`; and `ScaledPoint` has private field `s`. The layout for a `ScaledPoint` object would be {vtab, Pt:{x, y}, z, SPt:{s}}. With the private fields separated like this, it is easy to hide their types separately. (Using a flat representation is possible, but this separation allows a simpler, more orthogonal presentation.) We embed each class functor in an existential package, where the witness type includes the types of the private fields of that class. From inside class `ScaledPoint`, we **open** the `Point` package, binding a type variable α to represent the private fields of `Point`. Then, our local view of the object is {vtab, Pt:α, z, SPt:{s}}. Using *dot notation* [Cardelli and Leroy 1990] for existential types makes this encoding more convenient, but is not required.

Unfortunately, privacy interacts 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 described above, 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$\longrightarrow$("`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 limitations of the isorecursive types used in our target calculus. We have developed a solution which does not require extending the language. 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. The issues are subtle and a formal treatment is outside the scope of this article. We are considering extending FJ itself with privacy in order to elucidate the technique.

Interfaces

Given an object of interface type, we know nothing about the shape of its vtable. There are various ways of locating methods in interface objects. Proebsting et al. [1997] construct a per-class dictionary that maps method names to offsets in the vtable. Krall and Grafl [1997] construct a separate method table (called an *itable*) for each declared interface, storing them all somewhere in the vtable. Although they are not clear on how to *use* the itable, there appear to be two choices. First, we can search for the appropriate itable in the vtable, which amounts to lookup of interface names rather than method names. Second, when casting an object from class type to interface type, we can select the itable and then pair it with the object itself. This avoids name lookup entirely but requires minor coercions when casting to and between interface types.

Our translation can be extended to support both strategies. For the first strategy, all we need is to introduce *unordered* records into our target language (see [League et al. 1999]), with a primitive for dictionary lookup. All the itables for a class would be collected into a separate unordered record, itself an element of the still ordered vtable. Then, casting an object to an interface type only requires repackaging (a runtime no-op) to hide those entries not exported by the current interface.

We can also follow the latter strategy, representing interface objects as a pair where the type of the underlying object is concealed by an existential type. For example, an object which implements the `Runnable` interface includes a method `run()` which can be invoked to start a new thread. In our target language, a `Runnable` object is represented as $\exists\alpha{::}\mathsf{Type}.\,\{\mathsf{itab}:\{\mathsf{run}:\alpha\to\mathbf{1}\},\,\mathsf{obj}:\alpha\}$. To invoke the method, we open the existential, select the method from the itab, select the obj, and apply. With this representation, interface method invocations are about the same cost as normal method invocations, although upward casts to interface types are no longer free.

6. RELATED WORK

Fisher and Mitchell [1998] 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 [1997] 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 [Bruce et al. 1999; Pierce and Turner 1994]. As in the Abadi, Cardelli, and Viswanathan encoding [1996], 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 [1994] 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. [1998] 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 [1999] encodes the object calculus of Abadi and Cardelli [1996] using existential and intersection types in a calculus of coercions. Glew [2000a] translates a simple class-based object calculus into an intermediate language with F-bounded polymorphism [Canning et al. 1989; Eifrig et al. 1995] and a special 'self' quantifier.

Comparing object encodings

A more detailed comparison with the work of Glew and Crary is worthwhile. The three encodings share many similarities, and appear to be different ways of expressing the same underlying idea. As Glew remarks, "both Crary and League *et al.*'s ideas can be seen as encodings of the self quantifier introduced in this paper" [Glew 2000a, page 9]. He did not present a detailed comparison, but the statement is indeed true.

In this section, we will attempt to clarify the connections between these encodings. Following Bruce et al. [1999], we can specify object interfaces as type operators, so that the type of the self argument can be plugged in. The `Point` interface, for example, would be represented as $I_P = \lambda\alpha{::}\mathsf{Type}.\,\{\mathsf{getx}:\alpha\to\mathbf{int}\}$.

Glew used a twist on F-bounded polymorphism to encode method tables that could be reused in subclasses. This leads naturally to an object encoding using an F-bounded existential (FBE): $\exists\alpha \leq I(\alpha).\alpha$, which Glew writes as $\mathsf{self}\ \alpha.I(\alpha)$. Typically, the witness type is recursive; it is a subtype of its unrolling.

The connection between self and the F-bounded existential was recognized independently by Glew and ourselves.[1] We can derive the rules governing self from those for F-bounded existentials. Glew uses equi-recursive types in [2000a]; a restriction to iso-recursive types is possible, though awkward [Glew 2000b]. The rules for packing and opening self types must simultaneously fold and unfold in precisely the right places.

Self application is typable in FBE because the object, via subsumption, enjoys two types: the abstract type α and the interface type $I(\alpha)$. Crary [1999] encodes precisely the same property as an intersection type: $\exists\alpha.\,\alpha \wedge I(\alpha)$. Again, the witness type is recursive. With equi-recursive types, a value of type $\mu\, I$ also has type $I\,(\mu\, I)$; it could be packaged as $\alpha \wedge I(\alpha)$. Crary makes this encoding practical using a calculus of *coercions*—explicit retyping annotations. Coercions can drop fields from the end of a record, fold and unfold recursive types, mediate intersection types, and instantiate quantified types. All coercions are erasable.

We will now show how our own encoding, based on row polymorphism, relates to these. A known technique for eliminating an F-bound is to replace it with a higher-order bound and a recursive type. That is, we could represent $\exists\alpha \leq I(\alpha).\alpha$ as $\exists\delta \leq I.\mu\,\delta$. Using a point-wise subtyping rule, the interface type operators themselves enjoy a subtyping relationship. Iso-recursive types can be used directly with this technique because the fixpoint is separate from the existential.

Next, though it is less efficient, we can implement the higher-order subtyping with a coercion function:

$$\exists\delta :: \mathsf{Type}\Rightarrow\mathsf{Type}.\{c : \delta\,(\mu\,\delta) \rightarrow I\,(\mu\,\delta),\ o : \mu\,\delta\}$$

To select a method from an object, we first open the package, select the coercion c, and apply it to the unfolding of o. This yields an interface whose methods are then directly applicable to o.

Using a general function for this coercion yields more flexibility than we require to implement Java. All the function ever needs to do is drop fields from records. With row polymorphism, we can express the result of pre-applying the coercions at all levels. The encodings of Crary and Glew work by supplying two distinct views of the object: an abstract subtype of a concrete interface type. With row polymorphism, that distinction is unnecessary; we can hide just the unknown portion of the interface directly.

All three of these encodings are operationally efficient. The primary differences between them are in the complexity required of the target calculus. 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. We support mutual

[1] Personal communication, August 2000

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.

This translation is being implemented as a new front-end to the SML/NJ compiler. It loads Java class files, translates them to FLINT, and then into native code. We can currently compile toy Java programs; the intermediate code successfully type-checks after every phase. Preliminary measurements of both compilation and run times are promising, but some work remains before we can run real benchmarks.

APPENDIX

A. FEATHERWEIGHT JAVA SEMANTICS

These rules are reprinted from [Igarashi et al. 1999], with a few adaptations.

Field lookup

$$\mathit{fields}(\mathtt{Obj}) = \bullet \tag{30}$$

$$\frac{\begin{array}{l} CT(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{B}\ \{\mathtt{C}_1\ \mathtt{f}_1;\ldots \mathtt{C}_n\ \mathtt{f}_n;\ \mathtt{K}\ldots\} \\ \mathit{fields}(\mathtt{B}) = \mathtt{B}_1\ \mathtt{g}_1 \ldots \mathtt{B}_m\ \mathtt{g}_m \end{array}}{\mathit{fields}(\mathtt{C}) = \mathtt{B}_1\ \mathtt{g}_1 \ldots \mathtt{B}_m\ \mathtt{g}_m, \mathtt{C}_1\ \mathtt{f}_1 \ldots \mathtt{C}_n\ \mathtt{f}_n} \tag{31}$$

Method lookup

$$\frac{\begin{array}{l} CT(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{B}\ \{\ldots \mathtt{K}\ \mathtt{M}_1 \ldots \mathtt{M}_n\} \\ \exists j : \mathtt{M}_j = \mathtt{D\ m}(\mathtt{D}_1\ \mathtt{x}_1 \ldots \mathtt{D}_m\ \mathtt{x}_m)\ \{\uparrow \mathtt{e};\} \end{array}}{\begin{array}{c} \mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathtt{D}_1 \ldots \mathtt{D}_m\mathtt{->D} \\ \mathit{mbody}(\mathtt{m},\mathtt{C}) = (\mathtt{x}_1 \ldots \mathtt{x}_m, \mathtt{e}) \end{array}} \tag{32}$$

$$\frac{\begin{array}{l} CT(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{B}\ \{\ldots \mathtt{K}\ \mathtt{M}_1 \ldots \mathtt{M}_n\} \\ \mathtt{m} \text{ not defined in } \mathtt{M}_1 \ldots \mathtt{M}_n \end{array}}{\begin{array}{c} \mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathit{mtype}(\mathtt{m},\mathtt{B}) \\ \mathit{mbody}(\mathtt{m},\mathtt{C}) = \mathit{mbody}(\mathtt{m},\mathtt{B}) \end{array}} \tag{33}$$

Valid method overriding

$$\frac{\mathit{mtype}(\mathtt{m},\mathtt{B}) = \mathtt{C}_1 \ldots \mathtt{C}_n\mathtt{->C}_0}{\mathit{override}(\mathtt{m},\mathtt{B},\mathtt{C}_1 \ldots \mathtt{C}_n\mathtt{->C}_0)} \tag{34}$$

$$\frac{\nexists T \text{ such that } \mathit{mtype}(\mathtt{m},\mathtt{B}) = T}{\mathit{override}(\mathtt{m},\mathtt{B},\mathtt{C}_1 \ldots \mathtt{C}_n\mathtt{->C}_0)} \tag{35}$$

Computation

$\boxed{\mathtt{e} \longrightarrow \mathtt{e}'}$

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

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

$$\frac{\mathtt{C} <: \mathtt{D}}{\mathtt{(D)new}\ \mathtt{C(e}_1 \ldots \mathtt{e}_n\mathtt{)} \longrightarrow \mathtt{new}\ \mathtt{C(e}_1 \ldots \mathtt{e}_n\mathtt{)}} \qquad \text{(R-CAST)}$$

Subtyping $\boxed{\mathtt{C} <: \mathtt{D}}$

$$\mathtt{C} <: \mathtt{C} \tag{36}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{B}\ \{\ldots\} \qquad \mathtt{B} <: \mathtt{A}}{\mathtt{C} <: \mathtt{A}} \tag{37}$$

Class typing

$$\frac{\begin{array}{l}\mathtt{K} = \mathtt{C(B}_1\ \mathtt{g}_1 \ldots \mathtt{B}_n\ \mathtt{g}_n, \mathtt{C}_1\ \mathtt{f}_1 \ldots \mathtt{C}_m\ \mathtt{f}_m\mathtt{)} \\ \qquad \{\mathtt{super(g}_1 \ldots \mathtt{g}_n\mathtt{)};\\ \qquad\ \ \mathtt{this.f}_1\ \mathtt{=}\ \mathtt{f}_1\mathtt{;} \ldots \mathtt{this.f}_m\ \mathtt{=}\ \mathtt{f}_m\mathtt{;}\} \\ \mathit{fields}(\mathtt{B}) = \mathtt{B}_1\ \mathtt{g}_1 \ldots \mathtt{B}_n\ \mathtt{g}_n \\ \mathtt{M}_i\ \text{ok in}\ \mathtt{C} \quad \forall i \in \{1 \ldots k\}\end{array}}{\mathtt{class}\ \mathtt{C} \triangleleft \mathtt{B}\ \{\mathtt{C}_1\ \mathtt{f}_1\mathtt{;} \ldots \mathtt{C}_m\ \mathtt{f}_m\mathtt{;}\ \mathtt{K}\ \mathtt{M}_1 \ldots \mathtt{M}_k\}\ \text{ok}} \tag{38}$$

Method typing

$$\frac{\begin{array}{l}\mathtt{x}_1 : \mathtt{D}_1, \ldots, \mathtt{x}_n : \mathtt{D}_n, \mathtt{this} : \mathtt{C} \vdash \mathtt{e} \in \mathtt{E} \qquad \mathtt{E} <: \mathtt{D} \\ \mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{B}\ \{\ldots\}\end{array}}{\mathtt{D}\ \mathtt{m(D}_1\ \mathtt{x}_1 \ldots \mathtt{D}_n\ \mathtt{x}_n\mathtt{)}\ \{\uparrow \mathtt{e;}\}\ \text{ok in}\ \mathtt{C}} \tag{39}$$

Expression typing $\boxed{\Gamma \vdash \mathtt{e} \in \mathtt{C}}$

$$\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x}) \qquad \text{(T-VAR)}$$

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

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

$$\frac{\begin{array}{l}\mathit{fields}(\mathtt{C}) = \mathtt{D}_1\ \mathtt{f}_1 \ldots \mathtt{D}_n\ \mathtt{f}_n \\ \Gamma \vdash \mathtt{e}_i \in \mathtt{C}_i \quad \mathtt{C}_i <: \mathtt{D}_i \quad (\forall i \in \{1 \ldots n\})\end{array}}{\Gamma \vdash \mathtt{new}\ \mathtt{C(e}_1 \ldots \mathtt{e}_n\mathtt{)} \in \mathtt{C}} \qquad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash \mathtt{(C)e} \in \mathtt{C}} \qquad \text{(T-UCAST)}$$

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

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

B. TARGET LANGUAGE SEMANTICS

B.1 Static judgments

Φ maps type variables to their kinds and Δ maps term variables to their types.

Kind formation $\boxed{\vdash \kappa \ \mathit{kind}}$

$$\vdash \mathsf{Type}\ \mathit{kind} \tag{40}$$

$$\vdash \mathsf{R}^L\ \mathit{kind} \tag{41}$$

$$\frac{\vdash \kappa\ \mathit{kind} \qquad \vdash \kappa'\ \mathit{kind}}{\vdash \kappa \Rightarrow \kappa'\ \mathit{kind}} \tag{42}$$

$$\frac{\begin{array}{l} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \\ \vdash \kappa_i\ \mathit{kind} \quad (\forall i \in \{1 \ldots n\}) \end{array}}{\vdash \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\}\ \mathit{kind}} \tag{43}$$

Kind environment formation $\boxed{\vdash \Phi\ \mathit{kind\ env}}$

$$\vdash \circ\ \mathit{kind\ env} \tag{44}$$

$$\frac{\vdash \Phi\ \mathit{kind\ env} \qquad \vdash \kappa\ \mathit{kind}}{\vdash \Phi, \alpha :: \kappa\ \mathit{kind\ env}} \tag{45}$$

Type formation $\boxed{\Phi \vdash \tau :: \kappa}$

$$\frac{\vdash \Phi\ \mathit{kind\ env} \qquad \alpha \in \mathrm{dom}(\Phi)}{\Phi \vdash \alpha :: \Phi(\alpha)} \tag{46}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \kappa'}{\Phi \vdash \lambda\alpha{::}\kappa.\, \tau :: \kappa \Rightarrow \kappa'} \tag{47}$$

$$\frac{\Phi \vdash \tau_1 :: \kappa' \Rightarrow \kappa \qquad \Phi \vdash \tau_2 :: \kappa'}{\Phi \vdash \tau_1\ \tau_2 :: \kappa} \tag{48}$$

$$\frac{\begin{array}{l} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \\ \Phi \vdash \tau_i :: \kappa_i \quad (\forall i \in \{1 \ldots n\}) \end{array}}{\Phi \vdash \{l_1 = \tau_1 \ldots l_n = \tau_n\} :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\}} \tag{49}$$

$$\frac{\Phi \vdash \tau :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\}}{\Phi \vdash \tau \cdot l_i :: \kappa_i} \tag{50}$$

$$\frac{\Phi \vdash \tau_1 :: \mathsf{Type} \qquad \Phi \vdash \tau_2 :: \mathsf{Type}}{\Phi \vdash \tau_1 \rightarrow \tau_2 :: \mathsf{Type}} \tag{51}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \kappa}{\Phi \vdash \mu\alpha{::}\kappa.\, \tau :: \kappa} \tag{52}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \mathsf{Type}}{\Phi \vdash \forall\alpha{::}\kappa.\, \tau :: \mathsf{Type}} \tag{53}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \mathsf{Type}}{\Phi \vdash \exists\alpha{::}\kappa.\, \tau :: \mathsf{Type}} \tag{54}$$

Type equivalence $\boxed{\Phi \vdash \tau = \tau' :: \kappa}$

$$\frac{\Phi \vdash \tau_1 :: \kappa_1 \qquad \Phi, \alpha :: \kappa_1 \vdash \tau_2 :: \kappa_2}{\Phi \vdash (\lambda\alpha{::}\kappa_1.\, \tau_2)\ \tau_1 = \tau_2[\alpha := \tau_1] :: \kappa_2} \tag{55}$$

$$\frac{\Phi \vdash \tau :: \kappa \Rightarrow \kappa' \quad \alpha \notin \mathrm{dom}(\Phi)}{\Phi \vdash \lambda\alpha{::}\kappa.\, \tau\ \alpha = \tau :: \kappa \Rightarrow \kappa'} \tag{56}$$

$$\frac{\begin{array}{c} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \\ \Phi \vdash \tau_i :: \kappa_i \quad (\forall i \in \{1 \ldots n\}) \end{array}}{\Phi \vdash \{l_1 = \tau_1 \ldots l_n = \tau_n\} \cdot l_i = \tau_i :: \kappa_i} \tag{57}$$

$$\frac{\begin{array}{c} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \\ \Phi \vdash \tau :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\} \end{array}}{\Phi \vdash \{l_1 = \tau \cdot l_1 \ldots l_n = \tau \cdot l_n\} = \tau :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\}} \tag{58}$$

$$\frac{\Phi \vdash \tau :: \kappa}{\Phi \vdash \tau = \tau :: \kappa} \tag{59}$$

$$\frac{\Phi \vdash \tau_1 = \tau_2 :: \kappa}{\Phi \vdash \tau_2 = \tau_1 :: \kappa} \tag{60}$$

$$\frac{\Phi \vdash \tau_1 = \tau_2 :: \kappa \qquad \Phi \vdash \tau_2 = \tau_3 :: \kappa}{\Phi \vdash \tau_1 = \tau_3 :: \kappa} \tag{61}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau_1 = \tau_2 :: \kappa'}{\Phi \vdash \lambda\alpha{::}\kappa.\, \tau_1 = \lambda\alpha{::}\kappa.\, \tau_2 :: \kappa \Rightarrow \kappa'} \tag{62}$$

$$\frac{\Phi \vdash \tau_1 = \tau_1' :: \kappa' \Rightarrow \kappa \qquad \Phi \vdash \tau_2 = \tau_2' :: \kappa'}{\Phi \vdash \tau_1\ \tau_2 = \tau_1'\ \tau_2' :: \kappa} \tag{63}$$

$$\frac{\begin{array}{c} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \\ \Phi \vdash \tau_i = \tau_i' :: \kappa_i \quad (\forall i \in \{1 \ldots n\}) \end{array}}{\begin{array}{l} \Phi \vdash \{l_1 = \tau_1 \ldots l_n = \tau_n\} = \{l_1 = \tau_1' \ldots l_n = \tau_n'\} \\ \quad :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\} \end{array}} \tag{64}$$

$$\frac{\Phi \vdash \tau = \tau' :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\}}{\Phi \vdash \tau{\cdot}l_i = \tau'{\cdot}l_i :: \kappa_i} \tag{65}$$

$$\frac{\Phi \vdash \tau_1 = \tau_1' :: \mathsf{Type} \qquad \Phi \vdash \tau_2 = \tau_2' :: \mathsf{Type}}{\Phi \vdash \tau_1 \to \tau_2 = \tau_1' \to \tau_2' :: \mathsf{Type}} \tag{66}$$

$$\frac{\Phi \vdash \tau_1 = \tau_1' :: \mathsf{Type} \qquad \Phi \vdash \tau_2 = \tau_2' :: \mathsf{R}^{L \cup \{l\}}}{\Phi \vdash l \mathbin{\mathbf{:}} \tau_1 \mathbin{\mathbf{;}} \tau_2 = l \mathbin{\mathbf{:}} \tau_1' \mathbin{\mathbf{;}} \tau_2' :: \mathsf{R}^{L - \{l\}}} \tag{67}$$

$$\frac{\Phi \vdash \tau = \tau' :: \mathsf{R}^{\emptyset}}{\Phi \vdash \boldsymbol{\{}\tau\boldsymbol{\}} = \boldsymbol{\{}\tau'\boldsymbol{\}} :: \mathsf{Type}} \tag{68}$$

$$\frac{\Phi \vdash \tau = \tau' :: \mathsf{R}^{\emptyset}}{\Phi \vdash \boldsymbol{[}\tau\boldsymbol{]} = \boldsymbol{[}\tau'\boldsymbol{]} :: \mathsf{Type}} \tag{69}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau_1 = \tau_2 :: \kappa}{\Phi \vdash \mu\alpha{::}\kappa.\, \tau_1 = \mu\alpha{::}\kappa.\, \tau_2 :: \kappa} \tag{70}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau_1 = \tau_2 :: \mathsf{Type}}{\Phi \vdash \forall\alpha{::}\kappa.\, \tau_1 = \forall\alpha{::}\kappa.\, \tau_2 :: \mathsf{Type}} \tag{71}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau_1 = \tau_2 :: \mathsf{Type}}{\Phi \vdash \exists\alpha{::}\kappa.\, \tau_1 = \exists\alpha{::}\kappa.\, \tau_2 :: \mathsf{Type}} \tag{72}$$

Type environment formation $\boxed{\Phi \vdash \Delta \ \mathit{type\ env}}$

$$\Phi \vdash \circ \ \mathit{type\ env} \tag{73}$$

$$\frac{\Phi \vdash \Delta \ \mathit{type\ env} \qquad \Phi \vdash \tau :: \mathsf{Type}}{\Phi \vdash \Delta, x \mathbin{\mathbf{:}} \tau \ \mathit{type\ env}} \tag{74}$$

Term formation $\boxed{\Phi; \Delta \vdash e : \tau}$

$$\frac{\Phi \vdash \Delta \ \mathit{type\ env} \qquad x \in \mathrm{dom}(\Delta)}{\Phi; \Delta \vdash x : \Delta(x)} \tag{75}$$

$$\frac{\Phi; \Delta \vdash e : \tau \qquad \Phi \vdash \tau = \tau' :: \mathsf{Type}}{\Phi; \Delta \vdash e : \tau'} \tag{76}$$

$$\frac{\Phi; \Delta, x \mathbin{\mathbf{:}} \tau \vdash e : \tau'}{\Phi; \Delta \vdash (\boldsymbol{\lambda} x \mathbin{\mathbf{:}} \tau.\, e) : \tau \to \tau'} \tag{77}$$

$$\frac{\Phi; \Delta \vdash e_1 : \tau' \to \tau \qquad \Phi; \Delta \vdash e_2 : \tau'}{\Phi; \Delta \vdash e_1\ e_2 : \tau} \tag{78}$$

$$\frac{\begin{array}{c} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \\ \Phi; \Delta \vdash e_i : \tau_i \quad (\forall i \in \{1 \ldots n\}) \end{array}}{\Phi; \Delta \vdash \mathbf{\{} l_1 = e_1 \ldots l_n = e_n \mathbf{\}} : \mathbf{\{} l_1 \mathbf{:} \tau_1 \ldots l_n \mathbf{:} \tau_n \mathbf{\}}} \tag{79}$$

$$\frac{\Phi; \Delta \vdash e : \mathbf{\{} l_1 \mathbf{:} \tau_1 \mathbf{;} \ldots l_n \mathbf{:} \tau_n \mathbf{;} \tau \mathbf{\}}}{\Phi; \Delta \vdash e\mathbf{.}l_i : \tau_i} \tag{80}$$

$$\frac{\Phi, \alpha :: \kappa; \Delta \vdash e : \tau}{\Phi; \Delta \vdash (\mathbf{\Lambda} \alpha::\kappa.\, e) : \forall \alpha::\kappa.\, \tau} \tag{81}$$

$$\frac{\Phi; \Delta \vdash e : \forall \alpha::\kappa.\, \tau \qquad \Phi \vdash \tau' :: \kappa}{\Phi; \Delta \vdash e\; [\tau'] : \tau[\alpha := \tau']} \tag{82}$$

$$\frac{\Phi \vdash \tau :: \mathsf{Type}}{\Phi; \Delta \vdash \textsf{\textbf{abort}}\, [\tau] : \tau} \tag{83}$$

B.2 Properties of static judgments

LEMMA 1 (NORMALIZATION). *Type reductions are strongly normalizing.*

PROOF SKETCH. The type equivalence judgments can be read left-to-right as reductions. To demonstrate that these reductions are strongly normalizing, we view the type language as a simply-typed λ-calculus itself, extended with records (tuples), lists with labeled elements (rows), a base type (Type) and several constants ($\rightarrow$, $\mathbf{\{} \cdot \mathbf{\}}$, $\mathbf{[} \cdot \mathbf{]}$). The binding operators (μ, $\forall$, $\exists$) are also constants, since they are neither introduced nor eliminated by any reduction rule.

Standard proofs for strong normalization of the simply-typed λ-calculus (see, for example, [Goguen 1995]) can be adapted to this type language. □

LEMMA 2 (CONFLUENCE). *Type reductions are confluent.*

PROOF SKETCH. As above, we can adapt a standard proof for confluence of the simply-typed λ-calculus. □

THEOREM 2 (DECIDABILITY). *All static judgments in the previous section are decidable.*

PROOF. Judgments for the formation of kinds, kind environments, types, and type environments are all syntax-directed and trivially decidable.

Type equivalence is not syntax-directed. Since reductions are, however, strongly normalizing (lemma 1) we have an algorithm for deciding type equivalence: reduce τ_1 and τ_2 to normal form, then test whether they are syntactically congruent (modulo renaming of bound variables).

Term formation is syntax-directed except for rule (76), which accounts for type equivalences. If an algorithm always reduces types to normal forms, then the types of two different expressions can be checked for syntactic congruence, and rule (76) can be omitted. □

B.3 Operational semantics

$$\begin{array}{rl} \text{Values } v ::= & \boldsymbol{\lambda} x \mathbf{:} \tau.\, e \mid \mathbf{\{} (l = v)^* \mathbf{\}} \mid \textsf{\textbf{fix}}\, \mathbf{[}\tau\mathbf{]}\; e \mid \textsf{\textbf{inj}}_l^{\tau}\; v \mid \mathbf{\Lambda} \alpha::\kappa.\, e \\ & \mid \langle \alpha::\kappa = \tau, v \mathbf{:} \tau' \rangle \mid \textsf{\textbf{fold}}\; v\; \textsf{\textbf{as}}\; \mu\alpha::\kappa.\, \tau\; \textsf{\textbf{at}}\; \lambda\gamma::\kappa.\, s[\gamma] \end{array}$$

Primitive reductions $\boxed{e \hookrightarrow e'}$

$$\frac{}{(\boldsymbol{\lambda} x{:}\tau.\, e)\ v \hookrightarrow e[x := v]} \quad (84)$$

$$\frac{}{(\{l_1 = v_1 \ldots l_n = v_n\}).l_i \hookrightarrow v_i} \quad (85)$$

$$\frac{}{(\textsf{fix}\,[\tau]\ e).l \hookrightarrow (e\ (\textsf{fix}\,[\tau]\ e)).l} \quad (86)$$

$$\frac{l_i = l'_k}{\begin{array}{l}\textsf{case inj}_{l_i}^{[l_1:\tau_1;\ldots l_n:\tau_n;\tau]}\ v\ \textsf{of} \\ (l'_j\ x_j \Rightarrow e_j)^{j\in\{1\ldots m\}}\ \textsf{else}\ e' \\ \quad \hookrightarrow e_k[x_k := v]\end{array}} \quad (87)$$

$$\frac{l_i \neq l'_k \quad (\forall k \in \{1\ldots m\})}{\begin{array}{l}\textsf{case inj}_{l_i}^{[l_1:\tau_1;\ldots l_n:\tau_n;\tau]}\ v\ \textsf{of} \\ (l'_j\ x_j \Rightarrow e_j)^{j\in\{1\ldots m\}}\ \textsf{else}\ e' \\ \quad \hookrightarrow e'\end{array}} \quad (88)$$

$$\frac{}{\textsf{unfold}\ (\textsf{fold}\ v\ \textsf{as}\ \tau\ \textsf{at}\ \tau_s)\ \textsf{as}\ \tau\ \textsf{at}\ \tau_s \hookrightarrow v} \quad (89)$$

$$\frac{}{(\boldsymbol{\Lambda}\alpha{::}\kappa.\, e)\ [\tau] \hookrightarrow e[\alpha := \tau]} \quad (90)$$

$$\frac{}{\begin{array}{l}\textsf{open}\ \langle \alpha{::}\kappa = \tau', v : \tau\rangle\ \textsf{as}\ \langle \alpha{::}\kappa, x : \tau\rangle\ \textsf{in}\ e' \\ \quad \hookrightarrow e'[\alpha := \tau'][x := v]\end{array}} \quad (91)$$

$$\frac{}{\textsf{abort}\,[\tau] \hookrightarrow \textsf{abort}\,[\tau]} \quad (92)$$

Congruence rules $\boxed{e \hookrightarrow e'}$

$$\frac{e \hookrightarrow e'}{e\ e_2 \hookrightarrow e'\ e_2} \quad (93)$$

$$\frac{e \hookrightarrow e'}{v_1\ e \hookrightarrow v_1\ e'} \quad (94)$$

$$\frac{e \hookrightarrow e'}{\begin{array}{l}\{l_1 = v_1 \ldots l_{i-1} = v_{i-1}\ l_i = e\ l_{i+1} = e_{i+1} \ldots l_n = e_n\} \\ \quad \hookrightarrow \{l_1 = v_1 \ldots l_{i-1} = v_{i-1}\ l_i = e'\ l_{i+1} = e_{i+1} \ldots l_n = e_n\}\end{array}} \quad (95)$$

$$\frac{e \hookrightarrow e'}{e.l \hookrightarrow e'.l} \quad (96)$$

$$\frac{e \hookrightarrow e'}{\textsf{\textbf{inj}}_l^\tau\ e \hookrightarrow \textsf{\textbf{inj}}_l^\tau\ e'} \tag{97}$$

$$\frac{e \hookrightarrow e'}{\begin{array}{l}\textsf{\textbf{case}}\ e\ \textsf{\textbf{of}}\ (l_i\ x_i \Rightarrow e_i)^{i\in\{1\ldots m\}}\ \textsf{\textbf{else}}\ e'' \\ \quad \hookrightarrow \textsf{\textbf{case}}\ e'\ \textsf{\textbf{of}}\ (l_i\ x_i \Rightarrow e_i)^{i\in\{1\ldots m\}}\ \textsf{\textbf{else}}\ e''\end{array}} \tag{98}$$

$$\frac{e \hookrightarrow e'}{\textsf{\textbf{fold}}\ e\ \textsf{\textbf{as}}\ \tau\ \textsf{\textbf{at}}\ \tau_s \hookrightarrow \textsf{\textbf{fold}}\ e'\ \textsf{\textbf{as}}\ \tau\ \textsf{\textbf{at}}\ \tau_s} \tag{99}$$

$$\frac{e \hookrightarrow e'}{\textsf{\textbf{unfold}}\ e\ \textsf{\textbf{as}}\ \tau\ \textsf{\textbf{at}}\ \tau_s \hookrightarrow \textsf{\textbf{unfold}}\ e'\ \textsf{\textbf{as}}\ \tau\ \textsf{\textbf{at}}\ \tau_s} \tag{100}$$

$$\frac{e \hookrightarrow e'}{e\ [\tau] \hookrightarrow e'\ [\tau]} \tag{101}$$

$$\frac{e \hookrightarrow e'}{\langle \alpha{::}\kappa = \tau,\, e : \tau' \rangle \hookrightarrow \langle \alpha{::}\kappa = \tau,\, e' : \tau' \rangle} \tag{102}$$

$$\frac{e \hookrightarrow e'}{\begin{array}{l}\textsf{\textbf{open}}\ e\ \textsf{\textbf{as}}\ \langle \alpha{::}\kappa,\, x : \tau\rangle\ \textsf{\textbf{in}}\ e_1 \\ \quad \hookrightarrow \textsf{\textbf{open}}\ e'\ \textsf{\textbf{as}}\ \langle \alpha{::}\kappa,\, x : \tau\rangle\ \textsf{\textbf{in}}\ e_1\end{array}} \tag{103}$$

B.4 Soundness

LEMMA 3 (SUBSTITUTION OF TERMS). *If* $\Phi; \Delta \vdash e' : \tau'$ *and* $\Phi; \Delta, x : \tau' \vdash e : \tau$, *then* $\Phi; \Delta \vdash e[x := e'] : \tau$.

PROOF. By induction on the derivation of $\Phi; \Delta, x : \tau' \vdash e : \tau$. □

LEMMA 4 (SUBSTITUTION OF TYPES). *If* $\Phi \vdash \tau' :: \kappa$ *and* $\Phi, \alpha :: \kappa; \Delta \vdash e : \tau$, *then* $\Phi; \Delta[\alpha := \tau'] \vdash e[\alpha := \tau'] : \tau[\alpha := \tau']$.

PROOF. By induction on the derivation of $\Phi, \alpha :: \kappa; \Delta \vdash e : \tau$. □

THEOREM 3 (SUBJECT REDUCTION). *If* $e \hookrightarrow e'$ *and* $\Phi; \Delta \vdash e : \tau$ *then* $\Phi; \Delta \vdash e' : \tau$.

PROOF. By induction on the derivation of $e \hookrightarrow e'$.

Case (84). $(\boldsymbol{\lambda} x : \tau.\, e)\ v \hookrightarrow e[x := v]$. From antecedent, $\Phi; \Delta \vdash (\boldsymbol{\lambda} x : \tau.\, e)\ v : \tau'$. By inversion on (78) and (77), $\Phi; \Delta, x : \tau \vdash e : \tau'$, and $\Phi; \Delta \vdash v : \tau$. Finally, $\Phi; \Delta \vdash e[x := v] : \tau'$ using lemma 3.

Case (85). $(\{l_1 = v_1 \ldots l_n = v_n\}).l_i \hookrightarrow v_i$. From antecedent, $\Phi; \Delta \vdash \{l_1 = v_1 \ldots l_n = v_n\}.l_i : \tau$. By inversion on (80) and (79), $\Phi; \Delta \vdash v_i : \tau$.

Case (86). $(\textsf{\textbf{fix}}\,[\tau]\ e).l_i \hookrightarrow (e\ (\textsf{\textbf{fix}}\,[\tau]\ e)).l_i$. From antecedent, $\Phi; \Delta \vdash (\textsf{\textbf{fix}}\,[\tau]\ e).l_i : \tau_i$. By inversion on (80), $\Phi; \Delta \vdash \textsf{\textbf{fix}}\,[\tau]\ e : \{l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau'\}$. By inversion on (3), $\Phi; \Delta \vdash e : \{\tau\} \rightarrow \{\tau\}$, and $\tau = \{l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau'\}$. Using (78), $\Phi; \Delta \vdash e\ (\textsf{\textbf{fix}}\,[\tau]\ e) : \{l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau'\}$. Then, using (80), $\Phi; \Delta \vdash (e\ (\textsf{\textbf{fix}}\,[\tau]\ e)).l : \tau_i$.

Case (87). $\textsf{\textbf{case inj}}_{l_i}^{[l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau]}\ v$ **of** $(l'_j\ x_j \Rightarrow e_j)^{j \in \{1 \ldots m\}}$ **else** $e' \hookrightarrow e_k[x_k := v]$ where $l_i = l'_k$. From antecedent, $\Phi; \Delta \vdash$ **case** $\ldots : \tau'$. By inversion on (9), $\Phi; \Delta, x_k : \tau_i \vdash e_k : \tau'$ and $\Phi; \Delta \vdash$ $\textsf{\textbf{inj}}_{l_i}^{\cdots}\ v : [l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau]$. By inversion on (8) and lemma 3, $\Phi; \Delta \vdash e_k[x_k := v] : \tau'$.

Case (88). $\textsf{\textbf{case inj}}_{l_i}^{[l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau]}\ v$ **of** $(l'_j\ x_j \Rightarrow e_j)^{j \in \{1 \ldots m\}}$ **else** $e' \hookrightarrow e'$ where $l_i \neq l'_k, \forall k \in \{1 \ldots m\}$. From antecedent, $\Phi; \Delta \vdash$ **case** $\ldots : \tau'$. By inversion on (9), $\Phi; \Delta \vdash e' : \tau'$.

Case (89). **unfold** (**fold** v **as** τ **at** τ_s) **as** τ **at** $\tau_s \hookrightarrow v$. From antecedent, $\Phi; \Delta \vdash$ **unfold** $\ldots : \tau'$. By inversion on (11) and (10), $\tau \equiv \mu\alpha{::}\kappa.\, \tau_0$, $\tau' \equiv (\tau_s\ \tau_0)[\alpha := \tau]$, and $\Phi; \Delta \vdash v : (\tau_s\ \tau_0)[\alpha := \tau]$.

Case (90). $(\boldsymbol{\Lambda}\alpha{::}\kappa.\, e)\ [\tau] \hookrightarrow e[\alpha := \tau]$. From antecedent, $\Phi; \Delta \vdash (\boldsymbol{\Lambda}\alpha{::}\kappa.\, e)\ [\tau] : \tau'$. By inversion on (82) and (81), τ' must be in the form of $\tau_1[\alpha := \tau]$, and $\Phi, \alpha :: \kappa; \Delta \vdash e : \tau_1$, and $\Phi \vdash \tau :: \kappa$. Using lemma 4, $\Phi; \Delta \vdash e[\alpha := \tau] : \tau_1[\alpha := \tau]$, i.e. $\Phi; \Delta \vdash e[\alpha := \tau] : \tau'$.

Case (91). **open** $\langle \alpha{::}\kappa = \tau',\ v : \tau \rangle$ **as** $\langle \alpha{::}\kappa,\ x : \tau \rangle$ **in** $e' \hookrightarrow e'[\alpha := \tau'][x := v]$. From antecedent, $\Phi; \Delta \vdash$ **open** $\ldots : \tau_0$. By inversion on (2), $\Phi; \Delta \vdash \langle \alpha{::}\kappa = \tau',\ v : \tau \rangle : \exists\alpha{::}\kappa.\, \tau$, $\Phi, \alpha :: \kappa; \Delta, x : \tau \vdash e' : \tau_0$, and $\Phi \vdash \tau_0 :: \mathsf{Type}$. By inversion on (1), $\Phi; \Delta \vdash v : \tau[\alpha := \tau']$. Using lemma 4, $\Phi; \Delta, x : \tau[\alpha := \tau'] \vdash e'[\alpha := \tau'] : \tau_0[\alpha := \tau']$. Using lemma 3, $\Phi; \Delta \vdash e'[\alpha := \tau'][x := v] : \tau_0[\alpha := \tau']$. This is equivalent to τ_0 since α is not free in τ_0.

Case (92). **abort** $[\tau] \hookrightarrow$ **abort** $[\tau]$. Trivial.

Case (93). $e\ e_2 \hookrightarrow e'\ e_2$ where $e \hookrightarrow e'$. From antecedent, $\Phi; \Delta \vdash e\ e_2 : \tau$. By inversion on (78), $\Phi; \Delta \vdash e : \tau' \to \tau$; and $\Phi; \Delta \vdash e_2 : \tau'$. By induction hypothesis, $\Phi; \Delta \vdash e' : \tau' \to \tau$. Using (78), $\Phi; \Delta \vdash e'\ e_2 : \tau$.

The cases for all the remaining congruence rules (94–103) follow the same pattern: invert some typing rule, apply induction hypothesis, then apply the same typing rule. □

LEMMA 5 (CANONICAL FORMS). *If v is a value and $\Phi; \Delta \vdash v : \tau$ then v has the canonical form given by the following table.*

τ	v
$\tau_1 \to \tau_2$	$\boldsymbol{\lambda} x : \tau_1.\, e$
$\{l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau'\}$	$\{l_1 = v_1, \ldots, l_n = v_n, \ldots\}$
	or **fix** $[l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau']\ e$
$[l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau']$	$\textsf{\textbf{inj}}_{l_i}^{\tau}\ v'$
$s[\mu\alpha{::}\kappa.\, \tau']$	**fold** v' **as** $\mu\alpha{::}\kappa.\, \tau'$ **at** $\lambda\gamma{::}\kappa.\, s[\gamma]$
$\forall\alpha{::}\kappa.\, \tau'$	$\boldsymbol{\Lambda}\alpha{::}\kappa.\, e$
$\exists\alpha{::}\kappa.\, \tau'$	$\langle \alpha{::}\kappa = \tau'',\ v' : \tau' \rangle$

PROOF. By inspection, using lemma 2. □

THEOREM 4 (PROGRESS). *If $\Phi; \Delta \vdash e : \tau$ then either e is a value or $e \hookrightarrow e'$.*

PROOF. By induction on the derivation of $\Phi; \Delta \vdash e : \tau$.

Case (76). direct application of induction hypothesis.

Case (77). $\boldsymbol{\lambda} x\,\boldsymbol{:}\,\tau.\,e$ is a value.

Case (78). $\Phi;\Delta \vdash e_1\ e_2 : \tau$ where $\Phi;\Delta \vdash e_1 : \tau' \rightarrow \tau$. By induction hypothesis, there are three cases: (1) e_1 and e_2 are both values. Using lemma 5, e_1 must have the form $\boldsymbol{\lambda} x\,\boldsymbol{:}\,\tau.\,e$. Using (84), $(\boldsymbol{\lambda} x\,\boldsymbol{:}\,\tau.\,e)\ v \hookrightarrow e[x := v]$. (2) e_1 is a value and $e_2 \hookrightarrow e_2'$; use (94). (3) $e_1 \hookrightarrow e_1'$; use (93).

Case (79). $\Phi;\Delta \vdash \{l_1 = e_1 \ldots l_n = e_n\} : \{l_1 : \tau_1 \ldots l_n : \tau_n\}$. By induction hypothesis, there are two cases: (1) $e_1 \ldots e_n$ are all values; then $\{l_1 = e_1 \ldots l_n = e_n\}$ is a value. (2) $e_i \hookrightarrow e_i'$ for some i; use (95).

Case (80). $\Phi;\Delta \vdash e.l_i : \tau_i$. By induction hypothesis, there are three cases: (1) e is a value, and by lemma 5, it has the form $\{l_1 = v_1 \ldots l_n = v_n\}$. Then, progress can be made using rule (85). (2) e is a value, and by lemma 5, it has the form $\textsf{fix}\,[\tau]\ e$; then use rule (86). (3) $e \hookrightarrow e'$; use (96).

Case (3). $\textsf{fix}\,[\tau]\ e$ is a value.

Case (8). $\Phi;\Delta \vdash \textsf{inj}^{\tau}_{l_i}\ e : \tau$. By induction hypothesis, there are two cases: (1) e is a value; thus $\textsf{inj}^{\tau}_{l_i}\ e$ is a value. (2) $e \hookrightarrow e'$; then, use (97).

Case (9). $\Phi;\Delta \vdash \textsf{case}\ e\ \textsf{of}\ (l_j'\ x_j \Rightarrow e_j)^{j\in\{1\ldots m\}}\ \textsf{else}\ e' : \tau'$. By induction hypothesis, there are two cases: (1) e is a value. According to lemma 5, it has the form $\textsf{inj}^{\tau}_{l_i}\ v$. Thus, either (87) or (88) applies. (2) $e \hookrightarrow e'$; use (98).

Case (10). $\Phi;\Delta \vdash \textsf{fold}\ e\ \textsf{as}\ \tau\ \textsf{at}\ \tau_s : \tau_s\ \tau$. By induction hypothesis, there are two cases: (1) e is a value; then $\textsf{fold}\ e\ \textsf{as}\ \tau\ \textsf{at}\ \tau_s$ is a value. (2) $e \hookrightarrow e'$; then use (99).

Case (11). $\Phi;\Delta \vdash \textsf{unfold}\ e\ \textsf{as}\ \mu\alpha{::}\kappa.\,\tau\ \textsf{at}\ \tau_s : \tau'$. By induction hypothesis, there are two cases: (1) e is a value of type $\tau_s\ (\mu\alpha{::}\kappa.\,\tau)$. By lemma 5 it has the form $\textsf{fold}\ v\ \textsf{as}\ \mu\alpha{::}\kappa.\,\tau\ \textsf{at}\ \tau_s$—use (89). (2) $e \hookrightarrow e'$; then use (100).

Case (81). $\boldsymbol{\Lambda}\alpha{::}\kappa.\,e$ is a value.

Case (82). $\Phi;\Delta \vdash e\ [\tau'] : \tau[\alpha := \tau']$, where $\Phi;\Delta \vdash e : \forall\alpha{::}\kappa.\,\tau$. By induction hypothesis, there are two cases: (1) e is a value. By lemma 5, it must have the form $\boldsymbol{\Lambda}\alpha{::}\kappa.\,e'$; use (90). (2) $e \hookrightarrow e'$; use (101).

Case (1). $\Phi;\Delta \vdash \langle\alpha{::}\kappa = \tau',\ e : \tau\rangle : \exists\alpha{::}\kappa.\,\tau$. By induction hypothesis, there are two cases: (1) e is a value; then $\langle\alpha{::}\kappa = \tau',\ e\,\boldsymbol{:}\,\tau\rangle$ is a value. (2) $e \hookrightarrow e'$; then use (102).

Case (2). $\Phi;\Delta \vdash \textsf{open}\ e\ \textsf{as}\ \langle\alpha{::}\kappa,\ x\,\boldsymbol{:}\,\tau\rangle\ \textsf{in}\ e' : \tau'$. By induction hypothesis, there are two cases: (1) e is a value of type $\exists\alpha{::}\kappa.\,\tau$. By lemma 5, it has the form $\langle\alpha{::}\kappa = \tau',\ e\,\boldsymbol{:}\,\tau\rangle$; use (91). (2) $e \hookrightarrow e'$; use (103).

Case (83). $\Phi;\Delta \vdash \textsf{abort}\,[\tau] : \tau$. Evaluates to $\textsf{abort}\,[\tau]$ using (92).

$\square$

C. PROPERTIES OF THE TRANSLATION

C.1 Contents of field/method vectors

LEMMA 6 (METHOD VECTOR). *mtype*$(\texttt{m}, \texttt{C}) = \texttt{D}_1 \ldots \texttt{D}_n\texttt{->}\texttt{D}_0$ *if and only if* $(\texttt{m}, \texttt{D}_1 \ldots \texttt{D}_n\texttt{->}\texttt{D}_0) \in$ *methvec*$(\texttt{C})$.

PROOF. By induction on the derivation of `C <: Obj`. In the base case, the implication holds trivially. Otherwise, let $CT(\texttt{C}) = \texttt{class C} \lhd \texttt{B}\ \{\ldots\texttt{K}\ \texttt{M}_1 \ldots \texttt{M}_n\}$. We distinguish two cases:

(1) $\mathtt{m}$ is not defined in $\mathtt{M}_1 \ldots \mathtt{M}_n$. ($\Longrightarrow$) Then, $\mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathtt{D}_1 \ldots \mathtt{D}_n\texttt{->}\mathtt{D}_0 = \mathit{mtype}(\mathtt{m},\mathtt{B})$. Using the inductive hypothesis, $(\mathtt{m},\mathtt{D}_1 \ldots \mathtt{D}_n\texttt{->}\mathtt{D}_0)$ is in $\mathit{methvec}(\mathtt{B})$ and thus it is also in $\mathit{methvec}(\mathtt{C})$. ($\Longleftarrow$) $\mathit{addmeth}(\mathtt{B},[\mathtt{M}_1 \ldots \mathtt{M}_n])$ could not have added $\mathtt{m}$, so it must be that $(\mathtt{m},\mathtt{D}_1 \ldots \mathtt{D}_n\texttt{->}\mathtt{D}_0) \in \mathit{methvec}(\mathtt{B})$. Using the inductive hypothesis, $\mathit{mtype}(\mathtt{m},\mathtt{B}) = \mathtt{D}_1 \ldots \mathtt{D}_n\texttt{->}\mathtt{D}_0$ and, in this case, $\mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathit{mtype}(\mathtt{m},\mathtt{B})$.

(2) $\exists j$ such that $\mathtt{M}_j = \mathtt{D}_0\ \mathtt{m}(\mathtt{D}_1\ \mathtt{x}_1 \ldots \mathtt{D}_n\ \mathtt{x}_n)\ \{\uparrow \mathtt{e};\}$. In this case, $\mathit{mtype}(\mathtt{m},\mathtt{C})$ is directly defined as $\mathtt{D}_1 \ldots \mathtt{D}_n\texttt{->}\mathtt{D}_0$.
($\Longrightarrow$) *case* $\mathit{mtype}(\mathtt{m},\mathtt{B}) = \mathtt{C}_1 \ldots \mathtt{C}_n\texttt{->}\mathtt{C}_0$. Then, from class well-formedness we conclude that $\mathtt{C}_i = \mathtt{D}_i$ for $i \in \{0 \ldots n\}$. From the inductive hypothesis, we find that $(\mathtt{m},\mathtt{C}_1 \ldots \mathtt{C}_n\texttt{->}\mathtt{C}_0) \in \mathit{methvec}(\mathtt{B})$. Thus, $(\mathtt{m},\mathtt{D}_1 \ldots \mathtt{D}_n\texttt{->}\mathtt{D}_0) \in \mathit{methvec}(\mathtt{C})$.
($\Longrightarrow$) *case* $\nexists T$ *such that* $\mathit{mtype}(\mathtt{m},\mathtt{B}) = T$. From inductive hypothesis (in the reverse direction), $\nexists T$ such that $(\mathtt{m},T) \in \mathit{methvec}(\mathtt{B})$. Given this, we can show (by induction on j) that $\mathit{addmeth}$ adds $(\mathtt{m},\mathtt{D}_1 \ldots \mathtt{D}_n\texttt{->}\mathtt{D}_0)$ to $\mathit{methvec}(\mathtt{C})$.
($\Longleftarrow$) *case* $(\mathtt{m},\mathtt{C}_1 \ldots \mathtt{C}_n\texttt{->}\mathtt{C}_0) \in \mathit{methvec}(\mathtt{B})$. Therefore, by definition, $(\mathtt{m},\mathtt{C}_1 \ldots \mathtt{C}_n\texttt{->}\mathtt{C}_0) \in \mathit{methvec}(\mathtt{C})$. From class well-formedness, we argue that $\mathtt{C}_i = \mathtt{D}_i$ for $i \in \{0 \ldots n\}$.
($\Longleftarrow$) *case* $\nexists T$ *such that* $(\mathtt{m},T) \in \mathit{methvec}(\mathtt{B})$. Then, $\mathit{addmeth}$ and $\mathit{mtype}(\mathtt{m},\mathtt{C})$ both assign $\mathtt{m}$ the signature $\mathtt{D}_1 \ldots \mathtt{D}_n\texttt{->}\mathtt{D}_0$.

□

LEMMA 7 (FIELD VECTOR). *If* $\mathtt{D}\ \mathtt{f} \in \mathit{fields}(\mathtt{C})$*, then* $(\mathtt{f},\mathtt{D}) \in \mathit{fieldvec}(\mathtt{C})$.

PROOF. By induction on the derivation of $\mathtt{C} \mathrel{\texttt{<:}} \mathtt{Obj}$. □

C.2 Object layout

LEMMA 8 (WELL-KINDED ROWS). *If* $\mathtt{C} \mathrel{\texttt{<:}} \mathtt{A}$*, then*
$\Phi \vdash \mathit{Rows}[\mathtt{C},\mathtt{A}] :: \mathit{kcn} \Rightarrow \mathsf{Type} \Rightarrow \mathit{ktail}[\mathtt{C}] \Rightarrow \mathit{ktail}[\mathtt{A}]$.

PROOF. By induction on the derivation of $\mathtt{C} \mathrel{\texttt{<:}} \mathtt{A}$. Observe that $\vdash \mathit{kcn}\ \mathit{kind}$ and, for any $\mathtt{D} \in \mathit{cn}$, $\vdash \mathit{ktail}[\mathtt{D}]\ \mathit{kind}$. Then, the base case ($\mathtt{C} = \mathtt{A}$) holds trivially. Now, let $\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \triangleleft \mathtt{B}\ \ \{\mathtt{D}_1\ \mathtt{f}_1; \ldots \mathtt{D}_n\ \mathtt{f}_n;\ \mathtt{K}\ldots\}$ and $\mathtt{B} \mathrel{\texttt{<:}} \mathtt{A}$. Using the inductive hypothesis, $\mathit{Rows}[\mathtt{B},\mathtt{A}]$ has kind $\mathit{kcn} \Rightarrow \mathsf{Type} \Rightarrow \mathit{ktail}[\mathtt{B}] \Rightarrow \mathit{ktail}[\mathtt{A}]$ in kind environment Φ. The rule (21) constructs a tuple $\mathsf{tail}' = \{\mathsf{m} = \ldots,\ \mathsf{f} = \ldots\}$. Let $\Phi' = \Phi,\ \mathsf{w} :: \mathit{kcn},\ \mathsf{u} :: \mathsf{Type},\ \mathsf{tail} :: \mathit{ktail}[\mathtt{C}]$. It remains to be shown that tail' has kind $\mathit{ktail}[\mathtt{B}]$ in kind environment Φ'. Consider the f component; the argument for m is similar. Using the definition of $\mathit{ktail}[\mathtt{C}]$ and the tuple selection rule (50), $\Phi' \vdash \mathsf{tail}{\cdot}\mathsf{f} :: \mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathtt{C}))}$. Using the definition of kcn and class table well-formedness, $\Phi' \vdash \mathsf{w}{\cdot}\mathtt{D}_n :: \mathsf{Type}$. Finally, the row formation rule (5) assigns kind $\mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathtt{C}))-\{\mathtt{f}_n\}}$ to the row $(\mathtt{f}_n : \mathsf{w}{\cdot}\mathtt{D}_n\,;\mathsf{tail}{\cdot}\mathsf{f})$. Iterate for each label; the resulting row has kind $\mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathtt{C}))-\{\mathtt{C},\mathtt{f}_1\ldots\mathtt{f}_n\}}$ which is the same as $\mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathtt{B}))}$.

□

LEMMA 9 (TAIL POSITION). *If* $\mathtt{C} \mathrel{\texttt{<:}} \mathtt{B}$*,* $\Phi \vdash \mathsf{w} :: \mathit{kcn}$*,* $\Phi \vdash \mathsf{tail} :: \mathit{ktail}[\mathtt{C}]$*, and* $\Phi \vdash \mathsf{self} :: \mathsf{Type}$*, then* $(\mathit{Rows}[\mathtt{C},\mathtt{B}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}){\cdot}\mathsf{m}\ \mathsf{self}$ *has the form* $(\ldots\,;\mathsf{tail}{\cdot}\mathsf{m}\ \mathsf{self})$ *and* $(\mathit{Rows}[\mathtt{C},\mathtt{B}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}){\cdot}\mathsf{f}$ *has the form* $(\ldots\,;\mathsf{tail}{\cdot}\mathsf{f})$.

PROOF. By inspection. □

Lemma 10 (Method layout). *If* $\Phi \vdash \mathsf{w} :: \mathit{kcn}$, $\Phi \vdash \mathsf{tail} :: \mathit{ktail}[\mathtt{C}]$, $\Phi \vdash \mathsf{self} :: \mathsf{Type}$, *and* $(\mathtt{m}, T) \in \mathit{methvec}(\mathtt{C})$, *then* $(\mathit{Rows}[\mathtt{C}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}) \cdot \mathsf{m}\ \mathsf{self} = (\ldots\ \mathbf{;}\ \mathtt{m}\,\mathbf{:}\,\mathsf{self} \longrightarrow \mathit{Ty}[\mathsf{self}; \mathsf{w}; T]\ \mathbf{;}\ \ldots\ \mathbf{;}\, \mathsf{tail} \cdot \mathsf{m}\ \mathsf{self})$.

Proof. By induction on derivation of $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{Obj}$. $\mathit{methvec}(\mathtt{Obj})$ is empty, so the base case holds trivially. Otherwise, let $\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{B}\ \{\ldots\}$.

Case $(m, T) \in \mathit{methvec}(B)$. Let $\mathsf{tail}' = \{\mathsf{m} = \lambda \mathsf{self} :: \mathsf{Type}. \ldots, \mathsf{f} = \ldots\}$, as given in rule (21); according to lemma 8, this has kind $\mathit{ktail}[\mathtt{B}]$. Invoking the inductive hypothesis (with tail') we find that

$$\begin{aligned}&(\mathit{Rows}[\mathtt{B}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}') \cdot \mathsf{m}\ \mathsf{self}\\ &= (\ldots\ \mathbf{;}\ \mathtt{m}\,\mathbf{:}\,\mathsf{self} \longrightarrow \mathit{Ty}[\mathsf{self}; \mathsf{w}; T]\ \mathbf{;}\ \ldots\ \mathbf{;}\, \mathsf{tail}' \cdot \mathsf{m}\ \mathsf{self})\end{aligned}$$

Then, expanding the definition we get

$$\begin{aligned}&(\mathit{Rows}[\mathtt{C}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}) \cdot \mathsf{m}\ \mathsf{self}\\ &= (\ldots\ \mathbf{;}\ \mathtt{m}\,\mathbf{:}\,\mathsf{self} \longrightarrow \mathit{Ty}[\mathsf{self}; \mathsf{w}; T]\ \mathbf{;}\ \ldots\ldots\ \mathbf{;}\, \mathsf{tail} \cdot \mathsf{m}\ \mathsf{self})\end{aligned}$$

Case $(m, T) \notin \mathit{methvec}(B)$. Then, $\mathtt{m}$ must be one of the names $\mathtt{m}_1 \ldots \mathtt{m}_n$ enumerated in the definition. In this case, the row $\mathsf{tail}' \cdot \mathsf{m}\ \mathsf{self}$ will contain an element $\mathtt{m}$ of type $\mathsf{self} \longrightarrow \mathit{Ty}[\mathsf{self}; \mathsf{w}; T]$. This tail' is passed to $\mathit{Rows}[\mathtt{B}, \mathtt{Obj}]$, but according to lemma 9, it will still appear in the result.

□

Lemma 11 (Field layout). *If* $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{Obj}$, $\Phi \vdash \mathsf{w} :: \mathit{kcn}$, $\Phi \vdash \mathsf{tail} :: \mathit{ktail}[\mathtt{C}]$, *and* $\mathit{fieldvec}(\mathtt{C}) = \mathit{fieldvec}(\mathtt{Obj}) +\!\!+ [(l_1, \mathtt{D}_1) \ldots (l_n, \mathtt{D}_n)]$, *then* $\mathit{Rows}[\mathtt{C}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail} = l_1\,\mathbf{:}\,\mathsf{w} \cdot \mathtt{D}_1\ \mathbf{;}\ \ldots\ l_n\,\mathbf{:}\,\mathsf{w} \cdot \mathtt{D}_n\ \mathbf{;}\, \mathsf{tail} \cdot \mathsf{f}$.

Proof. By induction on the derivation of $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{Obj}$. Similar to the proof of lemma 10. □

Lemma 12 (Rows coherence). *If* $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{A}$, $\Phi \vdash \mathsf{u} :: \mathsf{Type}$, $\Phi \vdash \mathsf{w} :: \mathit{kcn}$, *and* $\Phi \vdash \mathsf{tail} :: \mathit{ktail}[\mathtt{C}]$, *then* $\mathit{Rows}[\mathtt{A}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ (\mathit{Rows}[\mathtt{C}, \mathtt{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}) = \mathit{Rows}[\mathtt{C}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}$.

Proof. By induction on the derivation of $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{A}$. The base case ($\mathtt{C} = \mathtt{A}$) holds trivially. Now, let $\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{B}\ \{\ldots\}$ where $\mathtt{B} \mathrel{\mathtt{<:}} \mathtt{A}$. The rule for $\mathit{Rows}[\mathtt{C}, \mathtt{A}]$ defines a tuple $\{\mathsf{f} = \ldots, \mathsf{m} = \ldots\}$ which we will call tail'. Specifically, $\mathit{Rows}[\mathtt{C}, \mathtt{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail} = \mathit{Rows}[\mathtt{B}, \mathtt{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}'$. Now, using tail' in the inductive hypothesis, we find that $\mathit{Rows}[\mathtt{A}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ (\mathit{Rows}[\mathtt{B}, \mathtt{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}') = \mathit{Rows}[\mathtt{B}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}'$. According to the definition, $\mathit{Rows}[\mathtt{B}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}' = \mathit{Rows}[\mathtt{C}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}$, where tail' is the same as above. Substituting equals for equals (twice) yields

$$\mathit{Rows}[\mathtt{A}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ (\mathit{Rows}[\mathtt{C}, \mathtt{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}) = \mathit{Rows}[\mathtt{C}, \mathtt{Obj}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}$$

□

C.3 Object transformations

Lemma 13 (Well-typed pack). *If* $\Phi \vdash \mathsf{tail} :: \mathit{ktail}[\mathtt{C}]$ *and* $\Phi; \Delta \vdash e : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}$, *then* $\Phi; \Delta \vdash \textsc{pack}[\mathtt{C}; \mathsf{u}; \mathsf{tail}; e] : (\mathit{World}\ \mathsf{u}) \cdot \mathtt{C}$.

Proof. By inspection of the definitions, using the term formation rules for fold (10) and pack (1). □

LEMMA 14 (WELL-TYPED UPCAST). *If* $\Phi; \Delta \vdash e : (\mathit{World}\ \mathsf{u}) \cdot \mathtt{C}$ *and* $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{A}$, *then* $\Phi; \Delta \vdash \text{UPCAST}[\mathtt{C}; \mathtt{A}; \mathsf{u}; e] : (\mathit{World}\ \mathsf{u}) \cdot \mathtt{A}$.

PROOF. By inspection of the definitions, using the term formation rules for open (2) and unfold (11) and lemmas 8, 12, and 13. Unfolding e produces a term of type $\mathit{ObjTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}$. Opening this introduces type variable $\mathsf{tail} :: \mathit{ktail}[\mathtt{C}]$ and term variable $x : \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}$; call this new environment $\Phi'; \Delta'$. The body of the open contains a PACK expression, but in order to use lemma 13, we must establish the following:

(1) $\Phi' \vdash \mathit{Rows}[\mathtt{C}, \mathtt{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail} :: \mathit{ktail}[\mathtt{A}]$, and
(2) $\Phi'; \Delta' \vdash x : \mathit{SelfTy}[\mathtt{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{Rows}[\mathtt{C}, \mathtt{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})$.

The first follows from lemma 8. The second reduces to

$$\begin{aligned}\Phi' \vdash\ & \mathit{SelfTy}[\mathtt{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{Rows}[\mathtt{C}, \mathtt{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) = \\ & \mathit{SelfTy}[\mathtt{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail} :: \mathsf{Type}\end{aligned}$$

By expanding the definition of $\mathit{SelfTy}[\cdot]$ and applying equivalence rules, it reduces again to

$$\begin{aligned}\Phi' \vdash\ & \mathit{Rows}[\mathtt{A}, \mathtt{Obj}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{Rows}[\mathtt{C}, \mathtt{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) = \\ & \mathit{Rows}[\mathtt{C}, \mathtt{Obj}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail} :: \mathit{ktail}[\mathtt{Obj}]\end{aligned}$$

which follows from lemma 12. Finally, lemma 13 can be invoked to show that the result of the upcast has type $(\mathit{World}\ \mathsf{u}) \cdot \mathtt{A}$.

□

C.4 Type preservation for expressions

FJ contexts are translated to type environments as follows:

$$\begin{aligned}\text{ENV}[\mathsf{u}; \Gamma, \mathtt{x} : \mathtt{D}] &= \text{ENV}[\mathsf{u}; \Gamma], \mathtt{x} : (\mathit{World}\ \mathsf{u}) \cdot \mathtt{D} \\ \text{ENV}[\mathsf{u}; \circ] &= \circ\end{aligned}$$

LEMMA 15 (CONTEXT TRANSLATION). *If* $\Phi \vdash \mathsf{u} :: \mathsf{Type}$ *and* $\mathrm{range}(\Gamma) \subseteq \mathit{cn}$, *then* $\Phi \vdash \text{ENV}[\mathsf{u}; \Gamma]$ *type env*.

PROOF. By inspection. □

THEOREM 5 (TYPE PRESERVATION). *If* $\Phi \vdash \mathsf{u} :: \mathsf{Type}$, $\Phi; \Delta \vdash \mathsf{classes} : \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\}$ *and* $\Gamma \vdash \mathsf{e} \in \mathtt{C}$, *then* $\Phi; \Delta, \text{ENV}[\mathsf{u}; \Gamma] \vdash \text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathsf{e}] : (\mathit{World}\ \mathsf{u}) \cdot \mathtt{C}$.

PROOF. By induction on the structure of e. We use the following abbreviations: Δ_Γ for $\text{ENV}[\mathsf{u}; \Gamma]$; Δ'_Γ for Δ, Δ_Γ; and e for $\text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathsf{e}]$.

Case VAR. $\mathsf{e} = \mathtt{x}$ and, from (T-VAR), $\mathtt{C} = \Gamma(\mathtt{x})$. Thus, $\Delta_\Gamma(\mathtt{x}) = (\mathit{World}\ \mathsf{u}) \cdot \mathtt{C}$ and $\Phi; \Delta'_\Gamma \vdash e : (\mathit{World}\ \mathsf{u}) \cdot \mathtt{C}$.

Case FIELD. $\mathsf{e} = \mathsf{e}_0.\mathtt{f}_i$ and $\mathtt{C} = \mathtt{C}_i$, where $\Gamma \vdash \mathsf{e}_0 \in \mathtt{C}_0$ and $\mathit{fields}(\mathtt{C}_0) = \mathtt{C}_1\ \mathtt{f}_1 \ldots \mathtt{C}_n\ \mathtt{f}_n$. By inductive hypothesis, $\Phi; \Delta'_\Gamma \vdash e_0 : (\mathit{World}\ \mathsf{u}) \cdot \mathtt{C}_0$. The code in (FIELD) unfolds and opens e_0. Using the same argument as in the proof of lemma 14, this introduces the type variable $\mathsf{tail} :: \mathit{ktail}[\mathtt{C}_0]$ and term variable $x : \mathit{SelfTy}[\mathtt{C}_0]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}$; call this new environment $\Phi'; \Delta''_\Gamma$. Unfolding x yields a term of type

$$\{\mathsf{vtab} : \ldots ; (\mathit{Rows}[\mathtt{C}_0, \mathtt{Obj}]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{f}\}$$

Using lemma 7, $(\mathtt{f}_i, \mathtt{C}_i) \in \mathit{fieldvec}(\mathtt{C}_0)$. Using lemma 11, we find that the row $(\mathit{Rows}[\mathtt{C}_0, \mathtt{Obj}]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{f}$ contains a binding $\mathtt{f}_i : (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}_i$. Using record selection, $\Phi; \Delta''_\Gamma \vdash (\textbf{unfold}\, x \ldots).\mathtt{f}_i : (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}_i$. Exiting the scope of the open, we conclude $\Phi; \Delta'_\Gamma \vdash e : (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}_i$.

Case INVOKE. $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1 \ldots \mathtt{e}_n)$, where $\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0$, $\mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \mathtt{D}_1 \ldots \mathtt{D}_n\text{->}\mathtt{C}$, $\Gamma \vdash \mathtt{e}_i \in \mathtt{C}_i$, and $\mathtt{C}_i <: \mathtt{D}_i$, for all $i \in \{1 \ldots n\}$. We use the inductive hypothesis on $\mathtt{e}_0$, and the same unfold-open-unfold argument as in the previous case. Selecting vtab yields a term of type $\{(\mathit{Rows}[\mathtt{C}_0, \mathtt{Obj}]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{m}\ (\mathit{SelfTy}[\mathtt{C}_0]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})\}$ Using lemma 6, $(\mathtt{m}, \mathtt{D}_1 \ldots \mathtt{D}_n\text{->}\mathtt{C}) \in \mathit{methvec}(\mathtt{C}_0)$. Using lemma 10, the above record contains a binding

$$\begin{aligned} &\mathtt{m} : (\mathit{SelfTy}[\mathtt{C}_0]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) \longrightarrow \mathit{Ty}[\mathsf{self}; \mathit{World}\ \mathsf{u}; \mathtt{D}_1 \ldots \mathtt{D}_n\text{->}\mathtt{C}] \\ = &\mathtt{m} : (\mathit{SelfTy}[\mathtt{C}_0]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) \longrightarrow (\mathit{World}\ \mathsf{u})\cdot\mathtt{D}_1 \longrightarrow \ldots \\ &\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{World}\ \mathsf{u})\cdot\mathtt{D}_n \longrightarrow (\mathit{World}\ \mathsf{u})\cdot\mathtt{C} \end{aligned}$$

Thus, selecting $\mathtt{m}$ and applying it to x yields a term of type

$$(\mathit{World}\ \mathsf{u})\cdot\mathtt{D}_1 \longrightarrow \ldots (\mathit{World}\ \mathsf{u})\cdot\mathtt{D}_n \longrightarrow (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$$

Now, for each i in $1 \ldots n$, we use the inductive hypothesis on e_i, concluding that $\Phi; \Delta'_\Gamma \vdash e_i : (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}_i$. Using this and $\mathtt{C}_i <: \mathtt{D}_i$, lemma 14 l tells us that $\Phi; \Delta'_\Gamma \vdash$ $\textsc{upcast}[\mathtt{C}_i; \mathtt{D}_i; \mathsf{u}; e_i] : (\mathit{World}\ \mathsf{u})\cdot\mathtt{D}_i$ Finally, using the application formation rule n times, $\Phi; \Delta'_\Gamma \vdash e : (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$.

Case NEW. $\mathtt{e} = \mathtt{new}\ \mathtt{C}(\mathtt{e}_1 \ldots \mathtt{e}_n)$, where $\Gamma \vdash \mathtt{e}_i \in \mathtt{C}_i$, $\mathit{fields}(\mathtt{C}) = \mathtt{D}_1\ \mathtt{f}_1 \ldots \mathtt{D}_n\ \mathtt{f}_n$, and $\mathtt{C}_i <: \mathtt{D}_i$ for all i in $1 \ldots n$. From the premise $\Phi; \Delta \vdash \mathsf{classes} : \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\}$ using the rules for selection (of C), application, and selection (of new), the new component has type $(\mathit{World}\ \mathsf{u})\cdot\mathtt{D}_1 \longrightarrow \ldots (\mathit{World}\ \mathsf{u})\cdot\mathtt{D}_n \longrightarrow (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$. Just as in the previous case, we use the inductive hypothesis and lemma 14 on each e_i. Again, using the application formation rule n times yields $\Phi; \Delta'_\Gamma \vdash e : (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$.

Case UPCAST. follows from inductive hypothesis and lemma 14.

Case DNCAST. $\mathtt{e} = (\mathtt{C})\mathtt{e}_0$ where $\Gamma \vdash \mathtt{e}_0 \in \mathtt{D}$. We use the inductive hypothesis on $\mathtt{e}_0$ and the usual unfold-open-unfold sequence. We select $\mathsf{dyncast}$ from the vtab and self-apply; this produces a polymorphic function of type

$$\forall\alpha.\,(\mathsf{u} \longrightarrow \textbf{maybe}\ \alpha) \longrightarrow \textbf{maybe}\ \alpha$$

Next we instantiate α with $(\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$ and apply to the class tag, which the correct type: $\mathsf{u} \longrightarrow \textbf{maybe}\ (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$. The result has type $\textbf{maybe}\ (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$, and using the case formation rule, the first branch has type $(\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$. The other branch aborts evaluation, but is regarded as having the same type. So, finally, $\Phi; \Delta'_\Gamma \vdash e : (\mathit{World}\ \mathsf{u})\cdot\mathtt{C}$.

□

C.5 Class components

LEMMA 16 (WELL-TYPED CONSTRUCTOR). *If* $\Phi \vdash \mathsf{u} :: \mathsf{Type}$ *and* $\Phi; \Delta \vdash \mathsf{vtab} : \mathit{Dict}[\mathtt{C}]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{SelfTy}[\mathtt{C}]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathit{Empty}[\mathtt{C}])$, *then* $\Phi; \Delta \vdash \textsc{new}[\mathtt{C}; \mathsf{u}; \mathsf{vtab}] : \mathit{Ctor}[\mathtt{C}]\,(\mathit{World}\ \mathsf{u})$.

PROOF. By inspection, using lemma 11. □

LEMMA 17 (WELL-TYPED DICTIONARY). *If* $\Phi \vdash \mathsf{u} :: \mathsf{Type}$,
$\Phi; \Delta \vdash \mathsf{inj} : (\mathit{World}\ \mathsf{u}){\cdot}\mathtt{C}{\longrightarrow}\mathsf{u}$, *and* $\Phi; \Delta \vdash \mathsf{classes} : \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\}$, *then*
$\Phi; \Delta \vdash \textsc{dict}[\mathtt{C}; \mathsf{u}; \mathsf{inj}; \mathsf{classes}] : \forall \mathsf{tail}.\ \mathit{Dict}[\mathtt{C}]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{SelfTy}[\mathtt{C}]\,(\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})$.

PROOF. By inspection, using lemma 10. □

THEOREM 6 (WELL-TYPED CLASS DECLARATION). $\Phi; \Delta \vdash \textsc{cdec}[\mathtt{C}] : \mathit{ClassF}[\mathtt{C}]$

PROOF. By inspection, using lemmas 16 and 17 for the non-trivial class components. □

ACKNOWLEDGMENTS

Dachuan Yu wrote the operational semantics for the target language (§B.3) and supplied many details for the soundness proofs (§B.4). We also wish to thank the anonymous referees for their many helpful comments.

REFERENCES

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Springer, New York.

ABADI, M., CARDELLI, L., AND VISWANATHAN, R. 1996. An interpretation of objects and object types. In *Proc. Symp. on Principles of Programming Languages*. ACM, New York, 396–409.

ABADI, M. AND FIORE, M. P. 1996. Syntactic considerations on recursive types. In *Proc. 11th Annual IEEE Symp. on Logic in Computer Science*. 242–252.

BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications*. ACM, New York, 183–200.

BRUCE, K. B. 1994. A paradigmatic object-oriented programming language: Design, static typing and semantics. *J. Functional Programming 4,* 2, 127–206.

BRUCE, K. B., CARDELLI, L., AND PIERCE, B. C. 1999. Comparing object encodings. *Information and Computation 155,* 1–2, 108–133.

CANNING, P., COOK, W., HILL, W., OLTHOFF, W., AND MITCHELL, J. C. 1989. F-bounded polymorphism for object-oriented programming. In *Proc. Int'l Conf. on Functional Programming and Computer Architecture*. ACM, 273–280.

CARDELLI, L. AND LEROY, X. 1990. Abstract types and the dot notation. In *Proc. IFIP Working Conf. on Programming Concepts and Methods*. Israel, 466–491.

CRARY, K. 1999. Simple, efficient object encoding using intersection types. Tech. Rep. CMU-CS-99-100, Carnegie Mellon University, Pittsburgh. January.

CRARY, K., HARPER, R., AND PURI, S. 1999. What is a recursive module? In *Proc. Conf. on Programming Language Design and Implementation*. ACM, New York.

EIFRIG, J., SMITH, S., TRIFONOV, V., AND ZWARICO, A. 1995. An interpretation of typed OOP in a language with state. *Lisp and Symbolic Comput. 8,* 4, 357–397.

FISHER, K. AND MITCHELL, J. C. 1998. On the relationship between classes, objects and data abstraction. *Theory and Practice of Object Systems 4,* 1, 3–25.

FISHER, K. AND REPPY, J. 1999. The design of a class mechanism for MOBY. In *Proc. Conf. on Programming Language Design and Implementation*. ACM, New York.

FISHER, K., REPPY, J., AND RIECKE, J. G. 2000. A calculus for compiling and linking classes. In *Proc. European Symp. on Program.* 135–149.

Girard, J. Y. 1972. Interpretation fonctionnelle et elimination des coupures dans l'arithmetique d'ordre superieur. Ph.D. thesis, University of Paris VII.

Glew, N. 2000a. An efficient class and object encoding. In *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications*. ACM.

Glew, N. 2000b. Low-level type systems for modularity and object-oriented constructs. Ph.D. thesis, Cornell University.

Goguen, H. 1995. Typed operational semantics. In *Typed Lambda Calculi and Applications*, M. Dezani-Ciancaglini and G. Plotkin, Eds. LNCS, vol. 902. Springer-Verlag, Berlin, 186–200.

Harper, R. and Morrisett, G. 1995. Compiling polymorphism using intensional type analysis. In *Proc. Symp. on Principles of Programming Languages*. ACM, New York, 130–141.

Harper, R. and Stone, C. 1998. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, Cambridge, Mass.

Igarashi, A. and Pierce, B. C. 2001. On inner classes. *Information and Computation ?,* ? (to appear).

Igarashi, A., Pierce, B. C., and Wadler, P. 1999. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications*. ACM, New York, 132–146.

Kamin, S. 1988. Inheritance in Smalltalk-80: A denotational definition. In *Proc. Symp. on Principles of Programming Languages*. ACM, New York, 80–87.

Krall, A. and Grafl, R. 1997. CACAO—a 64-bit Java VM Just-In-Time compiler. In *Proc. ACM PPoPP'97 Workshop on Java for Science and Engineering Computation.*

League, C., Shao, Z., and Trifonov, V. 1999. Representing Java classes in a typed intermediate language. In *Proc. Int'l Conf. Functional Programming*. ACM, Paris, 183–196.

League, C., Trifonov, V., and Shao, Z. 2001a. Functional Java bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

League, C., Trifonov, V., and Shao, Z. 2001b. Type-preserving compilation of Featherweight Java. In *Proc. Int'l Workshop on Foundations of Object-Oriented Languages*. London. Expanded version to appear in *ACM Trans. on Programming Languages and Systems*.

Mitchell, J. C. and Plotkin, G. D. 1988. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems 10,* 3 (July), 470–502.

Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., and Zdancewic, S. 1999. TALx86: A realistic typed assembly language. In *Proc. Workshop on Compiler Support for Systems Software*. ACM, New York, 25–35.

Morrisett, G., Tarditi, D., Cheng, P., Stone, C., Harper, R., and Lee, P. 1996. The TIL/ML compiler: Performance and safety through types. In *Proc. Workshop on Compiler Support for Systems Software*. ACM, New York.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems 21,* 3 (May), 528–569.

Necula, G. C. 1997. Proof-carrying code. In *Proc. Symp. on Principles of Programming Languages*. ACM, Paris, 106–119.

Necula, G. C. and Lee, P. 1996. Safe kernel extensions without run-time checking. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*. ACM, Seattle, 229–243.

Peyton Jones, S. L., Hall, C., Hammond, K., Partain, W., and Wadler, P. 1992. The Glasgow Haskell Compiler: A technical overview. In *Proc. UK Joint Framework for Inform. Tech.*

Pierce, B. C. and Turner, D. N. 1994. Simple type-theoretic foundations for object-oriented programming. *J. Functional Programming 4,* 2 (April), 207–247.

Proebsting, T. A., Townsend, G., Bridges, P., Hartman, J. H., Newsham, T., and Watterson, S. A. 1997. Toba: Java for applications: A way ahead of time (WAT) compiler. In *Proc. Third Conf. on Object-Oriented Technologies and Systems (COOTS'97)*.

Rémy, D. 1993. Syntactic theories and the algebra of record terms. Tech. Rep. 1869, INRIA.

Rémy, D. and Vouillon, J. 1997. Objective ML: A simple object-oriented extension of ML. In *Proc. Symp. on Principles of Programming Languages*. ACM, New York, 40–53.

Reynolds, J. C. 1974. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*. LNCS, vol. 19. Springer-Verlag, Berlin, 408–425.

Shao, Z. 1997. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03.

Shao, Z. and Appel, A. W. 1995. A type-based compiler for Standard ML. In *Proc. Conf. on Programming Language Design and Implementation*. ACM, La Jolla, 116–129.

Shao, Z., League, C., and Monnier, S. 1998. Implementing typed intermediate languages. In *Proc. Int'l Conf. Functional Programming*. ACM, Baltimore, 313–323.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. 1996. TIL: A type-directed optimizing compiler for ML. In *Proc. Conf. on Programming Language Design and Implementation*. ACM, New York.

Vanderwaart, J. C. 1999. Typed intermediate representations for compiling object-oriented languages. Williams College Senior Honors Thesis.

Wright, A., Jagannathan, S., Ungureanu, C., and Hertzmann, A. 1998. Compiling Java to a typed lambda-calculus: A preliminary report. In *Proc. Int'l Workshop on Types in Compilation*. LNCS, vol. 1473. Springer, Berlin, 1–14.

Received May 2001