Representing Java Classes in a Typed Intermediate Language*

Christopher League Zhong Shao Valery Trifonov
Dept. of Computer Science
Yale University
New Haven, CT 06520
{league,shao,trifonov}@cs.yale.edu

Abstract

We propose a conservative extension of the polymorphic lambda calculus ($F^\omega$) as an intermediate language for compiling languages with name-based class and interface hierarchies. Our extension enriches standard $F^\omega$ with recursive types, existential types, and row polymorphism, but only ordered records with no subtyping. Basing our language on $F^\omega$ makes it also a suitable target for translation from other higher-order languages; this enables the safe interoperation between class-based and higher-order languages and the reuse of common type-directed optimization techniques, compiler back ends, and runtime support.

We present the formal semantics of our intermediate language and illustrate its features by providing a formal translation from a subset of Java, including classes, interfaces, and private instance variables. The translation preserves the name-based hierarchical relation between Java classes and interfaces, and allows access to private instance variables of parameters of the same class as the one defining the method. It also exposes the details of method invocation and instance variable access and allows many standard optimizations to be performed on the object-oriented code.

1 Introduction

The explosive growth of the World Wide Web has, as the Java phenomenon demonstrates, induced newfound interest in mobile computation for “programming the Web.” In this domain, the safety and security properties of programs are more crucial than ever before. Recent work demonstrates a clear connection between security properties and formal type systems [28, 20, 24].

Type checking has long been used to ensure certain properties about the runtime behavior of programs written in strongly typed languages. In the conventional model, however, type information is discarded immediately after type checking. We must therefore trust that the compiler—through its many transformations and optimizations—faithfully preserves the source language semantics. Furthermore, given only the object code (without type information), it may be difficult or impossible to verify such properties. Thus, it is becoming increasingly important to preserve full type information throughout the compilation process.

The Java Virtual Machine Language (JVML) [25] was designed to address these issues. JVML byte code contains sufficient type information for an automatic verifier to prove memory safety and other properties. The translation from Java to JVML is mostly type-preserving.

While JVML does contain type information and submits to verification, it has several drawbacks. First, JVML may not be a good fit for source languages other than Java [27, 3]. For example, JVML does not provide direct support for tail-recursion, higher-order functions, and polymorphic functions, making it impractical for implementing functional languages. Second, JVML is designed for bytecode interpretation, not as a compiler intermediate language. The JVML instruction set is based on a stack machine so it is difficult to perform standard optimizations on JVML. Third, JVML has a complex semantics so it does not provide good support for formal reasoning. Much cutting edge research on security and information flow [28, 20, 24] is hard to incorporate into the JVML-based framework.

The FLINT project at Yale [34, 35] takes a different approach. We aim to build a compiler infrastructure for HOT (higher-order and typed) languages. Our goal is to use a richly typed intermediate language (based on the polymorphic $\lambda$-calculus $F^\omega$ [18, 33]) as a common target for compiling various source languages (e.g., Java, ML). Our system would allow the reuse of common type-directed optimization techniques, compiler back ends, and runtime support.

Building a production-quality type-preserving compiler is by no means trivial, especially in the presence of advanced language features such as parametric polymorphism and higher-order modules. Recent advances in compiler technology, however, are making type-preserving compilation a reality [37, 36]; the current version of FLINT has been in widespread use as the intermediate language of the SML/NJ compiler [2] since January 1997.

Extending our typed intermediate language to handle Java classes poses many new challenges. First, the Java type system is dramatically different from the $F^\omega$ type system. Naively combining Java and FLINT could easily lead to an incomprehensible language. The challenge is to abstract the commonality and to find a synergy between the two languages.

Another challenge in modeling Java classes is to find an object encoding that is faithful to the Java semantics yet still supports efficient implementation. Existing encodings of object-oriented features in typed $\lambda$-calculi [21, 5] use records of functions as dictionaries and existential types for dynamic binding. These encodings, however, do not support Java-like name-based class hierarchies; two different class types with exactly same set of methods and fields are considered as equivalent by these encodings. This

---

*This research was sponsored in part by the Defense Advanced Research Projects Agency ITO under the title “Software Evolution using HOT Language Technology,” DARPA Order No. D888, issued under Contract No. F30602-96-2-0232, and in part by NSF Grant CCR-9633390. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

### 1. REPORT DATE
SEP 1999

### 2. REPORT TYPE

### 3. DATES COVERED
-

### 4. TITLE AND SUBTITLE
Representing Java Classes in a Typed Intermediate Language

### 5a. CONTRACT NUMBER

### 5b. GRANT NUMBER

### 5c. PROGRAM ELEMENT NUMBER

### 5d. PROJECT NUMBER

### 5e. TASK NUMBER

### 5f. WORK UNIT NUMBER

### 6. AUTHOR(S)

### 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency, 3701 North Fairfax Dr, Arlington, VA, 22203-1714

### 8. PERFORMING ORGANIZATION REPORT NUMBER

### 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

### 10. SPONSOR/MONITOR’S ACRONYM(S)

### 11. SPONSOR/MONITOR’S REPORT NUMBER(S)

### 12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

### 13. SUPPLEMENTARY NOTES

### 14. ABSTRACT
see report

### 15. SUBJECT TERMS

### 16. SECURITY CLASSIFICATION OF:

<table>
<thead>
<tr>
<th>a. REPORT</th>
<th>b. ABSTRACT</th>
<th>c. THIS PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>unclassified</td>
<td>unclassified</td>
<td>unclassified</td>
</tr>
</tbody>
</table>

### 17. LIMITATION OF ABSTRACT

### 18. NUMBER OF PAGES
14

### 19a. NAME OF RESPONSIBLE PERSON

[Standard Form 298 (Rev. 8-98)]
Prescribed by ANSI Std Z39-18
is not acceptable because class and interface names in Java play a critical role during type-checking, linking, loading, runtime execution, and casting. The explicit class hierarchy is also crucial for many important optimizations on object-oriented languages [10].

Furthermore, because existing object encodings are intended as theoretical models rather than as intermediate representations, they often use non-trivial coercions in order to simplify the type system. For example, Pierce and Turner [29] model inheritance using a pair of functions for coercing between the superclass and subclass views of an object. For the purposes of compilation, the cost of calling an unknown function to obtain a different view of an object is unacceptable. For Java in particular, these coercions would typically be identity functions. In the presence of separate compilation, we could not expect optimizations to be able to replace these general coercions with more efficient operations.

Finally, the implicit subtyping typically used in these encodings can make them rather complex, in some cases rendering type checking undecidable. We want a more conservative extension, less powerful than, say, $F^ω$ [9], but able to support fast type-checking, an explicit class hierarchy, and efficient implementation.

Our main contribution is a formal translation of Java classes, interfaces, and privacy into a variant of $F^ω$ using simple and well-known extensions. Although we use several familiar techniques, their combination to support the efficient compilation of an interesting subset of Java is novel.

We use row polymorphism [31] to implement Java inheritance and to allow objects of subclasses to masquerade as superclasses. We use existential types with dot notation [8] to model Java's interfaces, and access control, with several simplifications that permit a concise formal semantics and a comprehensible translation.

Although our intermediate language contains full type information, the implementation of object creation, method invocation, and field selection are all quite efficient. This is because the coercions used to implement inheritance and superclass subsumption all reduce to no-ops under a type-erasure dynamic semantics. Although our approach to implementing interfaces appears to be unconventional (casting an object to interface type requires a simple coercion), it allows interface method invocations to have the same cost as ordinary method invocations. From our experience with implementing functional languages, we believe the cost of the coercion can be paid for by the fast method accesses it enables. Since all these operations are implemented in terms of standard type application, record selection, and function application, they are subject to standard optimizations. Furthermore, the interaction of these features with other (non-Java) $F^ω$ code can be well understood.

The remainder of this paper formulates the source and target languages (Sections 2 and 3), illustrates the object layout (Section 4), explains the translation algorithm (Section 5), discusses the possible extensions (Section 6), and then closes with related work (Section 7) and conclusions (Section 8).

## 2 The Source Language

The source language Javacito is a small calculus representing a restricted subset of Java. It contains the essence of Java classes, interfaces, and access control, with several simplifications that permit a concise formal semantics and a comprehensible translation.

The program in Figure 1 contains three classes and one interface that demonstrate some interesting features of Javacito. Class SPt overloads method move, and inherits methods bump and max. The keyword super is used to invoke a statically bound method from the superclass (cf. Figure 1, line 10). Methods are selected based on the dynamic class of the receiver. When the bump method is invoked on an object of class SPt, it is delegated to the bump method in class Pt (line 5), which then invokes the move method in class SPt (line 10).

Classes have private mutable fields and public methods. A class name may appear recursively in the types of its fields and methods (line 3). A method in class Pt can access the private fields of arguments which statically have the same class type. Objects can be passed to methods expecting arguments of a superclass (line 22) or interface type (line 19).

We also support several features not demonstrated by this sample program. An interface may extend several other interfaces and a class may implement several interfaces. A class may contain a field which is an instance of the class itself (discussed further in Section 5.1.1), and new instances of a class may be created inside its own methods and field initializers.

Java features we do not support in Javacito include null refer-
2.1 Static semantics

Figures 3 and 4 contain the syntax of static environments, rules for environment formation, and definitions of environment-summarizing relations. A hierarchy \( H \) is an environment which maps class or interface names to their immediate superclasses and superinterfaces. The hierarchy formation judgment \( \vdash_H \) guarantees that a new type is added to the hierarchy at most once, and that all supertypes are already defined and mentioned only once. The member type environment \( E \) maps types to lists of field and methods names, paired with the respective types. The formation rule for field access. A field

\[ f \in \text{dom}(E) \]

\[ \forall j \in \{1..n\}. \ (f_j = f_j \implies j = j' \) \]

\[ \forall k \in \{1..p\}. \ (T_k = T_k \implies k = k' \) \]

\[ \forall j', \ (Ty \equiv Ty' \text{ and } (m_k, T) \in E Ty') \implies T_k = T' \]

\[ \vdash_{E} E \cup \{ \{f, T\}_{f \in \text{dom}(E)}, \{m, T\}_{m \in \text{dom}(E)} \} \]

Figure 3: Environment formation.

The syntax of Javacito is given in Figure 2. The underlined terms in Figure 2 are annotations required by our operational semantics. In all terms, the prefix \( \text{exp} \) indicates that the term is statically enclosed in the declaration of class \( c \). In the super term, the additional \( \text{val} \) in the annotation holds the value of this. These annotations are required by the contextual operational semantics, as explained below.

\[ \vdash_{H} H \]

\[ \vdash_{E} E \]

\[ \forall j \in \{1..n\}. \ (j = j \implies j = j' \) \]

\[ \forall k \in \{1..p\}. \ (k = k \implies k = k' \) \]

\[ \forall j', \ (Ty \equiv Ty' \text{ and } (m_k, T) \in E Ty') \implies T_k = T' \]

\[ \vdash_{E} E \cup \{ \{f, j, f_j\}_{f \in \text{dom}(E)}, \{m_k, T_k\}_{m_k \in \text{dom}(E)} \} \]

Figure 4: Environment relations.

\[ \vdash_{H} H \]

\[ \vdash_{E} E \]

\[ \forall j \in \{1..n\}. \ (j = j \implies j = j' \) \]

\[ \forall k \in \{1..p\}. \ (k = k \implies k = k' \) \]

\[ \forall j', \ (Ty \equiv Ty' \text{ and } (m_k, T) \in E Ty') \implies T_k = T' \]

\[ \vdash_{E} E \cup \{ \{f, j, f_j\}_{f \in \text{dom}(E)}, \{m_k, T_k\}_{m_k \in \text{dom}(E)} \} \]

Rules for correctness of empty environments are omitted. Our notation for lists and list comprehensions is the usual, but we abuse set membership notation on lists.
a subclass of $c$, however.

**Proposition 1 (Decidability)** $\vdash_\rho$ is decidable.

### 2.2 Operational semantics

Figure 6 contains the contextual small-step operational semantics of Javacito. We introduce two new terms: object expressions $c \langle \{ f \mapsto \text{exp} \} \rangle$ and locations locs. The runtime store $\mathcal{S}$ maps each location to an object value, which consists of a class tag and a collection $\mathcal{F}$ of field-value bindings.

An evaluation context $E$ is an expression with a hole, which encodes the search for the next redex. $\{ E \ldots \}$ indicates that the first expression of a sequence is always the redex. An argument list of the form $(\text{val}_1, \ldots, \text{val}_n, E, \ldots)$ means that all expressions to the left of the redex must be values. The state of the computation is characterized completely by a term paired with a store. The primitive reduction relation $\rightarrow$ is defined between these states, given the class hierarchy $\mathcal{H}$ and the code $R$. We do not need error states; since Javacito does not have null references or dynamic casts, no runtime errors can occur. Rather, evaluation gets stuck if one of the side conditions of a reduction is not met.

An object expression is used to represent partially initialized objects during the reduction of new $c$. Once all the field expressions are reduced to values, we allocate a new location in which to store the object literal.

Note how the annotations are used in defining reductions. In (r-super), we retrieve the method $m$ from $c'$, the immediate superclass of the class $c$ which contains the super call. The loc annotation in (r-super) is the location of this. In the rules (r-get) and (r-set) for field access, the class annotation is used to enforce privacy; $f$ must be declared in the class which contains the field access expression.

Even though an upward cast has no effect at runtime, the side condition of (r-cast) ensures that evaluation will get stuck if a cast is somehow applied to an object of inappropriate type.

Javacito is sound, meaning that a program which passes the static type checking will not get stuck at runtime. Thanks to the strong side conditions on (r-get) and (r-set), this also means that privacy will not be violated at runtime. To formally state and prove soundness we use standard techniques and extend the typing relation to a relation $\mathcal{H}; R \vdash_\rho (\text{exp}, S) : ty$ on runtime configurations (see e.g. [13]), which allows us to prove the subject reduction and progress properties. Details can be found in the accompanying technical report (23).

### 3 The Target Language

The target language MINIFLINT is a call-by-value variant of the omega-order polymorphic lambda calculus (FO) [18, 33]. We extend $P^\rho$ with standard constructs for recursive types of base kind (with explicit fold and unfold), existential types with dot nota-
Syntactic extensions:

Object expression \( \text{exp} ::= \ldots \{ \text{c}, \text{f} \mapsto \text{exp})\} \)

Location value \( \text{val} ::= \ldots \{ \text{loc} \) 

Field map \( \mathcal{F} ::= \emptyset | \mathcal{F} \cup \{ \text{f} \mapsto \text{val} \} \)

Runtime store \( \mathcal{S} ::= \emptyset | \mathcal{S} \cup \{ \text{loc} \mapsto \{ \text{c}, \mathcal{F} \} \) 

Evaluation contexts:

\[
E ::= \bullet \{ E \ldots \} | (\text{ty}) E \\
| (\text{c}, \{ f \mapsto \text{val}, \ldots, f \mapsto \text{val}, f \mapsto \text{val} \} \ldots) \\
| \text{val} m (\ldots) \{ \text{val} m (\text{val} 1, \ldots, \text{val} n, E \ldots) \\
| \text{c} \cdot \text{val} \cdot \text{super} m (\text{val} 1, \ldots, \text{val} n, E \ldots) \\
| \text{eval} E.f \cdot \text{eval} E.f = \text{eval} | \text{eval} \text{val} f = E \\
\]

Primitive reductions:

(r-new) \( \mathcal{H}; \mathcal{R} \vdash \{ \text{new e}, S \} \mapsto \{ \text{e}, \mathcal{F}, S \} \)

where \( \mathcal{F} = \{ f \mapsto \text{exp} \} \) \( f \in \mathcal{R} \) with \( \text{c} \) 

(r-alloc) \( \mathcal{H}; \mathcal{R} \vdash \{ \text{loc}, \text{val} \} \mapsto \{ \text{e}, \mathcal{F}, S \} \) 

where \( \text{loc} \notin \text{dom}(\mathcal{S}) \) 

(r-body) \( \mathcal{H}; \mathcal{R} \vdash \{ \text{exp} ; \text{exp} \ldots \text{exp} \} \mapsto \{ \text{e}, \mathcal{F}, S \} \) 

(r-return) \( \mathcal{H}; \mathcal{R} \vdash \{ \text{val} \} \mapsto \{ \text{e}, \mathcal{F}, S \} \) 

(r-call) \( \mathcal{H}; \mathcal{R} \vdash \{ \text{loc}, m (\text{val} 1, \ldots, \text{val} n) \} \mapsto \{ \text{e}, \mathcal{F}, S \} \)

where \( \mathcal{S}(\text{loc}) = \{ \text{f}, \mathcal{F} \} \) and \( \text{c} \cdot \text{exp} \cdot \text{exp} \) 

(r-super) \( \mathcal{H}; \mathcal{R} \vdash \{ \text{loc}, m (\text{val} 1, \ldots, \text{val} n) \} \mapsto \{ \text{e}, \mathcal{F}, S \} \)

where \( \mathcal{S}(\text{loc}) = \{ \text{f}, \mathcal{F} \} \) and \( \text{c} \cdot \text{exp} \cdot \text{exp} \) 

(r-get) \( \mathcal{H}; \mathcal{R} \vdash \{ \text{loc}, m (\text{val} 1, \ldots, \text{val} n) \} \mapsto \{ \text{e}, \mathcal{F}, S \} \)

where \( \mathcal{S}(\text{loc}) = \{ \text{f}, \mathcal{F} \} \) and \( \text{c} \cdot \text{exp} \cdot \text{exp} \) 

(r-set) \( \mathcal{H}; \mathcal{R} \vdash \{ \text{loc}, m (\text{val} 1, \ldots, \text{val} n) \} \mapsto \{ \text{e}, \mathcal{F}, S \} \)

where \( \mathcal{S}(\text{loc}) = \{ \text{f}, \mathcal{F} \} \) and \( \text{c} \cdot \text{exp} \cdot \text{exp} \) 

(r-cast) \( \mathcal{H}; \mathcal{R} \vdash \{ \text{loc}, m (\text{val} 1, \ldots, \text{val} n) \} \mapsto \{ \text{e}, \mathcal{F}, S \} \)

where \( \mathcal{S}(\text{loc}) = \{ \text{f}, \mathcal{F} \} \) and \( \text{c} \cdot \text{exp} \cdot \text{exp} \) 

Figure 6: Operational semantics of Ja va cito.

Since these are ordered records, the use of labels here is strictly for clarity of presentation. In an implementation \( L \) could be an integer indicating the offset of the row from the beginning of the enclosing record. Note also that record terms are just non-extensible tuples.

An existential type \( \exists \text{ty} \) is a type of a pair containing a hidden type \( \text{ty} \) of kind \( \kappa \) and a term of type \( \alpha \cdot \text{ty} \). We use dot notation \( \alpha \) to access the type \( \alpha \cdot \text{ty} \) and term \( \alpha \cdot \text{val} \) components of a package \( \alpha \). Although this is a form of dependent type, the calculus remains decidable because we restrict the \( \cdot \) operation to term variables \( \alpha \). We provide a \text{let} construct to bind term variables and limit their scope. Abstract types \( \alpha \cdot \text{ty} \) and \( \alpha \cdot \text{val} \) are equivalent if and only if \( \alpha \cdot \text{val} \) and \( \alpha \cdot \text{ty} \) are bound by the same \text{let}.

The argument of an application must be a variable \( x \) that we can track this equivalence through function applications. Variables are prevented from escaping the scope of \( \lambda \)-abstractions \( \lambda \cdot \text{ty} \). \( \alpha \cdot \text{val} \) by disabling free occurrences of \( x \) in the type of \( \alpha \).

Proposition 2 (Decidability) The \( \text{MiniFLINT} \) typing relation \( \alpha \cdot \text{val} : \tau \) is decidable.

We omit the dynamic semantics of \( \text{MiniFLINT} \), which defines
Figure 9: Object layout. (a) and (b) are instances of classes Pt and SPt, respectively. (c) is a view of (b) through the Zm interface.

4 Object Layout

We start by examining the object layout used in our encoding. Figure 9 depicts the runtime representation of two Java class objects and an interface object. The classes and interfaces are taken from the example in Figure 1.

An object is a record containing a virtual table (vtab) and the private fields. The vtable is a per-class data structure containing pointers to the methods defined in the class or inherited from the superclass.

The object marked (a) is an instance of the Pt class, which has one field and three methods. The object marked (b) is an instance of the subclass SPt, which extends Pt with a scale field and a zoom method. SPt overrides the move method; the others are inherited from the superclass by simply pointing to the same code. Note that method selection regardless whether we are using a class or interface object. From our intuition and experience with implementing functional languages, we believe the cost of the coercion will be paid for by the fast method accesses it enables. Another reason to use the coercion in a typed implementation is that the dictionary lookup (and caching the results for quicker access later) is extremely difficult to prove type-safe.

5 Translation

The main result of this paper is a type-preserving translation algorithm that compiles a well-typed Javacito program into a term of the target language MiniFliNT. In this section, we first informally describe our algorithm, using specific examples and a set of type and term macros (see Figures 10 and 11). We then give the formal algorithm (see Figure 12) and show that it is both type-correct and sound.

Each class declaration is separately translated into a let-bound existential package. A skeleton of this package for a class c follows:

$$\begin{align*}
\text{let } x_c & = \{ \text{tnm} : \Omega = \{ 5.7 \}, \\
& \{ \text{dct} = \Lambda \text{vtab} \cdot \Lambda \text{vtab} \cdot \{ 5.3 \}, \\
& \text{init} = \lambda 1. \{ 5.3 \}, \\
& \text{new} = \lambda 1. \{ 5.3 \} \}
\end{align*}$$

Missing components in the package have been replaced with boxes indicating the subsections below which contain further detail.

The package is bound to $x_c$, a variable obtained directly from the class name c. Formally, $x_c$ is a map from ClassNames to Vars. In examples, we distinguish the two domains using different fonts; the package for class Pt is bound to the variable Pt.

The type component ($\text{tnm}$) of the package contains the type of the private fields declared in class c. Outside the package, this type can be selected using dot notation ($x_c.\text{Typ}$) but it remains abstract. The value part of the package is a record containing an extensible dictionary ($\text{dct}$), a field initializer ($\text{init}$), and a constructor ($\text{new}$). We say $\text{dct}$ is extendible because it is parameterized by row types for additional fields and methods. To produce a vtable from an extensible dictionary, we instantiate these row types to empty. Subclasses will instantiate them according to their own new fields and methods.

In Section 5.1, we describe the encoding of private fields and field initialization. Section 5.2 covers the typing issues for vtables and objects. Terms for method invocation, super, new, and casts are given in Section 5.3. Inheritance and interfaces are handled in sections 5.4 and 5.5, respectively. Finally, Section 5.6 makes some formal claims about the translation.

5.1 Fields and privacy

5.1.1 Own fields

Let the fields declared in class c be represented by a record of ML-style polymorphic ref cells called $\text{OwnFlds}[c]$. For example,
OwnFields[Pt] = {x : Ref Int}. Using a more efficient flat mutable record presents no problems, but for simplicity we prefer to preserve the orthogonality of features in MINIFLINT.

A tricky issue in defining OwnFields[c] is that the class c may contain a field of type c, whose privates are also visible to code within class c. Imagine a class A which has fields 'int data' and 'List next'. Within class List, it is legal to access not only 'this.next' but also 'this.next.next' and 'this.next.next.next', and so on, ad infinitum.

This implies that, in the general case, OwnFields[c] must be a recursive type:

OwnFields[c] = μtown. OwnFieldsGen[c] (ObjGen[c] town)

where ObjGen[c] (see Section 5.2.2) is the type of a class object parameterized by the type of its private fields. (All type operators used in the translation are listed in Figure 10.) OwnFieldsGen[c] takes an argument town, which contains a field containing ref cells for each field declared in c, and makes any fields of type c have type town instead. For the class List mentioned above,

OwnFieldsGen[ List ] = μtown. {data : Ref Int; next : Ref town}

Substituting this into OwnFields[ List ], we have:

OwnFields[ List ] = μtown. {data : Ref Int; next : Ref OwnFields[ List ]}

As required, the private fields of List include a next field which, in turn, is an object whose private fields include a next field, etc.

### 5.1.2 All fields

An object contains not only the fields of its own class, but the fields of all its superclasses as well. Field visibility, however, is based on the class in which the fields were declared. Fields from each class can be hidden or revealed independently. This implies that we need to segregate the fields by the class in which they were declared.

Let AllFieldsGen[c] produce the type of the segregated record containing all fields in an object of class c. In the example,

AllFieldsGen[Pt] = λtown. {Pt : town}

AllFieldsGen[SPt] = λtown. {Pt : Pt.Typ; SPt : town}

where Pt.Typ is the abstract type of private fields in Pt. town can be instantiated to OwnFields[c] for use inside the class c, or to x.Typ outside, where x is bound to the package implementing class c.

We use row polymorphism to extend superclass field record types in subclasses. We additionally parameterize the above operators by subfields, a row of fields defined by potential subclasses. Now we can define the SPt fields using the Pt fields directly:

AllFieldsGen[Pt] = λtown. λsubfields. {Pt : town; subfields}

AllFieldsGen[SPt] = λtown. λsubfields. AllFieldsGen[Pt] (SPt : town; subfields)

In the general case, the segregated record type is produced using two mutually recursive operators (see Figure 10).

### 5.1.3 Field initialization

As a part of the code for class c we define a function init which creates a record of ref cells, each initialized with the translated field initializer expression. Every class x creates init as a function of type 1 → x.Typ. For the Pt class in our example, Pt.valinit is

UnpackObj[c] (town, o : ObjGen[c] town, •) =

let x = unfold o in let x' = x.val in •


fold (tsubfids = tsubfids)

{tsubVtab = tsubVtab*

self : SelfGen[c] town tsubfids tsubVtab

: ?tsubVtab. SelfGen[c] town tsubfids tsubVtab}

as ObjGen[c] town

UnpackView[i] (io : ExtView[i], •) = let x = unfold io in •

PackView[i] (tobj, o : tobj, it : ExtTab[i] tobj) =

fold (tobj = tobj)

{tobj = o, ttab = it}

: ?tobj. ExtTab[i] (x, Typ) ExtView[i] tobj

as ExtView[i]

GetFields[c] (o : ObjGen[c] OwnFields[c]) =

UnpackObj[c] (OwnFields[c], o, unfold (unfold (x'.val), fieldsdc)

Call[i] (town, o : ObjGen[c] town, m) =

UnpackObj[c] (town, o, unfold (x'.val).vtabdlm (x'.val))

Call[i] (io : ExtView[i], m) =

UnpackView[i] (io, x.val.ilstabdlm (x.val, cobj))

Cast[c, c'] (town, o : ObjGen[c] town) =

PackObj[c] (town, o, PackView[i] (SelfGen[c] town x.Typ, x'.val, unfold (x'.val).vtabdlm))

if c' = Super[c]

Cast[c, i] (town, o : ObjGen[c] town) =

UnpackObj[c] (town, o, PackView[i] (SelfGen[c] town x.Typ, x'.val, unfold (x'.val).vtabdlm))

Cast[i, i'] (io : ExtView[i]) =

UnpackView[i] (io, PackView[i] (x.Typ, x.val, cobj, x.val.ilstabdlm))

Prolog[c] (tsubVtab, tsubVtab, x.λ) • =

λself : SelfGen[c]. OwnFields[c] tsubVtab tsubVtab (λx.λ)

let this = PackObj[c] (OwnFields[c], tsubVtab, tsubVtab, self)

in •

CDcll[c] (classBody : ClassGen[c] OwnFields[c], •) =

let x = (town = OwnFields[c], classBody : ClassGen[c] town) in •

IDcll[i] (•) =

let x = (tstamp = Abs Mlabels[i],

λtobj. λit : ItabGen[i] Abs Mlabels[i] tobj, it

: ∀tobj. ItabGen[i] Abs Mlabels[i] tobj

→ ItabGen[i] tstamp tobj)

in •

Figure 11: Term macros used in the translation.
By convention, the type variables used in the translation have the following kinds and intended meaning:

- \( t_{\text{self}} : \Omega \) — type of receiver object itself
- \( t_{\text{twin}} : \Omega \) — type of an argument/result of receiver’s class
- \( t_{\text{town}} : \Omega \) — type of private fields of object
- \( t_{\text{stubfids}} : \mathcal{R}^{\mathcal{C}\text{Labels}[c]} \) — type of row of field records of a subclass of \( c \)
- \( t_{\text{subvtab}} : \Omega \) — type of row of new methods/interfaces of a subclass of \( c \), parameterized by type of self
- \( t_{\text{obj}} : \Omega \) — type of object in an interface view
- \( t_{\text{stamp}} : \mathcal{R}^{\mathcal{M}\text{Labels}[i]} \) — type of interface’s “hidden methods”

Given environments \( \mathcal{H} \) and \( \mathcal{E} \):

- \( \text{Super}[c] = e' \) s.t. \( c \rightarrow_c e' \)
- \( \text{SCLabels}[c] = \{ c | c \leq^e \mathcal{H} e' \} \)
- \( \text{MLabel}[c] = \{ l_m | (m, \mathcal{H} e \leq c) \cup \{ l_i | c \rightarrow^e \mathcal{H} i \} \)
- \( \text{MLabel}[i] = \{ l_m | (m, \mathcal{H} e \leq i) \cup \{ l_i | i \leq^e \mathcal{H} i' \} \}

- \( \text{NewPublic}[c] = \lambda_{t\text{twin}}. \lambda_{\text{stubvtab}}. \lambda_{t\text{self}}. (l_l : \text{ExtItab}[i] t\text{self})^t (l_l : t\text{twin} \rightarrow [T] t\text{twin})^t (t\text{stubvtab} t\text{self}) \)
  where \( i \leftarrow \pi_2(\mathcal{H}(c)) \)

- \( \text{ObjTmp}[c] = \lambda_{t\text{own}}. \lambda_{t\text{twin}}. \lambda_{\text{stubfids}}. \lambda_{\text{stubvtab}}. \lambda_{t\text{self}}. \text{vtab} : \text{vtabGen}[c] t\text{twin} t\text{stubvtab} t\text{self} ; \text{fields} : \text{AllFidsGen}[c] t\text{own} t\text{stubfids} \)

- \( \text{ObjGen}[c] = \lambda_{t\text{own}}. \mu_{t\text{twin}}. \lambda_{\text{stubfids}}. \exists_{t\text{stubvtab}}. \text{ObjTmp}[c] t\text{own} t\text{twin} t\text{stubfids} t\text{stubvtab} \)

- \( \text{ExtObj}[c] = \text{ObjGen}[c] (x_c. \text{Typ}) \)

- \( \text{SelGen}[c] = \lambda_{t\text{own}}. \lambda_{\text{stubfids}}. \lambda_{\text{stubvtab}}. \text{ObjTmp}[c] t\text{own} (\text{ObjGen}[c] t\text{own}) t\text{stubfids} t\text{stubvtab} \)

- \( \text{OwnFidsGen}[c] = \lambda_{t\text{twin}}. \{(l_f : \text{Ref} ([T] t\text{twin})^t \}
  where (f, T) \leftarrow \pi_1(\mathcal{E}(c)) \)

- \( \text{OwnFids}[c] = \mu_{t\text{own}}. \text{OwnFidsGen}[c] (\text{ObjGen}[c] t\text{own}) \)

- \( \text{AllFidsGen}[c] = \lambda_{t\text{own}}. \lambda_{\text{stubfids}}. \text{AllFids}[\text{Super}[c]] (k : t\text{own}; t\text{stubfids}) \)

- \( \text{AllFids}[c] = \lambda_{t\text{stubfids}}. \text{AllFidsGen}[c] (x_c. \text{Typ}) t\text{stubfids} \)

- \( \text{VtabGen}[c] = \lambda_{t\text{twin}}. \lambda_{\text{stubvtab}}. \lambda_{t\text{self}}. \text{ExtVtab}[\text{Super}[c]] (\text{NewPublic}[c] t\text{twin} t\text{stubvtab} t\text{self}) \)

- \( \text{ExtVtab}[c] = \lambda_{\text{stubvtab}}. \lambda_{t\text{self}}. \text{VtabGen}[c] \text{ExtObj}[c] t\text{stubvtab} t\text{self} \)

- \( \text{ExtVtab} \text{Object} = \lambda_{\text{stubvtab}}. \lambda_{t\text{self}}. \{ \text{tstubvtab} t\text{self} \}

- \( \text{DictGen}[c] = \lambda_{t\text{own}}. \lambda_{\text{stubfids}}. \lambda_{\text{stubvtab}}. \text{VtabGen}[c] (\text{ObjGen}[c] t\text{own}) (\lambda_{t\text{self}}. \text{Abs}[\text{MLabel}[c]]) (\text{SelGen}[c] t\text{own} t\text{stubfids} t\text{stubvtab}) \)

- \( \text{ClassGen}[c] = \lambda_{t\text{own}}. \{ \text{dict} : \forall_{t\text{stubfids}}. \forall_{t\text{stubvtab}}. \text{DictGen}[c] t\text{own} t\text{stubfids} t\text{stubvtab}; \}

- \( \text{init} : 1 \rightarrow \text{OwnFidsGen}[c] (\text{ObjGen}[c] t\text{own}); \)

- \( \text{new} : 1 \rightarrow \text{ObjGen}[c] t\text{own} \}

- \( \text{ItabTmp}[i] = \lambda_{t\text{stamp}}. \lambda_{t\text{twin}}. \lambda_{t\text{cobj}}. \{ (l_j : \text{ExtItab}[i'] t\text{cobj})^t (l_j : t\text{cobj} \rightarrow [T] t\text{twin})^t t\text{stamp} \}
  where i' \leftarrow H(i), \{ m, T \} \leftarrow \pi_2(\mathcal{E}(i)) \)

- \( \text{ViewTmp}[i] = \lambda_{t\text{stamp}}. \lambda_{t\text{twin}}. \lambda_{t\text{cobj}}. \{ \text{cobj} : t\text{obj} : \text{Itab}[i] \} t\text{stamp} t\text{twin} t\text{cobj} \)

- \( \text{ViewGen}[i] = \lambda_{t\text{stamp}}. \mu_{t\text{twin}}. \lambda_{t\text{cobj}}. \{ \text{cobj} : \text{ViewGen}[i] t\text{stamp} t\text{twin} t\text{cobj} \)

- \( \text{Itab}[i] = \lambda_{t\text{stamp}}. \lambda_{t\text{twin}}. \lambda_{t\text{cobj}}. (l_j : \text{ExtItab}[i'] t\text{cobj})^t (l_j : t\text{cobj} \rightarrow [T] t\text{twin})^t t\text{stamp} \)

- \( \text{ExtView}[i] = \text{ViewGen}[i] (x_i. \text{Typ}) \)

- \( \text{ExtItab}[i] = \lambda_{t\text{cobj}}. \text{ItabGen}[i] (x_i. \text{Typ}) t\text{cobj} \)

- \( \{ y_1 \ldots y_n \rightarrow ty \} t\text{y}' = \lambda_{t\text{twin} : \Omega}. \{ y_1 t\text{y}' t\text{twin} \rightarrow \ldots y_n t\text{y}' t\text{twin} \rightarrow [B] t\text{y} t\text{twin} \}
  \begin{cases} 
    t\text{twin}, & \text{if } ty = ty' \\
    [b] t\text{y}' = \lambda_{t\text{twin} : \Omega}. \begin{cases} \text{ExtObj}[c] & \text{otherwise, if } ty = c \\
    \text{ExtView}[i] & \text{otherwise, if } ty = i \end{cases} 
  \end{cases} \)

Figure 10: Type operators used in the translation.
\[(\text{trans-var})\quad x \in \text{dom}(\Gamma)\quad \vdash \Gamma; \vdash x : \text{Type} \quad \leadsto \quad x\]

\[(\text{trans-new})\quad H; \vdash_1 c'\]
\[H; \vdash c \cdot \text{new } c' : c' \quad \leadsto \quad \{ (x_\chi \Omega) \cdot \text{new } \Omega \}, \text{if } c' = c \]
\[\text{otherwise } \quad \{ x_\chi \cdot \text{var} \cdot \text{new } \Omega \}, \text{if } c' \neq c\]

\[(\text{trans-get})\quad H; \vdash \Gamma; \vdash c \cdot \text{exp \_ f_ t_ y} : t'y \leadsto a \quad \leadsto \quad 1((\text{GetFields}[c](e))d_f )\]

\[(\text{trans-set})\quad H; \vdash \Gamma; \vdash c \cdot \text{exp_1 : t_1} \leadsto a \quad t'y' \leq c \quad \leadsto \quad (f, ty) \in c\]
\[H; \vdash \Gamma; \vdash c \cdot \text{exp_2 : t_2} \leadsto c \quad \leadsto \quad \{ (t', ty) \in c\}\]
\[H; \vdash \Gamma; \vdash c \cdot \text{exp_1 : t_1} \leadsto a \quad t'y' \leq c \quad \leadsto \quad \{ (t', ty) \in c\}\]
\[H; \vdash \Gamma; \vdash c \cdot \text{exp_2 : t_2} \leadsto c \quad \leadsto \quad \{ (t', ty) \in c\}\]
\[H; \vdash \Gamma; \vdash c \cdot \text{exp_1 : t_1} \leadsto a \quad t'y' \leq c \quad \leadsto \quad \{ (t', ty) \in c\}\]
\[H; \vdash \Gamma; \vdash c \cdot \text{exp_2 : t_2} \leadsto c \quad \leadsto \quad \{ (t', ty) \in c\}\]

\[(\text{trans-body})\quad \forall j \in \{1 \ldots n\}, H; \vdash \Gamma; \vdash c \cdot \text{exp_j : t_j} \leadsto a_j\]
\[\text{where } \forall a_j \quad (\text{trans-class})\quad H; \vdash \Gamma; \vdash c \cdot \text{exp} : t'y \leadsto a \quad t'y \leq c \quad \leadsto \quad H''; \vdash \text{Cast}[t'y', \text{ty}] (\text{Own}[c](t'y), \text{Own}[c](\text{ty}), e)\]
\[\text{where } \text{Own}[c](t'y) = \begin{cases} \text{OwnFds}[c], & \text{if } t'y = c \\ \text{Ty}_a & \text{otherwise} \end{cases}\]

\[\text{let init } = \lambda : 1. \{ \text{scale } = \text{ref } 1 \}\]
\[\text{in } \ldots \{ \text{Pt } = \text{Pt} \cdot \text{val} \cdot \text{init } \}, \text{SPt } = \text{init } \} \ldots\]

Then the type of the record will be \text{AllFdsGen}[\text{SPt}] applied to \text{OwnFds}[\text{SPt}] and the empty row \text{Abs'}c, where the label set \text{L} is \{\text{Pt}, \text{SPT}\}.

5.2 Types used in the translation

5.2.1 Virtual tables

Encoding the type of the vtable of a class is a bit trickier than encoding the fields. We need types for the self argument and for method arguments of the same class. The latter is not simply the same type as external class objects because the private fields need to be accessible.

For now, we will abstract over all of these types, and show how to resolve them in the next section. \text{VtabGen}[c] produces the type of a vtable for class c. Here is the parameterized type for the vtable of Pt:

\[\text{VtabGen}[\text{Pt}] = \lambda t_{\text{tw}} : \lambda t_{\text{sub}} : \lambda t_{\text{self}} : \lambda \text{max : } t_{\text{tw}} \rightarrow t_{\text{tw}} \rightarrow t_{\text{tw}} ; \lambda \text{move : } t_{\text{self}} \rightarrow \text{int} ; \lambda \text{bump : } t_{\text{self}} \rightarrow 1 ; \lambda t_{\text{sub}} : t_{\text{sub}} \rightarrow \text{ref}\]

where \text{twin} is the type of non-self arguments or results of class Pt. \text{tsubvtab} is the type of a row of new methods in some subclass of Pt. parameterized by \text{tself}, the type of the receiver object itself.

The tail of the vtable is produced by applying the \text{tsubvtab} operator to \text{tself}. The reason is that new methods added by subclasses will need to use exactly the same self type. To achieve this, we parameterize the row type. The parameterized type of the SPT vtable can now be defined using that of Pt (we are ignoring the Zm inter-
We combine the field and vtable operators in private fields in Figure 10. Note the fixed point for resolving parameters and which variations are needed, we must briefly examine (see Figure 10).

In the vtable for class Pt, it is clear that the argument of max is special because methods in the vtable can access its private fields. In the vtable for class SPT, however, the type of the argument to max becomes indistinguishable from that of any other Pt object seen from outside, regardless whether we inherit or override max.

As indicated by the types of functions in the vtable, we are using curried function application to implement methods with arguments. This is only to keep the target calculus as simple as possible. Since we never do partial function application, switching to multi-argument functions would present no problems.

5.2.2 Object types

We combine the field and vtable operators in ObjTmp[c], a parameterized template for generating class object types (see Figure 10). Note the fixed point for resolving tsub[s]; it ensures that a method selected from the vtab of an object o will accept o as its self argument.

This template may be used to generate several variations of class object types. In order to understand how we resolve the parameters and which variations are needed, we must briefly examine the extensible dictionary for a class:

\[
dict = \lambda \text{subFids}_o, \lambda \text{tsubVtab}_o.
\begin{align*}
\text{max} & : \text{tself}_o \rightarrow \text{ExtObj}[\text{Pt}] \rightarrow \text{ExtObj}[\text{Pt}]; \\
\text{move} & : \text{tself}_o \rightarrow \text{int} \rightarrow 1; \\
\text{bump} & : \text{tself}_o \rightarrow 1; \\
\text{zoom} & : \text{tself}_o \rightarrow \text{int} \rightarrow 1; \\
\text{tsubVtab}_o & \text{ tself}_o
\end{align*}
\]

where ExtObj[c] is the type of an external view of an object of class c, with all private fields hidden. It is instructive to expand the body of this operator, making it independent of the superclass vtable type operator:

\[
\text{VtabGen}[\text{SP}] = \lambda \text{twin}, \lambda \text{tsubVtab}, \lambda \text{tself}.
\begin{align*}
\text{VtabGen}[\text{Pt}]: \text{ExtObj}[\text{Pt}]; \\
(\lambda \text{tself}_o (\text{zoom} : \text{tself}_o \rightarrow \text{int} \rightarrow 1; \\
\text{tsubVtab}_o \text{ tself}_o))) \text{ tself}_o
\end{align*}
\]

In the vtable for class Pt, it is clear that the argument of max is special because methods in the vtable can access its private fields. In the vtable for class SPT, however, the type of the argument to max becomes indistinguishable from that of any other Pt object seen from outside, regardless whether we inherit or override max.

As indicated by the types of functions in the vtable, we are using curried function application to implement methods with arguments. This is only to keep the target calculus as simple as possible. Since we never do partial function application, switching to multi-argument functions would present no problems.

5.2.3 Name equivalence

By using abstract types in the fields record of every object type, we encode the class hierarchy and preserve Java’s name equivalence. Recall that two abstract types \( x \cdot \text{Typ} \) and \( x' \cdot \text{Typ} \) are equivalent if and only if \( x \) and \( x' \) are bound by the same let. Thus, structurally equivalent classes will translate to different, incompatible object types.

At first, it might seem that a target-calculus programmer could sabotage name equivalence. Anyone is permitted to use \( x \cdot \text{val} \cdot \text{init} \) to create records containing values having the abstract types of \( c \) and its superclasses. Indeed, a programmer can create a fields record that looks like it belongs to class \( c \), package it with his own devious methods, and have it masquerade as an object of class \( c \). But this is not as subversive as it seems; it is merely subclassing! A JavaObject programmer could have achieved the same feat.

5.3 Implementing the dictionary

So far, we have concentrated mostly on typing issues. In this and the next section, we explore the term-level translation of method bodies and expressions. We start with object creation, then move on to method bodies, method invocation, and casts.

5.3.1 Object creation

For the class Pt the extensible dictionary has this shape:

\[
dict = \lambda \text{tsubFids}_o, \lambda \text{tsubVtab}_o.
\begin{align*}
\{\text{max} & : \lambda \text{self}_o, \ldots, \\
\text{move} & : \lambda \text{self}_o, \\
\text{bump} & : \lambda \text{self}_o \\
\text{zoom} & : \lambda \text{self}_o
\end{align*}
\]

The vtab component of an object is a dictionary fixed to the object’s type; we obtain it from dict by instantiating to empty rows.

\[
\text{let } p_0 = \text{fold } \langle \text{vtab} = \text{dict } [\text{Abs}^{<c}] [\text{Abs}^{<m}], \\
\text{fields} = \langle \text{Pt } = \text{init} \{\} \rangle \rangle \\
as \text{SelfGen}[\text{Pt}] \text{ OwnFids}[\text{Pt}] \text{ Abs}^{<c} \text{ Abs}^{<m}
\]

where \( L_c \) and \( L_m \) are \( \{\text{max}, \text{move}, \text{bump}\} \).

This object can be passed as the self parameter to methods in the vtab, but its dynamic type is exposed—anyone can see that fields has only one component, \( Pt \). In order to make \( p_0 \)’s type indistinguishable from the type of objects created by subclasses of \( Pt \), we need to hide the tails:

\[
\text{let } p_1 = \text{fold } \langle \text{tsubFids} = \text{Abs}^{<c}, \\
\text{tsubVtab} = \text{Abs}^{<m}, \\
p_0 : \text{SelfGen}[\text{Pt}] \text{ town } \text{Abs}^{<c} \text{ tsubVtab} \\
: \text{SelfGen}[\text{Pt}] \text{ town } \text{tsubFids} \text{ tsubVtab} \\
as \text{ObjGen}[\text{Pt}] \text{ town}
\]

where, as usual, \( \text{town} \) is \( \text{OwnFids}[\text{Pt}] \). If this expression is inside the \( Pt \) class package, or \( Pt \cdot \text{Typ} \) otherwise. The final \text{fold} resolves the twin type, as in the non-self argument of \( \text{max} \). This argument has the same private fields accessible as does self, but its additional fields and methods are hidden since it might have a different dynamic type.

This operation, in its general form, is encapsulated as the term macro \text{PackObj}[c] in Figure 11. It is used not only for object creation but for various forms of casting. Its inverse is \( \text{UnpackObj}[c] \).

To implement the \text{new} function in the class package, we need to wrap a \( \lambda \) around the two let bindings above:

\[
\text{let } \text{new} = \lambda \cdot \text{p1. let } p_0 = \ldots \text{ in let } p_1 = \ldots \text{ in p1}
\]
Now we can simply select and apply this function whenever we encounter `new Pt`, unless it appears in the body of a method in class Pt itself. In this case, there is a circularity. The `new` function refers to `dict`, and a component of `dict` refers to `new`. (Similarly, `new Pt` might appear in a field initializer expression, in which case there is a circularity between `init` and `new`.)

We resolve this by passing a `frozen` class record to the call-by-value fixed point combinator \( \mathcal{Y} \).

\[
\text{let } x_c = (\{ \text{new} \} \ominus \text{OwnFlds}[c]), \\
\text{Y}_c(\text{ClassGen}[c], \text{OwnFlds}[c]) \\
(\lambda x. : \text{Int} \rightarrow \text{ClassGen}[c], \text{OwnFlds}[c], \\
\lambda : \text{Int} \rightarrow \{ \text{dict} = \ldots, \text{int} = \ldots, \text{new} = \ldots \}) \\
\in \ldots
\]

where `ClassGen[c]` is the type of the class record for class `c` (see Figure 10).

Using this technique, a method inside `dict` can create a new object of class `c` with the expression \((x_c, \{\})\). Note that the fixed point is `inside` the package, so that the private fields of the new object are exposed. If we applied the fixed point to the whole package, then `(new Pt).x` would be illegal.

### 5.3.2 Method bodies

As demonstrated in Section 5.2.2, methods are translated into curried functions with an explicit `self` parameter. `self` is different from other objects (including twin parameters such as `other`) because it has already been unpacked; it is just a recursive record. To access a field of `other`, one needs to unfold, unpack, unpack again, unfold again, and finally, select the fields record. To access a field of `self`, only the last two steps are necessary.

For convenience, we translate object operations (e.g., method invocation, casts, field selection) uniformly, regardless whether the object is `self`. To support this, we begin each method by packing `self` to make it the same shape as any other object:

\[
\text{let this } = \text{PackObj}[c] \text{ OwnFlds}[c], t_{subFIds}, t_{subVtab}, \text{self} \in \ldots
\]

Now selecting a field from `this` is the same as selecting from `other`. An optimization phase can easily remove any consecutive `pack/unpack` or `fold/unfold` expressions.

### 5.3.3 Method invocation (invokevirtual)

To invoke method `m` on an object `o`, we just unfold, unpack, unpack again, unfold again (all type manipulations), then select and apply:

\[
\text{let } x = \text{unfold } o \text{ in let } x' = x.\text{val} \\
\text{in unfold } (x'.\text{val}), t_{vtab}, l_m (x'.\text{val})
\]

The result can then be applied to any additional arguments.

Since method invocation is not atomic, we must prevent users from selecting a method from one object and applying it to some other object. This error is prevented because the tails of the fields and `vtab` records of `o` are abstract types. In the code above, `x'.\text{val}, t_{vtab}, l_m` will have a type which includes the abstract types `x.\text{Typ}` for the additional fields in `o` and `x'.\text{Typ}` for the additional methods in `o`. Since the only object which can have these types is `x'.\text{val}` itself, `l_m` cannot be applied to any other term.

### 5.3.4 Upward cast

The purpose of an upward cast is to allow an object of class `c` to masquerade as an object of its superclass, `c'`. Since we already keep objects inside nested packages to hide the tails of the fields and `vtab` records, we can simply repackage an object, hiding the components of fields and `vtab` that are part of `c` but not `c'`.

In the following example, let `o` have type `EstObj[Pt]`:

\[
\text{let } x = \text{unfold } o \\
\text{in let } x' = x.\text{val} \\
\text{in PackObj}[Pt] (Pt, Typ, (Sp, Typ, x, Typ), \\
(\lambda t_{self} (\text{zoom } t_{self} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow 1; \\
\text{x'.Typ } t_{self})), \\
x'.\text{val})
\]

The result is a value with type `EstObj[Pt]`. We achieved this using type manipulations only; the underlying object record was not touched. Dynamically, the entire process is a no-op. See the `Cast[c, c']` macro in Figure 11 for the generic form.

### 5.4 Inheritance and super

Since each class package exports an extensible dictionary, implementing inheritance is straightforward. A class `c` can select the extensible dictionary of its superclass, and instantiate the tails to its own new fields and methods. For example, inside the `SPr dict`, we bind `super` as follows:

\[
\text{let } \text{super } = \text{Pt, val, dict } [\text{SPr: OwnFlds}[SPr]; t_{subFIds}] \\
(\lambda t_{self} (\text{zoom } t_{self} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow 1; \\
t_{subVtab}, \text{self})) \\
\in \{ \text{max = super.max, } \\
\text{move = \lambda self, super.move self, } \\
\text{bump = super.bump, } \\
\text{zoom = \lambda self, } \}
\]

Inheritance is implemented simply by selecting methods out of the `super` dictionary (as in `max` and `bump`, for example) and placing them in the `dict` of the new subclass. Source language invocations through `super` are implemented by selecting a method from the `super` dictionary (as in `move`). After the one-time type application to produce `super`, no coercions are necessary.

### 5.5 Interfaces

Recall our approach to implementing interfaces (see Section 4). An `interface` is an arrangement of some of the methods in a `vtable`, made to correspond to an interface which the class implements. Given an object of `interface` type, we know nothing about the shape of its `vtable`. Thus, we use the `itable` to give a consistent view of the methods in the interface, independent of the underlying `vtable`.

#### 5.5.1 Interface types

The `itable` is a record of functions, just like the `vtable`. For the `Zm` interface in our example,

\[
\text{ItabGen}_n [Zm] = \lambda t_{\text{twin}}, \lambda t_{\text{objc}}. \text{zoom } t_{\text{objc}} \rightarrow \text{Int} \rightarrow 1
\]

where `t_{objc}` is the type of the underlying class object (to simplify the presentation, we have omitted the `t_{temp}` parameter from `ItabGen`; see section 5.5.4 and figure 10). As before, `t_{twin}` is the type for `non-self` arguments of interface `Zm` (there are no such arguments in this particular example).

An `interface` object (or `view`) is a pair containing some class object (`objc`) and the appropriate `itable`:

\[
\text{ViewTmp}_n [Zm] = \lambda t_{\text{twin}}, \lambda t_{\text{objc}}, \{ \text{objc } t_{\text{objc}}, \text{itable }, \text{ItabGen}_n [Zm] t_{\text{twin}}, t_{\text{objc}} \}
\]
We resolve the $t_{\text{cobj}}$ parameter using an existential type. We should be able to select a method from the itable and pass $\text{cobj}$ as its self parameter. We do not know (or it should not matter) what $t_{\text{cobj}}$ actually is:

$$View_t[Zm] = \mu t_{\text{cobj}}. \exists t_{\text{obj}}. View\text{Tmp}_k[Zm] t_{\text{cobj}} t_{\text{obj}}$$

Finally, we resolve $t_{\text{cobj}}$ using a recursive type. If the interface $Zm$ had any methods with arguments of type $Zm$, they would need to look the same, though they might have a different hidden $t_{\text{cobj}}$ type.

### 5.5.2 Invokeinterface

To invoke method $m$ on an interface object $io$, we bind $x$ to unfold $io$, apply $x.\text{val}.\text{tab}$.map to $x.\text{val}.\text{cobj}$, and apply the result to any additional arguments. Note that this is essentially the same select-and-apply operation that is used for ordinary method invocation (section 5.3.3).

Here again, the abstract type prevents users from doing anything devious with either $\text{itab}$ or $\text{cobj}$. The function $x.\text{val}.\text{itab}.map$ wants an argument of type $x.\text{Typ}$. The only possible expression of that type is $x.\text{val}.\text{cobj}$.

### 5.5.3 Interface casts

We have not yet mentioned how to create an interface object from a class object. As discussed in section 4 (and illustrated in Figure 9), the itable can actually contain the itable for all the interfaces a class implements. Then, casting to an interface type is just a matter of selecting the appropriate itable and pairing it with the object itself. For example, the SPl dict would contain the itable for $Zm$:

```plaintext
dict = \lambda t_{\text{subobj}}. \lambda t_{\text{subobj}}. t_{\text{cobj}}
let super = Pt.\text{val}.\text{dict}[\ldots][\ldots]
let newdict =
{move = \lambda \text{self}. \ldots \text{super}.move \text{self} \ldots ,
 zoom = \lambda \text{self}. \ldots }

in {max = \text{super}.max,
move = \text{newdict}.move,
bp = \text{super}.bpmp,
Zm = \{zoom = \text{newdict}.zoom\},
zoom = \text{newdict}.zoom}
```

Once this itable is present, the following code can be used to coerce an SPl object $o$ to a Zm interface object:

```plaintext
let $x = \text{unfold} \; o$

in let $x' = x.\text{val}$

in fold ($t_{\text{cobj}} : \Omega$ = SelfGen[$\text{SpT}$] SPl.Typ $x.\text{Typ}$ $x'.\text{Typ}$,
{cobj = $x'.\text{val}$,
\text{itab} = unfold ($x'.\text{val}$).\text{tab}.Zm})
: View\text{Tmp}_k[Zm] View[Zm] $t_{\text{cobj}}$

as View[Zm]
```

For this example, we have assumed that this code is contained in some class other than SPl. If we were inside SPl, then SPl.Typ above would be replaced with OwnFlds[SPl]. See the Cast[c, i] macro in Figure 11 for the generic form.

In section 5.3.4, we noted that upward casts consisted of type manipulations only. In contrast, casts to and between interfaces require a simple coercion at runtime. This is not surprising; types can have multiple superinterfaces. A single fixed-offset viable record cannot, in general, meet the needs of all possible superinterfaces. Either simple coercions or some form of dynamic name lookup are required.

### 5.5.4 Name equivalence for interfaces

Like class types, interface types in Java use name equivalence. In the formal treatment, we translate interface declarations into a let-bound package for the purpose of distinguishing different structurally equivalent interfaces. Each interface package has a hidden stamp type $x.\text{Typ}$; the type of an itable for that interface will contain $x.\text{Typ}$ as a component. The body of the package $x.\text{val}$ is the identity function, but the argument type accepts structurally equivalent dictionaries and the return type contains the stamp type.

This technique serves not only to distinguish between different, structurally equivalent interface types, but it also, in a sense, prevents forgery. We cannot construct an object that will satisfy interface $i$ unless the interface package $x_i$ is itself accessible.

### 5.6 Formalization

The formal algorithm translating a (well-typed) JavaCito program $p$ to a MINIFLINT term is presented in Figure 12, using meta-language macros defined in Figure 11. Some of the macros are parameterized contexts; free variables of a term placed in the hole may be captured in the context. The sequents representing the translation are based on those defining the static semantics of JavaCito (Figure 5). In the translation of a class declaration we have omitted the straightforward but tedious definition of the macro $\text{MkDict}[c]$, constructing a class dictionary out of a superclass dictionary $\text{super}$, specialized to its subclass $c$, and a record $\text{newdict}$ of the newly defined methods of $c$. The function of $\text{MkDict}[c]$ is to collect selected components of $\text{super}$ (inherited) or $\text{newdict}$ (overridden or new) into interface dictionaries and a class dictionary.

The translation is total on type-correct JavaCito programs, and it maps them to type-correct MINIFLINT terms.

**Proposition 3 (Type preservation)** If $\emptyset; \emptyset; \emptyset \vdash p : t v \Rightarrow h; r$, then $\emptyset; \emptyset \vdash p \bowtie h; r$; $e$ and $\emptyset \vdash e : [h]_c \Rightarrow \Omega$, where $c \notin \text{dom}(H)$.

The semantic correctness of the translation is shown by establishing a mapping from JavaCito runtime configurations to MINIFLINT runtime configurations which maps value-configurations to value-configurations, and proving that a JavaCito reduction step corresponds with respect to this mapping to finitely many MINIFLINT reduction steps [23]; we omit the proofs due to space constraints.

### 6 Extensions

Many features of Java can be accommodated directly in our framework, but were left out in order to simplify the formal presentation.

#### 6.1 Unordered records

We have used ordered records in our translation of JavaCito, but we can add unordered records [38] into the target language as well:

- **Types**
  - $\tau ::= \ldots \{x.\tau\}$

- **Terms**
  - $e ::= \ldots \{x.(\equiv e)\}$

Here, $\{x.\tau\}$ denotes the unordered record type where $\tau$ must be of kind $t^\text{ord}$, and $\{x.(\equiv e)\}$ is the unordered record term which (unlike ordered records) requires runtime dictionary construction. Selecting from an unordered record uses the same syntax as selecting from an ordered record but it requires runtime dictionary lookup.

Some Java compilers use the same representation for objects of interface types as for those of class types, so casting to interface...
type requires no runtime operations. Unordered records can be used to support this representation by collecting all the itable entries of a viable into a separate unordered record, itself an element of the still ordered viable. Casting an object into interface type only requires repackaging it (a runtime no-op) to hide those entries not exported by the current interface.

6.2 Access control

Jaavacito only allows private fields and public methods, but other access-scoping schemes can also be supported. Private methods can simply be let-bound within the class dictionary since they can neither be called from subclasses nor overridden. Public fields could be placed directly in an object’s fields record, without needing to be segregated.

Protected and package scopes require adding a notion of Java-style package into MINIFLINT. Otherwise, they can be supported using the technique proposed by Moby [16]. The dict and new fields in our class encoding roughly correspond to the class view and the object view in Moby. 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.

6.3 Dynamic casts and exception

Dynamic casts and runtime type identification can be addressed using extensible variant types similar to the exn data type in Standard ML. We add a new extensible variant declaration inside each class c or interface i (for simplicity we borrow the syntax of exception declaration in ML):

\[
\text{exception} \quad \text{Tag}_c \quad \text{of} \quad \text{ObjGen}[c] \quad \text{fown}; \quad \text{or} \quad \text{exception} \quad \text{Tag}_i \quad \text{of} \quad \text{ViewGen}[i] \quad t\text{stamp};
\]

Suppose our extensible variant type is named Tagged; then, inside each class declaration of c, we add a method named allviews that takes the self object and returns a list of Tagged values. More specifically, the allviews method takes the self object s, and for each ancestor class c' (and interface i') it upward-casts s into an object of c' (or i'), packages it with the Tagq_c' (or Tagq_i') flag, and then returns the entire list of resulting Tagged values. We also add the specification for the allviews method into each interface so that it can be invoked on objects of interface type as well.

To test if an object o is an instance of class c (or interface i), we invoke the allviews method on o and then check through the result Tagged list to see if any of them is tagged with Tagq_c (or Tagq_i). The checkcast operation can be implemented in the same way since the object associated with Tagq_c (or Tagq_i) is already of type c (or i). Notice unlike Grew’s recent work [19], our technique does not use hierarchical extensible sums or variance-based subtyping so it requires a much simpler target language. With careful coding, the allviews method can be implemented as efficiently as the untyped code used in typical Java compilers.

6.4 Miscellaneous

Null references are encoded by lifting all external object types to variant types with a null alternative, similar to the option data type in Standard ML. Then, all object operations must first verify that the object pointer is not null. Since this could fail, we also need support for unchecked exceptions, which can be achieved by adding an exception mechanism in the style of Standard ML.

Mutually recursive class and interface declarations need a fixed-point construction over the corresponding components in MINIFLINT; the fixed point already used in the definition of a single class cannot be generalized for this purpose since this will result in “friend” status of the other classes.

Other aspects of Java such as concurrency, final, reflection, class loaders, and dynamic loading are more challenging; we have preliminary ideas about implementing them, but developing the details remains as future work. Of course our F^ω-based framework is particularly well-suited for implementing Java extensions based on higher-order functions and types such as those in Pizza [27].

7 Related Work

Object and class encodings have been extensively studied. The use of row polymorphism positions our scheme most closely to that of Rémy and Vouillon [32], however the special object types of Objective ML reduce the use of row polymorphism to only the cases of binary methods [4], while the self-application semantics of our scheme uses rows to represent the open type of self even though Java lacks binary methods. The unordered records of Objective ML are geared towards support for multiple inheritance; in contrast our scheme makes single inheritance efficient, while an extension with multiple inheritance is still possible (but less efficient) using interface views as the basic representation.

In the context of object encodings in F^ω-based languages [5, 29], the genealogy of our encoding can be traced back to encodings based on F-bounded polymorphism [7, 13], from which the F-bounds have been eliminated using intersection types, which have further been replaced by row polymorphism (in the case of objects) or realized as tuples (in the case of interface views). At the most basic level, method invocation uses self-application (the whole object is a parameter of each method), showing similarity with the encoding due to Abadi, Cardelli, and Viswanathan [1]; however, hiding the actual class of the receiver is achieved 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 (modulo type applications, which are no-ops in the intended type-erasure semantics). Further we use an analog of the recursive-existential encoding due to Bruce [6] to give types to other arguments or results belonging to the same class or a subclass, as needed in Java, without restricting the type to be the same as the receiver’s. The private instance variables of these objects are thus accessible by methods of the class; they are protected by the final level of encapsulation.

Fisher and Mitchell [14, 15] show how to 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. In particular, extensibility of objects is only used when they have the role of prototypes, and the ability to extend an object in general takes us further from the intended Java semantics. In our translation these roles are clearly separated. Our encoding of classes guarantees restricted visibility and (as a consequence) correct initialization of private instance variables, and they provide directly reusable collections of methods as well as means to create new objects. One criterion our encoding fails is to automatically propagate changes in a base class to its descendants, but it is unclear if this criterion can coexist with Java’s binary compatibility.

8 Conclusions

We have presented a formal translation of Java classes, interfaces, and privacy into a call-by-value variant of F^ω using simple and well-known extensions. Even though the resulting code contains full type information, the runtime object layout corresponds to what
one might expect in an untyped implementation. The operations of object creation, method invocation, and field selection are implemented efficiently in terms of primitive \( F^\omega \) constructs. Thus, they are candidates for standard optimizations and we can reason about their interaction with foreign code.

An implementation of this encoding is in progress. As of April 1999, we finished a prototype implementation which can translate all the features of JavaCito into a simple, interpreted target calculus. We are working on connecting this simple target calculus implementation to the full-fledged FLINT compiler \([34, 35]\), in order to leverage its type-directed optimizations, compiler back ends, and runtime support. The actual FLINT intermediate language is surprisingly close to the target calculus presented here (MINIFLINT); all we need is just to add the row polymorphism.

Acknowledgments

We wish to thank the anonymous referees for their suggestions.

References


