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^ω) as an intermediate language for compiling languages with name-based class and interface hierarchies. Our extension enriches standard F^ω with recursive types, existential types, and row polymorphism, but only ordered records with no subtyping. Basing our language on F^ω 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 λ-calculus F^ω [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^ω 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 λ-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.

To appear in ICFP'99: Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming, September 1999, Paris, France.

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, $F^{\omega}_{\leq}$ [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 named types and to preserve the class and interface hierarchy. In addition, we informally describe how to extend our framework to support other interesting features of Java such as dynamic casts, checked exceptions, and mutually recursive declarations.

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).

```
1  class Pt {
2    private int x                = 0;
3    public Pt max (Pt other)     { (this.x > other.x)? this : other }
4    public void move (int dx)    { this.x = this.x + dx; }
5    public void bump ()          { this.move (1); }
6  }
7  interface Zm {public void zoom (int s);}
8  class SPt ext Pt imp Zm {
9    private int scale            = 1;
10   public void move (int dx)    { super.move (this.scale * dx); }
11   public void zoom (int s)     { this.scale = this.scale * s; }
12 }
13 class Main {
14   private Pt p                 = new Pt;
15   private SPt sp               = new SPt;
16   public void zoom2 (Zm z) { z.zoom (2); }
17   public void main () {
18     this.p.bump ();
19     this.zoom2 ((Zm) this.sp);
20     this.sp.bump ();
21     this.zoom2 ((Zm) (SPt)                       //cf. Section 6
22                 this.p.max ((Pt) this.sp)); }
23 }
24 (new Main).main ();
```

Figure 1: Sample *Javacito* program.

$$
\begin{array}{rcl}
p & ::= & decl^{*}\ exp \\
decl & ::= & \textbf{interface}\ i\ \textbf{ext}\ i^{*} \\
 & & \{(msig\textbf{;})^{*}\} \\
 & \mid & \textbf{class}\ c\ \textbf{ext}\ c'\ \textbf{imp}\ i^{*} \\
 & & \{field^{*}\ meth^{*}\} \\
field & ::= & \textbf{private}\ ty\ f = exp\textbf{;} \\
msig & ::= & \textbf{public}\ ty\ m\ ((ty\ x)^{*}) \\
meth & ::= & msig\ exp \\
ty & ::= & i \mid c
\end{array}
\qquad
\begin{array}{rcl}
exp & ::= & val \\
 & \mid & \{\ exp(\textbf{;}\ exp)^{*}\ \} \\
 & \mid & \textbf{new}\ c \\
 & \mid & (ty)\ exp \\
 & \mid & exp\textbf{.}m\ (exp^{*}) \\
 & \mid & \underline{c \cdot val \rhd}\ \textbf{super.}m\ (exp^{*}) \\
 & \mid & \underline{c \rhd} exp\textbf{.}f \\
 & \mid & \underline{c \rhd} exp\textbf{.}f = exp \\
val & ::= & x
\end{array}
$$

$$
\begin{array}{rcl}
i & \in & InterfaceNames \\
c & \in & ClassNames \cup \{\text{Object}\} \\
x & \in & VarNames \cup \{\textbf{this}\}
\end{array}
\qquad
\begin{array}{rcl}
f & \in & FieldNames \\
m & \in & MethNames
\end{array}
$$

Figure 2: Syntax of *Javacito*. Underlines indicate annotations required by the contextual operational semantics.

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 overrides 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-

Environment syntax:

Type signature
$$T \;::=\; ty \mid ty_1 \ldots ty_n \rightarrow ty$$
Hierarchy
$$\mathcal{H} \;::=\; \emptyset \mid \mathcal{H} \uplus \{i \mapsto [i^*]\} \mid \mathcal{H} \uplus \{c \mapsto \langle c', [i^*]\rangle\}$$
Member type environment
$$\mathcal{E} \;::=\; \emptyset \mid \mathcal{E} \uplus \{ty \mapsto \langle [\langle f, ty\rangle^*], [\langle m, T\rangle^*]\rangle\}$$
Member code environment
$$\mathcal{R} \;::=\; \emptyset \mid \mathcal{R} \uplus \{c \mapsto \langle \{\langle f, exp\rangle^*\}, \{\langle m, (x_1 \ldots x_n)\, exp\rangle^*\}\rangle\}$$

$\vdash_{\mathsf{H}}$

$$\frac{\vdash_{\mathsf{H}} \mathcal{H} \qquad i \notin \mathrm{dom}(\mathcal{H}) \qquad \forall j \in \{1..n\}.\ (i_j = i_{j'} \implies j = j' \text{ and } \mathcal{H} \vdash_{\mathsf{t}} i_j)}{\vdash_{\mathsf{H}} \mathcal{H} \uplus \{i \mapsto [i_1, \ldots, i_n]\}}$$

$$\frac{\vdash_{\mathsf{H}} \mathcal{H} \qquad c \notin \mathrm{dom}(\mathcal{H}) \qquad \mathcal{H} \vdash_{\mathsf{t}} c' \qquad \forall j \in \{1..n\}.\ (i_j = i_{j'} \implies j = j' \text{ and } \mathcal{H} \vdash_{\mathsf{t}} i_j)}{\vdash_{\mathsf{H}} \mathcal{H} \uplus \{c \mapsto \langle c', [i_1, \ldots, i_n]\rangle\}}$$

$\vdash_{\mathsf{E}}$

$$\frac{\begin{array}{ll} \mathcal{H} \vdash_{\mathsf{E}} \mathcal{E} & ty \in \mathrm{dom}(\mathcal{H}) \\ \forall j \in \{1..n\}.\ \mathcal{H} \vdash_{\mathsf{t}} ty_j & \forall j \in \{1..n\}.\ f_j = f_{j'} \implies j = j' \\ \forall k \in \{1..p\}.\ \mathcal{H} \vdash_{\mathsf{t}} T_k & \forall k \in \{1..p\}.\ m_k = m_{k'} \implies k = k' \\ \multicolumn{2}{l}{\forall k \in \{1..p\}.\ \forall ty'.\ (ty \prec_{\mathcal{H}} ty' \text{ and } \langle m_k, T'\rangle \in_{\mathcal{H}\mathcal{E}} ty') \implies T_k = T'} \end{array}}{\mathcal{H} \vdash_{\mathsf{E}} \mathcal{E} \uplus \{ty \mapsto \langle [\langle f_j, ty_j\rangle^{j \leftarrow [1..n]}], [\langle m_k, T_k\rangle^{k \leftarrow [1..p]}]\rangle\}}$$

Figure 3: Environment formation.

Hierarchy relations:

$c \prec^{\mathsf{c}}_{\mathcal{H}} c' \Leftrightarrow c' = \pi_1(\mathcal{H}(c))$
Class is declared as an immediate subclass

$i \prec^{\mathsf{i}}_{\mathcal{H}} i' \Leftrightarrow i' \in \mathcal{H}(i)$
Interface is declared as an immediate subinterface

$c \prec\!\!\prec^{\mathsf{c}}_{\mathcal{H}} i \Leftrightarrow i \in \pi_2(\mathcal{H}(c))$
Class declares implementation of an interface

Derived hierarchy relations:

$\prec_{\mathcal{H}} \;\equiv\; \prec^{\mathsf{c}}_{\mathcal{H}} \cup \prec^{\mathsf{i}}_{\mathcal{H}} \cup \prec\!\!\prec^{\mathsf{c}}_{\mathcal{H}}$: Type is an immediate subtype

$\leq^{\mathsf{c}}_{\mathcal{H}} \;\equiv\;$ reflexive transitive closure of $\prec^{\mathsf{c}}_{\mathcal{H}}$: Class is a subclass

$\leq^{\mathsf{i}}_{\mathcal{H}} \;\equiv\;$ reflexive transitive closure of $\prec^{\mathsf{i}}_{\mathcal{H}}$
Interface is a subinterface

$c \ll^{\mathsf{c}}_{\mathcal{H}} i \Leftrightarrow \exists c', i'$ s.t. $c \leq^{\mathsf{c}}_{\mathcal{H}} c'$ and $c' \prec\!\!\prec^{\mathsf{c}}_{\mathcal{H}} i'$ and $i' \leq^{\mathsf{i}}_{\mathcal{H}} i$
Class implements an interface

$\leq_{\mathcal{H}} \;\equiv\; \leq^{\mathsf{c}}_{\mathcal{H}} \cup \leq^{\mathsf{i}}_{\mathcal{H}} \cup \ll^{\mathsf{c}}_{\mathcal{H}}$: Type is a subtype

Membership relations:

$\langle f, X\rangle \in^{\mathsf{c}}_{\mathcal{E}} c \Leftrightarrow \langle f, X\rangle \in \pi_1(\mathcal{E}(c))$: Field is declared in class

$\langle m, X\rangle \in_{\mathcal{E}} ty \Leftrightarrow \langle m, X\rangle \in \pi_2(\mathcal{E}(ty))$: Method is declared in type

Derived membership relations:

$\langle f, X\rangle \in^{\mathsf{c}}_{\mathcal{H}\mathcal{E}} c \Leftrightarrow \langle f, X\rangle \in^{\mathsf{c}}_{\mathcal{E}} c'$ and $c \leq^{\mathsf{c}}_{\mathcal{H}} c'$
Field is contained in class

$\langle m, X\rangle \in^{\mathsf{c}}_{\mathcal{H}\mathcal{E}} c \Leftrightarrow \langle m, X\rangle \in_{\mathcal{E}} c'$ and
$c' = \min\{c'' \mid c \leq^{\mathsf{c}}_{\mathcal{H}} c'' \text{ and } \langle m, \_\rangle \in_{\mathcal{E}} c''\}$
Method is contained in class

$\langle m, X\rangle \in^{\mathsf{i}}_{\mathcal{H}\mathcal{E}} i \Leftrightarrow \langle m, X\rangle \in_{\mathcal{E}} i'$ and $i \leq^{\mathsf{i}}_{\mathcal{H}} i'$
Method is contained in interface

$\in_{\mathcal{H}\mathcal{E}} \;\equiv\; \in^{\mathsf{c}}_{\mathcal{H}\mathcal{E}} \cup \in^{\mathsf{i}}_{\mathcal{H}\mathcal{E}}$: Member is contained in type

Figure 4: Environment relations.

ences, public fields, private methods, static members, **protected**, package scope, **final**, constructors, finalizers, mutually recursive classes, exceptions, reflection, and concurrency. As discussed informally in Section 6, many of these features are simple extensions of the framework. Although a dynamic cast appears in the example (line 21), we also treat this feature as an extension.

We use integers, arithmetic operations, conditional expressions, and **void** in the example without defining them formally. We do not distinguish between statements and expressions; rather, we use an ML-like sequence expression with syntax { *exp*(**;** *exp*)* }. The last expression in this sequence is implicitly returned.

Finally, we assume several transformations have been made to the Java code in a pre-processing phase. For instance, all implicit references to **this** are made explicit. Overloaded method references are resolved statically by including argument types as part of the method name. Instance variable shadowing is not an issue in this restricted language because all fields are private, so **super** is only needed for invoking methods in the superclass. Implicit subsumption is made explicit by inserting sequences of upward casts. Similar transformations are defined formally in several papers by Drossopoulou and Eisenbach [11, 12].

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 $\underline{c \rhd}$ indicates that the term is statically enclosed in the declaration of class c. In the **super** term, the additional *val* in the annotation holds the value of **this**. These annotations are required by the contextual operational semantics, as explained below.

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 $\mathcal{H}$ is an environment which maps class or interface names to their immediate superclasses and superinterfaces. The hierarchy formation judgment $\vdash_{\mathsf{H}}$ 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 $\mathcal{E}$ maps types to lists of field and methods names, paired with the respective types. The formation rule ensures that the names are unique, the types are valid, and that overriding a method preserves its type.[1] The member code environment $\mathcal{R}$ maps class names to a set of field initializers and method bodies. This environment is created during elaboration, for use at runtime.

We use a set of hierarchy and membership relations similar to those used to describe ClassicJava [17], except that ours are defined in terms of environments $\mathcal{H}$ and $\mathcal{E}$ rather than on complete programs. The membership relations apply to both type and code environments. Note that for methods contained in class c ($\langle m, X\rangle \in^{\mathsf{c}}_{\mathcal{H}\mathcal{E}} c$), the auxiliary datum X is drawn from the *nearest* superclass of c that declares the method m. When a code environment $\mathcal{R}$ is used in place of $\mathcal{E}$, this has the effect of selecting the appropriate method body by using the inheritance hierarchy.

Figure 5 contains the typing judgments comprising the static semantics of *Javacito*. Declarations are processed one at a time; they may be recursive, but not mutually recursive. All declarations extend the hierarchy $\mathcal{H}$ and the member type environment $\mathcal{E}$. Class declarations also extend the member code environment $\mathcal{R}$, which is used to retrieve code at runtime. Unlike in ClassicJava, there are no rules for subsumption. Instead, we require explicit upward casts. Casts are only legal from some type to its immediate supertype (exp-cast). Privacy is enforced in (exp-get) and (exp-set), the rules for field access. A field f can only be accessed from within the class c that declares it. The object from which f is selected may be

[1] 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.

$\vdash_{\mathsf{p}}$

(prog-decl)

$$\frac{\begin{array}{l}\mathcal{H};\mathcal{E};\mathcal{R} \vdash_{\mathsf{d}} decl \Rightarrow \mathcal{H}';\mathcal{E}';\mathcal{R}' \\ \mathcal{H}';\mathcal{E}';\mathcal{R}' \vdash_{\mathsf{p}} decl_1 \ldots decl_n\ exp : ty \Rightarrow \mathcal{H}'';\ \mathcal{R}''\end{array}}{\mathcal{H};\mathcal{E};\mathcal{R} \vdash_{\mathsf{p}} decl\ decl_1 \ldots decl_n\ exp : ty \Rightarrow \mathcal{H}'';\ \mathcal{R}''}$$

(prog-exp)

$$\frac{\mathcal{H};\mathcal{E};\emptyset;\text{Object} \vdash_{\mathsf{e}} exp : ty}{\mathcal{H};\mathcal{E};\mathcal{R} \vdash_{\mathsf{p}} exp : ty \Rightarrow \mathcal{H};\ \mathcal{R}}$$

$\vdash_{\mathsf{t}}$

(typ-ok)

$$\frac{ty \in \text{dom}(\mathcal{H}) \cup \{\text{Object}\}}{\mathcal{H} \vdash_{\mathsf{t}} ty}$$

(typ-msig)

$$\frac{\begin{array}{l}\forall j \in \{1..n\}.\ \mathcal{H} \vdash_{\mathsf{t}} ty_j \\ \mathcal{H} \vdash_{\mathsf{t}} ty\end{array}}{\mathcal{H} \vdash_{\mathsf{t}} ty_1 \ldots ty_n \to ty}$$

$\vdash_{\mathsf{m}}$

(meth-body)

$$\frac{\mathcal{H};\mathcal{E};\{\mathbf{this} : c, x_1 : ty_1, \ldots, x_n : ty_n\}; c \vdash_{\mathsf{e}} exp : ty}{\mathcal{H};\mathcal{E};c \vdash_{\mathsf{m}} ty\ m(ty_1\ x_1, \ldots, ty_n\ x_n)\ exp}$$

$\vdash_{\mathsf{d}}$

(decl-iface)

$$\frac{\begin{array}{ll}\mathcal{H}' = \mathcal{H} \uplus \{i \mapsto [i_1, \ldots i_n]\} & \vdash_{\mathsf{H}} \mathcal{H}' \\ \mathcal{E}' = \mathcal{E} \uplus \{i \mapsto \langle [], [\text{MT}(msig_k)^{k \leftarrow [1..m]}] \rangle\} & \mathcal{H}' \vdash_{\mathsf{E}} \mathcal{E}'\end{array}}{\mathcal{H};\mathcal{E};\mathcal{R} \vdash_{\mathsf{d}} \mathbf{interface}\, i\ \mathbf{ext}\, i_1 \ldots i_n \{msig_1\mathbf{;} \ldots msig_m\mathbf{;}\} \Rightarrow \mathcal{H}';\mathcal{E}';\mathcal{R}}$$

(decl-class)

$$\frac{\begin{array}{l}\mathcal{H}' = \mathcal{H} \uplus \{c \mapsto \langle c', [i_1 \ldots i_n] \rangle\} \\ \vdash_{\mathsf{H}} \mathcal{H}' \\ \mathcal{E}' = \mathcal{E} \uplus \{c \mapsto \langle [\langle f_k, ty_k \rangle^{k \leftarrow [1..m]}], [\text{MT}(msig_l)^{l \leftarrow [1..p]}] \rangle\} \\ \mathcal{H}' \vdash_{\mathsf{E}} \mathcal{E}' \\ \mathcal{R}' = \mathcal{R} \uplus \{c \mapsto \langle \{\langle f_k, exp_k \rangle \mid k \in \{1..m\}\}, \\ \qquad\qquad\qquad \{\text{MR}(msig_l\ exp'_l) \mid l \in \{1..p\}\} \rangle\} \\ \forall j \in \{1..n\}.\ \text{if } \langle m', T' \rangle \in^{\mathsf{i}}_{\mathcal{H}\mathcal{E}} i_j \text{ then } \langle m', T' \rangle \in^{\mathsf{c}}_{\mathcal{H}\mathcal{E}} c \\ \forall k \in \{1..m\}.\ \mathcal{H}';\mathcal{E}';\emptyset;c \vdash_{\mathsf{e}} exp_k : ty_k \\ \forall l \in \{1..p\}.\ \mathcal{H}';\mathcal{E}';c \vdash_{\mathsf{m}} msig_l\ exp'_l\end{array}}{\mathcal{H};\mathcal{E};\mathcal{R} \vdash_{\mathsf{d}} \begin{array}{l}\mathbf{class}\, c\ \mathbf{ext}\, c'\ \mathbf{imp}\, i_1 \ldots i_n \\ \{ty_1\ f_1 = exp_1\mathbf{;} \ldots ty_m\ f_m = exp_m\mathbf{;} \\ \ msig_1\ exp'_1 \ldots msig_p\ exp'_p\}\end{array} \Rightarrow \mathcal{H}';\mathcal{E}';\mathcal{R}'}$$

$\vdash_{\mathsf{e}}$

(exp-var)

$$\frac{x \in \text{dom}(\Gamma)}{\_;\_;\Gamma;\_ \vdash_{\mathsf{e}} x : \Gamma(x)}$$

(exp-cast)

$$\frac{\mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} exp : ty' \qquad ty' \prec_{\mathcal{H}} ty}{\mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} (ty)\, exp : ty}$$

(exp-new)

$$\frac{\mathcal{H} \vdash_{\mathsf{t}} c}{\mathcal{H};\_;\_;\_ \vdash_{\mathsf{e}} \mathbf{new}\, c : c}$$

(exp-body)

$$\frac{\forall j \in \{1..n\}.\ \mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} exp_j : ty_j}{\mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} \{\ exp_1\mathbf{;} \ldots \mathbf{;}exp_n\ \} : ty_n}$$

(exp-call)

$$\frac{\begin{array}{l}\mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} exp : ty' \qquad \langle m, (ty_1 \ldots ty_n \to ty) \rangle \in_{\mathcal{H}\mathcal{E}} ty' \\ \forall j \in \{1..n\}.\ \mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} exp_j : ty_j\end{array}}{\mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} exp\mathbf{.}m\,(exp_1 \ldots exp_n) : ty}$$

(exp-get)

$$\frac{\begin{array}{l}\mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} exp : ty' \\ ty' \leq^{\mathsf{c}}_{\mathcal{H}} c \qquad \langle f, ty \rangle \in^{\mathsf{c}}_{\mathcal{E}} c\end{array}}{\mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} \underline{c \triangleright} exp\mathbf{.}f : ty}$$

(exp-set)

$$\frac{\begin{array}{l}\mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} exp_1 : ty' \qquad ty' \leq^{\mathsf{c}}_{\mathcal{H}} c \\ \mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} exp_2 : ty \qquad \langle f, ty \rangle \in^{\mathsf{c}}_{\mathcal{E}} c\end{array}}{\mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} \underline{c \triangleright} exp_1\mathbf{.}f = exp_2 : ty}$$

(exp-super)

$$\frac{\begin{array}{l}c \prec^{\mathsf{c}}_{\mathcal{H}} c' \qquad \langle m, (ty_1 \ldots ty_n \to ty) \rangle \in_{\mathcal{H}\mathcal{E}} c' \\ \forall j \in \{1..n\}.\ \mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} exp_j : ty_j\end{array}}{\mathcal{H};\mathcal{E};\Gamma;c \vdash_{\mathsf{e}} \underline{c \cdot \mathbf{this} \triangleright}\, \mathbf{super.}m\,(exp_1 \ldots exp_n) : ty}$$

Utility functions:

$$\begin{array}{lcl}\text{MT}(ty\ m(ty_1\ x_1, \ldots, ty_n\ x_n)) & = & \langle m, (ty_1 \ldots ty_n \to ty) \rangle \\ \text{MR}(ty\ m(ty_1\ x_1, \ldots, ty_n\ x_n)\ exp) & = & \langle m, (x_1 \ldots x_n)\ exp \rangle\end{array}$$

Figure 5: Static semantics of *Javacito*.

a subclass of c, however.

Proposition 1 (Decidability) *$\vdash_{p}$ is decidable.*

2.2 Operational semantics

Figure 6 contains the contextual small-step operational semantics of *Javacito*. We introduce two new terms: object expressions $\langle c, [(f \mapsto exp)^*] \rangle$ and locations *loc*. 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 $(val_1, \ldots, val_n, E, \ldots)$ 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 $\hookrightarrow$ is defined between these states, given the class hierarchy $\mathcal{H}$ and the code $\mathcal{R}$. 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 $\mathcal{H};\mathcal{R} \vdash_{\mathsf{c}} \langle exp, \mathcal{S} \rangle : ty$ 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 (F^{ω}) [18, 33]. We extend F^{ω} with standard constructs for recursive types of base kind (with explicit **fold** and **unfold**), existential types with dot nota-

Syntactic extensions:

$$
\begin{array}{rrcl}
\text{Object expression} & \mathit{exp} & ::= & \ldots \mid \langle c, [(f \mapsto \mathit{exp})^*] \rangle \\
\text{Location value} & \mathit{val} & ::= & \ldots \mid \mathit{loc} \\
\\
\text{Field map} & \mathcal{F} & ::= & \emptyset \mid \mathcal{F} \uplus [f \mapsto \mathit{val}] \\
\text{Runtime store} & \mathcal{S} & ::= & \emptyset \mid \mathcal{S} \uplus \{\mathit{loc} \mapsto \langle c, \mathcal{F} \rangle\}
\end{array}
$$

Evaluation contexts:

$$
\begin{array}{rcl}
E & ::= & \bullet \mid \{\, E \ldots \} \mid (\mathit{ty})\, E \\
& \mid & \langle c, [f \mapsto \mathit{val}_1, \ldots, f \mapsto \mathit{val}_n, f \mapsto E, \ldots] \rangle \\
& \mid & E.m\,(\ldots) \mid \mathit{val}.m\,(\mathit{val}_1, \ldots, \mathit{val}_n, E, \ldots) \\
& \mid & \underline{c \cdot \mathit{val} \triangleright} \, \mathbf{super}.m\,(\mathit{val}, \ldots, \mathit{val}_n, E, \ldots) \\
& \mid & \underline{c \triangleright} E.f \mid \underline{c \triangleright} E.f = \mathit{exp} \mid \underline{c \triangleright} \mathit{val}.f = E
\end{array}
$$

Primitive reductions:

$$
\begin{array}{rl}
\text{(r-new)} & \mathcal{H}; \mathcal{R} \;\vdash \langle E[\mathbf{new}\, c], \mathcal{S} \rangle \hookrightarrow \langle E[\langle c, \mathcal{F} \rangle], \mathcal{S} \rangle \\
& \quad \text{where } \mathcal{F} = [f \mapsto \mathit{exp} \mid \langle f, \mathit{exp} \rangle \in^{\mathsf{C}}_{\mathcal{HR}} c] \\
\text{(r-alloc)} & \mathcal{H}; \mathcal{R} \;\vdash \langle E[\langle c, [f_j \mapsto \mathit{val}_j{}^{j \in \{1..n\}}] \mapsto \rangle], \mathcal{S} \rangle \\
& \quad \hookrightarrow \langle E[\mathit{loc}], \mathcal{S}[\mathit{loc} \mapsto \langle c, [f_j \mapsto \mathit{val}_j{}^{j \in \{1..n\}}] \rangle] \rangle \\
& \quad \text{where } \mathit{loc} \notin \mathrm{dom}(\mathcal{S}) \\
\text{(r-body)} & \mathcal{H}; \mathcal{R} \;\vdash \langle E[\{\, \mathit{val};\, \mathit{exp}_1;\; \ldots;\, \mathit{exp}_n \,\}], \mathcal{S} \rangle \\
& \quad \hookrightarrow \langle E[\{\, \mathit{exp}_1;\; \ldots;\, \mathit{exp}_n \,\}], \mathcal{S} \rangle \\
\text{(r-return)} & \mathcal{H}; \mathcal{R} \;\vdash \langle E[\{\, \mathit{val} \,\}], \mathcal{S} \rangle \hookrightarrow \langle E[\mathit{val}], \mathcal{S} \rangle \\
\text{(r-call)} & \mathcal{H}; \mathcal{R} \;\vdash \langle E[\mathit{loc}.m\,(\mathit{val}_1 \ldots \mathit{val}_n)], \mathcal{S} \rangle \\
& \quad \hookrightarrow \langle E[\mathit{exp}[\mathit{loc}/\mathbf{this}, \mathit{val}_1/x_1 \ldots \mathit{val}_n/x_n]], \mathcal{S} \rangle \\
& \quad \text{where } \mathcal{S}(\mathit{loc}) = \langle c', \_ \rangle \\
& \quad \text{and } \langle m, (x_1 \ldots x_n)\, \mathit{exp} \rangle \in^{\mathsf{C}}_{\mathcal{HR}} c' \\
\text{(r-super)} & \mathcal{H}; \mathcal{R} \;\vdash \langle E[\underline{c \cdot \mathit{loc} \triangleright} \, \mathbf{super}.m\,(\mathit{val}_1 \ldots \mathit{val}_n)], \mathcal{S} \rangle \\
& \quad \hookrightarrow \langle E[\mathit{exp}[\mathit{loc}/\mathbf{this}, \mathit{val}_1/x_1 \ldots \mathit{val}_n/x_n]], \mathcal{S} \rangle \\
& \quad \text{where } c \prec^{\mathsf{C}}_{\mathcal{H}} c' \\
& \quad \text{and } \langle m, (x_1 \ldots x_n)\, \mathit{exp} \rangle \in^{\mathsf{C}}_{\mathcal{HR}} c' \\
\text{(r-get)} & \mathcal{H}; \mathcal{R} \;\vdash \langle E[\underline{c \triangleright} \mathit{loc}.f], \mathcal{S} \rangle \hookrightarrow \langle E[\mathit{val}], \mathcal{S} \rangle \\
& \quad \text{where } \mathcal{S}(\mathit{loc}) = \langle c', \mathcal{F} \rangle \text{ and } \mathcal{F}(f) = \mathit{val} \\
& \quad \text{and } \langle f, \_ \rangle \in^{\mathsf{C}}_{\mathcal{R}} c \text{ and } c' \leq^{\mathsf{C}}_{\mathcal{H}} c \\
\text{(r-set)} & \mathcal{H}; \mathcal{R} \;\vdash \langle E[\underline{c \triangleright} \mathit{loc}.f = \mathit{val}], \mathcal{S} \rangle \\
& \quad \hookrightarrow \langle E[\mathit{val}], \mathcal{S}[\mathit{loc} \mapsto \langle c, \mathcal{F}' \rangle] \rangle \\
& \quad \text{where } \mathcal{S}(\mathit{loc}) = \langle c', \mathcal{F} \rangle \text{ and } c' \leq^{\mathsf{C}}_{\mathcal{H}} c \\
& \quad \text{and } \mathcal{F}' = \mathcal{F}[f \mapsto \mathit{val}] \text{ and } \langle f, \_ \rangle \in^{\mathsf{C}}_{\mathcal{R}} c \\
\text{(r-cast)} & \mathcal{H}; \mathcal{R} \;\vdash \langle E[(\mathit{ty})\, \mathit{loc}], \mathcal{S} \rangle \hookrightarrow \langle E[\mathit{loc}], \mathcal{S} \rangle \\
& \quad \text{where } \mathcal{S}(\mathit{loc}) = \langle c, \_ \rangle \text{ and } c \leq_{\mathcal{H}} \mathit{ty}
\end{array}
$$

Figure 6: Operational semantics of *Javacito.*

$$
\begin{array}{rcl}
\kappa & ::= & \Omega \mid R^L \mid \kappa \to \kappa' \\
\tau & ::= & \mathsf{Ref}\ \tau \mid \tau \to \tau' \\
& \mid & t \mid x.\mathsf{Typ} \mid \forall t{::}\kappa.\, \tau \mid \exists t{::}\kappa.\, \tau \mid \mu t.\, \tau \mid \lambda t{::}\kappa.\, \tau \mid \tau\ \tau' \\
& \mid & \mathsf{Abs}^L \mid l : \tau; \tau' \mid \{\tau\} \\
e & ::= & \mathsf{ref}\ e \mid \mathsf{!}\, e \mid e \mathsf{:=} e' \\
& \mid & x \mid \lambda x : \tau.\, e \mid e\ x \mid \mathsf{let}\ x = e'\ \mathsf{in}\ e \\
& \mid & \mathsf{fold}\ e\ \mathsf{as}\ \tau \mid \mathsf{unfold}\ e \\
& \mid & \Lambda t{::}\kappa.\, e \mid e\, [\tau] \mid \langle t{::}\kappa = \tau, e : \tau' \rangle \mid x.\mathsf{val} \\
& \mid & \{(l = e)^*\} \mid e.l
\end{array}
$$

$t \in \mathit{TypeVars} \qquad x \in \mathit{Vars} \qquad l \in \mathit{Labels} \qquad L \in \mathcal{P}(\mathit{Labels})$

Derived forms:

$$
\begin{array}{c}
\{l_1 : \tau_1; \ldots l_n : \tau_n\} \triangleq \{l_1 : \tau_1; \ldots l_n : \tau_n; \mathsf{Abs}^{\{l_1, .. l_n\}}\} \\
\mathbf{1} \triangleq \{\mathsf{Abs}^{\emptyset}\} \\
e\ e' \triangleq \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{let}\ x' = e'\ \mathsf{in}\ x\ x',
\end{array}
$$

where x is not free in e'

Figure 7: Syntax of MINIFLINT.

Types

$$
\frac{\vdash A\ \mathsf{env} \quad A(x) = \exists t{::}\kappa.\, \tau}{A \vdash x.\mathsf{Typ}\ ::\ \kappa}
\qquad
\frac{\vdash A\ \mathsf{env}}{A \vdash \mathsf{Abs}^L\ ::\ R^L}
$$

$$
\frac{A \vdash \tau\ ::\ \Omega \quad A \vdash \tau'\ ::\ R^{L \cup \{l\}}}{A \vdash l : \tau; \tau'\ ::\ R^{L - \{l\}}}
\qquad
\frac{A \vdash \tau\ ::\ R^{\emptyset}}{A \vdash \{\tau\}\ ::\ \Omega}
$$

Terms

$$
\frac{A,\, x : \tau' \vdash e\ :\ \tau \quad A \vdash \tau\ ::\ \Omega}{A \vdash \lambda x : \tau'.\, e\ :\ \tau' \to \tau}
\qquad
\frac{A \vdash e\ :\ \{l_1 : \tau_1; ..l_n : \tau_n; \tau'\}}{A \vdash e.l_n\ :\ \tau_n}
$$

$$
\frac{\forall j \in \{1..n\}.\ A \vdash e_j\ :\ \tau_j}{A \vdash \{l_1 = e_1, ..l_n = e_n\}\ :\ \{l_1 : \tau_1; ..l_n : \tau_n\}}
$$

$$
\frac{A \vdash e\ :\ \tau[\mu t.\, \tau / t]}{A \vdash \mathsf{fold}\ e\ \mathsf{as}\ \mu t.\, \tau\ :\ \mu t.\, \tau}
\qquad
\frac{A \vdash \tau'\ ::\ \kappa \quad A \vdash e\ :\ \tau[\tau'/t]}{A \vdash \langle t{::}\kappa = \tau', e : \tau \rangle\ :\ \exists t{::}\kappa.\, \tau}
$$

$$
\frac{A \vdash e\ :\ \mu t.\, \tau}{A \vdash \mathsf{unfold}\ e\ :\ \tau[\mu t.\, \tau / t]}
\qquad
\frac{\vdash A\ \mathsf{env} \quad A(x) = \exists t{::}\kappa.\, \tau}{A \vdash x.\mathsf{val}\ ::\ \tau[x.\mathsf{Typ}/t]}
$$

Figure 8: Static semantics of MINIFLINT—
less common rules.

tion [8], tuples with row polymorphism, and a store. The syntax is given in Figure 7, and the static semantics in Figure 8. Notice that we only list the formation rules for environments, types, and terms; the type reduction and equivalence rules are omitted.

Following Rémy [31] we introduce a kind of rows R^L, where L is the set of labels banned from the row. Abs^L is the empty row missing L, and $l : \tau; \tau'$ constructs a row from the element l of type τ and the row τ'. The record constructor $\{\,\cdot\,\}$ lifts a complete row type (with no labels missing) to kind Ω.

Consider the following example, a function which fetches field l from a record, regardless of what other fields are present:

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{fst} = \Lambda t{::}R^{\{l\}}.\, \lambda x : \{l : \mathsf{Int}; t\}.\, (x.l) \\
\ \mathsf{in}\ \mathsf{fst}\, [\mathsf{Abs}^{\{l\}}]\ \{l = 5\} + \mathsf{fst}\, [m : \mathsf{Int}; \mathsf{Abs}^{\{l,m\}}]\ \{l = 6, m = 7\}
\end{array}
$$

We say the function fst is polymorphic in the *tail* of its argument. 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 $\exists t{::}\kappa.\, \tau$ is the type of a pair containing a hidden type τ' of kind κ and a term of type $\tau[\tau'/t]$. We use dot notation [8] to access the type ($x.\mathsf{Typ}$) and term ($x.\mathsf{val}$) components of a package x. Although this is a form of dependent type, the calculus remains decidable because we restrict the $\cdot.\mathsf{Typ}$ operation to term variables x. We provide a let construct to bind term variables and limit their scope. Abstract types $x.\mathsf{Typ}$ and $x'.\mathsf{Typ}$ are equivalent if and only if x and x' are bound by the same let. The argument of an application must be a variable so that we can track this equivalence through function applications. Variables are prevented from escaping the scope of λ-abstractions $\lambda x : \tau.\, e$ by banning free occurrences of x in the type of e.

Proposition 2 (Decidability) *The* MINIFLINT *typing relation* $A \vdash e\ :\ \tau$ *is decidable.*

We omit the dynamic semantics of MINIFLINT, which defines

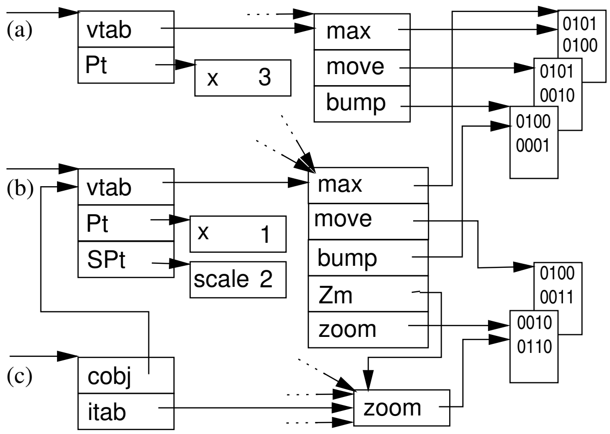

Figure 9: Object layout. (a) and (b) are instances of classes Pt and SPt, respectively. (c) is a view of (b) through the Zm interface.

locations, stores S, values v, and a reduction relation $\hookrightarrow$ between term-store pairs. Since runtime behavior does not depend on types, we can use a type-erasure semantics; this implies that **fold**, **unfold**, type application, package creation, and package projection are all no-ops at runtime. However for proving type soundness and correctness of the translation of *Javacito* to MINIFLINT we define a reduction semantics on typed terms. The reader is referred to the accompanying technical report for details.

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 (*vtable*) 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 both the object record and the vtable of (a) are prefixes of those in (b). This property allows superclass methods to access objects created by subclasses without any coercions.

In addition to its four methods, the vtable for class SPt contains a pointer to an interface table (*itable*). An itable is merely a rearrangement of some of the methods in the vtable 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 a particular interface, independent of the underlying vtable.

There are various ways of locating methods in interface objects. One technique, used in the Toba ahead-of-time compiler [30], is to construct a per-class dictionary that maps method names to offsets in the vtable. Another idea, implemented in the CACAO 64-bit JIT compiler [22] and described informally elsewhere [26], is to construct an itable for each interface we implement and store them all somewhere in the vtable. Although [22] is 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 dictionary lookup on interface names rather than on 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. The object marked (c) in Figure 9 is the result of casting an SPt object (b) to the Zm interface.

In our encoding, we use coercions to achieve fixed-offset 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:

$$
\begin{array}{l}
\textbf{let } x_c = \langle t_{\mathsf{own}} :: \Omega = \boxed{5.1.1}\ , \\
\qquad\qquad \{\mathsf{dict} = \Lambda t_{\mathsf{subFlds}}.\Lambda t_{\mathsf{subVtab}}.\{\ \boxed{5.3}\ \}, \\
\qquad\qquad\ \ \mathsf{init} = \lambda\_ : \mathbf{1}.\ \{\ \boxed{5.1.3}\ \}, \\
\qquad\qquad\ \ \mathsf{new} = \lambda\_ : \mathbf{1}.\ \boxed{5.3.1} \\
\qquad\qquad\ \ \} : \boxed{5.1.2, 5.2}\ \rangle \\
\ \textbf{in } \ldots
\end{array}
$$

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, $x_\bullet$ 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 (t_{own}) 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.**Typ**) but it remains abstract. The value part of the package is a record containing an extensible dictionary (dict), a field initializer (init), and a constructor (new). We say dict is *extensible* 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 $OwnFlds[c]$. For example,

$\mathit{OwnFlds}[\mathrm{Pt}] = \{\mathsf{x} : \textbf{Ref Int}\}$. 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 $\mathit{OwnFlds}[c]$ 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 List 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, $\mathit{OwnFlds}[c]$ must be a recursive type:

$$\mathit{OwnFlds}[c] = \mu t_{\mathsf{own}}.\ \mathit{OwnFldsGen}[c]\ \ (\mathit{ObjGen}[c]\ \ t_{\mathsf{own}})$$

where $\mathit{ObjGen}[c]$ (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.) $\mathit{OwnFldsGen}[c]$ takes an argument t_{twin}, creates a record containing **ref** cells for each field declared in c, and makes any fields of type c have type t_{twin} instead. For the class List mentioned above,

$$\mathit{OwnFldsGen}[\mathrm{List}] = \lambda t_{\mathsf{twin}}.\ \{\mathsf{data} : \textbf{Ref Int};\ \mathsf{next} : \textbf{Ref}\ t_{\mathsf{twin}}\}$$

Substituting this into $\mathit{OwnFlds}[\mathrm{List}]$, we have:

$$\begin{array}{rl}\mathit{OwnFlds}[\mathrm{List}] = \mu t_{\mathsf{own}}.\ \{\mathsf{data} : \textbf{Ref Int}; & \\ & \mathsf{next} : \textbf{Ref}\ (\mathit{ObjGen}[\mathrm{List}]\ \ t_{\mathsf{own}})\}\end{array}$$

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 $\mathit{AllFldsGen}_f[c]$ produce the type of the segregated record containing all fields in an object of class c. In the example,

$$\begin{array}{l}\mathit{AllFldsGen}_f[\mathrm{Pt}]\ \ = \lambda t_{\mathsf{own}}.\ \{\mathsf{Pt} : t_{\mathsf{own}}\}\\ \mathit{AllFldsGen}_f[\mathrm{SPt}] = \lambda t_{\mathsf{own}}.\ \{\mathsf{Pt} : \mathsf{Pt}.\textbf{Typ};\ \mathsf{SPt} : t_{\mathsf{own}}\}\end{array}$$

where $\mathsf{Pt}.\textbf{Typ}$ is the abstract type of private fields in Pt. t_{own} can be instantiated to $\mathit{OwnFlds}[c]$ for use inside the class c, or to $x_c.\textbf{Typ}$ outside, where x_c 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 t_{subFlds}, a row of field records defined by potential subclasses. Now we can define the SPt fields using the Pt fields directly:

$$\begin{array}{l}\mathit{AllFldsGen}[\mathrm{Pt}]\ \ = \lambda t_{\mathsf{own}}.\ \lambda t_{\mathsf{subFlds}}.\ \{\mathsf{Pt} : t_{\mathsf{own}};\ t_{\mathsf{subFlds}}\}\\ \mathit{AllFldsGen}[\mathrm{SPt}] = \lambda t_{\mathsf{own}}.\ \lambda t_{\mathsf{subFlds}}.\\ \qquad\qquad \mathit{AllFldsGen}[\mathrm{Pt}]\ \ \mathsf{Pt}.\textbf{Typ}\ \ (\mathsf{SPt} : t_{\mathsf{own}};\ t_{\mathsf{subFlds}})\end{array}$$

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_c exports init as a function of type $\mathbf{1} \to x_c.\textbf{Typ}$. For the Pt class in our example, $\mathsf{Pt}.\textbf{val}.\mathsf{init}$ is

$$\begin{array}{l}
\mathit{UnpackObj}[c]\ (t_{\mathsf{own}},\ o : \mathit{ObjGen}[c]\ \ t_{\mathsf{own}},\ \bullet) =\\
\quad \textbf{let}\ x = \textbf{unfold}\ o\ \textbf{in let}\ x' = x.\textbf{val in}\ \bullet\\
\mathit{PackObj}[c]\\
\quad (t_{\mathsf{own}},\ t_{\mathsf{subFlds}},\ t_{\mathsf{subVtab}},\ \mathsf{self} : \mathit{SelfGen}[c]\ \ t_{\mathsf{own}}\ \ t_{\mathsf{subFlds}}\ \ t_{\mathsf{subVtab}})\\
\quad =\\
\quad \textbf{fold}\ \langle t_{\mathsf{subFlds}} = t_{\mathsf{subFlds}},\\
\qquad\quad \langle t_{\mathsf{subVtab}} = t_{\mathsf{subVtab}},\\
\qquad\qquad \mathsf{self} : \mathit{SelfGen}[c]\ \ t_{\mathsf{own}}\ \ t_{\mathsf{subFlds}}\ \ t_{\mathsf{subVtab}}\rangle\\
\qquad\quad : \exists t_{\mathsf{subVtab}}.\ \mathit{SelfGen}[c]\ \ t_{\mathsf{own}}\ \ t_{\mathsf{subFlds}}\ \ t_{\mathsf{subVtab}}\rangle\\
\quad \textbf{as}\ \mathit{ObjGen}[c]\ \ t_{\mathsf{own}}\\
\mathit{UnpackView}[i]\ (io : \mathit{ExtView}[i],\ \bullet) = \textbf{let}\ x = \textbf{unfold}\ io\ \textbf{in}\ \bullet\\
\mathit{PackView}[i]\ (t_{\mathsf{cobj}},\ o : t_{\mathsf{cobj}},\ it : \mathit{ExtItab}[i]\ \ t_{\mathsf{cobj}}) =\\
\quad \textbf{fold}\ \langle t_{\mathsf{cobj}} :: \Omega = t_{\mathsf{cobj}},\\
\qquad\quad \{\mathsf{cobj} = o,\ \mathsf{itab} = it\}\\
\qquad\quad : \mathit{ViewTmpl}[i]\ \ (x_i.\textbf{Typ})\ \ \mathit{ExtView}[i]\ \ t_{\mathsf{cobj}}\rangle\\
\quad \textbf{as}\ \mathit{ExtView}[i]
\end{array}$$

$$\begin{array}{l}
\mathit{GetFields}[c]\ (o : \mathit{ObjGen}[c]\ \ \mathit{OwnFlds}[c]) =\\
\quad \mathit{UnpackObj}[c]\\
\qquad (\mathit{OwnFlds}[c],\ o,\ \textbf{unfold}\ (\textbf{unfold}\ (x'.\textbf{val}).\mathsf{fields}.l_c))\\
\mathit{Call}[c]\ (t_{\mathsf{own}},\ o : \mathit{ObjGen}[c]\ \ t_{\mathsf{own}}, m) =\\
\quad \mathit{UnpackObj}[c]\ (t_{\mathsf{own}},\ o,\ \textbf{unfold}\ (x'.\textbf{val}).\mathsf{vtab}.l_m\ \ (x'.\textbf{val}))\\
\mathit{Call}[i]\ (_,\ io : \mathit{ExtView}[i], m) =\\
\quad \mathit{UnpackView}[i]\ (io,\ x.\textbf{val}.\mathsf{itab}.l_m\ \ (x.\textbf{val}.\mathsf{cobj}))\\
\mathit{Cast}[c,\ c']\ (t_{\mathsf{own}},\ t'_{\mathsf{own}}\ o : \mathit{ObjGen}[c]\ \ t_{\mathsf{own}}) =\\
\quad \mathit{UnpackObj}[c]\ (t_{\mathsf{own}},\ o,\\
\qquad \mathit{PackObj}[c']\ (t'_{\mathsf{own}},\ (l_c : t_{\mathsf{own}};\ x.\textbf{Typ}),\\
\qquad\qquad\qquad \mathit{NewPublic}[c]\ \ \mathit{ExtObj}[c]\ \ x'.\textbf{Typ},\ x'.\textbf{val}))\\
\quad \text{if}\ c' = \mathit{Super}[c]\\
\mathit{Cast}[c, i]\ (t_{\mathsf{own}},\ _,\ o : \mathit{ObjGen}[c]\ \ t_{\mathsf{own}}) =\\
\quad \mathit{UnpackObj}[c]\ (t_{\mathsf{own}},\ o,\\
\qquad \mathit{PackView}[i]\ (\mathit{SelfGen}[c]\ \ t_{\mathsf{own}}\ \ x.\textbf{Typ}\ \ x'.\textbf{Typ},\ x'.\textbf{val},\\
\qquad\qquad\qquad \textbf{unfold}\ (x'.\textbf{val}).\mathsf{vtab}.l_i))\\
\mathit{Cast}[i, i']\ (_,\ _,\ io : \mathit{ExtView}[i]) =\\
\quad \mathit{UnpackView}[i]\ (io,\\
\qquad \mathit{PackView}[i']\ (x.\textbf{Typ},\ x.\textbf{val}.\mathsf{cobj},\ x.\textbf{val}.\mathsf{itab}.l_{i'}))\\
\mathit{Prolog}[c]\ (t_{\mathsf{subFlds}}, t_{\mathsf{subVtab}},\ (x_j : \tau_j)^*,\ \bullet) =\\
\quad \lambda\mathsf{self} : \mathit{SelfGen}[c]\ \ \mathit{OwnFlds}[c]\ \ t_{\mathsf{subFlds}}\ \ t_{\mathsf{subVtab}}.\ (\lambda x_j : \tau_j.\ )^*\\
\qquad \textbf{let}\ \mathsf{this} = \mathit{PackObj}[c]\ (\mathit{OwnFlds}[c],\ t_{\mathsf{subFlds}},\ t_{\mathsf{subVtab}},\ \mathsf{self})\\
\qquad \textbf{in}\ \bullet\\
\mathit{CDecl}[c]\ (\mathit{classBody} : \mathit{ClassGen}[c]\ \ \mathit{OwnFlds}[c],\ \bullet) =\\
\quad \textbf{let}\ x_c = \langle t_{\mathsf{own}} = \mathit{OwnFlds}[c], \mathit{classBody} : \mathit{ClassGen}[c]\ \ t_{\mathsf{own}}\rangle\\
\quad \textbf{in}\ \bullet\\
\mathit{IDecl}[i]\ (\bullet) =\\
\quad \textbf{let}\ x_i = \langle t_{\mathsf{stamp}} = \textbf{Abs}^{\mathit{MLabels}[i]},\\
\qquad\qquad \Lambda t_{\mathsf{cobj}}.\ \lambda it : \mathit{ItabGen}[i]\ \ \textbf{Abs}^{\mathit{MLabels}[i]}\ \ t_{\mathsf{cobj}}.\ it\\
\qquad\qquad : \forall t_{\mathsf{cobj}}.\ \mathit{ItabGen}[i]\ \ \textbf{Abs}^{\mathit{MLabels}[i]}\ \ t_{\mathsf{cobj}}\\
\qquad\qquad\qquad \to \mathit{ItabGen}[i]\ \ t_{\mathsf{stamp}}\ \ t_{\mathsf{cobj}}\rangle\\
\quad \textbf{in}\ \bullet
\end{array}$$

Figure 11: Term macros used in the translation.

By convention, the type variables used in the translation have the following kinds and intended meaning
(in the context of a template with parameter c or i):

$t_{\mathsf{self}} :: \Omega$	– type of receiver object itself	$t_{\mathsf{twin}} :: \Omega$	– type of an argument/result of receiver's class
$t_{\mathsf{own}} :: \Omega$	– type of private fields of object	$t_{\mathsf{subFlds}} :: R^{SCLabels[c]}$	– type of row of field records of a subclass of c
$t_{\mathsf{subVtab}} :: \Omega \to R^{MLabels[c]}$	– type of row of new methods/interfaces of a subclass of c, parameterized by type of self		
$t_{\mathsf{cobj}} :: \Omega$	– type of object in an interface view	$t_{\mathsf{stamp}} :: R^{MLabels[i]}$	– type of interface's "hidden methods"

Given environments $\mathcal{H}$ and $\mathcal{E}$,

$$
\begin{aligned}
Super[c] &= c' \text{ s.t. } c \prec^{\mathsf{c}} c' \\
SCLabels[c] &= \{l_{c'} \mid c \leq^{\mathsf{c}}_{\mathcal{H}} c'\} \\
MLabels[c] &= \{l_m \mid \langle m, \_\rangle \in_{\mathcal{H}\mathcal{E}} c\} \cup \{l_i \mid c \ll^{\mathsf{c}}_{\mathcal{H}} i\} \\
MLabels[i] &= \{l_m \mid \langle m, \_\rangle \in_{\mathcal{H}\mathcal{E}} i\} \cup \{l_{i'} \mid i \leq^{\mathsf{i}}_{\mathcal{H}} i'\} \\
NewPublic[c] &= \lambda t_{\mathsf{twin}}.\, \lambda t_{\mathsf{subVtab}}.\, \lambda t_{\mathsf{self}}.\, ((l_i : ExtItab[i]\ t_{\mathsf{self}};)^* (l_m : t_{\mathsf{self}} \to [\![T]\!]_c\ t_{\mathsf{twin}};)^*\ (t_{\mathsf{subVtab}}\ t_{\mathsf{self}})) \\
&\quad \text{where } i \leftarrow \pi_2(\mathcal{H}(c)),\ \langle m, T\rangle \leftarrow \pi_2(\mathcal{E}(c))
\end{aligned}
$$

$$
\begin{aligned}
ObjTmpl[c] &= \lambda t_{\mathsf{own}}.\, \lambda t_{\mathsf{twin}}.\, \lambda t_{\mathsf{subFlds}}.\, \lambda t_{\mathsf{subVtab}}.\, \mu t_{\mathsf{self}}.\, \{\mathsf{vtab} : VtabGen[c]\ t_{\mathsf{twin}}\ t_{\mathsf{subVtab}}\ t_{\mathsf{self}}; \\
&\qquad \mathsf{fields} : AllFldsGen[c]\ t_{\mathsf{own}}\ t_{\mathsf{subFlds}}\} \\
ObjGen[c] &= \lambda t_{\mathsf{own}}.\, \mu t_{\mathsf{twin}}.\, \exists t_{\mathsf{subFlds}}.\, \exists t_{\mathsf{subVtab}}.\, ObjTmpl[c]\ t_{\mathsf{own}}\ t_{\mathsf{twin}}\ t_{\mathsf{subFlds}}\ t_{\mathsf{subVtab}} \\
ExtObj[c] &= ObjGen[c]\ (x_c.\mathbf{Typ}) \\
SelfGen[c] &= \lambda t_{\mathsf{own}}.\, \lambda t_{\mathsf{subFlds}}.\, \lambda t_{\mathsf{subVtab}}.\, ObjTmpl[c]\ t_{\mathsf{own}}\ (ObjGen[c]\ t_{\mathsf{own}})\ t_{\mathsf{subFlds}}\ t_{\mathsf{subVtab}}
\end{aligned}
$$

$$
\begin{aligned}
OwnFldsGen[c] &= \lambda t_{\mathsf{twin}}.\, \{(l_f : \mathbf{Ref}\ ([\![T]\!]_c\ t_{\mathsf{twin}}))^*\} \\
&\quad \text{where } \langle f, T\rangle \leftarrow \pi_1(\mathcal{E}(c)) \\
OwnFlds[c] &= \mu t_{\mathsf{own}}.\, OwnFldsGen[c]\ (ObjGen[c]\ t_{\mathsf{own}}) \\
AllFldsGen[c] &= \lambda t_{\mathsf{own}}.\, \lambda t_{\mathsf{subFlds}}.\, AllFlds[Super[c]]\ (l_c : t_{\mathsf{own}}; t_{\mathsf{subFlds}}) \\
AllFlds[c] &= \lambda t_{\mathsf{subFlds}}.\, AllFldsGen[c]\ (x_c.\mathbf{Typ})\ t_{\mathsf{subFlds}}
\end{aligned}
$$

$$
\begin{aligned}
VtabGen[c] &= \lambda t_{\mathsf{twin}}.\, \lambda t_{\mathsf{subVtab}}.\, \lambda t_{\mathsf{self}}.\, ExtVtab[Super[c]]\ (NewPublic[c]\ t_{\mathsf{twin}}\ t_{\mathsf{subVtab}})\ t_{\mathsf{self}} \\
ExtVtab[c] &= \lambda t_{\mathsf{subVtab}}.\, \lambda t_{\mathsf{self}}.\, VtabGen[c]\ ExtObj[c]\ t_{\mathsf{subVtab}}\ t_{\mathsf{self}} \\
&\quad \text{and } ExtVtab[\text{Object}] = \lambda t_{\mathsf{subVtab}}.\, \lambda t_{\mathsf{self}}.\, \{t_{\mathsf{subVtab}}\ t_{\mathsf{self}}\} \\
DictGen[c] &= \lambda t_{\mathsf{own}}.\, \lambda t_{\mathsf{subFlds}}.\, \lambda t_{\mathsf{subVtab}}. \\
&\qquad VtabGen[c]\ (ObjGen[c]\ t_{\mathsf{own}})\ (\lambda t_{\mathsf{self}}.\, \mathbf{Abs}^{MLabels[c]})\ (SelfGen[c]\ t_{\mathsf{own}}\ t_{\mathsf{subFlds}}\ t_{\mathsf{subVtab}}) \\
ClassGen[c] &= \lambda t_{\mathsf{own}}.\, \{\mathsf{dict} : \forall t_{\mathsf{subFlds}}.\, \forall t_{\mathsf{subVtab}}.\, DictGen[c]\ t_{\mathsf{own}}\ t_{\mathsf{subFlds}}\ t_{\mathsf{subVtab}}; \\
&\qquad \mathsf{init} : \mathbf{1} \to OwnFldsGen[c]\ (ObjGen[c]\ t_{\mathsf{own}}); \\
&\qquad \mathsf{new} : \mathbf{1} \to ObjGen[c]\ t_{\mathsf{own}}\}
\end{aligned}
$$

$$
\begin{aligned}
ItabTmpl[i] &= \lambda t_{\mathsf{stamp}}.\, \lambda t_{\mathsf{twin}}.\, \lambda t_{\mathsf{cobj}}.\, \{(l_{i'} : ExtItab[i']\ t_{\mathsf{cobj}};)^* (l_m : t_{\mathsf{cobj}} \to [\![T]\!]_i\ t_{\mathsf{twin}};)^* t_{\mathsf{stamp}}\} \\
&\quad \text{where } i' \leftarrow \mathcal{H}(i),\ \langle m, T\rangle \leftarrow \pi_2(\mathcal{E}(i)) \\
ViewTmpl[i] &= \lambda t_{\mathsf{stamp}}.\, \lambda t_{\mathsf{twin}}.\, \lambda t_{\mathsf{cobj}}.\, \{\mathsf{cobj} : t_{\mathsf{cobj}}; \mathsf{itab} : ItabTmpl[i]\ t_{\mathsf{stamp}}\ t_{\mathsf{twin}}\ t_{\mathsf{cobj}}\} \\
ViewGen[i] &= \lambda t_{\mathsf{stamp}}.\, \mu t_{\mathsf{twin}}.\, \exists t_{\mathsf{cobj}}.\, ViewTmpl[i]\ t_{\mathsf{stamp}}\ t_{\mathsf{twin}}\ t_{\mathsf{cobj}} \\
ItabGen[i] &= \lambda t_{\mathsf{stamp}}.\, \lambda t_{\mathsf{cobj}}.\, ItabTmpl[i]\ t_{\mathsf{stamp}}\ (ViewGen[i]\ t_{\mathsf{stamp}})\ t_{\mathsf{cobj}}
\end{aligned}
$$

$$
\begin{aligned}
ExtView[i] &= ViewGen[i]\ (x_i.\mathbf{Typ}) \\
ExtItab[i] &= \lambda t_{\mathsf{cobj}}.\, ItabGen[i]\ (x_i.\mathbf{Typ})\ t_{\mathsf{cobj}}
\end{aligned}
$$

$$
[\![ty_1 \ldots ty_n \to ty]\!]_{ty'} = \lambda t_{\mathsf{twin}} :: \Omega.\ [\![ty_1]\!]_{ty'}\ t_{\mathsf{twin}} \to \ldots [\![ty_n]\!]_{ty'}\ t_{\mathsf{twin}} \to [\![ty]\!]_{ty'}\ t_{\mathsf{twin}}
$$

$$
[\![ty]\!]_{ty'} = \lambda t_{\mathsf{twin}} :: \Omega.\ \begin{cases} t_{\mathsf{twin}}, & \text{if } ty = ty' \\ ExtObj[c] & \text{otherwise, if } ty = c \\ ExtView[i] & \text{otherwise, if } ty = i \end{cases}
$$

Figure 10: Type operators used in the translation.

(trans-var)

$$\frac{x \in \mathrm{dom}(\Gamma)}{\_;\_;\Gamma;\_ \vdash x \ :\ \Gamma(x) \leadsto x}$$

(trans-new)

$$\frac{\mathcal{H} \vdash_{\mathsf{t}} c'}{\mathcal{H};\_;\_;c \vdash \mathbf{new}\, c' \ :\ c' \leadsto \begin{cases}(x_c\ \{\}).\mathsf{new}\ \{\}, \text{ if } c' = c\\ x_{c'}.\mathbf{val}.\mathsf{new}\ \{\}, \text{ if } c' \neq c\end{cases}}$$

(trans-get)

$$\frac{\mathcal{H};\mathcal{E};\Gamma;c \vdash \mathit{exp} \ :\ \mathit{ty}' \leadsto e \qquad \mathit{ty}' \leq^{\mathsf{c}}_{\mathcal{H}} c \qquad \langle f, \mathit{ty}\rangle \in^{\mathsf{c}}_{\mathcal{E}} c}{\mathcal{H};\mathcal{E};\Gamma;c \vdash \underline{c \triangleright} \mathit{exp}.f \ :\ \mathit{ty} \leadsto\ !\,((\mathit{GetFields}[c]\ (e)).l_f)}$$

(trans-set)

$$\frac{\mathcal{H};\mathcal{E};\Gamma;c \vdash \mathit{exp}_1 \ :\ \mathit{ty}' \leadsto e_1 \qquad \mathit{ty}' \leq^{\mathsf{c}}_{\mathcal{H}} c \qquad \mathcal{H};\mathcal{E};\Gamma;c \vdash \mathit{exp}_2 \ :\ \mathit{ty} \leadsto e_2 \qquad \langle f, \mathit{ty}\rangle \in^{\mathsf{c}}_{\mathcal{E}} c}{\mathcal{H};\mathcal{E};\Gamma;c \vdash \underline{c \triangleright} \mathit{exp}_1.f = \mathit{exp}_2 \ :\ \mathit{ty} \leadsto (\mathit{GetFields}[c]\ (e_1)).l_f := e_2}$$

(trans-body)

$$\frac{\forall j \in \{1..n\}.\ \mathcal{H};\mathcal{E};\Gamma;c \vdash \mathit{exp}_j \ :\ \mathit{ty}_j \leadsto e_j}{\mathcal{H};\mathcal{E};\Gamma;c \vdash \{\, \mathit{exp}_1; \ldots ;\mathit{exp}_n\ \} \ :\ \mathit{ty}_n \leadsto (e_1;\ \ldots;e_n)}$$

(trans-call)

$$\frac{\mathcal{H};\mathcal{E};\Gamma;c \vdash \mathit{exp} \ :\ \mathit{ty}' \leadsto e \qquad \langle m, (\mathit{ty}_1 \ldots \mathit{ty}_n \to \mathit{ty})\rangle \in_{\mathcal{HE}} \mathit{ty}' \qquad \forall j \in \{1..n\}.\ \mathcal{H};\mathcal{E};\Gamma;c \vdash \mathit{exp}_j \ :\ \mathit{ty}_j \leadsto e_j}{\mathcal{H};\mathcal{E};\Gamma;c \vdash \mathit{exp}.m\,(\mathit{exp}_1 \ldots \mathit{exp}_n) \ :\ \mathit{ty} \leadsto \mathit{Call}[\mathit{ty}']\,(\mathit{Own}[c]\,(\mathit{ty}'),\ e,\ m)\ (e_j)^*}$$

(trans-cast)

$$\frac{\mathcal{H};\mathcal{E};\Gamma;c \vdash \mathit{exp} \ :\ \mathit{ty}' \leadsto e \qquad \mathit{ty}' \prec_{\mathcal{H}} \mathit{ty}}{\mathcal{H};\mathcal{E};\Gamma;c \vdash (\mathit{ty})\,\mathit{exp} \ :\ \mathit{ty} \leadsto \mathit{Cast}[\mathit{ty}', \mathit{ty}]\,(\mathit{Own}[c]\,(\mathit{ty}'),\ \mathit{Own}[c]\,(\mathit{ty}),\ e)}$$

where $\mathit{Own}[c]\ (\mathit{ty}) = \begin{cases}\mathit{OwnFlds}[c], & \text{if } \mathit{ty} = c\\ x_{\mathit{ty}}.\mathbf{Typ} & \text{otherwise}\end{cases}$

(trans-class)

$$\frac{\mathcal{H};\mathcal{E};\emptyset;c \vdash \mathit{exp}_j \ :\ \mathit{ty}_j \leadsto e_j \qquad \mathcal{H};\mathcal{E};\mathbf{this} : c,\ \mathit{ArgTypes}[c](\mathit{msig}_j);c \vdash \mathit{exp}'_j \ :\ \mathit{RsltType}(\mathit{msig}_j) \leadsto e'_j}{\begin{array}{l}\mathcal{H};\ \mathcal{E};\ c;\ c' \\ \vdash \{(\mathit{ty}_j\ f_j = \mathit{exp}_j;)^*\ (\mathit{msig}_j\ \mathit{exp}'_j)^*\} \\ \leadsto \lambda x_c : \mathbf{1} \to \mathit{ClassGen}[c]\ \ \mathit{OwnFlds}[c].\ \lambda\_ : \mathbf{1}. \\ \quad \mathbf{let}\ \mathsf{dict} = \Lambda t_{\mathsf{subFlds}}.\ \Lambda t_{\mathsf{subVtab}}. \\ \quad\quad \mathbf{let}\ \mathsf{super} = x_{c'}.\mathbf{val}.\mathsf{dict}\,[l_c : \mathit{OwnFlds}[c]; t_{\mathsf{subFlds}}] \\ \quad\quad\quad [\mathit{NewPublic}[c]\ \ (\mathit{ObjGen}[c]\ \ \mathit{OwnFlds}[c])\ t_{\mathsf{subVtab}}] \\ \quad\quad \mathbf{in\ let}\ \mathsf{newdict} = \\ \quad\quad\quad \{(l_{m_j} = \mathit{Prolog}[c]\ (t_{\mathsf{subFlds}},\ t_{\mathsf{subVtab}},\ \mathit{ArgTypes}[c](\mathit{msig}_j),\ e'_j))^*\} \\ \quad\quad \mathbf{in}\ \mathit{MkDict}[c]\ (\mathsf{super},\ \mathsf{newdict}) \\ \quad \mathbf{in\ let}\ \mathsf{init} = \lambda\_ : \mathbf{1}.\ \{(l_{f_j} = \mathbf{ref}\ e_j)^*\} \\ \quad \mathbf{in\ let}\ \mathsf{new} = \lambda\_ : \mathbf{1}. \\ \quad\quad \mathit{PackObj}[c]\ (\mathit{OwnFlds}[c],\ \mathbf{Abs}^{\mathit{SCLabels}[c]},\ \mathbf{Abs}^{\mathit{MLabels}[c]}, \\ \quad\quad\quad \mathbf{fold}\ \{\mathsf{vtab} = \mathsf{dict}\,[\mathbf{Abs}^{\mathit{SCLabels}[c]}]\,[\mathbf{Abs}^{\mathit{MLabels}[c]}], \\ \quad\quad\quad\quad \mathsf{fields} = \{(l_{c''} = x_{c''}.\mathbf{val}.\mathsf{init}\ \{\})^{c'' \leftarrow \mathit{SCLabels}[c']}, \\ \quad\quad\quad\quad\quad l_c = \mathsf{init}\ \{\}\}\} \\ \quad\quad\quad \mathbf{as}\ \mathit{SelfGen}[c]\ \ \mathit{OwnFlds}[c]\ \ t_{\mathsf{subFlds}}\ t_{\mathsf{subVtab}}) \\ \quad \mathbf{in}\ \{\mathsf{dict} = \mathsf{dict},\ \mathsf{init} = \mathsf{init},\ \mathsf{new} = \mathsf{new}\}\end{array}}$$

where $\mathit{ArgTypes}[c](\mathit{ty}\ m\ (\mathit{ty}_k\ x_k)^*) = (x_k : [\![\mathit{ty}_k]\!]_c)^*$
$\mathit{RsltType}(\mathit{ty}\ m\ (\mathit{ty}_k\ x^*_k)) = \mathit{ty}$

(trans-class-decl)

$$\frac{\begin{array}{l}\mathcal{H}';\mathcal{E}';\_ \vdash_{\mathsf{d}} \mathbf{class}\, c\ \mathbf{ext}\, c'\ \mathbf{imp}\, i^*\{(\mathit{ty}_j\ \ f_j = \mathit{exp}_j;)^*\ (\mathit{msig}_j\ \mathit{exp}'_j)^*\} \Rightarrow \mathcal{H};\mathcal{E};\_ \\ \mathcal{H};\ \mathcal{E};\ c;\ c' \vdash \{(\mathit{ty}_j\ f_j = \mathit{exp}_j;)^*\ (\mathit{msig}_j\ \mathit{exp}'_j)^*\} \leadsto e \\ \mathcal{H};\ \mathcal{E} \vdash p \leadsto \mathcal{H}'';\ \mathcal{E}'';\ e'\end{array}}{\mathcal{H}';\ \mathcal{E}' \vdash \mathbf{class}\, c\ \mathbf{ext}\, c'\ \mathbf{imp}\, i^*\{(\mathit{ty}_j\ \ f_j = \mathit{exp}_j;)^*\ (\mathit{msig}_j\ \mathit{exp}'_j)^*\}\ p \leadsto \mathcal{H}'';\ \mathcal{E}'';\ \mathit{CDecl}[c]\ (Y_v\ e\ \{\},\ e')}$$

(trans-iface-decl)

$$\frac{\begin{array}{l}\mathcal{H}';\mathcal{E}';\_ \vdash_{\mathsf{d}} \mathbf{interface}\, i\ \mathbf{ext}\, i_1 \ldots i_n\ \{\mathit{msig}_1; \ldots \mathit{msig}_m;\} \Rightarrow \mathcal{H};\mathcal{E};\_ \\ \mathcal{H};\ \mathcal{E} \vdash p \leadsto \mathcal{H}'';\ \mathcal{E}'';\ e\end{array}}{\mathcal{H}';\ \mathcal{E}' \vdash \mathbf{interface}\, i\ \mathbf{ext}\, i_1 \ldots i_n\ \{\mathit{msig}_1; \ldots \mathit{msig}_m;\}\ p \leadsto \mathcal{H}'';\ \mathcal{E}'';\ \mathit{IDecl}[i]\ (e)}$$

(trans-super)

$$\frac{c \prec^{\mathsf{c}}_{\mathcal{H}} c' \qquad \langle m, (\mathit{ty}_1 \ldots \mathit{ty}_n \to \mathit{ty})\rangle \in_{\mathcal{HE}} c' \qquad \forall j \in \{1..n\}.\ \mathcal{H};\mathcal{E};\Gamma;c \vdash \mathit{exp}_j \ :\ \mathit{ty}_j \leadsto e_j}{\mathcal{H};\mathcal{E};\Gamma;c \vdash \underline{c \cdot \mathbf{this} \triangleright} \mathbf{super}.m\,(\mathit{exp}_1 \ldots \mathit{exp}_n) \ :\ \mathit{ty} \leadsto \mathit{UnpackObj}[c]\ (\mathsf{this},\ \mathsf{super}.l_m\ (x'.\mathbf{val})\ (e_j)^*)}$$

Figure 12: Translation of *Javacito* to MINIFLINT.

$\lambda\_ : \mathbf{1}.\ \{\mathsf{x} = \mathbf{ref}\ 0\}$. To create the segregated record containing the fields of each superclass, we simply call the init functions for each superclass. Suppose the following code is inside the SPt package:

$$\begin{array}{l}\mathbf{let}\ \mathsf{init} = \lambda\_ : \mathbf{1}.\ \{\mathsf{scale} = \mathbf{ref}\ 1\} \\ \ \mathbf{in} \ldots \{\mathsf{Pt} = \mathsf{Pt}.\mathbf{val}.\mathsf{init}\ \{\}, \mathsf{SPt} = \mathsf{init}\ \{\}\} \ldots\end{array}$$

Then the type of the record will be $\mathit{AllFldsGen}[\mathsf{SPt}]$ applied to $\mathit{OwnFlds}[\mathsf{SPt}]$ and the empty row $\mathbf{Abs}^L$, where the label set L is $\{\mathsf{Pt}, \mathsf{SPt}\}$.

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. $\mathit{VtabGen}[c]$ produces the type of a vtable for class c. Here is the parameterized type for the vtable of Pt:

$$\begin{array}{rl}\mathit{VtabGen}[\mathrm{Pt}] = \lambda t_{\mathsf{twin}}.\ \lambda t_{\mathsf{subVtab}}.\ \lambda t_{\mathsf{self}}.\ \{ & \\ & \mathsf{max} : t_{\mathsf{self}} \to t_{\mathsf{twin}} \to t_{\mathsf{twin}}; \\ & \mathsf{move} : t_{\mathsf{self}} \to \mathbf{Int} \to \mathbf{1}; \\ & \mathsf{bump} : t_{\mathsf{self}} \to \mathbf{1}; \\ & t_{\mathsf{subVtab}}\ \ t_{\mathsf{self}}\}\end{array}$$

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

The tail of the vtable is produced by applying the t_{subVtab} operator to t_{self}. 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 interface for now):

$$
\begin{aligned}
VtabGen\,[\text{SPt}] = \lambda t_{\mathsf{twin}}.\,\lambda t_{\mathsf{subVtab}}.\,\lambda t_{\mathsf{self}}.\\
VtabGen\,[\text{Pt}]\ ExtObj\,[\text{Pt}]\\
(\lambda t'_{\mathsf{self}}.\,(\mathsf{zoom} : t'_{\mathsf{self}} \to \mathbf{Int} \to \mathbf{1};\\
t_{\mathsf{subVtab}}\ t'_{\mathsf{self}}))\ t_{\mathsf{self}}
\end{aligned}
$$

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:

$$
\begin{aligned}
VtabGen\,[\text{SPt}] = \lambda t_{\mathsf{twin}}.\,\lambda t_{\mathsf{subVtab}}.\,\lambda t_{\mathsf{self}}.\,\{\\
\mathsf{max} : t_{\mathsf{self}} \to ExtObj\,[\text{Pt}] \to ExtObj\,[\text{Pt}];\\
\mathsf{move} : t_{\mathsf{self}} \to \mathbf{Int} \to \mathbf{1};\\
\mathsf{bump} : t_{\mathsf{self}} \to \mathbf{1};\\
\mathsf{zoom} : t_{\mathsf{self}} \to \mathbf{Int} \to \mathbf{1};\\
t_{\mathsf{subVtab}}\ t_{\mathsf{self}}\}
\end{aligned}
$$

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 $ObjTmpl[c]$, a parameterized template for generating class object types (see Figure 10). Note the fixed point for resolving t_{self}; 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:

$$
\begin{aligned}
&\mathsf{dict} = \Lambda t_{\mathsf{subFlds}}.\,\Lambda t_{\mathsf{subVtab}}.\\
&\{\mathsf{max} = \lambda \mathsf{self} : (SelfGen\,[\text{Pt}]\ \ OwnFlds\,[\text{Pt}]\ t_{\mathsf{subFlds}}\ t_{\mathsf{subVtab}}).\\
&\qquad \lambda \mathsf{other} : (ObjGen\,[\text{Pt}]\ \ OwnFlds\,[\text{Pt}]).\,\ldots,\\
&\ \ldots\}
\end{aligned}
$$

The difference between $SelfGen\,[c]$ and $ObjGen\,[c]$ is that the self type should also be extensible; this is why it takes t_{subFlds} and t_{subVtab} as parameters. $ObjGen\,[c]$, on the other hand, is an object generated by some unknown class. It might have some additional fields and methods but we cannot know what they are. The $ObjGen\,[c]$ operator closes the object type by existentially quantifying the two tail variables and taking a fixed point to resolve t_{twin} (see Figure 10).

The only parameter of $ObjGen\,[c]$ is t_{own}, the type of the private fields in c. Inside the class's dictionary, this is instantiated to $OwnFlds\,[c]$ so that the private fields are accessible. Outside the class, t_{own} is instantiated instead to x_c.**Typ**. Thus, the type of a class object viewed from the outside is $ExtObj\,[c] = ObjGen\,[c]\ \ (x_c.\mathbf{Typ})$.

Finally, $SelfGen\,[c]$ takes the template and instantiates t_{twin} to $ObjGen\,[c]\ \ t_{\mathsf{own}}$, but leaves the tails as parameters. This ensures that the private fields of self will match those of other arguments of the same class.

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 $x.\mathbf{Typ}$ and $x'.\mathbf{Typ}$ 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 $x_c.\mathbf{val}.\mathsf{init}$ 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 *Javacito* 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:

$$
\begin{aligned}
&\mathsf{dict} = \Lambda t_{\mathsf{subFlds}}.\Lambda t_{\mathsf{subVtab}}.\\
&\qquad \{\mathsf{max} = \lambda \mathsf{self}.\ldots,\ \mathsf{move} = \lambda \mathsf{self}.\ldots,\ \mathsf{bump} = \lambda \mathsf{self}.\ldots\}
\end{aligned}
$$

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. To get an object of class Pt we then pair vtab with the fields record (see section 5.1.3), and fold to fix the type of self:

$$
\begin{aligned}
\mathbf{let}\ p_0 = \mathbf{fold}\ \{&\mathsf{vtab} = \mathsf{dict}\,[\mathbf{Abs}^{L_c}]\,[\mathbf{Abs}^{L_m}],\\
&\mathsf{fields} = \{\mathsf{Pt} = \mathsf{init}\ \{\}\}\}\\
&\mathbf{as}\ SelfGen\,[\text{Pt}]\ \ OwnFlds\,[\text{Pt}]\ \ \mathbf{Abs}^{L_c}\ \ \mathbf{Abs}^{L_m}
\end{aligned}
$$

where $L_c = \{\mathsf{Pt}\}$ and $L_m = \{\mathsf{max}, \mathsf{move}, \mathsf{bump}\}$.

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:

$$
\begin{aligned}
\mathbf{let}\ p_1 = \mathbf{fold}\ \langle &t_{\mathsf{subFlds}} = \mathbf{Abs}^{L_c},\\
&\langle t_{\mathsf{subVtab}} = \mathbf{Abs}^{L_m},\\
&\ \ p_0 : SelfGen\,[\text{Pt}]\ \ t_{\mathsf{own}}\ \ \mathbf{Abs}^{L_c}\ \ t_{\mathsf{subVtab}}\rangle\\
&: \exists t_{\mathsf{subVtab}}.SelfGen\,[\text{Pt}]\ \ t_{\mathsf{own}}\ \ t_{\mathsf{subFlds}}\ \ t_{\mathsf{subVtab}}\rangle\\
&\mathbf{as}\ ObjGen\,[\text{Pt}]\ \ t_{\mathsf{own}}
\end{aligned}
$$

where, as usual, t_{own} is $OwnFlds\,[\text{Pt}]$ if this expression is inside the Pt class package, or Pt.**Typ** otherwise. The final **fold** resolves the twin type, as in the non-self argument of 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 $PackObj\,[c]$ in Figure 11. It is used not only for object creation but for various forms of casting. Its inverse is $UnpackObj\,[c]$.

To implement the new function in the class package, we need to wrap a λ around the two **let**-bindings above:

$$
\mathbf{let}\ \mathsf{new} = \lambda\_ : \mathbf{1}.\,\mathbf{let}\ p_0 = \ldots\ \ \mathbf{in}\ \mathbf{let}\ p_1 = \ldots\ \ \mathbf{in}\ p_1
$$

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 Y_v:

$$\begin{array}{l} \textbf{let } x_c = \langle t_{\mathsf{own}}{::}\Omega = \mathit{OwnFlds}[c], \\ \quad\quad Y_v\,[\mathit{ClassGen}[c]\ \ \mathit{OwnFlds}[c]] \\ \quad\quad\quad (\lambda x_c : \mathbf{1} \to \mathit{ClassGen}[c]\ \ \mathit{OwnFlds}[c]. \\ \quad\quad\quad\quad \lambda\_ : \mathbf{1}.\ \{\mathsf{dict} = \ldots, \mathsf{init} = \ldots, \mathsf{new} = \ldots\}) \\ \quad\quad\quad \{\}\rangle \\ \ \textbf{in} \ldots \end{array}$$

where $\mathit{ClassGen}[c]$ 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 $(x_c\ \{\}).\mathsf{new}\ \{\}$. 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:

$$\textbf{let this} = \mathit{PackObj}[c]\ (\mathit{OwnFlds}[c], t_{\mathsf{subFlds}}, t_{\mathsf{subVtab}}, \mathsf{self})\ \textbf{in}\ \ldots$$

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:

$$\begin{array}{l} \textbf{let } x = \textbf{unfold}\, o \textbf{ in let } x' = x.\textbf{val} \\ \ \textbf{in unfold}\,(x'.\textbf{val}).\mathsf{vtab}.l_m\ (x'.\textbf{val}) \end{array}$$

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, $x'.\textbf{val}.\mathsf{vtab}.l_m$ will have a type which includes the abstract types $x.\textbf{Typ}$ for the additional fields in o and $x'.\textbf{Typ}$ for the additional methods in o. Since the only object which can have these types is $x'.\textbf{val}$ 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 ExtObj [SPt]:

$$\begin{array}{l} \textbf{let } x = \textbf{unfold}\, o \\ \ \textbf{in let } x' = x.\textbf{val} \\ \quad\ \textbf{in } \mathit{PackObj}[\mathrm{Pt}]\ (\mathsf{Pt}.\textbf{Typ}, \\ \quad\quad\quad\quad (\mathsf{SPt} : \mathsf{SPt}.\textbf{Typ}; x.\textbf{Typ}), \\ \quad\quad\quad\quad (\lambda t_{\mathsf{self}}.\ (\mathsf{zoom} : t_{\mathsf{self}} \to \textbf{Int} \to \mathbf{1}; \\ \quad\quad\quad\quad\quad\quad x'.\textbf{Typ}\ \ t_{\mathsf{self}})), \\ \quad\quad\quad\quad x'.\textbf{val}) \end{array}$$

The result is a value with type ExtObj [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 $\mathit{Cast}[c,\ c']$ 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 SPt dict, we bind super as follows:

$$\begin{array}{l} \mathsf{dict} = \Lambda t_{\mathsf{subFlds}}.\Lambda t_{\mathsf{subVtab}}. \\ \quad \textbf{let } \mathsf{super} = \mathsf{Pt}.\textbf{val}.\mathsf{dict}\,[\mathsf{SPt} : \mathit{OwnFlds}[\mathrm{SPt}]; t_{\mathsf{subFlds}}] \\ \quad\quad\quad\quad\quad\quad\quad\quad [\lambda t_{\mathsf{self}}.\ (\mathsf{zoom} : t_{\mathsf{self}} \to \textbf{Int} \to \mathbf{1}; \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad t_{\mathsf{subVtab}}\ \ t_{\mathsf{self}})] \\ \quad\ \textbf{in } \{\mathsf{max} = \mathsf{super.max}, \\ \quad\quad\quad \mathsf{move} = \lambda\mathsf{self}.\ldots\mathsf{super.move\ self}\ldots, \\ \quad\quad\quad \mathsf{bump} = \mathsf{super.bump}, \\ \quad\quad\quad \mathsf{zoom} = \lambda\mathsf{self}.\ldots\} \end{array}$$

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 itable 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,

$$\mathit{ItabGen}_s[\mathrm{Zm}] = \lambda t_{\mathsf{twin}}.\ \lambda t_{\mathsf{cobj}}.\ \{\mathsf{zoom} : t_{\mathsf{cobj}} \to \textbf{Int} \to \mathbf{1}\}$$

where t_{cobj} is the type of the underlying class object (to simplify the presentation, we have omitted the t_{stamp} parameter from $\mathit{ItabGen}$; 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 (cobj) and the appropriate itable (itab):

$$\begin{array}{l} \mathit{ViewTmpl}_s[\mathrm{Zm}] = \lambda t_{\mathsf{twin}}.\ \lambda t_{\mathsf{cobj}}. \\ \quad\quad \{\mathsf{cobj} : t_{\mathsf{cobj}}; \mathsf{itab} : \mathit{ItabGen}_s[\mathrm{Zm}]\ \ t_{\mathsf{twin}}\ \ t_{\mathsf{cobj}}\} \end{array}$$

We resolve the t_{cobj} parameter using an existential type. We should be able to select a method from the itable and pass cobj as its self parameter. We do not know (or it should not matter) what t_{cobj} actually is:

$$View_s[\mathrm{Zm}] = \mu t_{\mathsf{twin}}.\ \exists t_{\mathsf{cobj}}.\ ViewTmpl_s[\mathrm{Zm}]\ t_{\mathsf{twin}}\ t_{\mathsf{cobj}}$$

Finally, we resolve t_{twin} 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_{cobj} type.

5.5.2 Invokeinterface

To invoke method m on an interface object io, we bind x to **unfold** io, apply x.**val**.itab.l_m to x.**val**.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 itab or cobj. The function x.**val**.itab.l_m wants an argument of type x.**Typ**. The only possible expression of that type is x.**val**.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 vtable 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 SPt dict would contain the itable for Zm:

$$
\begin{array}{l}
\mathsf{dict} = \Lambda t_{\mathsf{subFlds}}.\Lambda t_{\mathsf{subVtab}}. \\
\quad \mathbf{let}\ \mathsf{super} = \mathsf{Pt}.\mathbf{val}.\mathsf{dict}\,[\ldots]\,[\ldots] \\
\quad\ \mathbf{in\ let}\ \mathsf{newdict} = \\
\qquad\qquad \{\mathsf{move} = \lambda\mathsf{self}.\ldots\mathsf{super.move\ self}\ldots, \\
\qquad\qquad\ \ \mathsf{zoom} = \lambda\mathsf{self}.\ldots\} \\
\qquad \mathbf{in}\ \{\mathsf{max} \ = \mathsf{super.max}, \\
\qquad\quad\ \ \mathsf{move} = \mathsf{newdict.move}, \\
\qquad\quad\ \ \mathsf{bump} = \mathsf{super.bump}, \\
\Rightarrow \qquad\quad\ \ \mathsf{Zm} \ \ = \{\mathsf{zoom} = \mathsf{newdict.zoom}\}, \\
\qquad\quad\ \ \mathsf{zoom} = \mathsf{newdict.zoom}\}
\end{array}
$$

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

$$
\begin{array}{l}
\mathbf{let}\ x = \mathbf{unfold}\ o \\
\ \mathbf{in\ let}\ x' = x.\mathbf{val} \\
\quad \mathbf{in\ fold}\ \langle t_{\mathsf{cobj}}{::}\Omega = SelfGen[\mathrm{SPt}]\ \mathsf{SPt}.\mathbf{Typ}\ x.\mathbf{Typ}\ x'.\mathbf{Typ}, \\
\qquad\qquad \{\mathsf{cobj} = x'.\mathbf{val}, \\
\qquad\qquad\ \ \mathsf{itab} = \mathbf{unfold}\ (x'.\mathbf{val}).\mathsf{vtab.Zm}\} \\
\qquad\qquad : ViewTmpl_s[\mathrm{Zm}]\ \ View_s[\mathrm{Zm}]\ \ t_{\mathsf{cobj}}\rangle \\
\qquad \mathbf{as}\ View_s[\mathrm{Zm}]
\end{array}
$$

For this example, we have assumed that this code is contained in some class *other* than SPt. If we were inside SPt, then SPt.**Typ** above would be replaced with $OwnFlds$[SPt]. 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 vtable 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_i.**Typ**); the type of an itable for that interface will contain x_i.**Typ** as a component. The body of the package (x_i.**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 $MkDict[c]$, constructing a class dictionary out of a superclass dictionary super, specialized to its subclass c, and a record newdict of the newly defined methods of c. The function of $MkDict[c]$ is to collect selected components of super (inherited) or 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* $\emptyset; \emptyset; \emptyset \vdash_P p : ty \Rightarrow \mathcal{H}; \mathcal{R}$ *then* $\emptyset;\ \emptyset \vdash p \rightsquigarrow \mathcal{H};\ \mathcal{R};\ e$ *and* $\emptyset \vdash e : [\![ty]\!]_c\ \{\}$*, where* $c \notin dom(\mathcal{H})$.

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:

$$
\begin{array}{lcll}
\textit{types} & \tau & ::= & \ldots \mid \{\!\{\tau\}\!\} \\
\textit{terms} & e & ::= & \ldots \mid \{\!\{(l = e)^*\}\!\}
\end{array}
$$

Here, $\{\!\{\tau\}\!\}$ denotes the unordered record type where τ must be of kind $R^{\emptyset}$, and $\{\!\{(l = e)^*\}\!\}$ 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 vtable into a separate unordered record, itself an element of the still ordered vtable. 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

Javacito 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):

$$\textbf{exception } Tag_c \textbf{ of } ObjGen[c]\ t_{\mathsf{own}};$$
$$or \qquad \textbf{exception } Tag_i \textbf{ of } ViewGen[i]\ t_{\mathsf{stamp}};$$

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 $Tag_{c'}$ (or $Tag_{i'}$) 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 **instanceof** 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 Tag_c (or Tag_i). The **checkcast** operation can be implemented in the same way since the object associated with Tag_c (or Tag_i) is already of type c (or i). Notice unlike Glew's recent work [19], our technique does not use hierachical 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 over-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^{ω} 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

[1] M. Abadi, L. Cardelli, and R. Viswanathan. An interpretation of objects and object types. In *Proc. 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'96)*, pages 396–409, St. Petersburg, January 1996.

[2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[3] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proc. 1998 Int'l Conf. on Functional Programming (ICFP'98)*, pages 129–140, September 1998.

[4] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, December 1995.

[5] K. Bruce, L. Cardelli, and B. Pierce. Comparing object encodings. *Information and Computation (to appear)*, 1998.

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

[7] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[8] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proc. IFIP Working Conf. on Programming Concepts and Methods*, pages 466–491, Israel, April 1990.

[9] L. Cardelli and G. Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4), 1991.

[10] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proc. ACM SIGPLAN '96 Conf. on Object-Oriented Programming Systems, Languages, and applications*, pages 83–100, October 1996.

[11] S. Drossopoulou and S. Eisenbach. Java is type safe—probably. In *Proc. 11th European Conf. on Object-Oriented Prog.*, February 1997.

[12] S. Drossopoulou and S. Eisenbach. Towards an operational semantics and proof of type soundness for Java, April 1998.

[13] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation*, 8(4):357–397, 1995.

[14] K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, August 1996.

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

[16] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *Proc. ACM SIGPLAN '99 Conf. on Prog. Lang. Design and Implementation*, pages 37–49, May 1999.

[17] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report 97-293, Rice University, 1997.

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

[19] N. Glew. Type dispatch for named hierarchical types. In *Proc. 1999 ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, September 1999.

[20] N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Twenty-fifth Annual ACM Symp. on Principles of Prog. Languages*, pages 365–377, Jan 1998.

[21] M. Hofmann and B. Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, January 1994.

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

[23] C. League, Z. Shao, and V. Trifonov. Representing Java classes in a typed intermediate language (extended version). Technical Report YALEU/DCS/RR-1180, Department of Computer Science, Yale University, New Haven, CT, June 1999.

[24] X. Leroy and F. Rouaix. Security properties of typed applets. In *Twenty-fifth Annual ACM Symp. on Principles of Prog. Languages*, pages 391–403, Jan 1998.

[25] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition)*. Addison-Wesley, 1999.

[26] Microsoft Win32 VM for Java object model, 1998.

[27] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Twenty-fourth Annual ACM Symp. on Principles of Prog. Languages*, pages 146–159, Jan 1997.

[28] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.

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

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

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

[32] D. Remy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Twenty-fourth Annual ACM Symp. on Principles of Prog. Languages*, pages 40–53, Jan 1997.

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

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

[35] Z. Shao. Typed common intermediate format. In *Proc. USENIX Conf. on Domain-Specific Languages*, pages 89–101, Santa Barbara, CA, October 1997.

[36] Z. Shao. Typed cross-module compilation. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 141–152, September 1998.

[37] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 313–323, September 1998.

[38] A. Wright, S. Jaganathan, C. Ungureanu, and A. Hertzmann. Compiling Java to a typed lambda-calculus: A preliminary report. In X. Leroy and A. Ohori, editors, *Proc. 1998 International Workshop on Types in Compilation: LNCS Vol 1473*, pages 9–27, Kyoto, Japan, March 1998. Springer-Verlag.