Compiling Java to a Typed Lambda-Calculus: A Preliminary Report

Andrew Wright* Suresh Jagannathan† Cristian Ungureanu† Aaron Hertzmann‡

1 Introduction

A typical compiler for Java translates source code into machine-independent byte code. The byte code may be either interpreted by a Java Virtual Machine, or further compiled to native code by a just-in-time compiler. The byte code architecture provides platform independence at the cost of execution speed. When Java is used as a tool for writing applets—small ultra-portable programs that migrate across the web on demand—this tradeoff is justified. However, as Java gains acceptance as a mainstream programming language, performance rather than platform independence becomes a prominent issue. To obtain high-performance code for less mobile applications, we are developing an optimizing compiler for Java that bypasses byte code, and, just like optimizing compilers for C or Fortran, translates Java directly to native code.

Our approach to building an optimizing compiler for Java has two novel aspects: we use an intermediate language based on *lambda-calculus*, and this intermediate language is *typed*. Intermediate representations based on lambda-calculi have been instrumental in developing high-quality implementations of functional languages such Scheme [13, 19] and Standard ML [3]. By using an intermediate language based on lambda-calculus to compile Java, we hope to gain the same organizational benefits in our compiler.

The past few years have also seen the development in the functional programming community of a new approach to designing compilers for languages like ML and Haskell based on *typed intermediate languages* [15, 20]. By emphasizing formal definition of a compiler's intermediate languages with associated type systems, this approach yields several benefits. First, properties such as type safety of the intermediate languages can be studied mathematically outside the sometimes messy environment of compiler source code. Second, type checkers can be implemented for the intermediate languages, and by running these type checkers on the intermediate programs after various transformations, we can detect a large class of errors in transformations. Indeed, by running a type checker after each transformation, we may be able to localize a bug causing incorrect code to a specific transformation, without even running the generated code. Finally, a formal definition of a typed intermediate language serves as complete and precise documentation of the interface between two compiler passes. In short, using typed intermediate languages leads to higher levels of confidence in the correctness of compilers.

Our compiler first performs ordinary Java type checking on the source program, and then translates the Java program into an intermediate language (IL) of records and first-order procedures. The translation (1) converts an object into a record containing mutable fields for instance variables and immutable procedures for methods; (2) replaces a method call with a combination of record field selections and

*STAR Laboratory, InterTrust Technologies Corp., 460 Oakmead Parkway, Sunnyvale, CA 94086. wright@intertrust.com.

†NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. {suresh,cristian}@research.nj.nec.com.

‡Media Research Laboratory, New York University, 715 Broadway, Rm 1214, New York, NY 10003. hertzman@mrl.nyu.edu.

a first-order procedure call; (3) makes the implicit *self* parameter of a method explicit by adding an additional parameter to the procedure representing that method and passing the record representing the object as an additional argument at calls; and (4) replaces Java's complex name resolution mechanisms with ordinary static scoping. The resulting IL program typechecks since the source program did, but its typing derivation uses record subtyping where the derivation for the Java program used inheritance subtyping.

In contrast to our approach, traditional compilers for object-oriented languages typically perform analyses and optimizations on a graphical representation of a program. Nodes represent arithmetic operations, assignments, conditional branches, control merges, and message sends [8]. In later stages of optimization, message send nodes may be replaced with combinations of more primitive operations to permit method dispatch optimization. In earlier stages of optimization, program graphs satisfy an informal type system which is essentially that of the source language. In later stages, program graphs are best viewed as untyped, like the representations manipulated by conventional compilers for procedural languages.

By compiling Java using a typed lambda-calculus, we hope to gain increased confidence in the correctness of the generated code. Indeed, for languages like Java that are used to write web-based applications, whether mobile or not, correctness is vital. Incorrect code generated by the compiler could lead to a security breach with serious consequences. Additionally, by translating Java into an intermediate language of records and procedures, we hope to leverage not only optimizations developed for object-oriented languages [8], but also optimizations developed for functional languages [3, 15, 20] such as Standard ML and Haskell, as well as classical optimizations for static-single-assignment representations of imperative languages [7]. In particular, representing objects as records exposes their representations to optimization. The representations of objects can be changed by transformations on IL programs, and the type system ensures that the resulting representations are consistent. Even for optimizations like inlining and copy propagation that do not explicitly change object representations, the type system provides valuable assurance that representations remain consistent.

Unfortunately, the problem of designing a sound type system that incorporates object-oriented features into a record-based language appears to have no simple solution. With a straightforward translation of objects into records and a natural type system, contravariance in the subtyping rule for function types foils the necessary subtyping relation between the types of records that represent Java objects. The problem is that making the implicit recursion through an object's self parameter explicit as an additional argument to each method leads to function types that are recursive in both covariant and contravariant positions, and hence permit no subtyping. More sophisticated type systems that can express the necessary subtyping exist [2, 5, 16], but these type systems require more complex encodings of objects and classes. Object calculi that keep self-recursion implicit [1, 5] are more complex than record calculi and do not expose representations in a manner suitable for an intermediate language.

Rather than devise an unwieldy IL and translation, we take a more pragmatic approach. We assume that a Java program is first type-checked by the Java type-checker before it is translated into the IL. Now, optimizations and transformations performed on the IL must ensure that (1) IL typing is preserved, and (2) safety invariants provided by the Java type-checker are not violated. To satisfy the first requirement, self parameters in the IL are assigned type $\top$ (top), the type that is the supertype of any record type. To satisfy the second requirement, typecase operations are inserted within method bodies to recover the appropriate type of self parameters as dictated by the Java type system. The resulting IL program is typable and performs runtime checks at typecase expressions to ensure it is safe with respect to Java typing. However, since the source program has passed the Java type-checker, these checks should never fail. Failure indicates a compiler bug. During compiler development, these checks remain in the generated object code. For production code, the code generator simply omits the checks. In either case, we lose the ability to statically detect errors in transformations that misuse self parameters. On the other hand, we can still detect a large class of type errors involving misuse of other parameters and variables, and we gain the benefit of a simple, typed intermediate language that is easy to work with.

The remainder of the paper is organized as follows. The next section presents a core IL of records and procedures. Following that, Section 3 illustrates the translation from Java to our IL with several examples. Section 4 concludes with a summary of related work.

2 Language

The following grammar defines the types of our explicitly-typed intermediate language for Java:

$$\begin{array}{rcl}
t & ::= & pt \;\big|\; rt \;\big|\; t^* \to t \\
rt & ::= & \big[\mu\alpha\big].\{\big[\mathsf{tag} : tag\big]\;\big[x : ft\big]^*\} \;\big|\; \big[\mu\alpha\big].\{\!\{\mathsf{tag} : tag\;\big[x : ft\big]^*\}\!\} \;\big|\; \alpha \\
ft & ::= & pt\ array \;\big|\; rt\ array \;\big|\; vt \\
vt & ::= & t\ var \;\big|\; t \\
pt & ::= & boolean \;\big|\; byte \;\big|\; short \;\big|\; int \;\big|\; long \;\big|\; char \;\big|\; float \;\big|\; double \;\big|\; void \;\big|\; tag
\end{array}$$

where $x \in Var$ is a set of *variables* and $\alpha \in TyVar$ is a set of *type variables* used for recursive type definitions. There are four kinds of types t: *primitive types* pt, *function types* $t_1 \cdots t_n \to t$, *ordered record types* $\{x_1 : ft_1\ \cdots\ x_n : ft_n\}$, and *unordered record types* $\{\!\{x_1 : ft_1\ \cdots\ x_n : ft_n\}\!\}$. Two additional kinds, mutable variable types *t var* and mutable array types *pt array* and *rt array*, are not full-fledged types in their own right, but may be used as types of fields in records and as types of variables. Several restrictions, which are motivated below, apply to the formation of types. The field names $x_1 \ldots x_n$ of a record type must be distinct. The first field of an unordered record type must be named tag and of type *tag*. Tags encode the static type of an object, and are used to inspect the type of a record at runtime. An ordered record type need not include a field named tag of type *tag*, but if it does, this field must appear first. Unordered record types are considered equal under different orderings of their second through last fields; that is,

$$\{\!\{\mathsf{tag} : tag,\ x_2 : ft_2\ \cdots\ x_n : ft_n\}\!\} \;=\; \{\!\{\mathsf{tag} : tag,\ permute(x_2 : ft_2, \ldots, x_n : ft_n)\}\!\}$$

where *permute* yields an arbitrary permutation of its arguments. The fields of ordered record types may not be rearranged. Both kinds of record types may be recursive if prefixed by the binding operator μ, hence

$$t = \mu\alpha.\{x_1 : ft_1\ \cdots\ x_n : ft_n\} = \{x_1 : ft_1[\alpha \mapsto t]\ \cdots\ x_n : ft_n[\alpha \mapsto t]\}$$

and

$$t = \mu\alpha.\{\!\{x_1 : ft_1\ \cdots\ x_n : ft_n\}\!\} = \{\!\{x_1 : ft_1[\alpha \mapsto t]\ \cdots\ x_n : ft_n[\alpha \mapsto t]\}\!\}$$

where $t'[\alpha \mapsto t]$ denotes the substitution of t for free occurrences of α in t'.

Figure 1 defines the subtyping relation on types. The relation allows a longer ordered record type to be a subtype of a shorter record type, provided the sequence of field names of the shorter type is a prefix of the sequence of field names of the longer type, and provided that the types of like-named fields are subtypes. Since the fields of unordered record types can be reordered arbitrarily (except for the first), a longer unordered record type is a subtype of any shorter unordered record type with a subset of the longer type's fields. An ordered record type is also a subtype of an unordered record type with the same fields. The subtyping relation includes the usual contravariant rule for function types, as well as a covariant rule for array types.

Our translation uses ordered record types to represent Java classes. In the intermediate language, subtyping on ordered record types expresses Java's single inheritance class hierarchy. Because field offsets for ordered record types can be computed statically, the translation can implement access to a member of a Java object with efficient record-field selection operations. For example, our translation could represent objects of the following Java classes:

$$pt <: pt \qquad t\ var <: t\ var \qquad \frac{t_1 <: t_2 \qquad t_2 <: t_3}{t_1 <: t_3}$$

$$\frac{t_1' <: t_1 \ \cdots\ t_n' <: t_n \qquad t <: t'}{t_1 \cdots t_n \to t <: t_1' \cdots t_n' \to t'} \qquad \frac{t <: t'}{t\ array <: t'\ array}$$

$$\frac{ft_1 <: ft_1' \ \cdots\ ft_n <: ft_n'}{\{x_1 : ft_1\ \cdots\ x_n : ft_n\ \cdots\ x_{n+m} : ft_{n+m}\} <: \{x_1 : ft_1'\ \cdots\ x_n : ft_n'\}}$$

$$\frac{ft_1 <: ft_1' \ \cdots\ ft_n <: ft_n'}{\{\!\!\{x_1 : ft_1\ \cdots\ x_n : ft_n\ \cdots\ x_{n+m} : ft_{n+m}\}\!\!\} <: \{\!\!\{x_1 : ft_1'\ \cdots\ x_n : ft_n'\}\!\!\}}$$

$$\{x_1 : ft_1\ \cdots\ x_n : ft_n\} <: \{\!\!\{x_1 : ft_1\ \cdots\ x_n : ft_n\}\!\!\}$$

$$\frac{\alpha <: \alpha' \Rightarrow t <: t' \qquad \alpha \notin t' \qquad \alpha' \notin t}{\mu\alpha.t <: \mu\alpha'.t'}$$

Figure 1: Subtyping relation.

```
class A {
    int i;
    A f( A x ) { i = 0; return x; }
}
```

```
class B extends A {
    int get_i() { return i; }
}
```

with the following IL types:

$$t_A = \mu\alpha.\ \{\ \mathsf{tag} : tag,\ \mathsf{i} : int\ var,\ \mathsf{f} : \{\!\!\{\mathsf{tag} : tag\}\!\!\} \times \alpha \to \alpha\ \}$$

$$t_B = \{\ \mathsf{tag} : tag,\ \mathsf{i} : int\ var,\ \mathsf{f} : \{\!\!\{\mathsf{tag} : tag\}\!\!\} \times t_A \to t_A,\ \mathsf{get\_i} : \{\!\!\{\mathsf{tag} : tag\}\!\!\} \to int\ \}$$

(In fact the translated types are not quite this simple; see Section 3.) The type $\{\!\!\{\mathsf{tag} : tag\}\!\!\}$ plays the role of $\top$ discussed in the introduction since any record type containing a tag field is a subtype of this type. The Java typing rules permit an object of class B to be passed to methods like f that expect an A. Since $t_B <: t_A$, values of type t_B can be passed to both IL functions f. A reference to any field of a record of type t_A or t_B is implemented as a fixed-offset access into the record.

Since Java interfaces permit multiple inheritance, ordered record types cannot support the necessary subtyping for interface types. Hence our translation uses unordered record types to represent interfaces. Accessing a particular field of a record of unordered type is more expensive as record values with different field orders can belong to the same unordered record type. The field access operation for unordered record types determines the actual order of a value's fields from the initial tag field required of the unordered type. For example, consider the following Java interface and its corresponding IL type:

```
Interface J {
    int get_i();
    A f( A x );
}
```

$$t_J = \{\!\!\{\ \mathsf{tag} : tag,\ \mathsf{get\_i} : \{\!\!\{\mathsf{tag} : tag\}\!\!\} \to int,\ \mathsf{f} : \{\!\!\{\mathsf{tag} : tag\}\!\!\} \times t_A \to t_A\ \}\!\!\}$$

If we amend class B to implement interface J, type t_B does not change, and we have $t_B <: t_J$. (Again, the translated types are not quite this simple; see Section 3.)

Figure 2 specifies the expressions e, values v, and declarations d of our intermediate language, where $g \in Tag$ are tags, $c \in Const$ are basic constants, and $r \in Prim$ are primitive operations. Constants,

e	$::=$	v	syntactic value
	$\mid$	$\text{let } d^* \text{ in } e$	binding
	$\mid$	$\{\left[x : \mathit{ft} = f\right]^*\}$	record construction
	$\mid$	x	variable reference
	$\mid$	$x := e$	variable update
	$\mid$	$e.x$	ordered record field selection
	$\mid$	$e.x := e$	ordered record field update
	$\mid$	$e@x$	unordered record field selection
	$\mid$	$e@x := e$	unordered record field update
	$\mid$	$e;\ e$	sequencing
	$\mid$	$e(e^*)$	procedure invocation
	$\mid$	$r(e^*)$	primitive invocation
	$\mid$	$\text{if } e \text{ then } e \text{ else } e$	conditional
	$\mid$	$\text{typecase } e \text{ of } \left[[g \text{ as } x \Rightarrow e]\right]^* [\text{else } e]$	type conditional
	$\mid$	$\text{try } e\ e$	exception handler
	$\mid$	$\text{raise } e$	exception raise
	$\mid$	$e[e]$	array element selection
	$\mid$	$e[e] := e$	array element update, for primitive types
	$\mid$	$e[e] :=^{\{\}} e$	array element update, for record types
v	$::=$	c	simple constant
	$\mid$	g	tag
	$\mid$	$\lambda \left[x : t\right]^* . e$	first-order procedure
	$\mid$	$\{\left[x : \mathit{ft} = v\right]^*\}$	record of values
f	$::=$	e	initial value
	$\mid$	$[e^*]$	array construction
d	$::=$	$x : \mathit{vt} = e$	value declaration
	$\mid$	$\text{rec } [\left[x : t = v\right]^*]$	set of recursive value declarations
	$\mid$	$g \approx t \left[\prec: g^*\right]$	tag declaration

Figure 2: Expression syntax.

tags, and procedures are values, as well as records where all initializers are values. Primitive operations can only appear in call position. Procedures are called by value, bind their arguments as usual, and must be *first-order*: the only free variables a procedure is allowed are *global* variables bound by top-level let-expressions. A declaration $d_i \equiv (x : \mathit{vt} = e)$ appearing in an expression $\text{let } d_1 \cdots d_n \text{ in } e'$ binds x of type vt in d_{i+1} through d_n and e'. A recursive declaration $d_i \equiv \text{rec } [x_a : t_a = v_a \cdots x_z : t_z = v_z]$ binds $x_a \ldots x_z$ of types $t_a \ldots t_z$ in all of $v_a \ldots v_z$ and e'. A tag declaration $d_i \equiv g \approx t \prec: g_1 \cdots g_n$ introduces tag g and associates it with type t and tags $g_1 \cdots g_n$. Tags $g_1 \cdots g_n$ are called *supertags* of g. Conversely, g is a *subtag* of $g_1 \cdots g_n$. The translation places a tag in a record field named tag when the type a record was constructed with may need to be recovered by a language operation like typecase. In a record construction, a field type $t\ \mathit{var}$ indicates that the field is mutable, but its initializer must be a value of type t. Similarly, declarations of type $t\ \mathit{var}$ introduce mutable variables must have initializers of type t. Mutable fields and variables are automatically "dereferenced" when accessed. (There are no values of type $t\ \mathit{var}$.) The expressions $e.x$ and $e@x$ access fields of records of ordered and unordered

$$\frac{\mathrm{TypeOf}(c) = pt}{A \vdash c : pt} \qquad \frac{A[x_1 \mapsto t_1] \cdots [x_n \mapsto t_n] \vdash e : t}{A \vdash \lambda\, x_1 : t_1 \cdots x_n : t_n \,.\, e \;:\; t_1 \cdots t_n \to t}$$

$$\frac{A(x) = vt}{A \vdash x : \mathcal{D}(vt)} \qquad \frac{A(x) = t\ \mathit{var} \qquad A \vdash e : t}{A \vdash x := e : \mathit{void}}$$

$$\frac{A \vdash e_0 : t_1 \cdots t_n \to t \qquad A \vdash e_1 : t_1 \;\cdots\; A \vdash e_n : t_n}{A \vdash e_0(e_1 \cdots e_n) : t}$$

$$\frac{\mathrm{TypeOf}(r) = pt_1 \cdots pt_n \to t \qquad A \vdash e_1 : pt_1 \;\cdots\; A \vdash e_n : pt_n}{A \vdash r(e_1 \cdots e_n) : t}$$

$$\frac{A \vdash e_1 : \mathit{boolean} \qquad A \vdash e_2 : t \qquad A \vdash e_3 : t}{A \vdash \mathsf{if}\, e_1\ \mathsf{then}\ e_2\ \mathsf{else}\, e_3 : t} \qquad \frac{A \vdash e_1 : t_1 \qquad A \vdash e_2 : t_2}{A \vdash e_1;\ e_2 : t_2}$$

$$\frac{A \vdash d_1 \Rightarrow A_1 \quad \cdots \quad A + A_1 + \cdots + A_{n-1} \vdash d_n \Rightarrow A_n \qquad A + A_1 + \cdots + A_n \vdash e : t}{A \vdash \mathsf{let}\, d_1 \cdots d_n\ \mathsf{in}\, e : t}$$

$$\frac{A \vdash e : \mathcal{D}(vt)}{A \vdash x : vt = e \Rightarrow [x \mapsto vt]} \qquad \frac{t <: T(g_1) \;\cdots\; t <: T(g_n) \qquad T(g) = t \qquad G(g) = \{g_1, \ldots, g_n\}}{A \vdash g \approx t \prec: g_1 \cdots g_n \Rightarrow [\,]}$$

$$\frac{A[x_1 \mapsto t_1 \cdots x_n \mapsto t_n] \vdash v_1 : t_1 \;\cdots\; A[x_1 \mapsto t_1 \cdots x_n \mapsto t_n] \vdash v_n : t_n}{A \vdash \mathsf{rec}\ [\, x_1 : t_1 = v_1 \;\cdots\; x_n : t_n = v_n \,] \Rightarrow [x_1 \mapsto t_1 \cdots x_n \mapsto t_n]}$$

$$\frac{A \vdash e : t \qquad t <: t'}{A \vdash e : t'}$$

Figure 3: Typing rules for simple expressions.

type, respectively. The expressions $e.x := e$ and $e@x := e$ update such fields. The unordered record operations $e@x$ and $e@x := e$ use the initial tag field of a record to determine the appropriate offset into the record. Ordinary if-expressions provide boolean conditionals, and typecase tests the tag of a record-valued expression. A typecase expression evaluates the first clause $[g\ \mathsf{as}\ x \Rightarrow e]$ for which g is a supertag of the record's tag. In the clause body e, x is bound to the record, but with a more precise type. The expression try $e_1\ e_2$ evaluates e_1 with e_2 as an exception handler. If e_1 raises no exception, its value is returned as the value of the try-expression. If e_1 raises exception v, the expression $e_2(v)$ is evaluated and its value becomes the value of the try-expression. The expression raise e evaluates e to a record v and raises an exception.

Since arrays can only appear within records in our IL, the three expressions for accessing and updating arrays actually operate on records. These operations retrieve or modify array elements associated with a record field named array. Another field named length stores an array's size. The assignment operation $e_1[e_2] :=^{\{\}} e_3$ for arrays whose elements are records sets the element of e_1.array at index e_2 to the value of e_3. Due to the covariant rule for array subtyping, this operation must also perform a runtime check to ensure that the value of e_3 is a subtype of the runtime array component type. Hence a third field named elemtag holds a tag representing the component type of the array. Since Java arrays are implicitly subtypes of the Java class Object, our translation places additional fields such as clone and getClass in records that represent arrays. We explain our rationale for this treatment of arrays below.

IL expressions must obey a collection of type checking rules. To simplify the presentation, we describe these rules in two groups. Figure 3 defines the first group of rules which concern simple expressions and procedures. The function $\mathcal{D}$ strips *var* off a type:

$$\mathcal{D}(ft) \;=\; \begin{cases} t & \text{if } ft = t\ \mathit{var}\ ; \\ ft & \text{otherwise.} \end{cases}$$

$$\frac{g \in \mathrm{Dom}(T) \qquad g \in \mathrm{Dom}(G)}{A \vdash g : \mathit{tag}}$$

$$\frac{\begin{array}{c} A \vdash e_1 : \mathit{ft}_1 \ \cdots \ A \vdash e_n : \mathit{ft}_n \\ \text{if } x_1 = \mathsf{tag} \text{ and } \mathit{ft}_1 = \mathit{tag} \text{ then } e_1 = g \text{ and } T(g) = \{x_1 : \mathit{ft}_1 \ \cdots \ x_n : \mathit{ft}_n\} \\ \text{if } x_i = \mathsf{array} \text{ and } \mathit{ft}_i = t \ \mathit{array} \text{ then } e_i = [e'_1 \cdots e'_m] \text{ and } x_j = \mathsf{length} \text{ and } e_j = \mathsf{m} \text{ and } i < j \\ \text{if } x_i = \mathsf{array} \text{ and } \mathit{ft}_i = \mathit{rt} \ \mathit{array} \text{ then } x_k = \mathsf{elemtag} \text{ and } e_k = g' \text{ and } T(g') = \mathit{rt} \text{ and } k < i < j \end{array}}{A \vdash \{x_1 : \mathit{ft}_1 = e_1 \ \cdots \ x_n : \mathit{ft}_n = e_n\} : \{x_1 : \mathit{ft}_1 \ \cdots \ x_n : \mathit{ft}_n\}}$$

$$\frac{A \vdash e : \mathcal{D}(\mathit{vt})}{A \vdash e : \mathit{vt}} \qquad \frac{A \vdash e_1 : t \ \cdots \ A \vdash e_n : t}{A \vdash [e_1 \cdots e_n] : t \ \mathit{array}}$$

$$\frac{A \vdash e : \{\cdots \ x : \mathit{ft}\}}{A \vdash e.x : \mathcal{D}(\mathit{ft})} \qquad \frac{A \vdash e : \{\cdots \ x : t \ \mathit{var}\} \qquad A \vdash e_2 : t}{A \vdash e_1.x := e_2 : \mathit{void}}$$

$$\frac{A \vdash e : \{\!\{\mathsf{tag} : \mathit{tag}, \ x : \mathit{ft}\}\!\}}{A \vdash e@x : \mathcal{D}(\mathit{ft})} \qquad \frac{A \vdash e : \{\!\{\mathsf{tag} : \mathit{tag}, \ x : t \ \mathit{var}\}\!\} \qquad A \vdash e_2 : t}{A \vdash e_1@x := e_2 : \mathit{void}}$$

$$\frac{A \vdash e_1 : \{\cdots \ \mathsf{array} : t \ \mathit{array} \ \cdots \ \mathsf{length} : \mathit{int}\} \qquad A \vdash e_2 : \mathit{int}}{A \vdash e_1[e_2] : t}$$

$$\frac{A \vdash e_1 : \{\cdots \ \mathsf{array} : \mathit{pt} \ \mathit{array} \ \cdots \ \mathsf{length} : \mathit{int}\} \qquad A \vdash e_2 : \mathit{int} \qquad A \vdash e_3 : \mathit{pt}}{A \vdash e_1[e_2] := e_3 : \mathit{void}}$$

$$\frac{A \vdash e_1 : \{\cdots \ \mathsf{elemtag} : \mathit{tag} \ \cdots \ \mathsf{array} : t \ \mathit{array} \ \cdots \ \mathsf{length} : \mathit{int}\} \quad A \vdash e_2 : \mathit{int} \quad A \vdash e_3 : t \quad t <: \{\!\{\mathsf{tag} : \mathit{tag}\}\!\}}{A \vdash e_1[e_2] :=^{\{\}} e_3 : \mathit{void}}$$

$$\frac{A \vdash e_0 : \{\!\{\mathsf{tag} : \mathit{tag}\}\!\} \qquad A[x_1 \mapsto T(g_1)] \vdash e_1 : t \ \cdots \ A[x_n \mapsto T(g_n)] \vdash e_n : t \qquad A \vdash e_{n+1} : t}{\mathsf{typecase}\ e_0 \ \mathsf{of}\ [g_1 \ \mathsf{as}\ x_1 \Rightarrow e_1] \ \cdots \ [g_n \ \mathsf{as}\ x_n \Rightarrow e_n] \, [\mathsf{else}\ e_{n+1}] : t}$$

$$\frac{A \vdash e_1 : t \qquad A \vdash e_2 : \{\!\{\mathsf{tag} : \mathit{tag}\}\!\} \to t}{A \vdash \mathsf{try}\ e_1 \ e_2 : t} \qquad \frac{A \vdash e : \{\!\{\mathsf{tag} : \mathit{tag}\}\!\}}{A \vdash \mathsf{raise}\ e : \mathit{void}}$$

Figure 4: Typing rules for records and related expressions.

A is a type assignment that maps variables to types. The rules also refer to two global maps T and G. Map $T : \mathit{Tag} \xrightarrow{\text{fin}} \mathit{Type}$ associates types with tags, and map $G : \mathit{Tag} \xrightarrow{\text{fin}} \mathcal{P}(\mathit{Tag})$ associates sets of tags with tags. An IL expression e is typable if there exist maps T and G and a typing derivation concluding $[\,] \vdash e : t$.

Most of the typing rules for simple expressions are standard; we discuss only the exceptions. The last three rules produce environments for declarations. The rule for a tag declaration $g \approx t \prec: g_1 \cdots g_n$ requires the global map T to associate g with type t, and the map G to associate g with the set $\{g_1, \ldots, g_n\}$. T allows the type associated with g to be recovered by language operations such as typecase. G abstracts the Java type hierarchy and allows language operations such as typecase to test relations in this hierarchy. For soundness, the typing rule requires that the types associated with tags related in G be similarly related under subtyping; that is, if g is declared to be a subtag of g', then $T(g) <: T(g')$.

Figure 4 defines the typing rules for records and related expressions. We explain only the non-standard rules here. A tag has type *tag*, provided that T and G associate it with appropriate types and supertags. Record expressions receive ordered record types with several restrictions. First, if the first field is named tag and has type *tag*, then its initializer must be a tag g whose type in T is the type of the entire record. This ensures that a record's type can be recovered from its tag. Second, a field may have an array initializer of length m if and only if the field's name is array and there is a field named length whose initializer is the constant m. This restriction ensures that the length field can be used for bounds checking accesses to the array. Third, if an array field is present of type *rt array* where *rt* is a record type, then the record must include a field named elemtag whose initializer is a tag corresponding to *rt*. Array update uses the elemtag field to perform its runtime type check. The third and fourth typing rules handle initializers for fields. The rules for array access and update require e_1 to be a record containing array and length fields. The rule for update where the component type is a record type additionally requires an elemtag field. The rule for typecase requires that the expression being tested have a record type including a tag field. For each clause $[g_i \text{ as } x_i \Rightarrow e_i]$, variable x_i is bound in e_i to $T(g_i)$, since the typing rule for record construction ensures that any record containing tag g_i will have type $T(g_i)$. Finally, the typing rules for exception constructs require the exception be a tagged record as the translation uses typecase within handlers to distinguish different exceptions.

Provided that array access and update operations perform bounds checks, this type system is sound. But to achieve high performance code, we need to lift array bounds checks out of loops or eliminate them entirely. Our IL is designed so that a safe array access operation can be replaced with a combination of an explicit test and a corresponding unsafe operation. For instance, we replace $e_1[e_2]$ with

```
let a = $e_1$
    i = $e_2$
in  if i $\geq$ 0 & i < a.length then $_{unsafe}$[a]i
    else raise IndexOutOfBoundsException
```

The explicit tests so introduced can then be optimized as usual.

3 Translation

The translation from Java to our intermediate language of records and procedures:

- replaces method dispatch with simple record accesses and a first-order procedure call;
- passes object state explicitly through this parameters that are treated no differently from any other function parameter;
- supports efficient implementation of member access by representing objects as ordered records;
- replaces Java's complex mechanisms for name resolution (visibility keywords, overloading, super, inner classes, and packages) with ordinary static scoping;
- flattens the class inheritance hierarchy by explicitly including record fields defined by superclasses;
- expresses method sharing among objects of the same class by placing procedures that implement the methods in a shared record;
- accommodates subtyping between Java classes by assigning type $\top$ to this and using typecase to recover the appropriate type;
- uses type tags on records to support runtime type tests and casts;
- accommodates interface subtyping by using unordered record types;
- lifts static methods and constructor and initialization procedures out of classes and represents them as top-level procedures;
- expresses class initialization as explicit tests and calls that can be optimized;
- replaces implicit conversions on primitive types with explicit operations, eliminates widening conversions in favor of implicit subtyping, and expresses narrowing conversions with typecase;
- expresses local control constructs (for, while, break, etc.) with uses of tail-recursive procedures;
- places lock and unlock instructions where control enters or leaves synchronized blocks.

In this section, we illustrate some aspects of this translation with examples. All Java objects implicitly extend class Object and hence have members such as clone and getClass, but we omit such members in these examples to simplify the presentation.

Figure 5 presents Java code defining two classes Point and ColorPoint. In the example, a Point object contains x and y coordinate fields, and methods to move a point (mv), test whether two points are the

```
class Point {
    int x;
    int y = 3;
    Point() { x = 2;};
    public void mv( int dx, int dy ) { x += dx; y += dy; };
    public boolean eq( Point other ) { return (x == other.x && y == other.y); }
    Point like() { Point p = new Point(); p.x = x; p.y = y; return p; }
}

class ColorPoint extends Point {
    int c;
    public boolean eq( Point other ) {
      if ( other instanceof ColorPoint )
        return super.eq( other ) && c == ((ColorPoint) other).c;
      else
        return false;
    }
    ColorPoint sc( int c ) { this.c = c; return this; }
    ColorPoint() { super(); }
    ColorPoint( int c ) { super(); this.c = c; }
}
```

Figure 5: Example Java classes.

same (eq), and clone a new point from the current one (like). Class ColorPoint inherits from Point and adds a color field c. ColorPoint overrides the eq method of Point and also provides a new method sc to set its color. The ColorPoint class declares two constructors. The first initializes a new ColorPoint object with a default color; the second sets the color field explicitly to the color supplied as an argument.

Figure 6 presents the record types corresponding to Point and ColorPoint. In general, records corresponding to objects include a tag field, a methods field, and fields for the instance variables, both explicit and inherited. The methods field contains a record of functions corresponding to the instance methods of the class, both explicit and inherited. Initially, this record is shared by all objects of the class, although optimizations may replace it with a record of specialized functions in certain objects. The functions take an additional first argument which is the object record itself. The IL types do not include fields for constructors or static methods as these procedures are called directly without selecting them from an object.

The types of mv and eq in t_C and t_P are the *same*. This is because Java requires that an overriding method be of the same type as the overridden one. Since t_C has at least the same fields as t_P, and since the members in the shared prefix have the same type, we have $t_C <: t_P$. Hence a record denoting a ColorPoint can be passed to a function that expects a record denoting a Point.

A program in our intermediate language consists of a set of mutually recursive values corresponding to methods, constructors, and method tables. Other than references to other top-level definitions, these procedures have no free variables. Notably, this is supplied as an explicit argument, unlike its treatment in Java and other object-based languages. This property facilitates code-movement optimizations on our IL such as inlining.

Figure 7 shows the translation of the Point class. We elide some types that are obvious from context. The translation generates a procedure newP for constructing new Point objects, a procedure initP for initializing them, a record of functions corresponding to the methods of the class, and the functions themselves. Each method function dispatches on the type of its first argument. A tag encodes the static type of an object; this type is examined at runtime using typecase. Thus, if mv is invoked by an object that is not a Point, the argument tag supplied in the call will not be a subtag of tagP, and a runtime

```
$t_P = \mu\alpha.$ { tag: $tag$,
          methods: {
            mv: $\{\!\{\mathsf{tag} : tag\}\!\} \times int \times int \to void$,
            eq: $\{\!\{\mathsf{tag} : tag\}\!\} \times \alpha \to boolean$,
            like: $\{\!\{\mathsf{tag} : tag\}\!\} \to \alpha$
          },
          x: $int\ var$,
          y: $int\ var$
     }
```

```
$t_C = \mu\beta.$ { tag: $tag$,
          methods: {
            mv: $\{\!\{\mathsf{tag} : tag\}\!\} \times int \times int \to void$,
            eq: $\{\!\{\mathsf{tag} : tag\}\!\} \times t_P \to boolean$,
            like: $\{\!\{\mathsf{tag} : tag\}\!\} \to t_P$,
            sc: $\{\!\{\mathsf{tag} : tag\}\!\} \times int \to \beta$
          },
          x: $int\ var$,
          y: $int\ var$,
          c: $int\ var$
     }
```

Figure 6: Types for Point and ColorPoint objects in the IL.

```
let tagP $\approx t_P$
   rec [newP: $\to t_P$ =
            $\lambda$. { tag:$tag$ = tagP, methods:... = Pmethods, x:$int\ var$ = 0, y:$int\ var$ = 0 }
       initP: $t_P \to void$ =
            $\lambda$this:$t_P$ . this.y := 3; this.x := 2
       Pmethods: ... =
            { mv: ... = mvP, eq: ... = eqP, like: ... = likeP }
       mvP: $\{\!\{\mathsf{tag} : tag\}\!\} \times int \times int \to void$ =
            $\lambda$this:$\{\!\{\mathsf{tag} : tag\}\!\}$, dx:$int$, dy:$int$ .
               typecase this of
                  [tagP as this $\Rightarrow$ this.x := this.x + dx; this.y := this.y + dy]
                  [else raise CompilerError]
       eqP: $\{\!\{\mathsf{tag} : tag\}\!\} \times t_P \to boolean$ =
            $\lambda$this:$\{\!\{\mathsf{tag} : tag\}\!\}$, other:$t_P$ .
               typecase this of
                  [tagP as this $\Rightarrow$ if not(this.x == other.x) then false
                                    else this.y == other.y]
                  [else raise CompilerError]
       likeP: $\{\!\{\mathsf{tag} : tag\}\!\} \to t_P$ =
            $\lambda$this:$\{\!\{\mathsf{tag} : tag\}\!\}$ .
               typecase this of
                  [tagP as this $\Rightarrow$ let it = newP() in initP(it); it ]
                  [else raise CompilerError]
   ]
in ...
```

Figure 7: Translation of Point class.

exception will be raised. Such an error will not be caught at compile-time because the type expected by mv for this argument is $\top = \{\!\{\mathsf{tag} : tag\}\!\}$. Indeed, $\top$ is the *self* type expected by all translated methods.

Figure 8 shows the translation of the ColorPoint class. An interesting aspect of ColorPoint's definition is its use of super. Calls to super in ColorPoint constructors are translated to calls to initP. The call super.eq(other) becomes a direct call to eqP since Java's semantics dictate that such uses of super bypass the usual dynamic method dispatch. Uses of typecase capture the runtime behavior of instanceof and narrowing conversions. In particular, (typecase other ...) takes the tagC branch if other.tag is tagC or any subtag of tagC. All records containing such a tag are guaranteed to represent ColorPoints, or belong to subclasses of ColorPoint.

```
    let  ...code for Point ...
  in let tagC $\approx t_P$
       rec [newC: $\rightarrow t_C$ =
              $\lambda$. { tag:$tag$ = tagC, methods:... = Cmethods, x:$int\ var$ = 0, y:$int\ var$ = 0, c:$int\ var$ = 0 }
           initC$_1$: $t_C \rightarrow void$ =
              $\lambda$this:$t_C$. initP(this)
           initC$_2$: $t_C \times int \rightarrow void$ =
              $\lambda$this:$t_C$, c:$int$. initP(this); this.c := c
           Cmethods: ... =
              { mv: ... = mvP, eq: ... = eqC, like: ... = likeP, sc : ... = scC }
           eqC: ⦃tag : $tag$⦄ $\times t_P \rightarrow boolean$ =
              $\lambda$this:⦃tag : $tag$⦄, other: $t_P$ .
                 typecase this of
                    [tagC as this $\Rightarrow$ if (typecase other of
                                                  [tagC as other $\Rightarrow$ true]
                                                  [else false]) then
                                               (if not(eqP(this, other)) then false
                                                else this.c == (typecase other of
                                                                   [tagC as other $\Rightarrow$ other]
                                                                   [else raise CastException]).c)
                                             else false]
                    [else raise CompilerError]
           scC: ⦃tag : $tag$⦄ $\times int \rightarrow t_C$ =
              $\lambda$this:⦃tag : $tag$⦄, c:$int$ .
                 typecase this of
                    [tagC as this $\Rightarrow$ this.c := c; this]
                    [else raise CompilerError]
  in ...
```

Figure 8: Translation of ColorPoint class

```
     interface Widget {
             boolean eq( Point other );
             void mv( int dx, int dy );
     }
```

```
$t_W$ = { tag: $tag$,
          methods: ⦃ tag: $tag$,
                     eq: ⦃tag : $tag$⦄ $\times t'_P \rightarrow boolean$,
                     mv: ⦃tag : $tag$⦄ $\times int \times int \rightarrow void$
                   ⦄
        }
```

```
$t'_P = \mu\alpha$. { tag: $tag$,
          methods: {
            tag: $tag$,
            mv: ⦃tag : $tag$⦄ $\times int \times int \rightarrow void$,
            eq: ⦃tag : $tag$⦄ $\times \alpha \rightarrow boolean$,
            like: ⦃tag : $tag$⦄ $\rightarrow \alpha$
          },
          x: $int\ var$,
          y: $int\ var$
    }
```

```
$t'_C = \mu\beta$. { tag: $tag$,
          methods: {
            tag: $tag$,
            mv: ⦃tag : $tag$⦄ $\times int \times int \rightarrow void$,
            eq: ⦃tag : $tag$⦄ $\times t'_P \rightarrow boolean$,
            like: ⦃tag : $tag$⦄ $\rightarrow t'_P$,
            sc: ⦃tag : $tag$⦄ $\times int \rightarrow \beta$
          },
          x: $int\ var$,
          y: $int\ var$,
          c: $int\ var$
    }
```

Figure 9: Interface Widget and types for Widget, Point, and ColorPoint.

Figure 9 illustrates a Java interface Widget and its corresponding type t_W in our IL. Since the classes that implement Widget may have methods in different orders, the methods field of t_W has an unordered record type. If we amend Point to implement Widget, the translated types t'_P and t'_C for Point and ColorPoint, also shown in Figure 9, include a tag field in their methods record to achieve the subtyping $t'_C <: t'_P <: t_W$.

4 Related Work

Optimizations for object-oriented languages, type systems for object-oriented languages, and typed intermediate languages are three topics that have been investigated independently by other researchers and relate to the work presented here.

Optimizations for Objects

An important issue addressed by optimizing compilers for object-oriented languages is reducing the overhead introduced by encoding polymorphism. Statically-typed object-oriented languages such as Java support polymorphism through subclassing. Subclasses share implementations with their parents. Because methods can be overridden to provide alternative implementations, the exact method invoked at a call site may not be easily determined at compile time. Indeed, without aggressive analyses, compilers are unlikely to determine the control flow of a program that makes any significant use of inheritance. On the other hand, relying only on intraprocedural optimization may not be effective because methods are usually short and make frequent calls to other methods.

There are two main ways of eliminating the dispatch at a call $x.f(\ldots)$. Either (i) the value of the receiver x can be of only one type T, in which case we can call T's method f directly, or (ii) x can be of any of the types in a set S, but all types in S share the same implementation of f, in which case we can call f directly. Concrete type inference and class hierarchy analysis are two well-known analyses that have been devised to address the issue of dispatch elimination.

Concrete Type Inference [14, 17, 9, 10] is a form of flow analysis that identifies, for each expression, the set of possible types its values may belong to. When a receiver is found to have only one possible type, the method dispatch can be replaced by a direct function call to that type's method.

Class Hierarchy Analysis [9, 4, 10] is a program analysis that, based solely on the program's class structure, identifies a set of types S that share the same implementation of method f. An example of such a set is the set containing class C and all subclasses of C that do not override f. Such sets can be computed either from programmer's annotations ("final" in Java) or from inspection of the complete class hierarchy. The analysis can be adapted to work, although less beneficially, in the presence of separate compilation, where *implementations* are separated from *interfaces*. In such cases it is still possible to eliminate method dispatch at link time [11].

Even if the above analyses are unable to identify a call site as calling a unique function, it may still be possible to optimize the program by using a *type-case* statement with execution branching on the exact type of the value to code specific to each possible type [6]. Message splitting is a variation of this technique which consists of duplicating not only the method call on each branch of the type case, but subsequent statements as well, whenever this enables further optimizations.

Dynamically typed languages, and to a lesser extent statically typed languages, could benefit from *type feedback*—information about the set of concrete types that a receiver is observed to have during program's execution. Comparison of type feedback with either class hierarchy analysis [9] or concrete type inference [12] shows it to be a valuable technique.

In contrast to our typed intermediate language, the intermediate language on which these optimizations have typically been performed is an untyped control-flow graph. Low-level nodes in the graph are used to represent arithmetic operations, assignments, conditional branches, etc. High-level nodes are used to represent the semantics of method calls [6]. High-level nodes help the compiler postpone code-generation decisions for method dispatch until after optimizations aimed at replacing method calls with direct function calls are performed. Remaining method dispatches are then translated into more primitive operations, and the code is then subject to further intra-procedural optimizations. This approach is well-suited for implementing dynamically typed languages, where a method dispatch can be a rather heavy-weight construct. On the other hand, in a statically typed language with single inheritance such as Java, method dispatch consists of fetching a function pointer from a record from a known offset, and calling that function. We believe that in such a setting, an intermediate language based on first order functions and records is a viable alternative. All the complicated constructs of the source language, including method dispatch, are translated into simpler operations. Flow analysis techniques used to drive inter-procedural optimizations for functional languages can be directly applied to our intermediate language and need not be modified to understand the nuances of method dispatch. By having available the function tables constructed for each type, analyses can still compute a reasonably precise conservative approximation to the set of methods called at a call site, facilitating optimizations like inlining.

Type Systems for Objects

In designing our typed IL for Java, we considered and rejected several alternatives. A naive attempt to translate Java into a record-based IL uses the same language and type system as ours, but gives self parameters the object's record type rather than $\top$. That is, mv, eq, and like in class Point all expect a value of type t_P for their first argument. This solution fails because, translating ColorPoint the same way, we no longer have $t_C <: t_P$ due to contravariant subtyping of functions. Hence many Java-typable programs are not typable under such a translation.

Several object calculi have most of the language features found in Java and support the necessary subtyping [1, 5]. However, in these calculi, self parameters are implicitly bound, and method dispatch is not broken down into separate function selection and procedure call mechanisms. Consequently it would be difficult to adapt existing techniques for optimizing procedural languages to such calculi. Moreover, the complexity of these calculi make them inappropriate as the foundation for an IL.

Finally languages that employ a *split-self* semantics represent an object as a pair of a record containing the object's state and a record containing the object's code [16]. They use existential types to achieve subtyping, and include pack and unpack operations to manipulate values of existential type. The encoding of objects in this style is complex and unwieldy for use in a compiler.

Typed Intermediate Languages

Several advanced functional language implementations have embraced the use of a typed intermediate language to express optimizations and transformations [18, 20]. The motivation for using a typed intermediate language holds equally well in the context of a Java implementation. Like most functional languages, Java has a rich type system and requires aggressive compiler optimization to achieve acceptable performance. However, while the intermediate language type systems developed for functional language implementations have been based on a polymorphic λ-calculus, the type system in our IL more closely reflects features found in Java. Thus, it provides record subtyping to express single inheritance, unordered record types to express interfaces, and a tag type to express runtime type inspection.

To summarize, our typed intermediate language for Java serves three major roles: (1) it gives us increased confidence in the correctness of optimizations; (2) it exposes salient properties of an object's representation that may be then optimized; and (3) it facilitates type-specific decisions throughout the compiler and runtime system. We are confident that a typed intermediate language of this kind will be instrumental in realizing a high-performance Java implementation.

References

[1] Abadi, M., and Cardelli, L. *A Theory of Objects.* Springer-Verlag, 1996.

[2] Abadi, M., Cardelli, L., and Viswanathan, R. An Interpretation of Objects and Object Types. In *Proceedings of the Conference on Principles of Programming Languages* (1996), pp. 392–406.

[3] Appel, A. W. *Compiling with Continuations.* Cambridge University Press, 1991.

[4] Bacon, D., and Sweeney, P. Fast static analysis of C++ virtual function calls. *OOPSLA'96 Conference on Object-Oriented Programming Systems, Languages, and Applications* (1996).

[5] Bruce, K. B., Cardelli, L., and Pierce, B. C. Comparing Object Encodings. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan* (Sept. 1997).

[6] Chambers, C. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, March 1992.

[7] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *TOPLAS 13*, 4 (October 1991), 451–490.

[8] Dean, J., DeFouw, G., Grove, D., Litvinov, V., and Chambers, C. Vortex: An optimizing compiler for object-oriented languages. *OOPSLA'96 Conference on Object-Oriented Programming Systems, Languages, and Applications* (1996), 83–100.

[9] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. *ECOOP* (1995).

[10] Diwan, A., Moss, E., and McKinley, K. Simple and effective analysis of statically-typed object-oriented programs. *OOPSLA'96 Conference on Object-Oriented Programming Systems, Languages, and Applications* (1996).

[11] Fernandez, M. F. Simple and effective link-time optimization of modula-3 programs. *Proceedings of the Conference on Programming Language Design and Implementation* (1995), 103–115.

[12] Hölzle, U., and Agesen, O. Dynamic versus static optimization techniques for object-oriented languages. *OOPSLA'95 Conference on Object-Oriented Programming Systems, Languages, and Applications* (1995).

[13] Kranz, D., Kelsey, R., Rees, J. A., Hudak, P., Philbin, J., and Adams, N. I. Orbit: An optimizing compiler for scheme. *ACM SIGPLAN Conference Proceedings* (1986).

[14] Palsberg, J., and Schwartzbach, M. I. Object-oriented type inference. *OOPSLA '91 Conference on Object-Oriented Programming Systems, Languages, and Applications* (1991), 146–161.

[15] Peyton-Jones, S., Launchbury, J., Shields, M., and Tolmach, A. Briding the gulf: A common intermediate language for ML and Haskell. In *Proceedings of the Conference on Principles of Programming Languages* (1998), ACM Press, pp. 49–61.

[16] Pierce, B. C., and Turner, D. N. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming 4*, 2 (Apr. 1994), 207–247. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[17] Plevyak, J., and Chien, A. A. Precise concrete type inference for object-oriented languages. *OOPSLA '94 Object-Oriented Programming Systems, Language, and Applications* (1994), 324–340.

[18] Shao, Z. Flexible Representation Analysis. In *Proceedings of the International Conference on Functional Programming* (1997), ACM Press, pp. 85–98.

[19] Steele Jr., G. L. Rabbit: a compiler for scheme. Master's thesis, Massachusetts Institute of Technology, May 1977.

[20] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the Conference on Programming Language Design and Implementation* (1996), ACM Press, pp. 181–192.