A Design for Type-Directed Programming in Java

Stephanie Weirich, Liang Huang

University of Pennsylvania
Philadelphia, PA, USA
{sweirich, lhuang3}@cis.upenn.edu

Abstract

Type-directed programming is an important and widely used paradigm in the design of software. With this form of programming, an application may analyze type information to determine its behavior. By analyzing the structure of data, many operations, such as serialization, cloning, adaptors and iterators may be defined once, for all types of data. That way, as the program evolves, these operations need not be updated—they will automatically adapt to new data forms. Otherwise, each of these operations must be individually redefined for each type of data, forcing programmers to revisit the same program logic many times during a program’s lifetime.

The Java language supports type directed programming with the instanceof operator and the Java Reflection API. These mechanisms allow Java programs to depend on the name and structure of the run-time classes of objects. However, the Java mechanisms for type-directed programming are difficult to use. They also do not integrate well with generics, an important new feature of the Java language.

In this paper, we describe the design of several expressive new mechanisms for type-directed programming in Java, and show that these mechanisms are sound when included in a language similar to Featherweight Java. Basically, these new mechanisms pattern-match the name and structure of the type parameters of generic code, instead of the run-time classes of objects. Therefore, they naturally integrate with generics and provide strong guarantees about program correctness. As these mechanisms are based on pattern matching, they naturally and succinctly express many operations that depend on type information. Finally, they provide programmers with some degree of protection for their abstractions. Whereas instanceof and reflection can determine the exact run-time type of an object, our mechanisms allow any supertype to be supplied for analysis, hiding its precise structure.

Keywords: Type-Directed programming, Object-oriented programming, Reflection, Generics, Polytypism

1 This work is supported by NSF grant CCF-0347289 CAREER: Type-Directed Programming in Object Oriented Languages.
1 Introduction

The design and structuring of software is a difficult task. Good software engineering requires code that is concise, reusable and easy to modify. Consequently, modern statically-typed programming languages include abstraction mechanisms such as subtype and parametric polymorphism (the latter is also called generics) to allow programmers to decompose complicated software. While these abstraction mechanisms are useful, they do not cover all situations. They do not apply to operations that are most naturally defined by the structure of data. These operations require a different set of abstraction mechanisms called type-directed programming.

With type-directed programming, the program analyzes type information to determine its behavior. That way, if the arguments change structure, the operations adapt automatically. Without this mechanism, each of these operations must be individually defined and updated for each type of data, forcing programmers to revisit the same program logic many times. This redundancy increases the chance of error and reduces maintainability. It makes changing data representations unattractive to programmers because many lines of code must be modified.

A typical use of type-directed programming is for serialization. Serialization converts any data object into an appropriate form for display, network transmission, replication, or persistent storage. So that programmers can define their own version of routines like serialization, the Java programming language \[16\] includes run-time type identification (with the keyword instanceof) and the Reflection API \[17\]. Figure 1 demonstrates an implementation of serialization for cyclic data structures in Java.

The class Pickle contains the method pickle that converts any object to a string of characters by examining its type structure. For recursive data structures, this operation uses a hash table to note objects previously serialized. For objects that have not been previously serialized, it first determines whether the object’s class represents one of the primitive types (such as Integer or Boolean). If so, it uses one of the primitive operations for converting the object to a string. Otherwise, pickle recursively serializes each field of the object.

The benefit of implementing serialization in this manner is that it is independent of the class structure. Without this mechanism, each class must implement its own serialization routine. This scattering of program logic throughout the classes means that, as the application is updated, the serialization methods must be continually defined and updated in many places. Even if we do not mind this commingling of concerns, defining and maintain-
class Pickle {
    // hash table for cycle detection
    protected HashMap hashMap;

    public String pickle( Object obj ) {
        if (obj == null) return "null";
        // Check to see if we’ve seen obj before.
        // If not, store a unique id for this object.
        if ( hashMap.containsKey( obj ) )
            return (String)hashMap.get( obj );
        String id = "+#" + Integer.toString( hashMap.size() + 1 );
        hashMap.put(obj,id);
        // Switch on the class of the object
        Class<Object> objClass = obj.getClass();
        if ( obj instanceof Integer ) {
            Integer i = (Integer) obj.intValue();
            return Integer.toString( i );
        } else if ( obj instanceof Boolean ) {
            Boolean b = (Boolean) obj.booleanValue();
            return Boolean.toString( b );
        } else try {
            // If obj is not a primitive type or array, then
            // determine all fields and recursively pickle
            // each field, separated by commas.
            String result =
                "[" + id + ":" + objClass.getName() + ", ";
            Field[] f = objClass.getDeclaredFields();
            for ( int i=0; i<f.length; i++ ) {
                f[i].setAccessible(true);
                result += f[i].getName() + ":" + pickle( f[i].get( obj ) );
                if (i < (f.length - 1)) result += ",";
            }
            return result + "]";
        } catch ( IllegalAccessException e )
            return "Impossible";
    }

    // Constructor---creates an empty hashtable
    Pickle() { this.hashMap = new HashMap(); }
}

Fig. 1. Type-directed Serialization in Java
ing these distributed methods is tedious and error-prone, especially if the code maintainers are not the original authors. Type-directed programming allows the programmer to define operations in one location.

Furthermore, unlike other design patterns that separate operations from their data such as the Visitor pattern [15], type-directed programming does not suffer from the extensibility problem. As existing classes change, because type-directed operations are defined by type structure, they do not need to be updated. Palsberg and Jay have shown that implementing the visitor pattern with reflection solves the extensibility problem [30].

Type-directed programming plays a critical role in the development of software systems. Many basic operations are most naturally defined over the structure of type information. Besides serialization, cloning (making identical, deep copies of data), structural equality, and iteration (applying an operation to each data element in a collection) may be defined by type structure.

Type-directed programming is also important at the boundaries of software components. Extensible systems can use type information to ensure the stability of the system. They can check that newly loaded code satisfies the requirements of the running system and provides the necessary interfaces before accepting a dynamic update [20].

Furthermore, when interfaces are known during development, type-directed proxies may be used to adapt the interface of a component to a particular situation. For example, if an application always calls each method of a particular class with the same first argument, type-directed programming can define a wrapper for the class that automatically provides that argument [33]. Also, such proxies can be used to log, trace, profile or debug function calls to all of the methods of a specific component [19].

Finally, type-directed programming is also useful at the boundary between software and user. It allows functionality to be automatically reflected to the user as it is added to a system. For example, with JavaBeans [23], a system may examine the interface of a new component to directly provide user-interface control of the component in the form of check boxes, selection lists, buttons, etc.

Problems with current mechanisms in Java

Although the Java mechanisms for type-directed programming promote program modularity—the serialization routine in Figure 1 may be applied to any object—serialization also demonstrates the flaws of instanceof and the Reflection API when compared to type-directed programming in other languages.

For example, using instanceof or reflection in Java almost always requires
run-time type casting, leading to redundant checks and a potential for dynamic failure. In this example, when obj is an Integer, it must be cast to the Integer class before it may be converted to a string. Furthermore, in the case that obj is not one of the classes representing primitive types, an exception handler for the IllegalAccessException must be installed. This exception could be raised by each field access. However, because the only accessed fields are those provided by getDeclaredFields, this exception will never be raised. When reflection is used correctly, the run-time casts are redundant. However, because reflection could be used incorrectly, the programmer must consider the situation when the run-time check fails, and must write code to handle exceptions such as ClassCastException or IllegalAccessException. The fact that these run-time casts must be included in correct code is a symptom of the fact that reflection is a relatively low-level mechanism for defining type-directed operations.

Furthermore, reflection breaks user-defined abstractions in Java. The method getDeclaredFields produces a data structure that contains all fields of the object, including those declared to be protected or private. Therefore, programmers cannot rely on private fields to hide information or enforce program modularity. The call setAccessible(true) prevents IllegalAccessException from being raised when private and protected fields are accessed. To prevent access to private fields, the entire program may be run with a security manager that causes the setAccessible command to fail. However, such coarse control falls short of the programmable access control that is provided in other domains.

1.1 New mechanisms for Java

In this paper, we propose new mechanisms for type-directed programming in Java that may be used instead of instanceof or Reflection. These new mechanisms are based on an extension of Java with first-class genericity—one in which the types that instantiate the parameters to generic methods and classes are available at run time. Whereas the current implementation of generics in Java (based on GJ [5]) erases such types before the program is run, extensions such as NextGen [8,3] efficiently provide this type information at run-time. First-class generics already provide many benefits to the Java programming language [2].

Our new mechanisms analyze this first-class type information directly, instead of examining the run-time class of objects. There are several advantages to this approach:

- Our new mechanisms can provide stronger guarantees of correctness. Just
as the introduction of generics allowed some casts to be eliminated from Java programs, this mechanism also can remove potential failure points.

- Our new mechanisms are easier to use. The mechanisms that we propose provide sophisticated type matching capabilities, giving users a convenient way to program with type information. In particular, these mechanisms can more naturally encode the structure of type-directed algorithms.

- Our new mechanisms integrate well with generics. Because of the type erasure implementation of generics, the current implementation of Java Reflection and instanceof do not provide accurate information about generic classes and methods. Although it is possible to extend instanceof and Java Reflection [32] to generics, we think that our mechanisms are a more natural integration.

- Our new mechanisms are expressive in terms of protecting abstraction. Reflection and instanceof analyze the most specific type of an object. However, run-time type information that describes an object’s type can be any supertype of the actual type. Packages that do not want the complete structure of their objects to be determined through analysis can provide abbreviated versions of the objects’ types to type-directed operations.

In the next section, we informally describe mechanisms for analyzing the name and the structure of type parameters and show their expressiveness. In Section 3, we formalize their semantics in a core calculus similar to Featherweight Generic Java [21]. The main result of this paper is that we show that these new mechanisms are type-safe. Finally, we discuss related work and possible future extensions.

2 Analyzing type parameters

We can roughly divide type-directed mechanisms into two categories: those that determine the name of the run-time type (analogous to instanceof) and those that determine its structure (analogous to reflective mechanisms). Both sorts of mechanisms are necessary: recall the implementation of serialization. In the following subsection, we first describe operators that may be used in place of instanceof and in the next subsection we discuss replacements for reflection.

2.1 Nominal Analysis

Consider a new expression form for Java called ifsubtypeof. This expression form is a conditional—it chooses one of two branches based on whether a type variable \( T \) is a subtype of a specific type at run-time. If the condition
holds, the type checker can perform *type refinement*—the types of variables mentioning $T$ can change in the branch, eliminating redundant type casts [12]. For example, if $x$ has type $T$, then in the case below, we know that $T$ is a subtype of $\text{Integer}$ and that $x$ does not need to be cast to $\text{Integer}$ before being used.

```java
T x;
if subtypeof(T, Integer) {
    // $T=\text{Integer}$ in this branch so $x$ has type $\text{Integer}$
}
```

Furthermore, type variables create equations between types. Determining the run-time identity of a single type variable, may refine the class of many objects.

```java
List<T> x;
if subtypeof(T, Integer) {
    // here we know $T$ is a subtype of $\text{Integer}$,
    // so all of the elements of the list $x$ are $\text{Integers}$.  
}
```

By analyzing type parameters we remove potential failure points from the program. Otherwise, when examining the run-time classes of references to objects with $\text{instanceof}$, their types can change unexpectedly, due to aliasing. A version of $\text{instanceof}$ that incorporates type refinement would not be sound. We cannot statically eliminate casts such as the one below because we have no guarantee that the run-time type of $x$.field remains constant.

```java
if (x.field instanceof Integer) { writeInt ((Integer)x.field); }
```

If the type of $x$.field is $\text{Object}$, any thread may update $x$.field with an object of any type, at any time. However, if the type of $x$.field is a type parameter $T$, if we determine that $T$ is a subtype of $\text{Integer}$, then $x$.field must contain an object whose type is a subtype of $\text{Integer}$. No cast is necessary, because other threads may only assign $\text{Integers}$ to $x$.field.

```java
if subtypeof(T, Integer) { writeInt (x.field); }
```

However, $\text{if subtypeof}$, like $\text{instanceof}$, is limited with respect to parameterized classes. It cannot determine whether a type $T$ is any sort of list. Instead it must compare $T$ against $\text{List}</U>$ where $U$ is a specific type. Using $\text{List}</\text{Object}>$ is not sufficient because of invariance of type parameters: $\text{List}</\text{Integer}>$ is not a subtype of $\text{List}</\text{Object}>$.

We can make nominal analysis more expressive by combining it with pattern matching. The expression form $\text{typematch}$, below, generalizes $\text{if subtypeof}$. This expression matches an argument type against a number of type patterns—types that may contain pattern variables.
getClassName<T>  The name of the class as a string.
getFieldName<T,f>  The name of the field f in class T as a string.
getMethodName<T,m>  The name of the method m in class T as a string.
numFields<T>  The number of fields as an integer.
numMethods<T>  The number of methods as an integer.

Fig. 2. Simple operations on type structure

T x;
typematch T with
  Integer:  // Here x is an integer.
  List<U>:  // Here x is a list of U’s.
  default:  // Here we know nothing about x.

Like ifsubtypeof the type checker can refine the static type information to correspond to the branch of the expression. If a pattern does not contain any free type variables, then typematch behaves the same as ifsubtypeof.

2.2 Structural Analysis

The expressions ifsubtypeof and typematch correspond to (and generalize) operations such as instanceof that are useful when we know that the class of an object could be one of a finite set of classes. However, many times the programmer has no knowledge of the class of an object, but nevertheless wants to determine the structure of that class. In Java, the Reflection API provides this capability. In this subsection, we present our alternatives to reflection that are based on analyzing the structure of run-time type information.

Some of the capabilities of reflection are simple to define for type parameters. For example, Figure 2 lists several operations that operations that determine the name of a class or the number of methods or fields.

However, it is more difficult to define operations that determine precise information about fields and methods and provide safe access to them. Our approach is to add new expression forms for iterating over the fields and methods of a class. The forfield expression iterates over the fields in a class T, binding the type parameter X to the type of each field and an accessor variable f to the name of each field. The accessor variable may be used to project the current field from an object of type T. We might use forfield as follows:
public <T> String pickle(T obj) {
    if (obj == null) return "null";
    // Check if we’ve seen obj. If not, store a unique id for it.
    if (hashMap.containsKey(obj))
        return (String)hashMap.get(obj);
    String id = "#" + Integer.toString(hashMap.size() + 1);
    hashMap.put(obj, id);
    // Switch on the class of the object
    typematch T with
        Integer: return Integer.toString(obj);
        Boolean: return Boolean.toString(obj);
    ... : // Cases for other base types
        X[] : // Case for arrays
        default: {
            String result = "[" + id + ":" + getName<T> + " ";
            Int i = 0;
            forfield (X f in T) {
                result += getFieldName<T,f> + pickle<X>(obj.f);
                i++;
                if (i < numFields<T>) result += ",";
            }
            return result + "]" ;
        }
}

T obj;
// Iterate over all of the fields of T
forfield (X f in T) {
    // Bind type parameter X and accessor variable f. Then
    // Refine T so that it contains one field "f" of type X
    X fieldVal = obj.f;
    print<X>(fieldVal); // Can analyze X like any other type
}

Using forfield, we can rewrite the pickle method as shown in Figure 3. In this example, we use typematch to determine whether T is a base type or an array type. If it is neither, then forfield iterates over the fields in the object, calling the serialization routine recursively. If the pickle operation is called with the run-time type of its argument, then the result will be the
same as the previous version of pickle. However, not only is this version of pickle shorter and free from type casts, but it is more flexible with respect to type abstraction. The caller of pickle is free to determine what “view” of the object should be produced, by providing different type arguments. For example, if there is a component of the run-time type that the caller would like not to be printed, then the caller can provide a supertype that does not include that component, as in the example below.

```java
class NotSecret { int anynumber = 54321; }
class Secret extends NotSecret { int mysecretnumber = 12345; }
Secret y = new Secret();
System.out.println(Pickle.pickle<Secret>(y));
System.out.println(Pickle.pickle<NotSecret>(y));
```

Both calls to pickle are valid, but only the second prints both fields of y.

The semantics of the forfield expression form is not as simple as it appears. In the body of forfield, the identity of the type T should be refined to be a type that includes a field called f of type X. However, the type that contains a single field f probably does not exist. Because Java’s type system requires that objects may only be assigned types that are the names of pre-defined classes, there most likely will not be a class with the right structure that we can refine T to. Therefore, as described in the next section, we add a very limited form of structural object types to Java. This addition is not surprising given that we are verifying the structural analysis of object types. However, unlike other structural type systems, that may produce types that are confusing to programmers, this extension is rather benign. In essence, the only “structural” types are those that contain a single field or method.

To reflect common idioms in type-directed programs, we combine type pattern matching with forfield. The type of each field is represented by a type pattern. In the above examples, this pattern was always a single variable. However, any pattern may be used. For example, the pattern may be a literal type, in which case, the body executes for each field with that type.

```java
T obj = ...;
forfield (Integer f in T) {
    // Increment all integer-valued fields in obj.
    obj.f = obj.f + 1;
}
```

Other type patterns may select all fields of the class that are arrays (no matter what type of elements that they contain) or all static fields in a class.

Analogously forfield, the formethod expression iterates over the methods found in a class. Type patterns are very important for this iteration. For
example, suppose we would like to pick out all methods in a class that take no arguments, return void, and whose name starts with ”test”. We may do so with the following code:

```java
static <T> void runtests (T x) {
    for (method ( void m() in T ) { 
        if ( getMethodName<T,m>.startsWith("test") ) {
            x.m();
        }
    }
}
```

A difference between iterating over fields and iterating over methods is that because of method type parameters and multi-argument methods, it is impossible to write a pattern general enough to match every method in a class. The pattern must specify the number of type and term parameters of the method. It is possible that more complicated pattern languages could produce more expressiveness. For example, we could add a “wild card” pattern that matches any method type. However, this would be of limited use as we cannot call the method without being able to determine the number of type and term parameters that it requires.

### 3 Semantics

To more fully describe the semantics of our new type-analysis operators and to provide some assurance that they are sound within the context of the Java
programming language, we next formalize a small Java-like language extended with these constructs. Like Featherweight Generic Java (FGJ) [21], our new language, called TDJ for Type-Directed Java, is a functional core of an object-oriented language with nominal subtyping. Besides type-analyzing operators, this language includes only top-level class definitions, object instantiation, field access, method invocation, and type casts. We omit many of the features of Java that are orthogonal to our study, such as mutation, concurrency, exceptions, and interfaces.

Like NextGen [8], TDJ has a type-passing semantics. We allow type parameters in places that Generic Java does not. For example, type parameters may be used for run-time type casts and object instantiation. To support abstract object creation in this model, all classes must declare instance initializers for all fields. In a new $X(\overline{x})$ expression, because the class is abstract, the type checker does not know how many arguments must be supplied to the constructor. However, if not enough values are supplied with the new expression, then the values of the initializers may be used for the missing fields. To simplify the semantics of the language we require that the initializers be syntactic values in a class declaration.

The abstract syntax of TDJ is shown in Figure 4. We use the metavariables $C$ and $D$ to refer to class names, $x$ and $y$ to refer to expression variables, and $X, Y$ and $Z$ to refer to type variables. Like FGJ, we greatly abuse the sequence notation, for example using $T \overline{f}$ to refer to $T_0 f_1, \ldots, T_0 f_n$. The notation $|T|$ is the length of the sequence. Sequences of names (such as for types, variables, fields and methods) are required to contain no duplicates. Additionally, this should not be the name of a field or a variable.

The semantics of the core of this calculus is very similar to that of FGJ. For space reasons, the rules and auxiliary functions for typing appear in a companion technical report [36]. To make our model closer to Java, we give it a small-step call-by-value semantics instead general reduction rules.

There are a few notable differences in the type systems of this calculus and FGJ. Many rules require finding the least non-variable upper bound of a type. In FGJ, variables may not be bound by other variables, but in TDJ, we do not preserve this invariant. Therefore, we define the upper bound of $T$ in $\Delta$, written $\text{bound}_\Delta(T)$, recursively. Other typing rules that differ from FGJ include T-Class, where we check that the initializers for the fields are well formed and T-New where, because of the presence of field initializers, fewer arguments than fields may be supplied to a new expression.
3.1 Nominal analysis

The expression form `typematch T with T : e default : e` allows programmers to pattern match the names of run-time type information. The semantics of this expression are in Figure 5. The dynamic semantics of this expression form relies on the partial function `matches(T, U)` that produces a substitution Σ (if one exists) such that `T = Σ(U)`. (Σ(·) notates a simultaneous substitution.) In the first computation rule for `typematch`, the argument of the pattern match must be a closed type N. If this type matches the first pattern, then Σ replaces the pattern variables in the branch.

If the first pattern does not match—if we cannot derive the match judgment—then the semantics discards the first pattern and examines the remaining patterns. Because every match expression must end with a default branch, some branch will be taken. For a simple, deterministic semantics, even if several

\[\begin{align*}
\text{Computation:} & \quad matches(N, X) = X \mapsto N \\
& \quad matches(C X \langle S \rangle, C X \langle T \rangle) = equals(C X \langle S \rangle, C X \langle T \rangle) \\
& \quad equals(S, T) = \Sigma \quad \Sigma \text{ consistent} \\
& \quad equals(C X \langle S \rangle, C X \langle T \rangle) = \Sigma \\
& \quad matches(N, T) = \Sigma \\
& \quad typematch N with T : e \ T : e \ default : e' \mapsto e'(e) \quad [\text{R-Match}] \\
& \quad matches(N, T) \text{ is not defined} \\
& \quad typematch N with T : e \ T : e \ default : e' \\
& \quad \quad \leftarrow \text{typematch N with T : e \ default : e'} \\
& \quad \quad \quad [\text{R-NoMatch}] \\
& \quad \quad \quad \text{typematch N with \ default : e' \mapsto e'} \\
& \quad \quad \quad \quad [\text{R-Default}] \\
\end{align*}\]

Static Semantics:

\[\begin{align*}
\Delta \vdash T & \text{ ok} \quad \text{dom}(\Delta) = FV(T) \quad \forall X \in \text{dom}(\Delta), \Delta(X) = \text{Object} \quad [\text{M-MINOK}] \\
\Delta & \vdash T \text{ ok} \quad \text{dom}(\Delta) = FV(T) \quad \forall X \in \text{dom}(\Delta), \Delta(X) = \text{Object} \quad [\text{M-MINOK}] \\
S \lls: T & \in \Delta \quad \text{bound}_\Delta(T) = N \quad [\text{B-REFINE}] \quad S \lls: T \in \Delta \quad \Delta \vdash S \lls: T \quad [\text{S-REFINE}] \\
(1 \leq i \leq \lvert T \rvert) & \quad \Delta_i \vdash \text{ok} \quad \Delta_i \vdash \text{ok} \quad \Delta_i; \Gamma ; e' \in U' \lls: U \\
& \quad \Delta_i; \Gamma \vdash \text{typematch T with T : e \ default : e'} \in U \quad [\text{T-REFINE}] \\
\end{align*}\]

Fig. 5. Nominal type pattern matching
patterns match the analyzed type, the first match is selected. However it would be possible for the operation of \texttt{typematch} to select the most precise pattern.

The definition of \textit{matches} is at the top of Figure 5. Because of the invariance of the arguments to parameterized classes, we must determine not just when a type could be a subtype of a pattern (after some substitutions) but also when it could equal the pattern (after some substitutions). For this reason we define both \textit{matches}(S, T) and \textit{equals}(S, T). The first rule states that we can always match a type to a pattern variable. There is a similar rule for \textit{equals}(S, T). To match a non-variable type to a pattern, we must either match it to the same class, where all of the type arguments are equal, or we must see if its superclass matches the pattern. Likewise, to determine if a non-variable type is equal to a pattern, the pattern must be for the same class and all of the type arguments must be equal.

In a pattern match expression, each branch is checked in a refined context that assumes that the analyzed type is a subtype of the pattern. This context eliminates the need for type casts. We refine the context by adding a special assumption \( S \ll<: T \). Note that this assumption is between arbitrary type and is not a variable binding. In fact, the free variables of both \( S \) and \( T \) should also be bound in the context. (In the rule T-\textsc{Refine} the judgment \( \Delta \vdash T \text{\tt minok} \) determines a context that includes exactly the free variables of \( T \) with the highest bound. For the soundness of the language, it is important that we do not introduce any extra assumptions when we check each branch.

With context refinement, we can conclude that \( \Delta \vdash S <: T \) (see the rule S-\textsc{Refine}). Such an assumption also determines the bound of a type variable. Note that in some contexts, some variables may have multiple bounds. There are no restrictions about what refinement assumptions we may add to the context. If they are not satisfiable (for example, assuming \( C \ll<: D \) when there is no relationship between the classes \( C \) and \( D \)) then the branch of \texttt{typematch} that introduced that assumption could never be taken. It is sound to omit checking such branches.

Furthermore, for simplicity we do not make any “deep” conclusions from type matching assumptions. For example, from \( C<: X \ll<: C<: Y \) it would be sound to also conclude that the type \( X \) equals \( Y \). It would be straightforward to incorporate such deductions with equality assumptions, but to keep the language simple, we have not done so for this calculus.

\subsection*{3.2 Structural analysis}

Structural type analysis in TDJ determines what fields and methods are present in a class, with the expression forms \texttt{fieldfold} and \texttt{methfold}. Be-
cause of the functional nature of TDJ, these forms are designed as “folds”. A language with mutation could simplify these forms into iteration, such as forfield and formethod, described in Section 2.

The semantics of an expression $\text{fieldfold}_i(x = e; S f_x \in T) e'$ appears in Figure 6. This expression iterates over the fields of the type $T$. The variable $x$ is an accumulator, initialized with the value of $e$. The expression $e'$ executes once for each field whose type matches the pattern $S$. The variable $f_x$ is an accessor variable that refersto the current name of the field. The index $i$ is the index of the current field. In source programs, it should always be 1.

The operational semantics of $\text{fieldfold}$ is defined by four rules. First, a congruence rule allows the accumulator to be evaluated to a value. In the second rule, the index refers to a field in the class, and the type of that field matches the type in the pattern. In that case, the body of $\text{fieldfold}$ becomes the new accumulator, after substituting the current accumulator for $x$, the current field name for the accessor variable, and the types generated by the pattern match. The next rule is used when the type of the current field does not match the pattern, skipping that field. In the last rule, the index is
out of range so the accumulator is returned.

The body of a `fieldfold` expression is type checked in a refined context. A refinement assumption $S \ll: \{T \; f_x\}$ means that any expression of type $S$ may project a field $f_x$ with type $T$. This assumption is used in the rule $T\text{-FieldVar}$ to check a field access when the accessor is a variable. The rule for checking a field access for constant accessors is unchanged.

Method folding behaves like field folding, so we elide its semantics. (See the technical report [36] for details.) Like `fieldfold` the operational semantics iterates through the methods of an object, executing the body of the fold for each matching method type. To determine method type matches, like method overriding, we require equality patterns for the bounds and the arguments to the method, but we allow the return type to be a subtype.

We have shown that this language is type sound, using a similar proof to that for FGJ [21]. The full details of this proof appear elsewhere [36].

4 Related Work

The earliest examples of type-directed programming were based on nominal analysis. Many languages, such as Simula-67 [4], CLU [28], Cedar/Mesa [26], Modula-2+ and Modula-3 [7], have mechanisms to hide the names of types at compile time (via a dynamic type, variously called `any`, `REFANY`, or `Object`) and to recover the type name at run time (such as `INSPECT`, `instanceof` or `TYPECASE`). To some extent, dynamic dispatch in object languages is an example of nominal analysis. Haskell type classes [34] provide dynamic dispatch in a non-OO language by using type parameters.

Structural type analysis is important for languages with composite types. Java (like many other languages, such as Amber [6], Cedar/Mesa [26], and C#) provides a mechanism to `reflect` a type into a data structure. However, even though programmers can define operations based on the types, this mechanism cannot tie the type of operations to the reflected type stored in a data structure. As a result, static type checking is compromised. Type-directed operations rely on run-time casts to guarantee their type correctness. Crary et al. [12] showed that it was possible to reflect type information into a data structure, yet permit static type checking, by using a form of dependent type.

The approach taken in this paper most closely resembles systems that allow pattern matching of run-type information in functional programming languages. For example, to support dynamic typing in the statically typed ML language [29], the `Dynamic` type of Abadi et al. [1] and of Leroy [27] encapsulates run-time type information and provides a special elimination form (called `typecase`) to discover that type information.
However, with type Dynamic, only type information that is stored in dynamic values can be analyzed. In contrast, intensional polymorphism [18], extensional polymorphism [13] and structural polymorphism [31] analyze explicit type parameters. These frameworks require that the language semantics propagate type information at run-time, independently of values. The G’Caml [14] language implements of extensional polymorphism.

A related line of research, called polytypic programming or generic programming, pattern matches compile-time type information to generate type-indexed operations. These mechanisms define operations, such as maps and folds, that are defined by parameterized types (also called type constructors). The Charity [11] language automatically generates maps and folds for datatypes at compile time, but cannot be extended with new type-directed operations. Functorial ML [25] uses combinators to define parameterized types and then defines type-directed operations based on these combinators. The ideas behind this theory were incorporated into the FiSH language [24]. Polytypic programming [22] extends Haskell with a way to define type-directed operations. Generic Haskell extends polytypism in the Haskell language so that it can handle mutually recursive, nested or multiparameter datatypes, and datatypes that contain functions [10]. This framework is not the same as pattern matching type information—a type-directed operation is an interpretation of that lambda-calculus term. However, Weirich [35] showed that it can be reconciled with run-time type analysis.

The difference between systems that pattern match run-time or compile-time type information and the mechanisms that we propose here is that our type system must be compatible with subtyping. Because previous work only dealt with type equalities, they are able to use a particularly simple mechanism for type refinement—substitution. However, in TDJ, we do not know that a type X is equal to another type T, only that it is a subtype. Therefore we could not substitute T for X, but must have a special constraint in the context to record the fact that X is a subtype of T. These constraints are similar to those of Cheney and Hinze [9] and Xi et al. [37].

Another difference is that we provide a mechanism to structurally analyze the fields and methods of object types. The difficulty with object types is that their structure is quite complex. Fields and methods are indexed by name, and there may be an arbitrary number of components. Previous work has had difficulty dealing with the analogue in functional languages—record and variant types—and has resorted to ad hoc solutions. For example, Haskell type classes [34] require help from the user—they cannot automatically generate operations for variant and record types. Generic Haskell [10] converts variant and record types into an internal representation to define basic op-
erations, leading to a mismatch between the definition of the type-directed operation and the types at which it is used. Java Reflection uses accessors such as `getFields` and refers to method and field names as strings, but cannot statically guarantee the correctness of accessing fields or of invoking methods.

5 Conclusions and Future work

This paper describes a new approach to the design of mechanisms for run-time type analysis in Java. Instead of basing execution on the run-time classes of objects, `typematch`, `fieldfold` and `methfold` examine the structure of first-class type information. Because of this approach, these mechanisms statically refine the context to reflect dynamic type discovery. As a result, type-directed operations may be expressed without the need for type casting.

There are several extensions to this design that we plan to explore. The first is to provide a way for programmers to assign type parameters to the run-time type of objects. That way, type information does not need to be explicitly passed throughout the program. A drawback of this extension is a loss of abstraction—anyone may discover the most precise type of an object.

Other extensions allow us to discover more information about run-time types. For example, the type operator `Super<T>` returns the supertype of a class as a new type parameter. (For `Object` it returns `Object`.) We have not included `Super` in the current version of the language for simplicity. With such a type operator, we have a non-trivial definition of type equivalence.

Finally, we plan to explore ways to make the structural operators described in this calculus more flexible, through the use of dependent type systems. To make type checking in such a system tractable, we must limit what terms can determine type structure. Finding an expressive but tractable set of restrictions that the programmer can understand will require careful engineering.

Acknowledgement

Thanks to Benjamin Pierce and Steve Zdancewic. This work has been supported by NSF grant 0347289, CAREER: Type-Directed Programming in Object-Oriented Languages.

References


