A Design for Type-Directed Programming in Java*

Stephanie Weirich Liang Huang
University of Pennsylvania
{sweirich,lhuang3}@cis.upenn.edu

June 19, 2004

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 guarantess 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 from others.

1 Introduction

The design and structuring of software is a difficult task. Good software engineering requires code that is concise, manageable, 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 in useful ways. 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

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

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 to type-directed operations 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 at many times during a program's life cycle. This redundancy increases the chance of error and reduces program 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 (for fault tolerance), or persistent storage. So that programmers can define their own version of routines like serialization, the Java programming language [19] includes run-time type identification (with the keyword `instanceof`) and the Reflection API [20]. 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. So that it may serialize recursive data structures, this operation uses a hash table to record 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 disparate places. Even if we do not mind this commingling of concerns, defining and maintaining 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 (or package) without modification of the rest of the program and without dependence on the specific classes found in other packages. Doing so means that the system may be divided into more coherent semantic units because the new operations do not need to be interspersed into existing modules.

Furthermore, unlike design patterns that separate operations from their data, such as the Visitor pattern [18], type-directed programming does not suffer from the extensibility problem. As existing modules and classes change, because type-directed operations are defined by type structure, they are still valid and do not need to be updated. Palsberg and Jay have shown that implementing the visitor pattern with reflection solves the extensibility problem [36].

Type-directed programming also plays a critical role in the development of many other parts 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 [23]. For example, the Common Object Model (COM) [14], treats objects abstractly and provides access to clients through one or more interfaces. All objects must implement the interface `IUnknown`, which provides the function `QueryInterface`

```
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 if ... {  // Cases for other base types or if array
    } 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();  }
}
```

Figure 1: Type-directed Serialization in Java

for clients to call at runtime to determine whether the object implements a particular interface.

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 [40]. Also, such proxies can be used to log, trace, profile or debug function calls to all of the methods of a specific component [22].

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 [28], 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 the `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 Java 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 [6]) erases such types before the program is run, extensions such as NextGen [9, 4] efficiently provide this type information at run-time. First-class generics already provide many benefits to the Java programming language [3]. Our new mechanisms analyze this first-class type

information directly, instead of examining the run-time class of objects.

There are several advantages to analyzing first-class types instead of the run-time classes of objects.

- 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 [39] 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.

The structure of this paper is as follows. In the next section, we informally describe mechanisms for analyzing the name and the structure of type parameters. In that section, we also show the expressiveness of our new mechanisms by describing how to implement some of the algorithms that previously required `instanceof` and Java Reflection. In Section 3, we formalize the semantics of these new mechanisms in a calculus like Featherweight Generic Java [24]. The main result of this paper is that we show that these new mechanisms are type-safe additions to this core calculus. Finally, we discuss related work and possible future extensions of our mechanisms.

2 Analyzing type parameters

We can roughly divide the typed-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 that the implementation of serialization required both. In the following subsection, we first describe the 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 [15]. For example, if `x` has type `T`, then in the case below, we know that `T` is a subtype of `Integer` and that `x` does not need to be cast to `Integer` before being used.

```
T x;
ifsubtypeof(T, Integer) {
   // T=Integer in this branch so x has type Integer
}
```

Furthermore, type variables create equations between types. If we determine the run-time identity of a single type variable, we may discover the class of many objects.

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

By analyzing first-class type parameters we remove potential failure points from the program. Otherwise, when examining the run-time classes of references to objects with `instanceof`, their types can change unexpectedly, due to aliasing. A version of `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.

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

If the type of `x.field` is `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 the identity of `T`, then `x.field` must contain an object whose type is a subtype of that type. No cast is necessary, because other threads may only assign `Integers` to `x.field`.

```
ifsubtypeof(X,Integer) { writeInt (x.field); }
```

However, `ifsubtypeof`, like `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 `List<U>` where `U` is a specific type. Using `List<Object>` is not sufficient because of invariance of type parameters: `List<Integer>` is not a subtype of `List<Object>`.

We can make our operator more expressive by combining it with pattern matching. The expression form `typematch`, below, generalizes `ifsubtypeof`. This expression matches an argument type against a number of type patterns—types that may contain pattern variables.

```
T x;
typematch T with
   Integer:  // Here x is an integer.
   List<U>:  // Here x is a list of Us and we can analyze U further.
   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

`getClassName<T>`	Returns the name of the class as a string.
`getFieldName<T,f>`	Returns the name of the field `f` in class `T` as a string.
`getMethodName<T,m>`	Returns the name of the method `m` in class `T` as a string.
`numFields<T>`	Returns the number of fields as an integer.
`numMethods<T>`	Returns the number of methods as an integer.

Figure 2: Simple operations on type structure

```
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: Integer.toString(obj);
     Boolean: 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 + "]" ;
     }
}
```

Figure 3: Pickling with structural type analysis

of classes. However, many times the programmer has no knowledge of the class of an object, but nevertheless wants to determine the operation based on the *structure* of that class. In Java, the Reflection API currently provides this capability for finding out information about the run-time class of objects. However, in this section we present several alternatives to reflection that are based on analyzing the structure of run-time type information.

We can start with simple operations for determining the information about the structure of classes. For example, operations that determine the name of the class or the number of methods or fields are not difficult to add to the language. Figure 2 lists several straightforward operations that we might add.

More importantly, type-directed operations require a mechanism for determining precise infor-

mation about fields and methods and providing safe access to them. For example, the serialization code in Figure 1 recursively calls `pickle` for each field in the object. Our approach to this functionality is to add new expression forms for iterating over the fields and methods of a class. For example, 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 acessor variable may be used to project the current field from an object of type `T`. We might use `forfield` as follows:

```
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 the type
    // 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 also 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.

```
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 one prints out 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 predefined 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.

Types	S,T,U	::=	X | N
Non-variable types	N,P,Q	::=	C<$\overline{\mathtt{T}}$>
Class declarations	CL	::=	class C<$\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}$> $\lhd$ N{ $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ =$\overline{\mathtt{v}}$; $\overline{\mathtt{M}}$ }
Method declarations	M	::=	<$\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}$> T m ($\overline{\mathtt{T}}\ \overline{\mathtt{x}}$){ return e; }
Method types	MT	::=	<$\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}$>($\overline{\mathtt{T}}$)$\rightarrow$T
Expressions	e	::=	x | e.f | e.m <T>($\overline{\mathtt{e}}$) | new T($\overline{\mathtt{e}}$) | (T)e
Values	v	::=	new C($\overline{\mathtt{v}}$)
Type contexts	Δ	::=	$\varnothing$ | Δ, X <:S | Δ, S $\ll$: T | Δ, S $\ll$: {T $\mathtt{f_x}$} | Δ, T $\ll$: {MT $\mathtt{m_x}$}
Term contexts	Γ	::=	$\varnothing$ | Γ, x:S
Type substitutions	Σ	::=	$\varnothing$ | Σ, X $\mapsto$ T

Figure 4: TDJ Syntax

```
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:

```
static <T> void runtests (T x) {
  formethod( void m() in T ) {
     if ( getMethodName<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. However, despite this limitation, method iteration is a useful tool for type-directed programming.

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) [24], 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

Matching:

$$matches(\mathtt{N}, \mathtt{X}) = \mathtt{X} \mapsto \mathtt{N}$$

$$equals(\mathtt{N}, \mathtt{X}) = \mathtt{X} \mapsto \mathtt{N}$$

$$matches(\mathtt{C<\overline{S}>}, \mathtt{C<\overline{T}>}) = equals(\mathtt{C<\overline{S}>}, \mathtt{C<\overline{T}>})$$

$$\frac{equals(\overline{\mathtt{S}}, \overline{\mathtt{T}}) = \overline{\Sigma} = \Sigma \qquad \Sigma \text{ consistent}}{equals(\mathtt{C<\overline{S}>}, \mathtt{C<\overline{T}>}) = \Sigma}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}> \lhd N\{\ \ldots\ \}}}{matches(\mathtt{C<\overline{S}>}, \mathtt{T}) = matches([\overline{\mathtt{S}} \mapsto \overline{\mathtt{X}}]\mathtt{N}, \mathtt{T})}$$

Computation:

$$\frac{matches(\mathtt{N}, \mathtt{T}) = \Sigma}{\mathtt{typematch\ N\ with\ T : e\ \ \overline{T} : \overline{e}\ \ default : e'} \mapsto \Sigma(\mathtt{e})} \text{ [R-Match]}$$

$$\frac{matches(\mathtt{N}, \mathtt{T}) \text{ is not defined}}{\mathtt{typematch\ N\ with\ T : e\ \ \overline{T} : \overline{e}\ \ default : e'} \mapsto \mathtt{typematch\ N\ with\ \overline{T} : \overline{e}\ \ default : e'}} \text{ [R-NoMatch]}$$

$$\frac{}{\mathtt{typematch\ N\ with\ \ default : e'} \mapsto \mathtt{e'}} \text{ [R-Default]}$$

Static Semantics:

$$\frac{\Delta \vdash \mathtt{T} \text{ ok} \qquad dom(\Delta) = FV(\mathtt{T}) \qquad \forall \mathtt{X} \in dom(\Delta),\ \Delta(\mathtt{X}) = \mathtt{Object}}{\Delta \vdash \mathtt{T} \text{ minok}} \text{ [M-Minok]}$$

$$\frac{\mathtt{S} \ll: \mathtt{T} \in \Delta \qquad bound_{\Delta}(\mathtt{T}) = \mathtt{N}}{bound_{\Delta}(\mathtt{S}) = \mathtt{N}} \text{ [B-Refine]} \qquad \frac{\mathtt{S} \ll: \mathtt{T} \in \Delta}{\Delta \vdash \mathtt{S} <: \mathtt{T}} \text{ [S-Refine]}$$

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{T} \text{ ok} \qquad \Delta \vdash \mathtt{U} \text{ ok} \\ \Delta; \Gamma \vdash \mathtt{e'} \in \mathtt{U'} <: \mathtt{U} \qquad (1 \leq i \leq |\overline{\mathtt{T}}|) \quad \Delta_i \vdash \mathtt{T}_i \text{ minok} \quad \Delta, \Delta_i, \mathtt{T} \ll: \mathtt{T}_i; \Gamma \vdash \mathtt{e}_i \in \mathtt{U}_i <: \mathtt{U}\end{array}}{\Delta; \Gamma \vdash \mathtt{typematch\ T\ with\ \overline{T} : \overline{e}\ \ default : e'} \in \mathtt{U}} \text{ [T-Refine]}$$

Figure 5: Nominal type pattern matching

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 [9], TDJ has a type-passing semantics. We allow type parameters in places that Generic Java does not. For example, type parameters may be used in the arguments of run-time type casts. Furthermore, we allow type parameters to be used for object instantiation. To support abstract object creation in this model, all classes must declare instance initializers for all fields. In a `new X(e)` 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. We could relax this restriction by defining evaluation of class declarations in the class table.

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 $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ to refer to $\mathtt{T}_0\ \mathtt{f}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{T}_0\ \mathtt{f}_n$. The notation $|\overline{\mathtt{T}}|$ 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. We include the rules and auxiliary functions for typing in the Appendix (in Figures 7, 8, 9, and 10). To make our model closer to Java, we choose to give 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 Δ, written $\mathit{bound}_\Delta(\mathtt{T})$, 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 $\mathtt{typematch\ T\ with}\ \overline{\mathtt{T}} : \overline{\mathtt{e}}\ \mathtt{default} : \mathtt{e}$ allows programmers to pattern match the names of run-time type information. The semantics related to this expression are in Figure 5. The dynamic semantics of this expression form relies on the auxiliary function $\mathit{matches}(\mathtt{T}, \mathtt{U})$. When this function is defined, `T` can match the pattern `U`. 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 the produced substitution Σ replaces the pattern variables in the branch. The notation $\Sigma(\mathtt{e})$ stands for this simultaneous substitution.

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 patterns match the analyzed type, the first match is selected. However it would be possible for the operation of `typematch` to select the most precise pattern.

The definition of *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 $\mathit{matches}(\mathtt{S}, \mathtt{T})$ and $\mathit{equals}(\mathtt{S}, \mathtt{T})$. The first rule states that we can always match a type to a pattern variable, producing the substitution that replaces the pattern variable with the type. There is a similar rule for $\mathit{equals}(\mathtt{S}, \mathtt{T})$. 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. The substitutions that equalize the type arguments must also be consistent with eachother.

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 refinement means that, unlike `instanceof`, we do not need to cast an expression to match the new type. Technically, we refine the context by adding a special assumption $\mathtt{S} \lll: \mathtt{T}$. If such a refinement is in a context Δ, we can conclude that $\Delta \vdash \mathtt{S} <: \mathtt{T}$, with the rule S-Refine. Such an assumption can also be used to determine the bound

Computation:

$$\frac{\mathtt{e} \mapsto \mathtt{e}'}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}_0 \mapsto \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}'; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}_0} \text{ [E-FFCong]}$$

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad 1 \leq i \leq |\overline{\mathtt{f}}| \qquad \mathit{matches}(\mathtt{T}_i, \mathtt{T}) = \Sigma}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{fieldfold}_{i+1}(\mathtt{x} = [\mathtt{x} \mapsto \mathtt{v}, \mathtt{f_x} \mapsto \mathtt{f}_i]\Sigma(\mathtt{e}); \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}} \text{ [E-FFMatch]}$$

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad 1 \leq i \leq |\overline{\mathtt{f}}| \qquad \mathit{matches}(\mathtt{T}_i, \mathtt{T}) \text{ is not defined}}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{fieldfold}_{i+1}(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}} \text{ [E-FFSkip]}$$

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad i > |\overline{\mathtt{f}}|}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{v}} \text{ [E-FFBase]}$$

Static semantics:

$$\frac{\mathtt{T} \ll: \{\mathtt{U}\ \mathtt{f_x}\} \in \Delta}{\Delta \vdash \mathtt{T} \ll: \{\mathtt{U}\ \mathtt{f_x}\}} \text{ [RF-Hyp]} \qquad \frac{\Delta \vdash \mathtt{U} \ll: \{\mathtt{T}\ \mathtt{f_x}\} \qquad \Delta \vdash \mathtt{S} <: \mathtt{U}}{\Delta \vdash \mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\}} \text{ [RF-Trans]}$$

$$\frac{\begin{array}{c} i > 0 \qquad \Delta; \Gamma \vdash \mathtt{e} \in \mathtt{U}'' <: \mathtt{U} \\ \Delta \vdash \mathtt{T}' \text{ ok} \qquad \Delta' \vdash \mathtt{T} \text{ minok} \qquad \Delta, \Delta', \mathtt{T}' \ll: \{\mathtt{T}\ \mathtt{f_x}\}; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \end{array}}{\Delta; \Gamma \vdash \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{T}')\ \mathtt{e}' \in \mathtt{U}} \text{ [T-FieldFold]}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0 \ll: \{\mathtt{T}\ \mathtt{f_x}\}}{\Delta; \Gamma \vdash \mathtt{e.f_x} \in \mathtt{T}} \text{ [T-FieldVar]}$$

Figure 6: Field folding

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 `typematch` that introduced that assumption could never be taken. A smart type checker could soundly 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.

3.2 Structural analysis

Structural type analysis in TDJ determines what fields and methods are present in a class, with the expression forms `fieldfold` and `methfold`. Because 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 $\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{S}\ \mathtt{f_x} \in \mathtt{T})\ \mathtt{e}'$ 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 $\mathtt{e}'$ executes once for each field whose type matches the pattern `S`. The variable $\mathtt{f_x}$ is an accessor variable—a variable referring to the current name of the field. The index i is the index of the current field. In source programs, the index should always be 1.

The operational semantics of `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 `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 any analysis of that field. In the last rule, the index is out of range for the fields of the analyzed class so the accumulator is returned.

The body of a `fieldfold` expression is checked in a refined context. If a refinement assumption $\mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\}$ is present in a context, it means that any expression of type `S` may project a field $\mathtt{f_x}$ with type `T`. This assumption is used in the rule T-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 analogously to field folding, so its semantics appears in the Appendix, in Figure 11. 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 [24]. The full details of this proof appear in a companion technical report [45].

4 Related Work

The earliest examples of type-directed programming were based on nominal analysis. Many languages 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`). The languages Simula-67 [5], CLU [34], Cedar/Mesa [31], Modula-2+ and Modula-3 [8] have such mechanisms. To some extent, dynamic dispatch in object languages is an example of nominal analysis. Haskell type classes [42] provide dynamic dispatch in an non-OO language by using type parameters.

Structural type analysis is important for languages composite types, such as arrays, tuples, records, and variants, and objects. Java Reflection (like mechanisms in many other languages, such as Amber [7], Cedar/Mesa [31], and C# [25]) provides a mechanism to *reflect* type information into a data structure. However, even though programmers can define operations based on the type of a value, this mechanism cannot tie the type of operations to the reflected type information stored in the 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. [15] 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 to connect the reflected data with the type information. This reflected data must be analyzed by a special expression form. However, this special datatype and its eliminator may be encoded with higher-order parametric polymorphism, as shown by Weirich [43]. Cheney and Hinze [10] used a similar idea to implement Crary et al.'s language as a Haskell library.

The approach taken in this paper most closely resembles systems that allow the pattern matching of run-type information in functional programming languages. For example, to support dynamic typing in the statically ML language [35], the `Dynamic` type of Abadi et al. [1, 2] and of Leroy [33] encapsulated run-time type information and provided a special elimination form (called `typecase`) to pattern match that type information.

However, with type `Dynamic`, only type information that is stored in dynamic values can be analyzed. In contrast, intensional polymorphism [21, 41], extensional polymorphism [16] and structural polymorphism [37, 38] are mechanisms that analyze explicit type parameters. These frameworks require that the language semantics propagate type information at run-time, independently of values. The G'Caml [17] language is a current extension of O'Caml [32] with extensional polymorphism.

Also, there is related line of research, called polytypic programming or generic programming, that pattern matches compile-time type information to generate type-indexed operations. The mechanisms in this line of research can define operations, such as maps and folds, that are defined by *parameterized types* (also called *type constructors*). The Charity [13] language automatically generates maps and folds for datatypes at compile time, but cannot be extended with new type-directed operations. Functorial ML [30] 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 [29]. Polytypic programming [26, 27] 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 [12]. This framework is a little further from simply pattering matching type information—parameterized types are built from the simply-typed lambda-calculus, and a type-directed operation is an interpretation of that lambda-calculus term. However, it can be reconciled with run-time type analysis, as was shown by Weirich [44].

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 must only deal with type equalities, they are able to use a particularly simple mechanism for type refinement. They use substitution to reflect knowledge of type parameters. 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 just 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 the constraints produced by the phantom types of Cheney and Hinze [11], and the guarded recursive datatypes of Xi et al. [46].

Another difference is that in our system 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 have resorted to ad hoc solutions. For example, Haskell type classes [42] require help from the user—they cannot automatically generate operations for variant and record types. Generic Haskell [12] converts variant and record types into an internal representation to define basic operations, leading to a mismatch between the definition of the type-directed operation and the types at which it is used. In Figure 1 we saw that 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 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, `matchtype`, `fieldfold` and `methfold` examine the structure of first-class type information. Because of this approach, these mechanisms may statically refine the context to reflect the dynamic type discovery. As a result, type-directed operations may be expressed with our mechanisms without the need for type casting.

There are several extensions to this design that we plan to explore in the future. 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. For example, one might use this new capability as follows:

```
public void f(Object x) {
  // T is run-time type of x
  <T> local = x;
  // local is an alias for x with that type
  typeDirectedMethod<T>(local);
}
```

A drawback of this extension is a loss of abstraction—this extension provides a way for anyone to discover the most precise type of an object.

Another extension would allow us to discover more information about run-time types. For example, we could add a type operator `Super<T>` that would return the supertype of a class as a new type parameter. (For `Object` it would just return `Object`.) The reason that we have not done so in the current version of the language is for simplicity. Introducing such a type operator means that 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. With dependent types, the type of a term can be determined by arbitrary values of other terms. 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.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[3] Eric Allen and Robert Cartwright. The case for run-time types in Generic Java. In *Principles and Practice of Programming in Java*, June 2002.

[4] Eric Allen, Robert Cartwright, and Brian Stoler. Efficient implementation of run-time generic types for Java. In *IFIP WG2.1 Working Conference on Generic Programming*, July 2002.

[5] G. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur, Lund, Sweden, 1973.

[6] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[7] Luca Cardelli. Amber. In Guy Coisineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 48–70. Springer-Verlag, 1986.

[8] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, Digital Equipment Corporation, Systems Research Center, November 1989.

[9] Robert Cartwright and Guy L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, British Columbia*, pages 201–215. ACM, 1998.

[10] James Cheney and Ralf Hinze. Poor man's dynamics and generics. In Manuel M. Chakravarty, editor, *Proceedings of the ACM SIGPLAN 2002 Haskell Workshop*. ACM Press, 2002.

[11] James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[12] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[13] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[14] Microsoft COM technologies, January 2002. `http://www.microsoft.com/com/default.asp`.

[15] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

[16] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *Twenty-Second ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 118–129, San Francisco, January 1995.

[17] Jun Furuse. Generic polymorphism in ML. In *Journées Francophones des Langages Applicatifs*, 2001.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachussetts, 1994.

[19] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[20] Dale Green. Trail: The reflection API. In Mary Campione, Kathy Walrath, Alison Huml, and Tutorial Team, editors, *The Java Tutorial Continued: The Rest of the JDK(TM)*. Addison-Wesley Pub Co, 1998. `http://java.sun.com/docs/books/tutorial/reflect/index.html`.

[21] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[22] Tom Harpin. Using `java.lang.reflect.proxy` to interpose on Java class methods. `http://developer.java.sun.com/developer/technicalArticles/JavaLP/Interposing/`, July 2001.

[23] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In R. Harper, editor, *Types in Compilation: Third International Workshop, TIC 2000; Montreal, Canada, September 21, 2000; Revised Selected Papers*, volume 2071 of *Lecture Notes in Computer Science*, pages 147–176. Springer, 2001.

[24] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[25] International Organisation for Standardization and International Electrotechnical Commission. *ISO/IEC 23270:2003 Information technology–C# Language Specification*, April 2003.

[26] Patrick Jansson and Johan Jeuring. PolyP—A polytypic programming language extension. In *Twenty-Fourth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 1997.

[27] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University of Technology and Göteborg University, 2000.

[28] JavaBeans: The only component for Java technology, May 2002. `http://java.sun.com/products/javabeans/`.

[29] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

[30] C. Barry Jay, Gianna Bellè, and Eugenio Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, November 1998.

[31] Butler Lampson. A description of the Cedar language. Technical Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

[32] Xavier Leroy. *The Objective Caml System, Release 3.06*. Institut National de Recherche en Informatique et Automatique (INRIA), 2002.

[33] Xavier Leroy and Michel Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 406–426. Springer-Verlag, August 1991.

[34] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU reference manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[35] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[36] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 1998.

[37] Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns Over Types.* PhD thesis, University of Michigan, 1992.

[38] Fritz Ruehr. Structural polymorphism. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998.*, 1998.

[39] Jose H. Solorzano and Suad Alagić. Paramteric polymorphism for Java: A reflective solution. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 216–225, Vancouver, Canada, 1998.

[40] Paul Tremblett. Java reflection. *Dr. Dobbs Journal*, January 1998.

[41] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 82–93, Montreal, September 2000.

[42] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Sixteenth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

[43] Stephanie Weirich. Encoding intensional type analysis. In D. Sands, editor, *10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 92–106, Genova, Italy, 2001. Springer.

[44] Stephanie Weirich. Higher-order intensional type analysis. In Daniel Le Métayer, editor, *11th European Symposium on Programming*, pages 98–114, Grenoble, France, April 2002.

[45] Stephanie Weirich and Liang Huang. A design for type-directed Java (Extended version). Technical Report MS-CIS-04-11, University of Pennsylvania, June 2004. Available at `http://www.cis.upenn.edu/~lhuang3/pubs/tdj-tr.ps`.

[46] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

A Additional Semantics

$$\frac{\mathit{bound}_{\Delta}(\Delta(\mathtt{X})) = \mathtt{N}}{\mathit{bound}_{\Delta}(\mathtt{X}) = \mathtt{N}} \text{ [B-Var]}$$

$$\mathit{bound}_{\Delta}(\mathtt{N}) = \mathtt{N} \text{ [B-NonVar]}$$

$$\frac{}{\mathit{fields}(\mathtt{Object}) = \bullet} \text{ [F-Object]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ \lhd\ N\{\overline{T'}\ \overline{f'}\ =\overline{v'};\overline{M}\}} \qquad \mathit{fields}([\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N}) = \ \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathit{fields}(\mathtt{C<T>}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}](\overline{\mathtt{T'}}\ \overline{\mathtt{f'}})} \text{ [F-Class]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ \lhd\ N\{\overline{T}\ \overline{f}\ =\overline{v};\overline{M}\}}}{\mathit{fieldval}(\mathtt{f}_i, \mathtt{C<T>}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{v}_i} \text{ [FV-Class]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ \lhd\ N\{\overline{T}\ \overline{f}\ =\overline{v};\overline{M}\}} \qquad \mathtt{f} \notin \overline{\mathtt{f}}}{\mathit{fieldval}(\mathtt{f}, \mathtt{C<T>}) = \mathit{fieldval}(\mathtt{f}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N})} \text{ [FV-Super]}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ \lhd\ N\{\overline{S}\ \ \overline{f}\ =\overline{v};\ \overline{M}\ \ \}} \\ \mathtt{<\overline{Y} \lhd \overline{P}>\ U\ m(\overline{U}\ \ \overline{x})\{\ return\ e;\ \}} \in \overline{\mathtt{M}} \end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C<\overline{T}>}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}](\mathtt{<\overline{Y} \lhd \overline{P}>\ \overline{U}\ \rightarrow U})} \text{ [MT-Class]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ \lhd\ N\{\overline{S}\ \ \overline{f}\ =\overline{v};\ \overline{M}\ \ \}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C<\overline{T}>}) = \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N})} \text{ [MT-Super]}$$

$$\frac{\mathtt{class\ C<\overline{X} \lhd \overline{N}>\ \lhd\ N\{...;\ \overline{M}\ \ \}} \qquad \mathtt{<\overline{Y} \lhd \overline{P}>\ U\ m(\overline{U}\ \ \overline{x})\{\ return\ e;\ \}} \in \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m<\overline{S}>}, \mathtt{C<\overline{T}>}) = (\overline{\mathtt{x}}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}, \overline{\mathtt{Y}} \mapsto \overline{\mathtt{S}}]\mathtt{e}_0)} \text{ [MB-Class]}$$

$$\frac{\mathtt{class\ C<\overline{X} \lhd \overline{N}>\ \lhd\ N\{...;\ \overline{M}\ \ \}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m<\overline{S}>}, \mathtt{C<\overline{T}>}) = \mathit{mbody}(\mathtt{m<\overline{S}>}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N})} \text{ [MB-Super]}$$

Figure 7: Auxillary operations

$$\Delta \vdash \mathtt{T} <: \mathtt{Object} \ [\text{S-Object}]$$

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \ [\text{S-Refl}]$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}} \ [\text{S-Trans}]$$

$$\frac{}{\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})} \ [\text{S-Var}]$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}> \lhd N\{\ \ldots\}}}{\Delta \vdash \mathtt{C<\overline{T}>} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N}} \ [\text{S-Class}]$$

$$\frac{}{\Delta; \Gamma \vdash x \in \Gamma(x)} \ [\text{T-Var}]$$

$$\frac{\begin{array}{c} bound_{\Delta}(\mathtt{T}) = \mathtt{N} \\ fields(\mathtt{N}) = \overline{\mathtt{T}} \quad \overline{\mathtt{f}} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{T}'} \qquad |\overline{\mathtt{T}'}| \leq |\overline{\mathtt{T}}| \qquad \Delta \vdash \mathtt{T}'_i <: \mathtt{T}_i \ \ (for\ 1 \leq i \leq |\overline{\mathtt{T}'}|) \end{array}}{\Delta; \Gamma \vdash \mathtt{new\ T}(\overline{\mathtt{e}}) \in \mathtt{T}} \ [\text{T-New}]$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \qquad bound_{\Delta}(\mathtt{X}) = \mathtt{N} \\ mtype(\mathtt{m}, \mathtt{N}) = \mathtt{<\overline{Y} \lhd \overline{P}>\ \overline{U} \to U} \qquad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}} \qquad \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}} \end{array}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{m<\overline{T}>}(\overline{\mathtt{e}}) \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}} \ [\text{T-Invk}]$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad bound_{\Delta}(\mathtt{T}_0) = \mathtt{N} \qquad fields(\mathtt{N}) = \overline{\mathtt{T}} \quad \overline{\mathtt{f}}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{f}_i \in \mathtt{T}_i} \ [\text{T-Field}]$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{U}}{\Delta; \Gamma \vdash \mathtt{(T)e} \in \mathtt{T}} \ [\text{T-Cast}]$$

Figure 8: Subtyping and expression typing

Well-formed types:

$$\Delta \vdash \texttt{Object} \text{ ok } [\text{WF-Object}]$$

$$\frac{\texttt{X} \in \text{dom}(\Delta)}{\Delta \vdash \texttt{X} \text{ ok}} \; [\text{WF-Var}]$$

$$\frac{\Delta \vdash \overline{\texttt{T}} \text{ ok} \qquad \Delta \vdash \overline{\texttt{T}} <: [\overline{\texttt{X}} \mapsto \overline{\texttt{T}}]\overline{\texttt{N}} \qquad CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{> } \lhd \texttt{ N \{ ... \}}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \text{ ok}} \; [\text{WF-Class}]$$

Valid method overriding:

$$\frac{\mathit{mtype}(\texttt{m}, \texttt{N}) = \texttt{<}\overline{\texttt{Z}} \lhd \overline{\texttt{Q}}\texttt{>}\overline{\texttt{U}} \rightarrow \texttt{U}_0 \qquad \texttt{implies } \overline{\texttt{P}}, \overline{\texttt{T}} = (\overline{\texttt{Q}}, \overline{\texttt{U}}) \qquad \overline{\texttt{Y}} <: \overline{\texttt{P}} \vdash \texttt{T}_0 <: \texttt{U}_0}{\mathit{override}(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0)}$$

Method typing:

$$\frac{\begin{array}{c}\Delta = \overline{\texttt{X}} <: \overline{\texttt{N}}, \overline{\texttt{Y}} <: \overline{\texttt{P}} \qquad \Delta \vdash \overline{\texttt{T}}, \texttt{T}, \overline{\texttt{P}} \text{ ok} \qquad \Delta; \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C<}\overline{\texttt{X}}\texttt{>} \vdash \texttt{e}_0 \in \texttt{S} <: \texttt{T} \\ CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{> } \lhd \texttt{ N\{ ... \}} \qquad \mathit{override}(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{> } \overline{\texttt{T}} \rightarrow \texttt{T})\end{array}}{\texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{return e}_0\texttt{;\}} \text{ ok IN C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>}} \; [\text{T-Method}]$$

Class typing:

$$\frac{\begin{array}{c}\overline{\texttt{X}} <: \overline{\texttt{N}} \vdash \overline{\texttt{N}}, \texttt{N}, \overline{\texttt{T}} \text{ ok} \qquad \overline{\texttt{M}} \text{ ok IN C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \\ \mathit{fields}(\texttt{N}) = \overline{\texttt{T}'}\ \overline{\texttt{f}'} \qquad \overline{\texttt{f}'} \cap \overline{\texttt{f}} = \varnothing \qquad \overline{\texttt{X}} <: \overline{\texttt{N}};\ \texttt{this:C<}\overline{\texttt{X}}\texttt{>} \vdash \overline{\texttt{v}} \in \overline{\texttt{S}} <: \overline{\texttt{T}}\end{array}}{\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{> } \lhd \texttt{ N \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{ =}\overline{\texttt{v}}\texttt{; } \overline{\texttt{M}}\texttt{\}} \text{ ok}} \; [\text{T-Class}]$$

Figure 9: Core well-formedness rules

$$\frac{\varnothing \vdash \mathtt{N} <: \mathtt{P}}{\mathtt{(P)(new\ N(\overline{v}))} \mapsto \mathtt{new\ N(\overline{v})}} \text{ [E-Cast]}$$

$$\frac{\mathit{mbody}(\mathtt{m<\overline{P}>}, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e})}{\mathtt{(new\ N(\overline{v})).m<\overline{P}>(\overline{v'})} \mapsto [\overline{\mathtt{x}} \mapsto \overline{\mathtt{v'}}, \mathtt{this} \mapsto \mathtt{new\ N(\overline{v})}]\mathtt{e}} \text{ [E-Invk]}$$

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad i \leq |\overline{\mathtt{v}}|}{(\mathtt{new\ N(\overline{v})}).\mathtt{f}_i \mapsto \mathtt{v}_i} \text{ [E-Field]}$$

$$\frac{\mathit{fieldval}(\mathtt{f}_i, \mathtt{N}) = \mathtt{v} \qquad |\overline{\mathtt{v}}| < i}{(\mathtt{new\ N(\overline{v})}).\mathtt{f}_i \mapsto \mathtt{v}} \text{ [E-Default]}$$

$$\frac{\mathtt{e} \mapsto \mathtt{e}'}{(\mathtt{N})\mathtt{e} \mapsto \mathtt{e}'} \text{ [EC-Cast]}$$

$$\frac{\mathtt{e} \mapsto \mathtt{e}'}{\mathtt{e.f} \mapsto \mathtt{e}'.\mathtt{f}} \text{ [EC-Field]}$$

$$\frac{\mathtt{e} \mapsto \mathtt{e}'}{\mathtt{e.m<\overline{N}>}(\overline{\mathtt{e}}) \mapsto \mathtt{e}'.\mathtt{m<\overline{T}>}(\overline{\mathtt{e}})} \text{ [EC-Invk-Recv]}$$

$$\frac{\mathtt{e}_i \mapsto \mathtt{e}'_i}{\mathtt{v.m<\overline{N}>}(\overline{\mathtt{v}}, \mathtt{e}_i, \overline{\mathtt{e}}) \mapsto \mathtt{v.m<\overline{N}>}(\overline{\mathtt{v}}, \mathtt{e}'_i, \overline{\mathtt{e}})} \text{ [EC-Invk-Arg]}$$

$$\frac{\mathtt{e}_i \mapsto \mathtt{e}'_i}{\mathtt{new\ N}(\overline{\mathtt{v}}, \mathtt{e}_i, \overline{\mathtt{e}}) \mapsto \mathtt{new\ N}(\overline{\mathtt{v}}, \mathtt{e}'_i, \overline{\mathtt{e}})} \text{ [EC-New-Arg]}$$

Figure 10: Core evaluation rules

Method type matching

$$\frac{\begin{array}{c} \mathit{equals}(\overline{\mathtt{N}}, \overline{\mathtt{P}}) = \overline{\Sigma} = \Sigma_1 \\ \mathit{equals}(\overline{\mathtt{T}}, \overline{\mathtt{U}}) = \overline{\Sigma'} = \Sigma_2 \qquad \mathit{matches}(\mathtt{T}, \mathtt{U}) = \Sigma'' \qquad \Sigma = \Sigma_1, \Sigma_2, \Sigma'' \qquad \Sigma \text{ consistent} \end{array}}{\mathit{matches}(\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\ \overline{\mathtt{T}} \to \mathtt{T}, \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}} \to \mathtt{U}) = \Sigma}$$

Method Type Wellformedness

$$\frac{\Delta, \overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \overline{\mathtt{T}}, \mathtt{T} \text{ ok}}{\Delta \vdash \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\ \overline{\mathtt{T}} \to \mathtt{T} \text{ ok}} \text{ [MTOK]}$$

Method Type Minimal Context

$$\frac{\Delta \vdash \mathtt{MT} \text{ ok} \qquad \mathit{dom}(\Delta) = FV(\mathtt{MT}) \qquad \forall \mathtt{X} \in \mathit{dom}(\Delta),\ \Delta(\mathtt{X}) = \mathtt{Object}}{\Delta \vdash \mathtt{MT} \text{ minok}}$$

Method Type Subtyping

$$\frac{\Delta, \overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash (\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\ \overline{\mathtt{T}} \to \mathtt{T}) <: (\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\ \overline{\mathtt{T}} \to \mathtt{U})} \text{ [MTSub]}$$

Computation:

$$\frac{\mathit{mtype}(\mathtt{m}_i, \mathtt{N}) \textit{ is undefined}}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{v}} \text{ [E-MFBase]}$$

$$\frac{\mathit{mtype}(\mathtt{m}_i, \mathtt{N}) = \mathtt{MT}_i \qquad \mathit{matches}(\mathtt{MT}_i, \mathtt{MT}) = \Sigma}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{methfold}_{i+1}(\mathtt{x} = [\mathtt{x} \mapsto \mathtt{v}, \mathtt{m_x} \mapsto m_i]\Sigma(\mathtt{e}); \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}} \text{ [E-MFMatch]}$$

$$\frac{\mathit{mtype}(\mathtt{m}_i, \mathtt{N}) = \mathtt{MT}_i \qquad \mathit{matches}(\mathtt{MT}_i, \mathtt{MT}) \text{ is not defined}}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{methfold}_{i+1}(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}} \text{ [E-MFSkip]}$$

$$\frac{\mathtt{e} \mapsto \mathtt{e}'}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}_0 \mapsto \mathtt{methfold}_i(\mathtt{x} = \mathtt{e}'; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}_0} \text{ [E-MFCong]}$$

Static semantics:

$$\frac{\mathtt{T} \ll: \{\mathtt{MT}\ \mathtt{m_x}\} \in \Delta}{\Delta \vdash \mathtt{T} \ll: \{\mathtt{MT}\ \mathtt{m_x}\}} \text{ [RM-Hyp]} \qquad \frac{\Delta \vdash \mathtt{U} \ll: \{\mathtt{MT}\ \mathtt{m_x}\} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{T} \ll: \{\mathtt{MT}\ \mathtt{m_x}\}} \text{ [RM-Trans]}$$

$$\frac{\begin{array}{c} i > 0 \qquad \Delta' \vdash \mathtt{MT} \text{ minok} \qquad \Delta \vdash \mathtt{T} \text{ ok} \\ \Delta; \Gamma \vdash \mathtt{e} \in \mathtt{U}'' <: \mathtt{U} \qquad \Delta, \Delta', \mathtt{T} \ll: \{\mathtt{MT}\ \mathtt{m_x}\}; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \end{array}}{\Delta; \Gamma \vdash \mathtt{methfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{T})\ \mathtt{e}' \in \mathtt{U}} \text{ [T-MethFold]}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}_0 \qquad \Delta \vdash \overline{\mathtt{T}} \text{ ok} \\ \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}} \qquad \Delta \vdash \mathtt{T}_0 \ll: \{(\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U})\ \mathtt{m_x}\} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}} \end{array}}{\Delta; \Gamma \vdash \mathtt{e.m_x}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}}) \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}} \text{ [T-InvkVar]}$$

Figure 11: Method folding