Variable-Arity Generic Interfaces

T. Stephen Strickland, Richard Cobbe, and Matthias Felleisen

College of Computer and Information Science
Northeastern University
Boston, MA 02115
sstrickl@ccs.neu.edu

Abstract. Many programming languages provide variable-arity functions. Such functions consume a fixed number of required arguments plus an unspecified number of "rest arguments." The C++ standardization committee has recently lifted this flexibility from terms to types with the adoption of a proposal for variable-arity templates. In this paper we propose an extension of Java with variable-arity interfaces. We present some programming examples that can benefit from variable-arity generic interfaces in Java; a type-safe model of such a language; and a reduction to the core language.

1 Introduction

In April of 2007 the C++ standardization committee adopted Gregor and Järvi's proposal [1] for variable-length type arguments in class templates, which presented the idea and its implementation but did not include a formal model or soundness proof. As a result, the relevant constructs will appear in the upcoming C++09 draft. Demand for this feature is not limited to C++, however. David Hall submitted a request for variable-arity type parameters for classes to Sun in 2005 [2].

For an illustration of the idea, consider the remarks on first-class functions in Scala [3] from the language's homepage. There it says that "every function is a value. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and supports currying." To achieve this integration of objects and closures, Scala's standard library pre-defines ten interfaces (traits) for function types, which we show here in Java-like syntax:

```
interface Function0<Result> {
 Result apply();
}

interface Function1<Arg1,Result> {
 Result apply(Arg1 a1);
}

interface Function2<Arg1,Arg2,Result> {
```

```
 Result apply(Arg1 a1, Arg2 a2);
}
...
```

A cursory glance shows that these predefined interfaces differ only in the number of function arguments. They accept between zero and nine arguments, which means that an attempt to create a closure of ten arguments fails. If we think of generic interfaces as functions from types to types, Scala's problem is obviously one of a missing mechanism for abstracting over a variable number of types. In general, while many programming languages—including object-oriented languages—support variable-arity methods, their designers fail to recognize that such a construct is as useful at the type level as it is at the value level.

With this paper, we propose to add variable-arity abstractions at the type level to Java [4]. In contrast to the C++ approach, we design generic interfaces, not generic classes. Furthermore, we develop a formal model and show that our approach is type-sound; that it can be type-checked independently (not via template expansion à la C++); and that it is compatible with existing Java.

In sections 2 and 3, we provide an informal overview of the proposed extension and present some examples that illustrate the use of this new mechanism. We also argue that interfaces are the natural mechanism for introducing variabble-arity type abstractions. Then, in section 4 we introduce FlexiJava, a sound model of a Java-like language that includes a variable-arity mechanism for type parameters. We also prove in section 5 that a compilation back to an ordinary Java-like language exists. We conjecture that our work applies equally to other object-oriented languages with generics such as C# [5, 6] and Scala.

2 Overview

Java programmers can introduce type parameters for interfaces and classes. While it is in principle possible to change both to accommodate variable-arity type abstraction, our exploration suggests that interfaces are the more natural starting point in Java while still providing the power to handle most of the examples from Gregor and Järvi's proposal. The first two subsections sketch a syntax for writing down variable-length type parameter lists and their uses. By the third subsection, we have enough material to discuss the rationale for our design choice.

2.1 Variable-length type parameter lists

Extending Java with variable-arity type parameter lists requires two changes to the syntax. The first is a change to the interface header. With this addition, a programmer may use a new kind of parameter specification for the last position:

```
interface I<A extends S, B extends T ...>
```

Following the tradition of languages like Lisp and Scheme, we refer to this as the *rest* argument.

The natural interpretation of this specification is as a family (or function) of related interfaces. The dots are a natural notation for this purpose, suggesting an abbreviation. Spelling out the abbreviations creates natural, finite versions of variable-arity interfaces:

```
interface I<A extends S>
interface I<A extends S, B1 extends T>
interface I<A extends S, B1 extends T, B2 extends T>
...
```

That is, a member of this family of interfaces takes at least one type (bounded by `S`) plus an arbitrary number of additional types, each bounded by `T`. All members of this family share the same name, `I`, much like overloaded methods use the same name for a family of methods.

2.2 References to rest type parameters

The body of a variable-arity interface may mention the sequence of types in two different ways:

1. in the type parameters of a generic class or interface as `B ...`, which refers to the entire sequence of types; and
2. in the formal parameter list of a method signature as `B x, ...`, which denotes a list of formal parameters of these types.

Each kind of occurrence immediately suggests a useful example, and we consider those in the following section. Before doing so, let's quickly look at some abstract examples.

The following pair of interface definitions (with dashes for irrelevant details) employ variable-arity type parameter lists:

```
interface J<F extends U ...> { ___ }

interface I<A extends U, B extends U ...>
    extends J<A, B ...> { ___ }
```

While the first one is parameterized over just a rest-argument type sequence, the second one takes at least one type. In addition, the second one re-uses its type parameters to extend the first interface. This is just one way to re-use the entire sequence of types at once; there are many others.

The second example shows how a sequence of types can be used to specify a flexible method signature:

```
interface K<A extends T ...> {
    void m(A x, ...);
}
```

This interface takes an arbitrary number of types. It introduces a method that takes the same number of arguments with the corresponding types. The following two classes are legitimate implemenations of `K`:

```
class CK implements K<Integer, Boolean> {
    public void m(Integer i, Boolean b) { return ; }
}

class DK implements K<List<Integer>> {
    public void m(List<Integer> li) { return ; }
}
```

2.3 Rationale

Our proposal diverges from that for C++ in that it provides only variable-arity interfaces, not classes. This provides almost as much expressive power as in C++ while avoiding significant problems with name generation.

Let's analyze this difference with an example. Assume an extension of our proposal that allows variable-arity generic classes:

```
class C<A extends T ...> {
    private A fd ...;

    public C(A fd_in ...) { ___ }
}
```

In the natural interpretation of this class definition, the class `C<T1, T2, T3>` contains three fields, of types `T1`, `T2`, and `T3`, respectively. The question is what these fields are called and how the rest of the class can refer to them. Clearly, because they cannot all be named `fd`, the compiler has to synthesize field names. This synthesis must use a name creation protocol, so that the programmer can predict the names and write code that refers to these fields.[1]

Additionally, because this class contains one field for each type argument, `C`'s methods cannot make any assumptions about the number of fields in the class. As a result, it is impossible to write the simplest of methods, like an implementation of `equals()` that compares all of the object's fields with those of the argument:

```
public boolean equals(C<A...> rhs) {
    return (fd.equals(rhs.fd) && ___
    // Impossible to finish!  (See text below.)
```

The author of such a method cannot even know how many conjuncts the argument to `return` requires. In other words, allowing field declarations like those in `C` above would require a new construct for processing indeterminate numbers of fields.

[1] The protocol must satisfy basic "hygiene standards" [7], i.e., avoid accidental variable capture, and this requires significant additional complexity.

The example's constructor suffers from the same problems. In `C<T1, T2, T3>`, the constructor would have three parameters of the same types as the fields. Again, though, the language would have to name those parameters in a precisely-defined fashion, and it would have to provide a mechanism to allow the constructor to manipulate an unknown number of formal parameters.

Limiting variable-arity generics to interfaces elegantly avoids these problems. Java interfaces may contain only constant definitions and method signatures. Since constant definitions must include an initial value, specifying a variable number of constants (as in the class above) is not meaningful. While method signatures in interfaces may contain arguments associated with the type rest argument in our proposal, method bodies may not appear, so we avoid the naming problem. While it is true that any class which implements the interface must provide a body for such a signature, the class always implements the interface *as applied to concrete type arguments,* so the class author can (indeed, must) write out each of the method's formal parameters explicitly, with its name, as our examples `K`, `CK`, and `DK` of the previous section demonstrate.

While the C++ proposal focuses on the definition of concrete classes with rest type parameters, it too avoids the generation of new variable names by restricting the ways in which the programmer may use those type parameters. It is, for instance, impossible to introduce fields whose type is specified entirely by the rest type parameter. While it is possible to define concrete methods whose signature uses the rest type argument, as with the constructor in our `C<T...>` example above, their proposal effectively treats the corresponding formal parameter as a heterogeneous *value* rest parameter and limits its use specifically to avoid the naming problems.

Our proposal does not include the ability to use the rest type argument in this fashion. Supporting this feature would require additional changes to the language, involving modifications to overload resolution as well as introducing a C++-like template specialization mechanism. This would allow deconstructing this heterogeneous value rest parameter by extracting its first element and recurring on the rest. We leave this functionality to further research.

3 Examples

Variable-arity type arguments for interfaces allow for a natural integration of higher-order functions with an object-oriented programming language. In addition, they enable a natural encoding of tuples. For additional examples, see Gregor and Järvi's paper [1]; all but one of those are expressible within our proposed Java extension.

3.1 Functions

Given variable-arity interfaces, a Scala-style integration of functional and object-oriented programming is straightforward. A programmer can now specify a completely general function type:

```
class Fact implements Function<Integer, Integer> {
    Integer apply(Integer n) {
        if(n == 0)
            return 1;
        else
            return n * this.apply(n - 1);
    }
}
```

Fig. 1. Implementation of factorial

```
new Function<Integer, Integer>() {
    Integer apply(Integer n) {
        if(n == 0)
            return 1;
        else
            return n * this.apply(n - 1);
    }
}
```

Fig. 2. Factorial as an anonymous class

```
interface Function<Result, Domain ...> {
    Result apply(Domain d, ...);
}
```

This `Function` interface requires a single method, called `apply`, from an implementing class. The result type of the method is the first, non-optional type in the parameter list; the input types are the remaining types from the rest argument.

Any class that implements this `Function` interface for a given sequence of domain types must contain an apply method that takes in that many arguments, of those types, in order. Figure 1 presents an implementation of the factorial function that implements an appropriate instantiation of the `Function` interface, and figure 2 shows `Factorial` written as an anonymous inner class, similar to anonymous functions in functional languges.

In a language with autoboxing, which both Java and C# now implement, using `Factorial` is almost as natural as in a functional language:
`new Factorial().apply(`n`)` computes the factorial of n. With Scala's syntactic sugar for functions, variable-arity interfaces thus create a complete and smooth integration of functional and object-oriented programming.[2]

3.2 Tuples

Scala also includes interfaces for tuples, much like those for first-class functions. In contrast to the `Function` interface, however, it isn't obvious how variable-arity

[2] The integration of higher-order functions poses additional problems, which are completely orthogonal to variable-arity type parameters.

interfaces can help with tuples. While the flexible type parameter list allows a concise specification of a tuple's component types, the real problem is the `select` operation on tuples, which given an index i returns the ith element from a tuple:

```
interface Tuple<Element ...> {
    ??? select(Integer i);
}
```

Because a tuple is heterogeneous, there is no single return type for `select`. Worse, not all integers denote valid element indices.

With variable-arity parameter lists, we can provide a tuple-like structure using the usual theoretical trick of a case-like function:

```
interface Tuple<Element ...> {
    <S> S select(Function<S, Element ...> f);
}
```

That is, the `select` method takes a function that takes all the elements of the tuple. A benefit of this scheme is that this function can perform operations on multiple elements of the tuple at once. It is also easy to translate a more natural syntax for tuples which may be more programmer-friendly into this interface.

Here is one specific implementation of `Tuple`:

```
new Tuple<Integer,Boolean>() {
    Integer x = 3;
    Boolean y = true;
    <S> S select(Function<S, Integer, Boolean> f) {
        return f.apply(x, y);
    }
}
```

This anonymous class pairs an `Integer` with a `Boolean`.

4 The FlexiJava Model

Our theoretical work has two goals. First, we wish to prove that extending Java with variable-arity generic interfaces is type sound. Second, we wish to demonstrate that the extension is compatible with the existing Java compilation and run-time framework.

In support of these goals, we introduce two models: Interface Java (IJ) and FlexiJava. Work on both starts with the observation that our theory clearly needs a Java model that includes generic types as well as interfaces. We therefore synthesize Interface Java (IJ) from Featherweight Generic Java (FGJ) [8] and ClassicJava [9] and use it as the starting point of our explorations.

In the first subsection, we introduce IJ, its syntax, and its semantics. The second subsection concerns FlexiJava, the extension of IJ with variable-arity interfaces, covering syntax, static semantics, and dynamic semantics. The third section establishes type soundness.

$$
\begin{array}{lll}
\Pi & ::= & \overrightarrow{\texttt{L}}\ \texttt{e} \\
\texttt{T} & ::= & \alpha \mid \texttt{N} \\
\texttt{N} & ::= & \texttt{C<}\overrightarrow{\texttt{T}}\texttt{>} \\
\texttt{L} & ::= & \texttt{class C <}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> ext N impl }\overrightarrow{\texttt{N}}\texttt{ \{ }\overrightarrow{\texttt{T f;}}\ \overrightarrow{\texttt{M}}\texttt{ \}} \\
 & \mid & \texttt{ifc C <}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> ext }\overrightarrow{\texttt{N}}\texttt{ \{ }\overrightarrow{\texttt{MS}}\texttt{ \}} \\
\texttt{M} & ::= & \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> T m(}\overrightarrow{\texttt{T x}}\texttt{) \{ e \}} \\
\texttt{MS} & ::= & \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> T m(}\overrightarrow{\texttt{T x}}\texttt{)} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m<}\overrightarrow{\texttt{T}}\texttt{>(}\overrightarrow{\texttt{e}}\texttt{)} \mid \texttt{new N(}\overrightarrow{\texttt{e}}\texttt{)} \mid \texttt{(N) e} \\
 & \mid & \texttt{let }\overrightarrow{\texttt{N x = e}}\texttt{ in e} \\
\texttt{MT} & ::= & \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>}\overrightarrow{\texttt{T}} \rightarrow \texttt{T} \quad \text{(method type)} \\
\texttt{C, D, E} & ::= & \text{class or interface name} \\
\texttt{S, U, V} & ::= & \text{type (T)} \\
\texttt{P, Q, R} & ::= & \text{type application (N)}
\end{array}
$$

Fig. 3. IJ syntax

4.1 Interface Java (IJ)

IJ's syntax extends FGJ's syntax with Java-style interfaces and `let` expressions. The latter model Java-style local variable bindings, which we need to translate FlexiJava to Java 1.5. Consistent with Java's semantics for local bindings, the scope of a `let`-bound identifier includes the body of the `let` as well as the right-hand side of all *subsequent* binding clauses in the `let`.

Figure 3 defines the exact syntax of IJ. We abbreviate the Java keywords `interface`, `extends`, and `implements` as `ifc`, `ext`, and `impl`. The grammar includes the nonterminal `MT` for use in the typing rules and supporting relations.

Figure 4 summarizes the model's type judgments and environments. In the statement

$$\mathit{methDecl}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}(\texttt{m}) = (\texttt{MT}, \overrightarrow{\texttt{x}}, \texttt{e})$$

the types $\overrightarrow{\texttt{T}}$ have already been substituted for `C`'s type parameters in `MT` and `e`; the same holds for $\mathit{imethDecl}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}$, $\mathit{meth}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}$, and $\mathit{imeth}_{\Pi,\Delta,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}$.

The introduction of interfaces requires consistency checks for programs to be well-typed. Consider the following declaration:

```
interface I1 extends I2, I3, I4 { }
```

If more than one of `I2`, `I3`, and `I4` define a particular method `m`, then these declarations must be consistent. Specifically, they must all take the same arguments (in the absence of method overloading), and one of the methods must have a return type that is a subtype of all of the other return types (in keeping with the fact that Java now allows covariance in method return types). Otherwise, the type of the method `m` in `I1` is not well-defined. If, on the other hand, `I1` were to override the definition of `m`, then normal covariance would apply and no additional consistency checks would be necessary.[3]

[3] At time of writing, Sun's Java compiler implements a slight variation of this requirement, in which the superinterfaces' definitions of `m` must be consistent even

Environments:

Δ	maps type variables to their bounds
Γ	maps variables to their types

Relations:

$\texttt{S} <:_{\Pi\,\Delta} \texttt{T}$	S is a subtype of T
$\texttt{C} \trianglelefteq_{\Pi} \texttt{D}$	C is a subclass of D, ignoring type arguments
$methDecl_{\Pi,\texttt{C<}\vec{\texttt{T}}\texttt{>}}(\texttt{m}) = (\texttt{MT}, \vec{\texttt{x}}, \texttt{e})$	method m is declared (not inherited) in class $\texttt{C<}\vec{\texttt{T}}\texttt{>}$ with type MT, arguments $\vec{\texttt{x}}$, and body e
$meth_{\Pi,\texttt{C<}\vec{\texttt{T}}\texttt{>}}(\texttt{m}) = (\texttt{MT}, \vec{\texttt{x}}, \texttt{e})$	method m is contained in class $\texttt{C<}\vec{\texttt{T}}\texttt{>}$ with type MT, arguments $\vec{\texttt{x}}$, and body e
$imethDecl_{\Pi,\texttt{C<}\vec{\texttt{T}}\texttt{>}}(\texttt{m}) = \texttt{MT}$	method m is declared (not inherited) in interface $\texttt{C<}\vec{\texttt{T}}\texttt{>}$ with type MT
$imeth_{\Pi,\Delta,\texttt{C<}\vec{\texttt{T}}\texttt{>}}(\texttt{m}) = \texttt{MT}$	method m is contained in interface $\texttt{C<}\vec{\texttt{T}}\texttt{>}$ with type MT
$consIfcs(\Pi, \Delta, \texttt{N}, \vec{\texttt{P}})$	The interfaces $\vec{\texttt{P}}$, the immediate super-interfaces of N, are consistent
$dcast(\Pi, \texttt{C}, \texttt{D})$	it is safe to cast an instance of D to C, ignoring type arguments
$override(\Pi, \Delta, \texttt{m}, \texttt{N}, \texttt{MT})$	a subtype of N may override method m with type MT (allowing covariance in the return type)

Functions:

$fields(\Pi, \texttt{N})$	constructs a sequence of all fields in class N with their types
$bound(\Delta, \texttt{T})$	maps type T to its bound

Type judgments:

$\vdash \Pi : \texttt{T}$	program Π has type T
$\Pi, \Delta \vdash \texttt{T}$	type T is well-formed in program Π
$\Pi \vdash \texttt{L}$	class definition L is well-typed in Π
$\Pi, \texttt{C} \vdash \texttt{M}$	method definition M is well-typed in Π and class C
$\Pi, \Delta, \Gamma \vdash \texttt{e} : \texttt{T}$	expression e has exactly type T under Π, Δ, Γ

Fig. 4. IJ judgments and relations

While IJ is reasonably complete, its generics do not fully describe Java's. First, Java allows type variable to serve as bounds, so one could write

```
class C<X extends Object, Y extends X>···
```

IJ's syntax, however, requires bounds on type variables to be type applications, not type variables, a restriction inherited from FGJ. Second, IJ, also like FGJ, does not include support for Java's wildcard types and associated features.

Static and Dynamic Semantics For space reasons we do not include the full static semantics, but we do show subtyping rules in figure 5 and rules for method

when I1 overrides m. Sun has acknowledged that the consistency check described above is the intended behavior and opened a bug report against the compiler (bug #6294779) [Personal communication with Gilad Bracha and Peter von der Ahé, July 2005].

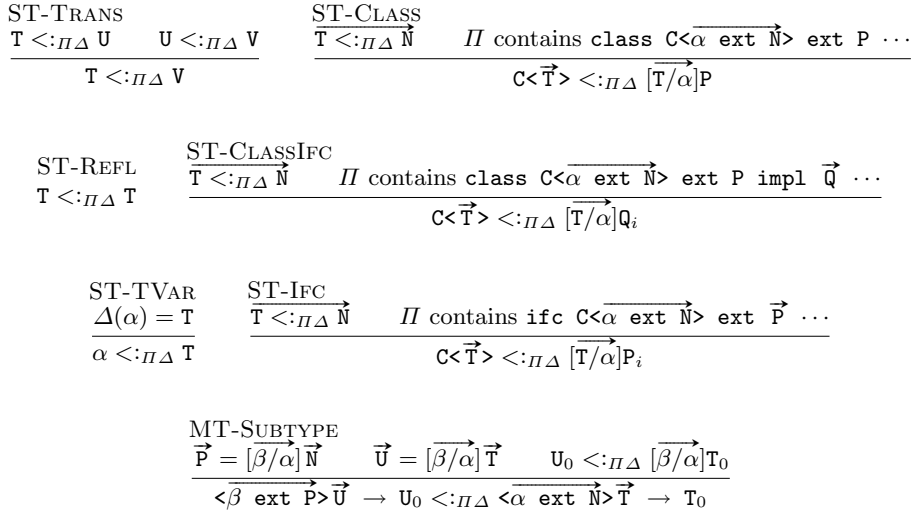

Fig. 5. Subtyping in IJ

lookup in figure 6. For conciseness we extend the $<:_{\Pi\Delta}$ relation to cover method types as well.

IJ's operational semantics is defined as a rewriting system with evaluation contexts, similar to the one for Classic Java. Figure 7 defines evaluation contexts, values, and the reductions, which depend on the method lookup function and the subtyping relation; plugging in different ones produces a different semantics. (The bullet • denotes an empty sequence.)

Soundness In addition to creating the IJ model, we have proved that our extensions result in a sound type system via an application of the standard method [10, 9].

Theorem 1 (Type Soundness). *If* $\Pi = \vec{\mathtt{L}}\ \mathtt{e}$ *and* $\vdash \Pi : \mathtt{T}$*, then one of the following must be true:*

- $\Pi \vdash \mathtt{e} \rightarrow^*_{\mathrm{IJ}} \mathtt{v}$*, where* $\Pi, \emptyset, \emptyset \vdash \mathtt{v} : \mathtt{S}$ *and* $\mathtt{S} <:_{\Pi\emptyset} \mathtt{T}$*, or*
- $\Pi \vdash \mathtt{e} \Uparrow$*, or*
- $\Pi \vdash \mathtt{e} \rightarrow^*_{\mathrm{IJ}}$ *error: bad cast.*

We omit the proof of soundness for space reasons.

4.2 FlexiJava

We are now ready to extend IJ to produce FlexiJava, our model for variable-arity generics.

$$\textsc{MethIn-Class}\quad \frac{\Pi \text{ contains } \texttt{class C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>}\ \cdots\ \texttt{\{}\ \cdots\ \overrightarrow{\texttt{M}}\ \texttt{\}} \qquad \texttt{<}\overrightarrow{\beta\ \texttt{ext P}}\texttt{>}\ \texttt{S}_0\ \texttt{m(}\overrightarrow{\texttt{S x}}\texttt{)\ \{ e \}} \in \overrightarrow{\texttt{M}} \qquad \texttt{MT} = [\overrightarrow{\texttt{T}/\alpha}](\texttt{<}\overrightarrow{\beta\ \texttt{ext P}}\texttt{>}\overrightarrow{\texttt{S}}\ \rightarrow\ \texttt{S}_0)}{\mathit{methDecl}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}(\texttt{m}) = (\texttt{MT}, \overrightarrow{\texttt{x}}, [\overrightarrow{\texttt{T}/\alpha}]\texttt{e})}$$

$$\textsc{MethIn-Ifc}\quad \frac{\Pi \text{ contains } \texttt{ifc C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>}\ \cdots\ \texttt{\{}\ \overrightarrow{\texttt{MS}}\ \texttt{\}} \qquad \Pi, \Delta \vdash \texttt{C<}\overrightarrow{\texttt{T}}\texttt{>} \qquad \texttt{<}\overrightarrow{\beta\ \texttt{ext P}}\texttt{>}\ \texttt{S}_0\ \texttt{m(}\overrightarrow{\texttt{S x}}\texttt{, } \gamma\ldots\texttt{)} \in \overrightarrow{\texttt{MS}} \qquad \texttt{MT} = [\overrightarrow{\texttt{T}/\alpha}](\texttt{<}\overrightarrow{\beta\ \texttt{ext P}}\texttt{>}\overrightarrow{\texttt{S}}\texttt{, } \gamma\ldots\ \rightarrow\ \texttt{S}_0)}{\mathit{imethDecl}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}(\texttt{m}) = \texttt{MT}}$$

$$\textsc{MD-ClassImm}\quad \frac{\mathit{methDecl}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}(\texttt{m}) = (\texttt{MT}, \overrightarrow{\texttt{x}}, \texttt{e})}{\mathit{meth}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}(\texttt{m}) = (\texttt{MT}, \overrightarrow{\texttt{x}}, \texttt{e})} \qquad \textsc{MD-IfcImm}\quad \frac{\mathit{imethDecl}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}(\texttt{m}) = \texttt{MT}}{\mathit{imeth}_{\Pi,\Delta,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}(\texttt{m}) = \texttt{MT}}$$

$$\textsc{MD-Superclass}\quad \frac{\Pi \text{ contains } \texttt{class C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> ext P}\ \cdots\ \texttt{\{}\ \cdots\ \overrightarrow{\texttt{M}}\ \texttt{\}} \qquad \texttt{m} \notin \overrightarrow{\texttt{M}} \qquad \mathit{meth}_{\Pi,[\overrightarrow{\texttt{T}/\alpha}]\texttt{P}}(\texttt{m}) = (\texttt{MT}, \overrightarrow{\texttt{x}}, \texttt{e})}{\mathit{meth}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}(\texttt{m}) = (\texttt{MT}, \overrightarrow{\texttt{x}}, \texttt{e})}$$

$$\textsc{MD-SuperIfc}\quad \frac{\Pi \text{ contains } \texttt{ifc C <}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> ext}\ \overrightarrow{\texttt{P}}\ \texttt{\{}\ \overrightarrow{\texttt{MS}}\ \texttt{\}} \qquad \texttt{m} \notin \overrightarrow{\texttt{MS}} \qquad \texttt{MT} = \min_{\Pi\Delta}\{\texttt{MT}' \mid \mathit{imeth}_{\Pi,\Delta,[\overrightarrow{\texttt{T}/\alpha}]\texttt{P}_i}(\texttt{m}) = \texttt{MT}', \texttt{P}_i \in \overrightarrow{\texttt{P}}\}}{\mathit{imeth}_{\Pi,\Delta,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}(\texttt{m}) = \texttt{MT}}$$

Fig. 6. IJ method lookup

The Abstract Syntax FlexiJava uses the same form of type judgments and environments as IJ (figure 4), though the full definitions differ due to the introduction of variable-arity generics. Figure 8 defines the abstract syntax of FlexiJava. An ellipsis on the baseline (...) always represents a literal token from the program text; we use centered dots ($\cdots$) to indicate an omission from the program text.

As in IJ, the class `Object` does not appear in the program directly; our model treats this class specially. Like Java, the model treats interfaces with no direct superinterface as a subtype of `Object` as in Java. Thus a class or interface may use `Object` as a type variable's bound to indicate that any type is acceptable for that variable, just as in IJ.

Static Semantics FlexiJava's type system is based on IJ's, with extensions to handle the new kinds of interface definitions and method signatures. The differences primarily concern the well-formedness of types and the definitions of subtyping. The judgments have the same form as in IJ, although in contexts

$$
\begin{array}{rcl}
\texttt{v} & ::= & \texttt{new N(}\vec{\texttt{v}}\texttt{)} \\
\mathcal{E} & ::= & [\,] \mid \mathcal{E}\texttt{.f} \mid \mathcal{E}\texttt{.m<}\vec{\texttt{N}}\texttt{>(}\vec{\texttt{e}}\texttt{)} \mid \texttt{v.m<}\vec{\texttt{N}}\texttt{>(}\vec{\texttt{v}}\ \mathcal{E}\ \vec{\texttt{e}}\texttt{)} \\
 & & \mid \texttt{new N(}\vec{\texttt{v}}\ \mathcal{E}\ \vec{\texttt{e}}\texttt{)} \mid \texttt{(N)}\mathcal{E} \mid \texttt{let N x =}\ \mathcal{E}\ \overrightarrow{\texttt{N x = e}}\ \texttt{in e}
\end{array}
$$

$$\Pi \vdash \mathcal{E}[\texttt{new N(}\vec{\texttt{v}}\texttt{).f}_i] \rightarrow_{\text{IJ}} \mathcal{E}[\texttt{v}_i] \qquad \text{R-Field}$$
where $\mathit{fields}(\Pi, \texttt{N}) = \overrightarrow{\texttt{T f}}$

$$\Pi \vdash \mathcal{E}[\texttt{new N(}\vec{\texttt{v}}\texttt{).m<}\vec{\texttt{T}}\texttt{>(}\vec{\texttt{u}}\texttt{)}] \rightarrow_{\text{IJ}} \mathcal{E}\left[\overrightarrow{[\texttt{u/x}}, \texttt{new N(}\vec{\texttt{v}}\texttt{)/this}, \overrightarrow{\texttt{T}/\beta}]\texttt{e}\right] \qquad \text{R-Call}$$
where $\mathit{meth}_{\Pi,\texttt{N}}(\texttt{m}) = (\texttt{<}\overrightarrow{\beta\ \texttt{ext P}}\texttt{>}\vec{\texttt{U}} \rightarrow \texttt{U}_0, \vec{\texttt{x}}, \texttt{e})$

$$\Pi \vdash \mathcal{E}[\texttt{(P) new N(}\vec{\texttt{v}}\texttt{)}] \rightarrow_{\text{IJ}} \mathcal{E}[\texttt{new N(}\vec{\texttt{v}}\texttt{)}] \qquad \text{R-Cast}$$
where $\texttt{N} <:_{\Pi\emptyset} \texttt{P}$

$$\Pi \vdash \mathcal{E}[\texttt{(P) new N(}\vec{\texttt{v}}\texttt{)}] \rightarrow_{\text{IJ}} \text{error: bad cast} \qquad \text{R-BadCast}$$
where $\texttt{N} \not<:_{\Pi\emptyset} \texttt{P}$

$$\Pi \vdash \mathcal{E}[\texttt{let} \bullet \texttt{in e}] \rightarrow_{\text{IJ}} \mathcal{E}[\texttt{e}] \qquad \text{R-EmptyLet}$$

$$\Pi \vdash \mathcal{E}[\texttt{let N x = v}\ \overrightarrow{\texttt{P y = e}}\ \texttt{in e}_0] \rightarrow_{\text{IJ}} \mathcal{E}\left[[\texttt{v/x}]\texttt{let}\ \overrightarrow{\texttt{P y = e}}\ \texttt{in e}_0\right] \qquad \text{R-Let}$$

Fig. 7. Dynamic semantics for IJ and FlexiJava

$$
\begin{array}{rcl}
\Pi & ::= & \vec{\texttt{L}}\ \texttt{e} \\
\texttt{T} & ::= & \alpha \mid \texttt{N} \\
\texttt{N} & ::= & \texttt{C<}\vec{\texttt{T}}\texttt{>} \mid \texttt{C<}\vec{\texttt{T}}\texttt{,}\ \alpha\texttt{...>} \\
\texttt{L} & ::= & \texttt{class C <}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> ext N impl}\ \vec{\texttt{N}}\ \texttt{\{}\ \overrightarrow{\texttt{T f;}}\ \vec{\texttt{M}}\ \texttt{\}} \\
 & & \mid \texttt{ifc C <}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> ext}\ \vec{\texttt{N}}\ \texttt{\{}\ \vec{\texttt{MS}}\ \texttt{\}} \\
 & & \mid \texttt{ifc C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{,}\ \alpha\ \texttt{ext N...> ext}\ \vec{\texttt{N}}\ \texttt{\{}\ \vec{\texttt{MS}}\ \texttt{\}} \\
\texttt{M} & ::= & \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> T m(}\overrightarrow{\texttt{T x}}\texttt{)\ \{ e \}} \\
\texttt{MS} & ::= & \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> T m(}\overrightarrow{\texttt{T x}}\texttt{)} \\
 & & \mid \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> T m(}\overrightarrow{\texttt{T x}}\texttt{,}\ \alpha\texttt{...)} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m<}\vec{\texttt{T}}\texttt{>(}\vec{\texttt{e}}\texttt{)} \mid \texttt{new N(}\vec{\texttt{e}}\texttt{)} \mid \texttt{(N) e} \\
 & & \mid \texttt{let}\ \overrightarrow{\texttt{N x = e}}\ \texttt{in e} \\
\texttt{MT} & ::= & \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>}\vec{\texttt{T}} \rightarrow \texttt{T} \mid \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>}\vec{\texttt{T}}\texttt{,}\ \alpha\ldots \rightarrow \texttt{T} \\
\texttt{C, D, E} & ::= & \text{class or interface name} \\
\texttt{S, U, V} & ::= & \text{type (T)} \\
\texttt{P, Q, R} & ::= & \text{type applications (N)}
\end{array}
$$

Fig. 8. FlexiJava syntax

$$\text{WF-TVar}\ \frac{\Delta(\alpha) = (\_, \mathrm{ff})}{\Pi, \Delta \vdash \alpha}$$

$$\text{WF-Class}\ \frac{\Pi \text{ contains } \texttt{class C <}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>} \cdots \quad \overrightarrow{\Pi, \Delta \vdash \mathtt{T}} \quad \overrightarrow{\mathtt{T} <:_{\Pi\Delta} \sigma(\mathtt{N})} \quad \text{where } \sigma(\mathtt{N}) = [\overrightarrow{\mathtt{T}/\alpha}]\mathtt{N}}{\Pi, \Delta \vdash \mathtt{C<}\overrightarrow{\mathtt{T}}\mathtt{>}}$$

$$\text{WF-TVar-Dots}\ \frac{\Delta(\alpha) = (\_, \mathrm{tt})}{\Pi, \Delta \vdash \alpha\ldots}$$

$$\text{WF-Ifc}\ \frac{\Pi \text{ contains } \texttt{ifc C <}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>} \cdots \quad \overrightarrow{\Pi, \Delta \vdash \mathtt{T}} \quad \overrightarrow{\mathtt{T} <:_{\Pi\Delta} \sigma(\mathtt{N})} \quad \text{where } \sigma(\mathtt{N}) = [\overrightarrow{\mathtt{T}/\alpha}]\mathtt{N}}{\Pi, \Delta \vdash \mathtt{C<}\overrightarrow{\mathtt{T}}\mathtt{>}}$$

$$\text{WF-IfcExt}\ \frac{\Pi \text{ contains } \texttt{ifc C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{,}\ \beta\ \texttt{ext P...>} \cdots \quad \overrightarrow{\Pi, \Delta \vdash \mathtt{T}} \quad (\overrightarrow{\mathtt{U}}, \overrightarrow{\mathtt{V}}) = \mathit{splitAt}(\overrightarrow{\mathtt{T}}, \#(\overrightarrow{\alpha})) \quad \overrightarrow{\mathtt{U} <:_{\Pi\Delta} \sigma(\mathtt{N})} \quad \overrightarrow{\mathtt{V} <:_{\Pi\Delta} \sigma(\mathtt{P})} \quad \text{where } \sigma(\mathtt{N}) = [\overrightarrow{\mathtt{U}/\alpha}, \overrightarrow{\mathtt{V}}/\beta\ldots]\mathtt{N}}{\Pi, \Delta \vdash \mathtt{C<}\overrightarrow{\mathtt{T}}\mathtt{>}}$$

$$\text{WF-IfcExtRest}\ \frac{\begin{array}{l}\Pi \text{ contains } \texttt{ifc C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{,}\ \beta\ \texttt{ext P...>} \cdots \quad \overrightarrow{\Pi, \Delta \vdash \mathtt{T}} \quad \Pi, \Delta \vdash \gamma\ldots \\ (\overrightarrow{\mathtt{U}}, \overrightarrow{\mathtt{V}}) = \mathit{splitAt}(\overrightarrow{\mathtt{T}}, \#(\overrightarrow{\alpha})) \quad \overrightarrow{\mathtt{U} <:_{\Pi\Delta} \sigma(\mathtt{N})} \quad \overrightarrow{\mathtt{V} <:_{\Pi\Delta} \sigma(\mathtt{P})} \quad \gamma <:_{\Pi\Delta} \sigma(\mathtt{P}) \\ \text{where } \sigma(\mathtt{N}) = [\overrightarrow{\mathtt{U}/\alpha}, (\overrightarrow{\mathtt{V}}, \gamma\ldots)/\beta\ldots]\mathtt{N}\end{array}}{\Pi, \Delta \vdash \mathtt{C<}\overrightarrow{\mathtt{T}}\mathtt{,}\ \gamma\ldots\mathtt{>}}$$

Fig. 9. Well-formed types

where we refer to both IJ and FlexiJava, we disambiguate with a subscript on the turnstile: $\vdash_{\mathrm{IJ}}$ and $\vdash_{\mathrm{FXJ}}$.

To determine if the type $\mathtt{C<}\overrightarrow{\mathtt{T}}\mathtt{>}$ is well-formed—i.e., the judgment $\Pi, \Delta \vdash \mathtt{C<}\overrightarrow{\mathtt{T}}\mathtt{>}$ is derivable—we must verify that all type arguments $\overrightarrow{\mathtt{T}}$ are well-formed, look up $\mathtt{C}$ in the program Π, and ensure that all type arguments $\overrightarrow{\mathtt{T}}$ satisfy the bounds specified by $\mathtt{C}$'s definition. Figure 9 defines this judgment formally.

If $\mathtt{C}$ is defined to be an interface with variable arity, then we must split the actual parameters into those that correspond to the fixed arguments and those that correspond to the rest argument. To do this, we use the *splitAt* function. We omit its formal definition; informally, $\mathit{splitAt}(\overrightarrow{\mathtt{T}}, n)$ returns a pair of lists, the first of which consists of the first n elements of $\overrightarrow{\mathtt{T}}$; the second result is the remainder of $\mathtt{T}$. If $\#(\overrightarrow{\mathtt{T}}) < n$, where $\#(\overrightarrow{\mathtt{T}})$ is the length of the sequence $\overrightarrow{\mathtt{T}}$, then $\mathit{splitAt}(\overrightarrow{\mathtt{T}}, n)$ is undefined.

To determine if $\mathtt{C<}\overrightarrow{\mathtt{T}}\mathtt{,}\ \alpha\ldots\mathtt{>}$ is well-formed, we use the same logic as above. In order for this type to make sense, it must appear somewhere within the definition of an interface that defines α to be a rest argument. To ensure this, we extend the mapping Δ slightly beyond what is needed for IJ; it now maps a type variable to a pair consisting of a type and either true or false (written tt

ST-FlexIfc

$$\frac{\Pi \text{ contains } \texttt{ifc C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{,}\ \beta\ \texttt{ext Q...> ext}\ \overrightarrow{\texttt{P}}\ \cdots \qquad (\overrightarrow{\texttt{U}}, \overrightarrow{\texttt{V}}) = \mathit{splitAt}(\overrightarrow{\texttt{T}}, \texttt{\#}(\overrightarrow{\alpha}))}{\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>} <:_{\Pi\Delta} [\overrightarrow{\texttt{U}/\alpha}, \overrightarrow{\texttt{V}}/\beta...]\texttt{P}_i}$$

ST-IfcDots

$$\frac{\Pi \text{ contains } \texttt{ifc C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{,}\ \beta\ \texttt{ext Q...> ext}\ \overrightarrow{\texttt{P}}\ \cdots \qquad (\overrightarrow{\texttt{U}}, \overrightarrow{\texttt{V}}) = \mathit{splitAt}(\overrightarrow{\texttt{T}}, \texttt{\#}(\overrightarrow{\alpha}))}{\texttt{C<}\overrightarrow{\texttt{T}}\texttt{,}\ \gamma...\texttt{>} <:_{\Pi\Delta} [\overrightarrow{\texttt{U}/\alpha}, (\overrightarrow{\texttt{V}}, \gamma...)/\beta...]\texttt{P}_i}$$

MT-SubtypeDots

$$\frac{\overrightarrow{\texttt{P}} = [\overrightarrow{\beta/\alpha}]\overrightarrow{\texttt{N}} \qquad \overrightarrow{\texttt{U}} = [\overrightarrow{\beta/\alpha}]\overrightarrow{\texttt{T}} \qquad \texttt{U}_0 <:_{\Pi\Delta} [\overrightarrow{\beta/\alpha}]\texttt{T}_0}{\texttt{<}\overrightarrow{\beta\ \texttt{ext P}}\texttt{>}\overrightarrow{\texttt{U}}\texttt{,}\ \gamma... \rightarrow \texttt{U}_0 <:_{\Pi\Delta} \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>}\overrightarrow{\texttt{T}}\texttt{,}\ \gamma... \rightarrow \texttt{T}_0}$$

Fig. 10. Additional subtyping rules in FlexiJava

and ff). The type is the variable's bound as with IJ; the boolean flag indicates whether or not the type variable was bound as a rest argument.

Determining whether one type is a subtype of another (figure 10) proceeds much as in IJ. FlexiJava adds rules to cover the additional cases of interfaces and types with rest args. In the case of the MT-SubtypeDots rule, the same type variable γ appears on both sides of the subtyping expression in the conclusion. This is intentional, as this rule is only used in the process of type-checking the definition of an interface with a rest argument. Since this relation is applied to the method types *after* the interface's type arguments have been substituted through, it is correct to require the same type variable in both places. Finally, there are additional rules (not shown) that guarantee that every class is a subtype of `Object` and that every interface without a direct superinterface is a subtype of `Object`.

Figure 11 specifies some of the rules for method lookup in FlexiJava. These rules are in addition to those detailed in figure 6 and they cover only some typical cases. There are variants of MethIn-IfcDots and MD-SuperIfcDots to deal with method signatures and interfaces with and without rest arguments in all feasible combinations.

Finally, figure 12 defines type checking for programs, as well as the supporting relations. Again, there is a variant of TIfc that handles the case where the interface is not defined with a rest argument. We omit these rules here because they are similar to those for FGJ and don't contribute any new insights.

Design Constraint The ConsistentRestArgs rule from this figure uses the *numArgs* function. For a program Π, method `m`, and class or interface `C`, $\mathit{numArgs}(\Pi, \texttt{m}, \texttt{C})$ returns a pair of numbers, indicating the number of fixed and rest arguments in the method's *original* definition(s) for `C`. (The number of rest arguments is always either zero or one.) By the method's original definitions,

$$\frac{\begin{array}{c}\textsc{MethIn-IfcDots}\\ \Pi \text{ contains } \texttt{ifc C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{,}\ \gamma\ \texttt{ext Q...>}\ \cdots\ \texttt{\{}\ \overrightarrow{\texttt{MS}}\ \texttt{\}} \qquad \Pi, \Delta \vdash \texttt{C<}\overrightarrow{\texttt{T}}\texttt{,}\ \delta\texttt{...>}\\ \texttt{<}\overrightarrow{\beta\ \texttt{ext P}}\texttt{>}\ \texttt{S}_0\ \texttt{m(}\overrightarrow{\texttt{S x}}\texttt{,}\ \gamma\texttt{...)} \in \overrightarrow{\texttt{MS}} \qquad (\overrightarrow{\texttt{U}}, \overrightarrow{\texttt{V}}) = \mathit{splitAt}(\overrightarrow{\texttt{T}}, \#(\overrightarrow{\alpha}))\\ \texttt{MT} = [\overrightarrow{\texttt{U}/\alpha}, (\overrightarrow{\texttt{V}}, \delta\texttt{...})/\gamma\texttt{...}](\texttt{<}\overrightarrow{\beta\ \texttt{ext P}}\texttt{>}\overrightarrow{\texttt{S}}\texttt{,}\ \gamma\texttt{...} \rightarrow \texttt{S}_0)\end{array}}{\mathit{imethDecl}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{,}\ \delta\texttt{...>}}(\texttt{m}) = \texttt{MT}}$$

$$\frac{\mathit{imethDecl}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{,}\ \alpha\texttt{...>}}(\texttt{m}) = \texttt{MT}}{\mathit{imeth}_{\Pi,\Delta,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{,}\ \alpha\texttt{...>}}(\texttt{m}) = \texttt{MT}}$$

$$\frac{\begin{array}{c}\textsc{MD-SuperIfcDots}\\ \Pi \text{ contains } \texttt{ifc C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{,}\ \beta\ \texttt{ext Q...> ext}\ \overrightarrow{\texttt{P}}\ \texttt{\{}\ \overrightarrow{\texttt{MS}}\ \texttt{\}}\\ \texttt{m} \notin \overrightarrow{\texttt{MS}} \qquad (\overrightarrow{\texttt{U}}, \overrightarrow{\texttt{V}}) = \mathit{splitAt}(\overrightarrow{\texttt{T}}, \#(\overrightarrow{\alpha}))\\ \texttt{MT} = \min_{\Pi\Delta}\{\texttt{MT}' \mid \mathit{imeth}_{\Pi,\Delta,[\overrightarrow{\texttt{U}/\alpha},(\overrightarrow{\texttt{V},\gamma\texttt{...}})/\beta\texttt{...}]\texttt{P}_i}(\texttt{m}) = \texttt{MT}', \texttt{P}_i \in \overrightarrow{\texttt{P}}\}\end{array}}{\mathit{imeth}_{\Pi,\Delta,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{,}\ \gamma\texttt{...>}}(\texttt{m}) = \texttt{MT}}$$

Fig. 11. Additional FlexiJava method lookup rules

we mean its definitions in all classes/interfaces `D` where $\texttt{C} \trianglelefteq_\Pi \texttt{D}$ but there is no class or interface `E` such that `E` defines `m` and $\texttt{D} \trianglelefteq_\Pi \texttt{E}$. This rule ensures that if one supertype of `C` defines the method with x fixed arguments and one rest argument, then *all* supertypes do. This requirement is necessary to ensure that our translation, described in the next section, is well-defined; it is not needed for type soundness for FlexiJava.

Dynamic Semantics FlexiJava's operational semantics is visually the same as IJ's; see figure 7 for the notation. We use $\rightarrow_{\text{FXJ}}$ for FlexiJava, and $\rightarrow_{\text{IJ}}$ for IJ. Semantically, the reduction relations differ because they use different method lookup functions ($\mathit{meth}_{\Pi,\texttt{C<}\overrightarrow{\texttt{T}}\texttt{>}}$ for FlexiJava) and subtyping relation ($<:_{\Pi\Delta}$ for FlexiJava) for casts.

4.3 Soundness and Compatibility

The FlexiJava model satisfies a type soundness theorem.

Theorem 2 (Type Soundness). *If* $\Pi = \overrightarrow{\texttt{L}}\ \texttt{e}$ *and* $\vdash \Pi : \texttt{T}$*, then one of the following must be true:*

- $\Pi \vdash \texttt{e} \rightarrow^*_{\text{FXJ}} \texttt{v}$*, where* $\Pi, \emptyset, \emptyset \vdash \texttt{v} : \texttt{S}$ *and* $\texttt{S} <:_{\Pi\emptyset} \texttt{T}$*, or*
- $\Pi \vdash \texttt{e} \Uparrow$*, or*
- $\Pi \vdash \texttt{e} \rightarrow^*_{\text{FXJ}}$ *error: bad cast.*

The proof follows the standard method [10, 9] of subject reduction and progress lemmas.

TProg

$$\frac{\Pi = \overrightarrow{\mathtt{L}}\ \mathtt{e} \qquad \overrightarrow{\Pi \vdash \mathtt{L}} \qquad \Pi, \emptyset, \emptyset \vdash \mathtt{e} : \mathtt{T} \qquad \text{all class, ifc names in } \overrightarrow{\mathtt{L}} \text{ distinct} \qquad \trianglelefteq_{\Pi} \text{ antisymmetric}}{\vdash \overrightarrow{\mathtt{L}}\ \mathtt{e} : \mathtt{T}}$$

TClass

$$\frac{\begin{array}{c}\Delta = \{\overrightarrow{\alpha : (\mathtt{N}, \mathrm{ff})}\} \qquad \overrightarrow{\Pi, \Delta \vdash \mathtt{N}} \qquad \Pi, \Delta \vdash \mathtt{P} \qquad \overrightarrow{\Pi, \Delta \vdash \mathtt{Q}} \qquad \overrightarrow{\Pi, \Delta \vdash \mathtt{T}} \\ \mathtt{P} = \mathtt{D}\texttt{<}\overrightarrow{\mathtt{V}}\texttt{>} \text{ where } \Pi \text{ contains } \texttt{class D} \cdots \qquad \mathit{consRestArgs}(\Pi, \Delta, (\mathtt{P}, \overrightarrow{\mathtt{Q}})) \\ \mathtt{Q}_i = \mathtt{E}\texttt{<}\cdots\texttt{>} \text{ where } \Pi \text{ contains } \texttt{ifc E} \cdots \text{ for all } \mathtt{Q}_i \in \overrightarrow{\mathtt{Q}} \qquad \mathit{fields}(\Pi, \mathtt{P}) = \overrightarrow{\mathtt{U}\ \mathtt{g}} \\ \overrightarrow{\mathtt{f}}, \overrightarrow{\mathtt{g}} \text{ all distinct} \qquad \Pi, \mathtt{C} \vdash \overrightarrow{\mathtt{M}} \qquad \overrightarrow{\mathtt{m}} \text{ all distinct, where } \overrightarrow{\mathtt{M}} = \overrightarrow{\texttt{<}\cdots\texttt{> W m(}\cdots\texttt{) \{ }\cdots\texttt{ \}}} \\ \forall \mathtt{Q}_i \in \overrightarrow{\mathtt{Q}}, \mathtt{m} \,.\, \mathit{imeth}_{\Pi,\Delta,\mathtt{Q}_i}(\mathtt{m}) = \mathtt{MT}' \text{ for some } \mathtt{MT}' \\ \Rightarrow \exists \mathtt{MT}, \overrightarrow{\mathtt{x}}, \mathtt{e} \,.\, \mathit{meth}_{\Pi,\mathtt{C}\texttt{<}\overrightarrow{\alpha}\texttt{>}}(\mathtt{m}) = (\mathtt{MT}, \overrightarrow{\mathtt{x}}, \mathtt{e})\end{array}}{\Pi \vdash \texttt{class C <}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> ext P impl }\overrightarrow{\mathtt{Q}}\texttt{ \{ }\overrightarrow{\mathtt{T}\ \mathtt{f}\texttt{;}}\ \overrightarrow{\mathtt{M}}\texttt{ \}}}$$

TFlexIfc

$$\frac{\begin{array}{c}\Delta = \{\overrightarrow{\alpha : (\mathtt{N}, \mathrm{ff})}, \beta : (\mathtt{Q}, \mathrm{tt})\} \qquad \overrightarrow{\Pi, \Delta \vdash \mathtt{N}} \qquad \overrightarrow{\Pi, \Delta \vdash \mathtt{P}} \qquad \mathit{consRestArgs}(\Pi, \Delta, \overrightarrow{\mathtt{P}}) \\ \Pi, \Delta \vdash \mathtt{Q} \qquad \text{for all } \mathtt{P}_i \in \overrightarrow{\mathtt{P}}\ \mathtt{P}_i = \mathtt{D}\texttt{<}\cdots\texttt{>} \text{ and } \Pi \text{ contains } \texttt{ifc D} \cdots \qquad \overrightarrow{\Pi, \mathtt{C} \vdash \mathtt{MS}} \\ \overrightarrow{\mathtt{m}} \text{ all distinct, where } \overrightarrow{\mathtt{MS}} = \overrightarrow{\texttt{<}\cdots\texttt{> W m(}\cdots\texttt{)}} \qquad \mathit{consIfcs}(\Pi, \Delta, \mathtt{C}\texttt{<}\overrightarrow{\alpha}\texttt{, } \beta\texttt{...>}, \overrightarrow{\mathtt{P}})\end{array}}{\Pi \vdash \texttt{ifc C<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{, } \beta\ \texttt{ext Q...> ext }\overrightarrow{\mathtt{P}}\texttt{ \{ }\overrightarrow{\mathtt{MS}}\texttt{ \}}}$$

ConsistentIfcs

$$\frac{\begin{array}{l}\forall \mathtt{m} \,.\, \mathtt{m} \in \mathrm{dom}(\mathit{imeth}_{\Pi,\Delta,\mathtt{P}_i}) \text{ for some } \mathtt{P}_i \in \overrightarrow{\mathtt{P}} \\ \quad \Rightarrow (\mathtt{m} = \mathrm{dom}(\mathit{imethDecl}_{\Pi,\mathtt{N}}) \\ \qquad \text{or } \exists \mathtt{P}_j \in \overrightarrow{\mathtt{P}} \,.\, \mathit{imeth}_{\Pi,\Delta,\mathtt{P}_j}(\mathtt{m}) = \mathtt{MT}_j) \\ \text{where } \mathtt{MT}_j = \\ \qquad \min\{\mathtt{MT}_i \mid \mathit{imeth}_{\Pi,\Delta,\mathtt{P}_i}(\mathtt{m}) = \mathtt{MT}_i, \mathtt{P}_i \in \overrightarrow{\mathtt{P}}\}\end{array}}{\mathit{consIfcs}(\Pi, \Delta, \mathtt{N}, \overrightarrow{\mathtt{P}})}$$

ConsistentRestArgs

$$\frac{\begin{array}{l}\forall \mathtt{m} \,.\, \exists x, y \,.\, \forall \mathtt{P}_i \in \overrightarrow{\mathtt{P}} \,.\, \mathtt{P}_i = \mathtt{C}\texttt{<\_>} \\ \qquad \text{and } \mathtt{m} \in \mathrm{dom}(\mathit{meth}_{\Pi,\mathtt{P}_i}) \\ \qquad \text{or } \mathtt{m} \in \mathrm{dom}(\mathit{imeth}_{\Pi,\Delta,\mathtt{P}_i}) \\ \quad \Rightarrow \mathit{numArgs}(\Pi, \mathtt{m}, \mathtt{C}) = (x, y)\end{array}}{\mathit{consRestArgs}(\Pi, \Delta, \overrightarrow{\mathtt{P}})}$$

TMethod

$$\frac{\begin{array}{c}\Pi \text{ contains } \texttt{class C<}\overrightarrow{\beta\ \texttt{ext P}}\texttt{> ext Q impl }\overrightarrow{\mathtt{R}}\texttt{ \{ }\cdots\texttt{ \}} \qquad \texttt{this} \notin \overrightarrow{\mathtt{x}} \\ \overrightarrow{\mathtt{x}} \text{ distinct} \qquad \overrightarrow{\Pi, \Delta \vdash \mathtt{N}} \qquad \Pi, \Delta \vdash \mathtt{T} \qquad \Gamma = \{\overrightarrow{\mathtt{x} : \mathtt{S}}, \mathtt{this} : \mathtt{C}\texttt{<}\overrightarrow{\beta}\texttt{>}\} \qquad \Pi, \Delta, \Gamma \vdash \mathtt{e} : \mathtt{U} \\ \mathtt{U} <:_{\Pi\Delta} \mathtt{T} \qquad \Delta = \{\overrightarrow{\alpha : (\mathtt{N}, \mathrm{ff})}, \overrightarrow{\beta : (\mathtt{P}, \mathrm{ff})}\} \qquad \mathit{override}(\Pi, \Delta, \mathtt{m}, \mathtt{Q}, \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>}\overrightarrow{\mathtt{S}} \to \mathtt{T}) \\ \forall \mathtt{R}_i \in \overrightarrow{\mathtt{R}} \,.\, \overrightarrow{\Pi, \Delta \vdash \mathtt{S}} \qquad \mathit{override}(\Pi, \Delta, \mathtt{m}, \mathtt{R}_i, \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>}\overrightarrow{\mathtt{S}} \to \mathtt{T})\end{array}}{\Pi, \mathtt{C} \vdash \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> T m(}\overrightarrow{\mathtt{S}\ \mathtt{x}}\texttt{) \{ e \}}}$$

TMethSig

$$\frac{\begin{array}{c}\Pi \text{ contains } \texttt{ifc C<}\overrightarrow{\beta\ \texttt{ext P}}\texttt{, } \gamma\ \texttt{ext R...> ext }\overrightarrow{\mathtt{Q}}\ \cdots \\ \Delta = \{\overrightarrow{\alpha : (\mathtt{N}, \mathrm{ff})}, \overrightarrow{\beta : (\mathtt{P}, \mathrm{ff})}, \gamma : (\mathtt{R}, \mathrm{tt})\} \qquad \overrightarrow{\mathtt{x}} \text{ all distinct} \qquad \overrightarrow{\Pi, \Delta \vdash \mathtt{S}} \qquad \Pi, \Delta \vdash \mathtt{T} \\ \overrightarrow{\Pi, \Delta \vdash \mathtt{N}} \qquad \texttt{this} \notin \overrightarrow{\mathtt{x}} \qquad \forall \mathtt{Q}_i \in \overrightarrow{\mathtt{Q}} \,.\, \mathit{override}(\Pi, \Delta, \mathtt{m}, \mathtt{Q}_i, \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{>}\overrightarrow{\mathtt{S}}\texttt{, } \gamma\texttt{...} \to \mathtt{T})\end{array}}{\Pi, \mathtt{C} \vdash \texttt{<}\overrightarrow{\alpha\ \texttt{ext N}}\texttt{> T m(}\overrightarrow{\mathtt{S}\ \mathtt{x}}\texttt{, } \gamma\texttt{...)}}$$

Override

$$\frac{\begin{array}{c}\forall \mathtt{MT}' \,.\, (\exists \overrightarrow{x}, \mathtt{e} \,.\, \mathit{meth}_{\Pi,\mathtt{N}}(\mathtt{m}) = (\mathtt{MT}', \overrightarrow{\mathtt{x}}, \mathtt{e}) \text{ or } \mathit{imeth}_{\Pi,\Delta,\mathtt{N}}(\mathtt{m}) = \mathtt{MT}') \Rightarrow \mathtt{MT} <:_{\Pi\Delta'} \mathtt{MT}' \\ \text{where } \mathtt{MT} = \texttt{<}\overrightarrow{\alpha\ \texttt{ext P}}\texttt{>}\ \cdots \text{ and } \Delta' = \{\overrightarrow{\alpha : (\mathtt{P}, \mathrm{ff})}\}\end{array}}{\mathit{override}(\Pi, \Delta, \mathtt{m}, \mathtt{N}, \mathtt{MT})}$$

Fig. 12. FlexiJava: well-typed definitions

Lemma 1 (Subject Reduction). *If* $\vdash \Pi : \mathtt{U}$ *and* $\Pi, \Delta, \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\Pi \vdash \mathtt{e} \rightarrow_{\mathrm{FXJ}} \mathtt{e}'$*, then either* $\mathtt{e}'$ *is an error configuration or* $\Pi, \Delta, \Gamma \vdash \mathtt{e}' : \mathtt{S}$*, where* $\mathtt{S} <:_{\Pi\Delta} \mathtt{T}$*.*

Proof: case analysis on the reduction $\Pi \vdash \mathtt{e} \rightarrow_{\mathrm{FXJ}} \mathtt{e}'$. Proving the case for R-CALL requires lemma 3, below. ∎

Lemma 2 (Progress). *If*

- $\vdash \Pi : \mathtt{U}$ *and*
- $\Pi, \Delta, \emptyset \vdash \mathtt{e} : \mathtt{T}$*,*

then either

- $\mathtt{e}$ *is a value, or*
- $\Pi \vdash \mathtt{e} \rightarrow_{\mathrm{FXJ}} \mathtt{e}'$*, where* $\mathtt{e}'$ *closed, or*
- $\Pi \vdash \mathtt{e} \rightarrow_{\mathrm{FXJ}}$ *error: bad cast.*

Proof: Assume $\mathtt{e}$ is not a value. Since $\mathtt{e}$ is closed, a unique decomposition lemma (not included) guarantees the existence of a unique context $\mathcal{E}$ and redex $\mathtt{r}$ such that $\mathtt{e} = \mathcal{E}[\mathtt{r}]$. The proof then proceeds by case analysis on $\mathtt{r}$. ∎

Lemma 3 (Method Typing). *If* $\vdash \Pi : \mathtt{W}$ *and*

- $meth_{\Pi,\mathtt{C}<\vec{\mathtt{T}}>}(\mathtt{m}) = (\overrightarrow{<\alpha \ \mathtt{ext}\ \mathtt{P}>}\vec{\mathtt{U}} \rightarrow \mathtt{U}_0, \vec{\mathtt{x}}, \mathtt{e})$*, and*
- $\Pi, \Delta \vdash \mathtt{C}<\vec{\mathtt{T}}>$*, and*
- $\overrightarrow{\Pi, \Delta \vdash \mathtt{V}}$*, and*
- $\overrightarrow{\mathtt{V} <:_{\Pi\Delta} \sigma(\mathtt{P})}$*, where* $\sigma(\mathtt{P}) = [\overrightarrow{\mathtt{V}/\alpha}]\mathtt{P}$*,*

then there exists some type $\mathtt{N}$ *such that*

- $\mathtt{C}<\vec{\mathtt{T}}> <:_{\Pi\Delta} \mathtt{N}$*, and*
- $\Pi, \Delta \vdash \mathtt{N}$*, and*
- $\Pi, \Delta, \{\overrightarrow{x : \sigma(\mathtt{U})}, \mathtt{this} : \mathtt{N}\} \vdash \mathtt{e} : \mathtt{S}$*, and*
- $\mathtt{S} <:_{\Pi\Delta} \sigma(\mathtt{U}_0)$*.*

Proof: Use an induction on derivation of $meth_{\Pi,\mathtt{C}<\vec{\mathtt{T}}>}(\mathtt{m})$, with the observation that if $\mathtt{C}<\vec{\mathtt{T}}>$ is well-formed and extends $\mathtt{D}<\vec{\mathtt{U}}>$ in a well-typed program, then $\Pi, \Delta \vdash \mathtt{D}<\vec{\mathtt{U}}>$. ∎

Also we demonstrate that our proposed extension does not change the behavior of existing IJ programs.

Theorem 3 (Backward Compatibility with IJ). *If* $\vdash_{\mathrm{IJ}} \Pi : \mathtt{T}$*, then* $\vdash_{\mathrm{FXJ}} \Pi : \mathtt{T}$*. Further, for all IJ expressions* $\mathtt{e}$ *and* $\mathtt{e}'$*,* $\Pi \vdash \mathtt{e} \rightarrow_{\mathrm{IJ}} \mathtt{e}'$ *if and only if* $\Pi \vdash \mathtt{e} \rightarrow_{\mathrm{FXJ}} \mathtt{e}'$*.*

Proof: The proof uses the fact that FlexiJava's type system and operational semantics are conservative extensions of IJ's, allowing for the fact that FlexiJava's Δ, $meth_{\Pi,\mathtt{C}<\vec{\mathtt{T}}>}$, $imeth_{\Pi,\Delta,\mathtt{C}<\vec{\mathtt{T}}>}$, and $<:_{\Pi\Delta}$ are also conservative extensions. ∎

```
class Pair<X, Y> {
    X fst;
    Y snd;
}
```

```
class Unit<> { }
```

Fig. 13. The `Pair` and `Unit` classes

5 Translating FlexiJava back to a Model of Java

To show that FlexiJava is compatible with existing models of the Java language, we provide a translation from FlexiJava to IJ. We prove that this translation preserves typing and run-time behavior.

5.1 Overview of the Translation

The translation requires two predefined IJ classes (figure 13). With these classes, a programmer can construct arbitrary tree structures, though the translation uses them only to build lists where a `Unit` instance, not `null`, signifies the end of the list. These lists, however, have two important properties: they are heterogeneous, and their types define their length[4] and the types of their elements precisely. Therefore, a program can extract a value from a `Pair` structure, however deeply nested, without worrying about reaching the end of the tree or having to cast the result to the desired type.

The translation uses these structures in the following ways:

1. It transforms each variable-arity interface into one that accepts a single required type argument in place of the original rest argument.
2. It replaces any occurrences of the rest argument in the interface's method signatures or `ext` clauses with the new single type argument.
3. It modifies any occurrences of the interface type to assemble their optional type arguments into a list structure as described above.
4. It modifies any methods that override or implement a method modified in point 2 above so that they accept the same argument signature and then decompose that structure in their bodies.
5. Finally, it modifies any calls to methods affected by points 2 or 4 above so that they assemble their arguments into a list structure.

Figure 14 presents the translation of an illustrative example, the `Function` interface and the `Factorial` class from section 3.1. In our model, we use `let` expressions to represent local variables, such as those in `Fact.apply`.

Finally, note that our translation is defined only on well-typed FlexiJava programs.

[4] Technically, it provides an upper bound because a programmer could always assign `null` to `snd`. Our translation uses the classes only as intended, so the type precisely encodes the length and the elements' types.

```
ifc Function<R ext Object, D ext Object> {
  R apply(D);
}

class Fact impl Function<Integer, Pair<Integer, Unit>> {
  Integer apply(Pair<Integer, Unit> args) {
    Integer n = args.fst;
    if (n == 0)
      return 1;
    else
      return n * this.apply(new Pair<Integer, Unit>(n - 1, new Unit()));
  }
}
```

Fig. 14. Translation example: `Function` and `Factorial`

5.2 The Formal Translation

Our translation is defined by a series of inference rules that define the following translation relations:

$\vdash \Pi \mapsto \Pi'$	Program Π translates to Π'
$\Pi \vdash \mathtt{T} \mapsto \mathtt{T}'$	Program Π translates type `T` to `T'`
$\Pi, \Delta, \Gamma \vdash \mathtt{e} \mapsto \mathtt{e}'$	Expression `e` translates to `e'` in the given context

Figure 15 contains selected inference rules that define the translation process. The missing rules are straightforward variations on those presented; the translation rules for expressions generally recur on subexpressions and subtypes without making any changes.

Some care is required to translate interface types of the form $\mathtt{C<}\vec{\mathtt{T}}\mathtt{,}\ \alpha\mathtt{...>}$. Consider the following example:

```
interface A<X, Y...> extends B<X, Y...> { ··· }
```

The translation of `A` accepts two type arguments, the second of which must be a pair structure as discussed above. If `B` expects one fixed argument, then the translation of the type application is trivial: `A` simply passes its second argument along to `B` without modifications. If, however, `B` does not expect any fixed arguments but only a rest argument, then the translation of this type application must construct a new type that includes `X` as well as all of the types supplied for `Y`.

Note: Had we used a tuple class, rather than a sequence of nested pairs, to implement these varargs, this translation of A would have been impossible, as there would be no way to decompose and then reconstruct this tuple in the context above. Our list structure makes this trivial, however; we simply cons the

TransClass

$$\frac{\overrightarrow{\Pi \vdash \mathtt{N} \mapsto \mathtt{N}'} \quad \Pi \vdash \mathtt{P} \mapsto \mathtt{P}' \quad \overrightarrow{\Pi \vdash \mathtt{Q} \mapsto \mathtt{Q}'} \quad \overrightarrow{\Pi \vdash \mathtt{T} \mapsto \mathtt{T}'} \quad \overrightarrow{\Pi, \mathtt{C} \vdash \mathtt{M} \mapsto \mathtt{M}'}}{\begin{array}{c}\Pi \vdash \mathtt{class\ C<}\overrightarrow{\alpha\ \mathtt{ext\ N}}\mathtt{>\ ext\ P\ impl\ }\overrightarrow{\mathtt{Q}}\mathtt{\ \{\ }\overrightarrow{\mathtt{T\ f;}}\ \overrightarrow{\mathtt{M}}\mathtt{\ \}} \mapsto \\ \mathtt{class\ C<}\overrightarrow{\alpha\ \mathtt{ext\ N}'}\mathtt{>\ ext\ P'\ impl\ }\overrightarrow{\mathtt{Q}'}\mathtt{\ \{\ }\overrightarrow{\mathtt{T}'\ \mathtt{f;}}\ \overrightarrow{\mathtt{M}'}\mathtt{\ \}}\end{array}}$$

TransFlexIfc

$$\frac{\overrightarrow{\Pi \vdash \mathtt{N} \mapsto \mathtt{N}'} \quad \overrightarrow{\Pi \vdash \mathtt{Q} \mapsto \mathtt{Q}'} \quad \overrightarrow{\Pi, \mathtt{C} \vdash \mathtt{MS} \mapsto \mathtt{MS}'}}{\begin{array}{c}\Pi \vdash \mathtt{ifc\ C<}\overrightarrow{\alpha\ \mathtt{ext\ N}}\mathtt{,\ }\beta\ \mathtt{ext\ P...>\ ext\ }\overrightarrow{\mathtt{Q}}\mathtt{\ \{\ }\overrightarrow{\mathtt{MS}}\mathtt{\ \}} \mapsto \\ \mathtt{ifc\ C<}\overrightarrow{\alpha\ \mathtt{ext\ N}'}\mathtt{,\ }\beta\ \mathtt{ext\ Any>\ ext\ }\overrightarrow{\mathtt{Q}'}\mathtt{\ \{\ }\overrightarrow{\mathtt{MS}'}\mathtt{\ \}}\end{array}}$$

TransMethRest

$$\frac{\begin{array}{c}\Pi \text{ contains } \mathtt{class\ C<}\overrightarrow{\beta\ \mathtt{ext\ P}}\mathtt{>}\ \cdots \qquad \overrightarrow{\Pi \vdash \mathtt{N} \mapsto \mathtt{N}'} \\ \Pi \vdash \mathtt{T}_0 \mapsto \mathtt{T}_0' \qquad \overrightarrow{\Pi \vdash \mathtt{T} \mapsto \mathtt{T}'} \qquad \Delta = \{\overrightarrow{\alpha : (\mathtt{N}, \mathtt{ff})}, \overrightarrow{\beta : (\mathtt{P}, \mathtt{ff})}\} \qquad \Gamma = \{\overrightarrow{\mathtt{x} : \mathtt{T}}\} \\ \Pi, \Delta, \Gamma \vdash \mathtt{e} \mapsto \mathtt{e}' \qquad \mathit{numArgs}(\Pi, \mathtt{m}, \mathtt{C}) = (n, 1) \qquad (\overrightarrow{\mathtt{U}}, \overrightarrow{\mathtt{V}}) = \mathit{splitAt}(\overrightarrow{\mathtt{T}'}, n) \\ (\overrightarrow{\mathtt{y}}, \overrightarrow{\mathtt{z}}) = \mathit{splitAt}(\overrightarrow{\mathtt{x}}, n) \qquad \mathtt{z}' \notin \overrightarrow{\mathtt{y}} \qquad (\mathtt{W}, \mathtt{e}'') = \mathit{mkLet}(\overrightarrow{\mathtt{V}}, \overrightarrow{\mathtt{z}}, \mathtt{z}', \mathtt{e}')\end{array}}{\Pi, \mathtt{C} \vdash \mathtt{<}\overrightarrow{\alpha\ \mathtt{ext\ N}}\mathtt{>T_0\ m(}\overrightarrow{\mathtt{T\ x}}\mathtt{)\ \{\ e\ \}} \mapsto \mathtt{<}\overrightarrow{\alpha\ \mathtt{ext\ N}'}\mathtt{>T_0'\ m(}\overrightarrow{\mathtt{U\ y}}\mathtt{,\ W\ z')\ \{\ e''\ \}}}$$

TransFlexSignature

$$\frac{\overrightarrow{\Pi \vdash \mathtt{N} \mapsto \mathtt{N}'} \quad \Pi \vdash \mathtt{T}_0 \mapsto \mathtt{T}_0' \quad \overrightarrow{\Pi \vdash \mathtt{T} \mapsto \mathtt{T}'} \quad \mathtt{rest} \notin \overrightarrow{\mathtt{x}}}{\Pi, \mathtt{C} \vdash \mathtt{<}\overrightarrow{\alpha\ \mathtt{ext\ N}}\mathtt{>\ T_0\ m(}\overrightarrow{\mathtt{T\ x}}\mathtt{,\ }\beta\ \mathtt{...)} \mapsto \mathtt{<}\overrightarrow{\alpha\ \mathtt{ext\ N}'}\mathtt{>\ T_0'\ m(}\overrightarrow{\mathtt{T'\ x}}\mathtt{,\ }\beta\ \mathtt{rest)}}$$

TypeListEmpty

$$\mathit{typeList}(\bullet) = \mathtt{Unit}$$

TransTAppRest

$$\frac{\begin{array}{c}\Pi \text{ contains } \mathtt{ifc\ C<}\overrightarrow{\alpha\ \mathtt{ext\ N}}\mathtt{,\ }\beta\ \mathtt{ext\ Q\ ...>}\ \cdots \\ \overrightarrow{\Pi \vdash \mathtt{T} \mapsto \mathtt{T}'} \qquad (\overrightarrow{\mathtt{U}}, \overrightarrow{\mathtt{V}}) = \mathit{splitAt}(\overrightarrow{\mathtt{T}'}, \#(\overrightarrow{\alpha}))\end{array}}{\Pi \vdash \mathtt{C<}\overrightarrow{\mathtt{T}}\mathtt{>} \mapsto \mathtt{C<}\overrightarrow{\mathtt{U}}\mathtt{,\ }\mathit{typeList}(\overrightarrow{\mathtt{V}})\mathtt{>}}$$

TypeList′Empty

$$\mathit{typeList}'(\bullet; \beta) = \beta$$

TransTAppFlex

$$\frac{\begin{array}{c}\Pi \text{ contains } \mathtt{ifc\ C<}\overrightarrow{\alpha\ \mathtt{ext\ N}}\mathtt{,\ }\gamma\ \mathtt{ext\ Q\ ...>}\ \cdots \\ \overrightarrow{\Pi \vdash \mathtt{T} \mapsto \mathtt{T}'} \qquad (\overrightarrow{\mathtt{U}}, \overrightarrow{\mathtt{V}}) = \mathit{splitAt}(\overrightarrow{\mathtt{T}'}, \#(\overrightarrow{\alpha}))\end{array}}{\Pi \vdash \mathtt{C<}\overrightarrow{\mathtt{T}}\mathtt{,\ }\beta\mathtt{...>} \mapsto \mathtt{C<}\overrightarrow{\mathtt{U}}\mathtt{,\ }\mathit{typeList}'(\overrightarrow{\mathtt{V}}; \beta)\mathtt{>}}$$

TypeList

$$\frac{\mathit{typeList}(\overrightarrow{\mathtt{T}}) = \mathtt{U}}{\mathit{typeList}((\mathtt{T}_0, \overrightarrow{\mathtt{T}})) = \mathtt{Pair<T_0,\ U>}}$$

TypeList′

$$\frac{\mathit{typeList}'(\overrightarrow{\mathtt{T}}; \beta) = \mathtt{U}}{\mathit{typeList}'((\mathtt{T}_0, \overrightarrow{\mathtt{T}}); \beta) = \mathtt{Pair<T_0,\ U>}}$$

TransCallRest

$$\frac{\begin{array}{c}\Pi, \Delta, \Gamma \vdash \mathtt{e}_0 \mapsto \mathtt{e}_0' \qquad \Pi, \Delta, \Gamma \vdash \mathtt{e}_0 : \mathtt{T} \\ \mathtt{T} = \mathtt{C<}\cdots\mathtt{>} \qquad \overrightarrow{\Pi \vdash \mathtt{V} \mapsto \mathtt{V}'} \qquad \overrightarrow{\Pi, \Delta, \Gamma \vdash \mathtt{e} \mapsto \mathtt{e}'} \qquad \mathit{numArgs}(\Pi, \mathtt{m}, \mathtt{C}) = (n, 1) \\ \mathit{meth}_{\Pi,\mathtt{T}}(\mathtt{m}) = (\overrightarrow{\mathtt{<}\alpha\ \mathtt{ext\ N}\mathtt{>}}\ \overrightarrow{\mathtt{U}} \to \mathtt{U}_0, \_, \_) \text{ or } \mathit{imeth}_{\Pi,\Delta,\mathtt{T}}(\mathtt{m}) = \overrightarrow{\mathtt{<}\alpha\ \mathtt{ext\ N}\mathtt{>}}\ \overrightarrow{\mathtt{U}} \to \mathtt{U}_0 \\ \overrightarrow{\Pi \vdash \mathtt{U} \mapsto \mathtt{U}'} \qquad (\overrightarrow{\mathtt{g}}, \overrightarrow{\mathtt{h}}) = \mathit{splitAt}(\overrightarrow{\mathtt{e}}, n) \qquad (\_, \overrightarrow{\mathtt{W}}) = \mathit{splitAt}(\overrightarrow{\mathtt{U}'}, n)\end{array}}{\Pi, \Delta, \Gamma \vdash \mathtt{e}_0\mathtt{.m<}\overrightarrow{\mathtt{V}}\mathtt{>(}\overrightarrow{\mathtt{e}}\mathtt{)} \mapsto \mathtt{e}_0'\mathtt{.m<}\overrightarrow{\mathtt{V}'}\mathtt{>(}\overrightarrow{\mathtt{g}}\mathtt{,\ }\mathit{pkgArgs}(\overrightarrow{\mathtt{W}}\mathtt{,\ }\overrightarrow{\mathtt{h}})\mathtt{)}}$$

Fig. 15. Selected translation rules

new type `X` onto the front of the list as implemented in the TransTAppRest rule. ∎

As a consequence of the TransCallRest and PkgArgs rules, the translation of an affected method-call expression must state the types of some of its arguments multiple times; for example, if `f` implements `Function<Integer, Integer, String>`, then a call to `f.apply` would translate as follows:

```
f.apply(new Pair<Integer, Pair<String, Unit>>
            (3, new Pair<String, Unit>
                    ("foo", new Unit())))
```

Each value appears exactly once, but for each `Pair` type that appears in this expression, its tail must also appear one additional time. As a result, the length of such method-call expressions increases by an amount that varies quadratically with the number of types supplied to the relevant rest argument. Thus the resulting source file may grow larger than the original but in a controlled fashion. We leave it as an open problem to find a correct translation that requires linear space.

5.3 Structural Properties of the Translation

This translation is correct with respect to the static semantics of FlexiJava.

Theorem 4 (Translation Preserves Typing). *If* $\vdash_{\text{FXJ}} \Pi : \mathtt{T}$ *and* $\vdash \Pi \mapsto \Pi'$ *and* $\Pi \vdash \mathtt{T} \mapsto \mathtt{T}'$*, then* $\vdash_{\text{IJ}} \Pi' : \mathtt{T}'$*.*

Proof: The proof follows the derivation of $\vdash_{\text{FXJ}} \Pi : \mathtt{T}$; it relies heavily on supplementary lemmas. These lemmas are not stated here for space reasons, but they establish that translation preserves many other properties of the program, including the following:

- expression typing;
- well-formedness of definitions, methods, signatures, types;
- interface consistency; and
- subtyping.

For many of these lemmas, it is also necessary to translate the types that appear in Δ and Γ. This in turn requires a lemma that states that translation and type substitution commute. ∎

Lemma 4 (Commutativity of Translation and Type Substitution). *If* $\Pi, \Delta \vdash \mathtt{T}$ *and*

- $\Pi, \Delta \vdash \mathtt{N}$ *and*
- $\Pi, \Delta \vdash \vec{\mathtt{U}}$ *and*
- $\Pi \vdash \mathtt{T} \mapsto \mathtt{T}'$ *and*
- $\Pi \vdash \mathtt{N} \mapsto \mathtt{N}'$ *and*

– $\overrightarrow{\Pi \vdash \mathtt{U} \mapsto \mathtt{U}'}$,

then all of the following hold:

– $\Pi \vdash [\mathtt{T}/\alpha]\mathtt{N} \mapsto [\mathtt{T}'/\alpha]\mathtt{N}'$, *and*
– $\Pi \vdash [\vec{\mathtt{U}}/\beta\ldots]\mathtt{N} \mapsto [\vec{\mathtt{U}'}/\beta]\mathtt{N}'$, *and*
– $\Pi \vdash [\vec{\mathtt{U}}, \gamma\ldots/\beta\ldots]\mathtt{N} \mapsto [\mathit{typeList}'(\vec{\mathtt{U}'}; \gamma)/\beta]\mathtt{N}'$

where α *is not in* $\mathtt{T}$ *and* β *is not in* $\vec{\mathtt{U}}$, γ.

Proof of Lemma 4: Induction on $\Pi, \Delta \vdash \mathtt{N}$. ∎

Finally, we establish that the translation preserves the behavior of the program. Informally, this result states that performing a single reduction step and then translating provides the same result as translating then performing possibly many reduction steps.

Theorem 5 (Translation Preserves Behavior). *Let* Π, $\mathtt{e}$ *be a FlexiJava program and expression such that* $\vdash \Pi : \mathtt{S}$ *and*

$$\Pi, \Delta, \Gamma \vdash \mathtt{e} : \mathtt{T}$$

for some well-formed Δ, Γ, $\mathtt{S}$ *and* $\mathtt{T}$. *Let* Π' *and* $\mathtt{d}$ *be an IJ program and expression such that* $\vdash \Pi \mapsto \Pi'$ *and* $\Pi, \Delta, \Gamma \vdash \mathtt{e} \mapsto \mathtt{d}$.

If $\Pi \vdash \mathtt{e} \rightarrow_{\mathrm{FXJ}} \mathtt{e}'$, *then* $\Pi' \vdash \mathtt{d} \rightarrow^*_{\mathrm{IJ}} \mathtt{d}'$ *and* $\Pi, \Delta, \Gamma \vdash \mathtt{e}' \mapsto \mathtt{d}'$.

Proof: Induction on $\Pi, \Delta, \Gamma \vdash \mathtt{e} : \mathtt{T}$. Multiple reductions in IJ are required when $\mathtt{e}$ is a method call expression that invokes a method whose translation involves a list argument. First, several reductions may be necessary to assemble the list structure, then the actual method call reduction takes place, and finally several more reductions occur to reduce the `let` expressions introduced by the translation. The result of these last reductions is equivalent to the translation of $\mathtt{e}'$.

Because we can exploit the heterogeneity of the constructed "lists," our translation algorithm does not add any type-cast expressions to the program, unlike the "erasure" translation from FGJ to FJ. This dramatically simplifies the statement and proof of this theorem, as we do not have to demonstrate that these additional casts cannot fail at run time. ∎

6 From a Model to a Full Language

Scaling FlexiJava to the full Java 1.5 programming language requires that we address four concerns: type erasure, method overloading, separate compilation, and library compatibility.

First, we must handle the fact that our translation using the Pair and Unit classes, which is designed for languages without type erasure, can lead to problems under Java's type erasure rules. Instead of the translation in the previous section, we could generate new interfaces by-need. New interfaces would need to

be generated only when a different number of type arguments were used. This would result in Scala-like versions of the interface on the back-end but without restricting instantiations to a maximum number of generic arguments.

Second, we need to address overloading. A generalization of Java's current overload resolution mechanism should suffice. Consider the following Java classes:

```
class Overloaded<X extends Object> {
    void m(X arg) { ... }
    void m(String s) { ... }
}
class Client {
    void m() {
        Overloaded<String> ov =
            new Overloaded<String>();
        ov.m("Hello, world!");
    }
}
```

Java compiles `Overloaded` without a warning, even though there exists a choice for `X` that causes an overload conflict. Java also allows the specialization of `Overloaded` at `String`, even though this specifically results in conflicting methods. Instead, Java delays the error until it is actually significant: it signals an error at the invocation of `ov.m`, because the overload resolution is ambiguous. We conjecture that the obvious generalization to overloading with variable arity generics suffices.

Third, existing separate compilation mechanisms continue to work with FlexiJava. Type-checking a FlexiJava class or interface requires the definitions of all supertypes and the definitions of all types used within the body of the definition. Translating a FlexiJava class or interface down to Java requires no additional information. As this is exactly the same information that is required to compile a Java class or interface, existing separate compilation techniques scale directly to FlexiJava.

Fourth, by "library compatibility" we mean the ability to use a pre-existing Java library—without recompilation—in a FlexiJava program. In this context, the library cannot refer to any classes or interfaces defined in the FlexiJava program (ignoring reflection), but we must still address whether the FlexiJava program can safely pass one of its objects to the Java library for use via a callback.

For a concrete example, assume that the library accepts callback objects that implement `ISubscriber`, an interface from the library that specifies a `notify` method. Furthermore, imagine that the FlexiJava program contains a `Subscriber` class that implements the `ISubscriber` interface. The program can now register an instance of the `Subscriber` class with some library object. After all, if the FlexiJava program is well-typed, then `Subscriber` must implement `ISubscriber`, meaning the FlexiJava type system ensures that the method `notify` in `Subscriber` does not implement or override a method with a

rest argument. Therefore, the translation does not affect the external interface of `notify` at all, so the linking succeeds. `notify` may refer to methods that are affected by the translation internally, but this is not visible to the Java 1.5 library.

7 Future Work and Conclusion

The paper introduces the idea of variable-arity generic interfaces in the context of Java. It shows how object-oriented languages can benefit from this type-level abstraction; that it is type-safe; and that it is compatible with existing languages.

In the future, we wish to study three potential extensions of this work. First, we could allow the programmer to require a method per argument. In a language such as C# with variable-arity parameters for interfaces, design patterns such as the Visitor pattern can be encoded at the type level:

```
interface Visitor<R, B...> {
  R visit(B x);
  ...
}
```

Then we could write the following Visitor class for a binary tree data structure:

```
class TreeCounter<A> : Visitor<int, Leaf<A>, Branch<A>> {
  int visit(Leaf<A> x) { ___ }
  int visit(Branch<A> x) { ___ }
}
```

Second, it ought to be possible to introduce type-level rest arguments via method signatures in addition to interfaces:

```
interface F<> {
    <T ...> S m(___, T x);
}
```

Unfortunately, the FlexiJava model cannot easily accommodate this extension.

Third, the application of variable-arity polymorphism to C#'s delegates also shows promise. Delegates in C# are essentially abstract method signatures for closures—in other words, light-weight interfaces. We therefore intend to study a model that incorporates delegates as well as interfaces.

Acknowledgments

We thank Gilad Bracha, formerly of Sun Microsystems, for his help in clarifying the consistency requirements for overriding methods in an interface.

References

1. Gregor, D., Järvi, J.: Variadic templates for C++. In: SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing, New York, ACM Press (2007) 1101–1108
2. Hall, D.A.: Java bug report 6261297. Available from `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6261279` (2005)
3. Odersky, M., et al.: Programming in Scala. Available from `http://scala.epfl.ch/docu/index.html` (2004)
4. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java(TM) Language Specification. third edn. Addison-Wesley (2005)
5. ECMA: Standard ECMA-334: C# language specification. Available at `http://www.ecma-international.org/publications/files/ecma-st/ECMA-334.pdf` (2002)
6. Kennedy, A., Syme, D.: Design and implementation of generics for the .NET Common Language Runtime. In: PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, New York, ACM Press (2001) 1–12
7. Barendregt, H.P.: Introduction to the lambda calculus. Nieuw Archief voor Wisenkunde **2** (1984) 337–372
8. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23**(3) (2001) 396–450
9. Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. In: Formal Syntax and Semantics of Java. Volume 1523 of Springer Lecture Notes in Computer Science. Springer-Verlag (1999) 241–269
10. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (1994) 38–94