Gradual Typing for Generics

Lintaro Ina
Graduate School of Informatics, Kyoto University
ina@kuis.kyoto-u.ac.jp

Atsushi Igarashi
Graduate School of Informatics, Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Abstract

Gradual typing is a framework to combine static and dynamic typing in a single programming language. In this paper, we develop a gradual type system for class-based object-oriented languages with generics. We introduce a special type to denote dynamically typed parts of a program; unlike dynamic types introduced to C♯ 4.0, however, our type system allows for more seamless integration of dynamically and statically typed code.

We formalize a gradual type system for Featherweight GJ with a semantics given by a translation that inserts explicit run-time checks. The type system guarantees that statically typed parts of a program do not go wrong, even if it includes dynamically typed parts. We also describe a basic implementation scheme for Java and report preliminary performance evaluation.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language Classifications—object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects, polymorphism

General Terms Languages, Design, Theory

Keywords Gradual typing, generics, dynamic types

1. Introduction

Statically and dynamically typed languages have their own benefits. On the one hand, statically typed languages enjoy type safety properties; on the other hand, dynamically typed languages are said to be suitable for rapid prototyping. There is a significant amount of work (e.g., [1, 3, 5, 6, 8, 10, 15, 16, 27–29] to cite some) to integrate both kinds of languages to have the best of both worlds. Siek and Taha have coined the term "gradual typing" [27] for a particular style of linguistic support of the seamless integration of static and dynamic typing in a single language. A typical gradual type system introduces to a statically typed language a special type (often called `dynamic` or `dyn`) to specify dynamically typed parts in a program and allows a program to be partially typed, or even fully dynamically typed.

One of the main challenges in the design of a gradual type system is to give a flexible type compatibility relation, which is an extension of subtyping and used for assignments and argument passing. For example, a gradual type system usually assumes `dyn` to be compatible with any type so that a statically typed expression can be used where `dyn` is expected and vice versa. Moreover, when types have structures (as in function types), the compatibility relation usually allows structural comparison: for example, a function type, say `dyn` $\to$ `int`, is compatible with `int` $\to$ `int` [27], which is useful in higher-order programs.

Another, more technical challenge is to establish some safety property even for partially typed programs. In fact, it is possible to ensure that run-time errors are always due to a dynamically typed part in a program. Roughly speaking, the main idea is to insert run-time checks between the "border" between the statically and dynamically typed worlds to prevent statically typed code from going wrong. A key idea here is that the insertion can be guided by the use of the compatibility relation.

In this paper, we develop a gradual type system for class-based object-oriented languages with generics. Although there are similar attempts at mixing static and dynamic typing in object-oriented languages [3, 5, 20, 28, 38], (to our knowledge) very few take generics into account. One notable exception is dynamic types for C♯ 4.0 [4], but the integration of dynamic and static typing is not as smooth as one might expect. For example, it requires tedious coding to convert a collection whose element type is statically known, say, to be integers to a collection of dynamically typed values. We design a flexible compatibility relation, which allows, for example, `List<Integer>` to be used as `List<dyn>` and vice versa. Since the type system has inheritance-based subtyping, it is not a trivial task to give a reasonable compatibility relation. We also introduce the notion of *bounded dynamic types*, which have characteristics of both dynamic and static types, to mediate bounded polymorphism and dynamic typ-

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

ing. We formalize these ideas as FGJ$^{\mathtt{dyn}}$, an extension of Featherweight GJ (FGJ) [18] with bounded dynamic types and prove the desired safety property, which states that statically typed parts in a program cannot go wrong. In particular, it implies the standard type safety for a program that does not contain any dynamic types. The semantics of FGJ$^{\mathtt{dyn}}$—the surface language in which programs are written—is given by a translation to an intermediate language FGJ$^{\langle\!(\rangle\!)}$, in which run-time checks are explicit. The translation is not only expediency for the formal proof but also a guide to implementation.

Our main contributions can be summarized as follows:

- A flexible compatibility relation for parametric types;
- The introduction of bounded dynamic types;
- Formalization of the language with generics and dynamic types; and
- Proof of safety properties, which show that statically typed parts in a program never go wrong.

We are currently developing a compiler for gradually typed Java. We also describe our basic implementation scheme. This work at an earlier stage has been reported at the STOP'09 workshop [19], where we have only sketched the combination of generics and dynamic typing and its formalization. In this paper, we have revised the formal definition of both surface and intermediate languages significantly and proved safety properties.

The rest of the paper is organized as follows. Section 2 gives an overview of gradual typing for a class-based language with generics. Then, Sections 3 and 4 give the formalization of our proposal and prove desired properties. Section 5 describes our implementation scheme for Java and reports very preliminary benchmark results. After Section 6 discusses related work, Section 7 gives concluding remarks. Some of formal definitions and proofs of the theorems are omitted for brevity; they appear in a full version of this paper, available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/gradual.html`.

2. Gradual Typing for Generics

Following the previous approaches to gradual typing [27, 28], we introduce a special type `dyn` that represents dynamically typed portions in a program to a class-based language with generics. A variable can be declared to have the dynamic type; then, any expression can be assigned to it and the variable can be used as an expression of any type. In this section, we first describe how `dyn` interacts with generics by means of examples and then what kind of dynamic checks are performed to prevent statically typed parts from going wrong.

We use the following simple generic class as a running example:

```
class Cell<X> {
  X x;  Cell(X x){ this.x=x; }
  void set(X x){ this.x=x; }
}
```

`Cell` is a class of one-element containers, where the element type is parameterized as `X`. The element is accessed through the field `x` and modified through method `set`. We will also use classes `Shape` and its subclasses `Rectangle` and `Polygon`. (Neither `Rectangle` nor `Polygon` extends the other.) Moreover, class `Shape` has a method with the signature

```
boolean contains(double x, double y);
```

which returns whether a given point at $(\mathtt{x}, \mathtt{y})$ is inside the shape.

2.1 Type `dyn` as Type Arguments

One natural consequence of the introduction of `dyn` as a type is that `dyn` can be used as a type argument to a generic class. For example, a programmer can use a variable `c1` of type `Cell<dyn>`. This type is similar to `Cell<Object>` in the sense that one can set anything to it.

```
Cell<dyn> c1 = ...;
c1.set(new Polygon(...));
c1.set(new Integer(1));
```

Unlike `Cell<Object>`, however, the type of field `x` is `dyn`, which represents dynamically typed code and accepts any method invocation and field access, which are assumed to return `dyn`.

```
dyn fld = c1.x.anyField;
dyn ret = c1.x.anyMethod(...);
```

Also, `dyn` can be assigned to any variable.

```
boolean b = c1.x.contains(1,1);
            // The type of RHS is dyn
```

Of course, it must be checked at run time whether these fields and methods really exists and whether an assignment is valid.

In the previous work on gradual typing for a language with subtyping, the subtyping relation is replaced with the compatibility relation [27, 28], which, for example, allows statically typed expressions to be passed where `dyn` is expected and vice versa. The compatibility relation should be rich enough to support flexible integration of statically and dynamically typed code and, for type systems with structural subtyping, its definition requires careful examination.

We introduce a rich compatibility relation for parametric types. In particular, we allow an expression of a `dyn`-free type, say `Cell<Rectangle>`, to be assigned to a variable whose type involves `dyn` as a type argument, say `Cell<dyn>`. For example, the following code is accepted by the type system:

```
Cell<dyn> c1
  = new Cell<Rectangle>(new Rectangle(...));
```

Note that `c1` will point to an object that can store only `Rectangle`s, rather than anything (as indicated by `dyn`).

So, actually, the invocation of `set` should check at run time whether the actual argument is a valid one.

```
c1.set( new Rectangle(...) ); // succeeds
c1.set( new Polygon(...) );   // fails
```

The intuition behind a parametric type to which `dyn` is given as an argument is that it denotes the set of types where `dyn` is replaced with any type. For example, a variable of type `Cell<dyn>` may point to objects `new Cell<Shape>()`, `new Cell<Rectangle>()`, `new Cell<Integer>()`, and so on. In this sense, type `Cell<dyn>` is closer to the wildcard type `Cell<?>` than `Cell<Object>`, but, unlike `Cell<?>`, potentially unsafe operations such as invocation of `set` are (statically) allowed.

Our compatibility relation allows the opposite direction of flow, too—that is, an expression whose type involves `dyn` as a type argument can be assigned to a variable of a `dyn`-free type as in the following code:

```
Cell<Rectangle> c2 = c1;
Cell<Polygon> c3 = c1;
```

Just as in an assignment of an expression of type `dyn` to a concrete type, the run-time system will check whether these are valid assignments: in this case, only the first assignment will succeed.

In summary, our compatibility relation allows `dyn` in a type expression to be replaced with a concrete type and vice versa. As we will see in the next section, however, its formal definition is more subtle than might have appeared when we take inheritance-based, nominal subtyping into account. Due to nominal subtyping, we take an approach different from the previous work.

Seamless Integration. Allowing `dyn` as a type argument lets us easily evolve a class definition from its non-generic version into a generic version. For example, at the beginning, a programmer had a non-generic class `List`, which has its head item `head` and the rest of list `tail`, and some client code using it.

```
class List { dyn head; List tail; }

List list = new List(...);
list.head.doSomething();
```

Then, the programmer wanted to have a generic version of `List` because he added another client code using `List` containing `Integer`s and nothing else. So, he modified the definition of class `List`.

```
class List<X> { X head; List<X> tail; }

List<Integer> intlist = new List<Integer>(...);
```

This modification breaks other client code using `List` and he need to fix them all, however, all he have to do is to replace `List` with `List<dyn>`.

```
List<dyn> list = new List<dyn>(...);
list.head.doSomething();
```

$(\!\lvert \mathtt{T} \rvert\!)\,\mathtt{e}$	checks if the run-time type of the value of `e` is compatible with `T`, and then returns the value.
`get(e,f)`	checks if the value of `e` has field `f`, and then reduces to the field value.
$\mathtt{e.m[}\overline{\mathtt{T}}\mathtt{\mid C<}\overline{\mathtt{X}}\mathtt{>]}(\overline{\mathtt{e}})$	checks if the types of arguments $\overline{\mathtt{e}}$ are correct (using the static type information $\overline{\mathtt{T}}$, $\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}$), and then invoke method `m` on receiver `e`.
$\mathtt{invoke(e,m,}\overline{\mathtt{e}}\mathtt{)}$	checks if the value of receiver `e` has method `m` and the types of arguments $\overline{\mathtt{e}}$ are correct, and then invoke the method on the receiver.

Table 1. Constructs for run-time checks.

If the compiler allows `List` as an abbreviation of `List<dyn>`[1], he could have done the modification even without any other type-name fixes!

Note that we need to allow `new List<dyn>(...)`. This is quite different from wildcards, since `new List<?>(...)` is disallowed.

2.2 Run-time Checks

To ensure that statically typed code (or, more precisely, code that would be well typed in the standard type system) will not go wrong, errors due to dynamically typed code have to be captured at the "border" between the two worlds. For this purpose, we introduce an intermediate language, which has explicit constructs for run-time checks; the semantics of the surface language, which we describe above, will be given in terms of the translation to the intermediate language.

Although there is no direct semantics for the surface language, a program in the surface language can be mostly directly understandable because the translation only inserts run-time checks and preserves the structure of a program. Moreover, run-time checks are inserted only where dynamic types are involved. So, as far as statically typed code is concerned, the translation is the identity map, inserting no run-time check. Then, to show that statically typed code never goes wrong, it suffices to show that all run-time failures are due to those explicit checks.

We will give an overview of constructs for run-time checks and the translation below. Table 1 shows constructs for run-time checks and their intuitive meanings. In what follows, we write $\mathtt{e} \rightsquigarrow \mathtt{e}'$ to mean that a surface language expression (or statement) `e` is translated to $\mathtt{e}'$.

First, when an expression of a type that involves `dyn` is passed to where a type without `dyn` is expected, a cast $(\!\lvert \;\rvert\!)$ is inserted:

[1] `List` is a raw type of `List<X>` in the current Java specification, where it is allowed to assign a value of `List<T>` to a variable of `List`. Since this behavior is similar to `List<dyn>`, it is quite natural to allow the abbreviation instead of using raw types. See Section 6 for details.

```
Cell<dyn> c1; Cell<Rectangle> c2; Cell<Polygon> c3;
c2 = c1;
  ⇝ c2 = ⟪Cell<Rectangle>⟫c1;
c3 = c1;
  ⇝ c3 = ⟪Cell<Polygon>⟫c1;
```

We use different parentheses ⟪ ⟫ to denote casts because the semantics is slightly different from Java's. Note that the first cast above has to check that the run-time value of `c1` is an instance of `Cell<Rectangle>` and not that of, say, `Cell<Integer>`. So, this cast requires run-time type argument information. There are other differences, which we discuss later, as well.

A member access on `dyn` will be translated to special forms `get()` or `invoke()`, which checks the existence of the member at run-time.

```
Cell<dyn> c1;
c1.x.radius;
  ⇝ get(c1.x, radius);
c1.x.contains(1, 1);
  ⇝ invoke(c1.x, contains, 1, 1);
```

When `c1.x` has method `contains`, `invoke` above also checks whether it can take two integers.

As we have already discussed, the invocation of `set` on type `Cell<dyn>` will have to check whether the run-time type of the argument is appropriate for the run-time type argument to the receiver's class, even though the existence of method `set` is statically guaranteed. For such cases, we use method invocation of the form e_0`.m[`$\overline{T}$`|C<`$\overline{X}$`>](`$\overline{e}$`)` (where an overline denotes a list). For example, we have the following translation.

```
c1.set( new Polygon(...) );
  ⇝ c1.set[X|Cell<X>]( new Polygon(...) );
```

The annotations `X` and `Cell<X>` record a parameter type and a receiver's static type *before type parameters are instantiated* and are used to check the argument. It works as follows: When `c1` evaluates to a value `new Cell<T>(...)` for some type `T`, the actual receiver type `Cell<T>` is matched against `Cell<X>` and `X` is bound to `T`. Then, the actual argument's type (here `Polygon`) is checked against `T`, which is obtained by replacing `X` with `T` in the recorded parameter type. So, this method invocation succeeds when `c1`'s value is an object of `Cell<Polygon>` (or `Cell<T>` where `T` is a supertype of `Polygon`).

2.3 Ensuring "statically typed parts cannot go wrong"

One of our goals of the gradual type system is to ensure that "statically typed parts in a program never go wrong", in particular, class definitions that pass the standard type checker should not go wrong. Another desirable property of the system is modularity of type checking, that is, determining whether the given part of the program is statically typed or not should be done by looking at no more than a single class definition and type information that it depends on. We also aim at implementation by erasure translation [7].[2] Actually, modular checking and erasure translation make it trickier to ensure the safety of "statically typed code".

First of all, even if a class definition contains no occurrence of `dyn`, it should not be considered statically typed because subexpressions may be given type `dyn`. So, a sensible definition of a statically typed class definition is something like "a class definition is statically typed if there is no occurrence of `dyn` and every subexpression is given a 'dyn-free' type." In fact, as we will see later, in our translation, a method invocation requires no run-time check if the receiver and actual argument types are all `dyn`-free. Then, a class definition that passes the standard type checking will translate to itself, without run-time checks.

However, the problem is more subtle than it might have appeared, due to the presence of type variables. In general, type variables should not be considered `dyn`-free simply because type variables can be instantiated with `dyn`. In fact, the following classes, which are typed under the standard type system of generics

```
class StrCell extends Cell<String> {
  void set(String x){ ... x.length() ... }
}
class Foo<Y> {
  void bar(Cell<Y> c, Y x) {
    c.set(x);
  }
}
```

will raise a run-time error when combined with the following code:

```
new Foo<dyn>().bar(new StrCell(...), new Object());
```

The last expression passes a `StrCell` and an `Object` to `Foo<dyn>.bar()`, which expects a `Cell<dyn>` and `dyn`, and this is allowed due to the extended compatibility relation we have already mentioned. In `Foo<Y>.bar()`, an object `x` is passed to `Cell<Y>.set()`. However, in this case, the receiver is a `StrCell` and `StrCell.set()` will be called with an argument `new Object()`, which does not have `length()`!

To avoid this problem, we separate type variables into two kinds: one can only be replaced with `dyn`-free types, and the other can be replaced with dynamic types. These kinds are indicated in the class definition, for example, `class Foo<`$Y^{\Diamond}$`>...` for the former and `class Foo<`$Y^{\blacklozenge}$`>...` for the latter. If `Foo` is defined as `class Foo<`$Y^{\Diamond}$`>...`, then no run-time check is inserted but the problematic expression above is rejected at compile time. Otherwise, if `Foo` is defined as `class Foo<`$Y^{\blacklozenge}$`>...`, then the invocation of `set` will check whether the actual argument types are valid for the formal (by using `e.m[`$\overline{T}$`|C<`$\overline{X}$`>](`$\overline{e}$`)`).

Although, in principle, a programmer can choose the kind for each type variable declaration in a single class

[2] Our compilation scheme actually requires support for run-time type arguments. However, method signatures are subject to erasure.

definition, we do not expect that a programmer wants to do it. A practical design would be that a compiler option will decide the kind of all the type variables in a compiled file at once. In the beginning of development, the programmer may compile most generic classes with kind ♦, and then switches some classes to ◊ gradually as the development progresses—such a switch would force their client code to remove the use of dynamic types. In the rest of the paper, we omit kinds of type variables when they do not make significant difference.

There can be another solution to the problem in which the `StrCell` is "wrapped" with another object when it is passed to `Cell<dyn>`. The wrapper object mimics the interface of `StrCell` by delegating member accesses to the original object and performs run-time checks on every delegation. In this case, the run-time check blames the misusage of the `StrCell` on the invocation of `set()`. We do not choose this solution mainly because of a difficulty of implementation and leave it for future work.

2.4 Bounded Dynamic Types

Another problem occurs when a type variable is given an upper bound. To illustrate the problem, consider the following class:

```
class ShapeCell<X♦ extends Shape> {
  X x;  ShapeCell(X x){ this.x=x; }
  void set(X x){ this.x=x; }
  boolean contains(double px, double py){
    return this.x.contains(px, py);
  }
}
```

Class `ShapeCell`, which is similar to class `Cell` above, specifies `Shape` as `X`'s upper bound. Note that `ShapeCell` does not contain `dyn` anywhere and the whole class definition will be well typed in the standard type system of generics (by removing ♦).

Now consider type `ShapeCell<dyn>`. The question here is what we can set to `x`. One choice would be to allow any object to be set to `x`, as we did for `Cell<dyn>`:

```
ShapeCell<dyn> sc = ...;
sc.set( new Object() );
```

However, this choice would not be compatible with the implementation by erasure, which translates the type of field `x` to be `Shape`, the upper bound of `X`. In a language without erasure semantics, for example in C♯, this choice would not conflict with the implementation, but, as long as homogeneous translation [25] is used, we would need to treat type variable `X` with kind ♦ as `dyn`. This leads to a major performance disadvantage since operations on expressions of type `X` need to be augmented with run-time checks that require membership tests: for example in the case above, the invocation of `contains` on `this.x` need to be replaced with an expensive run-time check by `invoke`, which checks the presence of method `contains` and (if it exists) whether the types of formal and actual arguments match. Thus, our choice here is to keep compatibility with implementation by erasure and to avoid performance penalty as much as possible: in other words, we reject the code above statically.

We introduce *bounded dynamic types*, written `dyn<T>` (where `T` stands for a parametric type). A type parameter with an upper bound `T` can be instantiated by a bounded dynamic type `dyn<T′>` when `T′` is a subtype of `T`. Thus, `ShapeCell<dyn<Shape>>` is a well-formed type, whereas `ShapeCell<dyn<Object>>` is not.

We define a bounded dynamic type `dyn<T>` to be compatible only with subtypes of `T`. So, the following code will be ill typed and rejected by the type checker.

```
ShapeCell<dyn<Shape>> sc = ...;
sc.set( new Object() );
// Object is not compatible with dyn<Shape>!
```

A bounded dynamic type has both static and dynamic typing natures. While it still allows potentially unsafe operations to be performed, it enforces static typing as far as members defined in the bound are concerned. So, the first two lines in the following code are still accepted (and checked at run time) but not the third[3] and fourth.

```
sc.x.anyField;
sc.x.anyMethod(...);
sc.x.contains(); // two arguments are expected!
Shape s = sc.x.contains(3, 4); // returns boolean!
```

In a real language, we do not expect programmers to write those upper bounds explicitly. Rather, when `dyn` is used as a type argument, the compiler can recover its upper bound automatically by assigning the upper bounds of the corresponding type parameters in the generic class definition.[4] For other uses of `dyn`, they can be regarded as `dyn<Object>`; in fact, we use `dyn` as an abbreviation of `dyn<Object>`, throughout the paper.

2.5 Two Compatibility Relations

As we have already mentioned, the type system of the surface language uses the compatibility relation, denoted by $\lesssim$, to check argument passing and assignments. This relation has both co- and contra-variant flavors when `dyn` is considered a top type: for example, both `Cell<Shape>` $\lesssim$ `Cell<dyn>` and `Cell<dyn>` $\lesssim$ `Cell<Shape>` hold and so both

```
Cell<Shape> c1 = ...; Cell<dyn> c2 = c1;
```

and

```
Cell<dyn> c1 = ...;  Cell<Shape> c2 = c1;
```

are accepted (statically). The reason to allow contravariance (the latter kind of compatibility) is simply because it *sometimes* runs safely. For example, when `c1` happens to be an

[3] It could be allowed in the presence of overloading.

[4] For F-bounded type variables, such automatic recovery is difficult. We would have to have programmers write upper bounds explicitly.

object `new Cell<Shape>(...)`, the latter code fragment is just fine.

However, we should not use this compatibility relation for casts. For example, consider the following (surface language) code:

```
Cell<dyn> c1 = new Cell<dyn>(new Polygon(...));
Cell<Rectangle> c2 = c1;       // accepted thanks
                               // to contravariance
c2.x.methodOnlyInRectangle(); // accepted since
                               // c2.x is Rectangle
```

On the second line, a run-time check $(\!|\texttt{Cell<Rectangle>}|\!)\,\texttt{c1}$ is performed. Since the run-time type of `c1` is `Cell<dyn>`, if $(\!|\ |\!)$ used the compatibility relation, the cast will succeed, resulting in the unexpected method-not-found error! (Notice that the invocation of `methodOnlyInRectangle` should involve no checks because the receiver's static type does not contain `dyn`.)

Thus, we use another relation $\prec$ called *run-time compatibility* for run-time checks. This relation is a subrelation of $\lesssim$ and disallows contravariance: for example, `Cell<dyn>` $\prec$ `Cell<Rectangle>` does not hold. However, it still allows covariance (such as `Cell<Rectangle>` $\prec$ `Cell<dyn>`), so it is more permissive than subtyping. (It is not completely safe—that is why we still need argument checks by $\texttt{e}_0\texttt{.m[}\texttt{T}_1,\ldots,\texttt{T}_n\texttt{|C<}\overline{\texttt{X}}\texttt{>](}\texttt{e}_1,\ldots,\texttt{e}_n\texttt{)}$.)

Having these discussions in mind, we formalize the core of the surface and intermediate languages in the following sections.

3. Featherweight GJ with Dynamic Types

In this section, we formalize the surface language FGJ$^{\texttt{dyn}}$, an extension of FGJ with dynamic types to model a type system of gradually typed generics. For simplicity, we omit some features found in FGJ: polymorphic methods and typecasts, which would be easy to add. As in FGJ, we also omit method overloading. We focus on the type system in this section and leave the definition of the intermediate language called FGJ$^{(\!|\ |\!)}$ and translation from FGJ$^{\texttt{dyn}}$ to FGJ$^{(\!|\ |\!)}$ to Section 4. For those who are familiar with FGJ, we use gray boxes to show main differences from FGJ.

3.1 Syntax and Lookup Functions

The abstract syntax of FGJ$^{\texttt{dyn}}$ classes, constructors and method declarations, and expressions are defined as follows:

Definition 1 (Syntax of FGJ$^{\texttt{dyn}}$)**.**

κ, ι ::= $\blacklozenge \mid \lozenge$

$\texttt{S}, \texttt{T}, \texttt{U}, \texttt{V}$ ::= $\texttt{X} \mid \texttt{N} \mid$ `dyn<N>`

$\texttt{N}, \texttt{P}, \texttt{Q}$ ::= $\texttt{C<}\overline{\texttt{T}}\texttt{>}$

$\texttt{e}$ ::= $\texttt{x} \mid \texttt{new N(}\overline{\texttt{e}}\texttt{)} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)}$

$\texttt{L}$ ::= $\texttt{class C<}\overline{\texttt{X}}^{\overline{\kappa}}\texttt{}\triangleleft\texttt{}\overline{\texttt{N}}\texttt{> }\triangleleft\texttt{ N }\texttt{\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}\ \texttt{K}\ \overline{\texttt{M}}\texttt{\}}$

$\texttt{K}$ ::= $\texttt{C(}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)\{super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;\}}$

$\texttt{M}$ ::= $\texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}}$

The metavariables `A`, `B`, `C`, `D` and `E` range over class names; `W`, `X`, `Y` and `Z` range over type variables; `N`, `P` and `Q` range over class types (`dyn<N>` is not a class type); `S`, `T`, `U` and `V` range over types; κ and ι range over kinds of type variables; `f` and `g` range over field names; `m` ranges over method names; `x` ranges over variables; `d` and `e` range over expressions; `L` ranges over class declarations; `K` ranges over constructor declarations; and `M` ranges over method declarations. We assume that the set of variables includes the special variable `this`.

We write $\overline{\texttt{f}}$ as shorthand for a possibly empty sequence $\texttt{f}_1\texttt{,f}_2\texttt{,...,f}_n$ (and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{X}}$, $\overline{\texttt{N}}$, $\overline{\texttt{T}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, etc.) and write $\overline{\texttt{M}}$ as shorthand for $\texttt{M}_1\ldots\texttt{M}_n$ (with no commas). The length of a sequence $\overline{\texttt{f}}$ is written $\#(\overline{\texttt{f}})$. We write $\texttt{f} \in \overline{\texttt{f}}$ when `f` equals to $\texttt{f}_i$ where $1 \le i \le \#(\overline{\texttt{f}})$, and write $\texttt{f} \notin \overline{\texttt{f}}$ otherwise. We also abbreviate various forms of sequences of pairs, writing "$\overline{\texttt{T}}\ \overline{\texttt{f}}$" as shorthand for "$\texttt{T}_1\ \texttt{f}_1\texttt{,...,T}_n\ \texttt{f}_n$" where $\#(\overline{\texttt{T}}) = \#(\overline{\texttt{f}})$, and similarly "$\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}$" for the sequence of declarations "$\texttt{T}_1\ \texttt{f}_1\texttt{;}\cdots\texttt{T}_n\texttt{f}_n\texttt{;}$", "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$" for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;}\cdots\texttt{this.f}_n\texttt{=f}_n\texttt{;}$", and "$\overline{\texttt{X}}^{\overline{\kappa}} \triangleleft \overline{\texttt{N}}$" for "$\texttt{X}_1^{\kappa_1} \triangleleft \texttt{N}_1\texttt{,...,X}_n^{\kappa_n} \triangleleft \texttt{N}_n$" . We write the empty sequence as $\bullet$ and denote concatenation of sequences using a comma. Sequences are assumed to contain no duplicate names. We abbreviate the keyword `extends` to the symbol $\triangleleft$.

`dyn<N>` is a type of dynamically typed expressions. Since it is not a class type, `dyn<N>` can neither be used to instantiate an object (by `new` expressions) nor be used as a bound of a type variable. We always write the bound `N` in the formal language.

A program in FGJ$^{\texttt{dyn}}$ is a pair $(CT, \texttt{e})$ of a class table, which is a finite mapping from class names `C` to class declarations `L`, and a closed expression corresponding to the body of the `main` method. We assume that CT satisfies some sanity conditions: (1) $CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{ ... \{...\}}$ for every $\texttt{C} \in \mathit{dom}(CT)$; (2) $\texttt{Object} \notin \mathit{dom}(CT)$; (3) for every class name `C` (except `Object`) appearing anywhere in CT, we have $\texttt{C} \in \mathit{dom}(CT)$; and (4) there are no cycles in the transitive closure of $\triangleleft$ (as a relation between class names). In what follows, we fix a class table.

As in FGJ, we use functions *fields* and *mtype* to look up field definitions and method types in a given class table. We also use a predicate *nomethod* to state non-existence of a method. We omit their straightforward definitions (see appendix in the full version of this paper or Igarashi, Pierce, and Wadler [18] for the definitions of *fields* and *mtype*); their functionalities are summarized in Table 2.

Some other auxiliary functions are defined in Figure 1. We use a function *bound* to compute the upper bound of a type in a bound environment Δ, which is a finite sequence of triples of a type variable, its kind and its bound, where type variables are pairwise distinct.[5] When *bound* is used with a class type, it returns the given type itself. When *bound* is used with a dynamic type, it returns the bound of the

[5] So, Δ can be considered a finite mapping.

$\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$	collects fields in class N and its super type.
$\mathit{mtype}(\mathtt{m}, \mathtt{N}) = \overline{\mathtt{T}} \to \mathtt{T}$	looks up the type of method m in class N or its super type.
$\mathit{nomethod}(\mathtt{m}, \mathtt{N})$	holds when there is no method m in class N or its super type.

Table 2. Definition of FGJ$^{\mathtt{dyn}}$: Auxiliary functions and predicates.

dynamic type. We have a function *kind* to look up kinds of type variables. We use a predicate *dynfree* to state that a type is dynamic-free, i.e., it contains no dynamic type.

3.2 Subtyping and Compatibility

Now we define subtyping and compatibility relations. As we have mentioned, there are two compatibility relations (written $\asymp$ for run-time compatibility and $\lesssim$ for static compatibility). We write $\Delta \vdash \mathtt{S} <: \mathtt{T}$ to mean S is a subtype of T under bound environment Δ. Similarly for $\Delta \vdash \mathtt{S} \asymp \mathtt{T}$ and $\Delta \vdash \mathtt{S} \lesssim \mathtt{T}$. We abbreviate a sequence of judgments $\Delta \vdash \mathtt{S}_1 <: \mathtt{T}_1, \ldots, \Delta \vdash \mathtt{S}_n <: \mathtt{T}_n$ to $\Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$ (and similarly for $\asymp$ and $\lesssim$).

Definition 2 (FGJ$^{\mathtt{dyn}}$ subtyping and compatibility)**.** The subtyping and compatibility judgments $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta \vdash \mathtt{S} \asymp \mathtt{T}$ and $\Delta \vdash \mathtt{S} \lesssim \mathtt{T}$ are defined by the rules in Figure 2.

The subtype relation <: is mostly the same as that of FGJ. The first two rules mean that it is reflexive and transitive; the third rule that a type variable is a subtype of its bound; the fourth rule is about inheritance-based subtyping—any instance of a $\triangleleft$ clause gives subtyping. The last rule says a bounded dynamic type dyn<N> is a subtype of its bound N. In fact, dyn<N> and N denote the same set of instances—instances of N and its subtypes and dyn<N> allows *more* operations (which are potentially unsafe, though) to be performed than N. So, dyn<N> <: N indeed agrees with the substitution principle [23].

The compatibility relations are defined with the help of the auxiliary relation $\prec$. Intuitively, $\mathtt{S} \prec \mathtt{T}$ means that T is obtained by replacing some class types in S with dynamic types (with an appropriate bound). For example, Rectangle $\prec$ dyn<Object> and Cell<Rectangle> $\prec$ Cell<dyn<Object>> hold (under any bound environment). So, $\prec$ represents a form of covariance. Then, the compatibility relations $\asymp$ and $\lesssim$ are defined as compositions of $\prec$ and <:—the former as $(<:; \prec)$ ($\cdot;\cdot$ is a composition of two relations) and the latter as $(\prec^{-1}; <:; \prec)$, where $\prec^{-1}$ is the inverse of $\prec$ and represents a form of contravariance. As a result, two types are statically compatible if replacing dynamic types with class types yields two types in the subtyping relation.

For example,

$$\Delta \vdash \texttt{Cell<dyn<Object>>} \lesssim \texttt{Cell<Rectangle>}$$

can be derived since Cell<dyn<Object>> $\prec^{-1}$ Cell<Rectangle>. Also,

$$\Delta \vdash \texttt{Cell<dyn<Shape>>} \lesssim \texttt{Cell<dyn<Object>>}$$

since Cell<dyn<Shape>> $\prec^{-1}$ Cell<Shape> and Cell<Shape> $\prec$ Cell<dyn<Object>>. (Note that Cell<dyn<Shape>> <: Cell<dyn<Object>> does *not* hold.)

dyn<Object> can be considered either a top type or a bottom type, i.e., $\Delta \vdash \mathtt{T} \lesssim$ dyn<Object> and $\Delta \vdash$ dyn<Object> $\lesssim \mathtt{T}$ are satisfied for any T, and the relation $\lesssim$ is not transitive because otherwise $\Delta \vdash \mathtt{S} \lesssim \mathtt{T}$ would be implied for any S,T (as mentioned in [27, 28]). dyn<N> can also be considered as a top/bottom type for subtypes of N.

3.3 Type Well-formedness and Typing

We, then, define well-formed types and typing.

Definition 3 (FGJ$^{\mathtt{dyn}}$ type well-formedness)**.** The type well-formedness judgment $\Delta \vdash \mathtt{T}$ ok, read as "in bound environment Δ, type T is well formed," is defined by the rules in Figure 3.

The last rule means that a bounded dynamic type is well formed if its upper bound is well formed. The third rule means that a class type is well formed if it has well formed type arguments that satisfy the corresponding type parameters' upper bounds. Note that we use $\asymp$ rather than <: or $\lesssim$. First, <: cannot be used because we want to use dynamic types as type arguments. For example, Cell<dyn<Object>> is well formed because dyn<Object> $\asymp$ Object. $\lesssim$ should not be used, either, because we want to reject a type like ShapeCell<Object>. (Note that Object $\lesssim$ Shape.) The third rule also requires that type arguments must be dynamic-free if the kind of the corresponding type variable is $\Diamond$.

Now we are ready to define typing. We use Γ as a type environment, which is a finite mapping from variables to types, written $\overline{\mathtt{x}}:\overline{\mathtt{C}}$.

Definition 4 (FGJ$^{\mathtt{dyn}}$ typing)**.** The type judgments $\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{e}:\mathtt{T}$, read as "in environment Δ and Γ, expression e has type T," and M OK IN C<$\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}$>, read as "method M is well-formed in class C<$\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}$>" and L OK, read as "class L is well-formed" are defined as in Figure 4.

Most of the rules are straightforward adaptation of those in FGJ [18], except that the relation $\lesssim$ is substituted for <:. TG-FIELD2 and TG-INVK2 are additional rules, used when the receiver type is a bounded dynamic type. Note that these rules are applied only when it is not known whether the receiver has a field or method to be accessed (the premises $\mathtt{f} \notin \overline{\mathtt{f}}$ in TG-FIELD2 and $\mathit{nomethod}(\mathtt{m}, \mathtt{N})$ in TG-INVK2).

Bound environment

$$\frac{\Delta(\mathtt{X}) = (\kappa, \mathtt{N})}{bound_\Delta(\mathtt{X}) = \mathtt{N}} \qquad bound_\Delta(\mathtt{N}) = \mathtt{N} \qquad bound_\Delta(\mathtt{dyn<N>}) = \mathtt{N} \qquad \frac{\Delta(\mathtt{X}) = (\kappa, \mathtt{N})}{kind_\Delta(\mathtt{X}) = \kappa}$$

Dynamic-free types

$$\frac{kind_\Delta(\mathtt{X}) = \Diamond}{dynfree_\Delta(\mathtt{X})} \qquad \frac{dynfree_\Delta(\overline{\mathtt{T}})}{dynfree_\Delta(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})}$$

Figure 1. Definition of FGJ$^{\mathtt{dyn}}$: Auxiliary functions and predicates.

Subtyping

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \qquad \frac{\Delta \vdash \mathtt{S} <: \mathtt{U} \qquad \Delta \vdash \mathtt{U} <: \mathtt{T}}{\Delta \vdash \mathtt{S} <: \mathtt{T}} \qquad \Delta \vdash \mathtt{X} <: bound_\Delta(\mathtt{X})$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}^{\overline{\kappa}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \mathtt{\{...\}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}} \qquad \Delta \vdash \mathtt{dyn<N>} <: \mathtt{N}$$

Compatibility

$$\Delta \vdash \mathtt{T} \prec \mathtt{T} \qquad \frac{\Delta \vdash \mathtt{S} \prec \mathtt{U} \qquad \Delta \vdash \mathtt{U} \prec \mathtt{T}}{\Delta \vdash \mathtt{S} \prec \mathtt{T}} \qquad \frac{\Delta \vdash \mathtt{S} <: \mathtt{U} \qquad \Delta \vdash \mathtt{U} \prec \mathtt{T}}{\Delta \vdash \mathtt{S} \nprec \mathtt{T}}$$

$$\frac{\Delta \vdash \overline{\mathtt{S}} \prec \overline{\mathtt{T}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{S}}\mathtt{>} \prec \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}} \qquad \frac{\Delta \vdash \mathtt{T} <: \mathtt{S} \qquad \Delta \vdash \mathtt{S} \prec \mathtt{N}}{\Delta \vdash \mathtt{T} \prec \mathtt{dyn<N>}} \qquad \frac{\Delta \vdash \mathtt{U} \prec \mathtt{S} \qquad \Delta \vdash \mathtt{U} \nprec \mathtt{T}}{\Delta \vdash \mathtt{S} \lesssim \mathtt{T}}$$

Figure 2. Definition of FGJ$^{\mathtt{dyn}}$: Subtyping and compatibility.

$$\Delta \vdash \mathtt{Object}\ \mathrm{ok} \qquad \frac{\mathtt{X} \in dom(\Delta)}{\Delta \vdash \mathtt{X}\ \mathrm{ok}} \qquad \frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}}^{\overline{\kappa}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \mathtt{\{...\}} \qquad \Delta \vdash \overline{\mathtt{T}}\ \mathrm{ok} \\ \forall \kappa_i \in \overline{\kappa}.(\ \kappa_i = \Diamond \text{ implies } dynfree_\Delta(\mathtt{T}_i)\ ) \\ \Delta \vdash \overline{\mathtt{T}} \nprec [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}\end{array}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \mathrm{ok}} \qquad \frac{\Delta \vdash \mathtt{N}\ \mathrm{ok}}{\Delta \vdash \mathtt{dyn<N>}\ \mathrm{ok}}$$

Figure 3. Definition of FGJ$^{\mathtt{dyn}}$: Type well-formedness.

In other words, TG-FIELD1 or TG-INVK1 can be used for expressions whose receiver type is `dyn<N>`, as long as the member is defined in the class definition of `N`.

The predicate *override*, used in method typing, is to check if a method `m` in class `N` can be overridden by a method of type $\overline{\mathtt{T}} \to \mathtt{T}$. Parameter types must be the same between the overriding and overridden methods[6] and the return type of the overriding method must be run-time compatible with that of the overridden method. We cannot use $\lesssim$ here: if $\lesssim$ were used, it would be possible to override a method that returns `T` by one that returns `dyn<Object>` in a subclass, and then to override it by another that returns `S` for *any* `S`. As a result, invoking a method, whose static return type is `T`, might actually invoke the third method that returns `S`, which can be very different from `T`, due to late binding!

We write $\vdash_{\mathrm{G}} (CT, \mathtt{e}) : \mathtt{T}$ to mean the program is well formed, i.e, if all the classes in CT are well formed and `e` is well typed under the empty environment.

We have not completed developing a type checking algorithm. In fact, it is not even clear that the compatibility relation $\lesssim$ is decidable for the same reason as variance-based subtyping [21].

Conservative Typing over FGJ. Even at this point, we can show an interesting property that typing in FGJ$^{\mathtt{dyn}}$ is a conservative extension of that in FGJ. Namely, as far as a class table written in the FGJ syntax is concerned, it is well typed under the FGJ rules if and only if it is well typed under the FGJ$^{\mathtt{dyn}}$ rules. The following lemma is a key to the conservative extension property (Theorem 6).

[6] This restriction can easily be relaxed by using the relation $\nprec$, but then we need a careful examination when we add method overloading.

Expression typing

$$\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{x} : \Gamma(\mathtt{x}) \ \text{(TG-VAR)} \qquad \frac{\Delta \vdash_{\mathrm{G}} \mathtt{N}\ \mathrm{ok} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta;\Gamma \vdash_{\mathrm{G}} \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta \vdash \overline{\mathtt{U}} \lesssim \overline{\mathtt{T}}}{\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}}) : \mathtt{N}}\ \text{(TG-NEW)}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0.\mathtt{f}_i : \mathtt{T}_i}\ \text{(TG-FIELD1)} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0 : \mathtt{dyn<N>} \qquad \mathtt{f} \notin \overline{\mathtt{f}} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0.\mathtt{f} : \mathtt{dyn<Object>}}\ \text{(TG-FIELD2)}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta;\Gamma \vdash_{\mathrm{G}} \overline{\mathtt{e}} : \overline{\mathtt{V}} \qquad \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{S}} \to \mathtt{S} \qquad \Delta \vdash \overline{\mathtt{V}} \lesssim \overline{\mathtt{S}}}{\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{S}}\ \text{(TG-INVK1)}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0 : \mathtt{dyn<N>} \qquad \Delta;\Gamma \vdash_{\mathrm{G}} \overline{\mathtt{e}} : \overline{\mathtt{V}} \qquad \mathit{nomethod}(\mathtt{m}, \mathtt{N})}{\Delta;\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{dyn<Object>}}\ \text{(TG-INVK2)}$$

Method typing

$$\frac{\mathit{mtype}(\mathtt{m}, \mathtt{N}) = \overline{\mathtt{U}} \to \mathtt{U} \text{ implies } \overline{\mathtt{T}} = \overline{\mathtt{U}} \text{ and } \Delta \vdash \mathtt{T} \curlywedge \mathtt{U}}{\mathit{override}_\Delta(\mathtt{m}, \mathtt{N}, \overline{\mathtt{T}} \to \mathtt{T})}$$

$$\frac{\Delta = \overline{\mathtt{X}^{\overline{\kappa}}} <: \overline{\mathtt{N}} \qquad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}\ \mathrm{ok} \qquad \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C<\overline{X}>} \vdash_{\mathrm{G}} \mathtt{e}_0 : \mathtt{S} \qquad \Delta \vdash \mathtt{S} \lesssim \mathtt{T} \qquad \mathtt{class\ C<\overline{X}^{\overline{\kappa}} \lhd \overline{N}> \lhd N\ \{...\}} \qquad \mathit{override}_\Delta(\mathtt{m}, \mathtt{N}, \overline{\mathtt{T}} \to \mathtt{T})}{\mathtt{T\ m(\overline{T}\ \overline{x})\{\ return\ e_0;\ \}\ OK\ IN\ C<\overline{X} \lhd \overline{N}>}}\ \text{(TG-METHOD)}$$

Class typing

$$\frac{\overline{\mathtt{X}^{\overline{\kappa}}} <: \overline{\mathtt{N}} \vdash \mathtt{N}, \overline{\mathtt{T}}, \overline{\mathtt{N}}\ \mathrm{ok} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C<\overline{X} \lhd \overline{N}>} \qquad \mathtt{K} = \mathtt{C(\overline{U}\ \overline{g}, \overline{T}\ \overline{f})\{super(\overline{g});\ this.\overline{f}=\overline{f};\}}}{\mathtt{class\ C<\overline{X}^{\overline{\kappa}} \lhd \overline{N}> \lhd N\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}\ OK}}\ \text{(TG-CLASS)}$$

Figure 4. Definition of FGJ$^{\mathtt{dyn}}$: Typing.

Lemma 5.

- If $\Delta \vdash \mathtt{S} \prec \mathtt{T}$ and $\mathit{dynfree}_\Delta(\mathtt{T})$, then $\mathtt{S} = \mathtt{T}$.
- If $\Delta \vdash \mathtt{S} \curlywedge \mathtt{T}$ and $\mathit{dynfree}_\Delta(\mathtt{T})$, then $\Delta \vdash \mathtt{S} <: \mathtt{T}$.
- If $\Delta \vdash \mathtt{S} \lesssim \mathtt{T}$ and $\mathit{dynfree}_\Delta(\mathtt{S})$, then $\Delta \vdash \mathtt{S} \curlywedge \mathtt{T}$.
- If $\Delta \vdash \mathtt{S} \lesssim \mathtt{T}$ and $\mathit{dynfree}_\Delta(\mathtt{S})$ and $\mathit{dynfree}_\Delta(\mathtt{T})$, then $\Delta \vdash \mathtt{S} <: \mathtt{T}$.

We write $\vdash_{\mathrm{FGJ}} (CT, \mathtt{e}) : \mathtt{T}$ if the program, which does not contain `dyn<N>`, is well formed under the FGJ rules.

Theorem 6 (FGJ$^{\mathtt{dyn}}$ Typing is Conservative over FGJ Typing)**.** If $\overline{\kappa} = \overline{\Diamond}$ for every `class C<`$\overline{\mathtt{X}^{\kappa}} \lhd \overline{\mathtt{N}}$`> ◁ N {...}` in CT and none of `dyn<P>` appears in $(CT, \mathtt{e})$, then $\vdash_{\mathrm{FGJ}} (CT, \mathtt{e}) : \mathtt{T} \iff \vdash_{\mathrm{G}} (CT, \mathtt{e}) : \mathtt{T}$.

4. From FGJ$^{\mathtt{dyn}}$ to FGJ$^{(\!(\,)\!)}$

In this section, we first define a formal model FGJ$^{(\!(\,)\!)}$ of the intermediate language, into which source programs are translated. FGJ$^{(\!(\,)\!)}$ has operational semantics, as well as a type system. After stating theorems about type safety of FGJ$^{(\!(\,)\!)}$, we present formal translation from FGJ$^{\mathtt{dyn}}$ to FGJ$^{(\!(\,)\!)}$ and state theorems that translation preserves typing. We have weak and strong versions of type safety: the weak version means that a well typed program can raise errors only at run-time checks and the strong version means that a well typed program without containing run-time checks never goes wrong in the usual sense.

4.1 The Target Language FGJ$^{(\!(\,)\!)}$

The syntax of FGJ$^{(\!(\,)\!)}$ extends that of FGJ$^{\mathtt{dyn}}$, by including special forms for run-time checks and run-time errors. We show only the grammar for expressions, values (used to define the semantics), and errors; the others are the same.

Definition 7 (Syntax of FGJ$^{(\!(\,)\!)}$)**.**

$$
\begin{array}{lcl}
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{new\ N(\overline{e})} \mid \mathtt{e.f} \mid \mathtt{e.m(\overline{e})} \\
& & \mid \mathtt{get(e,f)} \mid \mathtt{e.m[\overline{T}|C<\overline{X}>](\overline{e})} \\
& & \mid \mathtt{invoke(e,m,\overline{e})} \mid \mathtt{(\!(N)\!)e} \\
& & \mid \mathtt{Error[}\mathcal{E}\mathtt{]} \\
\mathtt{v,w} & ::= & \mathtt{new\ N(\overline{v})} \\
\mathcal{E} & ::= & \mathtt{NoSuchField} \mid \mathtt{NoSuchMethod} \\
& & \mid \mathtt{IllegalArgument} \mid \mathtt{BadCast}
\end{array}
$$

We avoid repeating the definitions of the following functions, predicates, and relations since they are defined exactly the same way as in FGJ$^{\mathtt{dyn}}$.

Functions	*bound*
	kind
	fields
	mtype
Predicates	*dynfree*
	nomethod
	override
Relations	Subtyping $<:$
	Compatibility $\prec$, $\nprec$
Judgments	Type well-formedness

Figure 5 introduces an auxiliary function $tyargs(\mathtt{N}, \mathtt{C})$, which is used in the reduction rules described later to get type arguments from run-time types. We also use a function $mbody(\mathtt{m}, \mathtt{N})$, which returns a pair $\overline{\mathtt{x}}.\mathtt{e}$ of a sequence of formal parameters and a method body expression, if N has m. Its straightforward definition is omitted.

We show the main typing rules of FGJ$^{(\!(\,)\!)}$ in Figure 6. An important point to note is that we use only the run-time compatibility $\nprec$ and no $\lesssim$ appears in the typing rules because a use of $\lesssim$ compiles to a cast $(\!(\,)\!)$, which uses $\nprec$ for its run-time check. TR-INVK1 is the rule for a method invocation without run-time checks. The receiver type must be dyn-free. TR-INVK2 is the rule for a method invocation with run-time argument checking. C<$\overline{\mathtt{T}}$> can be considered an initial (i.e., compile-time) static type of receiver $\mathtt{e}_0$; the condition N $\nprec$ C<$\overline{\mathtt{T}}$> is required since the receiver type may change as reduction proceeds. TR-INVK3 is the rule for a method invocation that checks whether the method m exists at run time. The receiver and arguments can be arbitrary typeable expressions and the type of the invocation is dyn<Object>. TR-CAST is the rule for casts; and TR-ERROR is the rule for errors.

We omit typing rules for object constructions, field accesses, methods and classes, since they are similar to those in FGJ$^{\mathtt{dyn}}$. We write $\vdash_{\mathrm{R}} (CT, \mathtt{e}) : \mathtt{T}$ to mean the FGJ$^{(\!(\,)\!)}$ program $(CT, \mathtt{e})$ is well formed.

We give main reduction rules in Figure 7. The evaluation order is, unlike FGJ, fixed to be left-to-right and call-by-value to deal with run-time errors more precisely. R-FIELD1 and R-INVK1 are quite standard. They check the existence of a field/method in the receiver type but the check should never fail for well-typed expressions as we will see later. R-FIELD2, R-INVK2, R-INVK3 and R-CAST are for run-time checks, which can raise a run-time error. In R-INVK2, the type arguments in method parameter types are filled in according to the run-time class of the receiver value, and checked against the run-time class of the actual arguments. In R-INVK3, we need to look up the method argument types by *mtype* to perform run-time checks for actual arguments. The rule R-CAST means that a cast succeeds when the subject type is run-time compatible with the target type.

We also have reduction rules for errors shown in Figure 8. Each rule has premises negating those in the corresponding reduction rule. Note that only run-time checks have error-raising reductions. We also need rules that propagate raised errors upwards, but we omit them.

FGJ$^{(\!(\,)\!)}$ is (weakly) type-safe in the sense that a well-typed program, if it terminates, yields a value or raises an error. Moreover, if a program does not contain dynamic types, then it is strongly type safe.

Theorem 8 (FGJ$^{(\!(\,)\!)}$ weak type safety)**.** If $\vdash_{\mathrm{R}} (CT, \mathtt{e}) : \mathtt{T}$ and $\mathtt{e} \longrightarrow^* \mathtt{e}'$ where $\mathtt{e}'$ is a normal form, then $\mathtt{e}'$ is either

1. a value v with $\bullet; \bullet \vdash_{\mathrm{R}} \mathtt{v} : \mathtt{N}$ and $\bullet \vdash \mathtt{N} \nprec \mathtt{T}$, or
2. an error Error[$\mathcal{E}$].

Theorem 9 (FGJ$^{(\!(\,)\!)}$ strong type safety)**.** If $\vdash_{\mathrm{R}} (CT, \mathtt{e}) : \mathtt{T}$ where $(CT, \mathtt{e})$ does not contain run-time checks and $\mathtt{e} \longrightarrow^* \mathtt{e}'$ where $\mathtt{e}'$ is a normal form, then $\mathtt{e}'$ is a value v with $\bullet; \bullet \vdash_{\mathrm{R}} \mathtt{v} : \mathtt{N}$ and $\bullet \vdash \mathtt{N} \nprec \mathtt{T}$.

These theorems are proved in a standard manner of combining subject reduction and progress [37]. The statements and proofs of both properties are, in fact, very similar to those for FGJ. One non-trivial property required is transitivity of $\nprec$.

Theorem 8 should be understood with the typing and error-raising reduction rules of FGJ$^{(\!(\,)\!)}$. First, in Figure 6, TR-INVK1 says that the receiver type of an ordinary method invocation is always dyn-free. Then, in Figure 8, Error[$\mathcal{E}$] is reduced only from run-time checks, not from an ordinary member access. Considering these and the subject reduction property together, we can see that no member access on dyn-free receiver raises an error.

4.2 Translation from FGJ$^{\mathtt{dyn}}$ to FGJ$^{(\!(\,)\!)}$

The judgment of translation from FGJ$^{\mathtt{dyn}}$ to FGJ$^{(\!(\,)\!)}$ is of the form $\Delta; \Gamma \vdash \mathtt{e} \rightsquigarrow \mathtt{e}' : \mathtt{T}$, read "FGJ$^{\mathtt{dyn}}$ expression e of type T under environments Γ and Δ translates to FGJ$^{(\!(\,)\!)}$ expression $\mathtt{e}'$." The translation is directed by typing in FGJ$^{\mathtt{dyn}}$. We show only the rules for method invocations in Figure 9.

$\langle \mathtt{S} \Leftarrow \mathtt{T} \rangle_{\Delta} \mathtt{e}$ inserts a cast when source type T is not run-time compatible with S. This reduces unnecessary casts. TRNS-INVK1 is for ordinary invocations; casts are inserted for testing run-time compatibility of arguments. If $\overline{\mathtt{V}}$ are dyn-free, then actually no casts will be inserted, thanks to Lemma 5. TRNS-INVK2 is for invocations whose arguments

$$tyargs(\texttt{C<}\overline{\texttt{T}}\texttt{>}, \texttt{C}) = \overline{\texttt{T}} \qquad \frac{\bullet \vdash \texttt{N} <: \texttt{P} \qquad tyargs(\texttt{P}, \texttt{C}) = \overline{\texttt{T}}}{tyargs(\texttt{N}, \texttt{C}) = \overline{\texttt{T}}}$$

Figure 5. Definition of FGJ$^{(\!(\,)\!)}$: Auxiliary function.

$$\frac{\Delta;\Gamma \vdash_{\mathrm{R}} \texttt{e}_0 : \texttt{T}_0 \qquad \Delta;\Gamma \vdash_{\mathrm{R}} \overline{\texttt{e}} : \overline{\texttt{V}} \qquad bound_{\Delta}(\texttt{T}_0) = \texttt{P} \qquad \Delta \vdash \texttt{P} \prec \texttt{N} \qquad dynfree_{\Delta}(\texttt{N}) \qquad mtype(\texttt{m}, \texttt{N}) = \overline{\texttt{S}} \to \texttt{S} \qquad \Delta \vdash \overline{\texttt{V}} \prec \overline{\texttt{S}}}{\Delta;\Gamma \vdash_{\mathrm{R}} \texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{)} : \texttt{S}} \text{ (Tr-Invk1)}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{R}} \texttt{e}_0 : \texttt{T}_0 \qquad \Delta;\Gamma \vdash_{\mathrm{R}} \overline{\texttt{e}} : \overline{\texttt{V}} \qquad bound_{\Delta}(\texttt{T}_0) = \texttt{N} \qquad \Delta \vdash \texttt{N} \prec [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{C<}\overline{\texttt{X}}\texttt{>} \qquad mtype(\texttt{m}, \texttt{C<}\overline{\texttt{X}}\texttt{>}) = \overline{\texttt{U}} \to \texttt{U} \qquad \Delta \vdash \overline{\texttt{V}} \prec [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{U}}}{\Delta;\Gamma \vdash_{\mathrm{R}} \texttt{e}_0\texttt{.m[}\overline{\texttt{U}}\texttt{|C<}\overline{\texttt{X}}\texttt{>](}\overline{\texttt{e}}\texttt{)} : [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{U}} \text{ (Tr-Invk2)}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{R}} \texttt{e}_0 : \texttt{T}_0 \qquad \Delta;\Gamma \vdash_{\mathrm{R}} \overline{\texttt{e}} : \overline{\texttt{V}}}{\Delta;\Gamma \vdash_{\mathrm{R}} \texttt{invoke(e}_0\texttt{, m, }\overline{\texttt{e}}\texttt{)} : \texttt{dyn<Object>}} \text{ (Tr-Invk3)} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{R}} \texttt{e} : \texttt{S}}{\Delta;\Gamma \vdash_{\mathrm{R}} (\!(\texttt{T})\!)\,\texttt{e} : \texttt{T}} \text{ (Tr-Cast)}$$

$$\Delta;\Gamma \vdash_{\mathrm{R}} \texttt{Error[}\mathcal{E}\texttt{]} : \texttt{T} \text{ (Tr-Error)}$$

Figure 6. Definition of FGJ$^{(\!(\,)\!)}$: Typing.

$$\frac{fields(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\texttt{new N(}\overline{\texttt{v}}\texttt{).f}_i \longrightarrow \texttt{v}_i} \text{ (R-Field1)} \qquad \frac{fields(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\texttt{get(new N(}\overline{\texttt{v}}\texttt{), f}_i\texttt{)} \longrightarrow \texttt{v}_i} \text{ (R-Field2)}$$

$$\frac{\bullet \vdash \texttt{N} \prec \texttt{T}}{(\!(\texttt{T})\!)\,\texttt{new N(}\overline{\texttt{v}}\texttt{)} \longrightarrow \texttt{new N(}\overline{\texttt{v}}\texttt{)}} \text{ (R-Cast)} \qquad \frac{mbody(\texttt{m}, \texttt{N}) = \overline{\texttt{x}}\texttt{.e}_0}{\texttt{new N(}\overline{\texttt{v}}\texttt{).m(}\overline{\texttt{w}}\texttt{)} \longrightarrow [\overline{\texttt{w}}/\overline{\texttt{x}}, \texttt{new N(}\overline{\texttt{v}}\texttt{)/this}]\texttt{e}_0} \text{ (R-Invk1)}$$

$$\frac{mbody(\texttt{m}, \texttt{N}) = \overline{\texttt{x}}\texttt{.e}_0 \qquad \overline{\texttt{w}} = \texttt{new }\overline{\texttt{P}}\texttt{(...)} \qquad \bullet \vdash \overline{\texttt{P}} \prec [tyargs(\texttt{N}, \texttt{C})/\overline{\texttt{X}}]\overline{\texttt{U}}}{\texttt{new N(}\overline{\texttt{v}}\texttt{).m[}\overline{\texttt{U}}\texttt{|C<}\overline{\texttt{X}}\texttt{>](}\overline{\texttt{w}}\texttt{)} \longrightarrow [\overline{\texttt{w}}/\overline{\texttt{x}}, \texttt{new N(}\overline{\texttt{v}}\texttt{)/this}]\texttt{e}_0} \text{ (R-Invk2)}$$

$$\frac{mbody(\texttt{m}, \texttt{N}) = \overline{\texttt{x}}\texttt{.e}_0 \qquad mtype(\texttt{m}, \texttt{N}) = \overline{\texttt{U}} \to \texttt{U} \qquad \overline{\texttt{w}} = \texttt{new }\overline{\texttt{P}}\texttt{(...)} \qquad \bullet \vdash \overline{\texttt{P}} \prec \overline{\texttt{U}}}{\texttt{invoke(new N(}\overline{\texttt{v}}\texttt{), m, }\overline{\texttt{w}}\texttt{)} \longrightarrow [\overline{\texttt{w}}/\overline{\texttt{x}}, \texttt{new N(}\overline{\texttt{v}}\texttt{)/this}]\texttt{e}_0} \text{ (R-Invk3)}$$

Figure 7. Definition of FGJ$^{(\!(\,)\!)}$: Reductions.

$$\frac{\bullet \vdash \texttt{N} \not\prec \texttt{T}}{(\!(\texttt{T})\!)\,\texttt{new N(}\overline{\texttt{v}}\texttt{)} \longrightarrow \texttt{Error[BadCast]}} \text{ (E-Cast)} \qquad \frac{fields(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \qquad \texttt{f} \notin \overline{\texttt{f}}}{\texttt{get(new N(}\overline{\texttt{v}}\texttt{), f)} \longrightarrow \texttt{Error[NoSuchField]}} \text{ (E-Field)}$$

$$\frac{nomethod(\texttt{m}, \texttt{N})}{\texttt{invoke(new N(}\overline{\texttt{v}}\texttt{), m, }\overline{\texttt{w}}\texttt{)} \longrightarrow \texttt{Error[NoSuchMethod]}} \text{ (E-Invk)}$$

$$\frac{\overline{\texttt{w}} = \texttt{new }\overline{\texttt{P}}\texttt{(...)} \qquad \bullet \vdash \texttt{P}_i \not\prec [tyargs(\texttt{N}, \texttt{C})/\overline{\texttt{X}}]\texttt{U}_i}{\texttt{new N(}\overline{\texttt{v}}\texttt{).m[}\overline{\texttt{U}}\texttt{|C<}\overline{\texttt{X}}\texttt{>](}\overline{\texttt{w}}\texttt{)} \longrightarrow \texttt{Error[IllegalArgument]}} \text{ (E-Invk-Arg1)}$$

$$\frac{mtype(\texttt{m}, \texttt{N}) = \overline{\texttt{U}} \to \texttt{U} \qquad \overline{\texttt{w}} = \texttt{new }\overline{\texttt{P}}\texttt{(...)} \qquad \bullet \vdash \texttt{P}_i \not\prec \texttt{U}_i}{\texttt{invoke(new N(}\overline{\texttt{v}}\texttt{), m, }\overline{\texttt{w}}\texttt{)} \longrightarrow \texttt{Error[IllegalArgument]}} \text{ (E-Invk-Arg2)}$$

Figure 8. Definition of FGJ$^{(\!(\,)\!)}$: Error-raising reductions.

Cast insertion

$$\langle \mathtt{S} \Leftarrow \mathtt{T} \rangle_{\Delta} \mathtt{e} \stackrel{\text{def}}{=} \begin{cases} \mathtt{e} & (\text{if } \Delta \vdash \mathtt{T} \prec\!\!\!\!\cdot\; \mathtt{S}) \\ (\!|\mathtt{S}|\!)\,\mathtt{e} & (\text{otherwise}) \end{cases}$$

Translation of method invocation

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 \rightsquigarrow \mathtt{e}_0' : \mathtt{T}_0 \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \rightsquigarrow \overline{\mathtt{e}}' : \overline{\mathtt{V}} \qquad \mathit{bound}_{\Delta}(\mathtt{T}_0) = \mathtt{N} \\ \mathit{dynfree}_{\Delta}(\mathtt{N}) \qquad \mathit{mtype}(\mathtt{m}, \mathtt{N}) = \overline{\mathtt{S}} \to \mathtt{S} \qquad \Delta \vdash \overline{\mathtt{V}} \lesssim \overline{\mathtt{S}}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \rightsquigarrow \mathtt{e}_0'.\mathtt{m}(\langle \overline{\mathtt{S}} \Leftarrow \overline{\mathtt{V}} \rangle_{\Delta} \overline{\mathtt{e}}') : \mathtt{S}} \text{ (TRNS-INVK1)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 \rightsquigarrow \mathtt{e}_0' : \mathtt{T}_0 \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \rightsquigarrow \overline{\mathtt{e}}' : \overline{\mathtt{V}} \qquad \mathit{bound}_{\Delta}(\mathtt{T}_0) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \\ \neg \mathit{dynfree}_{\Delta}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{X}}\mathtt{>}) = \overline{\mathtt{U}} \to \mathtt{U} \qquad \Delta \vdash \overline{\mathtt{V}} \lesssim [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \rightsquigarrow \mathtt{e}_0'.\mathtt{m}[\overline{\mathtt{U}}\,|\,\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}](\langle [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}} \Leftarrow \overline{\mathtt{V}} \rangle_{\Delta} \overline{\mathtt{e}}') : [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}} \text{ (TRNS-INVK2)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 \rightsquigarrow \mathtt{e}_0' : \mathtt{dyn<N>} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \rightsquigarrow \overline{\mathtt{e}}' : \overline{\mathtt{V}} \qquad \mathit{nomethod}(\mathtt{m}, \mathtt{N})}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \rightsquigarrow \mathtt{invoke}(\mathtt{e}_0', \mathtt{m}, \overline{\mathtt{e}}') : \mathtt{dyn<Object>}} \text{ (TRNS-INVK3)}$$

Figure 9. Translation from FGJ$^{\mathtt{dyn}}$ to FGJ$^{(\!|\,|\!)}$.

must be checked at run time. Note that the receiver type $\mathtt{T}_0$ in these two rules can be `dyn<N>` when there is an appropriate method `m` in `N` because the existence of `m` is statically guaranteed. TRNS-INVK3 is for invocations whose receiver type is `dyn<N>` and `N` has no appropriate method `m`.

Although we omit its definition, we write $(CT, \mathtt{e}) \rightsquigarrow (CT', \mathtt{e}')$ to mean that the FGJ$^{\mathtt{dyn}}$ program $(CT, \mathtt{e})$ is translated to the FGJ$^{(\!|\,|\!)}$ program $(CT', \mathtt{e}')$. Then, the translation preserves well-typedness, i.e., a well-typed FGJ$^{\mathtt{dyn}}$ program translates to a well-typed FGJ$^{(\!|\,|\!)}$ program.

Theorem 10 (Weak translation)**.** If $\vdash_{\mathrm{G}} (CT, \mathtt{e}) : \mathtt{T}$, then $(CT, \mathtt{e}) \rightsquigarrow (CT', \mathtt{e}')$ and $\vdash_{\mathrm{R}} (CT', \mathtt{e}') : \mathtt{T}$ for some $(CT', \mathtt{e}')$.

Theorem 11 (Strong translation)**.** If $\overline{\kappa} = \overline{\Diamond}$ for every `class` $\mathtt{C<}\overline{\mathtt{X}}^{\overline{\kappa}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}$ `{...}` in CT and none of `dyn<P>` appears in $(CT, \mathtt{e})$ and $\vdash_{\mathrm{G}} (CT, \mathtt{e}) : \mathtt{T}$, then $(CT, \mathtt{e}) \rightsquigarrow (CT', \mathtt{e}')$ and $\vdash_{\mathrm{R}} (CT', \mathtt{e}') : \mathtt{T}$ for some $(CT', \mathtt{e}')$ and $(CT', \mathtt{e}')$ does not contain run-time checks.

Combining Theorem 8 and Theorem 10, we can see that a member access which translates to an ordinary member access without run-time checks will not fail. In other words, a statically typed portion (method invocations whose receiver and argument types are dyn-free) will never fail. This is the type safety of FGJ$^{\mathtt{dyn}}$.

5. Implementation

In this section, we report the basic implementation scheme for Java. Our plan is to add a new compilation phase to transform a code tree to have the run-time checks inserted as the same way as the other existing compilation phases such as the type erasing transformation. The transformation follows the rules in Section 4 and the run-time checks can be implemented using existing reflective features of Java. Since the current JVM has no run-time information of type arguments of generics, we need a mechanism to look up full run-time type information. We follow the technique of type passing [32–34] for this.

5.1 Run-time Checks

The run-time checks $(\!|\mathtt{T}|\!)\,\mathtt{e}$, `get(e,f)` and `invoke(e,m,`$\overline{\mathtt{e}}$`)` can be implemented by reflection APIs such as `java.lang.Class`, `java.lang.reflect.Field` and `java.lang.reflect.Method`. We implement a class `Cla` to represent type descriptors and a static method `Cla.$()` to get the run-time type information of `obj` including type arguments. The return value of `Cla.$(...)` is a type descriptor `d` (an instance of class `Cla` described in more detail later), which has field `cl` of a `java.lang.Class` instance, field `p` of an array of type descriptors of type arguments, and `h` of an array of a superclass chain, where `d.h[0]` is `d` itself and `d.h[d.h.length-1]` is `Object`.

The cast $(\!|\mathtt{T}|\!)\,\mathtt{e}$, which corresponds to R-CAST, can be implemented as the following method `cast()`, which casts `obj` to class `klass`.

```
Object cast(Cla klass, Object obj) {
  if ((obj instanceof Parametric &&
       isRuntimeCompatible(Cla.$(obj), klass)) ||
      klass.cl.isInstance(obj)) {
    return obj;
  } else throw new ClassCastException();
}
```

If the type of `obj` has type arguments, then `isRuntimeCompatible()` method checks if the type of `obj` and the target type of the cast satisfy the relation $\prec\!\!\!\!\cdot$. Otherwise, the cast acts as a normal one, which uses the subtype relation $<:$. The argument type checking for

`e.m[`$\overline{\texttt{T}}$`|C<`$\overline{\texttt{X}}$`>](`$\overline{\texttt{e}}$`)` in R-INVK2 can also be done by this method. Although not proved, we conjecture that the explicit transitivity rule for $\prec$ is actually redundant. Without transitivity, it is easy to check run-time compatibility.

Since `⟪T⟫` is inserted by the translation and types $\overline{\texttt{T}}$ in `e.m[`$\overline{\texttt{T}}$`|C<`$\overline{\texttt{X}}$`>](`$\overline{\texttt{e}}$`)` are specified by the translation, the target type of a cast can be determined mostly at compile time, except that the type arguments for $\overline{\texttt{X}}$ have to be filled at run time.

The implementation of `get(e, f)`, which corresponds to R-FIELD2, is quite easy. We have `Object.getClass()` to get an instance of `java.lang.Class`, which has method `getField()`, which looks up a field by a field name and throws `NoSuchField` exception when no field is found. The return value of `getField()` is an instance of `java.lang.reflect.Field`, which has method `get()`, which retrieves the field value from the receiver.

```
Object get(Object r, String f) {
  Field fld = r.getClass().getField(f);
  return fld.get(r);
}
```

The special form `invoke(e,m,`$\overline{\texttt{e}}$`)`, which corresponds to R-INVK3, can be implemented as the method `invoke()` below. Suppose `Met` is a class for parameter types of a method, which has field `m` of `java.lang.reflect.Method` and field `params` of method type descriptors. Also suppose `mtypes()` is a method to look up method signatures by a method name.

```
Object invoke(Object r, String m, Object[] args) {
  Met[] mets = mtypes(r, m);
  M: for (int i=0, l=mets.length; i < l; i++) {
    if (mets[i].args.length != args.length)
      continue M;
    for (int j=args.length-1; 0 <= j; j--) {
      try {
        mets[i].params[j].cast(args[j]);
      } catch (ClassCastException e) {
        continue M;
      }
    }
    return mets[i].m.invoke(r, args);
  }
  throw new NoSuchMethodException();
}
```

This method looks up methods named `m`, and calls the first method[7] whose parameter types match the argument types. `cast()` is used to match parameters and arguments. If the casts succeed, then `java.lang.Method.invoke` API performs real invocation of the method. Otherwise, it throws `NoSuchMethodException`. Note that the target type of the cast must be determined at run time because we have no information of the receiver type at compile time.

[7] We are not considering method overloading here for simplicity.

5.2 Information on Type Arguments at Run Time

As we have already seen, we need information on type arguments at run time. It requires additional memory usage and time costs, but a relatively high performance technique is proposed by Viroli et al. [32–34]. We quickly review this technique.

The basic idea of this technique is to pass type information as a field of an object by transforming the code. For example, the following code describes how the transformation goes.

```
Cell<Shape> c
  = new Cell<Shape>(new Rectangle(...));
// Cell<Shape> c
//   = new Cell<Shape>(
//       new Cla(Cell.class,
//               new Cla[]{
//                 new Cla(Shape.class) }),
//       new Rectangle(...));
```

The first line is the original code, and the comment is the translated code. The instance of `Cla`, which is a type descriptor class, is passed as a first argument of the constructor. Of course, the definition of class `Cell` is also transformed to receive a type descriptor at the constructor, and to implement an interface, which provides access to the type descriptor.

The transformation described above is not optimized at all: a new type descriptor is generated every time the constructor is called. So, a mechanism to reduce the number of generations of type descriptors to one for a distinct type, using double hashing, is reported in [32–34].

5.3 Preliminary Benchmark Evaluation

Since we have no working compiler yet, we only give benchmarks for each run-time check with minimal hand-translated code using those checks separately. These may help to see if our implementation idea is reasonable and how much dynamic types slow down execution of the program. Execution of dynamically typed code is quite expensive especially for method invocations on a receiver of dynamic type, but we believe that the cost is not unacceptable.

Benchmarks for each run-time check is shown in Table 3. For each run-time check, we used test code including a single expression with a run-time check and the same expression without it. Each test code iterates 100/10000/1000000 times and overall time consumptions are listed.

For the expression `⟪T⟫e`, we used a static type for the target type of the cast. The cast looks into type arguments but we can say it is not so expensive according to the result. We can say the same thing about the run-time check in the expression `e.m[`$\overline{\texttt{T}}$`|C<`$\overline{\texttt{X}}$`>](`$\overline{\texttt{e}}$`)` since it only needs a cast for each argument. The run-time check in the expression `get(e,f)` is relatively expensive because it uses a feature of `java.lang.reflect.Field`. The run-time check in the expression `invoke(e,m,`$\overline{\texttt{e}}$`)` is quite expensive but the cost

# of iterations	100	10000	1000000
⟪T⟫e	0.006	0.019	0.125
e	0.004	0.008	0.087
get(e,f)	0.003	0.043	0.356
e.f	0.001	0.003	0.056
e.m[$\overline{\text{T}}$|C<$\overline{\text{X}}$>]($\overline{\text{e}}$)	0.000	0.003	0.072
e.m($\overline{\text{e}}$)	0.000	0.002	0.080
invoke(e,m,$\overline{\text{e}}$)	0.007	0.093	3.430
e.m($\overline{\text{e}}$)	0.000	0.003	0.069

Table 3. Execution time of each run-time check (sec.)

seems somewhat inevitable since it must resolve a method signature and check run-time types of the arguments.

In Table 4, we have tested casts with more complex target types. We used a dynamic type argument for the target type. The dynamic type argument has a bound, which is a class type possibly including another dynamic type argument. We count the nested dynamic type argument as depth and the result for each target type of the depth is listed in the rows.

Depth(s) \ # of iterations	100	10000	1000000
1	0.000	0.004	0.105
2	0.000	0.007	0.171
3	0.001	0.009	0.235
4	0.001	0.008	0.268
5	0.001	0.009	0.301

Table 4. Execution time of casts (sec.)

We can conclude that the operation for checking the run-time compatibility $\prec$ is not too expensive in comparison with the other run-time checks.

6. Related Work

There is much work on mixing dynamic and static types (see, for example, Siek and Taha [27, 28] for a more extensive survey). Here, we compare our work mainly with related work on object-oriented programming languages and parametric polymorphism.

We first review proposals to apply static type checking to dynamically typed languages. Bracha and Griswold [6] have proposed *Strongtalk*, which is a typechecker for a downward compatible variant of Smalltalk, a dynamically typed class-based object-oriented programming language. The type system of *Strongtalk* is structural and supports subtyping and generics but does not accept partially typed programs. Thiemann [30] has proposed a type system for (a subset of) JavaScript, which is a prototype-based object-oriented language, to avoid some kinds of run-time errors by static type checking. Furr, An, Foster, and Hicks [11] have developed Diamondback Ruby, an extension of Ruby with a static type system. Their type system, which seems useful to find bugs, however, does not offer static type safety.

Anderson and Drossopoulou [3] have proposed a type system for (a subset of) JavaScript for the evolution from JavaScript to Java. Although it is nominal and concerned with script-to-program evolution, their type system does not have subtyping, inheritance, or polymorphism; moreover, this work is not concerned about safety of partially typed programs in the middle of the evolution.

Lagorio and Zucca [22] have developed Just, an extension of Java with unknown types. Although there is some overlap in the expected uses of this system and gradual typing, the main purpose of unknown types is to omit type declarations; possibly unsafe use of unknown types is rejected by the type system. They use reflection to implement member access on unknown types.

Gray, Findler, and Flatt [14] have implemented an extension of Java with dynamic types and contract checking [9] for interoperability with Scheme. They mainly focus on the design and implementation issues and give no discussion on the interaction with generics. Their technique to implement reflective calls can be used for our setting.

Gray [12, 13] studied mixing Java as a statically typed language and JavaScript as a dynamically typed scripting language. Unlike our language, classes with dynamically typed methods are allowed to inherit from a class that is statically typed, and vice versa.

As we have already mentioned, Siek and Taha have studied gradual typing for Abadi-Cardelli's object calculus [28]. However, the language is object-based (as opposed to class-based) and parametric polymorphism is not studied. Another point is that the implementation of run-time checks for class-based languages seems easier than that for object-based languages, since, in class-based languages, every value is tagged with its run-time type information and the check can be performed in one step (unlike higher-order contract checking, which checks inputs to and outputs from functions separately).

Sage [15], a functional language based on hybrid type checking [10], supports both parametric polymorphism and dynamic types. Matthews and Ahmed [24] and, more recently, Ahmed, Findler, Siek, and Wadler [2] give theoretical accounts for the combination of impredicative polymorphism with dynamic typing. In all of these works, a dynamic type is compatible (in our terminology) with universal types whereas there is no counterpart of universal types in our setting. None of them has addressed *bounded* polymorphism.

Wrigstad, Nardelli, Lebresne, Östlund and Vitek [5, 38] have developed a language called Thorn, which integrates static and dynamic types in a different way. They have introduced the notion of *like types*, which interface between statically and dynamically typed code. A variable of `like C` is treated as type `C` at compile time but any value can flow into the variable at run time (subject to run-time checks).

This is different from dyn<C>, which allows any operations statically but only subtypes of C can flow into a dyn<C>.

Bierman, Meijer and Torgersen[4] added dynamic types to C♯. They also translate a program of the surface language into intermediate code, which has explicit run-time checks. In their setting, dynamic types can be arguments of generic classes, but their subtype relation is only invariant with respect to type parameters, so, for example, it is not possible to pass Cell<Rectangle> to Cell<dyn>.

There is some work on applying the gradual approach to advanced type systems for object-oriented programming languages. Wolff, Garcia, Tanter and Aldrich [36] developed a gradual system for typestate-oriented programming, where operations on an object are restricted by the internal state of the object. Potentially safe code which rejected by other typestate-oriented type systems can be accepted in their system thanks to a dynamic type. Sergey and Clarke [26] proposed a gradual system for ownership types, which represent encapsulation of objects and are used as invariants in type checking. Their gradual system allows a programmer to omit ownership annotations without losing encapsulation properties of annotated code.

There are some related features that already exist in Java. Wildcards, studied by Igarashi and Viroli [17] and Torgersen, Ernst, Hansen, Ahé, Bracha and Gafter [31], enable a flexible subtyping with both co- and contra-variant parametric types, though only statically resolved members can be accessed on a receiver of a wildcard type and it is not allowed to specify a wildcard type as a type argument in a new expression. Raw types are proposed by Bracha, Odersky, Stoutamire, and Wadler [7] to deal with compatibility between legacy monomorphic Java code and new polymorphic Java code. A generic class C<$\overline{\mathtt{X}}$> can be used as a raw type C, without type arguments, and assigning a value of C<$\overline{\mathtt{T}}$> to a variable of C, or even creating an instance of C via new expression are allowed. This behavior is similar to FGJ$^{\mathtt{dyn}}$'s if we consider C to be an abbreviation of C<dyn, ..., dyn>. However, with raw types, even statically typed code can go wrong.

7. Conclusions

We have designed a language that combines dynamic types and generics. The language allows dynamic types to be used as type arguments of a generic class and realizes smooth interfacing between dynamically and statically typed code thanks to the flexible compatibility relation. We have introduced bounded dynamic types to deal with the case where a type parameter has an upper bound.

The language is formalized as a minimal core model of Java including the feature of generics. As in other gradual type systems, we have proved safety properties, which ensure that statically typed parts in a program never go wrong.

We have also reviewed the sketch of an implementation scheme, which is an idea to develop a gradually typed Java compiler by extending an existing Java compiler without modifying JVM.

Future Work. Our run-time compatibility does not allow argument passing, for example, from Cell<dyn> to Cell<Rectangle>. It might be too early to abort the execution at this point since the value in the Cell may not be used at all. We think we can relax the restriction by deferring the check until the field is accessed. Then, we need a blame assignment system [2, 35] for precise error reports. Implementation of the blame tracking system for this design we think is done by "wrapping" an object with another object. The wrapping object has the same interface as the original one, but it performs a run-time check on every member access. In a simple setting, a subclass of the class of the original object suffices for the class of the wrapping object. However, there are at least two problems in this implementation strategy for Java-like languages. First, we need some mechanism to "wrap" an ordinary field access. Second, a wrapping class cannot extend a final class.

When bytecode of a generic library class compiled by the standard Java compiler is used with client bytecode compiled by our compiler, the client is not allowed to use dynamic type arguments to the generic class since type variables in the generic class are declared without kind ♦. We think that converting the library bytecode at load time will help to relax this restriction.

We also plan to investigate the interactions of dynamic types with other features such as overloading to make the language more realistic.

Acknowledgments

We thank the anonymous reviewers very much for thoughtful and thorough reviews. Ina is a Research Fellow of the Japan Society of the Promotion of Science. This work was supported in part by Grant-in-Aid for Young Scientists (A) No. 21680002 and by Grant-in-Aid for JSPS Fellows No. 10J06019.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Prog. Lang. Syst.*, 13(2):237–268, 1991.

[2] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *Proc. of ACM POPL'11*, pages 201–214, Austin, TX, Jan. 2011.

[3] C. Anderson and S. Drossopoulou. BabyJ - from object based to class based programming via types. In *Proc. of WOOD'03*, volume 82 of *Elsevier ENTCS*, pages 53–81, 2003.

[4] G. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C♯. In *Proc. of ECOOP'10*, volume 6183 of *Springer LNCS*, pages 76–100, Maridbor, Slovenia, June 2010.

[5] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn — robust, con-

current, extensible scripting on the JVM. In *Proc. of ACM OOPSLA'09*, pages 117–136, Orlando, FL, Oct. 2009.

[6] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proc. of ACM OOPSLA'93*, pages 215–230, 1993.

[7] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of ACM OOPSLA'98*, pages 183–200, Vancouver, BC, Oct. 1998.

[8] R. Cartwright and M. Fagan. Soft typing. In *Proc. of ACM PLDI'91*, pages 278–292, 1991.

[9] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proc. of ACM ICFP'02*, pages 48–59, 2002.

[10] C. Flanagan. Hybrid type checking. In *Proc. of ACM POPL'06*, pages 245–256, Charleston, SC, Jan. 2006.

[11] M. Furr, J. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *Proc. of ACM Symposium on Applied Computing (SAC'09)*, pages 1859–1866, Mar. 2009.

[12] K. E. Gray. Safe cross-language inheritance. In *Proc. of ECOOP'08*, volume 5142 of *Springer LNCS*, pages 52–75, 2008.

[13] K. E. Gray. Interoperability in a scripted world: Putting inheritance & prototypes together. In *Proc. of FOOL'10*, Oct. 2010.

[14] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *Proc. of ACM OOPSLA'05*, pages 231–245, 2005.

[15] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Proc. of the Scheme and Functional Programming Workshop*, pages 93–104, Sept. 2006.

[16] F. Henglein. Dynamic typing. In *Proc. of ESOP'92*, volume 582 of *Springer LNCS*, pages 233–253, Rennes, France, Feb. 1992.

[17] A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Prog. Lang. Syst.*, 28(5):795–847, Sept. 2006.

[18] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Syst.*, 23(3):396–450, May 2001.

[19] L. Ina and A. Igarashi. Towards gradual typing for generics. In *Proc. of STOP'09*, pages 17–29, Genova, Italy, July 2009. Available also in the ACM Digital Library.

[20] L. Ina and A. Igarashi. Gradual typing for Featherweight Java. *Computer Software*, 26(2):18–40, Apr. 2009. In Japanese.

[21] A. J. Kennedy and B. C. Pierce. On decidability of nominal subtyping with variance. In *Proc. of FOOL/WOOD'07*, Nice, France, Jan. 2007.

[22] G. Lagorio and E. Zucca. Just: safe unknown types in Java-like languages. *Journal of Object Technology*, 6(2):69–98, Feb. 2007.

[23] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[24] J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing, or, thorems for low, low prices! In *Proc. of ESOP'08*, volume 4960 of *Springer LNCS*, pages 16–31, 2008.

[25] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. of ACM POPL'97*, pages 146–159, Jan. 1997.

[26] I. Sergey and D. Clarke. Towards gradual ownership types. In *Proc. of IWACO'11*, Lancaster, UK, July 2011.

[27] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proc. of the Scheme and Functional Programming Workshop*, pages 81–92, Sept. 2006.

[28] J. G. Siek and W. Taha. Gradual typing for objects. In *Proc. of ECOOP'07*, volume 4509 of *Springer LNCS*, pages 2–27, 2007.

[29] S. Thatte. Quasi-static typing. In *Proc. of ACM POPL'90*, pages 367–381, Jan. 1990.

[30] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Proc. of ESOP'09*, volume 3444 of *Springer LNCS*, pages 408–422, 2005.

[31] M. Torgersen, E. Ernst, C. P. Hansen, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, Dec. 2004. Special issue: OOPS track at SAC 2004.

[32] M. Viroli. A type-passing approach for the implementation of parametric methods in Java. *The Computer Journal*, 46(3): 263–294, 2003.

[33] M. Viroli. Effective and efficient compilation of run-time generics in Java. In *Proc. of WOOD'04*, volume 138 of *Elsevier ENTCS*, pages 95–116, 2004.

[34] M. Viroli and A. Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. In *Proc. of ACM OOPSLA'00*, pages 146–165, Oct. 2000.

[35] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. of ESOP'09*, volume 5502 of *Springer LNCS*, pages 1–16, York, UK, Mar. 2009.

[36] R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In *Proc. of ECOOP'11*, volume 6813 of *Springer LNCS*, pages 459–483, Lancaster, UK, July 2011.

[37] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994.

[38] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proc. of ACM POPL'10*, pages 377–388, 2010.