DuctileScala: Combined static and dynamic feedback for Scala

CSE 501 Project Report

Ricardo Martin
University of Washington
rmartin@cs.washington.edu

Daniel Perelman
University of Washington
perelman@cs.washington.edu

Jinna Lei
University of Washington
jinna@cs.washington.edu

Brian Burg
University of Washington
burg@cs.washington.edu

ABSTRACT

Programmers receive feedback about program correctness in several ways. The most common *static feedback* is typechecking: if a program typechecks successfully, then all program executions are guaranteed to be free of certain classes of errors. *Dynamic feedback* is obtained by running a program and observing the output of a single program execution. Dynamically-typed programs can yield dynamic feedback at any time but are never able to provide meaningful static feedback, whereas statically-typed programs can only yield dynamic feedback (that is, be executed) after they properly typecheck. Recent research [6] has investigated the possibility of obtaining static and dynamic feedback at any point and in any order by deferring the typechecking normally performed during compilation of Java programs. This in essence temporarily disables the typechecker: as in dynamically-typed languages, the program will only fail at runtime (as opposed to failing statically with a type error).

We push this concept further by targeting Scala, a hybrid object-oriented and functional language with an expressive and powerful type system. What happens if one "temporarily disables" a sophisticated typechecker? Is it still possible to obtain meaningful dynamic feedback? We describe new techniques and challenges in deferring typechecking until runtime of (possibly type-incorrect) Scala programs. We informally describe an alternate semantics for executing type-incorrect programs, and explore the technical feasibility of *a detyping transformation* for use on other real-world code.

1. INTRODUCTION

Usually, developers have to choose between statically-typed and dynamically-typed languages, both of which have advantages and disadvantages in different stages of development. It is always possible to obtain dynamic feedback from programs written in a dynamically-typed language. In such languages, semantic errors halt program execution only *at the moment they are encountered*. This is in contrast to static type systems which prevent such semantic errors at compile-time along all program paths. Consider some method `foo` of object `A`. If `A.foo` is invoked in a dynamically-typed language, the presence of method `foo` in `A` is checked immediately prior to calling the method at runtime, whereas a statically-typed language would ensure the presence of `A.foo` during program compilation. The upside of dynamically checking the existence of `foo` is pure flexibility: only those methods that are actually needed are checked, and as a result errors can only originate from code that was (attempted to be) executed. This flexibility is most helpful during prototyping when algorithmic and architectural details change rapidly, and where the main goal is to explore a design, or to quickly obtain correct output.

Statically-typed languages provide certain correctness guarantees to type-correct (that is, typecheckable) programs. Checking the existence of `foo` at compile-time means that the program is guaranteed to never encounter a *method not found* error at runtime. Type annotations commonly serve as a kind of code documentation, and also aid in automated program verification of properties beyond those checked by the compile-time typechecker. Lastly, static type information can be used by the compiler to improve performance: for example, many runtime checks can be elided safely. However, statically-typed languages require that the programmer present a type-correct program at all stages of development in order to receive dynamic feedback (execute the program). This fixes the order of feedback to be first static, then dynamic.

Ultimately, only the programmer knows which sort of feedback (static or dynamic) is more valuable at any given time. We can unfix the order of feedback; by *temporarily disabling the type checker* for a statically-typed language, the programmer can run the program as if it were dynamically typed. As with an equivalent program written in a dynamically typed language, only the executed program paths must be free of semantic errors. The programmer can run the typechecker to get whole-program static feedback when desired, and can run the program to get dynamic feedback on a subset of their program free of semantic errors. For ex-

ample, after refactoring an interface that is implemented by several modules, the test suite could be run for each module when its respective refactoring is done, without having to finish the refactoring on all the modules that implement the interface. If the developer realizes that the refactoring actually was a bad idea, s/he wastes much less time on the aborted refactoring.

This approach has already been implemented for Java in DuctileJ [6] with promising results. *We hypothesize that putting the developer in control of when to obtain dynamic and static feedback will be even more useful when dealing with a more complex type system.* Programs can fail a complex typechecker in many more ways, and discerning to which type system feature a type error is related takes a long time and delays testing and other dynamic feedback. Another setback is that complex type systems often produce hard to understand error messages which further obscures the nature of type errors [19]. It might be the case that a dynamically-typed version of type-incorrect program has correct runtime behavior, but is rejected at compile-time for reasons the programmer does not understand.

Before our hypothesis can be tested, we first need to answer the question of whether "temporarily disabling the type checker" is possible for a language more complex than Java. The Scala programming language [11] is a hybrid object-oriented and functional programming language that builds on Java and other languages. The close relationship between Scala and Java allows us to provide a meaningful comparison.

Contribution

We present DuctileScala, a system for running statically-typed Scala programs as if they were dynamically-typed. This enables a programmer to run a type-incorrect Scala program to obtain dynamic (execution) feedback before static feedback from the typechecker. Section 2 briefly outlines Scala's language features. Section 3 further details the design of our de-typing transformation, and explains salient language and type system features as needed. Section 4 describes our transformation's implementation at compile-time and runtime. Section 5 describes our methodology, Section 6 presents preliminary results, and Section 7 explores alternative ways of combining advantages of static and dynamic typing. Section 8 describes future work, and Section 9 concludes.

2. SCALA

Scala is a statically-typed programming language that explicitly supports both object-oriented and functional programming paradigms. It runs on the Java Virtual Machine (JVM) and .NET Common Language Runtime (CLR), and interoperates cleanly with legacy Java/C# code.

2.1 Scala is object-oriented

Unlike Java, Scala is a pure object-oriented language: all values are objects. The compiler transparently handles the boxing and unboxing of values to provide a consistent object abstraction for programmers, but retains the performance benefits of Java primitives. Behaviors in Scala are specified with classes and traits, and are extended via single inheritance subclassing and mixin-like composition of traits. Static methods are unsupported; instead, Scala supports singleton objects (also called module objects). Classes, modules, functions, and packages can be nested arbitrarily. Examples of each are available online [12].

2.2 Scala is functional

As in classical functional programming languages, functions are first-class values in Scala. Also supported are features such as higher-order functions, currying, sequence comprehensions, lazy values, call-by-name evaluation, and pattern matching over extractors.

2.3 Typing Scala

The integration of these paradigms requires a very expressive type system. The type system supports generic classes, polymorphic methods, implicit arguments, implicit type conversions, higher-order types, higher-order functions, type refinements, variance annotations, explicit class selftypes, abstract type members, path-dependent types, and other features. Scala also allows cyclic references in type definitions. General type inference on first-order and some higher-order types can ease the burden of explicit type annotations.

2.4 Method Dispatch

Scala uses argument types and expected return type to select a specific method implementation. The dispatch algorithm filters implementation candidates based on the shapes of the arguments and return value, and among those chooses the "most specific" candidate. Complex rules involving argument types and the return type gover the relative specificity of implementation candidates.

2.5 Implicit Formal Parameters

Scala methods allow implicit parameters, marked with the `implicit` keyword. These parameters which do not need explicit arguments at invocation. If the caller does not explicitly provide such arguments, the compiler will automatically insert values of the matching type found in the current lexical scope. Only values marked with the `implicit` keyword and possessing the correct type are eligible to "substitute in" as an implicit argument. As with normal arguments, it is an error if no acceptable implicit argument is provided to the method call (or if there is ambiguity as to which value to substitute).

2.6 Views (Implicit Conversions)

Views, also known as implicit type conversions, are special functions that are automatically applied by the compiler if necessary. For example, suppose there is a type conversion from `A` to `B`, named `AtoB`. If some function is expecting an argument of type `B`, but is given an argument of type `A`, the compiler automatically inserts `AtoB` in order to make the function call typecheck.

3. DETYPING SCALA

With so many complex language features, we must pick a subset of Scala's language features to initially focus on. Below, we describe transformations for a subset[1] of Scala lan-

[1]For a more thorough account of the Scala language features supported by DuctileScala, consult Table 1.

guage features that are necessary to execute any Scala program.

Our detyping transformation conceptually modifies the program AST after parsing and before the compiler's normal typechecking phase. This transformation *defers* typechecking operations until runtime. The Scala compiler serializes some type and symbol information in class files to support separate compilation. This information is accessible at runtime and can be repurposed to reflectively emulate method dispatch decisions that would have been made by the compiler (provided the same information).

Not all code can be detyped because library functions impose types on their arguments. These type constraints cannot be altered because libraries may already be compiled (and their source may not be available). For our transformation to produce typecheckable code, we must keep some type annotations at the boundary between user code and library code.

As in DuctileJ's transformation, our detyping transformation should have little or no effect on the runtime semantics of type-correct programs, and give an alternate runtime semantics to type-incorrect programs. This "alternate" semantics is more permissive: for example, if a class does not implement all abstract methods of a superclass or trait, it should only fail *if that method is actually called*. Similarly, a missing field should only cause an error when actually accessed, and ambiguous references should only cause an error when they are referenced.

3.1 Variable declarations

The transformation replaces type annotations on variable declarations with `Any`, the most-general type in Scala[2]. In Figure 1, we show the effect of this transformation on simple variable declarations. Note that `var` declarations are mutable references, while `val` declarations denote immutable references.

```
// Original code
val a: String = "foo"
var b: Int = someCall(...)

// Transformed code
val a: Any = "foo"
var b: Any = someCall(...)
```

Figure 1: Transformed variable declarations.

3.2 Method invocations

All methods in Scala are conceptually just fields with first-class function values. Method declarations in Scala are of the form `apply(arg1:t1, arg2:t2, ...):t3`. Function objects inherit these `apply` methods from fixed-arity `Function`*N* traits. For example, a 2-arity function with the `apply` method would mix in the `Function[t1, t2, t3]` trait.[3]

[2] `Any` is the supertype of value types (`AnyVal`) and reference types (`AnyRef`). `AnyRef` is analogous to Java's `Object` class type.

[3] Note that type arguments are passed via square brackets and value arguments are passed by parentheses.

We force dynamic dispatch of methods by transforming all function applications into calls to DuctileScala's runtime library, as seen in Figure 2. Arguments to this library function are roughly the original arguments, their declared types (if resolvable), the invocation receiver, the function name, and the declared return type. A similar transformation is performed for constructor invocations, but they must be handled differently due to slightly different dynamic dispatch rules. Construction of arrays of primitive values also warrants a special case due to limitations of the JVM.

```
// Original code
var a: String = "foo"
scala.Predef.println(a) //returns Unit

// Transformed code
var a: Any = "foo"
RT.invoke(scala.Predef,     //call receiver
     typeOf(scala.Predef), //receiver type
     "println",            //method name
     List(a),              //actual arguments
     List(typeOf(a)),      //resolved arg. types
     Unit)                 //declared return type[4]
```

Figure 2: Transforming a function application.

Note that `RT.invoke` and other library calls have return type `Any`.

3.3 Method declarations

Our goal is to allow methods to take any arguments, as in dynamically-typed languages. To achieve this, we must adjust function types to be as general as possible. A simple solution would be to replace every formal parameter type with `Any` (first example in Figure 3). This is inadequate in the face of method overloading: two methods with the same name and arity would be translated to the same detyped function type signature, and the compiler will terminate when it encounters two method declarations with the same function type and name. A workaround for this was used by DuctileJ (second example in Figure 3): in essence, dummy formal parameters are introduced for each original type, and the correct version of the overloaded method is picked at runtime based on actual parameter types. The dummy parameters exist solely to differentiate function signatures, so arbitrary arguments are passed at runtime and are unused.

Unfortunately, even DuctileJ's transformation is insufficient in a language with *first-class functions*. External library code that has not been detyped may have constraints on the types of functions passed as arguments. For example, suppose there is a library method `foo` that takes functions of type `A => B` as an argument. Even if a programmer calls `foo` correctly in her code, DuctileScala's detyping transformation may change the type of the argument to be `Any => Any`, resulting in a compile-time error at `foo` if passed to a library function, or a runtime error if called by external code.

To handle this case, we duplicate the original method, leaving one copy un-mangled and one copy mangled (with dummy

```
//original code
def gcd(a: Int, b: Int) : Int = { ... }

//naive translation
def gcd(a: Any, b: Any) : Int = { ... }

//DuctileJ translation
def gcd(a: Any, a$dummy: Int,
        b: Any, b$dummy: Int) : Any = { ... }

//DuctileScala translation
def gcd$(a: Any, a$dummy: Int,
          b: Any, b$dummy: Int) : Any = { ... }
def gcd(a: Int, b: Int) : Int { ... }
```

Figure 3: Varying approaches for transforming function declarations in Scala.

```
// library code
def foo(fun: A => B): Unit = { ... }

// user code
def bar(a:A): B = { ... }
foo(bar)   //correct in original code
```

Figure 4: DuctileScala will change the type of `bar` from `Function1[A,B]` to `Function2[Any,A,Any]`.

parameters). A similar procedure is also applied to constructors. The benefit is that we can call into external libraries without detyping them. However, this effectively doubles the code size and number of functions. One way to reduce code explosion is to turn the original function into a wrapper that calls the detyped version of the function. Building off of the example in Figure 3, we could replace the last line with:

```
def gcd(a: Int, b: Int) = {
  gcd$(a, b)
}
```

3.4 Generics, type variables, and bounds

In order to be compatible with Java and the JVM platform, Scala performs *type erasure* of generics and type variables, and removes type parameter constraints (bounds, type refinements) after typechecking. All type parameters will be converted to the type `Any` in the `erasure` phase of the compiler, so we don't have to explicitly handle them. Similarly, type variables behave like type aliases and are replaced with their definition at compile-time by the existing compiler infrastructure. Type bounds are completely ignored in our detyping transformation, since they have no effect on the language's runtime semantics.

3.5 Other features

Inferred types The Scala compiler infers the type of all expressions which do not have an explicit type annotation. Since we transform the type of all fields to `Any` and all invocations through the runtime library return `Any`, all inferred types will be be `Any` as well.

First-class functions and Closures First-class functions and closures are converted to anonymous classes with an `apply` method. This is a syntactic transformation done by the parser, so nothing special has to be done.

Views The availability of implicit type conversions is controlled by a scoping mechanism. Selection of a most-specific implicit conversion among several alternatives is governed by normal rules of overloading and overriding for methods. Static implicit type conversions are applied by the compiler during typechecking. To defer the application of these conversions until runtime, all scopes and the implicits available within them must be consulted at runtime.

Multiple argument lists Method signatures with multiple argument lists are converted by the Scala compiler to methods with only one argument list by concatenating all the argument lists in the type erasure phase. The compiler ensures that no two functions have the same signature after type erasure. A valid transformation compliant with our detyping is then to concatenate the different argument lists when creating the `RT.invoke` calls.

Pattern Matching Pattern Matching matches objects based on their runtime types and their constructor parameters. In our transformation, the case selectors containing the pattern to match are not detyped, though the contents of the if statement guarding the case selector and the code inside each case are detyped.

4. IMPLEMENTATION

DuctileScala is comprised of two interacting components: a compile-time syntactic transformation that simulates the "detyping" of a Scala program to an equivalent dynamically-typed program, and a runtime support library that reimplements many of the decisions usually made at compile-time by the Scala compiler. We explain the architecture of each major component, and how the compile-time and runtime parts interact.

4.1 Compiler phases

Scala exposes a flexible, extensible plugin and compiler phase interface [20] that allows plugins to add several new compiler phases, replace old phases, or skip entire phases. We implement the detyping transformation as a plugin for the Scala 2.8.1 compiler. Figure 5 shows the resulting compiler phases.

First, we summarize several original Scala compiler phases that are important to our work:

- `parser` **phase**: The compiler's frontend phase. It parses the source code text and produces an abstract syntax tree (AST).
- `namer` **phase**: This phase creates the symbols corresponding to program entities, and assigns their types, if explicitly annotated.
- `packageobjects` **phase**: This phase loads the packages imported in the code.
- `typer` **phase**: Infers the types of all symbols that remain untyped after the `namer` phase, and does some simple typechecking.

```
parser
signature *     (Signature mangling)
earlynamer*     (Early namer)
earlypackageobjects*  (Early package objects)
earlytyper*     (Early typer)
detyper*        (Detyping)
namer           (Create symbols)
packageobjects (Loads packages)
typer           (Infer types)
...
pickler         (Pickle symbols and types)
peekpickle*     (Peek at pickle)
refchecks
...
terminal
```

Figure 5: Phases of the compiler when the DuctileScala plugin is active. Asterisks mark phases belonging to DuctileScala.

- `pickler` **phase**: The pickler serializes type and symbol information that is needed to support separate compilation. For every top-level class[5], it pickles the necessary information into a binary format, which is then stored in the classfile as a runtime annotation.
- `refchecks` **phase**: Does more advanced typechecking than `typer` and simplifies the intermediate code.

The new phases are described below, in order of execution.

Signature Mangling (Phase `signature`*)*

This phase performs several syntactic transformations on the source program.

1. First, it duplicates and mangles the signatures of methods and constructors, as described in Section 3.3.
2. It inserts import statements to include our runtime library.
3. Certain basic types are added into the definition, which forces these types to get pickled into the classfile. Since only the type and symbol information for fields, methods, objects, and classes are needed for separate compilation, this is all that is pickled by the compiler. An alternative is to recreate the `pickle` phase, deciding what to write into the pickle, but this approach has not been explored yet.

Early Namer, Early Package Objects and Early Typer (`earlynamer`*,* `earlyobjects`*,* `earlytyper`*)*

The detyper needs type information to resolve methods and pass in declared types when rewriting a method invocation. Type information comes from the `namer`, `packageobjects` and `typer`, but if incorrect types are specified, those phases will terminate compilation with a type error. The `earlynamer`, `earlypackageobjects` and `earlytyper` phases mimic the original versions of those phases, with one important difference: they silently record errors, instead of halting the entire compilation process. If the type of an expression cannot be inferred/resolved, then it is set to `Any` and resolution is deferred until runtime.

The original `namer`, `packageobjects` and `typer` phases run after `detyper`, but they do not affect type annotations attached to symbols. In effect, our early phases fully annotate the AST with types, and the built-in phases need not annotate anything.

Detyper (Phase `detyper`*)*

The `detyper` phase traverses the AST twice. The first time, it replaces method, select and constructor invocations with a call to the corresponding `RT.invoke` and `RT.construct` methods in the runtime library, using the types produced by the `earlynamer` and `earlytyper` phases: the receiver object type, argument types, and method return type. The second traversal replaces all type annotations with `Any`, except for the arguments of `main` (which cannot be detyped) and method and constructor signatures, which are already mangled, and case selector in `match` statements, used for pattern matching.

PeekPickle (Phase `peekpickle`*)*

Since types and symbols are possibly cyclic, we pass types to the runtime as indices into a pickle instead of specifying a class object. The pickler runs after the detyper. `Peekpickle` retrieves the type references needed in the transformed code, and converts these type references to something the runtime can understand. Using the pickled symbol and type information, for each type we insert the byte offset of that type in its respective pickle and a string representing the classname where the pickle is. Figure 6 shows the final form for transformed method calls.

To find a type's offset, we first try to look in the pickle of the current lexically enclosing top-level class. If that pickle doesn't contain the type, we look in the pickle of the top-level class of the type's owner symbol. If it is still not found, two possibilities arise. For types with instantiated type arguments, the erased types can be used to reference that type, since the runtime cannot distinguish between the different instantiations of the types[6]. For types of that are defined in Java classes, the fully qualified name of the erased type can be used to uniquely identify them, so a special dummy index and the fully qualified name are passed to the runtime, to be specially processed using Java's reflection library. If none of these apply, we can default and reference to Scala's `Any` type.

For example, to reference the `String` type in Figure 6, we first search the pickle of the `Test` class. If that search is unsuccessful, the algorithm retrieves the symbol that owns `String` and searches there.

4.2 Runtime support

The runtime library provides three methods which implement method dispatch at runtime: `RT.invoke` for calling

[5]In Scala, a top-level class is not lexically enclosed by any other class or module. It is equivalent to the granularity of a single Java source file. Scala allows several classes to be defined in the same source file.

[6]This doesn't hold for the construction of `Array`s of primitive types as explained in Section 3.3.

```
// Original code
object Test {
    var a: String = "foo"
    scala.Predef.println(a)
}

// Transformed code
object Test {
    var a: Any = "foo"
    RT.invoke(scala.Predef,  //call receiver
       32,                   //re-
ceiver type offset
       "scala.Predef",       //re-
ceiver type classname
       "println",            //method name
       List[Any](a),         //actual arguments
       List[Int]47),         //argu-
ment type offset
       List[String]("Test"), //argu-
ment type classname
       47,                   //return type offset
       "Test")               //re-
turn type classname
}
```

Figure 6: Final method call transformation, type `Any` has offset 47 in the classfile `Test` and type `scala.Predef` has the offset 32 in the classfile `scala.Predef`.

methods, `RT.construct` for calling constructors, and `RT.constructArray` for constructing arrays. Unlike DuctileJ's runtime library, we do not have a function for accessing fields: the Scala compiler desugars all field accesses into getter and setter methods before typechecking, so they are unneccessary. The runtime `invoke` and `construct` methods take as arguments the call receiver, the method or constructor name, the original types of arguments, the original arguments themselves, and the return type. The library code then uses the compiler's type inference engine to determine what (if any) method should be called, given the argument types, method name, and return type. If it finds a matching method, then it converts the Scala types to the corresponding JVM classes. With these JVM-friendly classes, we can load and invoke the appropriate method or constructor using normal Java reflection.

Implementation details:

- **Passing types** Because Scala types cannot be represented fully as JVM classes due to potential circular dependencies, and Scala lacks its own reflection library, there is no simple way to pass types as arguments to the runtime library's methods. Our approach, as described in 4.1, is to reuse the top-level class pickles along with a copied version of the compiler's unpickler routine to recreate the compiler's representation of the full Scala types at runtime. This representation is also used by the compiler itself during compilation, so reuse of the type inference algorithm, type, and symbol methods is simple.

- **Primitive operations** Scala does not compile all Scala method calls to JVM-bytecode method calls. Although Scala presents a uniform object abstraction across both reference and value types , it compiles arithmetic operations, string concatenation, and array construction in a more efficient manner. The Scala typechecker treats these operations as methods, but the bytecode generator converts them into JVM primitive bytecodes.

  The compiler plugin does not need to handle these cases specially, but the runtime library must be aware of and handle them. In some instances it may need to execute a primitive operation instead of a method call. This requires a large case block of repetitive code to call the right operation for each special-cased type. To create both regular and primitive arrays, the DuctileScala runtime passes in the types to the Java runtime, which handles array creation.

- **Static and dynamic type mismatch** In the ideal case, the runtime type of an object is always a subtype of the statically declared type. However, if there is an error that halts execution (for example, no method of the specified name is found), the runtime should try to dispatch on the actual dynamic types of the objects before giving up and throwing an exception. If no exact match is found, our alternate semantics tries several heuristics to find a similar method to call. For example, it assumes that the static type of the receiver is more likely to be correct than the static type of the arguments, so methods with differing argument types are examined first. In order to get the dynamic Scala type, the Java `Object.getClass()` method is called on the object and the compiler code looks up the resulting name to get the equivalent Scala type, with special cases for null, arrays, and primitives.

  In type-incorrect code, it is possible that the method that was intended to be invoked is found but the dynamic types of the arguments are invalid for the method. If the method is not detyped, as with external library code, DuctileScala throws an exception. If the method is user code, then its signature may simply be wrong, so we should invoke the method anyway using the mangled version with detyped arguments as described in Section 3.3. The runtime also recognizes that library classes do not have mangled methods and falls back to calling unmangled methods if mangled equivalents are not found.

- **Implicit parameters** If a caller does not pass in a argument for a parameter marked `implicit`, then Scala searches the environment for a type-correct match (which also must be marked `implicit`). (See Section 2.5.) At its current state, the runtime can only find implicit objects defined in `scala.Predef`, which is imported by default in all Scala programs (and thus always in scope for implicits selection).

  Scala's typechecker automatically resolves missing implicit parameters. To use it, the runtime constructs a syntax tree for the method call and passes it to the typechecker, which modifies the tree to fill in the missing arguments. The runtime converts the modified AST to Scala runtime values, and takes the relevant

objects as values to pass to the method. Note that currently the typechecker only considers implicit objects defined in `scala.Predef`.

- **Views (implicit conversions)** Implicit conversions (see Section 2.6), can be applied to call receivers or arguments. [7] To find implicit conversions for the receiver, the runtime builds an AST for the method call and run the compiler's typechecker on it. If an implicit conversion is supposed to be applied to the receiver, the typechecker will modify the AST correspondingly. If the runtime detects a modification it will deconstruct the AST to find the inserted conversion and apply it. Similarly for arguments, when the runtime uses the typechecker to determine the correct method to call, that method may have formal parameter types which do not match the actual argument types given to the typechecker. That means the typechecker needs to be asked for each pair of argument and parameter types what implicit conversion to use to convert that argument to the proper type; that conversion is then applied.

It bears repeating that we are able to reuse much of the actual Scala compiler, since it is self-hosted. This reuse has saved a lot of effort and avoided a lot of possible reimplementation bugs. The runtime hooks into parts of the compiler comprising thousands of lines of code with thousands of lines of code with myriad obscure edge cases and implicit (and fragile) assumptions. If we observe the correct types at runtime, the method dispatch algorithm will mimic what Scala would have done at compile time.

On the other hand, there are some issues in reworking the Scala compiler as a dynamic Scala runtime. In order to get it to act as such, we create stubbed versions of some Scala compiler objects. Creating these compiler objects may trigger slower and useless compilation code, and some code duplication was inevitable for compiler classes with private members that needed to be modified in our implementation. Loading the Scala compiler JAR for every program execution could also be detrimental to performance.

5. EVALUATION

Our primary contribution is to determine whether the core idea of DuctileJ scales (in a technical sense) to more complex type systems. Another possible contribution is to obtain a first approximation to the question of whether such always-available dynamic feedback is more, less, or equally useful in a language with a more sophisticated type system. To validate our first contribution, we need to qualify the kinds of type-correct and type-incorrect programs that our transformation and runtime can support. The second contribution is hard to quantify, but supporting evidence could be gathered in a similar fashion as in the user study of DuctileJ. Instead, we provide use cases to motivate our choice of permissive semantics for type-incorrect code, and defer 3rd-party testing of these semantics to future work. To evaluate our first contribution, we assembled a suite of 62 programs that test various aspects of Scala on these.

[7]Scala allows at most one implicit type conversion to be applied to any value.

At the lowest level, our transformation must preserve the runtime semantics of type-correct input programs. We plan to use an informal approach to verifying equivalent (up to observable output) runtime semantics by running transformed and original versions of type-correct programs from our test suite and comparing their output.

The real power of the Ductile approach to combining static and dynamic typing is that type-incorrect programs can still in some cases provide meaningful feedback through execution. Thus, we can measure the outcome of our approach by the set of classes of type-incorrect programs that we can execute using more permissive semantics. To demonstrate these permissive semantics, we will create several test cases for each class of type error that we can work around.

6. RESULTS

Currently, we are able to run a limited subset of type-correct Scala programs in addition to variants with certain kinds of type errors.

The features we support include:

- Method invocation, including recursive methods with method dispatch for class hierarchies, as in Figure 7.
- Object construction with parameters.
- Closures, including for loops and other Scala control structures which use them, as shown in Figure 8.
- All primitive operations ($+, -, *, <=$), including string concatenation.
- Static implicit conversions (views), for conversions that can be determined at compile-time.
- Dynamic implicit conversion (views) for receivers for views in Predef, as as shown in Figure 9.
- Static implicit parameters.
- Dynamic implicit parameters for implicits in Predef.
- Multiple argument lists.
- Pattern matching, including correct handling of case classes, which can be constructed without a `new` statement.
- Import of Scala packages.
- Correct handling of Java class.

The detyping transformation also allows us to run code with incorrect static types but correct dynamic types, interacting with a diversity of Scala features as shown in Figures 7 and 9.

7. RELATED WORK

There are many other approaches besides our own that try to combine the strengths of static and dynamic typing. We can categorize these approaches by the strengths they aim to import from dynamically-typed languages into statically-typed languages (or vice versa).

Figure 7: Allowing incorrect static types, the invocation of `speak` is done based on runtime types, even if the static types are incorrectly declared.

```
object Test {
  class Cat {
    def speak() = println("I'm a cat!")
  }
  class Cow {
    def speak() = println("I'm a cow!")
  }
  def main(args: Array[String]) = {
    val b:Cat = new Cow
    b.speak()   // prints "I'm a cow!"
  }
}
```

Figure 8: Closures and function objects are both handled correctly: the argument `{a()}` is a simple closure that calls method `a` and the argument `b` is passed as `Function0[Unit]` object.

```
object Test {
  def a() : Boolean = {return true}
  def b() = {println("In body!")}
  def main(args: Array[String]) = {
    def ifLoop(cond: => Boolean, body: => Unit) = {
      if (cond) body
    }
    ifLoop ({a()}, b) //prints "In body!"
  }
}
```

Figure 9: Implicit runtime conversion of `String` into `StringOps`, note that this example doesn't type check because the type `Any` has no `capitalize` method and also the runtime type `String` has no `capitalize` method defined.

```
object Test {
  def main(args: Array[String]) {
    println(someString.capitalize) // prints "Testing."
  }
  def someString : Any = "testing."
}
```

7.1 Improving statically-typed languages

Our approach attempts to integrate the quick development cycle of dynamic languages into a statically-typed language by deferring typechecking until the last possible moment (namely, runtime). This technique was pioneered in Java by Bayne, et al. with DuctileJ [6]. Though Java and Scala are similar and we employ a similar "detyping" transformation as DuctileJ, our implementation strategy differs significantly. DuctileJ reimplemented many of the type resolution routines at compile-time, whereas we could reuse most of the existing compiler code. Similarly, their runtime system had to reimplement the dispatch algorithm in its entirety, while we were able to reuse the machinery of the Scala compiler. On the other hand, DuctileScala requires much more information at runtime to match the language's dispatch semantics, so our approach has a larger space and time overhead.

Another way to speed up the development cycle is through use of an interactive read-eval-print loop (REPL). While REPLs are historically associated with dynamic languages such as Lisp, Scheme, and Python, several modern statically-typed languages include a REPL-like program in their standard distribution. REPL-ready languages include Haskell [24], Scala [23], and Ocaml [16]. REPL programs accelerate the process of obtaining feedback from the compiler (relative to whole-program batch compilation), but they are still fundamentally more restrictive than our approach in that dynamic feedback always follows static feedback.

Sometimes the explicit goal is not development speed, but program flexibility. With dynamic typing, variables may reference several different types freely without any constraints. In contrast, consider a variable with type `Any`: this variable could be a reference to objects with any real type, but the typechecker ensures that `Any` actually declares all methods that the variable is a receiver of. The `Dynamic` type [7] recently introduced to C# 4.0 allows the programmer to sidestep typechecking of any variable with type `Dynamic`. Compared to DuctileScala, their approach limits the scope of dynamism to just those variables with type `Dynamic`. This has advantages and disadvantages: only certain annotated parts of the program become more dynamic, and the rest of the program remains statically typed (through automatically-inserted runtime checks and casts). This can also be viewed negatively: the programmer must painstakingly reason about and manage the boundary between dynamic and static types in the program. Our approach also provides dynamism and flexibility, but transparently without any extra annotation burden.

Some dynamically-typed programming languages are designed as sister languages to specific statically-typed languages, with the intent of providing a more lightweight and flexible syntax but allowing equivalent program structure and usage of familiar libraries. The Groovy scripting language [18] is one such "dynamic doppelganger" to the static Java language, and BeanShell [21] is an even more faithful dynamic version of Java. Using two distinct but related languages has obvious drawbacks: the application must be implemented in both languages, an unacceptable increase in required developer effort. Conceptually, DuctileScala's detyping transformation and runtime library support the execution of a "dynamic version" of the original program. However, this doppelganger has identical semantics as the static language by construction, and is transparent to the tool user.

The two previous approaches, along with our approach, make static type systems more flexible by way of diluting their power; an alternative approach to increasing a type system's flexibility is to *increase* the type system's power. For example, Scala's statically-typed implicit type conversions can emulate the primitive type coercions of dynamic languages without syntactic overhead. In some sense, adding support for generic programming [15, 22, 25] is one way to approach the raw flexibility of dynamic types by making statically-

typed programs more general.

7.2 Improving dynamically-typed languages

Though DuctileScala focuses on importing the benefits of dynamism into a statically-typed language, most work at the intersection of static and dynamic typing has taken the path of enriching dynamically-typed languages with the benefits of static typing (most importantly, static feedback). All such approaches essentially create a new static type system for the dynamic language. Use of the type system is accomplished by transparently inferring types of dynamic programs during compilation, by adding language/parser support for explicit type annotations, or other techniques.

Automatically inferring types in dynamic languages imposes the least burden on programmers (comparable to DuctileJ or DuctileScala). Unfortunately, this convenience comes at a cost: such inference-friendly type systems are typically weak or unsound (unable to provide safety guarantees) or brittle (small program changes may cause large changes to inferred types). Static type inference of dynamic languages was first investigated as "soft typing" in Scheme [10, 30]; more recent work has targeted Ruby [14], Python [3, 4, 9, 26], JavaScript [2, 17], and other languages. Some researchers have abandoned purely static inference in favor of approaches that combine static inference with runtime feedback [13] with moderate success. Others have investigated tractable subsets of existing dynamically-typed languages [1, 28] or entirely new languages [8] that are more suitable for static type analysis and inference; these approaches are much more successful but require refactoring or rewriting of programs.

Finally, adding explicit type annotations to dynamically-typed languages has been proposed many times as a means to aid program verification and optimization. A goal common to our work and several works based on annotating is to support typed and untyped code within the same program. We support this at the granularity of entire executions: an execution can either be typed normally, or detyped[8] in its entirety. At this resolution, there is nothing to annotate. Others have investigated more fine-grained divisions at the granularity of modules [29] or individual types [27, 31]. These approaches allow for better control and program evolution, but also require extra reasoning by the programmer. There are many other type systems for dynamically-typed languages (e.g. for verifying security properties), but they are ou[tside the scope of this discussion.

8. FUTURE WORK

We have implemented a substantial subset of Scala, but there are still several language features that must be addressed to run real-world code. Most missing features are omitted for a lack of time, but a few features lack a simple, straightforward translation. Below, we describe some of the more challenging implementation obstacles that remain. A fuller list of partial and unsupported features exists in the feature matrix of Table 1.

[8]Remember that even after DuctileScala's detyping transformation, external library code is still "typed" though the runtime semantics of typed and detyped code are intended to be equivalent.

General support for implicits and views

We are unsure how to tractably implement selection of implicit type conversions and implicit parameters at runtime. A brute-force approach to selecting implicits would be to pass names and types of all available (i.e., in scope) implicit type conversion methods to every program expression where such conversions could be applied. This approach has obvious code explosion issues.

A more conventional approach would be to serialize the scoping "topology" of the running program, and have the runtime library simulate the program's current scope with calls like `RT.enterScope` and `RT.exitScope` at entry and exit from scopes. When an implicit argument or implicit type conversion is necessary, the runtime will know (based on the simulated scope) the context from which these implicits can be selected. While less brute-force, this scope simulation approach is greatly complicated by exceptional control flow and its effects on the current scope.

Call-by-name function arguments

Scala supports *thunks*(call-by-name arguments), values that are not evaluated prior to function application. The type of an `A` passed by name is written as `(=>A)` The Scala compiler desugars these into anonymous classes with a nullary `apply` method. Since these closures are anonymous and their signatures are not pickled, there is no way to know their return type. This means that dispatching methods based on call-by-name values is unsupported by DuctileScala.

We also leave to future work the task of evaluating DuctileScala in realistic programming situations. A user study similar to that conducted for DuctileJ would permit a useful comparison between the two in terms of impact on software development.

9. CONCLUSION

From both a theory and implementation standpoint, detyping Scala code is much more involved and tricky than detyping Java code. We present DuctileScala, a detyping approach based on compile-time transformations and runtime support. Our conclusion is that such an approach is too buggy and troublesome to be feasible for complex Scala programs. Emulating the semantics of typed Scala code at runtime requires a lot of reified compiler context (compared to Java). Passing this context to the runtime libary is error-prone in the face of compiler-controlled serialization of types and symbols.

We believe that a dynamic Scala interpreter (a la BeanShell [21]) endowed with a more relaxed typechecker has a better chance of running real-world Scala programs. With full access to the program AST at runtime, there is virtually no engineering necessary to pass compiler context to runtime: they are one in the same. Regardless, our work is a useful data point for other researchers who may be curious as to the limits of the compiler transformation/runtime library approach for detyping.

10. REFERENCES

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards

reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 symposium on Dynamic languages (DLS)*, pages 53–64, New York, NY, USA, 2007. ACM.

[2] Christopher Anderson. *Type inference for JavaScript*. PhD thesis, Department of Computing, Imperial College London, March 2006.

[3] Joe Angell. Gradual Python with colored local type inference. http://www.cs.colorado.edu/ bec/courses/csci5535-s09/slides/angell-presentation.pdf, 2009.

[4] John Aycock. Aggressive type inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2005.

[5] Michael Bayne, Richard Cook, and Michael D. Ernst. DuctileJ project page. `http://code.google.com/p/ductilej/`.

[6] Michael Bayne, Richard Cook, and Michael D. Ernst. Always-available static and dynamic feedback. In *33rd International Conference on Software Engineering (ICSE)*, 2011. To appear.

[7] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *Proceedings of the 24th European conference on Object-Oriented Programming (ECOOP)*, 2010.

[8] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 117–136, New York, NY, USA, 2009. ACM.

[9] Brett Cannon. Localized type inference of atomic types in Python. Master's thesis, California Polytechnic State University, 2005.

[10] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation (PLDI)*, pages 278–292, New York, NY, USA, 1991. ACM.

[11] EPFL. The Scala programming language. `http://www.scala-lang.org/`.

[12] EPFL. A tour of Scala. `http://www.scala-lang.org/node/104`.

[13] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 283–300, New York, NY, USA, 2009. ACM.

[14] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC)*, pages 1859–1866, New York, NY, USA, 2009. ACM.

[15] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17, March 2007.

[16] INRIA. The toplevel system (`ocaml` command). `http://caml.inria.fr/pub/docs/manual-ocaml/manual023.html`.

[17] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[19] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 425–434, New York, NY, USA, 2007. ACM.

[20] Anders Bach Nielsen. Scala compiler phase and plug-in initialization for Scala 2.8. `http://www.scala-lang.org/sid/2`.

[21] Pat Niemeyer. BeanShell: Lightweight scripting for Java. `http://www.beanshell.org`.

[22] Bruno C.d.S. Oliveira and Jeremy Gibbons. Scala for generic programmers. In *Proceedings of the ACM SIGPLAN workshop on Generic programming (WGP)*, pages 25–36, New York, NY, USA, 2008. ACM.

[23] Paul Phillips. The interactive interpreter (REPL). `http://www.scala-lang.org/node/2097`, 2009.

[24] Bernie Pope. Step inside the GHCi debugger. *Monad Reader*, 1(10), March 2008.

[25] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 111–122, New York, NY, USA, 2008. ACM.

[26] Michael Salib. Starkiller: a static type inferencer and compiler for Python. Master's thesis, Massachusetts Institute of Technology, May 2004.

[27] Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European conference on ECOOP 2007: Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2007. Springer-Verlag.

[28] The Caja Team. Caja: a source-to-source translator for securing JavaScript-based web content. `http://code.google.com/p/google-caja/`, 2010.

[29] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 964–974, New York, NY, USA, 2006. ACM.

[30] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Trans. Program. Lang. Syst.*, 19:87–152, January 1997.

[31] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 377–388, New York, NY, USA, 2010. ACM.

Table 1: A Feature Matrix for DuctileScala

Language Feature	Support Level	Related test cases	Details
Variable declarations	Full	various	
Method declarations	Full	various	
Function application	Partial	`no_args_call`, `one_arg_call`, various	
Object construction	Partial	`construct_type_param`, `construct_type_param2`, `constructors` various	We do not support constructors with type arguments (as described in Section 8).
Array construction	Full	`array_construct`, `array_primitive_construct`, `array_primitive_class`	
Multiple argument lists	Full	`multiple_argument_list`	
Implicit type conversions	Partial	`arg_view`, `arg_view_int`, `arg_view_int_typed`, `captialize`, `captialize_detyped`, `captialize_runtime`	Only with type conversions defined in `scala.Predef` or in correctly typed code (found at compile time).
Partial function application	Partial	`partial_application`	It is syntactic sugar that creates a function object that calls the original function with all parameters.
Abstract type members	Full	`abstract_types`	
Module objects	Full	various	
Nested classes/objects	Partial	`nested_objects`, `nested_classes`	Only nested objects supported.
Polymorphic methods	Partial	`poly_dispatch`, `poly_dispatch_detyped`	Polymorphic dispatch not supported with detyped objects.
Closures	Full	`closure`, `for_loop`	
Primitive binary operators	Full	`int_add`, `byte_shift`	
Pattern Matching	Full	`case_case_class_no_arg`, `case_case_class_one_arg`, `case_int`, `case_trivial`, `unapply_case`, `pattern_polar`	Generated code signatures are not mangled, but the code inside is.
Implicit arguments	Partial	`implicit_arg`, `implicit_arg_detyped`, `implicit_arg_detyped_only`, `implicit_arg_typed`, `implicit_argument`	Only implicit arguments defined in `scala.Predef` are used.
Java types/Module imports	Full	`java_types`, `pattern_polar`	
Type Variances	No explicit support		It is a check done only at compile time.
Type bounds	No explicit support		It is a check done only at compile time and it can't be used dispatch at runtime since only erased types are available.
Traits	No explicit support		It is a check done only at compile time.
Self references	No explicit support		It is a check done only at compile time.
Pass-by-name values	None		Not implemented, as a new interface to the runtime library would be needed.
Varargs	None		Very important since `List`'s and other collections' constructors depend on varargs. Requires a change to the runtime to make it properly recognize the difference between varargs and regular `Seq` arguments.